

Ultra Ethernet™ Specification v1.1

October 12, 2026

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Copyright © 2026 Ultra Ethernet Consortium.

The Ultra Ethernet Consortium (UEC) is a consensus-based standards organization operating under the Joint Development Foundation. Its members collaborate openly to define and promote high-performance Ethernet technologies for modern computing environments. This specification ("Specification") contains Approved Deliverables of UEC as such term is defined in the UEC Charter.

This document is made available under the terms of the Creative Commons Attribution-NoDerivatives 4.0 International License (CC BY-ND 4.0). As stated in the license, you may copy and redistribute the document, provided that attribution is given to the Ultra Ethernet Consortium. However, if you create a derivative work from the material, you may not distribute that work. A copy of the license is available at https://creativecommons.org/licenses/by-nd/4.0/.

This Specification is provided on an "AS IS" basis. UEC and The Joint Development Foundation (the "Parties") make no warranties of any kind, either express or implied, including without limitation warranties of merchantability, fitness for a particular purpose or noninfringement of any third party intellectual property rights, or compliance with applicable laws. Use of the information in this Specification is at your own risk. The entire risk as to the results and performance of this Specification is assumed by the user. In no event will the Parties (as defined above) be liable to any other party for lost profits or any form of indirect, special, incidental, or consequential damages of any character from any causes of action of any kind with respect to this Specification or its governing document, whether based on breach of contract, tort (including negligence), or otherwise, and whether or not the recipient of this Specification has been advised of the possibility of such damage.

This Specification may include references to technologies for which patent rights might be claimed. UEC does not make any representation regarding the existence or status of any such rights. Implementers of this Specification are solely responsible for obtaining any necessary licenses or permissions from the appropriate rights holders.

UEC reserves the right to adopt any changes or alterations to this Specification as it deems necessary or appropriate.

Ultra Ethernet, Ultra Ethernet Consortium, and the Ultra Ethernet Consortium logo are the unregistered trademarks of Ultra Ethernet Consortium in the United States and other countries. All rights reserved. Other brands, names or logos contained herein are the property of their respective owners.

For more information about the Ultra Ethernet Consortium, visit: https://ultraethernet.org.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Contents

1 Introduction 28

1.1 Background 28

1.1.1 UEC Organization 28

1.1.2 UE Transport Profiles 29

1.2 UE Specification Conventions 30

1.2.1 Normative, Informative, and Implementation Statements 30

1.2.2 Terminology 30

1.2.3 Formatting 47

1.2.4 References 49

1.3 System View and Nomenclature 49

1.3.1 Workloads [Informative] 57

1.4 Software 60

1.4.1 AI and HPC API Interface 60

1.4.2 Fabric Endpoint Software Stack 60

1.4.3 Switch Software Stack 61

1.4.4 Network Operating System (NOS) Interface 61

1.5 Networking 62

1.5.1 AI and HPC Network Taxonomy 62

1.5.2 UE Transport (UET) Objectives 65

1.5.3 Network Fabric 66

1.6 UE Specification Overview: Layers 71

1.6.1 Software Layer 71

1.6.2 Transport Layer 72

1.6.3 Network Layer 73

1.6.4 Link Layer 74

1.6.5 Physical Layer 75

2 UE Software Layer 77

2.1 UE Software Overview 77

2.1.1 Software Specifications 77

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

2.1.2 Software Components and Interfaces 78
2.1.3 Reference Software Models and Supplementary Software 79
2.1.4 References 80
2.2 UE Libfabric Mapping 81
2.2.1 Application Use Cases 84
2.2.2 UET Profiles 84
2.2.3 Configuration Information 88
2.2.4 JobIDs 89
2.2.5 Libfabric APIs 94
2.2.6 Packet Delivery Modes 121
2.2.7 Traffic Classes 122
2.2.8 Transmit and Receive Queues 123
2.2.9 Security Protocol 125
2.2.10 Wire Protocol Mapping 129
2.2.11 Linux Implementation of UET Control API 133
2.2.12 References 134
2.3 UE Verbs Mapping 135
2.3.1 Verbs Terminology 135
2.3.2 Verbs Overview 136
2.3.3 UET Verbs Object Model 139
2.3.4 Device Discovery 145
2.3.5 UET Services 145
2.3.6 JobIDs 150
2.3.7 Job Keys 151
2.3.8 Work Requests 151
2.3.9 Immediate Data Support 151
2.3.10 Work Completions 151
2.3.11 Memory Registration 152
2.3.12 QP Semantics 153
2.3.13 References 153
2.4 In-Network Collectives (INC) 154
2.4.1 Assumptions 154

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

2.4.2 Theory of Operation 155
2.4.3 INC Operations 156
2.4.4 Datatypes 157
2.4.5 Reduction Operators 157
2.4.6 Operator Ordering Modes 158
2.4.7 Operator Rounding Modes 158
2.4.8 INC Libfabric 159
2.4.9 Libfabric Provider Extensions 161
2.4.10 INC Switch Required and Optional Capabilities 161
2.4.11 INC Transport 162
2.4.12 INC Data Plane Overview 164
2.4.13 Error Handling 166
2.4.14 INC Manager 166
2.4.15 UET Libfabric Provider and IM 166
2.4.16 IM Management 171
2.4.17 IM and INC Switches 172
2.4.18 References 177
3 UE Transport Layer 178
3.1 UET Scope, Scale, and Reach 178
3.1.1 Virtualization 179
3.2 UET Layers, Components, and Capabilities 179
3.2.1 Semantic Sublayer (SES) 181
3.2.2 Packet Delivery Sublayer (PDS) 182
3.2.3 Congestion Management Sublayer (CMS) 184
3.2.4 Transport Security Sublayer (TSS) 185
3.2.5 Layering Summary 185
3.2.6 Sublayer Interfaces 186
3.2.7 Error Handling 187
3.3 Profiles and Capabilities [normative] 187
3.3.1 SES Transactions 188
3.3.2 Buffer Addressing Mechanisms 189
3.3.3 Authorization 189

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

3.3.4 Buffer Behavior 189
3.3.5 Packet Formats 190
3.3.6 PDS Ordering Modes 191
3.3.7 CMS Congestion Control Algorithms 191
3.3.8 INC Support 192
3.3.9 Encapsulation 192
3.4 Semantics Sublayer (SES) 193
3.4.1 Semantic Use Cases 193
3.4.2 Definition of Semantic Concepts 194
3.4.3 Semantic Header Formats 218
3.4.4 Semantic Processing 240
3.4.5 Semantic Protocol Sequences 254
3.4.6 Error Handling 271
3.4.7 Enumerated Types Used in Headers 275
3.4.8 Device Expectations 282
3.4.9 UE Transport Semantics: Memory Model 283
3.4.10 Mapping of *CCL Send/Receive to Proposed Semantics [Informative] 287
3.5 Packet Delivery Sublayer (PDS) 292
3.5.1 PDS Terminology 292
3.5.2 Illustration of PDS Terms 295
3.5.3 Packet Delivery Services 297
3.5.4 PDS-SES Logical Interface 298
3.5.5 PDS Configuration Parameters 302
3.5.6 Reliability and Ordering 306
3.5.7 Packet Delivery Modes Overview 307
3.5.8 Packet Delivery Contexts (PDC) 309
3.5.9 PDS Event State Machine 325
3.5.10 Header Formats 332
3.5.11 Header Fields 346
3.5.12 Requests and Acknowledgements 356
3.5.13 Default SES Responses 374
3.5.14 Transmit Scheduling 375

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

3.5.15 Loss Detection and Recovery 376
3.5.16 Control Packet (CP) 386
3.5.17 Semantic Responses 398
3.5.18 Reserved PDCs 398
3.5.19 Sequence Diagrams 400
3.5.20 Reliable Unordered Delivery 401
3.5.21 Reliable Ordered Delivery 415
3.5.22 Reliable Ordered Delivery for Local Networks 420
3.5.23 RUDI 424
3.5.24 Error Model 426
3.5.25 Full Header Format 427
3.5.26 UET CRC 435
3.5.27 Padding of Ethernet Packets 437
3.6 Congestion Management Sublayer (CMS) 439
3.6.1 UET CC Guidelines [Informational] 439
3.6.2 Congestion Control Algorithms 440
3.6.3 Congestion Control Algorithm Design Targets 440
3.6.4 Telemetry and Network Switch Services 440
3.6.5 UET CC Protocol Operation Overview 451
3.6.6 Congestion Control Context (CCC) 456
3.6.7 CCC for ROD PDCs 456
3.6.8 Source Context 457
3.6.9 UET-CC Header Formats and Fields 461
3.6.10 Common Congestion Control Event Processing 467
3.6.11 Congestion Control Modes 471
3.6.12 Overall CCC Pseudocode 471
3.6.13 Network Signal-based Congestion Control 476
3.6.14 UET Receiver-Credit Congestion Control 491
3.6.15 Transport Flow Control (TFC) 505
3.6.16 Multipath Path Selection 508
3.6.17 Switch Configuration for UET CC 514
3.6.18 Programmable Congestion Management 516

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

3.6.18.9 Algorithm 546
3.7 Transport Security Sublayer (TSS) 548
3.7.1 Introduction 548
3.7.2 Security Model 548
3.7.3 Architecture 555
3.7.4 Secure Domains 559
3.7.4.1 Joining a Secure Domain 560
3.7.5 Key Lifetime and Security Considerations 561
3.7.6 Secure Domain Key Database (SDKDB) 565
3.7.7 KDF Modes 566
3.7.8 KDF Construction 567
3.7.9 Replay Protection 573
3.7.10 Epoch-based packet rejection 573
3.7.11 TSS over the UFH_32 Header 574
3.7.12 Statistics, Parameters, and Events 584
3.8 References 585
4 UE Network Layer 590
4.1 Packet Trimming 590
4.1.1 Interactions with explicit congestion notification 594
4.1.2 Where can trimming be enabled? 594
4.1.3 Interactions with upper protocols 594
4.1.4 Mapping DSCPs to traffic classes for Ultra Ethernet transport 595
4.1.5 Mapping DSCPs to traffic classes for other transports 598
4.1.6 Security considerations 598
4.1.7 References 599
4.2 Congestion Signaling (CSIG) 600
4.2.1 Terminology 602
4.2.2 Overview 602
4.2.3 Use Cases 604
4.2.4 CSIG Tag Formats 605
4.2.5 Functionality Within the CSIG Domain 612
4.2.6 CSIG Tag Layout 616

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

4.2.7 CSIG Profiles 619
4.2.8 CSIG Traffic Rate Measurement Interval 620
4.2.9 CSIG Signals 621
4.2.10 CSIG Deployment Considerations 627
4.2.11 References 629
5 UE Link Layer 630
5.1 Link Layer Retry (LLR) 631
5.1.1 Frame structure 633
5.1.2 Interface modifications 634
5.1.3 LLR Operation 635
5.1.4 LLR configuration 637
5.1.5 LLR transmit path operation 638
5.1.6 Transmission of LLR_ACKs/LLR_NACKs 643
5.1.7 LLR receive path operation 644
5.1.8 Received ACK/NACK processing 645
5.1.9 Control Ordered Set transmission and reception 646
5.1.10 Error propagation 647
5.1.11 Counters 648
5.2 Credit-based Flow Control 650
5.2.1 Lossless Packet Delivery Use Cases 650
5.2.2 CBFC and PFC Relative Advantages 651
5.2.3 CBFC Feature List 651
5.2.4 CBFC Overview 652
5.2.5 CBFC Operation 653
5.2.6 CBFC Message Formats 662
5.2.7 MAC and MAC Control Layer Interfaces to CBFC 664
5.2.8 CBFC Initialization 664
5.2.9 Interactions Between CBFC, PFC, and LLR 674
5.2.10 Compliance Requirements 678
5.2.11 Control Ordered Sets (CtlOS) in UE Link Layer 680
5.2.12 CBFC Message Examples (Informative) 682
5.2.13 References 683

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

5.3 UE Link Negotiation 685
5.3.1 LLDP Overview 685
5.3.2 UE Organizationally Specific LLDP TLVs 687
5.3.3 UE LLDP YANG 698
5.3.4 References 715
5.4 Optimized Headers and Packet Rate Improvement 716
5.4.1 Requirements and Considerations 716
5.4.2 Header Optimization Approach 717
5.4.3 UFH Definition 718
5.4.4 UFH Variants 720
5.4.5 UFH and Header Stacking 722
5.4.6 UFH Termination and Forwarding 723
5.4.7 UFH and Queue Assignment 724
5.4.8 UFH and Packet Trimming 724
5.4.9 UFH and VLANs 724
5.4.10 UFH and Transport Layer Checksums 725
5.4.11 UFH Deployment Considerations 725
5.4.12 UFH Efficiency Calculations 726
5.4.13 References 727
5.5 Idle Reduction 728
5.5.1 Approach 728
5.5.2 Rate Control 729
5.5.3 Maximum Packet Rate 729
5.5.4 Requirements 729
5.5.5 Enabling Idle Reduction 729
6 UE Physical Layer 730
6.1 UE PHY Support 730
6.1.1 UE PHYs for 100 Gb/s per Lane Signaling 730
6.1.2 UE PHYS for 200 Gb/s per Lane Signaling 731
6.2 Control ordered sets 732
6.2.1 Sequence ordered sets and control ordered set background 732
6.2.2 Control ordered sets format 733

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

6.2.3 PCS required modifications 733
6.2.4 RS required modifications 735
6.3 FEC statistics for prediction of link quality 736
6.3.1 Relationship between performance metrics 738
6.3.2 Estimation of UCR from FEC statistics 741
6.3.3 Examples 747
6.4 Recommendations 749
6.4.1 Low error rate 750
6.4.2 Low power 750
6.4.3 Low latency 750
6.5 References 750
7 UE Management 752
7.1 UE Management Scope 752
7.2 High-Level UE Management Architecture 752
7.3 UE Fabric Management 754
7.4 Redfish for UE Endpoint Manageability 755
7.5 YANG Model for UE Network Manageability 756
7.5.1 Network Switch Planes 756
7.5.2 UE Switch Manageability Architecture 757
7.5.3 YANG Model Extensions for UE 758
7.6 Switch Abstraction Interface (SAI) for UE 759
7.6.1 SAI Enhancements for UE 759
8 UE Compliance 761
8.1 Compliance Statement 761
8.1.1 UE Support Requirements 762
8.1.2 Declaration Format 762
8.1.3 Compliance verses Support Terminology 762
Release Notes 763

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Tables

Table 1-1 - Distinctive characteristics by network type (circa 2024) 64
Table 1-2 - Characteristics of UET Deployment Model 65
Table 2-1 - UE Software Components and Interfaces 78
Table 2-2 - UE Reference Software 79
Table 2-3 - UET Application Categories 84
Table 2-4 - Per-Profile Libfabric Parameter Requirements 85
Table 2-5 - Application Use Case Mapping to UET Profiles 87
Table 2-6 - Profile Logical Priorities 87
Table 2-7 - Libfabric UET Provider Configuration Parameters 88
Table 2-8 - UET Control API JobID Mapping Parameters 90
Table 2-9 - Libfabric API Groups 95
Table 2-10 - UET Libfabric Endpoint Address 97
Table 2-11 - fi_getinfo() Parameters 100
Table 2-12 - Pre-Defined UET Service Names 101
Table 2-13 - Memory Region Key Format 104
Table 2-14 - Criteria for Optimized Non-Matching SES Header for RMA Operations 105
Table 2-15 - Criteria for Small RMA SES Header with RMA Operations 105
Table 2-16 - Criteria for RUDI Packet Delivery Mode with RMA Operations 106
Table 2-17 - Memory Key Scope Requirements 106
Table 2-18 - UET Control API Address Assignment Request Parameters 109
Table 2-19 - UET Control API Address Assignment Response Parameters 109
Table 2-20 - Completion Counter Requirements 110
Table 2-21 - fi_msg() Receive API Requirements 112
Table 2-22 - fi_msg() Send API Requirements 112
Table 2-23 - fi_tagged() Receive API Requirements 115
Table 2-24 - fi_tagged() Send API Requirements 115
Table 2-25 - fi_rma() API Requirements 117
Table 2-26 - fi_atomic() API Requirements 118
Table 2-27 - Libfabric APIs for which Support is Not Required 120
Table 2-28 - Libfabric API Options for which Support is Not Required 120
Table 2-29 - Packet Delivery Mode Selection Criteria for AI Base Profile 122
Table 2-30 - Packet Delivery Mode Selection Criteria for AI Full Profile 122
Table 2-31 - Packet Delivery Mode Selection Criteria for HPC Profile 122
Table 2-32 - Default Traffic Classes 123
Table 2-33 - UET Security Binding Parameters 126
Table 2-34 - UET Rekey Parameters 126
Table 2-35 - UET Control API Rekey Request Parameters 128
Table 2-36 - Libfabric to IP Header Mapping 129
Table 2-37 - Libfabric Fields Used to Select Packet Delivery Context 130

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Table 2-38 - Summary of Semantic Header Formats 130
Table 2-39 - Mappings for Standard Request (UET HDR_REQUEST_STD) 131
Table 2-40 - Mappings for Optimized Non-Matching (UET HDR_REQUEST_SMALL) 131
Table 2-41 - Criteria for Small Message Header (UET HDR_REQUEST_SMALL) 131
Table 2-42 - Mappings for Small Message (UET HDR_REQUEST_SMALL) 132
Table 2-43 - Mappings for Small RMA (UET HDR_REQUEST_MEDIUM) 132
Table 2-44 - Libfabric Mappings for Rendezvous Send Extension Header 133
Table 2-45 - Verb Objects 136
Table 2-46 - UET Verbs Objects 139
Table 2-47 - Memory Region Binding Requirements 141
Table 2-48 - SES Field Mapping to UET Verbs 144
Table 2-49 - INC Collective Operations 156
Table 2-50 - INC Reduction Operations 157
Table 2-51 - Required Datatype Combinations 158
Table 2-52 - INC Operation ses.inc_info Encoding 162
Table 2-53 - Dataplane Participation 165
Table 2-54 - Fabric Capabilities 175
Table 2-55 - Switch INC Resources 175
Table 3-1 - Profile Requirements for Supporting UET Transaction Types 188
Table 3-2 - Addressing Requirements for Implementations of Profiles 189
Table 3-3 - Profile Buffer Behavior Requirements 189
Table 3-4 - Profile Summary—SES header formats 190
Table 3-5 - Profile Summary—PDS ordering modes 191
Table 3-6 - Profile Summary—Best Effort 192
Table 3-7 - Profile Summary—Lossless 192
Table 3-8 - Profile Summary—INC 192
Table 3-9 - Standard Header Format Fields when ses.som is 1 219
Table 3-10 - Standard Header Format Fields when ses.som is 0 221
Table 3-11 - Optimized Header Format Fields 225
Table 3-12 - Unified Local Semantic Header with ac=0 (UET_HDR_REQUEST_LCL) 227
Table 3-13 - Unified Local Semantic Header with ac=1 (UET_HDR_REQUEST_LCL) 230
Table 3-14 - Target Address and Offset when ses.ac=1 230
Table 3-15 - Synchronized Operation Extension Header Format 233
Table 3-16 - Response Header Fields 234
Table 3-17 - Response with Data Header Fields 235
Table 3-18 - Optimized Response with Data Header Fields 236
Table 3-19 - Unified Local Semantic Header Response Header (UET_HDR_RESPONSE_DATA_LCL) 238
Table 3-20 - Parsing Guide 238
Table 3-21 - Header Formats and Legal Opcodes 239
Table 3-22 - Next Header Enumeration 275
Table 3-23 - Supported Request Messages (Opcode) 276
Table 3-24 - Supported Response Messages (Opcode) 276

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Table 3-25 - Defined Semantic Return Codes 277
Table 3-26 - List Where the Message was Delivered 279
Table 3-27 - Atomic Operation Opcodes 279
Table 3-28 - Supported Atomic Datatypes 280
Table 3-29 - AMO Semantic Control 281
Table 3-30 - Valid Combinations of Operations and Datatypes (Alternative) 281
Table 3-31 - Synchronized Operation Error Cases 287
Table 3-32 - PDS Terminology 292
Table 3-33 - Packet Contexts 299
Table 3-34 - Summary of Function-Based PDS-SES Interface Example 300
Table 3-35 - PDS Configuration Parameters 303
Table 3-36 - PDS Status and Error Indications 305
Table 3-37 - PSN Offset Field 313
Table 3-38 - Fields of UET Entropy Header 332
Table 3-39 - Fields of PDS Prologue 334
Table 3-40 - Header Fields for RUD/ROD Request 335
Table 3-41 - RODL Request Fields 336
Table 3-42 - RODL Request Fields over UFH_16 337
Table 3-43 - Header Fields for RUD/ROD Request with CC State 338
Table 3-44 - Header Fields for ACK 339
Table 3-45 - Header Fields for ACK_CC 340
Table 3-46 - Header Fields for ACK_CCX 341
Table 3-47 - Header Fields for Control Packet 342
Table 3-48 - Header Fields for RUDI Request /Response 343
Table 3-49 - Header Fields for NACK 344
Table 3-50 - Header Fields for NACK_CCX 345
Table 3-51 - Header fields for UUD Request 346
Table 3-52 - Example PSN OFFSET Calculation 349
Table 3-53 - REQ Field Definition 352
Table 3-54 - pds.flags by pds.type 352
Table 3-55 - pds.flags for RODL over UFH_16 353
Table 3-56 - NACK Payload Contents 353
Table 3-57 - CC_TYPE Format 354
Table 3-58 - CCX_TYPE Format 354
Table 3-59 - NCCX_TYPE Contents 354
Table 3-60 - PSN Tracking Resources per PSN Range 355
Table 3-61 - Triggers for Setting pds.flags.ar in PDS Requests when Using Coalesced ACKs 360
Table 3-62 - Triggers for Generating an ACK 361
Table 3-63 - ACK Configuration Parameters 361
Table 3-64 - Local Variables for Last Packet ACK Trigger - Coalesced ACKs 362
Table 3-65 - Local Variables for SACK Bitmap - Coalesced ACKs 363
Table 3-66 - Triggers for Generating a NACK 368

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Table 3-67 - Configuration Parameters for Handling NACK and RTO 370
Table 3-68 - PDS NACK Codes 371
Table 3-69 - Rules for Constructing UET_DEFAULT_RESPONSE 375
Table 3-70 - Loss Detection 376
Table 3-71 - Rules for NACK Generation with Trimmed Packets 377
Table 3-72 - Early Loss Detection Fields per CCC—OOO Example 378
Table 3-73 - UET_HDR_CP_ACK Payload 387
Table 3-74 - Close Command CP ACK Payload 391
Table 3-75 - Credit CP Payload 393
Table 3-76 - Credit Request CP Payload 393
Table 3-77 - Negotiation CP Payload - General 394
Table 3-78 - Negotiation Type (pds.neg_type) 395
Table 3-79 - Negotiation CP Payload for pds.neg_type = PDS_NEG_ON_OFF 396
Table 3-80 - Negotiation CP Payload pds.neg_on_off_mask Format 396
Table 3-81 - CP ACK Payload Fields for pds.neg_type = PDS_NEG_ON_OFF 396
Table 3-82 - PDS CP Summary 397
Table 3-83 - Reserved PDC Configuration 399
Table 3-84 - Sequence Diagram Key 400
Table 3-85 - UET DSCP Mappings without Trimming 445
Table 3-86 - UET DSCP Mappings with Trimming 446
Table 3-87 - UET DSCP to TC Mappings for Best Effort Networks 448
Table 3-88 - UET Control Packet to TC Mappings for Lossless Networks 449
Table 3-89 - Header Fields for pds.req_cc_state for RCCC/TFC 461
Table 3-90 - pds.ack_cc_state for ACK_CC with CC_TYPE = CC_NSCC 462
Table 3-91 - pds.ack_cc_state for ACK_CC with CC_TYPE = CC_CREDIT 464
Table 3-92 - CSIG Reflection Header Fields for Compact CSIG Tag 465
Table 3-93 - CSIG Reflection Header Fields for Wide CSIG Tag 465
Table 3-94 - pds.ack_ccx_state for ACK_CCX and CCX_TYPE = NCSS_CSIG 466
Table 3-95 - pds.ack_ccx_state for ACK_CCX with CCX_TYPE = RCCC_CSIG 467
Table 3-96 - pds.nack_ccx_state for NCXX_TYPE = CSIG 467
Table 3-97 - Congestion Control Configuration Parameters 480
Table 3-98 - NSCC Source State 482
Table 3-99 - NSCC Destination CC State 490
Table 3-100 - RCCC Source State 495
Table 3-101 - RCCC Global Destination State 497
Table 3-102 - RCCC Per-Source Destination State 498
Table 3-103 - RCCC Global Destination State for Aggregate Rate Control 503
Table 3-104 - RCCC Aggregate Rate Control Parameters 503
Table 3-105 - YAML Schema Concepts and Primitives 520
Table 3-106 - Mandatory Signal and Global Variable Accumulation Operators 522
Table 3-107 - Optional Signal Accumulation Operators 523
Table 3-108 - Signal Access and Management Functions 523

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Table 3-109 - Mandatory Signals 524
Table 3-110 - CC Signal Accumulation Functions 527
Table 3-111 - LB Signal-Accumulation Functions (ET = Event Table, PP = Per-Packet) 529
Table 3-112 - Optional Signals 531
Table 3-113 - Trigger Field Resolution 533
Table 3-114 - Trigger Modification API 533
Table 3-115 - Dropped Event Detection Functions 534
Table 3-116 - Per-EV Access and Manipulation APIs 535
Table 3-117 - Per-Packet Operation Mode 535
Table 3-118 - Mandatory CCC Control Functions 536
Table 3-119 - Optional CCC Control Functions 536
Table 3-120 - Local and Global State Controls 536
Table 3-121 - Telegram Type Controls 538
Table 3-122 - Telegram Field Access Controls 538
Table 3-123 - Telegram Transmit Controls 539
Table 3-124 - Control Packet Sending 540
Table 3-125 - Load Balancing Controls 540
Table 3-126 - ERROR State Transition Function 542
Table 3-127 - Threat Model Definitions 548
Table 3-128 - Tools Available to Attacker 549
Table 3-129 - Threats and Mitigations 550
Table 3-130 - Example Functions between PDS/SDME and TSS 554
Table 3-131 - Example Functions between link layer and TSS 555
Table 3-132 - Client-Server Key Generation 558
Table 3-133 - SDKDB Fields 565
Table 3-134 - KDF Parameter Summary for AES-CMAC-256 568
Table 3-135 - TSS Headers Fields 579
Table 3-136 - IV Construction 580
Table 3-137 - TSS Security Counters 584
Table 3-138 - TSS Security Parameters 584
Table 3-139 - TSS Security Events/Errors 585
Table 4-1 - Trim Size Requirements for Various Transport Protocols 595
Table 4-2 - 4-Byte Compact CSIG Tag Field Definitions 606
Table 4-3 - 8-Byte Wide CSIG Tag Field Definitions 606
Table 4-4 - Signal Types for the Compact Tag 607
Table 4-5 - Signal Types for the Wide Tag 607
Table 4-6 - Quantization Algorithm—Compact Tag 609
Table 4-7 - LLDP Port Configuration for CSIG 613
Table 4-8 - Signal Initial Values 614
Table 4-9 - CSIG Profiles 620
Table 4-10 - Available Bandwidth Computation Terms 621
Table 4-11 - Available Bandwidth /C Computation Terms 623

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Table 5-1 - MII Format for UE Link Frame Preamble 633
Table 5-2 - 64B/66B Block Format for UE Link Frame Preamble 634
Table 5-3 - MAC Client to LLR Transmit Path Additional Fields 634
Table 5-4 - LLR to MAC Control to MAC Transmit Path Additional Fields 634
Table 5-5 - MAC to MAC Control to LLR Receive Path Additional Fields 635
Table 5-6 - LLR to MAC Client Receive Path Additional Field 635
Table 5-7 - LLR Control Ordered Sets 636
Table 5-8 - UE LLR Link Control Ordered Set 64B/66B Block Format 636
Table 5-9 - LLR Configuration Registers 637
Table 5-10 - LLR TX Path Variables 639
Table 5-11 - LLR RX Path Variables 645
Table 5-12 - Control Ordered Set Transmission Priority 647
Table 5-13 - LLR counters 648
Table 5-14 - Configuration Parameters and Initialization 654
Table 5-15 - Sender Cyclic Counters 655
Table 5-16 - Receiver Cyclic Counters 655
Table 5-17 - Sender State Variables 656
Table 5-18 - Counter and Variable Widths and Initialization 656
Table 5-19 - CBFC Message Types 657
Table 5-20 - CF_Update CtlOS Message Format 662
Table 5-21 - CF_Update CtlOS Data Field Definitions 662
Table 5-22 - CC_Update Message Packet Fields 663
Table 5-23 - CBFC TLV Receiver Port Information Fields 667
Table 5-24 - CBFC TLV Receiver Per-VC Information Fields 667
Table 5-25 - CBFC TLV Sender Information Fields 668
Table 5-26 - LLR and CBFC Combined Handling for Data Packets 674
Table 5-27 - Conformance Requirements 679
Table 5-28 - CtlOS Format on xMII 680
Table 5-29 - UE CtlOS Message Type Values 681
Table 5-30 - Modified 64B/66B PCS Encoding for Ordered Sets 681
Table 5-31 - CC_Update Message - Counter values 683
Table 5-32 - LLDP Database Group Addresses 687
Table 5-33 - UE Link Negotiation Options TLV LLR-W Field Options 689
Table 5-34 - UE Link Negotiation Options TLV CSIG-C Field Options 690
Table 5-35 - UE Link Negotiation Options TLV UFH-C and UFH-S Field Options 691
Table 5-36 - UE Link Negotiation CBFC Error Codes 695
Table 5-37 - UE Link Negotiation CBFC TLV R_PktID_Sel Field Options 697
Table 5-38 - Header Optimizations 717
Table 5-39 - IEEE Std 802-2024 SLAP Encoding 720
Table 5-40 - UFH_32 Field Names 721
Table 5-41 - UFH_16 Field Names 721
Table 5-42 - Header Overhead 726

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Table 5-43 - Efficiency Comparison 726
Table 6-1 - Ethernet standard sequence ordered sets on MII 732
Table 6-2 - Ethernet standard 64B/66B sequence ordered set encoding 733
Table 6-3 - Ordered set format on MII 733
Table 6-4 - Modified 64B/66B PCS encoding for ordered sets 733
Table 6-5 - PCS stateless decoder rules for 100GBASE-R without FEC error overwrite marking 735
Table 6-6 - RS Layer CtlOS spacing constraints 736
Table 6-7 - Ethernet-specified FLR and resulting MTBPE 738
Table 6-8 - CIR and CG sizes 739
Table 6-9 - UCR to FLR conversion factors 740
Table 6-10 - Codeword times 740
Table 6-11 - MTBPE to UCR conversion 741
Table 6-12 - Number of codewords received in one minute, N_{total} (approximate) 742
Table 6-13 - Calculation of UCR_{est2} from CCR 743
Table 6-14 - Mapping from UCR to P6, P8, and P10 assuming uncorrelated errors 746
Table 7-1 - OCP Ethernet NIC profile use cases 756

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Figures

Figure 1-1 - Working Group Organization 29
Figure 1-2 - Example Full Header Format 47
Figure 1-3 - Example Individual Header Format 48
Figure 1-4 - Example Sequence Diagram Figure 49
Figure 1-5 - System Overview 51
Figure 1-6 - Multi-plane Networks and Multi-port FEPs 52
Figure 1-7 - Parallel Job Model 53
Figure 1-8 - Client/Server Job Model 54
Figure 1-9 - Addressing Modes 55
Figure 1-10 - Transport Data Delivery and Packet Delivery Contexts 57
Figure 1-11 - UE Software Endpoint Stack 60
Figure 1-12 - Switch Module Layering 61
Figure 1-13 - Network Types 63
Figure 1-14 - Layered View of Networking Functionality 66
Figure 1-15 - Traffic Class Mapping 69
Figure 1-16 - UE Specifications by Layers 71
Figure 1-17 - UE Link Layer Specification Focus Areas 74
Figure 1-18 - UE Physical Layer Specification Focus Areas 76
Figure 2-1 - Components and Interfaces Defined By UEC 78
Figure 2-2 - Libfabric Software Architecture 82
Figure 2-3 - Libfabric UET Provider Software Architecture 83
Figure 2-4 - JobID Assignment at Job Initialization Time 90
Figure 2-5 - UET Control API JobID Mapping Request Structure 91
Figure 2-6 - UET Control API JobID Unmapping Request Structure 91
Figure 2-7 - Key Libfabric Objects and Associated APIs 97
Figure 2-8 - UET Address Assignment Architecture 107
Figure 2-9 - Transmit Queue Example 124
Figure 2-10 - Example Receive Queue and Registered MR Data Structures 125
Figure 2-11 - Rekey Parameter Acquisition Architecture 127
Figure 2-12 - Libfabric Mapping to UET Wire Protocol Headers 129
Figure 2-13 - UET Memory Region Access 140
Figure 2-14 - Verbs to UET Object Mapping 141
Figure 2-15 - Verbs Memory Region Access 141
Figure 2-16 - Verbs Absolute Addressing Model 143
Figure 2-17 - Verbs Relative Addressing Model 144
Figure 2-18 - QP State Machine 146
Figure 2-19 - CQ Management 148
Figure 2-20 - Verbs Example Absolute Addressing 149
Figure 2-21 - Verbs Example Relative Addressing 150

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Figure 2-22 - Example INC Communicator Creation 159
Figure 2-23 - Example INC Communicator Teardown 160
Figure 2-24 - INC Operation ses.inc_info Encoding 162
Figure 2-25 - Allreduce Sequence 164
Figure 2-26 - Deferrable RTR Sequence 165
Figure 3-1 - Overview of UET 180
Figure 3-2 - UET Packet Structure 181
Figure 3-3 - Component Interface Overview 186
Figure 3-4 - Relative Addressing 196
Figure 3-5 - Absolute Addressing 197
Figure 3-6 – Simplified Absolute Addressing 198
Figure 3-7 - Relative Addressing for RMA Operations using Resource Index Namespace 198
Figure 3-8 - Relative Addressing for RMA Operations using PIDonFEP Namespace 199
Figure 3-9 - Overview of Relative Addressing for RMA Operations Using Optimized Headers 200
Figure 3-10 - Overview of Direct Memory Addressing 200
Figure 3-11 - Standard Header Format when ses.som is 1 (UET_HDR_REQUEST_STD) 218
Figure 3-12 - Standard Header Format when ses.som is 0 (UET_HDR_REQUEST_STD) 221
Figure 3-13 - Standard Header Format as Used for Deferrable Sends 223
Figure 3-14 - Standard Header Format as Used for Ready-to-Restart Requests 224
Figure 3-15 - Optimized, Non-Matching Format (UET_HDR_REQUEST_SMALL) 224
Figure 3-16 - Small Message/Small RMA Format (UET_HDR_REQUEST_MEDIUM) 226
Figure 3-17 - Unified Local Semantic Header with ac=0 (UET_HDR_REQUEST_LCL) 227
Figure 3-18 - Packed messages using ses.mm=1 228
Figure 3-19 - Unified Local Semantic Header with ac=1 (also uses UET_HDR_REQUEST_LCL) 229
Figure 3-20 - Unified Local Semantic Header with ac=1, ext_hdr > 0, and atomic operation 231
Figure 3-21 - Rendezvous Extension Header Format 231
Figure 3-22 - Atomic Operation Extension Header Format 232
Figure 3-23 - Compare and Swap Operation Atomic Header and Payload Format 233
Figure 3-24 - Synchronized Operation Extension Header Format 234
Figure 3-25 - Semantic Response Header Format (UET_HDR_RESPONSE) 234
Figure 3-26 - Semantic Response with Data Header Format (UET_HDR_RESPONSE_DATA) 235
Figure 3-27 - Optimized Response with Data Header Format (UET_HDR_REQUEST_DATA_SMALL) 236
Figure 3-28 - Unified Local Semantic Header Response Header (UET_HDR_RESPONSE_DATA_LCL) 238
Figure 3-29 - Single-Packet Request, Expected Message 254
Figure 3-30 - Multi-Packet Request, Expected Message 255
Figure 3-31 - Multipacket Read Request – Standard 256
Figure 3-32 - Multi-Packet Read Request – Large PDS_MAX_ACK_DATA 257
Figure 3-33 - Rendezvous Transaction, Expected Message Case 259
Figure 3-34 - Rendezvous Transaction, Unexpected Message Case 260
Figure 3-35 - Deferrable Sends, Expected Message Case 261
Figure 3-36 - Deferrable Sends, Unexpected Message Case 262
Figure 3-37 - Deferrable Sends, Unexpected Message Case, No Reserved Buffer 263

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Figure 3-38 - Synchronized Operation Sequence with Write with Data 265
Figure 3-39 - Synchronized Operation Sequence with Atomic 266
Figure 3-40 - Single Packet Messages using Backoff and Retry 267
Figure 3-41 - Messages using Resource Index Generation 268
Figure 3-42 - Multi-Packet Request, Expected Message, with Initiator Error 269
Figure 3-43 - Multi-Packet Request, Expected Message, with Message Error 270
Figure 3-44 - Unified Local UET_WRITE with ses.dc=0 270
Figure 3-45 - Unified Local UET_WRITE with ses.dc=1 271
Figure 3-46 - Unified Local UET_READ 271
Figure 3-47 - Use of MSN Tables 288
Figure 3-48 - Tag-Based Sequence for *CCL Send and Receive 289
Figure 3-49 - Rendezvous Queue Data Structure 289
Figure 3-50 - Write-Based Sequence for *CCL Send and Receive 290
Figure 3-51 - PDS High-Level Architecture Diagram 292
Figure 3-52 - Illustrated PDS Terms 295
Figure 3-53 - Example PDS-SES Interface 300
Figure 3-54 - Example PDS-SES Interface between Initiator and Target 302
Figure 3-55 - PDC Startup using SYN Flag 316
Figure 3-56 - PDC Establishment State Machine 317
Figure 3-57 - Single PDC Close State Machine 323
Figure 3-58 - Sequence for PDC Setup and Teardown 324
Figure 3-59 - PDS Top Level State Machines 326
Figure 3-60 - PDS Manager State Machine 327
Figure 3-61 - PDC Initiator State Machine 330
Figure 3-62 - PDS Target State Machine 331
Figure 3-63 - UET Entropy Header Format 332
Figure 3-64 - UET UFH Entropy Header 333
Figure 3-65 - PDS Prologue Format 333
Figure 3-66 - RUD/ROD Request Header Format 334
Figure 3-67 - RODL Header Format - General 336
Figure 3-68 - RODL Header Format over UFH_16 337
Figure 3-69 - RUD/ROD Request Header with CC State Format 338
Figure 3-70 - ACK Header Format 339
Figure 3-71 - ACK_CC Format 340
Figure 3-72 - ACK_CCX Format 341
Figure 3-73 - Control Packet Header Format 342
Figure 3-74 - RUDI Request Header Format 343
Figure 3-75 - NACK Header Format 344
Figure 3-76 - NACK_CCX Header Format 345
Figure 3-77 - UUD Header Format 346
Figure 3-78 - RUD/ROD Default SES Header Format 346
Figure 3-79 - Example ACK Packet 358

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Figure 3-80 - ACK State Sections 358
Figure 3-81 - SACK Bitmap 364
Figure 3-82 - PDS_MPR Concept 367
Figure 3-83 - PDS NACK Format 368
Figure 3-84 - NACK Sections 369
Figure 3-85 - Example Transmit Scheduler 376
Figure 3-86 - Example PSN Tracking per EV 383
Figure 3-87 - UET_HDR_CP_ACK Payload Format 387
Figure 3-88 - CP Format 388
Figure 3-89 - Close Command CP Payload Format 391
Figure 3-90 - Close Command CP ACK Payload Format 391
Figure 3-91 - Credit CP Payload 393
Figure 3-92 - Credit Request CP Payload 393
Figure 3-93 - Negotiation CP Payload Format - General 394
Figure 3-94 - Negotiation CP Payload Format 395
Figure 3-95 - Format of CP ACK Payload (pds.neg_type = PDS_NEG_ON_OFF) 396
Figure 3-96 - Example Sequence with Key 400
Figure 3-97 - Standard RUD Send Sequence for Single- Packet Message, Non-Guaranteed Delivery 403
Figure 3-98 - Standard RUD Send Sequence for Single- Packet Message, Guaranteed Delivery 404
Figure 3-99 - Standard RUD Send Sequence for Multi-Packet Message, Non-Guaranteed Delivery 405
Figure 3-100 - Standard Sequence for RUD Send of Multi-Packet Message with Guaranteed Delivery .. 406
Figure 3-101 - Standard RUD Sequence for Single-Packet Read–Standard SES Header 408
Figure 3-102 - Standard RUD Sequence for Single-Packet Read–Medium & Small SES header 409
Figure 3-103 - Standard RUD Sequence for Multi-Packet Read 411
Figure 3-104 - PDS NACK Sequence for RUD Send of Single-Packet Message 412
Figure 3-105 - PDS Request Dropped Sequence for RUD Send of Single-Packet Message 413
Figure 3-106 - PDS ACK Dropped Sequence for RUD Send of Single-Packet Message 414
Figure 3-107 - ROD Send Sequence for Dropped PDS Request Packet 418
Figure 3-108 - ROD Send Sequence for Dropped ACK Packet Using ACK Request 420
Figure 3-109 - Bidirectional PDC 423
Figure 3-110 - Standard RUDI RMA Write Sequence for Multi-Packet Message 425
Figure 3-111 - Standard RUDI RMA Read Sequence for Multi-Packet Message 426
Figure 3-112 - Full Header: UET/UDP/IP – no UET CRC, no TSS 428
Figure 3-113 - Full Header: UET/UDP/IP with UET CRC, no TSS 429
Figure 3-114 - Full Header: UET/UDP/IP with Encryption 429
Figure 3-115 - Full Header: UET/IP no UET CRC, no TSS 430
Figure 3-116 - Full Header: UET/IP with UET CRC, no TSS 430
Figure 3-117 - Full Header: UET/IP with Encryption 431
Figure 3-118 - Header Stack for General UET and RUD Requests over the UFH_32 header 432
Figure 3-119 - Header Stack for UET RODL 433
Figure 3-120 - Header Format for UET over the UFH_16 Header 434
Figure 3-121 - UET CRC Coverage 435

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Figure 3-122 - UET CRC coverage when using the UFH_16 header 437
Figure 3-123 - Example packets using padding 438
Figure 3-124 - PDC mapped over Best-Effort Ethernet: Traffic Class Mapping (ROD and RUD) 448
Figure 3-125 - PDC mapped over Lossless Ethernet: Traffic Class Mapping (ROD and RUD) 449
Figure 3-126 - CCC State Machine 459
Figure 3-127 - pds.req_cc_state Header Format for RCCC / TFC 461
Figure 3-128 - pds.ack_cc_state for UET TYPE = ACK_CC and CC_TYPE = CC_NSCC 462
Figure 3-129 - pds.ack_cc_state for UET TYPE = ACK_CC and CC_TYPE = CC_CREDIT 464
Figure 3-130 - CSIG Reflection Header for Compact CSIG Tag 465
Figure 3-131 - CSIG Reflection Header for Wide Compact CSIG Tag 465
Figure 3-132 - pds.ack_ccx_state for UET_TYPE = ACK_CCX and CCX_TYPE = NSCC_CSIG 466
Figure 3-133 - pds.ack_ccx_state for ACK_CCX and CCX_TYPE = RCCC_CSIG 466
Figure 3-134 - pds.nack_ccx_state for NCCX_TYPE = CSIG 467
Figure 3-135 - PCM Logical Architecture Overview 518
Figure 3-136 - Example PDS-TSS-Link Logical Interface 553
Figure 3-137 - TSS Architecture 556
Figure 3-138 - Single Job in a Secure Domain 557
Figure 3-139 - Multiple Jobs in a Single SD 557
Figure 3-140 - Secure Domain with Four FEPs 559
Figure 3-141 - KDF Rekeying Example 563
Figure 3-142 - SDKDB database and crypto interfaces 567
Figure 3-143 - Cluster KDF mode for IPv4 and Packets with Explicit SSI 570
Figure 3-144 - Cluster KDF mode for IPv6 without explicit SSI 571
Figure 3-145 - UET Secure Transport Packets 575
Figure 3-146 - TSS with Native IPv4 Transport 576
Figure 3-147 - TSS with Native IPv6 Transport 577
Figure 3-148 - TSS with UDP Encapsulation 578
Figure 3-149 - TSS with UFH_32 Encapsulation 579
Figure 4-1 - Drop Threshold Settings 598
Figure 4-2 - Example of CSIG Endpoint and Fabric Processing 604
Figure 4-3 - 4-Byte Compact CSIG Tag 605
Figure 4-4 - 8-Byte Wide CSIG Tag 606
Figure 4-5 - Example Local Signal Quantization and Packet Signal Compare-and-Update 608
Figure 4-6 - CSIG LLDP Example 614
Figure 4-7 - CSIG Reflection Entity Interactions 617
Figure 4-8 - CSIG Tag Position in Ethernet Headers 618
Figure 4-9 - Example CSIG Tag Position with MACSec 619
Figure 4-10 - Timestamping Points in Switch Pipeline 624
Figure 4-11 - CSIG Domain 628
Figure 5-1 - Architectural position of LLR. 631
Figure 5-2 - MAC Control interface connectivity. 632
Figure 5-3 - LLR transmit state machine 639

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Figure 5-4 - ACK/NACK transmit state machine. 644
Figure 5-5 - Basic Packet Data and Credit Update Sequence 652
Figure 5-6 - Cyclic Counter Example 652
Figure 5-7 - Architectural Position of CBFC in Ethernet Functional Layer Model 654
Figure 5-8 - Packet and CBFC Message Sequence with Counter Updates 658
Figure 5-9 - CC_Update Packet Format 664
Figure 5-10 - Virtual Channel State 666
Figure 5-11 - Lossless VC Initialization Process 669
Figure 5-12 - Lossless VC Removal Process 672
Figure 5-13 - Example CC_Update Packet During LLR Discard Window 676
Figure 5-14 - Example CC_Update Packet Triggers the LLR Discard Window 677
Figure 5-15 - Example CF_Update CtlOS 682
Figure 5-16 - Example CC_Update Frame 683
Figure 5-17 - LLDP Agent With the UE LLDP Databases 686
Figure 5-18 - Ethernet Frame Containing an LLDPDU 687
Figure 5-19 - UE Link Negotiation Options TLV (Options TLV) 688
Figure 5-20 - UE Link Negotiation CBFC TLV (CBFC TLV) 694
Figure 5-21 - YANG Root Hierarchy with UE Link LLDP Extensions 699
Figure 5-22 - UE Link Negotiation Options TLV Model 700
Figure 5-23 - UE Link Negotiation CBFC TLV Model 701
Figure 5-24 - UFH Header Overlay 718
Figure 5-25 - UFH Header Identification 719
Figure 5-26 - UFH Formats 720
Figure 5-27 - UFH Coexistence and Extensibility 722
Figure 5-28 - Conceptual Layering of UFH 723
Figure 5-29 - UFH Domains 726
Figure 5-30 - Efficiency Comparison 727
Figure 6-1 - UCR_{est2} as a function of CCR, assuming uncorrelated errors 744
Figure 6-2 - Expected values of P6, P8, and P10 as a function of UCR 747
Figure 7-1 - UE Management Architecture 753
Figure 7-2 - UE Fabric Manager 755
Figure 7-3 - Server Management Architecture 755
Figure 7-4 - Network Switch Planes 757
Figure 7-5 - UE Switch Manageability Architecture 758
Figure 7-6 - SAI Architecture 759
Figure 8-1 - Example of Profile Matrix 761
Figure 8-2 - Example of Compliance Checklist 761

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Acknowledgments

This specification is developed under the Process, Procedures, and Guidelines of the Ultra Ethernet Consortium (UEC). The organization would like to thank all that are involved. A special acknowledgment goes out to the following individuals for their leadership, authoring, and technical contributions to this specification.

Steering Committee at Time of Publication

Role	Name	Affiliation
Chair	J Metz	AMD
Vice Chair	Abdul Kabbani	Microsoft
TAC Chair	Tom Emmons	Arista
TAC Vice Chair	Brad Matthews	Marvell

Working Group Leadership at Time of Publication

Working Group	Name	Affiliation
Physical Layer WG	Jannik Hammel Nielsen	Huawei
	Adee Ran	Cisco
Link Layer WG	Bob Alverson	HPE
	Yan Zhuang	Huawei
Transport Layer WG	Torsten Hoefler	Microsoft
	Karen Schramm	Broadcom
Software Layer WG	Rip Sohan	Broadcom
	Jianxin Xiong	Intel
Management WG	Ravindranath Kanakarajan	Marvell
	Hemal Shah	Broadcom
Compliance & Test WG	Dave Estes	Viavi Solutions
	Cliff Tavares	Keysight
Performance & Debug WG	Rajalaxmi Angadi	Intel
	Nathan Hanford	Lawrence Livermore NL
Storage WG	Venkat Goli	Broadcom
	Claudio DeSanti	Dell

Specification Development

Role	Name	Affiliation
Author	Cedell Alexander	Broadcom
Author	Bob Alverson	HPE
Author	Surendra Anubolu	Broadcom
Author	Jeremias Blendin	AMD
Author	Paul Bottorff	HPE
Author	Josh Collier	Intel
Author	Eric Davis	Broadcom
Author	Mark Handley	Broadcom

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Role	Name	Affiliation
Author	Sean Hefty	Nvidia
Author	Torsten Hoefler	Microsoft
Author	Cathy Huang	Intel
Author	Adbul Kabbani	Microsoft
Author	Brad Karp	Google
Author	Vivek Kashyap	Intel
Author	Jai Kumar	Broadcom
Author	Vlad Olteanu	Broadcom
Author	Eugene Opsasnick	Broadcom
Author	Rong Pan	AMD
Author	Costin Raiciu	Broadcom
Author	Adee Ran	Cisco
Author	Karen Schramm	Broadcom
Author	Hemal Shah	Broadcom
Author	Rip Sohan	AMD
Author	Eric Spada	Broadcom
Author	Keith Underwood	HPE
Author	Niranjan Vaidya	Broadcom
Contributor	Puneet Agarwal	Marvell
Contributor	Anup Agarwal	Microsoft
Contributor	Nikolay Aleksandrov	Enfabrica
Contributor	Alex Badea	Keysight
Contributor	Przemyslaw Baranski	Marvell
Contributor	Martin Belanger	Dell
Contributor	Rakhahari Bhunia	Keysight
Contributor	Devin Bidstrup	HPE
Contributor	Manomugdha Biswas	Keysight
Contributor	Michael Blocksome	Cornelis Networks
Contributor	Tommaso Bonato	Microsoft
Contributor	Mihai Brodschi	Broadcom
Contributor	Rupa Budhia	Marvell
Contributor	Thomas Calvert	AMD
Contributor	Ron Cohen	Marvell
Contributor	Barry Davis	Intel
Contributor	Mark Debbage	Arista
Contributor	Iulian Dracea	Broadcom
Contributor	Roland Dreier	Enfabrica
Contributor	Uri Elzur	Intel
Contributor	Matthew Fernandez	DreamBig Semiconductor
Contributor	Ahmad Ghalayini	Microsoft
Contributor	Evan Green	Meta
Contributor	Xingling Han	AMD
Contributor	Hugh Holbrook	Arista
Contributor	Nusrat Islam	AMD
Contributor	Vipin Jain	AMD
Contributor	Bob Lantz	HPE
Contributor	Yanfang Le	AMD
Contributor	John Leung	Intel
Contributor	Meg Lin	Marvell

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Role	Name	Affiliation
Contributor	Nicholas Lurski	Microsoft
Contributor	Kingshuk Mandal	Keysight
Contributor	Orr Mazor	NeuReality
Contributor	Antonios Michaloliakos	Meta
Contributor	Guglielmo Morandin	Broadcom
Contributor	Shawn Nicholl	AMD
Contributor	Svyatoslav Nikita	ViraSemi
Contributor	Abhiram Ravi	Google
Contributor	Vedvyas Shanbhogue	Meta
Contributor	Kapil Shrikhande	Marvell
Contributor	Ramesh Sivakolundu	Cisco
Contributor	Ashkan Sobhani	Huawei
Contributor	Chris Sommers	Keysight
Contributor	Jerome Tollet	Cisco
Contributor	Daniel Verkamp	Meta
Contributor	Mike Vildibill	HPE
Contributor	Michael Wilkins	Cornelis Networks
Contributor	Jianxin Xiong	Intel
Contributor	Aibing Zhou	Juniper
Contributor	Yan Zhuang	Huawei
Contributor	Nazar Zibilyuk	ViraSemi
Editor	Paul Congdon	Linux Foundation
Program Manager	Rob Craig	Linux Foundation

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

1 Introduction

1.1 Background

The Ultra Ethernet Consortium (UEC) is an industry effort—involving many contributors including hyperscalers, system vendors, silicon providers, and others—with a mission to enhance Ethernet for use in AI and HPC.

Contributors are aiming to cover all aspects of a successful specification, including software APIs, network protocols, hardware friendliness and scalability, network operation, compliance, and extensibility. In service of the UEC's mission, this document provides a specification of new protocols for use over Ethernet networks and optional enhancements to existing Ethernet protocols that improve performance, function, and interoperability of AI and HPC applications.

The Ultra Ethernet (UE) specification covers a broad range of software and hardware relevant to AI and HPC workloads: from the API supported by UE-compliant devices to the services offered by the transport, link, and physical layers, as well as management, interoperability, benchmarks, and compliance requirements.

UE does not require or mandate changes to the network layer or the Ethernet PHY and link layers. For instance, a UE-compliant implementation might use Ethernet switches common in the market at the time of publishing the specification. However, UE offers optional network, Ethernet PHY, and link layer features that enable better performance for demanding applications. Over time, and as experience is gathered, it is possible that some of these optional features might become commonplace and even required. A UE-compliant implementation supports the mandatory requirements in this specification.

1.1.1 UEC Organization

UE architecture comprises the four lower layers of the classic ISO/OSI networking model along with software services and the APIs that expose these services to the upper layers. Each of the four layers is addressed by a UEC working group (depicted as rows in Figure 1-1), which defines the required architecture with strict requirements and characterization in terms of scalability, capability, performance, and interoperability. The UEC management working group provides Ethernet fabric and endpoint management. The UE management architecture includes management protocols, transports, and data models. The UEC Compliance working group, in collaboration with the Technical Advisory Committee (TAC), defines the compliance and interoperability requirements. The UEC Management and Debug & Performance working groups interact with all the aforementioned working groups.

Additionally, UEC is adding storage services alongside the network services for relevant application and workloads. The working groups depicted as columns in Figure 1-1 were formed after the initial set of working groups depicted as rows. Their contributions to the UE specification and other standalone documents are scheduled for a future release.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

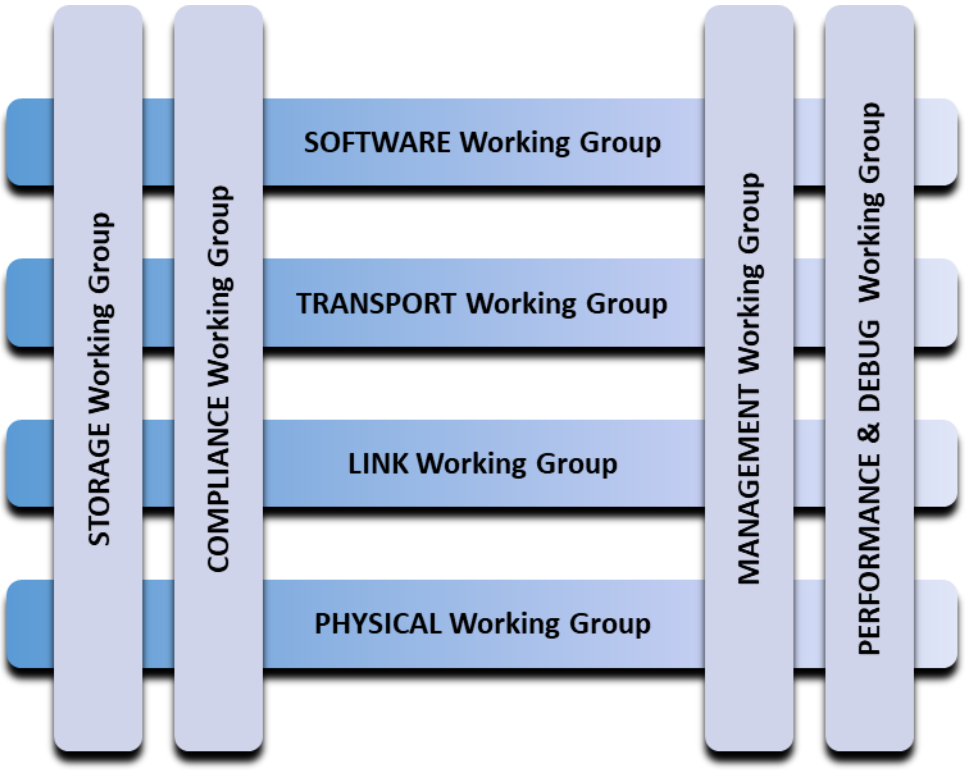


Figure 1-1 - Working Group Organization

UEC is incorporated under the Linux Foundation Joint Development Foundation (JDF) as an International Standards Development Organization (SDO). UE specifications are OPEN for public download. Future versions, once created and ratified, will also be publicly available for download. It is the intention of UEC members to propose their work products to appropriate standards organizations and/or relevant open-source communities to encourage broad adoption and to contribute as appropriate to mainstream industry standardization efforts for Ethernet, Internet Protocol (IP), software, and API development. Potential relevant SDOs to consider include, but are not limited to, IEEE, IETF, OCP, OFA, SONiC/SAI, and various storage and management SDOs.

1.1.2 UE Transport Profiles

UET specifies four profiles: Unified Local, AI Base, AI Full, and HPC. The Unified Local profile is optimized for local networks (i.e., scale up) with an emphasis on wire efficiency and small write/read transactions. The AI Base profile is designed to provide the functionality required by current and future AI applications where high performance is required at the lowest cost. The AI Full profile adds additional features (e.g., deferrable send, exact match, and support for atomic primitives). The HPC profile addresses the needs of High Performance Computing applications and has all the features of the AI Full profile except deferrable send.

Each profile lists the services provided and the distinct features required at the transport layer of a compliant product. The profiles themselves are defined in Section 3.3. The details of the hardware interface at the endpoint are out of scope for the UE specifications. Software APIs to the upper layers are specified to provide interoperability with higher-layer software. The goal is that different vendors'

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

devices supporting a given profile exhibit interoperability and functionality as described in these specifications.

Profiles may include optional-to-implement features. If optional features are implemented, they MUST follow the defined specification to claim compliance.

1.2 UE Specification Conventions

The UE specification uses the following conventions for normative language, informative notes, terminology, units, numbers, and figure formatting.

1.2.1 Normative, Informative, and Implementation Statements

Normative language is identified using terms defined by IETF BCP14. The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “NOT RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described in IETF BCP 14 [4], IETF RFC 2119 [1], and IETF RFC 8174 [2] when, and only when, they appear in all capitals, as shown here.

All text not explicitly identified as informative comment is normative. An [informative] marking in the section title applies to the entire section including any subsections. Diagrams and tables are considered normative unless marked in the title as [informative].

Sections of text are marked as informative using the following convention:

Informative Text:
Informative text is included in this area.

Occasionally, notes to the implementer of this specification are included for informational purposes. These notes are intended to clarify the intent of the specification and to provide guidance to the implementer. They are indicated with the following format:

Implementation Note:
Implementation note text is included in this area.

1.2.2 Terminology

1.2.2.1 Abbreviations

Abbreviation	Definition
AAD	Additional authentication data
ABI	Application binary interface
ACK	Acknowledgement
AEAD	Authenticated encryption associated data
AES	Advanced encryption standard
AI	Artificial intelligence
AN	Association number

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Abbreviation	Definition
API	Application programming interface
AV	Libfabric address vector
BDP	Bandwidth-delay product
BER	Bit error ratio
BTS	Back to sender
BW	Bandwidth
C2C	Chip to chip
C2M	Chip to module
CBFC	Credit-based flow control
CC	Credits consumed (Link Layer) / congestion control (Transport Layer)
CCC	Congestion control context
CCL	Collective communications library
CCR	Corrected codeword ratio
CF	Credits freed
CG	Codeword group
CID	Company identifier
CIR	Codeword interleaving ratio
CL	Credit limit
CMS	Congestion management sublayer
CP	Credit packet
CQ	Libfabric completion queue
CRC	Cyclic redundancy check
CSIG	A layer 2 congestion signaling mechanism
CtlOS	Control ordered set
CU	Credits in use
DPA	Differential power analysis
DIC	Deficit idle count
ECMP	Equal cost multi-path
ECN	Explicit congestion notification (RFC 3168)
EP	Libfabric endpoint
EQ	Libfabric event queue
EQDS	Edge queued datagram service
EV	Entropy value
FA	Fabric address
FEC	Forward error correction
FEP	Fabric endpoint
FI	Fabric interface
FLR	Frame loss ratio
FPC	Frames per codeword

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Abbreviation	Definition
GCM	Galois counter mode
GMAC	Galois message authentication code
gNMI	GRPC network management interface
gNOI	GRPC network operations interface
gRPC	gRPC remote procedure calls
HPC	High-performance computing
ICV	Integrity check value
INC	In-Network Collectives
IP	Internet protocol (RFC 791 and/or RFC 8200)
IPG	Interpacket gap
IV	Initialization vector
KDF	Key derivation function
KMD	Kernel mode driver
LACP	Link Aggregation Control Protocol (IEEE Std 802.1AX)
LAG	Link aggregation
LLDP	Link Layer Discovery Protocol (IEEE Std 802.1AB)
LLR	Link layer retry
MAC	Media access control
MDIO	Management data input/output
MID	Message identifier
MII	Media-independent interface
MMF	Multi-mode fiber
MPI	Message passing interface
MR	Libfabric memory region
MTBPE	Mean time between PHY errors
MTTFPA	Mean time to false 802.3 packet acceptance
NACK	Negative acknowledgement
OSI	Operating system instance
PASID	Process address space identifier
PCS	Physical coding sublayer
PDC	Packet delivery context
PDS	Packet delivery sublayer
PFC	Priority-based flow control
PGAS	Partitioned global address space
PHY	Physical layer device
PID	Process identifier
PLS	Physical layer signaling
PSN	Packet sequence number
QoS	Quality of service
RI	Resource index

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Abbreviation	Definition
RoCE	RDMA over converged Ethernet
ROD	Reliable ordered delivery
RODL	Reliable ordered delivery for local networks
RS	Reconciliation sublayer
RS-FEC	Reed-Solomon forward error correction
RTO	Retransmission timeout
RTR	Restart transmission request
RTT	Round trip time
RUD	Reliable unordered delivery
RUDI	Reliable unordered deliver of idempotent operations
SACK	Selective acknowledgement
SD	Secure domain
SDK	Secure domain key
SDKDB	Secure domain key database
SDI	Secure domain identifier
SDME	Secure domain management entity
SER	Symbol error ratio
SES	Semantic sublayer
SHMEM	Shared memory / Symmetric hierarchical memory
SMF	Single-mode fiber
SSI	Secure source identifier
TC	Traffic class
TSC	Timestamp counter
TSS	Transport security sublayer
UCR	Uncorrectable codeword ratio
UE	Ultra Ethernet
UET	Ultra Ethernet Transport
UEC	Ultra Ethernet Consortium
UFH	Unified Forwarding Header
UUD	Unreliable unordered delivery
VAS	Virtual address space
VC	Virtual channel
VLAN	Virtual LAN (local area network)
WDM	Wavelength division multiplexing
YANG	Yet another next generation

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

1.2.2.2 Terms

Class	Term	Description
Operating System Communication	CPU	Central processing unit; a generic processor for arbitrary computation functions.
	Memory	An electronic holding place for the instructions and data used by CPUs and accelerators.
	Operating system instance (OSI)	An instance of an operating system (e.g., a virtual machine).
	Process	An instance of a program executing on an OSI owned by a specific user and having a private virtual address space (VAS). A process is identified by a process ID unique to the OSI it runs in.
	Process address space ID (PASID)	A unique identifier for a VAS within each OSI. A target PASID can be determined based on a combination of JobID and PIDonFEP.
	Process ID (PID)	Operating system-assigned identifier for a process.
	User	An entity with access privileges to nodes of a cluster capable of executing compute processes.
	Virtual address space (VAS)	Process-specific virtual address for memory allocation.
Fabric Communication	Absolute addressing	A mode of addressing a resource within the client/server job model. Includes three complementary parts of the destination address: (1) an FA identifying a FEP, (2) a PIDonFEP interpreted without a JobID, and (3) a Resource Index.
	Accelerator	A compute module or device designed for the efficient execution of specific functions.
	Acknowledgment packet (ACK)	A packet used by the UET to implement reliability. ACKs are transmitted by the destination PDC to the source PDC to indicate successful reception of packets at the PDS layer. ACKs can carry a semantic response.
	Best-effort network	A network (as opposed to a lossless network) where packets are sent on at least some links without any explicit communication of buffer

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Class	Term	Description
		availability between a transmitter and the link peer. Packets may be dropped due to insufficient buffering.
	Cluster	A set of nodes connected by one or more fabric planes.
	Congestion control context (CCC)	Used to control the rate of data transfer between two FEPs for RUD and ROD traffic. In some cases, a CCC is shared by one or more PDCs.
	Congestion management sublayer (CMS)	The part of the Ultra Ethernet Transport (UET) protocol responsible for managing congestion.
	Entropy value (EV)	The value of the field within a packet header (e.g., UDP source port) used to load balance packets across paths within the fabric
	Fabric	One or more fabric planes.
	Fabric address (FA)	IPv4 (RFC 791) or IPv6 (RFC 8200) address. When using UFH headers, the fabric addresses are in the **ufh.dst_addr** and **ufh.src_addr** fields.
	Fabric endpoint (FEP)	A logical entity addressable by a single (assigned) FA. The UE transport protocol, including the optional security context, terminates at a FEP. A FEP connects to a fabric using a port and can be used only by a single OSI. A FEP can have multiple ports using a single FA, as long as the ports are each connected to completely isolated fabric plane. A node can have one or more FEPs.
	Fabric path or path	An ordered set of links (hops between nodes and/or switches) through which a specific packet is transmitted from a source FEP to a destination FEP. Packets can be routed along multiple paths or planes between two FEPs.
	Fabric plane or plane	A set of FEPs connected with links and switches (optionally) allowing any FEP to communicate with other FEPs in the same set. Communication between FEPs on different fabric planes is beyond the scope of this specification.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Class	Term	Description
	Folded Clos	Type of multistage network topology composed of crossbar switches. Also known as a fat tree.
	Frame	A unit of data transmission on an Ethernet network using layer 2 encapsulation, starting with the MAC address and ending with the CRC.
	Initiator	The FEP that initiates the creation of a PDC for RUD and ROD modes.
	Link	A physical connection between two ports.
	Local network	A network interconnecting a limited number of accelerators, typically with very high bandwidth and optimized for small read and write operations. Also referred to as ‘scale-up’.
	Lossless network	A network in which all network devices (switches and endpoints) avoid packet loss due to buffer overflow by transmitting a packet only when it is known that the link peer has available buffers to receive and store the packet. Packet loss avoidance is applied to all links in the network.
	Message	One or more packets with the same message ID. A message is split into a spatially (with respect to memory addressing) ordered set of packets at the source. One or more messages (and supporting packets) make up a transaction.
	Negative acknowledgement packet (NACK)	A packet used by the UET to implement reliability. NACKs are transmitted by the destination PDC to the source PDC to provide an explicit indication of packet loss.
	Node	A computing device with one or more FEPs. A node may contain one or more CPUs and/or one or more accelerators.
	Packet	An IPv4 or IPv6 datagram or UFH packet transferred across the network. Packets are routed through the network along paths. Packets of the same PDC traversing different paths may arrive in a different order than how they were sent.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Class	Term	Description
	Packet delivery context (PDC)	A logical unidirectional and often transient entity (established by the transport) between two FEPs, which exists at the initiator and at the target FEP to control the successful transmission of packets.
	Packet delivery sublayer (PDS)	The part of the Ultra Ethernet Transport (UET) protocol responsible for delivering packets with desired ordering and reliability over IP/Ethernet networks.
	Packet window	The maximum number of unacknowledged bytes that can be outstanding on a congestion control context between two FEPs.
	Port	A single Media Access Control (MAC) as defined by IEEE 802.3 and any UE extensions.
	Relative addressing	A mode of addressing a resource within the parallel job model to ensure scalability to the largest process counts. Includes four complementary parts of a destination address: (1) an FA identifying a FEP, (2) a JobID identifying a job in a cluster uniquely, (3) a local PIDonFEP ranging from 0 to P-1 identifying one of the P OSI PIDs that are associated with a destination FEP (that are part of the job), and (4) a Resource Index.
	Semantic sublayer (SES)	The part of the Ultra Ethernet Transport (UET) protocol that implements the OFI libfabric API.
	Switch	A device with two or more ports that forwards packets received based on the packet's FA or other information/state.
	Target	The FEP that responds to the initiation of a PDC from an initiator. The target does not initiate messages on the PDC and only responds to messages from the initiator.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Class	Term	Description
	Traffic class (TC)	A classification of network traffic that identifies mechanisms and resources within endpoints and switches used for the isolated transmission of packets (e.g., queues, buffers, schedulers). Traffic classes are distinct from one another and can be prioritized between one another. Attributes and fields of received packets are used to identify a traffic class (e.g., Differentiated Services Code Point (DSCP) [3]).
	Transaction	One or more messages (and supporting packets) needed to implement the libfabric request and deliver the payload requested by the user.
	UE Transport Protocol (UET)	A method including protocol, packet formats, and FEP policies by which FEPs communicate.
	Unified Forwarding Header (UFH)	A compact encoding of the fabric address and key protocol fields that overlays the MAC SA and DA in the frame header.
Parallel Communication	Job	A job consists of one or more ranks.
	JobID	A unique identifier for a parallel job within a cluster. JobIDs are used for addressing and authorization purposes.
	Parallel job	A collection of processes running on a cluster that belongs to the same user and can communicate.
	Parallel job model	A mode of cluster operation that involves MPI/*CCL or SHMEM. The parallel job execution is characterized by a “run to completion” model (in which checkpoint/restart is a simple reliability technique).
	PIDonFEP	An identifier of a process associated with a FEP numbered from 0 to P-1. If each FEP has the same number of processes associated with it, each endpoint can easily compute the PIDonFEP of a particular RankID.
	Rank	A process that cooperates to compute a particular workload. A job can spawn multiple ranks on

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Class	Term	Description
		each OSI, and an OSI can host ranks of multiple jobs.
	RankID	An assigned rank number per job starting from 0.
Client-Server Communication	Client	A software entity on a node that communicates to a server through FEPs.
	Client/server job model	A mode of cluster operation in which clients connect to servers, e.g., storage or Function as a Service (FaaS). The client/server execution typically runs the server for an indefinite time, serving an indefinite number of clients (often with complex reliability and availability guarantees).
	Discovery	The process of finding servers using static fabric addresses or a discovery service such as DNS or LLDP.
	Resource Index (RI)	Identifies resources within a process such as a service, library, or other entity (e.g., MPI vs. *CCL).
	Server	A software entity on a node that provides a service to one or more clients.
	Server PIDonFEP	In combination with a Resource Index, a server PIDonFEP identifies a service available on a specific FEP. The same FEP may be used by multiple servers (on a single OSI), and a single server may offer services through multiple PIDonFEPs in combination with resource indices on multiple FEPs.
Security Threat Model	Attacker	An entity that wants to extract information from a communication or modify communicated data.
	Ciphertext	The packet data containing the encrypted plaintext that is sent on the wire between sender and receiver.
	Information	Data or properties of the data exchanged between two participants that would allow the attacker to take or cause an adverse action. Examples include cryptographic keys, decisions of the FEP, etc.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Class	Term	Description
	In-scope threat	Threat that is explicitly addressed by TSS and that has defined mitigations.
	Intermediary/switch	An entity that routes or forwards packets to a receiver.
	Out-of-scope threat	A threat that is not considered or addressed in this specification.
	Plaintext	The original data that needs to be encrypted by the sender before transmission and the resulting data after it is decrypted by the receiver.
	Protocol secrets	UET secrets that are protected from users of the protocol and/or attackers to maintain the trusted connection.
	Side channel	A method for an attacker to extract information without the knowledge of the sender or receiver.
	Threat	Damage or danger that could expose protocol secrets, allow the leaking of packet data, or degrade the integrity of the network.
	Threat mitigation	How TSS specifically addresses the possible threat.
	Trusted entity	The portion of the FEP entrusted to handle key material and perform cryptographic functions.
	Privileged entity	A portion of the FEP and kernel driver that is responsible for assigning transport-critical information such as JobID and security context.
	User entity	User application that uses a UET transport service.
Transport Security	Additional authentication data (AAD)	The additional data authenticated with the ciphertext. Used in conjunction with an AEAD cipher.
	Association number (AN)	Selects between one of the two active keys (SDK) for an SD. The active AN is carried in the TSS header allowing the use of both the old and new keys until the key rotation is complete. The AN is used for key rotation.
	Authenticated Encryption with Associated Data (AEAD)	A symmetric cryptographic scheme that combines confidentiality and authenticity.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Class	Term	Description
	Advanced Encryption Standard (AES)	A symmetric encryption algorithm used with AES-GCM.
	Cryptographic key	Either a truly random binary string of a length specified by the cryptographic algorithm or a pseudorandom binary string of the specified length that is computationally indistinguishable from one selected uniformly at random from the set of all binary strings of that length. This definition is from NIST SP800-108.
	Differential power analysis (DPA)	A side-channel attack that involves statistically analyzing power consumption measurements from a cryptosystem.
	Epoch	A key epoch is a subinterval between changes in security association. Key epochs are managed by the SDME to ensure the IV is unique and MAY be used to automatically generate new SDK by using a KDF.
	Initial secure domain key (SDKi)	A symmetric key from the SDKDB. This can be used directly as the SDK or optionally used in a KDF to form the SDK.
	Galois/Counter Mode (GCM)	A mode of operation for symmetric-key cryptographic block ciphers.
	Galois message authentication code (GMAC)	An authentication algorithm used with AES-GCM.
	Initialization vector (IV)	The initial block/condition of the block cipher.
	Integrity check value (ICV)	The checksum calculated by the sender over the AAD and ciphertext and sent with the packet. The receiver uses the ICV to validate the cryptographic integrity of the packet.
	IPv4SIP	IPv4 (RFC 791) source address.
	IPv6SIP	IPv6 (RFC 8200) source address.
	IPv4DIP	IPv4 (RFC 791) destination address.
	IPv6DIP	IPv6 (RFC 8200) destination address.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Class	Term	Description
	Key derivation function (KDF)	A process to derive a new symmetric key (Ko) from an input key (Ki) using a pseudo random function (PRF). A KDF uses a derivation key (Ki), label and context input parameters to generate an output key (Ko). In pseudo code form, (Ko = KDF(Ki, label=x, context=y)).
	SDK database (SDKDB)	An SD database indexed by SDI and used to store/retrieve security parameters for an SD. The SDI, AN, and possibly SSI are used to obtain the key for the packet.
	Secure domain (SD)	A set of FEPs that communicate using the security services (confidentiality and encryption) of TSS. Members of an SD share a common set of security parameters (keys, confidentiality offset, etc.). The SD is represented in the packet as using an SDI.
	Secure domain management entity (SDME)	Abstract secure domain administrator.
	Secure domain identifier (SDI)	An identifier of the SD, carried within the packet. The SDI in conjunction with the association number (AN) identifies the SDKDB key slot. This is used for rekeying.
	Secure domain key (SDK)	A symmetric key used for packet AEAD cipher or KDF operation. SDK is a cryptographic key per NIST security definition.
	Secure source identifier (SSI)	A unique identifier of the source of the packets. This identifier can be explicitly carried within the packet (SSI) or the source IP header address.
	Timestamp counter (TSC)	A monotonic counter (different for each packet sent).
	Transport Security Sublayer (TSS)	The UE Transport Security Sublayer defined by this specification
Libfabric mapping	Application binary interface (ABI)	An interface between two binary program modules (e.g., user program and a library or operating system). The interface defines how data structures and computational

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Class	Term	Description
		routines are accessed in a low-level hardware-dependent format.
	Application programming interface (API)	A type of software interface. An API offers a service to other pieces of software and provides a way for two or more computer programs or components to communicate.
	Collective Communications Library (*CCL)	One of any number of Collective Communication Libraries that implement collective operations common in parallel computing (e.g., Broadcast, AllReduce, AllGather, etc). An accelerator vendor commonly provides a proprietary *CCL implementation that supports the vendor's accelerator capabilities.
	Kernel mode driver (KMD)	A component of the operating system that runs in the privileged kernel mode, allowing code to access the system memory and hardware directly.
	Libfabric address vector (AV)	A mapping of higher-level addresses, which may be more natural for an application to use, into fabric-specific addresses. See libfabric <https://ofiwg.github.io/libfabric/v1.20.1/man/fi_av.3.html>.
	Libfabric completion queue (CQ)	High-performance event queues used to report the completion of data transfer operations. See libfabric <https://ofiwg.github.io/libfabric/v1.20.1/man/fi_cq.3.html>.
	Libfabric endpoint (EP)	A communication endpoint using the libfabric API that can listen for connection requests and perform data transfers. Endpoints are configured with specific communication capabilities and data transfer interfaces. See libfabric <https://ofiwg.github.io/libfabric/v1.20.1/man/fi_endpoint.3.html>.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Class	Term	Description
	Libfabric event queue (EQ)	A queue used to collect and report the completion of asynchronous operations and events. Event queues report events that are not directly associated with data transfer operations. See libfabric <https://ofiwg.github.io/libfabric/v1.20.1/man/fi_eq.3.html>.
	Message passing interface (MPI)	A standardized and portable message-passing communications library interface designed for parallel computing (e.g., MPI-4.1 as defined by the MPI Forum).
	Partitioned global address space (PGAS)	A parallel programming model that uses a logically partitioned global memory address space to enhance performance and efficiency in distributed systems.
	Shared memory / symmetric hierarchical memory parallel programming library (SHMEM)	A communications library for distributed memory environments focusing on one-sided communication allowing applications to read and write each other's memory.
In-Network Collectives (INC)	Collective operation (short: Collective)	A collective operation is a group communication operation as defined in MPI. Examples are reduce, allreduce, or barrier. Collective operations generally execute on predefined groups of FEPs that are arranged into communicators. To trigger the execution of a collective operation, all ranks of a communicator must call the same operation with the same communicator and matching arguments.
	Communicator	An ordered set of ranks of a job that participate in collective operations that are initiated in a common shared order. Each communicator has a specific reproducibility mode defined.
	Collective sequence number (CSN)	A sequence number for collective operations. CSNs must be consistent across all ranks in a communicator. The CSN is increased by an invocation of a collective operation.
	Communicator ID (CID)	A 20-bit number that identifies a communicator within a fabric.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Class	Term	Description
		Communicators are identified by IDs. The IM allocates a new ID to each created communicator. Note that two identical communicators with different reproducibility modes will be assigned two different CIDs.
	In-place collective operation	A collective operation where some ranks use the same buffer for both input and output data.
	INC function	Aggregation and distribution capability required in an INC switch to perform collective offload.
	INC manager (IM)	A software entity that discovers INC switch capability and manages INC switch resources for a UE fabric. It is responsible for configuration of the resources allocated to communicators for collective offload.
	INC switch	A UE switch with the capability to support INC.
	Switch endpoint (SEP)	The logical entity inside an INC switch to provide connections to FEPs and other SEPs.
Credit-based flow control	Best-effort VC	A virtual channel configured to not use the CBFC credit mechanisms.
	CBFC message	A link-level message between a CBFC sender and receiver. Different CBFC messages are formatted as either a CtlOS or a fully formed Ethernet packet.
	Cell	A unit of storage in a data buffer. Packets are usually divided into one or more cells for storage. The number of cells in a receiver's data buffer is directly related to the number of credits that a sender can use.
	Control ordered set	An 8-byte message format used by CBFC and LLR to pass information between link partners.
	Credit	A token representing a unit of data storage at the receiver. Credits allow a sender to transmit data packets and are consumed at the sender as packets are transmitted. Credits are returned to the sender by the receiver when the receiver has released buffer resources that

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Class	Term	Description
		can be utilized for new packet arrivals.
	Lossless VC	A virtual channel configured to require CBFC credits for transmission of packets with guaranteed buffering available at the receiver. A lossless virtual channel can be flow controlled separately from other lossless VCs on a single physical link.
	Receiver	The link partner function that receives packet data and transmits CBFC credits.
	Sender	The link partner function that transmits packet data and receives CBFC credits.
	Virtual channel (VC)	An entity that contains a subset of a port's traffic with similar traffic characteristics, dedicated buffering, and flow control management.
Link Layer Discovery	Link Layer Discovery Protocol (LLDP)	A media-independent protocol, standardized by IEEE as IEEE Std 802.1AB, capable of running on all IEEE 802® LAN stations allowing ports to learn the connectivity and management information from adjacent stations.
	Company ID (CID)	IEEE Std 802: A unique 24-bit identifier assigned by the IEEE to identify an organization. A CID cannot be used to generate universally unique MAC addresses.
Management	gNMI	A standard gRPC-based network management interface defined by the OpenConfig project and used to retrieve and modify network device configuration as well as provide control and generation of telemetry.
	gNOI	A standard gRPC-based network operations interface defined by the OpenConfig project and used for executing operational commands on network devices.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Class	Term	Description
	gRPC	A high-performance open-source framework for universal remote procedure call (RPC).
	Yet Another Next Generation (YANG)	A data modeling language for the definition of data sent over network management protocols such as the NETCONF and RESTCONF.

1.2.3 Formatting

Figures are used to define protocol headers and protocol sequence exchanges. The conventions for these figures are shown in Figure 1-2, Figure 1-3, and Figure 1-4. Wherever diagrams and figures inadvertently contradict the textual description, the text always takes precedence.

1.2.3.1 Header Format Figures

Figure 1-2 is an example of a full-header stack as illustrated in other sections of the specification. These full-header stacks do not show all the details of each header in each layer, but rather identify the important fields needed for parsing the headers and finding the next layer of the stack. The layers of the header stack are shown on the left and are differentiated by color.

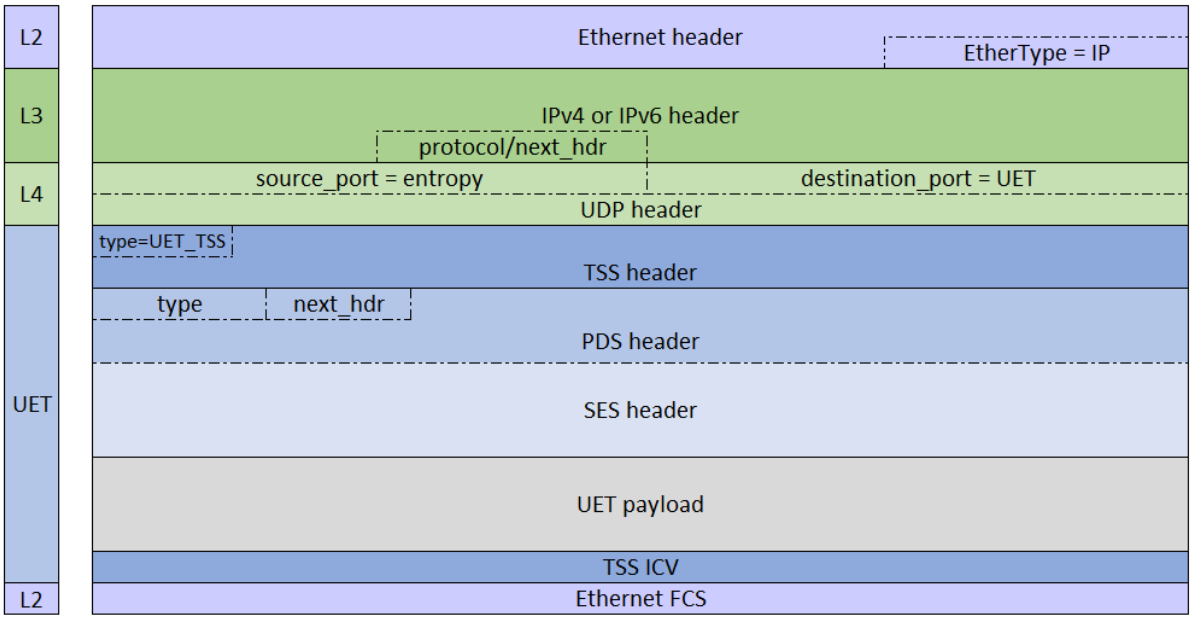


Figure 1-2 - Example Full Header Format

Figure 1-3 is an example of an individual detailed header figure used in other parts of the specification. Bytes are labeled across the top and down the left side. The least significant bit (LSB) of a byte is the first bit on the left, and the most significant bit (MSB) of a byte is on the right. Header fields are labeled below the byte labels at the top and to the right of the byte offset labels. Header field widths are based

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

on actual bit sizes. Reserved fields are to be ignored upon receipt and transmitted as zeros. Each individual header is a stand-alone figure and shows the header starting at a byte offset of zero. The actual offset of the header within a receive packet depends upon the specific format of the previous headers, which are not shown in the stand-alone figure.

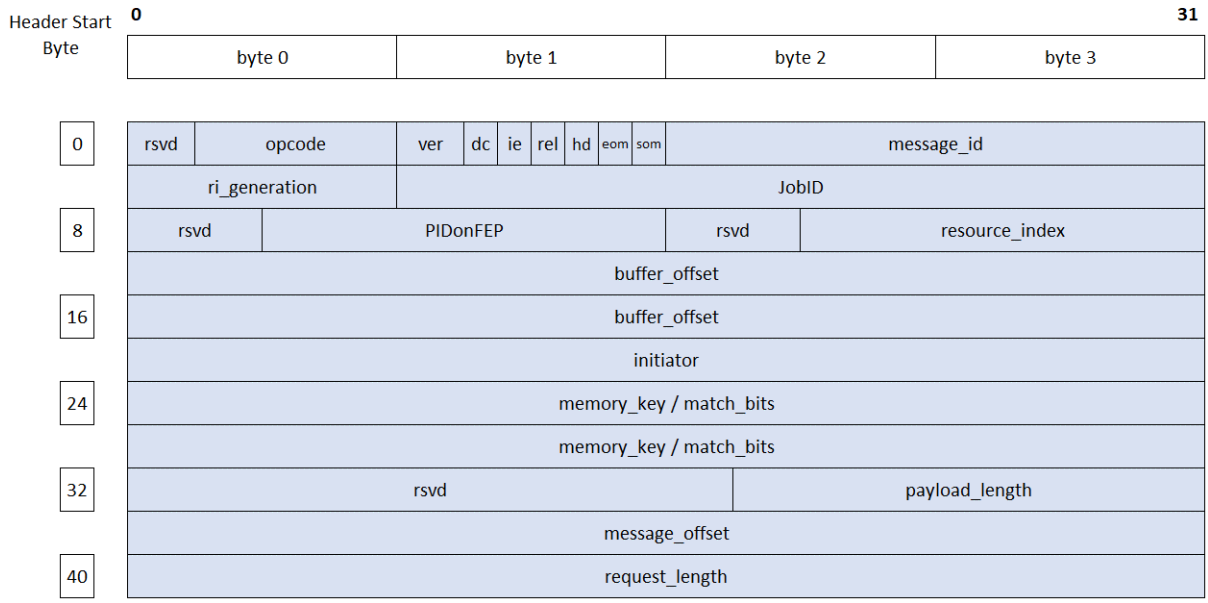


Figure 1-3 - Example Individual Header Format

A description of the header fields is provided in a table following the detailed header figure.

1.2.3.2 Sequence Diagrams

Figure 1-4 shows an example of a sequence diagram used in this specification. Sequence diagrams are used to illustrate the timeline for specific information exchanges between two entities and across functional layers within entities. The event timeline flows from top to bottom. The sequence diagrams do not provide a normative depiction of the complete set of information exchanged between entities, but rather are used to describe a particular instance of a communication scenario. The example shows a message provided to the UET semantics layer by an external entity (e.g., a libfabric provider), and how it is broken into packets and passed to the PDS layer for transmission over the wire to a remote target. Some of the important UET header fields are shown above the arrows indicating packets transmitted on the wire. Actions and events that occur during the information exchange are highlighted using dashed arrows and supporting text.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

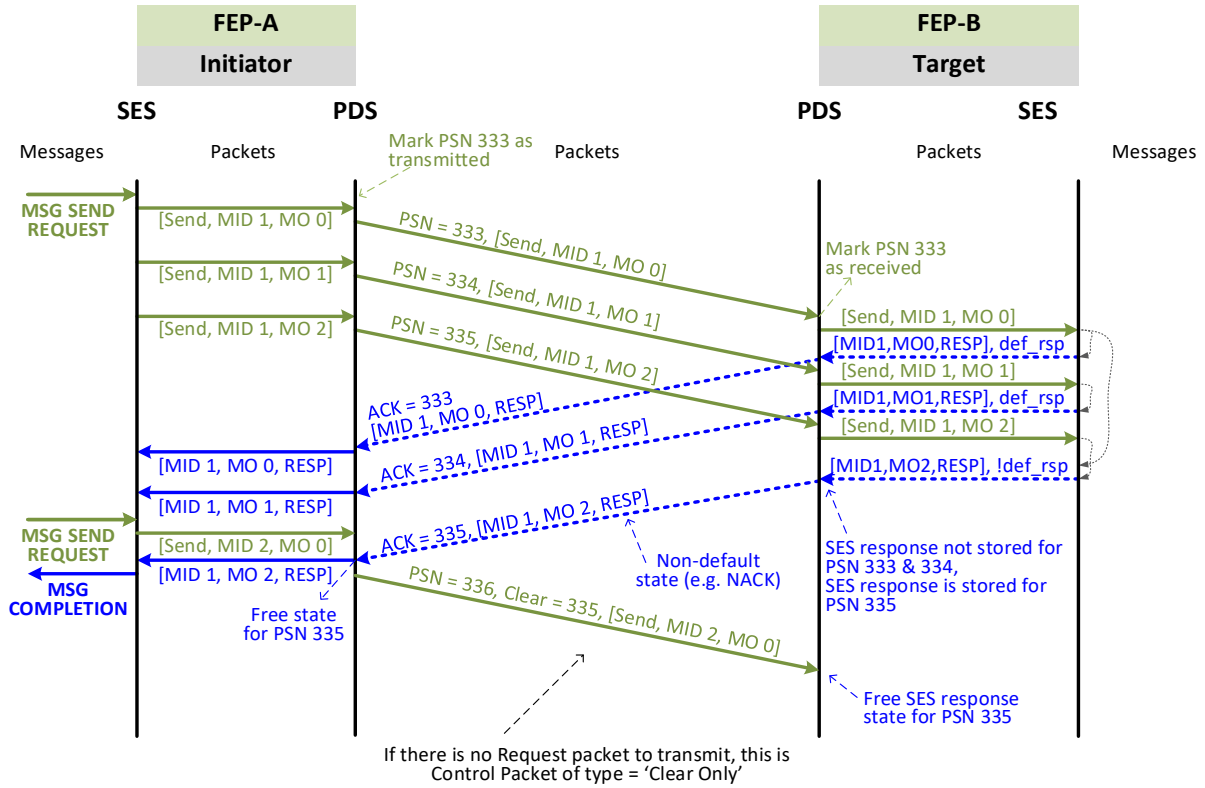


Figure 1-4 - Example Sequence Diagram Figure

1.2.4 References

The UE specification makes both normative and informative references. Normative references are required to assure interoperability among UE components. Informative references are intended to provide additional background and further understanding of the UE operating environment. Each chapter of the UE specification may provide a list of normative and informative references.

The following normative references are used in this introductory material:

[1] IETF RFC 2119, "Key words for use in RFCs to Indicate Requirement Levels," 1997. [Online]. Available: https://www.rfc-editor.org/rfc/rfc2119.

[2] IETF RFC 8174, "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words," 2017. [Online]. Available: https://www.rfc-editor.org/rfc/rfc8174.

[3] IETF RFC 2474, "Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers," 1998. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc2474.

[4] IETF BCP14, "IETF Best Pratice 14," 2023. [Online]. Available: https://datatracker.ietf.org/doc/bcp14/.

1.3 System View and Nomenclature

The field of AI and HPC is evolving at a very fast pace. AI models are changing at an even faster pace. This has created a need for fine-tuned systems that scale horizontally and vertically for various AI and

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

HPC workloads. Horizontal scaling involves adding more endpoints and/or fabric switches. Vertical scaling involves endpoint expansion by adding more processing, memory, and/or storage.

Algorithm and model evolution is outpacing the hardware (mainly on memory, storage, and network aspects). An optimal system balances a triangle of compute, network (i.e., fabric), and the associated data (i.e., model parameters and training data). The UE specification explicitly addresses network technology and implicitly addresses other elements.

The UE specification addresses distributed workloads, whether they are AI or HPC, and naturally inherits some nomenclature from parallel computing. Figure 1-5 provides a system overview of the parallel computing components, terms, and concepts addressed by the UE specification. The following text provides a high-level overview and introduction of the UE environment and associated nomenclature. Technical details of the procedures, protocols and operation of the components specified in this standard are provided in subsequent normative chapters.

UE is specified to operate within a cluster, which includes an interconnection of nodes through a fabric. A port implements a single Media Access Control (MAC) as defined by IEEE Std 802.3, optionally including any UE specific extensions, as required by the UET profile it complies with. A link connects two ports. A fabric interface (FI) is a physical entity that provides one or more ports and exposes one or more fabric endpoints (FEPs) to one or more operating system instances (OSIs). A fabric address (FA) is either an IPv4 or IPv6 address, and a FEP is a logically addressable entity assigned a single FA. The UE transport protocol terminates at a FEP, optionally including a security context. Specifically, a FEP can only be used by a single OSI, and a node may have one or more FEPs. Each OSI may use one or more FIs with one or more FEPs to connect to a fabric.

A switch has two or more ports and is a packet forwarding device that is part of the fabric. Packets are forwarded along a path based on forwarding information that includes the packet's FA, other header fields (e.g., the UDP source port used as an entropy value), and switch state. Any two packets with the same path forwarding information are expected to take the same path through the fabric. A fabric plane is a set of FEPs connected with links and optionally switches allowing any FEP to communicate with other FEPs in the same set. Communication between FEPs across different fabric planes is beyond the scope of this specification. A path exists within a fabric plane, but not across fabric planes. A fabric is made up of one or more fabric planes.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

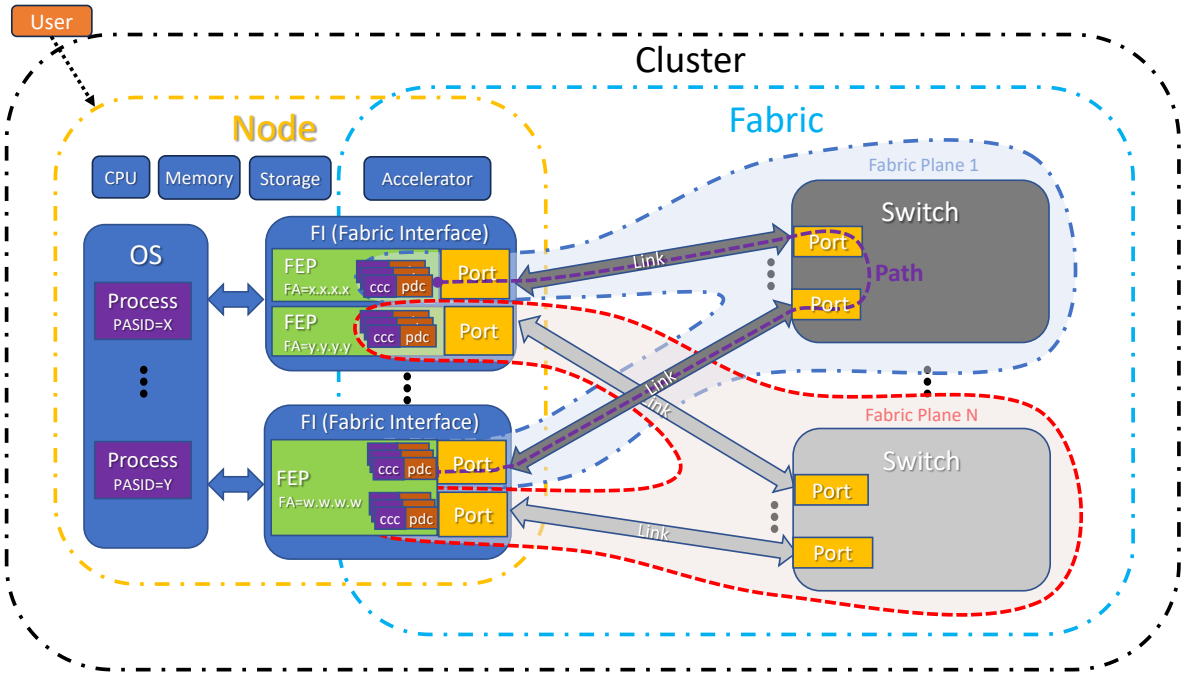


Figure 1-5 - System Overview

A node is a computing device with one or more FEPs. A cluster is a set of such nodes connected by a fabric (Note that the simplified diagram in Figure 1-5 is for illustration purposes and shows only a single node in the cluster. A typical UE deployment can have hundreds of thousands of nodes). An accelerator is a computing module or device designed for the efficient execution of specific functions. A CPU is a generic processor for arbitrary computation. CPUs and accelerators have memory attached, and a FEP has access to that memory through virtual addressing. A node may have local storage and contain one or more CPUs and/or one or more accelerators.

A user is an entity with access privileges to nodes of a cluster. A user can execute processes. A process is an instance of a program executing on an OSI owned by a specific user and having a private virtual address space (VAS) in the memory. A process is identified by a process ID (PID) unique to the OSI it runs in. A process address space ID (PASID) is a unique identifier for a VAS within each OSI.

Clusters can be used in two fundamentally different ways that can potentially co-exist:

1. Executing in a parallel job model (e.g., MPI/*CCL ,or SHMEM).
2. Executing in a client/server model in which clients can connect to servers (e.g., storage or a Function as a Service (FaaS)).

Each packet carries an identifier indicating which model it is participating in. The parallel job execution is characterized by a “run to completion” model, while the client/server execution typically runs the server for an indefinite time, serving an indefinite number of clients (often with complex reliability and availability guarantees).

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Two FEPs require IP-level connectivity to communicate. Multi-ported FEPs are supported in a UE network provided there is one FA (i.e., IP address) associated with any FEP. The FA associated with the FEP is used as the source for all packets sent from the FEP, or as the destination for all packets sent to the FEP.

A FEP can have multiple ports (e.g., scenario A in Figure 1-6). UE does not mandate how those ports are used. Many multi-ported configurations are possible, including: a link aggregation (LAG) with members terminating on the same switch (e.g., scenario D in Figure 1-6), a multi-switch LAG with members terminating on different switches (e.g., scenario E in Figure 1-6), or IP-level multipath connectivity to reach the FEP's individual address via multiple ports or in an active-standby configuration. Other scenarios for supporting multiple ports on a common fabric interface are possible and are implementation-dependent (e.g., Scenarios B, C, and F in Figure 1-6). Scenario F of Figure 1-6 shows an embedded switch on a fabric interface card.

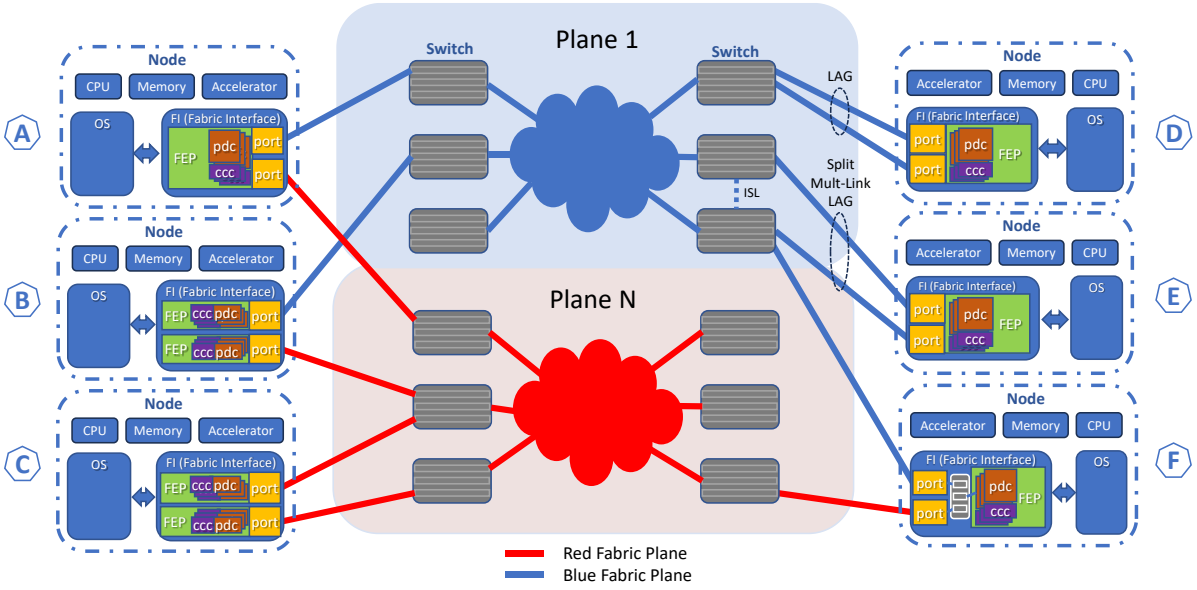


Figure 1-6 - Multi-plane Networks and Multi-port FEPs

In multi-ported FEP architectures, the UET congestion management sublayer chooses the entropy value for each packet. The UET does not mandate any specific port-selection method for multi-ported hosts; it can vary between implementations.

Informative Text:
In multi-ported configurations, the network is expected to provide a mechanism that allows a FEP to detect if a destination IP address (e.g., the FA of a peer FEP) is unreachable on any given plane and, if so, avoid using that plane for that destination. Possible methods to do this include using IP routing protocols or ICMP unreachable messages to communicate unreachability to a FEP., The UET, however, does not mandate any specific technique. In the future, mechanisms might be defined in the UET to allow the determination of reachability.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

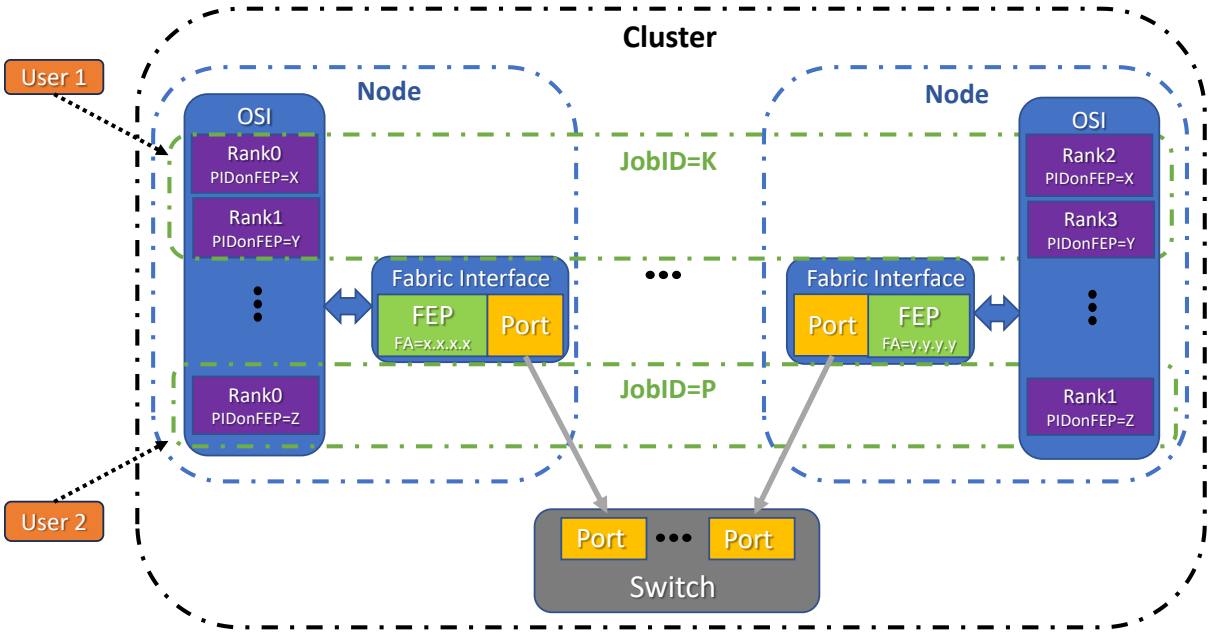


Figure 1-7 - Parallel Job Model

Two different computing models are differentiated in terms of addressing: parallel job and client/server. A single FEP may be designed to support the traffic type of both or just one computing model. Both modes can operate simultaneously over a single FEP. Figure 1-7 and Figure 1-8 are overviews of the two different computing models within a cluster.

A parallel job is a collection of processes running and communicating on a cluster that belongs to the same user. A job is often started collectively and is uniquely identified by a JobID within a cluster. JobIDs are used for addressing and authorization purposes. Jobs consist of one or more ranks, which are processes that cooperate to compute a particular workload. A job can spawn multiple ranks on each OSI, and an OSI can host ranks of multiple jobs. If a job globally contains R processes/ranks, then they are numbered with RankIDs from 0 to R-1. In the parallel job model, the local PIDonFEP addressing ranges from 0 to P-1 where P processes are associated with a FEP for a given job. If each FEP has the same number of processes associated with it, each endpoint can easily compute the PIDonFEP of a particular RankID. As an example, assume a job runs on R ranks across N nodes, where each node has F FEPs. The RankID range is 0 to R-1, the ranks per node is R/N, and the ranks per FEP is R/N/F. In this case, P is R/N/F and the PIDonFEP range is 0 to P-1, or 0 to R/N/F-1.

A server is a software entity on a node that provides a service to one or more clients. Clients and servers communicate through FEPs. A server PIDonFEP identifies a service available on a specific FEP, and a Resource Index identifies a resource within that service. The same FEP may be used by multiple servers (on a single OSI), and a single server may offer services through multiple PIDonFEPs on a FEP. A client/server JobID is conceptually the same as an “N to 1” connection in that it is used for authorization

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

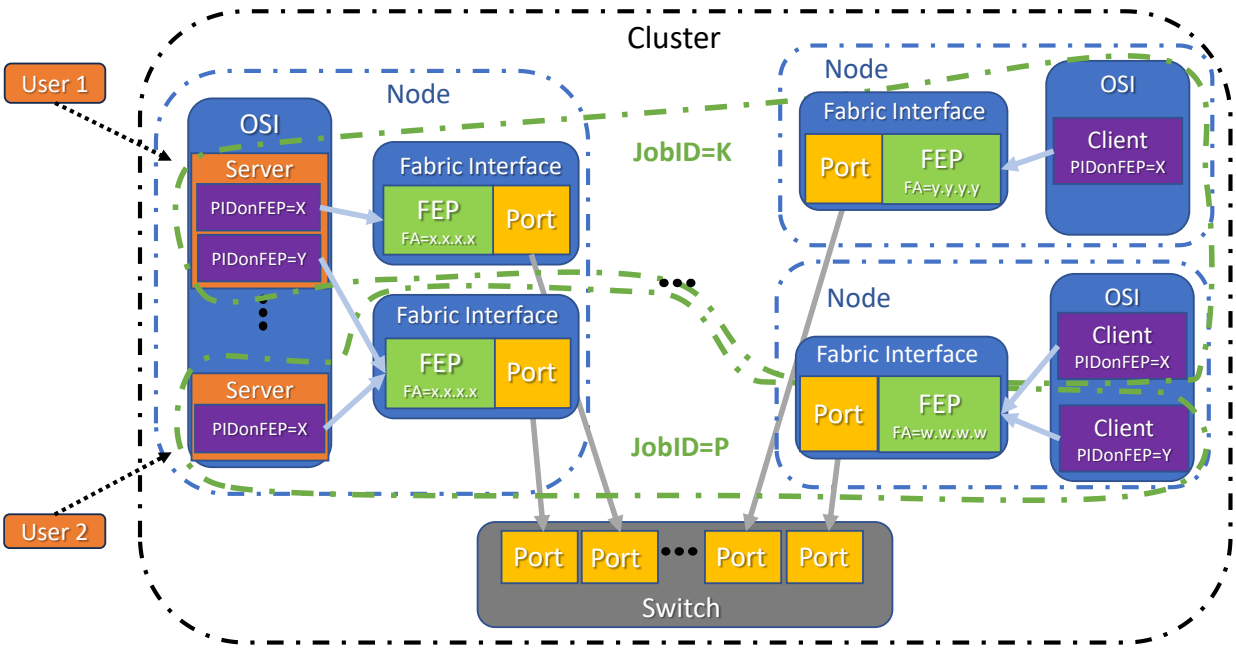


Figure 1-8 - Client/Server Job Model

of a client toward a server, but it may be ephemeral in that it exists only while traffic is active. The combination of server PIDonFEP and Resource Index is conceptually the same as a TCP/UDP port.

UET servers can use well-known server PIDonFEP values and a set of resource indices for specific services, and UET clients may use either preconfigured static server FAs or various discovery mechanisms to resolve a server's hostname or alias to an FA, such as DNS or a 'hosts' file, allowing clients to establish communications with a server. Additionally, UET clients and servers can choose to utilize non-UET discovery methods such as first establishing a TCP/IP connection and subsequently exchanging UET capabilities, addresses, and identifiers.

UET has relative, absolute, and direct memory endpoint addressing modes (see Figure 1-9). The relative addressing mode provides consecutive addressing within the parallel job model to ensure scalability to largest process counts. Absolute addressing is used in the client/server model. Direct memory addressing is available only for the Unified Local profile.

In the relative addressing mode, UET defines four complementary parts of a destination address:

1. An FA identifying a FEP.
2. A JobID uniquely identifying a job in a cluster.
3. A PIDonFEP ranging from 0 to P-1.
4. A Resource Index (RI) identifying a service, library, or other entity within the target process (e.g., MPI vs. *CCL).

When a packet arrives at a destination FEP, the target PASID can be determined based on a combination of JobID and PIDonFEP.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

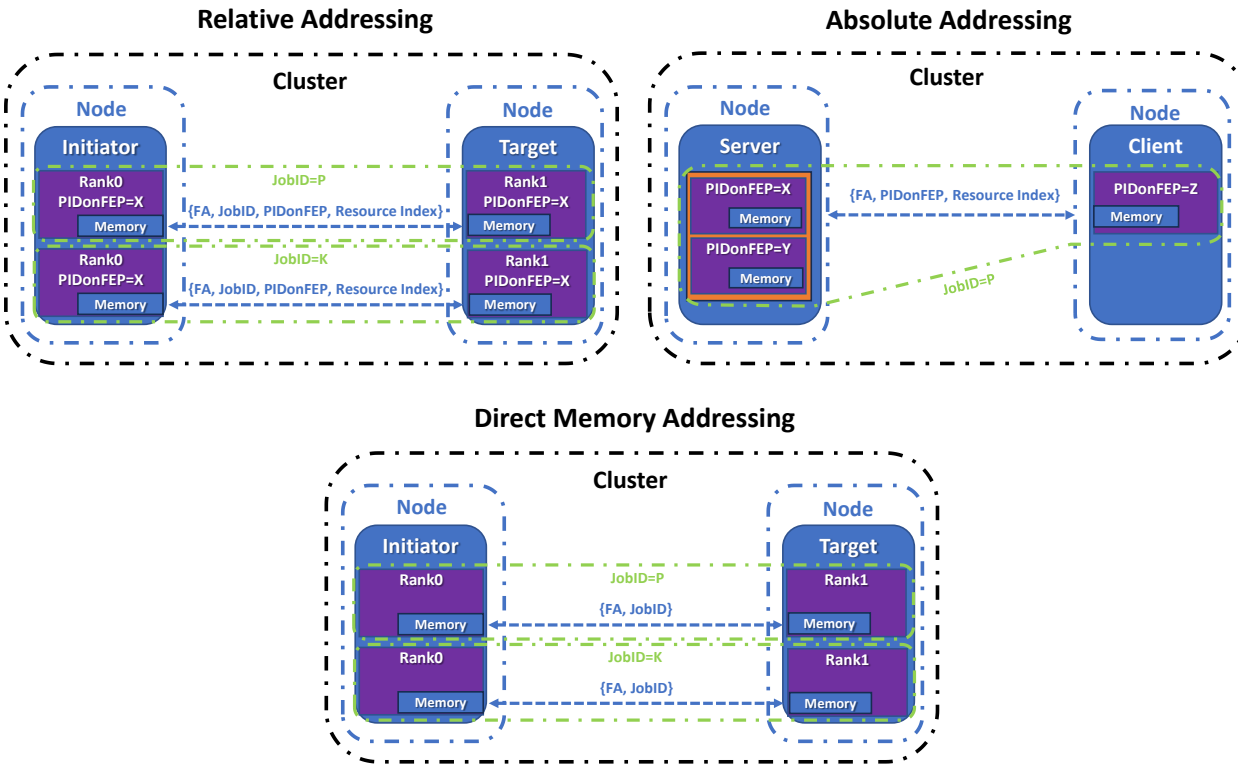


Figure 1-9 - Addressing Modes

Informative Text:

Assume a job contains K endpoints and P processes per endpoint. The relative addressing scheme allows for a table of size K-1 at the source to translate to a destination FA and then, at this destination, a table of size P to translate to the target process at the destination. In a system that supports multiple jobs per endpoint, the JobID disambiguates different jobs at the destination to find the right PIDonFEP table of size P. This avoids O(K*P) table entries at the source that would be required if one flat address space was used. This approach replaces the O(K*P) table entries with O(K+P) entries. K is expected to be in the tens of thousands, P in the hundreds.

In the absolute addressing mode, UE defines three complementary parts of an address:

1. An FA identifying a FEP.
2. A PIDonFEP identifying one of the OSI PIDs that are associated with a destination FEP.
3. A Resource Index (RI) identifying a service or other entity within the target process.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

The Resource Index is useful to address a specific subroutine or function in a server (e.g., running a FaaS or rPC service). When a packet arrives at a destination FEP, the target PASID is determined based on the PIDonFEP. The JobID is not part of addressing but is used for authorization.

Direct memory addressing mode is used only for the Unified Local profile. The direct memory address has two complementary parts:

1. An FA identifying a FEP.
2. A JobID uniquely identifying a job in a cluster.

With direct memory addressing, a single network visible address space is associated with the entire JobID at the target FEP. The JobID is used to provide isolation. System-level software is responsible for managing addresses in this addressing mode.

Messages represent a single communication transaction in the UE network. Figure 1-10 shows the main concepts behind message communication delivery. A transaction is created by a process calling into the UET libfabric provider, which in turn calls into the semantics layer. The semantic sublayer creates various messages to complete the transaction. A message and its associated buffer are split into a spatially ordered set of packets**,** with respect to memory addressing, at the source FEP. The packets are sent and routed, potentially out of order, through the network along paths. A message may initially involve a rendezvous protocol step as discussed in the UET semantics specification.

The source of such a message is called the initiator, and the destination is called the target. Each packet has a source FEP and a destination FEP. A path is an ordered set of links (between nodes and/or switches) through which two FEPs communicate. Without considering packet loss in the fabric, packets of the same traffic class (TC) sent along the same path are always received at the destination FEP in the order they were sent at the source. When packets are routed along multiple paths between two FEPs, no ordered arrival among different paths is guaranteed. UET expects that switches will, in the absence of routing changes, deliver two packets from the same PDC, with the same entropy value and traffic class, along the same path (see Section 1.5.3.5). The UE transport protocol (UET) defines protocols, packet formats, and FEP policies by which FEPs communicate. A FEP can be designed to support either parallel, client/server, or both traffic types.

A packet delivery context (PDC) is a unidirectional logical and often transient entity (defined by the transport) between two FEPs, that exists at the initiator FEP and at the target FEP to control the successful transmission of packets. The packet delivery sublayer (PDS) creates and uses PDCs to provide the requested ordering and reliability delivery mode. Reliability is implemented using acknowledgment packets (i.e., ACKs and NACKs), which the destination transmits to the source to indicate successful end-to-end reception. The congestion management sublayer (CMS) controls the number of bytes that a source can transmit to a destination at any given time. A packet window is the maximum number of unacknowledged bytes that can be outstanding on a CCC. Multiple processes of the same job can share a single PDC, and a single process can use multiple PDCs to communicate with another process. Between two FEPs, there can be one or more PDCs, and each packet identifies an associated PDC.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

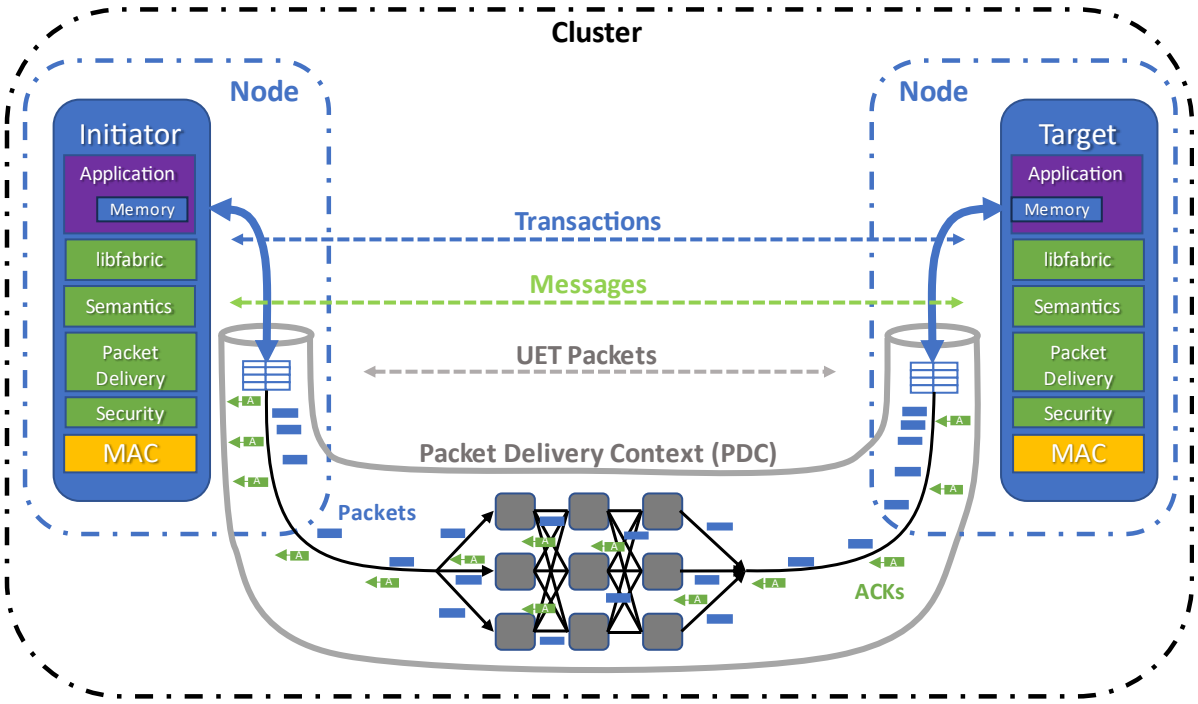


Figure 1-10 - Transport Data Delivery and Packet Delivery Contexts

1.3.1 Workloads [Informative]

The information in this subclause is background material and represents the state of the art for workloads as of the publication of the UE Specification v1.0. UE targets the following four workloads:

1. AI training (AIT)
2. AI inference (AII)
3. High-performance computing (HPC)
4. Client/server (e.g., storage traffic)

1.3.1.1 AI Training Workloads

AI training workloads were originally characterized by “3D parallelism,” where the communication pattern could be expressed as a 3D torus if it were beneficial. The first dimension is data parallelism (DP), where the examples in a single minibatch are processed through multiple model replicas, and gradients or weights are synchronized with allreduce operations.

The second dimension is pipeline parallelism (PP), where each pipeline stage is a set of layers of the model, and communications of the activations in the forward direction and errors in the backward direction are performed between neighboring layers on different accelerators forming a pipeline with point-to-point communications.

The third dimension is operator parallelism (OP), which depends on the type of layer. For large language models (LLMs), the main layer operation is matrix multiplication. Thus, layers would implement a parallel matrix multiplication that can also be expressed with allreduce operations. “Mixture of Experts”

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

models often use expert parallelism (EP) that is similar to operator parallelism in the 3D parallelism view. EP bundles k (for typical k=16 to 256) models together and performs an alltoall(v) operation among them. The alltoall(v) operation is not always balanced. AI inference parallelism is very similar. It differs in that it does not consider data parallelism and usually uses very small batches. Thus, both job sizes and transmitted messages are generally smaller.

Later, additional parallelism dimensions, such as sequence and context parallelism were added and may lead to higher-dimensional communication structures.

As a workload example (circa 2023), the well-known GPT-3 model had 175 billion parameters in 96 transformer layers. Storing those parameters in FP16 required 350 GiB, which required multiple state-of-the-art accelerators available in 2023 (not accounting for stashed activations or other temporary values or copies during training). In this scenario, if there are six accelerators along the pipeline and four in the operator parallelism dimension, then there are 16 GPT decoder layers per four accelerators. On accelerators available in 2023, this would have taken approximately 160 msec compute time. The communication is then 50 MiB per layer along each dimension (pipeline and operator).

Assuming a target service level objective of 200 msec, then 40 msec are for pipeline communications and the ring allreduce (which sends data twice). Each accelerator now sends 50 MiB 16*2 times for allreduce and once along the pipeline dimension. Thus, it needs to communicate 1.7 GiB in 40 msec, requiring 41 GiB/s. For lower latency, one can scale the operator dimension and decrease the communication time. For a 10 msec service level objective, one would need approximately 150-200 GiB/s throughput.

AI training workloads follow a 3D parallelism scheme but in addition to the weights, each accelerator typically stores a “golden copy” of the weights as well as all activation outputs for its layers until it applies the gradients on the backward pass. In general, AI training workloads require extremely high (cheap) bandwidth and have low to moderate latency sensitivity. Typical message sizes are in the megabyte or larger regime.

1.3.1.2 AI Inference Workloads

AI inference workloads follow a 3D parallelism scheme but do not offer data parallelism, as each input sample is a user request. They may be batched, but usually this is used only to improve accelerator efficiency. AI inference workloads sometimes have stringent service level objective requirements to satisfy interactive usage patterns. In this case, batches are smaller, leading to smaller activations (pipeline messages) as well as operations (allreduce). These often remain in the kilobyte-size range.

Consider generative AI inference on GPT-3 as an example. In this case only one token is input instead of a full sequence in the GPT-3 example above. Thus, everything is approximately a sequence length smaller (2048 for GPT-3). In practice, generative inference systems often use beam search for improved quality. With a beam width of four, there is a total shrinking factor of 512. Unfortunately, the weight memory remains the same, leading to the same distribution (OP=4, PP=6). The computation now can be as small as 1-msec and would send 3.3 MB of data per accelerator. With a service level objective of 1.2 msec, a reduction would be performed in 0.2 msec, leading to a bandwidth requirement of

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

16 GB/s—but now with allreduce of size around 100 KB. If those were split across multiple planes, then message sizes would be in the single-digit kilobyte range. The bandwidth requirements grow with distributed KV caching! Inference requires lower latency than training in general.

1.3.1.3 HPC Workloads

HPC workloads fall into two categories: (1) low depth (LD), which is highly parallel and (2) high depth (HD), which has long dependency chains. HD workloads often have long and skinny-directed acyclic graphs (DAGs) leading to latency-sensitive execution. Many strong-scaling problems have this shape.

Consider weather forecasting as an example. In this case the computation must finish in a certain time. The algorithms run many iterations during the simulation, where the time step inversely depends on the resolution. More accurate higher resolution models lead to smaller timesteps. In practice, such models have high communication overheads (25-50%) with mostly small (single-packet) messages (2.5 kiB) in recurring patterns.

Many other workloads are LD and thus less latency-sensitive. Most weak-scaling workloads where one can adjust the local domain size to run well on a given system fall into this category. Some of those workloads may need high bandwidths (e.g., AI training) or high message rate (e.g., graph algorithms).

1.3.1.4 Client/Server (e.g., Storage Traffic)

Storage traffic serves data from storage servers to endpoints and is an example of client/server communication. The communication stacks often split larger requests into multiple smaller messages (e.g., some MiB or even some KiB). Those sizes can, to some extent, be tuned to the transport protocol. The data access patterns and sizes are user-dependent and are rarely controlled by the system administrator. Thus, random incast events (e.g., many customers accessing the same storage server) regularly occur.

1.3.1.5 Scale-up Workloads (aka Local Network Workloads)

Scale-up networking (also referred to here as local networking) is quickly gaining importance, as many workloads have two scaling regimes: one regime that requires low latency and high bandwidth, and a second regime that does not have such stringent requirements. Scale-up networks can be designed to offer higher bandwidth and lower latencies using shorter cables and distances. Wire efficiency is enhanced, trading off traditional network layering and scale for compressed L2/L3/L4 network headers.

Of course, scale-up networking domains are limited to dozens, maybe hundreds, of endpoints today and low thousands in the future. Global networking domains can reach hundreds of thousands to millions of endpoints.

AI training is a workload that highlights the importance of high-performance local networking. As described in section 1.3.1.1, training frameworks commonly employ 3D parallelism, with each dimension imposing distinct network requirements. In practice, users map DP and PP onto lower-performance global networks, while the most latency- and bandwidth-sensitive dimension OP (also referred to as tensor parallelism) is placed on fast local networks. For many training jobs, this separation of local and global networking domains is critical to achieving high overall performance.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

AI inference is often run only in scale-up networks because it does not require more than 100 endpoints.

HPC workloads are more diverse than AI training and AI inference. Some optimize for local versus global networks, but many assume a flat address space driven by MPI. Many HPC workloads are expected to optimize for scale-up networking once such architectures become prevalent in HPC.

1.4 Software

1.4.1 AI and HPC API Interface

UE is designed to support libibverbs and libfabric v2.0 APIs. UEC collaborates with the libfabric community to allow an endpoint to interact with AI frameworks and HPC workloads. Some UE optional features require support from network devices (e.g., switches) for advanced capabilities such as packet trimming, CSIG, and INC. To that end, the network operating system (NOS) requires extensions to support UE features.

UE does not currently address interactions across administrative domains.

1.4.2 Fabric Endpoint Software Stack

Figure 1-11 shows the software stack running on a FEP.

UE is designed to support existing AI training (AIT) and AI inference (AII) frameworks. These frameworks, like TensorFlow, PyTorch, JAX, and others, are expected to work seamlessly on top of a UE software stack. In other words, a goal of UE is to enable the migration of applications, relying on these

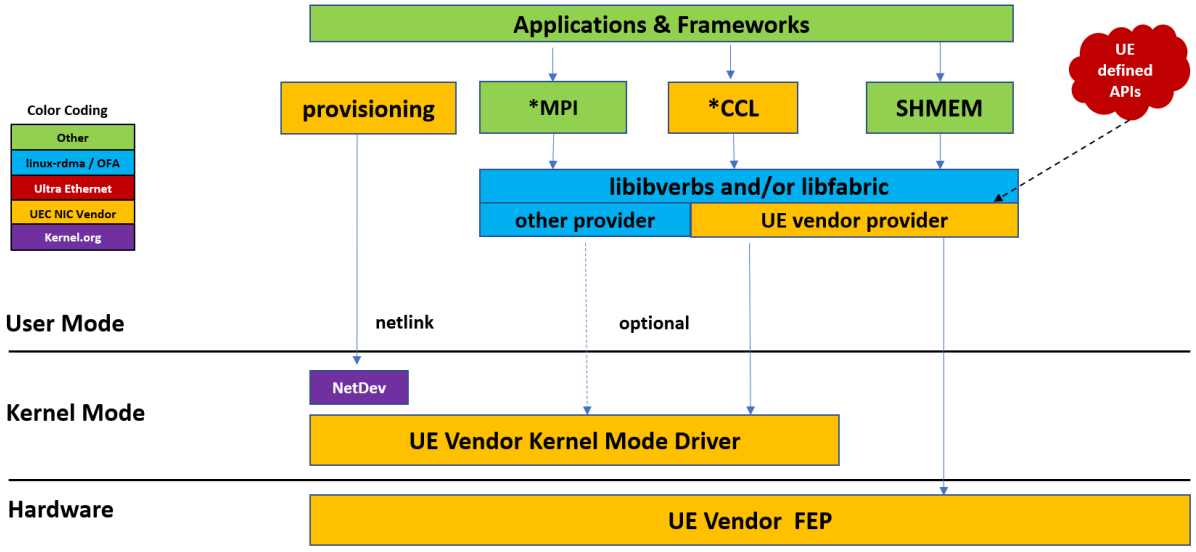


Figure 1-11 - UE Software Endpoint Stack

frameworks, to UE-powered nodes without requiring change. It is common for the frameworks to leverage a hardware-dependent vendor-specific *CCL library. However, a UE-compliant *CCL library, while not directly specified, does not require an application change.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

1.4.3 Switch Software Stack

Figure 1-12 shows an example of a switch software stack supporting UE features. UE operates over existing Ethernet switches, but additional capability may be obtained by supporting optional UE features in switches. Switches in the UE environment are expected to run a variety of network operating systems (e.g., SONiC, FBOSS, Junos OS, IOS, etc.). Optional UE functionality within the switch silicon can be accessed through a switch chip abstraction interface (e.g., SAI) that has been suitably enhanced for UE. No changes are required to the forwarding paradigm of the switch and its associated network operating system (NOS). IP-based forwarding can remain unchanged; however, optional features are defined for UE (e.g., packet trimming, CSIG and INC).

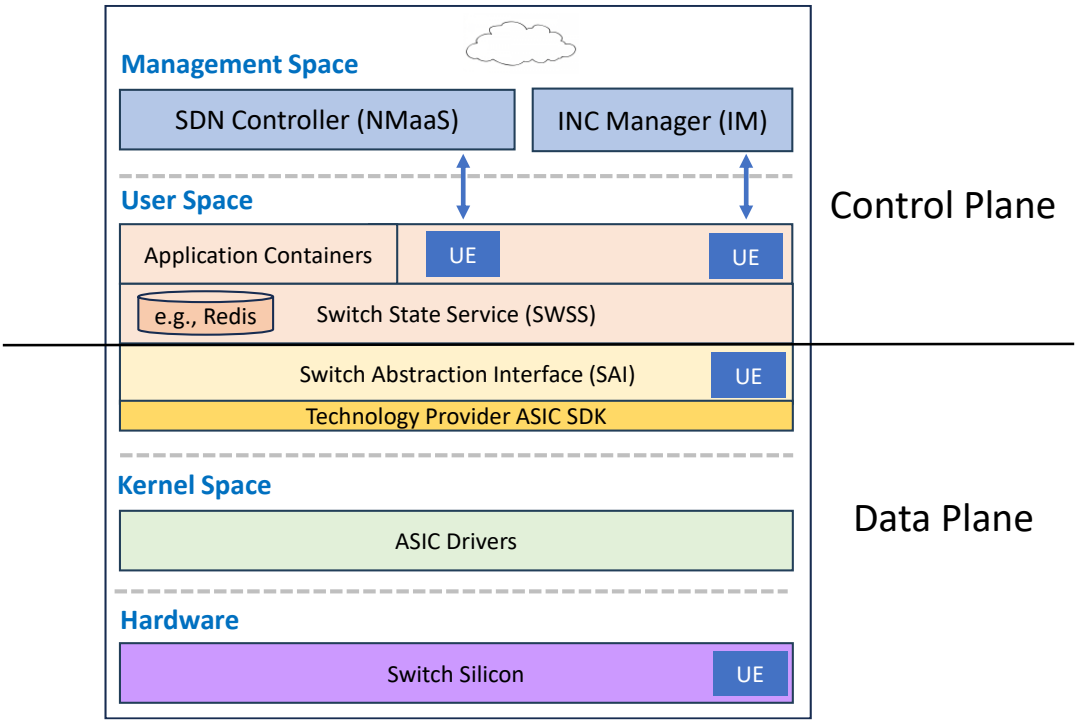


Figure 1-12 - Switch Module Layering

1.4.4 Network Operating System (NOS) Interface

The NOS provides essential services on switches for configuration and control. Some NOS components may need to be updated to support UE features. The following are examples:

- UE organizationally specific TLVs for LLDP.
- Packet trimming.
- SAI support for CSIG, CBFC, and LLR.
- INC agent software support.

Figure 1-12 shows the layering of various switch modules. The layer above a switch abstraction interface (SAI) is referred to as the NOS control plane, and functionality below the SAI is referred to as the NOS data plane.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

The control plane software interacts with data plane functionality through an abstract API. One example interface is the switch abstraction interface (SAI) developed in the Open Compute Project (OCP). SAI provides an abstraction layer on top of a vendor-provided software development kit. This abstraction helps the NOS interact with a single consistent set of APIs for hardware programming. See the following references:

- <https://www.opencompute.org/wiki/Networking>
- <https://www.opencompute.org/wiki/Networking/SAI>

The INC functionality includes an INC manager (IM) with specified APIs as discussed in Section 2.4. An SDN controller may be part of UE implementations but is not within scope of the UE specifications.

1.5 Networking

1.5.1 AI and HPC Network Taxonomy

UE differentiates three network types as illustrated in Figure 1-13:

1. Frontend network
2. Backend scale-out network
3. Local scale-up network

1.5.1.1 Frontend Network

The frontend network is the operational network in datacenters that connects all compute nodes to the outside world (e.g., other datacenters or end customers on the Internet). This makes the frontend network one of the most important components in the datacenter. Any loss of availability in the frontend transport leads to direct customer impact and related costs. Because the frontend network connects customers and distant datacenters, it may support a variety of transport protocols (e.g., TCP/IP, UDP/IP, and QUIC) that can operate over long-distance links with millisecond-level delays. Furthermore, multi-tenancy of the compute node is frequently used and may require network overlays to support virtual machine migration and network virtualization.

Fundamentally, frontend networks carry two types of traffic: “north-south” (NS) traffic to and from the outside world (i.e., other datacenters and customers) and “east-west” (EW) traffic from network endpoints within the same datacenter. Each traffic type has fundamentally different characteristics. For example, EW traffic is often higher bandwidth than NS traffic, and packets are less “valuable” (i.e., they can be discarded and retransmitted at lower cost than for NS traffic). Furthermore, EW traffic usually has more stringent latency (i.e., microseconds) and higher bandwidth requirements (e.g., tens-of-gigabit/s) as it often connects diverse services, such as deep call chains of microservices, serverless functions, or storage access. NS traffic is often customer-facing, and bottlenecks often occur outside the datacenter. These characteristics may limit latencies to single-digit milliseconds and tens-of-megabit/sec bandwidths.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

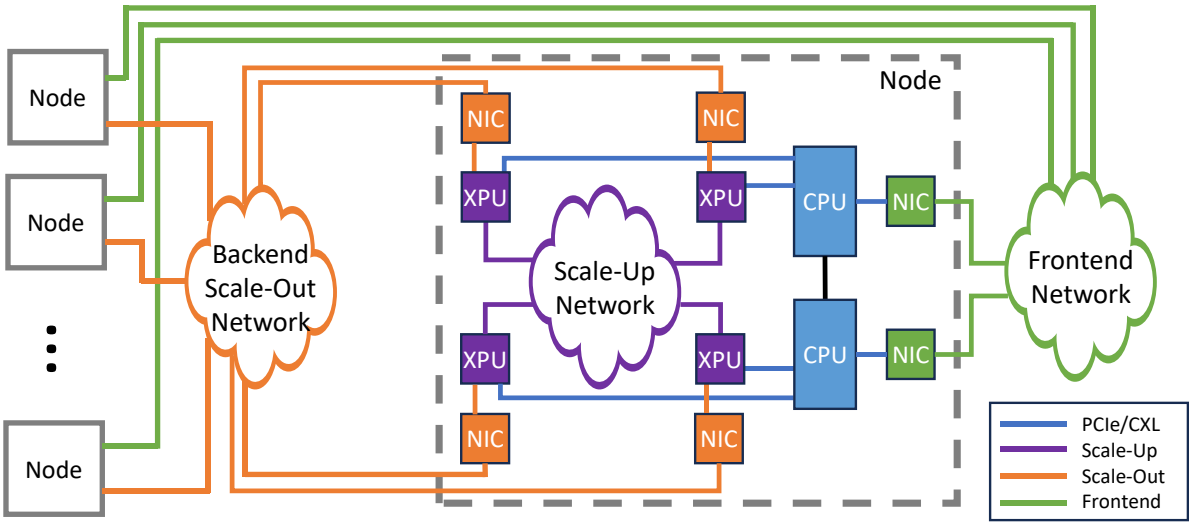


Figure 1-13 - Network Types

Handling these two types of traffic in addition to providing high availability make frontend networks quite complex. Switches and NICs need to support complex functionality such as filtering, policing, encapsulation, and security.

1.5.1.2 Backend Scale-out Network

The backend network is usually a specialized high-performance network of limited scope relative to the frontend network—often deployed across a "cluster" (e.g., a set of rows). It is sometimes also called a "scale-out" network. The backend scale-out network often forms its own layer-3 subnet and is not usually connected directly to the frontend network. Communication between frontend and backend networks often occurs through compute nodes with network interfaces into both networks.

The backend scale-out network serves very special purposes. For example, an HPC backend scale-out network enables communication via a message passing interface (MPI), while a deep learning backend scale-out network delivers training traffic. An AI-oriented backend scale-out network might include special-purpose optimizations such as switch support for collective operations over bulk data, while an HPC-oriented backend scale-out network might support latency optimizations only for small collectives.

Having two networks may increase the cost of the overall system (i.e., two networks instead of one — separate frontend and backend networks). However, two networks provide a clean separation of traffic (i.e., no interference) and design (i.e., allowing for different architectures and technology deployments). In some classic HPC systems, the backend scale-out network provides all the connectivity and network services for the compute nodes. However, this is achieved by retrofitting traditional transport protocols (e.g., TCP/IP) over the top of the HPC transport (e.g., portals, IPoIB) and accepting the trade-offs that are implied.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

1.5.1.3 Local Scale-up Network

Scale-up networks are typically very specialized short-range interconnects that often come with only a single or limited tier of switches or possibly no switch at all.

Historical scale-up networks support I/O coherency that is not always present in all types of interconnects. Modern examples used to connect accelerators (e.g., XPUs known as GPUs, FPGAs, or specialized SoCs) include AMD's XGMI, NVIDIA's NVLINK, Intel's Xe Link, switched PCIExpress, and CXL systems. The capabilities of these networks usually include memory semantics (which is similar to RDMA for bulk transfers) at the lowest available latencies (targeting sub-microsecond, for a small-scale or programmed memory access). The Unified Local profile can be used to build local scale-up networks with Ultra Ethernet. The optimized, compact header formats support high-performance and low-latency read/write operations. UE does not specify the user interface to those operations.

In typical 2024 datacenter environments, the three types of networks are distinguished by different characteristics, summarized in Table 1-1.

Table 1-1 - Distinctive characteristics by network type (circa 2024)

Characteristic (2024)	Frontend (Intra Data Center)	Backend Scale-out	Local Scale-up
Latency requirements[1] (One-way delay)	100 µs+	< 10 µs	< 1 µs
Single-link bandwidth requirements	up to 100 Gbit/s	up to 800 Gbit/s	up to 800 Gbit/s
Number of links	1 per node	1 to 2 (per accelerator)	Many (per accelerator)
Multi-tenancy requirements	Hundreds of tenants per endpoint	Up to tens of tenants per endpoint	Usually, single tenant
Security requirements	Important today (SSL/TLS encrypt everything)	Important in the future for some (depends on offering, encrypt everything should be optional)	Optional
Protocol requirements	Wide variety of transport protocols (all IP compatible, some workload-specialized proprietary, need to co-exist)	Proprietary protocols ok (desired for workload specialization, probably L3 headers for compatibility, no need to co-exist)	Proprietary protocols necessary (probably no IP for performance reason, L1/L2 headers only, some provide coherency)
Maximum link length requirement	1500 m	150 m	5-10 m
Deployment scale (network endpoints)	256k+	256k	100-1000
Topology	3 level Clos	2-3 level Clos or Dragonfly	full mesh or single stage switch
Single-job scale (network endpoints)	100	256k	100-1000

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Characteristic (2024)	Frontend (Intra Data Center)	Backend Scale-out	Local Scale-up
Note: 1. The latency requirement is a bound on the range of the congestion window for full bandwidth.			

1.5.2 UE Transport (UET) Objectives

UET is focused on enabling the workloads and use cases for HPC and AI (AIT and AII). UET mainly targets RDMA service and attempts to provide the best, modernized, and highly optimized transport service for carrying RDMA in AI and HPC workloads. The general characteristics are summarized in Table 1-2. The three use cases (i.e., dedicated AI training cluster, cloud AI/HPC, and at-scale HPC) involve organizations that leverage the full system scale of a single application as well as organizations who fill a significant fraction of the machine with single-node applications–and the full spectrum in between. UET also defines smaller optimized headers that can be used in local communication use cases. This allows vendors to implement efficient short-range interconnects with UE, which is crucial to support larger AI and HPC installations. The objective for UET is to serve the breadth of these use cases with a single transport protocol.

Table 1-2 - Characteristics of UET Deployment Model

Characteristic (2026+)	Dedicated AI Training Cluster	Cloud AI/HPC	At Scale HPC	Local Networks
Network scale (Ethernet ports)	100k—256k	256k	80k—256k	32—4K
Target unloaded one-way latency	2—10 µs	2—10 µs	2—10 µs	< 5µs
Ethernet port speed	800 G+	400 G+	800 G+	800 G+
Average network utilization	Up to 85%	20-40% overall BW 60-80% for AI cloud	Varies, 60-80% for BW-intensive apps	85% for small transactions
Packet rate	Low	Mixed	High	High
Message size	Relatively large	Mixed	Tiny to Mixed	Small
Encryption	Optional	Required	Optional	Optional
Multi-tenancy	Node-level job isolation	Node-level job isolation + network virtualization	Node-level job isolation	Node-level job isolation

An additional UET goal is to provide an accelerator-friendly interface. This involves defining a specification that minimizes the required hardware complexity for integrated endpoints. Another aspect involves defining a software solution that enables accelerators and other processors to do more in hardware. For example, UET may allow an accelerator to own the ‘fast path’ and move other functions (e.g., management and complex error handling) to a separate processor (e.g., host CPU). The details of a hardware implementation and interface within an endpoint to an accelerator are out of scope for this specification.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

While UET provides excellent performance on best-effort networks leveraging multi-pathing and improved congestion control assisted by network telemetry, it is also architected to run on lossless networks. For best-effort networks, UET embraces two fundamental lessons learned from the success of Ethernet, TCP/IP, and large-scale networks deployed for various applications including the cloud—that transport protocols should provide loss recovery and that many large-scale lossless fabrics are challenging to operate without triggering head-of-line blocking and congestion spreading. Following these principles, the UE transport builds on the proven path of distributed routing algorithms and endpoint-based reliability and congestion control.

1.5.3 Network Fabric

The network fabric consists of Ethernet switches and the associated elements described below.

1.5.3.1 Elements

The UE switch fabric incorporates three common functional planes: control plane, data plane, and management plane. These are depicted in Figure 1-14 and described as follows.

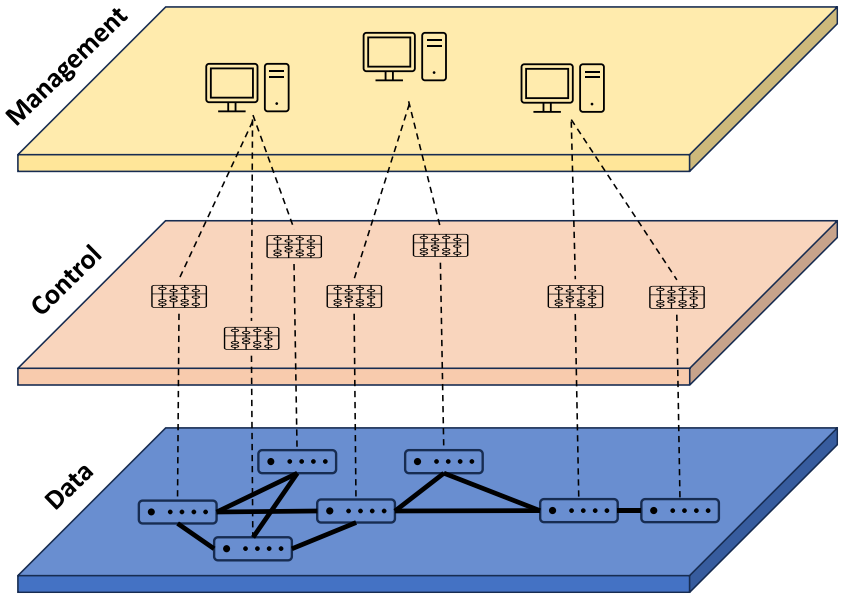


Figure 1-14 - Layered View of Networking Functionality

1.5.3.1.1 Control Plane

The control plane is responsible for running critical functions such as routing protocols to maintain communication between fabric switches. This layer is managed by networking operating systems (NOSs) such as SONiC, FBOSS, and others. The control plane interacts with the switch data plane using standard APIs such as SAI or vendor-specific APIs.

1.5.3.1.2 Data Plane

The data plane, also referred to as the forwarding plane, is responsible for forwarding packets in the network. This layer spans the UE endpoint (i.e., FEP) and the network switches. For clarity, the data

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

plane does not control or manage the UET FEPs. This layer comprises a lower-level abstraction of the switch hardware and is responsible for forwarding packets according to the forwarding information provided by the control plane.

1.5.3.1.3 Management Plane

The management plane is responsible for ensuring that the switch fabric is operational, reliable, and secure. Management systems and associated protocols perform software upgrades, monitoring, and other administrative activities. The management plane interacts with the control plane using standard interfaces such as Netconf, gNMI, SNMP, and others. The managed objects manipulated by the management plane are defined by standardized data models such as YANG and supported by vendor-neutral software such as OpenConfig (see: <https://openconfig.net>).

Informative Text:
Traditionally the management of endpoints has been separated from the management of the fabric. UE follows the traditional separation that industry and organizations are accustomed to. The UEC management working group is responsible for ensuring full functionality, performance, and interoperability of UE-compliant devices.

1.5.3.2 UE Switch Operation in Physical Networks

A UE-compliant switch operates in two types of physical networks:

1. **UE data plane network**: A network connecting FEPs to one another through UE switches. This network carries application traffic for various workloads and is optimized for UE specifications.

2. **Switch management network**: Every switch provides at least one dedicated Ethernet port to connect non-fabric endpoints such as SDN controllers, fabric managers, telemetry collectors, SNMP servers, and other devices responsible for managing the infrastructure. This network is not latency-sensitive and typically has low bandwidth requirements. However, some new capabilities, such as the INC manager (IM) interface and associated protocol operations, may be latency sensitive.

Informative Text:
Management functions such as the INC manager and fabric management running over the switch management network are within the scope of UE; however, the characterization of protocols used for these activities, such as the transport or link layer, are out of scope of UE.

1.5.3.3 Topologies

Topology is a critical part of an AI and HPC fabric as it sets the performance bounds by establishing the network diameter and bisection bandwidth. Deployments need to consider the optimal system cost in terms of energy consumption and physical aspects such as cable costs.

Congestion management in this specification targets Clos networks, while not excluding other topologies. However, no optimization or performance objectives are set for topologies other than a

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

folded Clos network (i.e., a fat tree). Congestion management in this specification has been simulated over a fat tree network topology.

1.5.3.4 In-Network Collectives (INC)

Network-based collectives can consume a significant portion of the total training time. In these cases, it is desirable to deploy mechanisms that optimize network use, reduce completion times, improve application performance, and improve total cost of ownership. One such mechanism is in-network collectives (INC). The theoretical gain from INC in network bandwidth utilization for associated completion time is 2x. The allreduce collective operation requires transmitting and receiving 2x the message size for distributed computation (e.g., to the GPUs), while placing a computation element in the center of the network (e.g., in a switch) cuts the required BW to 1x.

INC reduces the completion time of collective operations and at the same time frees CPU or accelerator resources for computation.

1.5.3.5 Network Constraints

The UE fabric is constrained to use IPv4/IPv6-based layer 3 forwarding or UFH forwarding when operating under the Unified Local profile. A UE fabric that uses tunnels (e.g., VXLAN) is not currently specified and is left to implementors. Multitenancy can be addressed at the FEP level through encrypted tenant application data, specific allocation of JobIDs and may also take advantage of existing tunneling mechanisms.

UE does not require changes to the network layer and can use existing routing protocols. UE switches use equal cost multipath (ECMP) routing for load balancing where the entropy values are managed by the UET congestion management sublayer (CMS). The congestion management algorithms are designed with the expectation that fabric switches do not modify the entropy values and that any two packets with the same entropy values take the same path through the UE fabric. CMS expects UE switches to support explicit congestion notification (ECN) as specified in IETF RFC 3168, but with the additional constraint of marking congested packets when dequeuing for transmission rather than enqueuing. Packet trimming, as specified in Section 4.1, is an additional mechanism for congestion notification and makes use of multiple differentiated services code points (DSCP) to identify packets that can be trimmed and have been trimmed as well as assure they are mapped to appropriate traffic classes for expedited forwarding.

Traffic classes are embodied in the mechanisms and resources within endpoints and switches used for the differentiated transmission of packets (e.g., queues, buffers, schedulers). Traffic classes are differentiated from one another and can be prioritized between one another. Packets are mapped to traffic classes using attributes and header fields of the received packets. UE primarily relies upon the DSCP field in the IP header to identify the traffic class of a received packet.

Informative Text:
Traffic classes are specified at multiple levels. UE maps the traffic class specified at the libfabric layer to traffic classes across the fabric. For example, UET recommends separate traffic classes for requests and ACKs/NACKs. UET chooses a broad definition of traffic class to acknowledge implementations that

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

incorporate queuing resources and forwarding mechanisms that enable the differentiated forwarding of packets identified by the differentiated services code point (DSCP). The DSCP can identify 64 distinct differentiated services, 16 of which are available for local definition and use. Implementations provide a variety of ways to configure the mapping of packets identified by DSCP to traffic classes at the endpoint link and switch level. There is no current standard identified or defined by UE to perform this mapping. In some cases, multiple DSCP values may map to the same traffic class (see the UET DSCP mappings table in the CMS Section 3.6.4.7). UE features depend upon the consistent configuration and use of traffic classes at the endpoint and across the fabric. UE recommends a consistent mapping, and the network operator is responsible for assuring this consistency.

Figure 1-15 shows the mapping of application requested traffic classes to traffic classes available at the link layer on endpoints and switches. Items depicted within the dashed boxes are configuration values available to the UE operator at different layers of the UE solution stack. Applications can specify the desired libfabric traffic class using the *fi_domain()* library API. If unspecified, the UE libfabric mapping Section 2.2 provides default DSCP values to use for requests. The CMS specification provides a table of DSCP to Traffic Class mapping in Section 3.6.4.7. This table describes how different classes of DSCP values are mapped to traffic classes at the link.

The DSCP values provided for message requests from the libfabric layer are passed through and categorized as either DSCP_TRIMMABLE or DSCP_NO_TRIM. The UET protocol includes DSCP values for generated ACKs, NACKs, and control packets that are categorized as DSCP_CONTROL. The UET protocol can also categorize retransmitted data packets with DSCP_TRIMMABLE_RETX. All these DSCP categories are mapped into the link-level traffic classes TC_high and TC_low that are prioritized higher and lower with respect to one another.

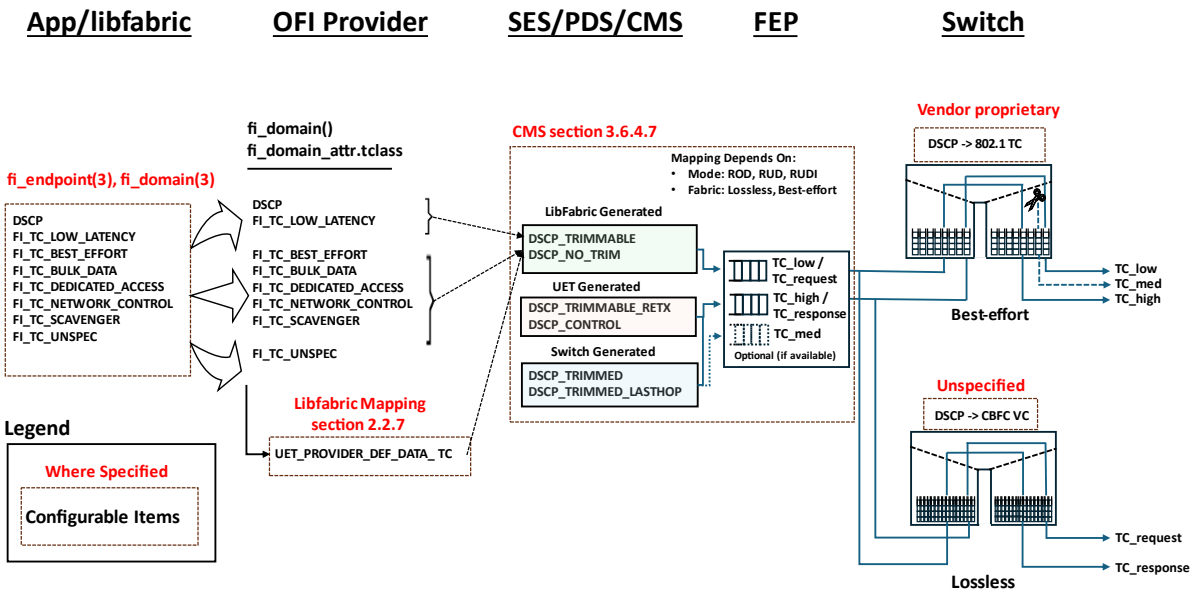


Figure 1-15 - Traffic Class Mapping

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Switches that trim packets (denoted by the scissor icon in Figure 1-15) change the DSCP values to DSCP_TRIMMED or DSCP_TRIMMED_LASTHOP depending upon where they are in the fabric topology. The DSCP values for trimmed packets can be mapped to a third traffic class (TC_med) if available, otherwise TC_high is used. The management actions for mapping DSCP values to the traffic classes within the switches is either vendor-specific or currently unspecified.

The congestion management sublayer of UET is designed with the expectation of using at least two traffic classes for PDCs to achieve high performance. The network operator is responsible for allocating and configuring the traffic classes used by UET. The mapping of UET packet types to traffic classes is dependent on whether the network is best effort or lossless. See CMS Section 3.6.4.7 for further details.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

1.6 UE Specification Overview: Layers

The UE specification spans multiple layers from software down through the physical layer. Figure 1-16 shows the required and optional components of the UE specification by layers. An overview of each layer is provided in the following sections:

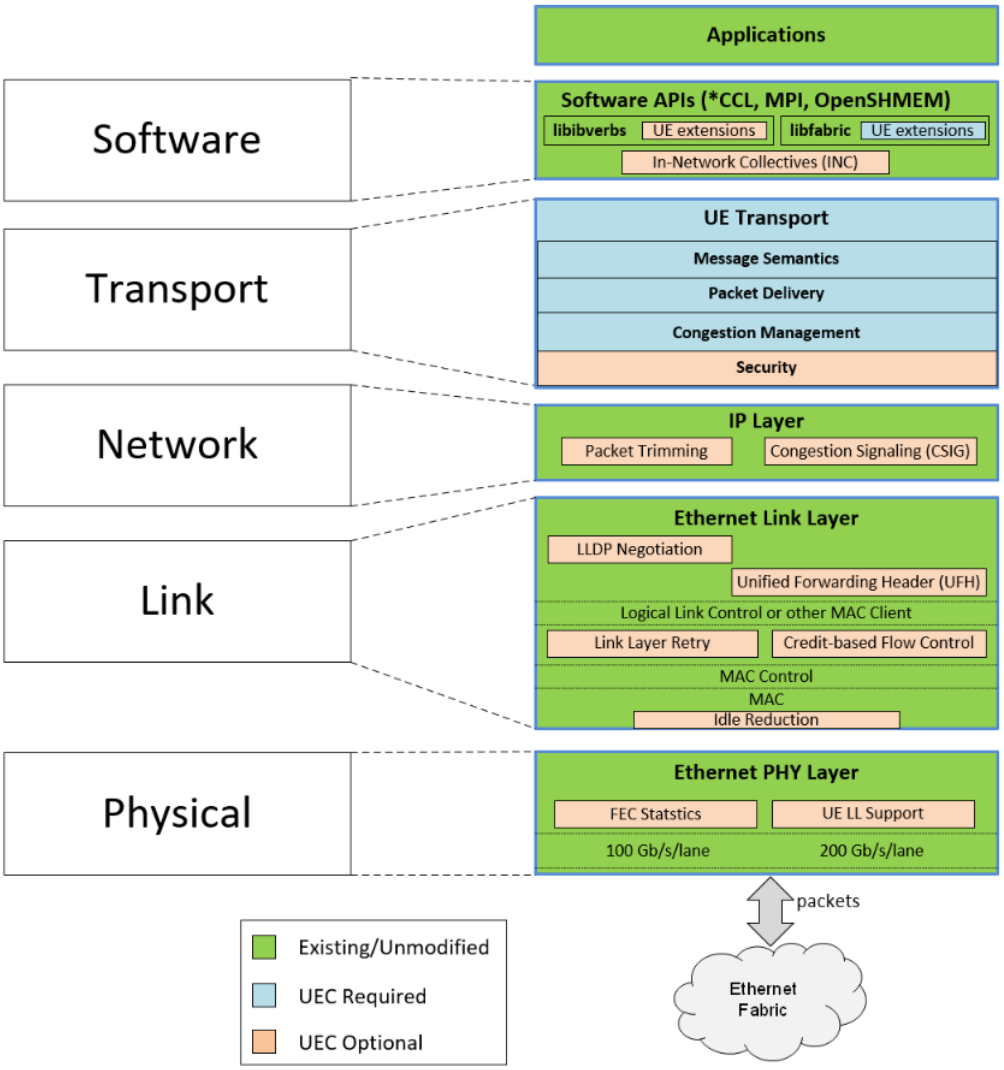


Figure 1-16 - UE Specifications by Layers

1.6.1 Software Layer

UE Software specifications are provided in Section 2. This includes a mapping to the libfabric API, a mapping to libibverbs, and the specification of UE INC.

libfabric mapping: UE-compliant implementations support the Open Fabrics Interface–libfabric API <https://ofiwg.github.io/libfabric/>. Libfabric v2.0 denotes the baseline API for UE-compliant endpoints. Libfabric is the northbound API where the UE API is defined and compliance is checked. UE specifications are expected to maintain alignment with the libfabric community. Libfabric is chosen because it supports many workloads over RDMA-based fabrics and is implemented by many vendors building

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

hardware and software to meet the specification. Multiple vendors have successfully created products that enable MPI-based HPC applications while also allowing PGAS, SHMEM, and other programming models. At the same time, vendors have created libraries that support popular AI frameworks (e.g., PyTorch, TensorFlow, or ONNX). All these vendor offerings have proven to be easily and efficiently mapped over libfabric. UEC collaborates with the libfabric community to extend the libfabric API, as appropriate and necessary to support new UE features.

libibverbs mapping: The UET libibverbs mapping specification defines a standardized set of extensions to the widely adopted RDMA verbs API that enables applications and software stacks to access Ultra Ethernet Transport (UET) capabilities through a familiar programming model. By maintaining compatibility with existing RDMA infrastructure while introducing support for UET, the specification provides a practical migration path for applications, libraries, and operating system components to adopt Ultra Ethernet with minimal disruption. This approach accelerates adoption, simplifies integration with the Linux RDMA software stack, enables multi-vendor software implementations, and provides a foundation for AI, HPC, and future Ultra Ethernet use cases while preserving long-term extensibility and transport independence.

In-network collectives (INC): Collective operations such as allreduce are common in many AI and HPC applications. The completion of collective operations gates the next computing phase in many jobs, therefore accelerating the completion of the operation and optimizing the network transfer of collective data is an important optimization. The optional INC specification enables switches to offload multiple collective operations and provides application-level performance improvements as well as network-level traffic optimizations.

1.6.2 Transport Layer

UE transport layer specifications are provided in Section 3. The UE transport protocol is designed to serve the networking demands of both HPC and AI workloads. Different profiles are defined to allow product optimization for satisfying the unique needs of the workloads. It is anticipated that the network requirements for AI and HPC workloads will increasingly overlap. The UE transport protocol enables a wide range of implementations. The components of the UE transport protocol include message semantics, packet delivery reliability modes, congestion management, and security.

Semantics Sublayer (SES): The SES sublayer is designed to integrate into broadly deployed AI frameworks and HPC libraries through a libfabric mapping. Applications using libfabric exchange messages over the fabric and place those messages directly into one another's buffer memory using popular zero-copy techniques. The SES sublayer specifies a protocol that defines how application messages are identified, how the associated buffers are addressed, and how the preferred operations on the messages are employed. The SES sublayer is the primary interface between the UE transport and the libfabric provider.

Packet Delivery Sublayer (PDS): Application requirements determine the selection of the appropriate UET packet delivery services. Different applications are optimized for various reliability and packet ordering constraints on message delivery. Through the UET layering model and associated libraries

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

applications can select the transport protocol functionality that best suits their needs. The PDS sublayer defines a protocol with multiple modes of operation that offer all combinations of reliable, unreliable, ordered, and unordered packet delivery services.

Congestion Management Sublayer (CMS): End-to-end congestion management is essential to achieve high network efficiency, reduce packet loss, and minimize latency while maintaining fairness between competing flows. Traffic classes are used in the network to separate traffic flows that have different characteristics and requirements from the network. To maintain fairness and assure a low latency control loop, UE congestion management is designed to be used on all traffic in the same traffic class. Traffic class configuration is the responsibility of the network operator. High network efficiency and reduced latency are achieved by allowing UET congestion management to enable multi-path packet spraying across the fabric and avoid hot spots when congestion signals arrive. Under UET, PDCs with unordered flows may simultaneously use all paths to the destination, achieving a more balanced use of all network paths. Link load imbalances are avoided by the coordinated choice of the paths between the endpoints and switches guided by real-time congestion management. This fine-grained load balancing results in improved network utilization and reduced tail latency.

Transport Security Sublayer (TSS): AI training and inference often occur in hosted networks where job isolation is required. Moreover, AI models are increasingly sensitive and become valuable business assets. Recognizing this, the UE transport incorporates network security by design and can encrypt and authenticate all network traffic sent between computation endpoints in an AI training or inference job. As jobs grow in scale, it is necessary to support encryption without ballooning the session state in hosts and network interfaces. In service of this, UET incorporates new key management mechanisms that allow efficient sharing of keys among large numbers of compute nodes participating in a job. It is designed to be efficiently implemented at the high speeds and scales required by AI training and inference. HPC jobs hosted on large Ethernet networks have similar characteristics and require comparable security mechanisms. Note that TSS is an optional feature.

1.6.3 Network Layer

Optional UE network layer feature specifications are provided in Section 4. UE does not require any changes to the network layer, but UET congestion management expects support for explicit congestion notification (ECN) as specified in IETF RFC 3168, but with the additional constraint of marking congested packets when dequeuing for transmission instead of enqueuing.

Packet Trimming: Congestion is inevitable within the fabric. As fabric speeds increase and more pressure is put on limited switch chip buffering, congestion signals become more prevalent and the information within those signals becomes more important in determining a corrective course of action. UE defines a packet trimming feature that allows switches to truncate contested packets, modify the DSCP field of the truncated packet, and forward the truncated packet toward the destination as the congestion signal. Packet trimming provides considerably more congestion information than the ECN bits alone. Packet trimming is optional for switches to implement and mandatory for FEPs to receive trimmed packets.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Congestion Signaling (CSIG): Congestion Signaling (CSIG) is a lightweight in-band telemetry mechanism designed to meet the demands of modern AI, HPC, and high-speed datacenter networks, where traditional congestion signals such as ECN, packet loss, and RTT are too slow and coarse-grained. By embedding a small fixed-size tag in packets, CSIG collects real-time, per-path congestion information—such as available bandwidth, capacity, delay, and queue buildup—directly from network switches, enabling faster and more accurate congestion control, improved flow completion times, better network utilization, and enhanced visibility for troubleshooting, monitoring, and traffic engineering, all with minimal overhead and compatibility with encrypted traffic.

1.6.4 Link Layer

Optional UE link layer specifications are provided in Section 5. The UE specification adds several optional features to the Link Layer. Figure 1-17 shows the areas of focus for the UE link layer specifications with respect to IEEE 802 architecture at the link layer. UE's optional recommendations for the Link Layer relate to the shaded areas. All features are optional for UE compliance.

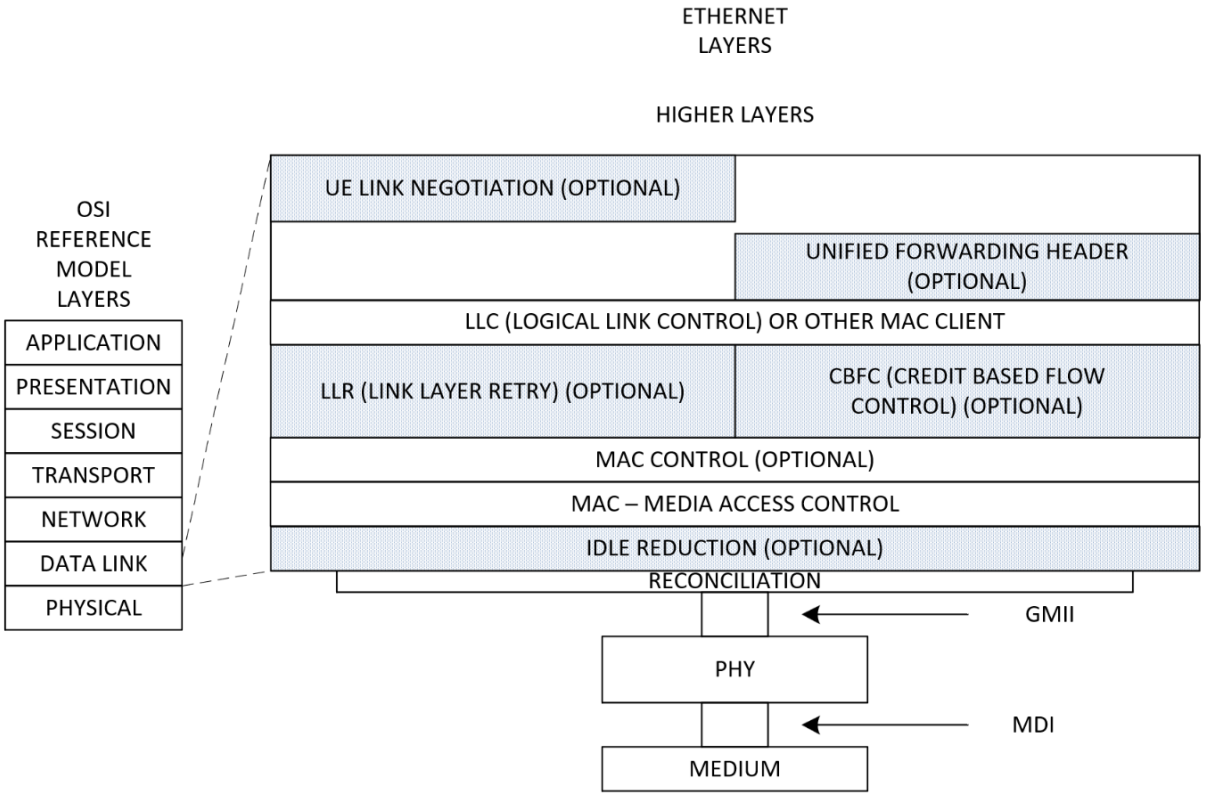


Figure 1-17 - UE Link Layer Specification Focus Areas

Idle Reduction (IPGR): The Idle Reduction feature is an optional enhancement that reduces the number of idle bytes transmitted between Ethernet frames, improving wire efficiency and increasing achievable packet rates, particularly for small packets. By minimizing unnecessary inter-packet overhead while maintaining frame alignment requirements, the feature allows more payload data to be carried over the same link bandwidth. This helps improve network performance, throughput, and efficiency in high-

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

speed environments where maximizing packet processing rates and reducing transmission overhead are important.

Link Layer Retry (LLR): With speeds and scale increasing, and with the extreme bandwidth densities common in accelerator networks, the traditional approach of relying only on end-to-end retry to address packet drops is increasingly burdensome for latency-sensitive workloads. Local error handling at the Link Layer has proven valuable in scale-out HPC networks, such as those used in exascale systems. The UE specification provides this capability for Ethernet.

Credit-Based Flow Control (CBFC): Traditionally Ethernet networks refrain from using credit-based links, which are common in fabric technologies. However, some recently introduced products support it with optional improvements for some workloads. CBFC is an optional feature of the UE link layer.

Unified Forwarding Header (UFH): Unified Forwarding Header (UFH) is a new Ethernet-compatible header format that improves network efficiency and throughput for AI, HPC, and other performance-sensitive workloads by significantly reducing packet overhead. UFH achieves this by compressing and simplifying traditional Ethernet and IP header information, eliminating redundancies and unnecessary fields while preserving coexistence with standard Ethernet traffic and management protocols. By increasing packet rates and improving bandwidth utilization—particularly for small-message communications that dominate many scale-up and scale-out deployments—UFH delivers the efficiency benefits of specialized high-performance fabrics while retaining compatibility with the broader Ethernet ecosystem.

UE Link Negotiation: UE specifications support “existing Ethernet switches” but introduce multiple optional new features that benefit from discovery and feature negotiation capabilities. While it is an objective of UE to operate on a dedicated backend network for inter-accelerator communication, the objective does not mitigate the need to support discovery and feature negotiation amongst all entities (endpoints as well as switches) on the network. UE promotes the notion of “profiles” that describe required and optional features. It is necessary to detect, discover, and reach consensus among all network entities to interoperate with profile-supported features. UE assumes that standard negotiation mechanisms like LLDP are prevalent in the industry and are extensible for the aforementioned purposes.

1.6.5 Physical Layer

UE physical layer specifications are provided in Section 6. UE is specified for physical layers using 100G per lane signaling as defined by IEEE Std 802.3.

Figure 1-18 shows the areas of focus for the UE physical layer specifications with respect to IEEE 802.3 PHY architecture. UEC’s optional recommendations for the Physical Layer relate to the shaded areas.

IEEE 802.3 100G and 200G Per-Lane Signaling: The UE physical layer section lists the IEEE 802.3 specifications that are within the scope of UE.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

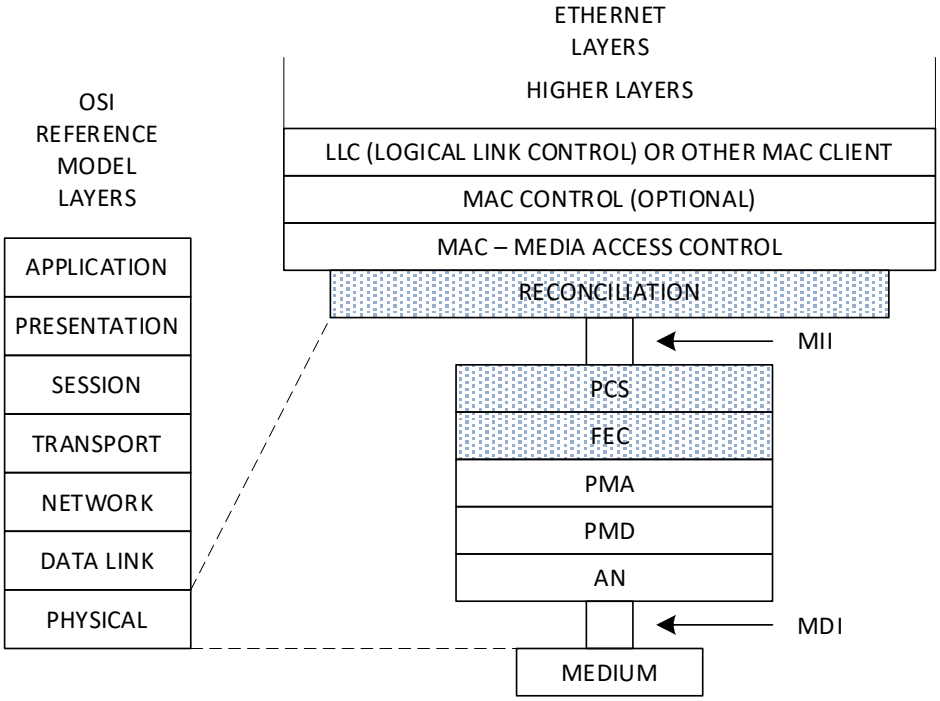


Figure 1-18 - UE Physical Layer Specification Focus Areas

Channel Quality Assumption: Accelerator nodes are more complex than standard endpoints or TOR switches. At the time of publication, it is assumed that IEEE standards are sufficient, while UE products are encouraged to build more robust channels.

FEC Statistics for Prediction of Link Quality: UE networks are assumed to comprise multiple high-performance links protected by forward error correction (FEC), on which data loss due to physical layer errors is extremely infrequent. However, on a large-scale network there can be a few outlier links with more frequent errors than the rest of the network. With massively parallel applications, such links can require frequent retransmissions and thus become the performance bottlenecks of the whole network. The UE PHY specification includes a method of estimating the mean time between PHY errors (MTBPE) on each link from the statistics of the FEC decoder. This estimation enables identifying poor performance and thus provides an opportunity to improve the network performance (e.g., by removing the weakest links from the network or servicing their endpoints).

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

2 UE Software Layer

2.1 UE Software Overview

The UEC develops specifications, software APIs, and open-source code for various AI/HPC use cases/applications. This includes, but is not limited to, software related to the following areas:

- Support of UE transports and API within the OFI ecosystem (libfabric).
- Support of UE transports and API within the Linux RDMA subsystem.
- Software reference models of UET transport.
- Linux kernel API changes and extensions to support UET.
- Interface additions and changes required to support UET where appropriate (e.g., within the Linux kernel).
- Example applications.

2.1.1 Software Specifications

2.1.1.1 Libfabric Mapping

The HPC and AI industries have established the use of abstraction layers within communication libraries to isolate low-level details of the transport layer from the inner workings of the communication libraries. UE adopts libfabric [1], also known as Open Fabric Interfaces (OFI), as the communication abstraction layer. The libfabric mapping specification defines how the libfabric APIs are implemented to support the Ultra Ethernet Transport. The goal of this mapping is to extend and change libfabric as required to provide a vendor-interoperable mapping that supports all the Ultra Ethernet profiles defined by the transport working group. The specification also identifies the requirements imposed on UET libfabric by this mapping.

2.1.1.2 Libibverbs Mapping

The verbs API, or more simply known as “verbs,” has broad industry adoption and direct *CCL use. As defined, verbs targets RDMA transports and is tightly coupled with the Linux RDMA kernel subsystem. The UE libibverbs mapping extends libibverbs to operate over UET. The key benefits of this specification include:

- Increases adoption of UET by simplifying the migration and transition of RDMA verbs applications over UET.
- Strengthens UET integration within the Linux RDMA subsystem.
- Allows NICs that support multiple transports, such as RoCE and UET, to leverage a common software API model and infrastructure.
- Allows multiple vendors to share higher-level software implementations for advanced UE profile support. For example, a common libfabric provider may be implemented over the UE libibverbs mapping for devices that do not natively implement tag matching or other advanced semantics in hardware. A common provider can support the AI full or HPC profiles.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

2.1.1.3 In-Network Collectives (INC)

Various collective operations are utilized in both AI and HPC workloads and *CCL and MPI libraries. These operations can be offloaded to the fabric or FEPs to gain performance benefits, such as reduced latency or increased throughput, or to gain efficiency benefits such as reduced processing or memory utilization. The INC feature requires the specification of software entities and APIs. An INC solution includes participation between end-point agents (part of the UEC libfabric provider), an INC manager (IM), and switch agents. Ranks interact with the IM using a gRPC API. This interface identifies fabric capabilities, supports error handling, and allows members to join and leave collective groups.

The IM interacts with switch agents using a gRPC API. This interface is used by the IM to identify switch capabilities, allocate and deallocate resources for a collective group, and support error handling and telemetry.

2.1.2 Software Components and Interfaces

UE software spans components and interfaces in user space and (Linux) kernel space. The complete scope of the work is illustrated in Figure 2-1 and summarized in Table 2-1.

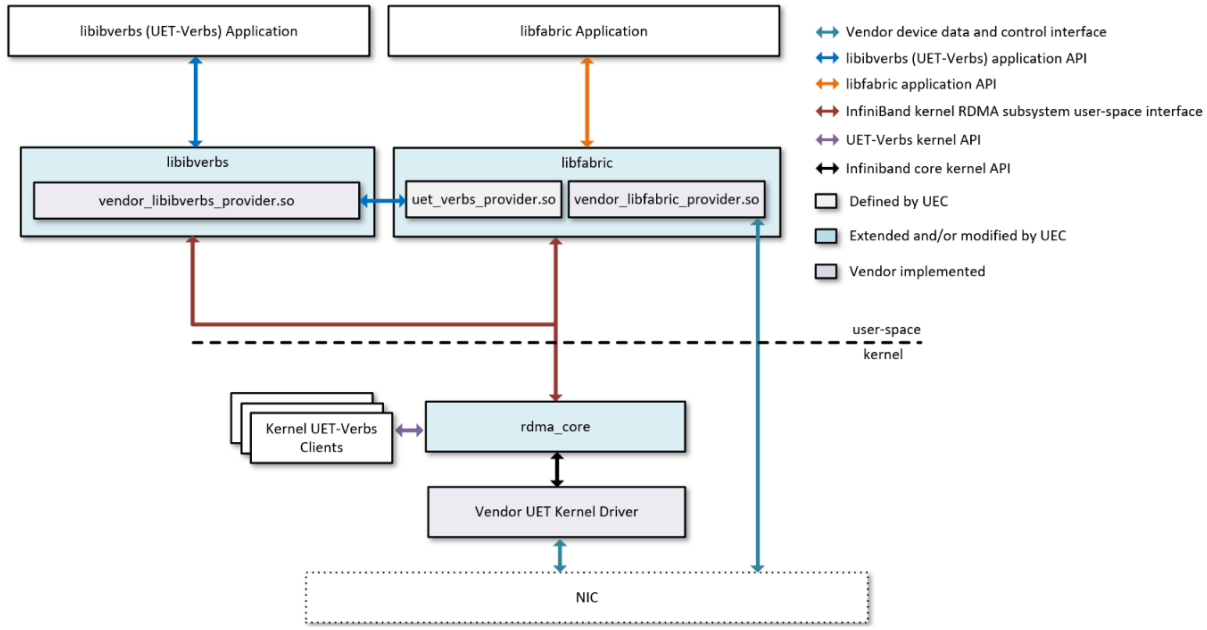


Figure 2-1 - Components and Interfaces Defined By UEC

Table 2-1 - UE Software Components and Interfaces

Interface or Component	Functionality	Scope of Work
• libfabric application API	User space application API	• Definition and implementation of changes and extensions required of libfabric and extensions.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Interface or Component	Functionality	Scope of Work
		• Definition of interoperability requirements between vendor provider implementations. • Definition and implementation of libfabric provider for Transport Reference Provider Model [3] and Linux kernel implementation [4].
• libibverbs application API	User space application API	• Definition and implementation of changes and extensions required of libibverbs and extensions. • Definition of interoperability requirements between vendor provider implementations. • Definition and implementation of libibverbs provider Linux kernel RDMA subsystem changes.
Kernel subsystem APIs	Access to common kernel functions and interface with specific subsystems (e.g., rdma_core)	• Define and implement extensions to other kernel subsystems to support UET. • Upstream as necessary.

2.1.2.1 Integration Model of FEP

A FEP exposes a raw L2 Ethernet device to the operating system to facilitate integration into the Ethernet ecosystem (i.e., ARP, ICMP, etc.). A UE FEP implements a netdev [6] Ethernet device for integration into the Linux kernel networking stack and a vendor-supplied libfabric provider driver for integration into the appropriate middleware. In the Linux kernel, the UE transport is included via a stacked driver architecture similar to how a RoCE device is integrated into the kernel. Vendor implementations may vary from the FEP integration model described in this specification.

2.1.3 Reference Software Models and Supplementary Software

UEC provides several software artifacts to assist implementors as follows:

Table 2-2 - UE Reference Software

Software	Description
UE transport user-space reference model [3]	A user space implementation of a subset of UET (SES, PDS, CMS, and TSS core algorithms) deployable as a standalone model or as a libfabric provider driver.
UE transport kernel-space reference model [4]	Kernel implementation of a subset of UET (SES, PDS, TSS, and CMS sublayers). Includes kernel subsystem interface and functional changes as appropriate
RCCL UET reference plugin [5]	A version of the RCCL networking plugin that provides end-to-end connectivity for a subset (*ncclSend, ncclRecv*) of the RCCL networking

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Software	Description
	calls over UET using the UET reference provider model. This provides an example of how an application can use the UET libfabric implementation.
UE Wireshark plugin [2]	A fork of the Wireshark open-source network protocol analyzer with a dissector for UET

2.1.4 References

The following references are utilized by the UEC Software Working Group. Github repositories [2], [3], [4], and [7] are currently private and require UEC membership for access. These repositories are planned to be transitioned to allow public visibility.

[1] GitHub, "ofiwg/libfabric OpenFabrics Release v2.0.0," 2024. [Online]. Available: https://github.com/ofiwg/libfabric/releases/tag/v2.0.0.

[2] Github, "ultraethernet/uet-wireshark UE Wireshark Plugin," 2025. [Online]. Available: https://github.com/ultraethernet/uet-wireshark.

[3] Github, "ultraethernet/uet-ref-prov UE transport user-space reference model," 2025. [Online]. Available: https://github.com/ultraethernet/uet-ref-prov.

[4] Github, "ultraethernet/uet-linux-kernel UE transport kernel-space reference model," 2025. [Online]. Available: https://github.com/ultraethernet/uet-linux-kernel.

[5] Github, "ultraethernet/uet-rccl-plugin RCCL UET reference plugin," 2025. [Online]. Available: https://github.com/ultraethernet/uet-rccl-plugin.

[6] Netdev, "Network Devices, the Kernel, and You!," [Online]. Available: https://www.kernel.org/doc/html/latest/networking/netdevices.html. [Accessed 2025].

[7] Github, “ultraethernet/uet-rdma-core RDMA core user-space libraries and daemons,” 2026. [Online]. Available: https://github.com/ultraethernet/uet-rdma-core

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

2.2 UE Libfabric Mapping

This document specifies the mapping between libfabric v2.0, also known as Open Fabrics Interfaces (OFI), and the Ultra Ethernet Transport (UET v1.0), as well as the requirements imposed on UET OFI providers. The mappings are an important part of enabling communication libraries and services over UET, including support for MPI, *CCL, SHMEM/PGAS, and other communication uses. UET must uphold the expectations of these communication libraries and services. Additionally, the mapping is dependent on the UET semantics and profiles. It is expected that each UET Fabric End Point (FEP) vendor will provide a UET libfabric provider that is optimized for their FEP; however, this specification defines requirements that enable interoperability between UET FEPs.

Libfabric has been selected as the primary network data plane API for UET as it is a flexible open-source framework utilized by various communication libraries for AI and HPC workloads. Libfabric is incorporated into this specification by normative reference. This specification does not attempt to provide a tutorial description of libfabric. Details about libfabric and the libfabric APIs are available from libfabric.org. Where possible, references to the libfabric main pages are provided. Note that these references are subject to change.

In summary, libfabric provides a communication API tailored for high-performance, parallel, and distributed applications. As a low-level communication library, it abstracts various networking technologies. However, in this context, we define its use for UET, aiming to eliminate ambiguity while enhancing interoperability and simplifying debugging. Libfabric has been deployed at scale, and its open-source ecosystem is suitable for future enhancements that can be delivered as UET evolves. Libfabric supports Linux, Windows, FreeBSD, and macOS platforms and uses a dual GPLv2/BSD license or a compatible license, such as MIT. Extensibility has been designed into the API.

Libfabric is built around the concept of plugins known as “providers.” The software components of UET functionality are part of a vendor-supplied UET provider. The libfabric core communicates with all providers via a common provider API. The libfabric software architecture is illustrated in Figure 2-2. The applications shown in the diagram are illustrative and not meant to be comprehensive.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

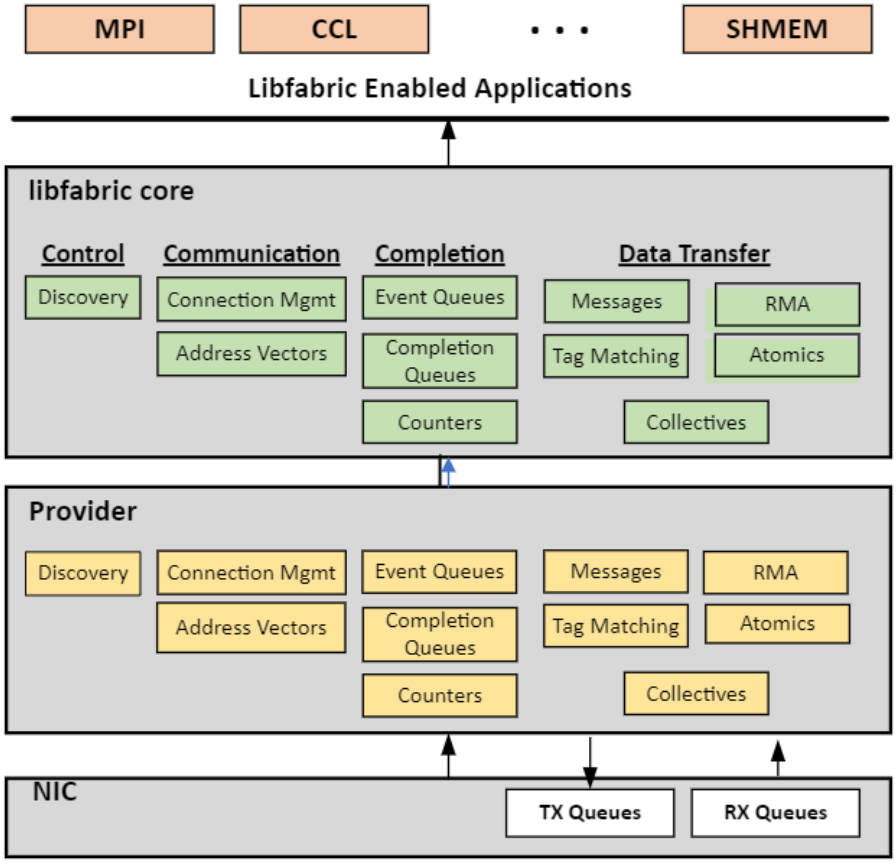


Figure 2-2 - Libfabric Software Architecture

The libfabric APIs are partitioned into four main categories:

- Control (discovery)
  - Used to determine the types of communication services available
- Communication (connection management, address vectors)
  - Used to set up communication between endpoints
- Completion (event queues, completion queues, counters)
  - Used to report data transfer operation results, connection setup status, collective joining results, and other asynchronous events
- Data Transfer (messages, tag matching, RMA, atomics, collectives)
  - Used to transfer data between endpoints, supporting different communication paradigms
  - Four of the data transfer paradigms shown in Figure 2-2 (messages, tagged messages, RMA, and atomics) target point-to-point communication. Collectives are a fifth data transfer paradigm targeting coordinated atomic operations among a large set of peers.

Each of these API categories is covered in this specification.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

A libfabric provider implements the libfabric API over vendor-specific lower-level software and hardware interfaces. Libfabric does not define the software/hardware interfaces that a provider uses to access network hardware. The libfabric UET Provider Software Architecture is illustrated in Figure 2-3.

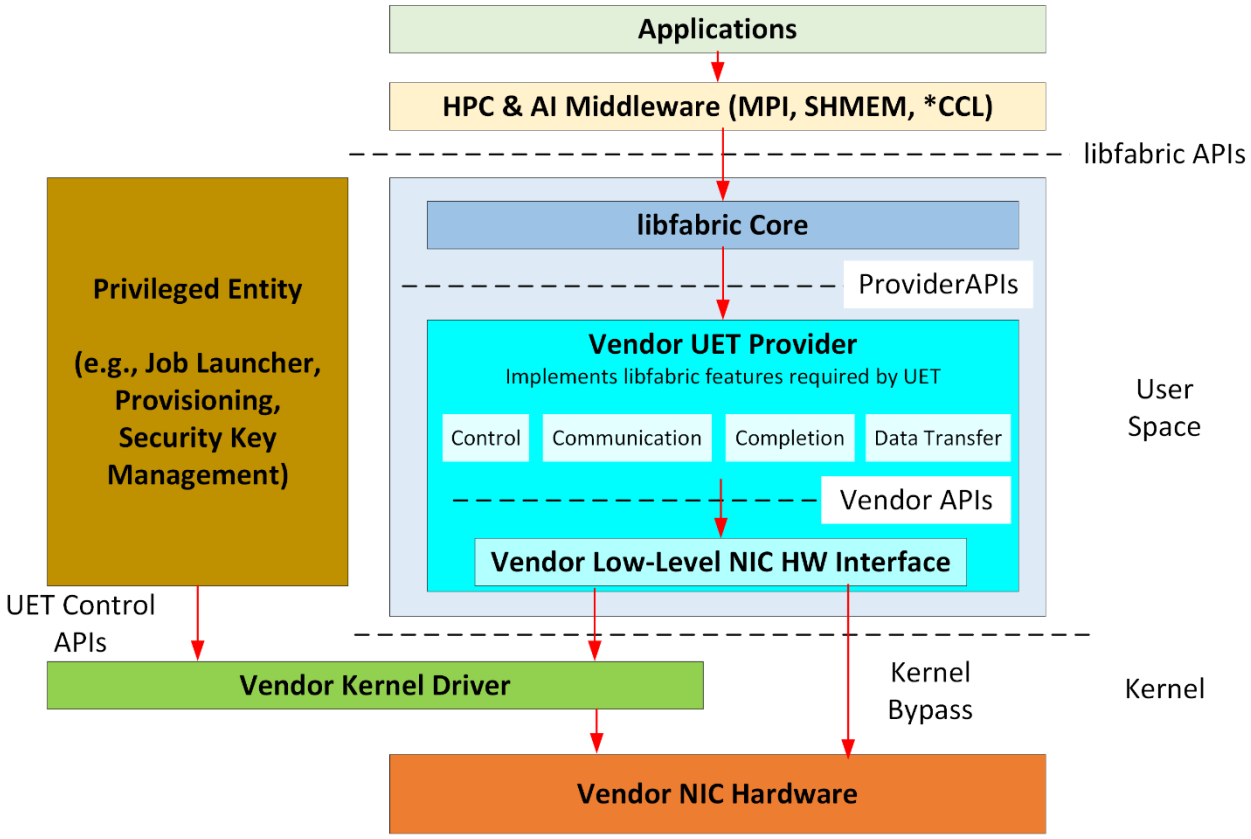


Figure 2-3 - Libfabric UET Provider Software Architecture

A kernel driver MUST be provided to facilitate operations that require a privileged entity, such as JobID assignment, provisioning, or security key management. The vendor low-level NIC hardware interface SHOULD use kernel bypass techniques when accessing the hardware for performance reasons.

The interface between the kernel driver and a privileged entity is referred to as the UET Control API. The privileged entities that interface with the kernel driver are expected to vary across deployments. Thus, UET standardizes the UET Control APIs for Linux implementations to allow:

- Kernel drivers that implement the standard interface to interoperate across deployments, and
- Privileged entities to interoperate with kernel drivers from multiple vendors.

Netlink was selected as the foundation for the UET Control APIs in Linux environments due to its flexibility, extensibility, and wide availability. Implementations for non-Linux operating systems are expected to provide similar functionality. More details on the UET Control APIs are provided in subsequent sections of this specification.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

2.2.1 Application Use Cases

Different applications have different communication requirements. This section highlights several application use cases that were considered by UET along with a high-level summary of their expected usage of the libfabric APIs. Additional use cases were considered, but not documented here, such as a variety of client/server applications for purposes such as storage or remote procedure calls (RPC). Table 2-3 summarizes the expected libfabric API usage for selected UET targeted applications.

Table 2-3 - UET Application Categories

Application Use Case	Summary of Expected API Usage
*CCL	• Reliable communication • Matching based on sender ID • Tagged messages using exact matching ○ With no guarantees about which tag matches if duplicate tags are used • Data transfers using RMA Write • Message and data ordering not required • Optional use of collective offload
*MPI	• Reliable communication • Matching based on sender ID • Tagged messages with wildcard matching and duplicated tags • Send/receive untagged messages • Data transfers using RMA Read and RMA Write • Tagged message ordering is required • Data ordering is not required • Optional support for atomic operations • Optional use of collective offload
SHMEM	• Reliable communication • Message and data ordering not required • Data transfers using RMA Read, RMA Write, and atomics • Optional use of collective offload
UD	• Unreliable datagram communication • Send/receive MTU-sized messages • Message and data ordering not required
Load/Store	• Data transfer using write and read • In-order delivery of transactions • Optimized for small, efficient transfers—typically between two accelerators.

2.2.2 UET Profiles

UET defines multiple profiles that allow for tradeoffs between implementation complexity/cost and capabilities. Each UET profile corresponds to a set of required capabilities. The details of the capabilities required by each profile are defined in the UE Transport specification. The four UET profiles in order of functionality (least to most) are:

1. Unified Local
2. AI Base

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

3. AI Full
4. HPC

Profiles specify the minimum required set of functionality and features for support of the profile. UET vendors providing additional features beyond the profile specification MUST provide a means of selectively enabling and disabling those features.

Libfabric provides flexibility by making a broad set of parameters available to applications. These parameters are typically carried in libfabric structures. UET places constraints on the set of parameters that MUST be supported. Table 2-4 summarizes the parameters that MUST be supported for each profile. Parameters not shown have vendor-specific values. Unified Local is not targeted at libfabric applications.

Table 2-4 - Per-Profile Libfabric Parameter Requirements

Structure	Field	Value(s)	Required			Notes
			AI BASE	AI FULL	HPC	
fi_info						
	caps	FI_LOCAL_COMM	Y	Y	Y	libfabric capabilities
		FI_REMOTE_COMM	Y	Y	Y	
		FI_MSG	Y	Y	Y	
		FI_SEND	Y	Y	Y	
		FI_RECV	Y	Y	Y	
		FI_TAGGED (exact match)[1]		Y	Y	See footnote 1
		FI_TAGGED (wildcard)[1]			Y	
		FI_DIRECTED_RECV[2]			Y	See footnote 2
		FI_TAGGED_DIRECTED_RECV[2]			Y	
		FI_SELECTIVE_COMPLETION	Y	Y	Y	
		FI_SHARED_AV	Y	Y	Y	
		FI_RMA	Y	Y	Y	
		FI_WRITE	Y	Y	Y	
		FI_REMOTE_WRITE	Y	Y	Y	
		FI_READ		Y	Y	
		FI_REMOTE_READ		Y	Y	
		FI_RMA_EVENT	Y	Y	Y	See footnote 3
		FI_ATOMIC (non-fetching)[3]	Y	Y	Y	
		FI_ATOMIC (fetching)[3]		Y	Y	
		FI_HMEM	Y	Y	Y	
		FI_FENCE	Y	Y	Y	
	addr_format	FI_ADDR_UET	Y	Y	Y	New for UET
fi_fabric_attr						
	name	UET	Y	Y	Y	
	prov_name	Vendor specific				
fi_domain_attr						
	name	Vendor specific				Usually in the form of [vendor]_[instance]
	threading	FI_THREAD_SAFE	Y	Y	Y	Support for other values is vendor-specific
	mr_mode	Provider MAY use FI_MR_ENDPOINT				
	mr_key_size	<= 6	Y	Y	Y	See Section 2.2.5.3.4.1

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Structure	Field	Value(s)	Required			Notes
			AI BASE	AI FULL	HPC	
	cq_data_size	8	Y	Y	Y	
	max_ep_tx_ctx	>= 1	Y	Y	Y	
	max_ep_rx_ctx[4]	>= 1	Y	Y	Y	See footnote 4
	max_ep_stx_ctx	0	Y	Y	Y	No shared transmit queues
	max_ep_srx_ctx	0	Y	Y	Y	No shared receive queues
	mr_iov_limit	>= 1	Y	Y	Y	
	mr_cnt	>= 8	Y	Y	Y	Floor is subject to overall availability of MR resources if MRs are a shared resource
	tclass	FI_TC_BEST_EFFORT FI_TC_UNSPEC DSCP value	Y Y Y	Y Y Y	Y Y Y	Maps to default TC Maps to default TC
	auth_key_size	0 or 3 or FI_AV_AUTH_KEY	Y	Y	Y Y	See Section 2.2.4
	max_ep_auth_key	>= 1			Y	Valid only in conjunction with *auth_key_size* = FI_AV_AUTH_KEY
fi_ep_attr						
	type	FI_EP_RDM FI_EP_DGRAM	Y Y	Y Y	Y Y	Reliable datagram Unreliable datagram
	protocol	FI_PROTO_UET	Y	Y	Y	New for UET
	protocol_version	FI_VERSION (1, 0)	Y	Y	Y	
	max_msg_size		Max PDU[5]	4GB-1	4GB-1	See footnote 5
	max_order_raw_size	0	Y	Y	Y	No data ordering
	max_order_war_size	0	Y	Y	Y	No data ordering
	max_order_waw_size	0	Y	Y	Y	No data ordering
	tx_ctx_cnt	>= 1	Y	Y	Y	
	rx_ctx_cnt	>= 1	Y	Y	Y	See footnote 4
	auth_key	JobID when *auth_key_size* is 3	Y	Y	Y	See Section 2.2.4
fi_tx_attr						
	op_flags	FI_COMPLETION FI_INJECT FI_INJECT_COMPLETE FI_TRANSMIT_COMPLETE FI_DELIVERY_COMPLETE[6]	Y Y Y Y Y	Y Y Y Y Y	Y Y Y Y Y	 Applies only to reliable endpoints
	msg_order	0 FI_ORDER_SAS Any combination of defined Message ordering modes	Y Y	Y Y Y	Y Y Y	No message ordering
	comp_order	0	Y	Y	Y	Field being deprecated by libfabric
	inject_size	Vendor specific				
	iov_limit	>= 1	Y	Y	Y	
	rma_iov_limit	>= 1	Y	Y	Y	
fi_rx_attr						
	op_flags	FI_COMPLETION	Y	Y	Y	
	msg_order	0	Y	Y	Y	No message order

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Structure	Field	Value(s)	Required			Notes
			AI BASE	AI FULL	HPC	
	comp_order	0	Y	Y	Y	Field being deprecated by libfabric
fid_nic	All fields supported by the platform should be set		Y	Y	Y	

Note:

1. Exact match versus wildcard tags are distinguished by the 'ignore' parameter of the *fi_tagged()* APIs; see Section 2.2.5.4.2.
2. The level of required support for FI_DIRECTED_RECV/FI_TAGGED_DIECTED_RECV is limited to the authorization requirements specified in Section 2.2.4.3 and the initiator ID matching requirements specified in Section 2.2.5.4.2.1.
3. A libfabric provider indicates the atomic operations that it supports via *fi_atomic()* APIs; fetching and non-fetching atomics are different operations; see Section 2.2.5.4.4.
4. If scalable receive queues are supported, consecutive indices must be allocated for the endpoint with one index for each receive queue. The allocation of consecutive indices can be accomplished using the service configuration file described in Table 2-7 or by the provisioning system as described in Section 2.2.5.3.5.1.
5. Max PDU is the maximum payload of a single frame (i.e., UET payload) per definition in UE SES specification.

Libfabric v2.0 has added the capability to separately set the maximum message size for RMA operations via the *fi_setopt()* API. The UET requirements for all profiles are:

```
1 GB <= UET Maximum RMA Message Size <= (4GB – 1)
```

The per-profile requirements for completion counters are specified in Section 2.2.5.3.7.

Table 2-5 contains a mapping of the application use cases shown in Table 2-3 to UET profiles.

Table 2-5 - Application Use Case Mapping to UET Profiles

Application Use Case	UET Profile(s) with Optimal Use-Case Support
*CCL	Any of AI Base, AI Full, or HPC
*MPI	HPC
SHMEM	Either AI Full or HPC
UD	Any of AI Base, AI Full, or HPC

2.2.2.1 Profile Negotiation and Inter-Profile Interoperability

A logical priority, shown in Table 2-6, is associated with each profile to resolve conflicts when source and destination FEPs have overlapping profile support.

Table 2-6 - Profile Logical Priorities

Profile	Logical Priority
AI Base	0
AI Full	1
HPC	2

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

UET libfabric endpoint addresses, which are specified in Section 2.2.5.1, include a Fabric Endpoint Capabilities field that indicates the profiles supported by the fabric endpoint. If there is an overlap between the profiles supported by the source and destination FEPs, the common profile with the highest logical priority MUST be used for communication between the FEPs.

The AI Base profile is a subset of both the AI Full and HPC profiles. So, the AI Base profile can always be used. The HPC profile is not a superset of the AI Full profile. However, deferrable send, which is discussed in Section 2.2.5.4.1.2, is the only AI Full capability that is not supported by the HPC profile. Thus, the AI Full profile and the HPC profile can interoperate subject to the following restrictions:

- The operations MUST be limited to those supported by the AI Full profile,
- The HPC profile MUST treat a deferrable send operation the same as a send operation, and
- The HPC profile MUST treat a deferrable tagged send operation the same as a tagged send operation.

2.2.3 Configuration Information

The libfabric UET provider configuration parameters are shown in Table 2-7.

Table 2-7 - Libfabric UET Provider Configuration Parameters

Parameter	Data Type	Description
UET_PROVIDER_SERVICE_PATH	string	Environment variable specifying the path name for an optional service configuration file containing user-defined service name strings and associated Indices. The service name strings can be used as a parameter to the *fi_getinfo()* API to allocate an index or a range of indices for the service. The file format is simply one service entry per line formatted as follows: service_name start_index num_indices The service_name is a string with a maximum length of 64 characters. The start_index and num_indices are integers, where (start_index + num_indices] <= 4096.
UET_PROVIDER_MSG_RENDEZVOUS_SIZE	uint32_t	Messages with sizes >= UET_PROVIDER_MSG_RENDEZVOUS_SIZE bytes SHOULD be sent with a rendezvous protocol.
UET_PROVIDER_TAG_RENDEZVOUS_SIZE	uint32_t	Tagged messages with sizes >= UET_PROVIDER_TAG_RENDEZVOUS_SIZE bytes SHOULD be sent with a rendezvous protocol.
UET_PROVIDER_MAX_EAGER_SIZE	uint32_t	Maximum amount of eager data in bytes for the rendezvous protocol. The SES specification defines eager as follows: The initial rendezvous request MAY have an “eager” portion of data that is transferred with it. Eager transfers are payload transfers before the buffer for the transfer has been identified at the target.
UET_PROVIDER_DEF_DATA_TC	uint8_t	Optional override of default DSCP codepoint for data traffic class. See Section 2.2.7.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Parameter	Data Type	Description
UET_PROVIDER_FALLBACK_JOBID_SUPPORT	boolean	Configuration support for assigning endpoints to a fallback JobID if one cannot be obtained from the job provisioning system. See Section 2.2.4.2.
UET_PROVIDER_INITIATOR_ID	uint32_t	Environment variable containing the initiator ID for endpoints configured through the fallback JobID mechanism described in Section 2.2.4.2.
UET_IM_URL_PRIMARY	string	An environment variable specifying a URL that can be used to communicate with the primary INC manager.
UET_IM_URL_SECONDARY	string	An environment variable specifying a URL that can be used to communicate with the secondary INC manager.

2.2.4 JobIDs

The JobID is part of UET addressing, is carried in the SES header as the **ses.JobID** field, and is used for authorization. The JobID MUST be assigned by a privileged entity. A privileged entity MUST provide the assigned JobID to the provider kernel driver as described in either Section 2.2.4.1 or Section 2.2.5.3.5.1. After assignment, the JobID MAY be passed to libfabric user-space software, but JobIDs presented to the UET provider by libfabric user-space software MUST be validated within a hardware context. Multiple methods for assigning JobIDs are specified:

1. JobID assignment at job initialization time
2. JobID assignment at libfabric endpoint creation time
3. Fallback JobID assignment

2.2.4.1 JobID Assignment at Job Initialization Time

A privileged entity, such as a job launcher, MAY assign JobIDs at job initialization time. When JobID assignment is performed at job initialization time, the privileged entity MUST configure {OS Process ID, Service Name} => {JobID} mappings via the UET Control API with the provider kernel driver, as illustrated in Figure 2-4. The privileged entity MAY also configure components of the local UET address and security bindings as part of the mapping.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

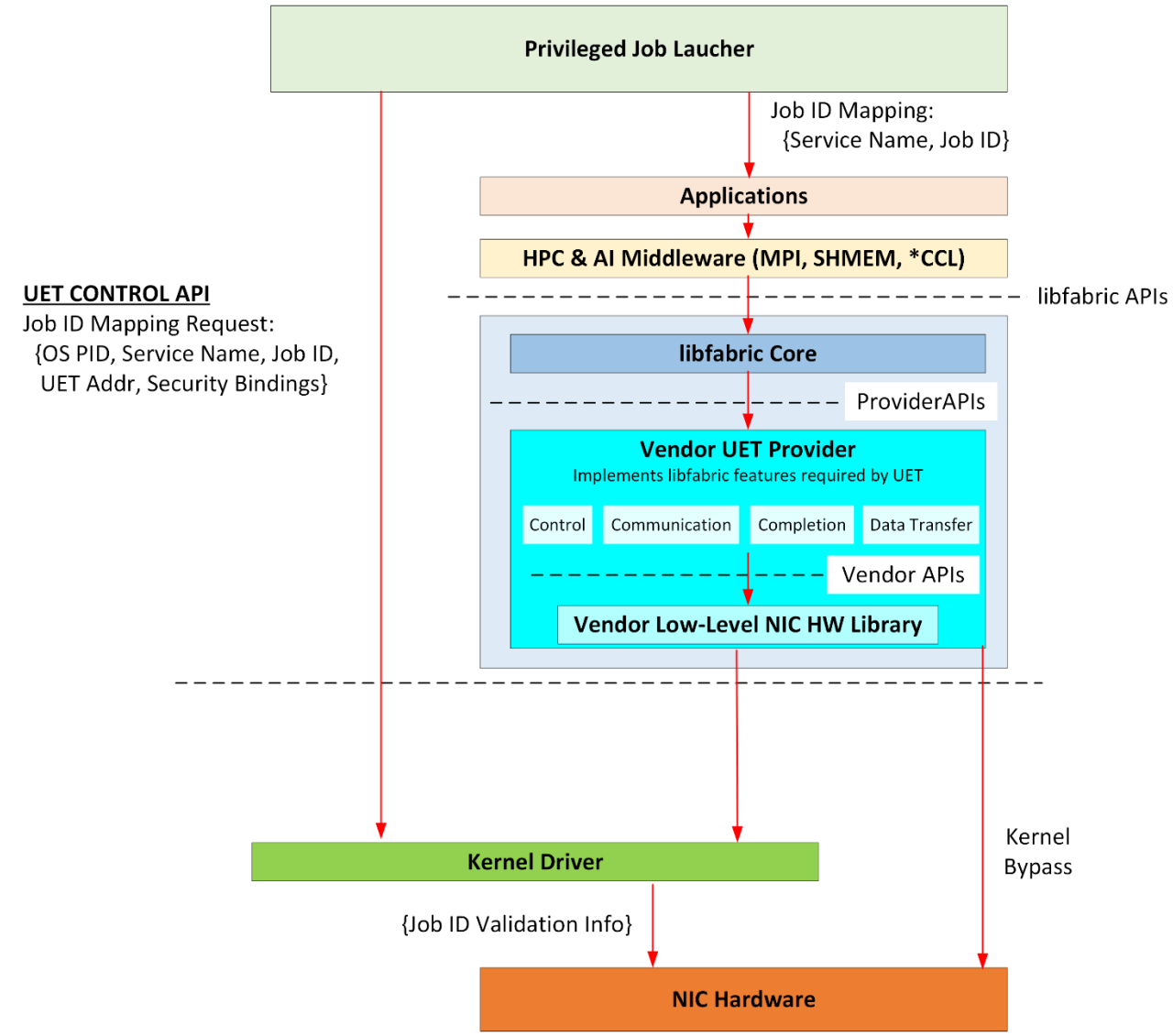


Figure 2-4 - JobID Assignment at Job Initialization Time

The UET Control API JobID mapping parameters are specified in Table 2-8.

Table 2-8 - UET Control API JobID Mapping Parameters

Parameter Name	Size (bits)	Description
Flags	8	Valid Flags Bit 0:0 Indicates validity of UET Address Field 0 => UET Address field is NOT valid 1 => UET Address field is valid Bit 1:1 Indicates validity of security bindings field 0 => Security bindings field is NOT valid 1 => Security bindings field is valid Bits 2:7 Reserved, MUST be 0
JobID	24	JobID assigned
OS PID	32	Operating system process ID that JobID is assigned to

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Parameter Name	Size (bits)	Description
Service Name	136	Null-terminated string identifying a service that JobID is assigned to. A NULL string indicates the JobID is assigned to all services of the process.
UET Address	See Table 2-10	MAY be used to assign components of the local UET address (e.g., PIDonFEP or Initiator ID) The UET address format is specified in Table 2-10
Security Bindings	See Table 2-33	MAY be used to assign security bindings Security binding parameters are specified in Table 2-33

A 'C' structure representation of the UET Control API JobID mapping request is shown in Figure 2-5.

```
#define UET_CTRL_FLAG_ADDR_V (1 << 0)  /* uet address field valid */
#define UET_CTRL_FLAG_SEC_V (1 << 1)   /* security bindings field valid */
#define UET_MAX_SERVICE_NAME_CHARS 64  /* max len of service name str */

struct uet_ctrl_job_id_map_req {
       uint8_t flags;
       uint32_t job_id;
       uint32_t os_pid;
 char service[UET_MAX_SERVICE_NAME_CHARS+1];
 struct uet_addr uet_addr;
 struct uet_sec_bindings sec_bindings;
};
```

Figure 2-5 - UET Control API JobID Mapping Request Structure

Security bindings MAY be assigned for the entire JobID mapping using the *sec_bindings* field of the *uet_ctrl_job_id_map_req* structure. Alternatively, in the Linux implementation of the UET Control API (see Section 2.2.11), security bindings MAY be assigned for each allocated resource index using a series of UET_NL_ATTR_SEC_BINDINGS Netlink attributes.

JobID mappings MAY be removed using the UET Control API JobID unmapping request. A 'C' structure representation of the UET Control API JobID unmapping request is shown in Figure 2-6.

```
struct uet_ctrl_job_id_unmap_req {
      uint8_t flags;
      uint32_t job_id;
      uint32_t os_pid;
      char service[UET_MAX_SERVICE_NAME_CHARS+1];
struct uet_addr uet_addr;
};
```

Figure 2-6 - UET Control API JobID Unmapping Request Structure

More than one JobID MAY be assigned to the same {OS PID, Service Name}. When multiple JobIDs are assigned to the same {OS PID, Service Name}, the libfabric FI_AV_AUTH_KEY capability MUST be used to select the JobID associated with a data transfer operation. As an example, multiple JobIDs allow a single libfabric endpoint of a server to communicate with multiple client jobs. Another use case is when multiple jobs need to communicate with one another.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

When JobIDs are configured at job initialization time by a privileged entity, the provider kernel driver MUST maintain the configured JobID mappings for subsequent use. The privileged entity MAY also pass the JobIDs to user-space applications for use as parameters to subsequent libfabric API calls.

A UET provider MUST support at least one JobID per libfabric endpoint and MAY support multiple. When a single JobID per libfabric endpoint is provided, the JobID MAY be carried in the libfabric *auth_key* attribute (the FI_AV_AUTH_KEY capability MAY also be used to support a single JobID per endpoint). When multiple JobIDs per libfabric endpoint are supported, the JobIDs MUST be inserted into the address vector bound to the endpoint using the *fi_av_insert_auth_key()* API (i.e., the JobIDs are inserted as authorization keys). The inserted JobIDs MAY then be used for:

- Authorization of posted receive buffers,
- Authorization of registered memory regions, and
- Selection of the JobID used for message transmission operations.

When the *auth_key_size* attribute is set to FI_AV_AUTH_KEY, all authorization keys are associated with the address vector. The *auth_key_size* MUST be set to FI_AV_AUTH_KEY when multiple JobIDs are supported. Otherwise, the *auth_key_size* MUST be set to either 0 or 3. When the *auth_key_size* is set to 0, the UET provider supplies the assigned JobID on behalf of the user. When the *auth_key_size* is set to 3, the JobID MUST be carried in the *auth_key* attribute. The *max_ep_auth_key* domain attribute indicates the maximum number of authorization keys that are supported per libfabric endpoint.

For notation convenience, the following terms are defined:

- Indirect JobID method (*auth_key_size* = 0) —this is the default behavior
- Direct JobID method (*auth_key_size* = 3)
- AV JobID method (*auth_key_size* = FI_AV_AUTH_KEY)

The HPC profile MUST support the AV JobID method.

The AV JobID method MUST support the *fi_av_insert_auth_key* and the *fi_av_lookup_auth_key* libfabric APIs.

JobID assignment MAY also occur when the libfabric endpoint is created as specified in Section 2.2.5.3.5.1 and illustrated in Figure 2-8.

2.2.4.2 Fallback JobID Assignment

The provider configuration may enable fallback JobID assignment. In this mode, if an endpoint's JobID cannot be determined through the previous methods it is assigned a fallback JobID. The value of fallback JobID is 16777215. When fallback JobID support is enabled, the Initiator ID of the endpoint is sourced from the UET_PROVIDER_INITIATOR_ID environment variable. If the environment variable is unset, endpoint creation MUST fail.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

2.2.4.3 Authorization

The authorization procedures described in this section MUST be implemented for absolute addressing mode. The procedures are not required for relative addressing mode because the JobID is a component of the addressing information that is used to locate the buffer and memory region. Please refer to the SES specification for a description of the absolute and relative addressing modes.

Authorization allows access to receive buffers and registered memory regions to be controlled. The JobID is used for authorization of access to receive buffers and registered memory regions. The specific requirements defined in the SES specification are:

- Implementations MUST allow an option for buffers and memory regions to be exposed for exactly one JobID.
- Implementations MUST allow an option for buffers and memory regions to be exposed for "any" JobID.

The JobID associated with a receive buffer is determined when the buffer is posted using one of the *fi_msg()* APIs. The JobID associated with a memory region is determined when the memory region is registered using the fi_mr APIs. The procedures for determining the JobID that is associated with a posted receive buffer or registered memory region are specified in the following subsections.

Authorization is performed by checking that the **ses.JobID** field in the SES header is allowed to access the targeted receive buffer or registered memory region.

2.2.4.3.1 Untagged Message Buffer Authorization

This section covers how the authorization requirements are managed for untagged message buffers.

The JobID for buffer authorization is obtained differently based on whether the JobID method used is the indirect, direct, or AV JobID method.

With the indirect JobID method, the UET provider supplies the assigned JobID, and untagged message buffer access MUST be authorized only for operations carrying that **ses.JobID** field in the SES header.

With the direct JobID method, the JobID for untagged message buffer authorization is taken directly from the *auth_key* attribute of the libfabric endpoint, and buffer access MUST be authorized only for operations carrying that **ses.JobID** field in the SES header.

With the AV JobID method, the JobID for untagged message buffer authorization is determined using the *src_addr* parameter of the *fi_recvmsg()* API when the FI_AV_AUTH_KEY flag is set. In this case, the *src_addr* is treated as a source authorization key returned by the *fi_av_insert_auth_key()* API when the JobID was inserted into the address vector bound to the libfabric endpoint. Buffer access MUST be authorized only for operations carrying this **ses.JobID** field in the SES header. In all other cases (e.g., *fi_recv()* API used, *fi_recvv()* API used, FI_AV_AUTH_KEY not set):

- The posted buffer MUST be for any JobID, and

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

- Components of the UET address referenced by the *src_addr* parameter MAY be used to direct received messages to specific buffers according to the FI_DIRECTED_RECV semantics.

2.2.4.3.2 Tagged Message Buffer Authorization

The tagged message buffer authorization requirements are the same as specified for untagged message buffers in Section 2.2.4.3.1.

2.2.4.3.3 Memory Region Authorization

This section covers how the authorization requirements are managed for registered memory regions.

To associate a registered memory region with a JobID, the *fi_mr_regattr()* API MUST be used.

When the *auth_key_size* field of the *attr* parameter to *fi_mr_regattr()* is 0, the memory region MUST be for any JobID.

When the *auth_key_size* field of the *attr* parameter to *fi_mr_regattr()* is 3, the memory region MUST be associated with the JobID carried in the *auth_key* attribute of the libfabric endpoint.

When the *auth_key_size* field of the *attr* parameter to *fi_mr_regattr()* is set to FI_AV_AUTH_KEY:

- The *auth_key* field of the *attr* parameter to *fi_mr_regattr()* MUST point to a user-defined struct *fi_mr_auth_key* that specifies:
  - An address vector, and
  - An address that has been inserted into the address vector.
- The memory region MUST be associated with the JobID represented by the authorization key of the address specified in the *fi_mr_auth_key* struct.

In all other cases (e.g., use of *fi_mr()* registration APIs other than *fi_mr_regattr()*), the memory region MUST be for any JobID.

When a registered memory region is associated with a JobID, access MUST be authorized only for operations carrying that **ses.JobID** field in the SES header.

2.2.4.4 JobID Selection for Data Transmission Operations

Using the indirect JobID method, the provider references the assigned **ses.JobID** field within the SES header to facilitate data transfer operations.

In the direct JobID method, the **ses.JobID** field within the SES header of data transfer operations MUST be taken directly from the *auth_key* attribute of the libfabric endpoint.

With the AV JobID method, the **ses.JobID** field within the SES header of data transfer operations MUST be determined from the *fi_addr_t dest_addr* parameter of the libfabric API (i.e., the authorization key associated with the destination address when it was inserted into the address vector).

2.2.5 Libfabric APIs

This specification targets the libfabric v2.0 release.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Libfabric provides many APIs that are currently documented in the groups summarized by Table 2-9.

Table 2-9 - Libfabric API Groups

API Group Name	Description
fi_atomic	Remote atomic operations
fi_av	Address vector operations
fi_av_set	Address vector set operations
fi_cm	Connection management operations
fi_cntr	Completion and event counter operations
fi_collective	Collective operations
fi_control	Fabric resource operations
fi_cq	Completion queue operations
fi_domain	Fabric domain operations
fi_endpoint	Fabric endpoint operations
fi_eq	Event queue operations
fi_errno	Fabric error operations
fi_fabric	Fabric network operations
fi_getinfo	Fabric discovery operations
fi_msg	Message data transfer operations
fi_mr	Memory region operations
fi_peer	Provider to provider operations
fi_poll	Polling and wait set operations (being deprecated by libfabric)
fi_provider	Provider operations
fi_rma	Remote memory access operations
fi_tagged	Tagged data transfer operations
fi_version	Library interface version operations

Each individual API is not explicitly covered in this specification. Instead:

- Detailed coverage is limited to the key APIs in each of the four main API categories identified in Section 2.2 (i.e., control, communication, completion, and data transfer).
- Requirements for other libfabric APIs are specified in Section 2.2.5.5.
- Unless explicitly stated otherwise, all libfabric APIs SHOULD be supported.

Some of the libfabric APIs also have parameters that were not covered by the UET profile parameter requirements specified in Section 2.2.2, and a subset of those parameters require special handling by a UET provider. An effort has been made to identify such parameters and specify the required handling.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Figure 2-7 provides a top-level depiction of relationships between key libfabric objects and the associated APIs. The purpose of the diagram is to provide context that promotes easier understanding of subsequent text in this specification. A typical API flow for creating the objects is summarized in the following bullets:

- The *fi_getinfo()* API is used to identify locally available providers and their capabilities.
    - *fi_getinfo()* returns a list of *fi_info* structures.
- The application selects the *fi_info* structure associated with the desired provider and uses the *fi_info* structure as a parameter to the *fi_fabric()* API, which creates a fabric object.
    - In libfabric, a fabric represents a network.
- The *fi_eq_open()* API is used to create an event queue that is bound to the fabric.
    - Event queues are used to report the completion of asynchronous control operations and events.
- The *fi_domain()* API is used to create a domain object that is bound to the fabric.
    - In libfabric, a domain represents a NIC.
- The *fi_endpoint()* API is used to create an endpoint object that is bound to the domain.
    - In libfabric, an endpoint represents a transport-level communication portal.
- The *fi_cq_open()* API is used to create one or more completion queues.
    - Completion resources (i.e., completion queues or completion counters) are used to report the results of submitted data transfer operations.
- The *fi_cntr_open()* API is optionally used to create one or more completion counters.
- The *fi_ep_bind()* API is used to bind the resource to the endpoint, such as event queues, completion queues, completion counters, address vectors, or shared transmit/receive contexts.
- The *fi_mr_reg()* API is used to create a memory region that is bound to the domain.
- The *fi_mr_bind()* API is used to bind the memory region to the endpoint (instead of the domain).
    - UET requires memory regions be bound to endpoints as specified in Table 2-4.
- The *fi_av_open()* API is used to create an address vector under the domain, to be bound to the endpoint.
    - Address vectors are used to efficiently represent destination endpoints.
- The *fi_av_insert()* API is used to insert one or more entries in the address vector.
- The data transfer APIs are be used for endpoint communication.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

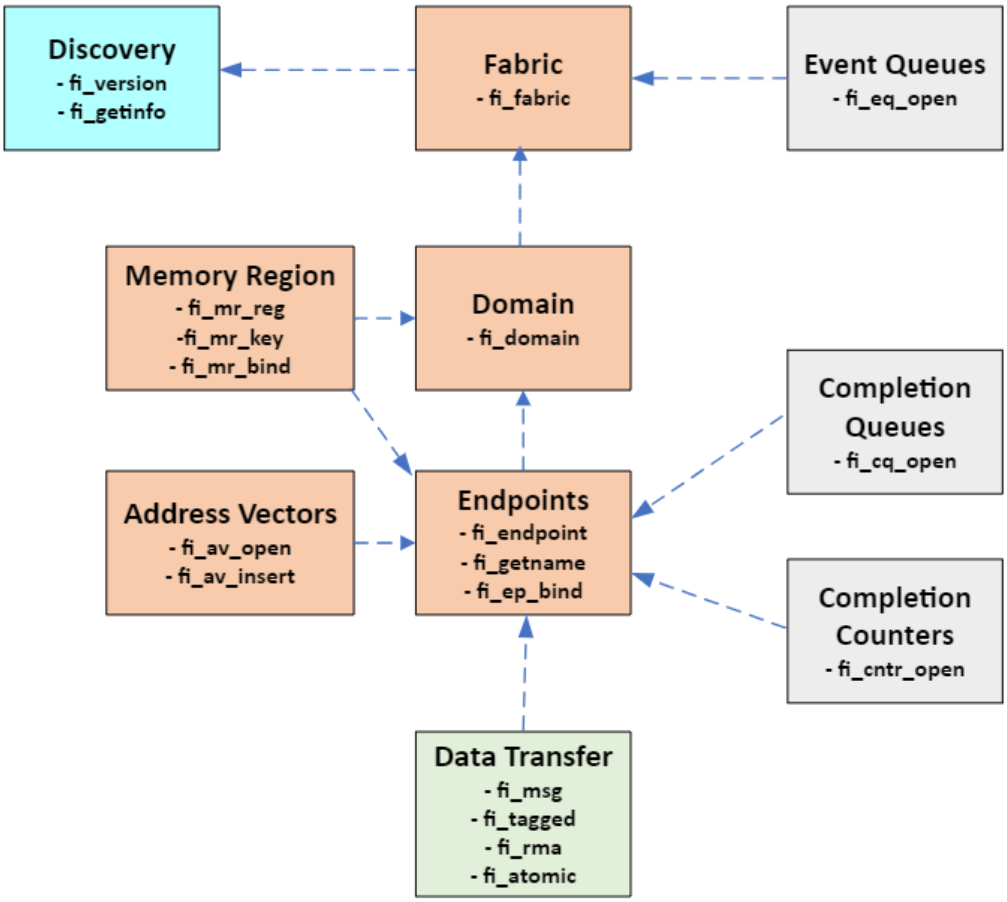


Figure 2-7 - Key Libfabric Objects and Associated APIs

2.2.5.1 Libfabric Addressing

Libfabric provides flexible endpoint addressing, where the available address formats are defined by an enumeration. A new value, FI_ADDR_UET, is added to the enumeration to identify the UET address format. Support for representation of UET libfabric endpoint addresses in the FI_ADDR_STR format is NOT required.

The components of a UET libfabric endpoint address are shown in Table 2-10.

Table 2-10 - UET Libfabric Endpoint Address

Field Name	Size (bits)	Description
Version	8	Version number that identifies the format of the address • The fields in this table are associated with version 0

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Field Name	Size (bits)	Description
Flags	16	Valid flags, see footnote 1 Bit 0:0 Indicates validity of fabric endpoint capabilities field 0 => Fabric endpoint capabilities field is NOT valid 1 => Fabric endpoint capabilities field is valid Bit 1:1 Indicates validity of fabric address fields (fabric address type and fabric address) 0 => Fabric address fields are NOT valid 1 => Fabric address fields are valid Bit 2:2 Indicates validity of PIDonFEP field 0 => PIDonFEP field is NOT valid 1 => PIDonFEP field is valid Bit 3:3 Indicates validity of index fields (start resource index and num resource indices) 0 => Resource Index fields are NOT valid 1 => Resource Index fields are valid Bit 4:4 Indicates validity of Initiator ID field 0 => Initiator ID field is NOT valid 1 => Initiator ID field is valid <u>Other Flags</u> Bit 5:5 Indicates whether relative or absolute address mode is used 0 => Relative addressing 1 => Absolute addressing Bit 6:6 Fabric address type, IPv4 or IPv6 0 => IPv4 1 => IPv6 Bit 7:7 Indicates if maximum message size is limited to MTU 0 => Maximum message size not limited to MTU 1 => Maximum message size = MTU Bits 8:15 Reserved, MUST be 0
Fabric Endpoint Capabilities	16	Bit 0:0 Indicates support for AI Base profile 0 => AI Base profile NOT supported 1 => AI Base profile supported Bit 1:1 Indicates support for AI Full profile 0 => AI Full profile NOT supported 1 => AI Full profile supported Bit 2:2 Indicates support for HPC profile 0 => HPC profile NOT supported 1 => HPC profile supported Bits 3:6 Reserved, MUST be 0 Bit 7: 7 Indicates support for optimized non-matching SES header 0 => Optimized non-matching SES header NOT supported 1 => Optimized non-matching SES header supported Bits 8:15 Reserved, MUST be 0
PIDonFEP	16	Process ID in context of fabric endpoint Meaning depends on the address mode
Fabric Address	128	IP address
Start Resource Index	12	Service Identifier

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Field Name	Size (bits)	Description
		• Index 0 is reserved for a UET provider-to-provider control channel o No standardized provider-to-provider control channel operations have been defined o The provider-to-provider control channel MAY be used for vendor-specific operations
Num Resource Indices	12	Number of sequential indices assigned to the service
Initiator ID	32	Initiator identifier (as defined in the SES specification)

Note:
1. The valid flags enable requests for specific components of a UET address on the *fi_getinfo()* API (see Section 2.2.5.2.2) and the *fi_endpoint()* API (see Section 2.2.5.3.5).

Informative Text:
The Resource Index field of the UET address enables a multi-threaded application supporting multiple services to open a libfabric endpoint for each service. Each service gets a unique index that is used to differentiate the endpoints.

Libfabric endpoint addresses are allocated when libfabric endpoints are opened, as discussed in Section 2.2.5.3.

A 'C' structure representation of a UET Libfabric Endpoint Address is shown below.

```
#define UET_ADDR_FLAG_FEP_CAP_V    (1 << 0) /* FEP capabilities valid flag */
#define UET_ADDR_FLAG_FA_V         (1 << 1) /* fabric address valid flag */
#define UET_ADDR_FLAG_PID_V        (1 << 2) /* PIDonFEP valid flag */
#define UET_ADDR_FLAG_RI_V         (1 << 3) /* resource index valid flag */
#define UET_ADDR_FLAG_INI_V        (1 << 4) /* initiator id valid flag */
#define UET_ADDR_FLAG_ABS_MODE     (1 << 5) /* absolute address mode */
#define UET_ADDR_FLAG_REL_MODE     (0 << 5) /* relative address mode */
#define UET_ADDR_FLAG_IPV6         (1 << 6) /* IPv6 fabric address type */
#define UET_ADDR_FLAG_IPV4         (0 << 6) /* IPv4 fabric address type */
#define UET_ADDR_FLAG_MTU_MSG_SIZE (1 << 7) /* max message size is MTU */

#define UET_ADDR_FEP_AI_MIN        (1 << 0) /* AI base profile supported */
#define UET_ADDR_FEP_AI_FULL       (1 << 1) /* AI full profile supported */
#define UET_ADDR_FEP_HPC           (1 << 2) /* HPC profile supported */

#define UET_ADDR_FEP_OPT_NM_SEM    (1 << 7) /* non-matching hdr supported */

#define UET_ADDR_IPV6_ADDR_OCTETS 16

struct uet_fa { /* fabric address */
     union {
          uint32_t v4;
          uint8_t v6[UET_ADDR_IPV6_ADDR_OCTETS];
     }
};

struct uet_addr { /* UET address */
     uint8_t ver;
```


Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

```
    uint8_t reserved; /* for alignment */
    uint16_t flags;
    uint16_t fep_cap;
    uint16_t pid_on_fep;
    struct uet_fa fa;
    uint16_t start_resource_index;
    uint16_t num_resource_indices;
    uint32_t initiator_id;
};
```

2.2.5.2 Discovery APIs

Discovery APIs are used to discover the available libfabric communication services.

The following two discovery APIs are discussed in this section:

```
uint32_t fi_version(void);
```

<https://ofiwg.github.io/libfabric/main/man/fi_version.3.html>

```
int fi_getinfo(int version, const char *node, const char *service,uint64_t
        flags, const struct fi_info *hints, struct fi_info **info);
```

<https://ofiwg.github.io/libfabric/main/man/fi_getinfo.3.html>

2.2.5.2.1 fi_version() API

The *fi_version()* API is used to discover the libfabric version. The API returns an encoded version, which can be decoded using the *FI_MAJOR ()* and *FI_MINOR ()* macros. The information MAY be used to ensure the libfabric version meets the minimum that the application requires. For UET, this minimum should be Major Version 2 and Minor Version 0.

2.2.5.2.2 fi_getinfo() API

The *fi_getinfo()* API is used to identify locally available providers and their capabilities. The parameters to *fi_getinfo()* in Table 2-11 merit additional description:

Table 2-11 - fi_getinfo() Parameters

Parameter	Description
node	The node parameter is usually set to NULL and ignored by the provider, but MAY be used as a filter to limit the returned providers. For UET, if the node parameter is non-NULL, it MUST point to a UET address. If the node parameter is non-NULL and the FI_SOURCE flag is set, UET providers SHOULD filter the returned info based on the fields of the UET address that are valid (see Table 2-10). If the node parameter is non-NULL and the FI_SOURCE flag is not set, UET providers SHOULD NOT return info.
service	Previously, the service parameter has usually been set to NULL and ignored by the provider. For UET, the service parameter MAY contain a service name string. The string MAY be a pre-defined value associated with a particular service supported by UET or a user-defined value read by the provider from an optional service configuration file as described in Section 2.2.3. The set of pre-defined service strings is shown in Table 2-12. If a pre-defined service string value appears in the service configuration file, the

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Parameter	Description
	mapping defined in the service configuration MUST take precedence. When the service parameter is specified, UET providers MUST filter the returned providers based on the service parameter, and only return providers that support the specified service.
flags	UET providers MUST support the following flag values: • FI_SOURCE
hints->addr_format	Indicates the format of addresses referenced by the fabric interfaces and data structures. If the value is not FI_ADDR_UET, UET providers MUST NOT return info.
hints->src_addr	This parameter MAY be used as a filter to limit the returned providers in a manner like the case where the node parameter is non-NULL and the FI_SOURCE flag is set. If hints->src_addr is non-NULL, the UET provider SHOULD filter the returned info based on the fields of the UET address that are valid.
hints->dst_addr	This parameter is intended as a filter to limit the returned providers. If the hints->dst_addr parameter is specified, UET providers SHOULD NOT return info.
(*info)->src_addr	Because full UET addresses are not available until the libfabric endpoint is opened, the components of the UET address that MUST be returned are limited to the following: • Version • Flags • Fabric address type • Fabric address • Start resource index ○ The returned start resource index is NOT required to be the actual start resource index, but the provider MUST be able to identify the service based on the returned start resource index value (i.e., when this *src_addr* is used as a parameter to the *fi_endpoint()* API). ○ If the service parameter is not specified, the start resource index is associated with the generic service. • Fabric endpoint capabilities
(*info)->dst_addr	The UET provider MUST NOT return a destination address.

Table 2-12 - Pre-Defined UET Service Names

Pre-Defined Service Name Strings	Reserved Starting Resource Index Value	Number of Resource Indices
“generic”	None (dynamically allocated)	1
“ccl”	1	5
“mpi”	6	5
“shmem”	11	5

Multiple resource indices MAY be assigned to a service. Scalable endpoints (created with the *fi_scalable_ep()* API) MUST assign a unique resource index to each rx context (see *fi_rx_context()* API). The UET address of an endpoint created with the *fi_endpoint()* API MUST be assigned a single

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

application-visible resource index (there MAY also be an additional resource index associated with the endpoint that is not application-visible but used as a provider-to-provider control channel).

2.2.5.3 Communication and Completion APIs

The communication and completion APIs are used to perform the setup required for data transfer and to report data transfer operation results.

The following communication and completion APIs are discussed in this section:

```
int fi_fabric(struct fi_fabric_attr *attr, struct fid_fabric **fabric,
        void *context);
```

<https://ofiwg.github.io/libfabric/main/man/fi_fabric.3.html>

```
int fi_eq_open(struct fid_fabric *fabric, struct fi_eq_attr *attr,
         struct fid_eq **eq, void *context);
```

<https://ofiwg.github.io/libfabric/main/man/fi_eq.3.html>

```
int fi_domain(struct fid_fabric *fabric, struct fi_info *info,
        struct fid_domain **domain, void *context);
```

<https://ofiwg.github.io/libfabric/main/man/fi_domain.3.html>

```
int fi_mr_reg(struct fid_domain *domain, const void *buf, size_t len,
        uint64_t access, uint64_t offset, uint64_t requested_key,
        uint64_t flags, struct fid_mr **mr, void *context);

uint64_t fi_mr_key(struct fid_mr *mr);
```

<https://ofiwg.github.io/libfabric/main/man/fi_mr.3.html>

```
int fi_endpoint(struct fid_domain *domain, struct fi_info *info,
          struct fid_ep **ep, void *context);

int fi_ep_bind(struct fid_ep *ep, struct fid *fid, uint64_t flags);
```

<https://ofiwg.github.io/libfabric/main/man/fi_endpoint.3.html>

```
int fi_getname(fid_t fid, void *addr, size_t *addrlen);
```

<https://ofiwg.github.io/libfabric/main/man/fi_cm.3.html>

```
int fi_cq_open(struct fid_domain *domain, struct fi_cq_attr *attr,
         struct fid_cq **cq, void *context);
```

<https://ofiwg.github.io/libfabric/main/man/fi_cq.3.html>

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

```
int fi_cntr_open(struct fid_domain *domain, struct fi_cntr_attr *attr,
           struct fid_cntr **cntr, void *context);
```

<https://ofiwg.github.io/libfabric/main/man/fi_cntr.3.html>

```
int fi_av_open(struct fid_domain *domain, struct fi_av_attr *attr,
         struct fid_av **av, void *context);

int fi_av_insert(struct fid_av *av, void *addr, size_t count,
           fi_addr_t *fi_addr, uint64_t flags, void *context);
```

<https://ofiwg.github.io/libfabric/main/man/fi_av.3.html>

2.2.5.3.1 fi_fabric() API

The *fi_fabric()* API is called to open a fabric network provider object. A fabric represents a collection of hardware and software resources that access a single physical or virtual network. A pointer to a fabric attributes structure is a parameter to *fi_fabric()*. The *fi_info* structure returned by the *fi_getinfo*() API contains a pointer to a fabric attributes structure.

2.2.5.3.2 fi_eq_open() API

The *fi_eq_open()* API is called to create a new event queue for the fabric. EQs are for control operations and are not for completion of data transfer operations such as sends and receives. EQs are used to collect and report the completion of asynchronous control operations and events. EQs are used for control events that are not directly associated with data transfer operations such as:

- Asynchronous completion of libfabric control API calls
    - Some libfabric control APIs support either synchronous or asynchronous operation
- Asynchronous error notification for problems with fabric resources such as completion queues or endpoints

EQs are typically implemented completely in software.

An EQ MAY also be bound to a domain using the fi_domain_bind API.

UET providers MUST support event queues. A minimum of one event queue per libfabric endpoint MUST be supported.

2.2.5.3.3 fi_domain() API

The *fi_domain()* API is called to open an access domain on the fabric. A domain is a logical connection into a fabric, often corresponding to a physical or virtual NIC. The *fi_domain()* API takes a pointer to a *fi_info* structure as a parameter. The *fi_info* structure returned by the *fi_getinfo()* API contains a pointer to a domain attributes structure.

2.2.5.3.4 fi_mr_reg() and fi_mr_key() APIs

A memory buffer MUST be registered with a resource domain before it can be used as the target of a remote RMA or atomic data transfer. Additionally, a fabric provider MAY require that data buffers be registered before being used in local transfers (e.g., the buffer is the source for a write operation or the

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

destination for a read operation). The *fi_mr_reg()* API is used to register a memory region on the domain. A memory region is bound to an endpoint using the *fi_mr_bind()* API.

The *fi_mr_key()* API is used to obtain the key that remote endpoints need to access a registered memory region. Libfabric offers options that enable the key for a memory region to be assigned by the application or the provider. A UET provider SHOULD support user-assigned keys and therefore SHOULD NOT require the FI_MR_PROV_KEY mode flag.

Informative Text:

Support for user-assigned keys can enable applications to avoid exchanging memory keys if the application uses a convention where the appropriate key values are well-known.

2.2.5.3.4.1 Memory Key Format

For interoperability, the format of the memory region key is standardized as specified in Table 2-13.

Table 2-13 - Memory Region Key Format

Field Name	Bit Location	Size (bits)	Description
IDEMPOTENT_SAFE	63	1	0 => Memory region MUST NOT be used as target of idempotent transport operations. 1 => Memory region MAY be used as target of idempotent transport operations (i.e., there are no completion counters bound to the memory region). Idempotent operations can improve the efficiency of the transport protocol.
OPTIMIZED	62	1	0 => Memory region does not support optimized non-matching headers (the optimized non-matching header format is defined in the UET SES specification; optimized refers to a header that is smaller than the standard SES request header). 1 => Memory region supports optimized non-matching headers.
RESERVED	56:61	6	Reserved for future UET definitions; MUST be 0.
VENDOR_SPECIFIC	48:55	8	MAY be used in vendor-specific manner for provider-assigned memory keys; MUST be 0 for user-assigned memory keys.
OPTIMIZED = 0			
RKEY	0:47	48	Memory key for MR.
OPTIMIZED = 1			
RESERVED	12:47	36	Reserved for future UET definitions; MUST be 0.
RKEY / INDEX	0:11	12	Index for MR.

The memory region key format shown in Table 2-13 MUST be used for both user-supplied keys and provider-supplied keys. The full 64-bit key format MUST be passed across the *fi_mr()* APIs. Bit mask definitions will be provided to assist in forming keys with the desired format. The size of the RKEY field is fixed at 48 bits, but the range of RKEY values that are supported MUST be based on the value of the

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

mr_key_size attribute in the *fi_domain_attr* structure. If *mr_key_size* is < 6, the unused most-significant bits MUST be 0.

The format of provider-supplied RKEYs is provider-specific (since the format is only locally significant at the assigning provider). The provider implementation MUST choose how the RKEY is used in identifying a registered memory region (e.g., the key MAY be used as a table index, part of hash lookup tuple, etc.). A UET provider MAY choose to partition the RKEY into a portion that carries the memory key and another portion that carries additional authentication information.

The RKEY values in the range [0..4095] merit additional discussion, since these values MAY be used in conjunction with the optimized non-matching SES header in some cases. The optimized non-matching SES header SHOULD be used when the criteria specified in Table 2-14 are satisfied. The RMA operations that the criteria specified in this section are applied to are associated with the libfabric APIs defined in the *fi_rma()* API Group <https://ofiwg.github.io/libfabric/v1.20.1/man/fi_rma.3.html>.

Table 2-14 - Criteria for Optimized Non-Matching SES Header for RMA Operations

Criteria for Use of Optimized Non-Matching SES Header for RMA Operations
OPTIMIZED bit is 1 in Memory region key format
RMA operation size <= MTU
FI_REMOTE_CQ_DATA is NOT set for the RMA operation

When the optimized non-matching SES header is used for RMA operations, the RKEY/INDEX value MUST be carried in the **ses.resource_index** field of the SES header.

The criteria that MUST be satisfied to use the small RMA SES header for RMA operations are shown in Table 2-15. The small RMA SES header SHOULD be used for RMA operations when the criteria specified in Table 2-15 are satisfied.

Table 2-15 - Criteria for Small RMA SES Header with RMA Operations

Criteria for Use of Small RMA SES Header for RMA Operations
RMA operation size <= MTU
FI_REMOTE_CQ_DATA is NOT set for the RMA operation

If the criteria for use of the optimized non-matching SES header is NOT satisfied and the criteria for use of the small RMA SES header is NOT satisfied, then the standard SES header MUST be used for RMA operations.

The criteria that MUST be satisfied to use the RUDI packet delivery mode for RMA operations are shown in Table 2-16. The RUDI packet delivery mode is optimized for idempotent operations and is described in the PDS reliability specification. If supported, the RUDI packet delivery mode SHOULD be used when the criteria specified in Table 2-16 are satisfied.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Table 2-16 - Criteria for RUDI Packet Delivery Mode with RMA Operations

Criteria for Use of RUDI PDC for RMA Operations
Libfabric endpoint is NOT configured for R/W message ordering
IDEMPOTENT_SAFE bit is 1 in memory region key format
Target supports use of RUDI packet delivery mode As indicated by the fabric endpoint capabilities field of the destination UET address

Both user-supplied and provider-supplied memory keys MAY be marked as IDEMPOTENT_SAFE and/or OPTIMIZED. A UET provider SHOULD honor the IDEMPOTENT_SAFE and OPTIMIZED bits in the *requested_key* parameter of the *fi_mr()* APIs even when FI_MR_PROV_KEY is configured. If the user attempts to bind a completion counter to a memory region marked as IDEMPOTENT_SAFE, a UET provider MUST fail the bind operation. If an OPTIMIZED provider-supplied key is requested but cannot be allocated, the provider SHOULD fall back to non-optimized operation.

Requirements for the scope of the memory keys are specified in Table 2-47.

Table 2-17 - Memory Key Scope Requirements

Memory Key Format	Scope in Relative Addressing Mode	Scope in Absolute Addressing Mode
Optimized	{FA, JobID, PIDonFEP}	{FA, PIDonFEP}
All Others	{FA, JobID, PIDonFEP, Resource Index}	{FA, PIDonFEP, Resource Index}

2.2.5.3.5 fi_endpoint() and fi_ep_bind() APIs

The *fi_endpoint()* API is used to open an active endpoint on the domain. Endpoints are transport-level communication portals. The data transfer interfaces are associated with active endpoints, which typically have transmit and receive queues. The *fi_endpoint()* API takes a *fi_info* structure as a parameter. The *fi_info* parameter is typically the *fi_info* structure returned by the *fi_getinfo()* API. The *fi_info* structure includes a source address field that is used as part of opening an active endpoint. The source address field MAY be used to request assignment of a specific UET address. Selected components of a UET address MAY be requested using the valid bits in the flags field of the UET address.

Libfabric endpoint address assignment MAY be a multi-stage process, where a portion of the endpoint address MAY be assigned at job initialization time or by the *fi_getinfo()* API; the remainder is assigned when the endpoint is opened. An application can learn its full UET address only by calling the *fi_getname()* API, which is described in Section 2.2.5.3.6.

The JobID MAY be assigned at job initialization time or when the endpoint is opened. The JobID MUST be programmed into the hardware by a privileged entity.

UET providers MUST support the following libfabric endpoint types:

- FI_EP_DGRAM
- FI_EP_RDM

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Each libfabric endpoint is of a single type, either FI_EP_DGRAM or type FI_EP_RDM.

The *fi_ep_bind()* API is used to associate resources with an endpoint, such as event queues, completion queues, completion counters, address vectors, or shared transmit/receive contexts.

2.2.5.3.5.1 UET Address Assignment Architecture

The following bullets summarize the procedure for assigning UET address fields:

- The UET provider makes a request to a kernel mode driver that is relayed to a privileged user-mode process, which is part of the provisioning system responsible for UET address assignment.
- The request contains information about the endpoint being opened.
- The privileged entity returns the needed address information.
- Additional information MAY also be returned such as JobID and security domain bindings.

An architecture diagram depicting the UET address assignment procedure is shown in Figure 2-8.

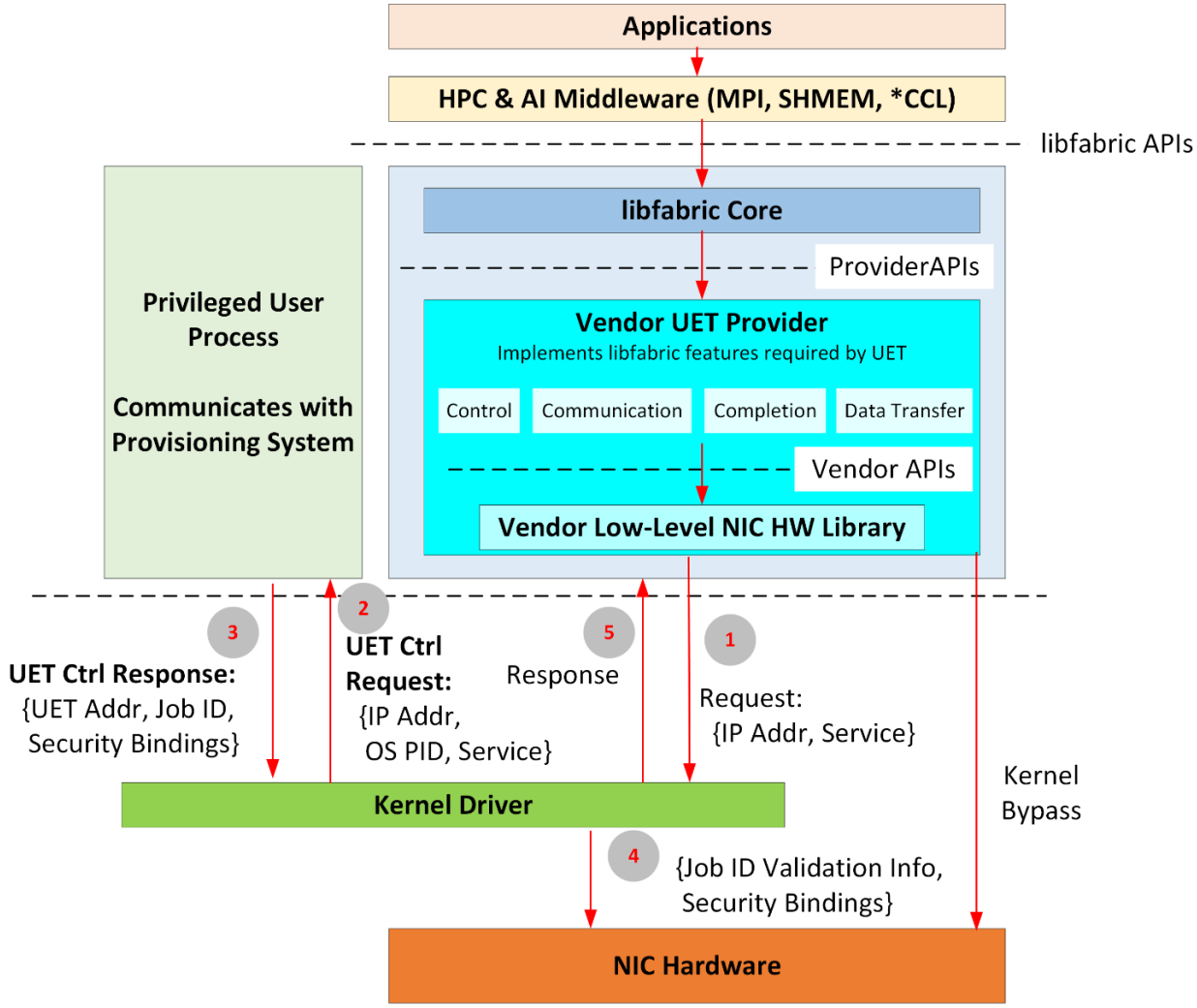


Figure 2-8 - UET Address Assignment Architecture

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

The steps shown in Figure 2-8 are:

1. UET provider makes an address assignment request to the kernel driver.
2. The kernel driver relays the request to the privileged user process via the UET Control API.
3. The privileged user process communicates with a provisioning system and returns the requested address information.
4. The kernel driver programs the JobID validation information and security bindings into the NIC hardware.
5. The kernel driver relays the response, without the security bindings, to the UET provider.

Requirements associated with this architecture are:

- A UET provider MUST send address assignment requests to the kernel driver.
- The address assignment request MUST contain the following information:
    - FEP IP address
        - The FEP IP address is a parameter of the address assignment request from the UET provider.
        - The FEP IP address parameter is used as part of the UET address.
        - The UET address assignment request does not configure IP addresses of NIC interfaces.
    - OS process ID
    - Service name
- In Linux implementations, the UET provider SHOULD use a Netlink interface for address assignment communication with the kernel driver.
- The kernel driver MUST relay the address assignment request to a privileged user process using the UET Control API.
- The privileged user process MUST return the address assignment response to the kernel driver using the UET Control API.
    - In Linux implementations, the kernel driver MUST accept responses only from a privileged process running as root.
- If not provided at job initialization time (see Section 2.2.4), the address assignment response MUST include the following information:
    - JobID
        - If the requesting process was configured with a single JobID at job initialization time, a JobID provided in the response SHOULD take precedence.
        - If the requesting process was configured with multiple JobIDs at job initialization time and the FI_AV_AUTH_KEY capability is enabled, a JobID provided in the response SHOULD be ignored.
    - Address mode
    - PIDonFEP
    - Initiator ID
- The address assignment response MAY optionally include the following information:
    - Security bindings for crypto operations (see Section 2.2.9)

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

        - Security bindings are assigned as specified in Section 2.2.4.1
    - Start resource index
    - Num resource indices
        - Resource index bindings provided in the response SHOULD take precedence over other methods of Index configuration.
- The kernel driver MUST program the JobID validation information and security bindings into the NIC hardware.
- The kernel driver MUST relay the address assignment response, without the security bindings, to the UET provider and SHOULD use a Netlink interface when relaying the response in Linux implementations.

The parameters comprising the UET Control API address assignment request are specified in Table 2-18.

Table 2-18 - UET Control API Address Assignment Request Parameters

Parameter Name	Size (bits)	Description
Flags	8	Bit 0:0 Indicates FEP IP address type 0 => IPv4 1 => IPv6 Bits 1:7 Reserved, MUST be 0
FEP IP Address	128	
OS Process ID	32	
Service Name	136	NULL-terminated character string identifying service

The parameters comprising the UET Control API address assignment response are specified in Table 2-19.

Table 2-19 - UET Control API Address Assignment Response Parameters

Parameter Name	Size (bits)	Description
Flags	8	Bit 0:0 Indicates validity of JobID field 0 => JobID field is NOT valid 1 => JobID field is valid Bit 1: 1 Indicates validity of Security Bindings Field 0 => Security bindings field is NOT valid 1 => Security bindings field is valid Bits 2:7 Reserved, MUST be 0
JobID	24	
UET Address	See Table 2-10	The UET Address format is specified in Table 2-10
Security Bindings	See Table 2-33	Security binding parameters are specified in Table 2-33

'C' structure representations of the UET Control API address assignment request and response are shown below.

```
#define UET_CTRL_ADDR_REQ_FLAG_IPV4    (0 << 0) /* IPv4 fabric address type */
```


Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

```
#define UET_CTRL_ADDR_REQ_FLAG_IPV6   (1 << 0) /* IPv6 fabric address type */
#define UET_CTRL_ADDR_RESP_FLAG_JOB_V (1 << 0) /* JobID valid */
#define UET_CTRL_ADDR_RESP_FLAG_SEC_V (1 << 1) /* security bindings valid */

struct uet_ctrl_addr_req {
            uint8_t flags;
            uint8_t reserved[3];
            struct uet_fa fa;
            uint32_t os_pid;
            char service[UET_MAX_SERVICE_NAME_CHARS+1];
};

struct uet_ctrl_addr_resp {
            uint8_t flags;
            uint8_t reserved[3];
            uint32_t job_id;
            struct uet_addr addr;
            struct uet_sec_bindings sec;
};
```

2.2.5.3.6 fi_getname() API

The *fi_getname()* API is called to retrieve the local UET address of a libfabric endpoint. The call returns an address object that is typically shared with other endpoints of the job.

2.2.5.3.7 fi_cq_open() and fi_cntr_open() APIs

Completion resources are used to report the results of submitted data transfer operations. The completion resource MAY be a completion queue, which is often implemented in hardware, or a completion counter, which can be implemented in hardware or software. The *fi_ep_bind()* API is called to bind a completion resource to an endpoint.

Completion counters simply return the number of operations that have been completed. Counters are intended as lightweight completion objects that increment whenever an identified type of data transfer has occurred, which avoids the overhead of conveying completion queue entries to the application. Completion counters are used by applications such as MPI. The *fi_cntr_open()* API is called to open a completion counter.

The *fi_cq_open()* API is called to open a CQ. Multiple pre-defined CQ entry formats are supported. Provider-specific CQ entry formats are also supported.

The UET provider MUST support completion queues.

The per-profile requirements for completion counters are specified in Table 2-20.

Table 2-20 - Completion Counter Requirements

Profile	Completion Counter Requirements
AI Base	None
AI Full	FI_SEND, FI_RECV, FI_READ, FI_WRITE
HPC	FI_SEND, FI_RECV, FI_READ, FI_WRITE, FI_REMOTE_READ, FI_REMOTE_WRITE

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

The profiles that are required to support completion counters MUST support both the FI_CNTR_EVENTS_COMP and FI_CNTR_EVENT_BYTES event types.

2.2.5.3.8 fi_av_open() and fi_av_insert() APIs

Address vectors are used to map higher-level addresses into fabric-specific addresses. The purpose of the AV is to associate a higher-level address with a simpler, more efficient value that is used by the libfabric API in a fabric-agnostic way.

The *fi_av_open()* API is called to create an address vector, the *fi_ep_bind()* API is called to bind an address vector to an endpoint, and the *fi_av_insert()* API is called to insert the addresses of destination endpoints into the address vector. The *fi_av_insert()* API returns a mapped address of type *fi_addr_t*, which is passed to data transfer APIs to identify the destination endpoint, thereby avoiding the need to pass the full address of a target endpoint with every data transfer. There are two types of address vectors:

- FI_AV_MAP
  - Addresses inserted into an AV are mapped to a native fabric address for application use. The use of FI_AV_MAP requires that an application store the returned *fi_addr_t* value that is associated with each inserted address. FI_AV_MAP is being deprecated in libfabric v2.0. The enum will stay, but the behavior will be like FI_AV_TABLE.
- FI_AV_TABLE
  - Addresses that are inserted into an AV of type FI_AV_TABLE are accessible using a simple index. When FI_AV_TABLE is used, the returned *fi_addr_t* is an index, with the index for an inserted address being the same as its insertion order into the table. The index of the first address inserted into an FI_AV_TABLE will be 0, and successive insertions will be given sequential indices. Sequential indices will be assigned across insertion calls on the same AV.

The AV attributes structure, *fi_av_attr*, contains a name field and a *map_addr* field that are useful for sharing an AV between processes.

The UET provider MUST support the FI_AV_TABLE type. The unspecified type, FI_AV_UNSPEC, MUST be treated as the FI_AV_TABLE type.

Informative Text:
One approach for using an FI_AV_TABLE is to insert addresses for each rank sequentially such that the AV table indices are the same as the rank number.

2.2.5.4 OFI Data Transfer APIs

This section covers the following groups of data transfer APIs:

- *fi_msg*() <https://ofiwg.github.io/libfabric/v1.20.1/man/fi_msg.3.html>
- *fi_tagged*() <https://ofiwg.github.io/libfabric/v1.20.1/man/fi_tagged.3.html>
- *fi_rma*() <https://ofiwg.github.io/libfabric/v1.20.1/man/fi_rma.3.html>

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

- *fi_atomic*() <https://ofiwg.github.io/libfabric/v1.20.1/man/fi_atomic.3.html>
- *fi_collective*() <https://ofiwg.github.io/libfabric/v1.20.1/man/fi_collective.3.html>

2.2.5.4.1 fi_msg() APIs

The *fi_msg()* APIs are used to perform message data transfer operations. There are APIs to post receive buffers for incoming messages and APIs for initiating transmission of outgoing messages. The *fi_msg()* APIs MUST be supported for both FI_EP_DGRAM and FI_EP_RDM endpoints. The *fi_msg()* receive API requirements are summarized in Table 2-21.

Table 2-21 - fi_msg() Receive API Requirements

fi_msg() API	Description	Requirements
fi_recv	Posts a data buffer to the receive queue of the corresponding endpoint. Posted receive buffers are searched in the order they were posted to match sends. Message boundaries are maintained. The *src_addr* parameter MAY be used to indicate that a buffer should be posted to receive incoming data from a specific remote endpoint.	API MUST be supported The requirements for supporting the *src_addr* parameter are specified in Section 2.2.4.3.1
fi_recvv	The *fi_recvv()* API adds support for a scatter-gather list to the *fi_recv()* API	API MUST be supported Maximum size of scatter-gather list is vendor specific
fi_recvmsg	The *fi_recvmsg()* API supports more granular control of the receive operation per call using flag parameters The following flags MUST be supported: • FI_COMPLETION	API MUST be supported FI_MULTI_RECV / FI_TAGGED_MULTI_RECV capabilities and associated flag parameters SHOULD be supported

UET providers SHOULD allocate independent receive queues (i.e., lists of posted receive buffers) for messages and tagged messages.

Table 2-22 summarizes the *fi_msg()* send API requirements and shows the mapping of the APIs to the UET SES opcodes.

Table 2-22 - fi_msg() Send API Requirements

fi_msg() API	Description	Requirements	SES Opcode
fi_send	The *fi_send()* API transfers data to a remote endpoint	API MUST be supported	UET_DATAGRAM_SEND for FI_EP_DGRAM endpoint type UET_SEND for FI_EP_RDM endpoint type
fi_sendv	The *fi_sendv()* API adds support for a scatter-gather list to the *fi_send()* API	API MUST be supported Maximum size of scatter-gather list is vendor specific	UET_DATAGRAM_SEND for FI_EP_DGRAM endpoint type AI Full profile:

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

fi_msg() API	Description	Requirements	SES Opcode
			UET_SEND or UET_DEFERRABLE_SEND for FI_EP_RDM endpoint type Use of UET_SEND vs. UET_DEFERRABLE_SEND is based on message size as specified in Section 2.2.5.4.1.2 HPC profile: UET_SEND or UET_RENDEZVOUS_SEND for FI_EP_RDM endpoint type Use of UET_SEND vs. UET_RENDEZVOUS_SEND is based on message size as in Section 2.2.5.4.1.2
fi_sendmsg	The *fi_sendmsg()* API supports more granular control of the send operation per call using flag parameters	API MUST be supported The following flags MUST be supported: • FI_REMOTE_CQ_DATA • FI_COMPLETION • FI_INJECT • FI_INJECT_COMPLETE • FI_TRANSMIT_COMPLETE • FI_DELIVERY_COMPLETE • FI_FENCE	Same as fi_sendv
fi_inject	The *fi_inject()* API is an optimized version of *fi_send()* with the following characteristics: • The data buffer is available for reuse immediately on return from the call • No CQ entry will be written if the transfer completes successfully • An error CQ entry MUST be written when used with a libfabric endpoint of type FI_EP_RDM and when the message cannot be delivered (this requirement applies to all inject operations)	API MUST be supported The message size used with *fi_inject()* is limited by the *inject_size* attribute of the transmit context, which is vendor specific	Same as fi_sendv

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

fi_msg() API	Description	Requirements	SES Opcode
fi_senddata	The *fi_senddata()* API is like *fi_send()*, but allows for the sending of remote CQ data as part of the transfer (the remote CQ data is written into the target endpoint CQ)	API MUST be supported The remote CQ data is carried in the **ses.header_data** field of the SES header	Same as fi_sendv
fi_injectdata	The *fi_injectdata()* API is like *fi_inject()*, but allows for the sending of remote CQ data as part of the transfer	API MUST be supported The remote CQ data is carried in the **ses.header_data** field of the SES header	Same as fi_inject

2.2.5.4.1.1 Unexpected Messages

Unexpected messages MUST be supported using one of the approaches defined in the SES specification, where the choice of which approach to use at the target is vendor specific. The initiator MUST respond appropriately to all target behaviors.

2.2.5.4.1.2 Message Rendezvous

The SES specification currently defines two types of rendezvous protocols that are referred to as:

- Rendezvous, and
- Deferrable send.

The AI Full profile MUST support the deferrable send option, while the HPC profile MUST support the rendezvous option.

Additionally, the SES specification describes two rendezvous approaches for mapping *CCL send and receive APIs to the libfabric and UET semantics APIs. Both approaches utilize protocols implemented by *CCL plugins that are layered on top of the libfabric APIs. One approach utilizes the *fi_tagged()* APIs, and the other approach is based on the *fi_rma()* APIs.

For the AI Full profile, messages with sizes >= UET_PROVIDER_MSG_RENDEZVOUS_SIZE bytes SHOULD be sent with the UET_DEFERRABLE_SEND semantic opcode.

For the HPC profile, messages SHOULD be sent with the UET_RENDEZVOUS_SEND semantic opcode when the following criteria are met:

- Message size >= UET_PROVIDER_MSG_RENDEZVOUS_SIZE bytes, and
- The source buffer is associated with a local memory region registered for remote read access

The amount of eager data sent as part of a UET_RENDEZVOUS_SEND operation MUST be <= UET_PROVIDER_MAX_EAGER_SIZE bytes.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

2.2.5.4.2 fi_tagged() APIs

The *fi_tagged()* APIs are used to perform tagged data transfer operations. There are APIs to post receive buffers for incoming messages and APIs for initiating transmission of outgoing messages.

The *fi_tagged()* APIs are NOT supported for FI_EP_DGRAM endpoints.

The *fi_tagged()* APIs MUST be supported by the AI Full and HPC profiles.

The *fi_tagged()* Receive API requirements are summarized in Table 2-23.

Table 2-23 - fi_tagged() Receive API Requirements

fi_tagged() API	Description	Requirements
fi_trecv	Like *fi_recv()* but with a tag. Posted receive buffers are searched in the order they were posted to match sends. The ignore parameter contains a bitmask that is applied to the tag to support wildcard tag matches.	API MUST be supported by AI Full and HPC profiles The AI Full profile is required to support only exact match tags. If wildcard tags are not supported, the UET provider MUST fail the API request if any of the ignore bits are set The HPC profile MUST support wildcard tag matching
fi_trecvv	Like *fi_recvv()* but with a tag	API MUST be supported by AI Full and HPC profiles
fi_trecvmsg	Like *fi_recvmsg()* but with a tag	API MUST be supported by AI Full and HPC profiles

Table 2-24 summarizes the *fi_tagged()* send API requirements and shows the mapping of the APIs to the UET SES opcodes.

Table 2-24 - fi_tagged() Send API Requirements

fi_tagged() API	Description	Requirements	SES Opcode
fi_tsend	Like *fi_send()* but with tag	API MUST be supported by AI Full and HPC profiles	AI Full profile: • UET_TAGGED_SEND or • UET_DEFERRABLE_TSEND Use of UET_TAGGED_SEND vs. UET_DEFERRABLE_TSEND is based on message size as described in Section 2.2.5.3.4.1 HPC profile: • UET_TAGGED_SEND or • UET_RENDEZVOUS_TSEND Use of UET_TAGGED_SEND vs. UET_RENDEZVOUS_TSEND is based on message size as described in Section 2.2.5.3.4.1
fi_tsendv	Like *fi_sendv()* but with tag	Same as *fi_tsend()*	Same as *fi_tsend()*

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

fi_tagged() API	Description	Requirements	SES Opcode
fi_tsendmsg	Like *fi_sendmsg()* but with tag	Same as *fi_tsend()*	Same as *fi_tsend()*
fi_tinject	Like *fi_inject()* but with tag	Same as *fi_tsend()*	Same as *fi_tsend()*
fi_tsenddata	Like *fi_senddata()* but with tag	Same as *fi_tsend()*	Same as *fi_tsend()*

2.2.5.4.2.1 Tagged Message Initiator ID Matching

The SES header for received tagged messages includes a **ses.initiator** field used as part of the matching criteria. The Initiator ID associated with a posted tagged buffer is determined based on the *src_addr* parameter of the *fi_tagged()* APIs. If the *src_addr* parameter is not set to FI_ADDR_UNSPEC, the posted tagged buffer MUST be matched only when the tag matches the **ses.match_bits** field in the SES header, and the **ses.initiator** field in the SES header matches the Initiator ID component of the UET address referenced by the *src_addr* parameter. Other components of the UET address referenced by the *src_addr* parameter MAY be used to direct received messages to specific buffers according to the FI_TAGGED_DIRECTED_RECV semantics. If the *src_addr* parameter is set to FI_ADDR_UNSPEC, the **ses.initiator** field MUST NOT be used as part of the matching criteria.

Implementations of the AI Full and HPC profiles MUST support the use of FI_ADDR_UNSPEC with the *fi_tagged()* APIs. Implementations of the AI Full profile SHOULD support setting the *src_addr* parameter of the *fi_tagged()* APIs to reference a specific source UET address. Providers can indicate that FI_ADDR_UNSPEC is not supported and that setting the *src_addr* parameter to reference a specific source UET address is supported via the FI_EXACT_DIRECTED_RECV capability. Implementations of the HPC profile MUST support setting the *src_addr* parameter of the *fi_tagged()* APIs to reference a specific source UET address.

Informative Text:
When used with FI_AV_TABLE, the value of **ses.initiator** on the wire should be the index to the table. The Initiator ID component of the *src_addr* should just be the CCL/MPI rank here, and the two should match.

2.2.5.4.2.2 Tagged Message Rendezvous

The tagged message rendezvous requirements are similar to the message rendezvous requirements specified in Section 2.2.5.4.1.2.

For the AI Full profile, tagged messages with sizes >= UET_PROVIDER_TAG_RENDEZVOUS_SIZE bytes SHOULD be sent with the UET_DEFERRABLE_TSEND semantic opcode.

For the HPC profile, tagged messages with sizes >= UET_PROVIDER_TAG_RENDEZVOUS_SIZE bytes SHOULD be sent with the UET_RENDEZVOUS_TSEND semantic opcode.

The amount of eager data sent as part of a UET_RENDEZVOUS_TSEND operation MUST be <= UET_PROVIDER_MAX_EAGER_SIZE bytes.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

2.2.5.4.3 fi_rma() APIs

The *fi_rma()* APIs are used to perform remote memory access operations. There are APIs for read and write operations.

The *fi_rma()* APIs are NOT supported for FI_EP_DGRAM endpoints.

Table 2-25 summarizes the *fi_rma()* API requirements and shows the mapping of the APIs to the UET SES opcodes.

Table 2-25 - fi_rma() API Requirements

fi_rma() API	Description	Requirements	SES Opcode
fi_read	The *fi_read()* API requests that the remote endpoint transfer data from the remote memory region into the local data buffer	API MUST be supported by AI Full and HPC profiles	UET_READ
fi_readv	The *fi_readv()* API adds support for a scatter-gather list to *fi_read()*	Same as *fi_read()* Maximum size of scatter-gather list is vendor specific	UET_READ
fi_readmsg	The *fi_readmsg()* API supports more granular control of the read operation per call using flag parameters	Same as *fi_read()* The following flags MUST be supported: • FI_COMPLETION	UET_READ
fi_write	The *fi_write()* API transfers the data contained in the user-specified data buffer to a remote memory region	API MUST be supported by AI Base, AI Full, and HPC profiles	UET_WRITE
fi_writev	The *fi_writev()* API adds support for a scatter-gather list to *fi_write()*	Same as *fi_write()* Maximum size of scatter-gather list is vendor specific	UET_WRITE
fi_writemsg	The *fi_writemsg()* API supports more granular control of the write operation per call using flag parameters	Same as *fi_write()* The following flags MUST be supported: • FI_REMOTE_CQ_DATA • FI_COMPLETION • FI_INJECT • FI_INJECT_COMPLETE • FI_TRANSMIT_COMPLETE • FI_DELIVERY_COMPLETE • FI_FENCE	UET_WRITE
fi_inject_write	The *fi_inject_write()* API is an optimized version of *fi_write()* that provides similar completion semantics as *fi_inject()*	Same as *fi_write()* The message size used with *fi_inject_write()* is limited by the *inject_size* attribute of the transmit context, which is vendor specific	UET_WRITE
fi_writedata	The *fi_writedata()* API is like *fi_write()*, but allows for the sending of remote	Same as *fi_write()*	UET_WRITE

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

fi_rma() API	Description	Requirements	SES Opcode
	CQ data as part of the transfer (the remote CQ data is written into the target endpoint CQ)	The remote CQ data is carried in the **ses.header_data** field of the SES header • The **ses.hd** bit is set in the SES header to indicate that the **ses.header_data** field is valid	

2.2.5.4.4 fi_atomic() APIs

The *fi_atomic()* APIs enable remote atomic operations. There are APIs for:

- Initiating an atomic operation to remote memory (sometimes referred to as non-fetching atomics).
- Initiating an atomic operation to remote memory and retrieving the initial value (referred to as fetching atomics).
- Initiating an atomic compare operation to remote memory and retrieving the initial value (which is a type of fetching atomic).
- Querying provider support for specific atomic operations.

The *fi_atomic()* APIs are NOT supported for FI_EP_DGRAM endpoints.

UET providers MUST support bulk non-fetching atomics operations; however, fetching atomics MUST be limited to a single unit of the indicated data type.

Table 2-26 summarizes the *fi_atomic()* API requirements and shows the mapping of the APIs to the UET SES opcodes.

Table 2-26 - fi_atomic() API Requirements

fi_atomic() API	Description	Requirements	SES Opcode
diatomic	The *fi_atomic()* API transfers the data contained in the user-specified data buffer to a remote node	API MUST be supported by all profiles	UET_ATOMIC
fi_atomicv	The *fi_atomicv()* API adds support for a scatter-gather list to *fi_atomic()*	API MUST be supported by all profiles Maximum size of scatter-gather list is vendor-specific	UET_ATOMIC
fi_atomicmsg	The *fi_atomicmsg()* API supports more granular control of the atomic operation per call using flag parameters	API MUST be supported by all profiles The following flags MUST be supported: • FI_COMPLETION • FI_INJECT • FI_FENCE	UET_ATOMIC or UET_TSEND_ATOMIC UET_TSEND_ATOMIC is used when the FI_TAGGED flag is set

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

fi_atomic() API	Description	Requirements	SES Opcode
		• FI_TAGGED	
fi_inject_atomic	The *fi_inject_atomic()* API is an optimized version of *fi_atomic()* that provides similar completion semantics as *fi_inject().*	API MUST be supported by all profiles The message size used with *fi_inject_atomic()* is limited by the *inject_size* attribute of the transmit context, which is vendor-specific	UET_ATOMIC
fi_fetch_atomic	Fetching version of *fi_atomic()*	API MUST be supported by AI Full and HPC profiles	UET_FETCHING_ATOMIC
fi_fetch_atomicv	Fetching version of *fi_atomicv()*	API MUST be supported by AI Full and HPC profiles	UET_FETCHING_ATOMIC
fi_fetch_atomicmsg	Fetching version of *fi_atomicmsg()*	API MUST be supported by AI Full and HPC profiles Flag requirements are the same as *fi_atomicmsg()*	UET_FETCHING_ATOMIC
fi_compare_atomic	The compare atomic APIs are used for operations that require comparing the target data against a value before performing a swap operation	API MUST be supported by AI Full and HPC profiles	UET_FETCHING_ATOMIC
fi_compare_atomicv	Adds support for a scatter-gather list to *fi_compare_atomic()*	API MUST be supported by AI Full and HPC profiles	UET_FETCHING_ATOMIC
fi_compare_atomicmsg	Supports more granular control of the compare atomic operation per call using flag parameters	API MUST be supported by AI Full and HPC profiles Flag requirements are the same as *fi_atomicmsg()*	UET_FETCHING_ATOMIC
fi_atomicvalid	Checks whether a provider supports a specific non-fetching atomic operation for a given datatype and operation	API MUST be supported by all profiles The set of supported operations and data types are vendor-specific	Not applicable
fi_fetch_atomicvalid	Like *fi_atomicvalid()* but for fetching atomics	Same as *fi_atomicvalid()*	Not applicable
fi_compare_atomicvalid	Like *fi_atomicvalid()* but for compare atomic operations	Same as *fi_atomicvalid()*	Not applicable
fi_query_atomic	Advanced atomic valid operation whose behavior is based on a flags parameter; MAY be used to query whether tagged operations are supported	Same as *fi_atomicvalid()*	Not applicable

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

2.2.5.4.5 Collective APIs

A collective operation is a group communication exchange that involves multiple peers exchanging data with other peers participating in the collective call. Collective operations can be thought of as coordinated atomic operations between a set of peer endpoints.

The collective APIs are required to support UET INC. UET INC support is optional for all profiles. When UET INC is supported, the UET provider MUST advertise the FI_COLLECTIVE capability. When UET INC is not supported, the UET provider SHOULD NOT advertise the FI_COLLECTIVE capability.

A provider that supports INC MUST be able to communicate with the INC manager. The UET_IM_URL_PRIMARY and UET_IM_URL_SECONDARY environment variables specify URLs used to communicate with the INC manager.

2.2.5.5 Other APIs

This section summarizes requirements for the set of other libfabric APIs that have not been described elsewhere in this specification.

Table 2-27 shows libfabric APIs for which support is NOT required.

Table 2-27 - Libfabric APIs for which Support is Not Required

API Group	Support is Not Required	Description
fi_av	fi_av_insertsvc, fi_av_insertsym	
fi_cm	All APIs in fi_cm API Group except fi_getname	Connection-oriented APIs are not needed by UET
fi_domain	fi_domain2, fi_open_ops, fi_set_ops	fi_domain2 for opening peer domain
fi_endpoint	fi_endpoint2 fi_passive_ep, fi_pep_bind fi_scalable_ep, fi_scalable_ep_bind fi_srx_context, fi_stx_context fi_rx_size_left, fi_tx_size_left	fi_endpoint2 for peer transfers Connection-oriented APIs are not needed by UET Scalable endpoint support is not required Shared context support is not required Deprecated by libfabric
fi_mr	fi_mr_raw_attr, fi_mr_map_raw, fi_mr_unmap_key, fi_hmem_ze_device	Raw memory region key support is not required
fi_peer	All APIs in *fi_peer()* API Group	Peer APIs are experimental, and support is not required
fi_trigger	fi_trigger	

Table 2-28 shows libfabric API options (e.g., flags, operations, parameters, etc.) for which support is NOT required.

Table 2-28 - Libfabric API Options for which Support is Not Required

API Group	Support is Not Required
fi_av	FI_SYMMETRIC flag
fi_cq	FI_COMMIT_COMPLETE flag

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

API Group	Support is Not Required
fi_domain	FI_SET_OPS_HMEM_OVERRIDE operation
fi_endpoint	FI_OPT_BUFFERED_LIMIT, FI_OPT_BUFFERED_MIN, FI_OPT_CM_DATA_SIZE, FI_OPT_FI_HMEM_P2P, FI_OPT_XPU_TRIGGER, and FI_OPT_CUDA_API_PERMITTED options
fi_msg	FI_CLAIM and FI_DISCARD flags
fi_mr	FI_RMA_PMEM, FI_HMEM_DEVICE_ONLY, FI_HMEM_HOST_ALLOC, and FI_MR_DMABUF flags

2.2.5.6 Libfabric API Error Codes

SES Section 3.4.6.6 describes an extensive set of error codes. A subset of the semantic errors MAY result in a completion error for the associated libfabric API function via the *fi_cq_err_entry* data structure. For semantic errors that result in a completion error, the err field of the *fi_cq_err_entry* structure MUST be populated with an appropriate error code. If a libfabric error code corresponding to a semantic error is not found, the FI_EIO error code MUST be returned. In addition, when a semantic completion error occurs, the provider MUST populate the semantic error code into the *prov_errno* field of the *fi_cq_err_entry* structure.

2.2.6 Packet Delivery Modes

This section specifies how the UET packet delivery mode is selected.

Libfabric endpoints of type FI_EP_DGRAM MUST use the UET UUD packet delivery mode.

Libfabric endpoints of type FI_EP_RDM MUST use one of the following UET packet delivery modes:

- RUD (supported by all profiles)
- ROD (supported by all profiles)
- RUDI (supported by HPC profile)

The AI Base and AI Full profiles MUST select either ROD or RUD. The selection is based on the operation type and the message ordering modes that are configured.

To clarify the selection criteria, the following message ordering modes are defined:

- Send message ordering is in effect when any of the following message ordering modes are configured (send message ordering refers to any ordering mode that specifies the ordering of send operations relative to other operations):
    - FI_ORDER_RAS, FI_ORDER_SAR, FI_ORDER_SAS, FI_ORDER_SAW, FI_ORDER_WAS
- R/W message ordering is in effect when any of the following message ordering modes are configured (R/W message ordering refers to any ordering mode that specifies the ordering of read or write operations relative to other operations):
    - FI_ORDER_ATOMIC_RAR, FI_ORDER_ATOMIC_RAW, FI_ORDER_ATOMIC_WAR, FI_ORDER_ATOMIC_WAW, FI_ORDER_RAR, FI_ORDER_RAS, FI_ORDER_RAW, FI_ORDER_RMA_RAR, FI_ORDER_RMA_RAW, FI_ORDER_RMA_WAR, FI_ORDER_RMA_WAW, FI_ORDER_SAR, FI_ORDER_SAW, FI_ORDER_WAR, FI_ORDER_WAS

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

The AI Base profile MUST select the packet delivery mode according to Table 2-29.

Table 2-29 - Packet Delivery Mode Selection Criteria for AI Base Profile

SEND Ordering	R/W Ordering	SEND Operation	RMA or ATOMIC Operation
N	N	RUD	RUD
N	Y	RUD	ROD
Y	N	ROD	RUD
Y	Y	ROD	ROD

The AI Full profile MUST select the packet delivery mode according to Table 2-30.

Table 2-30 - Packet Delivery Mode Selection Criteria for AI Full Profile

SEND Ordering	R/W Ordering	SEND or TAGGED SEND Operation	DEFERRABLE TAGGED SEND Operation	RMA or ATOMIC Operation
N	N	RUD	RUD	RUD
N	Y	RUD	RUD	ROD
Y	N	ROD	Initial/eager data: ROD Remaining data: RUD	RUD
Y	Y	ROD	Initial/eager data: ROD Remaining data: RUD	ROD

The HPC profile MUST select either ROD, RUD, or RUDI. The selection is based on the operation type, the message ordering modes that are configured, and vendor-specific policy. The HPC profile MUST select the packet delivery mode according to Table 2-31. In the cases that show RUD/RUDI, the selection of whether to use RUD or RUDI SHOULD be made based on vendor-specific policy.

Table 2-31 - Packet Delivery Mode Selection Criteria for HPC Profile

Send Ordering	R/W Ordering	SEND or TAGGED SEND Operation	RENDEZVOUS Operation	RMA Operation	ATOMIC Operation
N	N	RUD	RUD	RUD/RUDI	RUD
N	Y	RUD	RUD	ROD	ROD
Y	N	ROD	Initial/eager data: ROD Remaining data: RUD	RUD/RUDI	RUD
Y	Y	ROD	Initial/eager data: ROD Remaining data: RUD	ROD	ROD

2.2.7 Traffic Classes

The SES sublayer is aware of only data traffic classes. The libfabric application is not aware of traffic classes used by PDS. The default value is shown in Table 2-32.

Implementation Note:
Other sublayers of UET use additional traffic classes with associated DSCP values for services that are not visible to the libfabric provider. These traffic classes are configured via OS-specific means (e.g., Linux TC) and SHOULD be consistent throughout the network.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Table 2-32 - Default Traffic Classes

Traffic Class	Default
Data Traffic Class	Default Forwarding (DF) PHB DSCP Codepoint = '000000' See RFC 2474 [1]

The default traffic classes can be overridden by setting the UET_PROVIDER_DEF_DATA_TC configuration parameters defined in Table 2-7.

The libfabric application controls the data traffic class using the *tclass* field of struct *fi_domain_attr*. The value of the *tclass* field indicates whether the default traffic class or a specific DSCP should be used for the data traffic class as specified in Table 2-4.

2.2.8 Transmit and Receive Queues

The libfabric data transfer operations are typically implemented with a set of transmit and receive queues that are accessed by the NIC hardware. This section provides requirements and guidance regarding the operational characteristics of the transmit and receive queues with the goal of promoting common NIC behavior and an associated collective understanding of that behavior, which should simplify performance tuning.

The details of transmit and receive queue operation are vendor specific.

2.2.8.1 Transmit Queues

Transmit queues are used for libfabric APIs that initiate transmissions on the network (e.g., *fi_send(), fi_tsend(), fi_write(), fi_read()*, etc.). Transmit queues contain work elements, where the work elements describe the operation that is to be performed and identify the associated data buffer. The provider inserts work elements into a transmit queue, and NIC hardware removes work elements from a transmit queue. When the NIC hardware removes a work element, it performs the associated network transmission.

In the simplest case, a single transmit queue could be used to initiate all network transmissions. However, use of a single transmit queue might not achieve optimal performance due to head-of-line blocking issues and lack of support for multiple traffic classes with different transport characteristics. Figure 2-9 shows an example transmit queue configuration.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

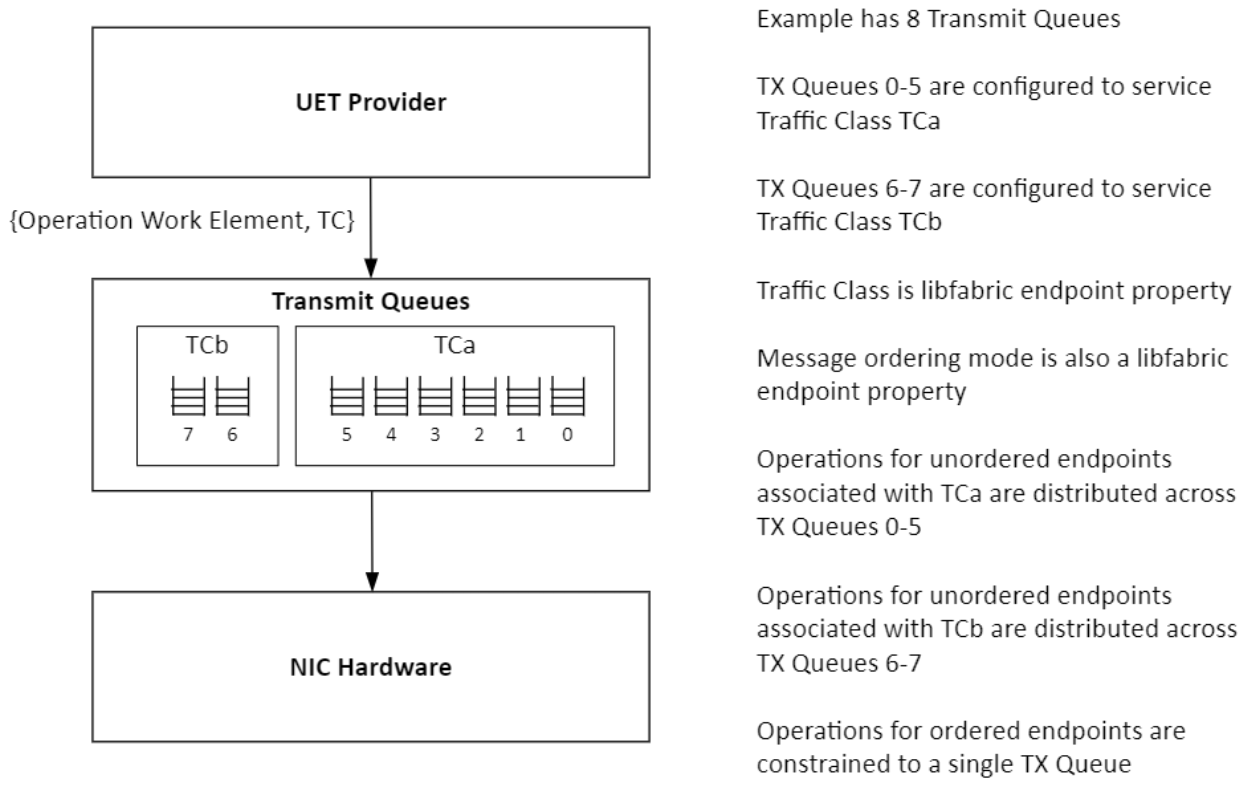


Figure 2-9 - Transmit Queue Example

A UET provider SHOULD support multiple transmit queues.

A UET provider SHOULD support mapping a traffic class to one or more transmit queues, such that different traffic classes MAY be mapped to different sets of transmit queues.

A UET provider SHOULD distribute unordered message operations associated with a particular traffic class (e.g., TCa) across the set of transmit queues that are configured to service that traffic class (e.g., TCa).

When a libfabric endpoint is configured for message ordering, a UET provider MUST constrain the operations for that endpoint to a single transmit queue. The libfabric message ordering modes were discussed in Section 2.2.6.

2.2.8.2 Receive Queues and Registered Memory Regions

Receive queues are used for libfabric APIs that post buffers to be used for receiving messages from the network (e.g., *fi_recv(), fi_trecv(),* etc.). Receive queues contain elements that identify the associated data buffer and its attributes (such as a tag). The provider inserts elements into a receive queue, and NIC hardware removes elements from a receive queue. When the NIC hardware removes an element, it stores message data received from the network in the associated data buffer.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Memory regions that are registered with the libfabric provider are like receive queues in that they identify buffers used as the target network operations. Registered memory regions MAY be targeted by remote RMA operations.

Figure 2-10 shows an example set of data structures for the receive queues and registered memory regions associated with a libfabric endpoint. In the example:

- Separate receive queues are allocated for untagged messages and tagged messages.
- A table is used to manage the registered memory regions. The table contains descriptors that identify the associated data buffer and attributes (such as access permissions).

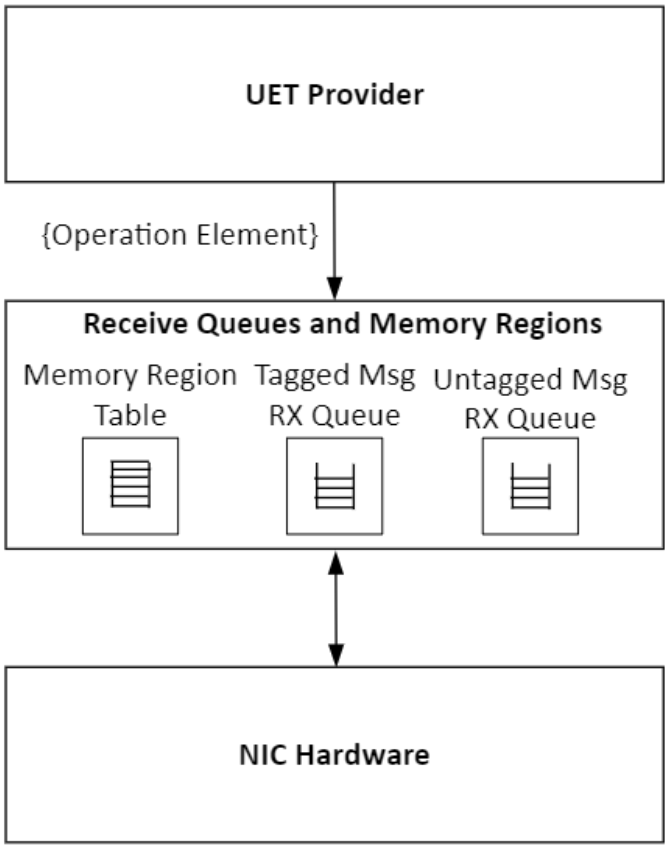


Figure 2-10 - Example Receive Queue and Registered MR Data Structures

A UET provider that supports tagged messages SHOULD allocate independent receive queues for untagged messages and tagged messages on a libfabric endpoint basis.

2.2.9 Security Protocol

This section is devoted to provider support for the optional UET security protocol. The security protocol support is implemented by the kernel driver associated with the provider under the covers in a manner that is transparent to the libfabric APIs. When a libfabric endpoint is opened, the security binding parameters are received by the provider kernel driver as part of the UET address assignment procedure

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

that is described in Section 2.2.5.3.5.1. The security Binding Parameters are specified in Table 2-33. A UET provider that implements the UET security protocol MUST support the security binding parameters specified in Table 2-33.

Table 2-33 - UET Security Binding Parameters

Parameter Name	Size (bits)	Description
Alg	4	Cipher Algorithm 0x00: AES-GCM-256 0x01-0x0F: Reserved
Rekey	4	Rekey Mode Bit 0:0 0 => not AN, 1 => AN Bit 1:1 0 => not Automatic, 1 => Automatic Bits 2:3 Reserved, MUST be 0
Mode	4	Crypto Mode 0x00: Direct 0x01: Cluster 0x02: Client-server 0x03-0x0F: Reserved
Rekey Shift	6	Shift for automatic rekeying operation
Rekey Mask	64	Mask for automatic rekeying operation
Encap-type	2	0b00: Native IPv4 0b01: Native IPv6 0b10: UDP over IPv4 0b11: UDP over IPv6
Coff	12	Crypto Offset (in units of 4B)
Aoff	12	Authentication offset
AN	1	Association Number (i.e., Key Generation ID)
SDI	31	Secure Domain Identifier
SSI	32	Secure Source Identifier

A kernel driver that implements the UET security protocol MUST also support rekeying using the rekey parameters specified in Table 2-34.

Table 2-34 - UET Rekey Parameters

Parameter Name	Size (bits)	Description
AN	1	Association Number (i.e., Key Generation ID)
SDI	31	Secure Domain Identifier
SSI	32	Secure Source Identifier
current_epoch	16	Current key epoch
IVMASK	96	IVMASK
Key Type	8	Type of Key: Raw or Wrapped
Key Length	16	Length of Key (in bytes)
Key	512	Key / Wrapped Key

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

The rekey parameters MAY be requested by the kernel driver or pushed by a key management system. The architecture for obtaining the rekey parameters is shown in Figure 2-11.

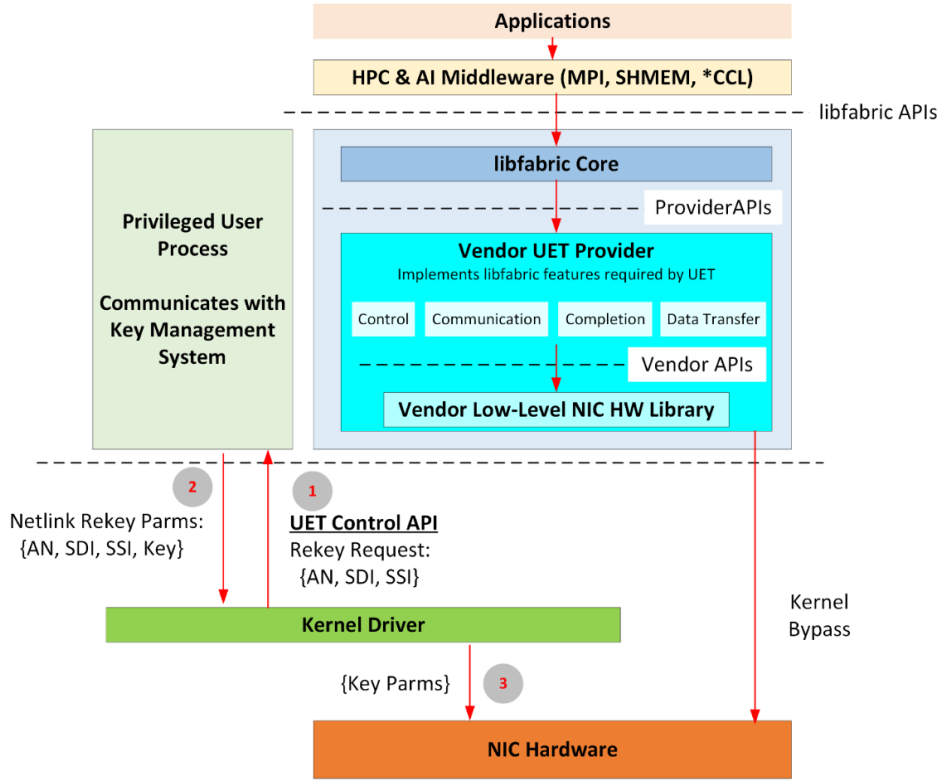


Figure 2-11 - Rekey Parameter Acquisition Architecture

The steps shown in Figure 2-11 are:

1. UET provider kernel driver makes a rekey request to privileged user process via the UET Control API.
2. The privileged user process communicates with a key management system and returns the requested rekey information.
3. The provider kernel driver programs the key parameters into the NIC hardware.

Requirements associated with this architecture are:

- A UET provider kernel driver MUST support sending rekey requests to a privileged user process using the UET Control API.
- The privileged user process MUST respond to rekey requests from the provider kernel driver with rekey parameters.
- The provider kernel driver MUST accept both solicited and unsolicited rekey parameters received from the privileged user process, and it MUST program the key parameters into the NIC hardware.

The provider kernel driver MUST include the parameters specified by Table 2-35 in rekey requests.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Table 2-35 - UET Control API Rekey Request Parameters

Parameter Name	Size (bits)	Description
AN	1	Association Number (i.e., Key Generation ID)
SDI	31	Secure Domain Identifier
SSI	32	Secure Source Identifier

'C' structure representations of the UET security binding parameters, rekey parameters, and rekey request parameters are shown below.

```
#define UET_SEC_MAX_KEY_OCTETS      64
#define UET_SEC_ALG_AES_GCM_256     0        /* AES-GCM-256 algorithm */
#define UET_SEC_REKEY_MODE_AN      (1 << 0)  /* association num rekey mode */
#define UET_SEC_REKEY_MODE_AUTO    (1 << 1)  /* automatic rekey mode */
#define UET_SEC_CRYPTO_MODE_DIRECT  0        /* direct crypto mode */
#define UET_SEC_CRYPTO_MODE_CLUSTER 1        /* cluster crypto mode */
#define UET_SEC_CRYPTO_MODE_CSERVER 2        /* Client-server crypto mode */
#define UET_SEC_AN_BIT             (1 << 31) /* msb of sdi */
#define UET_SEC_RAW_KEY_TYPE        1        /* key is not wrapped */
#define UET_SEC_WRAPPED_KEY_TYPE    2        /* key is wrapped */
#define UET_SEC_IVMASK_OCTETS       96       /* IVMASK size in octets */

struct uet_sec_bindings {
          uint8_t alg;
          uint8_t rekey;
          uint8_t crypto_mode;
          uint8_t rekey_shift;
          uint64_t rekey_mask;
          uint16_t reserved;
          uint16_t coff;
    uint16_t aoff;
    uint32_t an_sdi;
          uint32_t ssi;
          uint8_t encap_type;
};

struct uet_ctrl_rekey_parms {
          uint32_t an_sdi;
    uint32_t ssi;
    uint16_t current_epoch;
    uint8_t ivmask[UET_SEC_IVMASK_OCTETS];
          uint8_t key_type;
          uint8_t reserved_1;
          uint16_t key_len;
          uint32_t reserved_2;
          uint8_t key[UET_SEC_MAX_KEY_OCTETS];
};

struct uet_ctrl_rekey_req {
          uint32_t an_sdi;
          uint32_t ssi;
};
```


Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

2.2.10 Wire Protocol Mapping

This section specifies how the libfabric APIs and associated data structures are mapped to fields in the following wire protocol headers:

- IP header
- UET TSS header
- UET PDS header
- UET SES headers

Figure 2-12 contains a high-level depiction of the libfabric mapping to UET wire protocol headers.

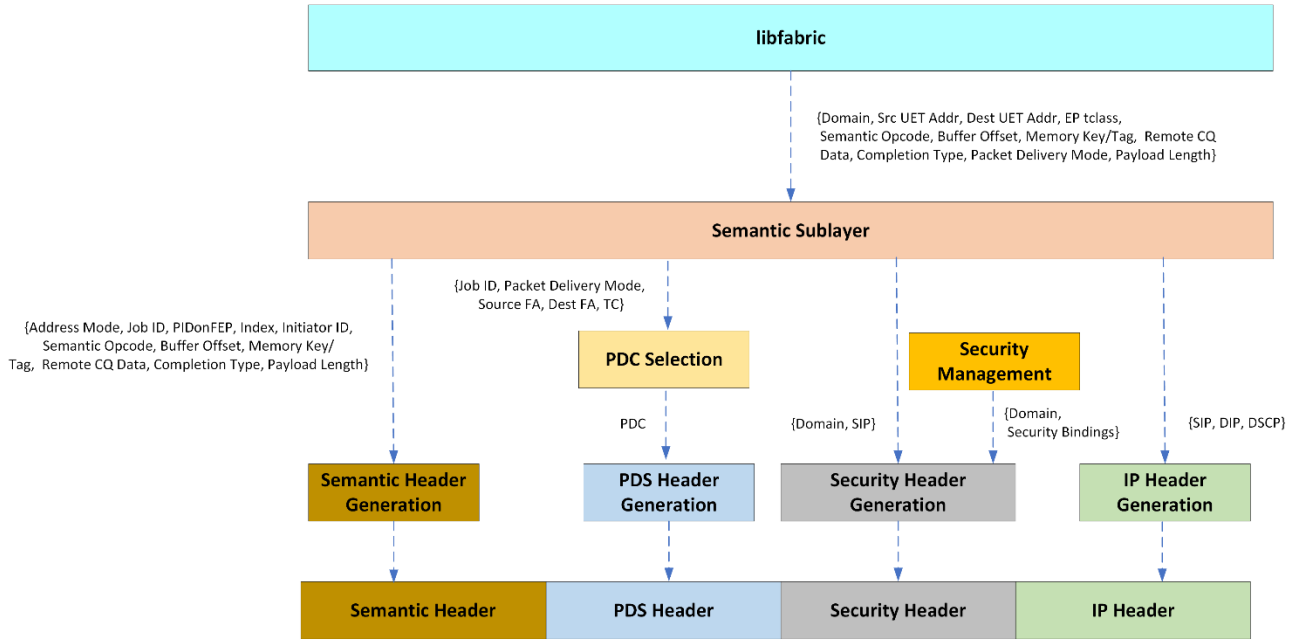


Figure 2-12 - Libfabric Mapping to UET Wire Protocol Headers

2.2.10.1 IP Header Field Mappings

The libfabric to IP header field mappings are shown in Table 2-36.

Table 2-36 - Libfabric to IP Header Mapping

IP Header Field	Libfabric Source
source_ip_address	IP address component of UET address assigned to source libfabric endpoint
destination_ip_address	IP address component of UET address assigned to destination libfabric endpoint
dscp	*tclass* field of struct *fi_domain_attr* associated with source libfabric endpoint

2.2.10.2 UET TSS Field Mappings

The TSS header field generation is based on security bindings from the provider kernel driver and per-packet information from libfabric. The security bindings are initialized when the libfabric endpoint is opened and are refreshed when rekeying occurs. The per-packet information from libfabric includes a libfabric domain identifier and the source IP address of the FEP. The domain identifier is used to find the

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

security binding parameters that should be applied, while the source IP address is used by the key derivation function.

2.2.10.3 UET PDS Header Field Mappings

The PDS headers vary based on the packet delivery mode. The UUD and RUDI packet delivery modes do not use packet delivery contexts and have small PDS headers. For the RUD and ROD delivery modes, the libfabric to PDS header field mapping is primarily an indirect mapping, where libfabric data is used to select the PDC, and then the PDC state determines the PDS header field contents. The libfabric fields used to select the PDC are shown in Table 2-37.

Table 2-37 - Libfabric Fields Used to Select Packet Delivery Context

PDC Selection Field	Libfabric Source
JobID	JobID associated with a libfabric API operation
Source FA	IP address component of a UET address assigned to the source libfabric endpoint
Destination FA	IP address component of a UET address assigned to the destination libfabric endpoint
Traffic Class	*tclass* field of struct *fi_domain_attr* associated with the source libfabric endpoint
Packet Delivery Mode	As specified in Section 2.2.6

2.2.10.4 UET SES Header Field Mappings

The SES specification describes multiple header formats as summarized in Table 2-38.

Table 2-38 - Summary of Semantic Header Formats

SES Header Format	Description
Standard Request	Format used by most requests
Non-Matching Request	Optimized format for requests that do not require matching or header data
Small Message / Small RMA	Specialized format for: • Single-packet tagged messages • Single-packet RMA operations that cannot use the non-matching request format
Deferrable Send Request	Specialized format for deferrable send requests that replaces the offset field in the standard request with a restart token
Ready To Restart	Variation of standard request used to restart paused deferrable send
Rendezvous Extension	Used in conjunction with rendezvous send operations • The extension includes an eager length that indicates how much message payload is being pushed with the request, and the addressing information needed to issue a read operation
Atomic Extension	Used in conjunction with atomic operations • The extension contains an atomic opcode and datatype derived from parameters of the associated *fi_atomic*() API call
Compare and Swap	Used in conjunction with compare and swap atomic operations • In addition to the atomic opcode and datatype, the extension also contains the compare value and the swap value parameters from the associated *fi_compare_atomic()* API call
Response	Used for SES acknowledgements and responses with data

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

The libfabric mappings for the SES standard request header are shown in Table 2-39.

Table 2-39 - Mappings for Standard Request (UET HDR_REQUEST_STD)

Request Field	Libfabric Source
opcode	Based on an associated libfabric API as specified in Section 2.2.5.4
delivery complete (dc)	Set when an operation completion mode associated with libfabric API call is FI_DELIVERY_COMPLETE
Relative (rel)	Address mode component of UET address assigned to destination libfabric endpoint
header data present (hd)	Set when the associated libfabric API provided remote CQ data
resource_index	Resource Index component of UET address assigned to destination libfabric endpoint
ri_generation	Managed by the provider in accordance with the SES specification
JobID	JobID associated with libfabric API operation (see Section 2.2.4)
PIDonFEP	PIDonFEP component of the UET address assigned to the destination libfabric endpoint
buffer_offset	Offset from the buffer starting address specified in the associated libfabric API call
initiator	Initiator ID component of the UET address assigned to the source libfabric endpoint
match_bits	Remote memory key for RMA/atomic opcodes, tag for tagged send opcodes
header_data (som=1) payload_length + message_offset (som=0)	Remote CQ data from an associated libfabric API call Managed by the provider in accordance with the SES specification
request_length	Length of the payload provided on an associated libfabric API call

The criteria for using the UET HDR_REQUEST_SMALL header for RMA operations are specified in Section 2.2.5.3.4.1. The same criteria MUST be applied for atomic operations. The UET HDR_REQUEST_SMALL header SHOULD be used only for RMA or atomic operations that satisfy the specified criteria. The libfabric mappings for UET HDR_REQUEST_SMALL are shown in Table 2-40.

Table 2-40 - Mappings for Optimized Non-Matching (UET HDR_REQUEST_SMALL)

Request Field	Libfabric Source
opcode	Same as standard format
delivery complete (dc)	Same as standard format
resource_index	Resource Index for MR as specified in Section 2.2.5.3.4.1
JobID	Same as standard format
PIDonFEP	Same as standard format
request_length	Length of payload provided on associated libfabric API call
buffer_offset	Same as standard format

A single-format, UET_HDR_REQUEST_MEDIUM supports both small message and small RMA use cases. The criteria for the small message use case are specified in Table 2-41. The small message SES header SHOULD be used when the criteria specified in Table 2-41 are satisfied.

Table 2-41 - Criteria for Small Message Header (UET HDR_REQUEST_SMALL)

Criteria for Use of Small Message SES Header
Message size <= MTU
Operation is tagged send

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

The libfabric mappings for the small message SES header are shown in Table 2-42.

Table 2-42 - Mappings for Small Message (UET HDR_REQUEST_SMALL)

Request Field	Libfabric Source
opcode	UET_TAGGED_SEND
delivery complete (dc)	Same as standard Format
relative (rel)	Same as standard format
header data present (hd)	Same as standard format
resource_index	Same as standard format
ri_generation	Same as standard format
JobID	Same as standard format
PIDonFEP	Same as standard format
request_length	Length of payload provided on associated libfabric API call
header_data	Same as standard format
initiator	Same as standard format
match_bits	Tag

The criteria and requirements for the small RMA use case are specified in Section 2.2.5.3.4.1. The same criteria MUST be applied for atomic operations. The small RMA SES header SHOULD be used for RMA or atomic operations that satisfy the specified criteria. The libfabric mappings for the small RMA SES header are shown in Table 2-43.

Table 2-43 - Mappings for Small RMA (UET HDR_REQUEST_MEDIUM)

Request Field	Libfabric Source
opcode	Same as standard format
delivery complete (dc)	Same as standard format
relative (rel)	Same as standard format
header data present (hd)	MUST be 0
resource_index	Same as standard format
ri_generation	Same as standard format
JobID	Same as standard format
PIDonFEP	Same as standard format
request_length	Length of payload provided on associated libfabric API call
buffer_offset	Same as standard format
initiator	Same as standard format
match_bits	Memory key

The criteria and requirements for use of the deferrable send operation are specified in Section 2.2.5.4. The deferrable send SES header format is similar to UET_HDR_REQUEST_STD except the **ses.buffer_offset** field is replaced by provider-supplied **ses.initiator_restart_token** and **ses.target_restart_token** fields. The ready-to-restart (RTR) message in the deferrable send sequence also carries the **ses.iniator_restart_token** and **ses.target_restart_token** fields. The restart tokens are used to identify the operation that is being restarted. The provider MUST ensure that all active restart tokens are unique at the initiating FEP.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

The criteria and requirements for use of the rendezvous send operation are specified in Section 2.2.5.4. The libfabric mappings for the rendezvous send extension header, which follows UET_HDR_REQUEST_STD, are shown in Table 2-44.

Table 2-44 - Libfabric Mappings for Rendezvous Send Extension Header

Request Field	Libfabric Source
eager_length	<= UET_PROVIDER_MAX_EAGER_SIZE
read_PIDonFEP	PIDonFEP component of UET address assigned to source libfabric endpoint
read_resource_index	Resource Index component of UET address assigned to source libfabric endpoint
read_ri_eneration	Managed by provider in accordance with SES specification
read_offset	• Offset of source buffer parameter from the send API within a registered local memory region (memory region is determined using desc parameter of send API) • The source buffer MUST be associated with a local memory region registered for remote read access in order to perform a rendezvous send • This offset reflects the start of the message within the memory region and, therefore, MAY be used to read the entire message
read_memory_key	RKEY of the local memory region associated with the source buffer

The atomic extension header is used in conjunction with the *fi_atomic()* APIs and MAY follow UET_HDR_REQUEST_STD or either of the optimized headers. The atomic extension header includes an **ses.atomic_opcode** field and an **ses.atomic_datatype** field. A UET provider MUST map the atomic operation and atomic datatype specified in the *fi_atomic()* API to the associated SES header atomic mnemonics, which are defined in the UET SES specification.

The compare and swap extension header is a specialized header used with compare and swap atomic operations. In addition to the **ses.atomic_opcode** and **ses.atomic_datatype** fields, the extension also contains the compare value and the swap value parameters from the associated *fi_compare_atomic()* API. These parameters map to the **ses.compare_value** and **ses.swap_value** fields respectively. The size of the compare and swap parameters MUST be <= 16 bytes.

2.2.11 Linux Implementation of UET Control API

The Linux implementation of the UET Control API described in this section is a proposal to the upstream community. It is expected that changes and feedback will be incorporated as a part of the upstream process. The definitive location of the interfaces will be in a Linux include file.

Multiple UET Control API messages between the provider KMD and management entities have been defined previously in this specification. More specifically, the following:

- UET Control API JobID mapping request message (see Section 2.2.4.1)
- UET Control API address assignment request message (see Section 2.2.5.3.5.1)
- UET Control API address assignment response message (see Section 2.2.5.3.5.1)
- UET Control API security rekey message (see Section 2.2.9)
- UET Control API security rekey request message (see Section 2.2.9

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

In Linux implementations, the UET Control API MUST be implemented with Netlink messages. The Netlink messages are all exchanged with the UET provider kernel driver using the new “UET” family. The base set of encodings for the Netlink commands that represent each message are shown below.

```
enum uet_nl_cmd {
      UET_NL_CMD_JOB_ID = 1,
      UET_NL_CMD_ADDR_REQ,
      UET_NL_CMD_ADDR_RESP,
      UET_NL_CMD_REKEY,
      UET_NL_CMD_REKEY_REQ,
      __UET_NL_CMD_MAX
};
```

For extensibility, Netlink does not represent messages as ‘C’ structures but uses a sequence of typed attributes instead. The base set of attributes used to represent UET Netlink messages are shown below.

```
enum uet_nl_attr {
      UET_NL_ATTR_JOB_ID,
      UET_NL_ATTR_OS_PID,
      UET_NL_ATTR_SERVICE,
      UET_NL_ATTR_FLAGS,
      UET_NL_ATTR_FA,
      UET_NL_ATTR_UET_ADDR,
      UET_NL_ATTR_SEC_BINDINGS,
      UET_NL_ATTR_SEC_AN_SDI,
      UET_NL_ATTR_SEC_SSI,
      UET_NL_ATTR_SEC_KEY,
      __UET_NL_ATTR_MAX
};

static const struct nla_policy uet_policy[__UET_NL_ATTR_MAX] = {
      [UET_NL_ATTR_JOB_ID] = { .type = NLA_U32 },
      [UET_NL_ATTR_OS_PID] = { .type = NLA_U32 },
      [UET_NL_ATTR_SERVICE] = { .type = NLA_NUL_STRING,
                                .len = UET_MAX_SERVICE_NAME_CHARS },
      [UET_NL_ATTR_FLAGS] = { .type = NLA_U8 },
      [UET_NL_ATTR_FA] = NLA_POLICY_EXACT_LEN(sizeof(struct uet_fa)),
      [UET_NL_ATTR_UET_ADDR] = NLA_POLICY_EXACT_LEN(sizeof(struct uet_addr)),
      [UET_NL_ATTR_SEC_BINDINGS] = NLA_POLICY_EXACT_LEN(
sizeof(struct uet_sec_bindings)),
      [UET_NL_ATTR_SEC_AN_SDI] = { .type = NLA_U32 },
      [UET_NL_ATTR_SEC_SSI] = { .type = NLA_U32 },
      [UET_NL_ATTR_SEC_KEY] = NLA_POLICY_EXACT_LEN(UET_SEC_MAX_KEY_OCTETS)
};
```

2.2.12 References

[1] IETF RFC 2474, "Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers," 1998. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc2474.

[2] IETF RFC 3246, "An Expedited Forwarding PHB (Per-Hop Behavior)," 2002. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc3246.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

2.3 UE Verbs Mapping

UET verbs are device and transport primitive operations for accessing hardware devices supporting the Ultra Ethernet Transport. UET verbs are defined as fully backward-compatible extensions to RDMA verbs (via the libibverbs API). For the purposes of this document, RDMA verbs refers to the portions of the libibverbs API that support specification-compliant hardware and transport operations defined for InfiniBand, RoCE, RoCEv2, and iWARP.

The RDMA verbs APIs and proposed extensions that target UET support are collectively referred to as UET verbs. The objectives of UET verbs include the following, which are used to drive UET verbs API requirements:

- Existing binary RDMA verbs applications must continue to work unmodified over RDMA networks (i.e., UET verbs will maintain ABI compatibility). Existing RDMA verbs applications must be modified to work over UET.
- Existing RDMA verbs applications that are recompiled against a version of libibverbs that supports UET must continue to work unmodified over RDMA networks (i.e., API compatibility is maintained).
- The AI Base profile is the primary target of this specification. Select features of additional profiles are included where such mappings are straightforward. Full support for additional profiles is left for future extensions.
- It should be possible to develop a libfabric provider compliant with the libfabric mapping specification over UET verbs.
- UET verbs shall maintain independence from the API used by the peer application as well as preserve transport interoperability.
- UET verbs extensions to RDMA verbs shall be generalized where possible, such that they may be usable by other transports or RDMA enhancements.
- UET verbs should permit simultaneous use of multiple transport services (e.g., an application may use UET for CCL and an RDMA transport to access storage).

This document defines explicit UET verbs extensions to RDMA verbs. It is the intent that the UET extensions will be proposed to the RDMA-Core upstream community as defined. However, UET extensions should be viewed in terms of the concepts being exposed, as the upstreaming process will result in negotiating with the upstream community on how UET verbs is ultimately defined.

Informative Text:
This spec provides a high-level description of how verbs is mapped over UET. Design specifics are purposefully left out. Code changes specific to UET verbs in rdma-core can be found here: <https://github.com/ultraethernet/uet-rdma-core/tree/uet>

2.3.1 Verbs Terminology

Verbs is defined in Chapter 11 of the InfiniBand Architecture Specification [1] as an abstract definition of the functionality provided by an InfiniBand NIC. libibverbs was designed as a formal software API aligned

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

with that abstraction. As a result, API names, including the library name, are closely aligned with those defined for InfiniBand.

However, the library and API have evolved to support additional high-performance transports and NICs. libibverbs constructs have expanded beyond their traditional roles and definitions, except that the original InfiniBand naming has been kept for backwards compatibility purposes.

Today, verbs can be viewed as defining software primitives for NIC hardware supporting one or more of the following:

- NIC hardware queues are directly accessible from user space.
- NIC hardware can directly access application memory buffers.
- Transport supporting RDMA operations.

Even the meaning of RDMA has evolved to refer to the key features of devices that support RDMA transports and not just the transport operation.

The impact of integrating UET support into verbs is that UET concepts will adopt RDMA naming. However, such naming should not be interpreted as requiring that UET must adhere to RDMA semantics, as RDMA objects have evolved beyond their original definitions. Select RDMA objects are defined in the following section.

2.3.2 Verbs Overview

Conceptually, "verbs" is an abstract description of the functionality of an RDMA NIC. RDMA verbs is an API definition of that abstraction, with these general characteristics:

- The API is device-centric. Functions map to a specific NIC.
- There is an identifiable mapping between API calls and transport operations and fields.

RDMA verbs defines several objects that are relevant for UET verbs. These are highlighted in Table 2-45 with applicable implementation details.

Table 2-45 - Verb Objects

Object	Description
Device	An RDMA or verbs device is one that is accessible through the Linux RDMA subsystem and, as a result, plugs into the libibverbs and rdma-core framework. NICs plug into the RDMA subsystem to expose hardware primitives supported by verbs. NICs do not necessarily need to support transport offload to leverage the rdma-core infrastructure. It is sufficient for a NIC to expose similar features found in RDMA devices. The device list is obtained using a netlink socket to communicate with the Linux RDMA subsystem.
Device Context	Identifies an instance of an opened NIC.
Command FD (cmd_fd)	File descriptor used to communicate with the kernel device driver. Associated with the device context and opened by libibverbs.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Object	Description
	The cmd_fd communicates with the kernel via ioctls and is used to allocate, configure, and release device resources. Applications interact with the cmd_fd indirectly by calling libibverbs function calls.
Async Event FD (async_fd)	File descriptor used to report asynchronous events. Associated with the device context and opened by libibverbs. Applications may interact directly with the async_fd, such as waiting on the fd via select/poll, to receive notifications when an async event has been reported.
Port	Identifies a physical port on the NIC.
Protection Domain (PD)	A protection domain provides process-level isolation of resources and is considered a fundamental security construct for Linux RDMA devices. A PD defines a boundary between memory regions and queue pairs. A network data transfer is associated with a single queue pair. That queue pair may access only a memory region that shares the same protection domain as itself. Protection domains provide security for memory regions accessed by both local and remote operations. Local access includes work requests posted to hardware command queues that reference memory regions. Remote access includes RDMA operations that read or write memory regions. A queue pair is associated with a single PD. The PD verifies that hardware access to a given LKey or RKey is valid for the specified QP and that the initiating or targeted process has permission to the LKey or RKey. Vendors may implement a PD using a variety of mechanisms but are required to meet the defined security isolation.
Memory Region (MR)	A memory region identifies a virtual address range known to the NIC. MRs are registered address ranges accessible by the NIC for local and remote operations. The process of creating an MR associates the given virtual address range with a protection domain, to ensure process-level isolation. Once allocated, data transfers reference the MR using a key value (LKey and/or RKey). When accessing an MR as part of a data transfer, an offset into the memory region is specified. The offset is relative to the start of the region and may be either 0-based or based on the region’s base virtual address.
MR LKey	The LKey is designed as a hardware identifier for a locally accessed data buffer. Because work requests are formatted by user space software and may be written directly to hardware queues, hardware must validate that the memory buffers being referenced are accessible to the application. NIC hardware may not have access to the operating system's virtual address translation table. Instead, hardware can use the LKey to identify the registered memory region, which in turn identifies a protection domain, which finally identifies the calling process. The protection domain of the processing queue pair must match that of the accessed memory region. This prevents an application from sending data from buffers outside of its virtual address space.
MR RKey	The RKey is designed as a transport identifier for remotely accessed data buffers. It's conceptually like an LKey, but the value is shared across the network. An RKey is associated with transport permissions.
Queue Pair (QP)	A queue pair is a transport addressable set of hardware queues, with a QP consisting of send and receive queues. A QP's address is identified as a queue pair number (QPN). In networking stack models, a QP is considered a transport layer object.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Object	Description
QP Receive Queue (RQ)	Receive queues are command queues belonging to queue pairs. Receive commands post application buffers to receive incoming data. Receive queues are configured as part of queue pair setup. An RQ is accessed indirectly through the QP when submitting receive work requests.
QP Send Queue (SQ)	A send queue is a command queue belonging to queue pairs. It represents a command queue for operations that initiate a network operation. A send queue may also be used to submit local commands that update hardware resources, such as updating memory regions. Network operations submitted through the send queue include message sends, RDMA reads, RDMA writes, and atomic operations, among others. Send queues are configured as part of queue pair setup. An SQ is accessed indirectly through the QP when submitting send work requests.
QP Type	QP type identifies the communication semantic between two QPs. Supported UET QP types include unreliable datagram (UD) and reliable unconnected (RD) communication (2.3.5.1.1).
Completion Queue (CQ)	A completion queue is designed to represent a hardware queue where the status of asynchronous operations is reported. Each operation that was triggered by a signaled work request results in a single entry written to the completion queue.
Address Handle (AH)	An address handle identifies the link and/or network layer address to a network port or multicast group. Address handle attributes identify the addressing for headers that reside under the transport header(s).
Global Identifier (GID)	InfiniBand defines a GID as an optional network-layer or multicast address. Because GIDs are large enough to store an IPv6 address, their use has evolved to support other transports. A GID identifies a network port, with the most well-known GIDs being IPv4 and IPv6 addresses.
GID Type	The GID type determines the specific type of GID address being referenced. Additionally, it identifies the set of addressing headers underneath the transport header. An RDMA transport protocol may be layered over different networking stacks. It may layer directly over a link layer (like InfiniBand or Ethernet), over the network layer (such as IP), or over another transport layer (such as TCP or UDP). The GID type conveys how the RDMA transport stack is constructed, as well as how the GID address is interpreted.
GID Index / GID Entry	RDMA addresses are securely managed to ensure that unprivileged applications do not inject arbitrary source addresses into the network. Transport addresses are injected by the queue pair. Network addresses are selected from a set of addresses stored in a source addressing table. The source addressing table is referred to as a GID table. The GID index identifies an entry into that table. The GID table exposed to a user space process contains only those addresses usable by that process. Queue pairs are frequently assigned a specific GID index to use for their source network address when initially configured.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

2.3.3 UET Verbs Object Model

UET leverages and extends the previously identified verbs objects. The existing RDMA verbs model is insufficient to support UET, so an additional set of verbs objects is required to support UET job-based communication. At a high level, Table 2-46 describes each of the new objects introduced.

Table 2-46 - UET Verbs Objects

Object	Description
JobID	A JobID identifies a distributed application. The JobID object is a device-level object and may be shared between processes. The configuration of a JobID object is considered a privileged operation. Note that a single, logical application-level job may require multiple UET verbs job objects (e.g., a job that spans multiple devices needs one job object per device).
Job Key	A job key associates a JobID with a specific protection domain. This provides secure access to the actual JobID value stored with the JobID, while restricting which memory regions data transfers may access.
Job Address Table	An address table is a virtual array of extended addresses associated with a JobID object. The address table allows local processes that belong to the same job to share addressing information of peer QPs.
Extended Address Handle	An extension of the address handle to represent both network (i.e., port, destination address, traffic class, etc.) and transport information (i.e., remote PIDonFEP, Resource Index, etc.).
QP Semantics	QP semantics are an extended set of QP attributes that describe the message and data ordering semantics supported by a QP, as well as other requirements for using that QP. QP semantics also defines a mechanism to learn the capabilities of a specific QP type on a device and aids in the selection of the UET ROD or RUD packet delivery types.

Figure 2-13 shows the access rules for supported MR models associated with the absolute and relative UE addressing modes. MR access is classified as unrestricted or restricted. With unrestricted, the MR spans a PIDonFEP, whereas restricted limits MR access to a specific JobID or Resource Index. For example, an MR accessed using relative addressing with restricted access is accessible only through the linked Resource Index, and the lookup for the MR is defined by a hierarchy of {JobID, PIDonFEP, Resource Index}. Alternatively, an MR accessed using relative addressing with unrestricted access is accessible by all RIs under a PIDonFEP, and the hierarchy is defined as {JobID, PIDonFEP}.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

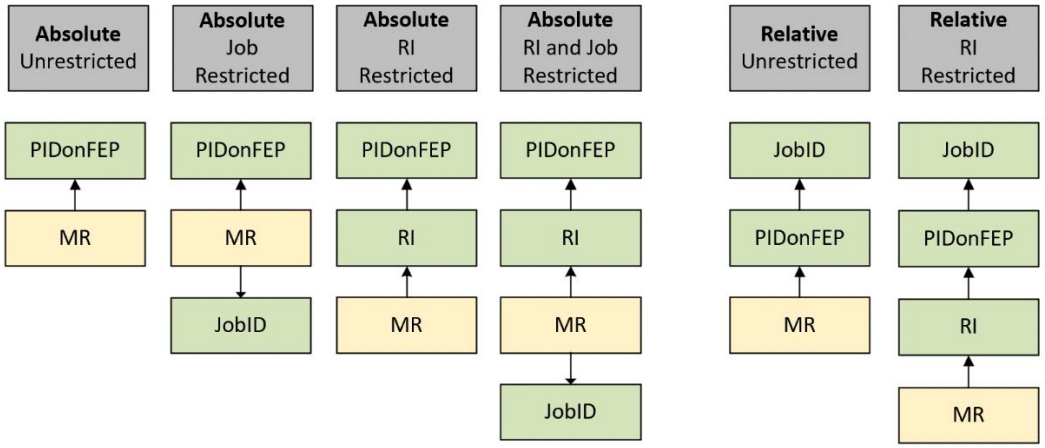


Figure 2-13 - UET Memory Region Access

Figure 2-14 provides a logical mapping to understand how existing and newly defined verbs objects map to UET. With UET, a JobID can be viewed as a global network identifier. The verbs JobID maps that identifier with a specific device. Finally, the job key (identified by a JKey) associates the JobID with a protection domain (PD) and provides a secure way to identify a JobID as part of a work request. All three conceptually capture the concept of a UET job.

A PD enables process-level isolation of hardware resources. It conceptually fulfills a similar role as PIDonFEP, and a UET PD is associated with exactly one PIDonFEP. If an application requires multiple PIDonFEP values, it must create one PD per value. Note that, if supported by a vendor, a single PIDonFEP may be associated with multiple PDs. Such a situation is useful in supporting multiple libraries within the same process space, with each library managing independent resources.

A QP represents a service at a Resource Index. A separate QP is needed for each Resource Index the application requires under a PIDonFEP (i.e., PD). Note that if multiple PDs are allocated that are associated with the same PIDonFEP, none of the QPs under those PDs may share the same Resource Index. Depending on the addressing model, access to a QP (i.e., PIDonFEP + Resource Index) is defined differently. For relative addressing it's {JobID, PIDonFEP, Resource Index}. For absolute addressing it's simply {PIDonFEP, Resource Index}.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

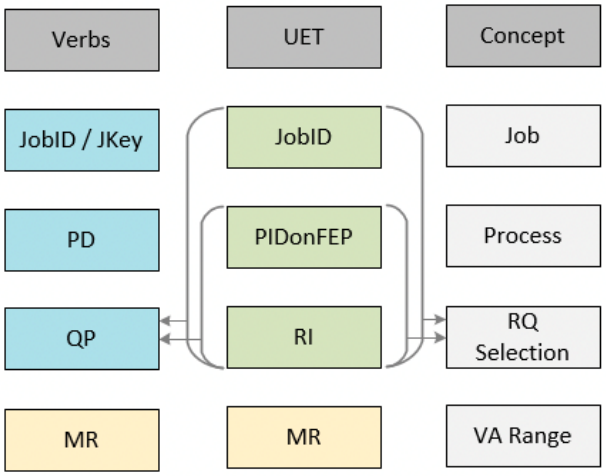


Figure 2-14 - Verbs to UET Object Mapping

Based on Figure 2-14, Figure 2-13 is updated to construct Figure 2-15 to show the MR access rules using the verbs objects.

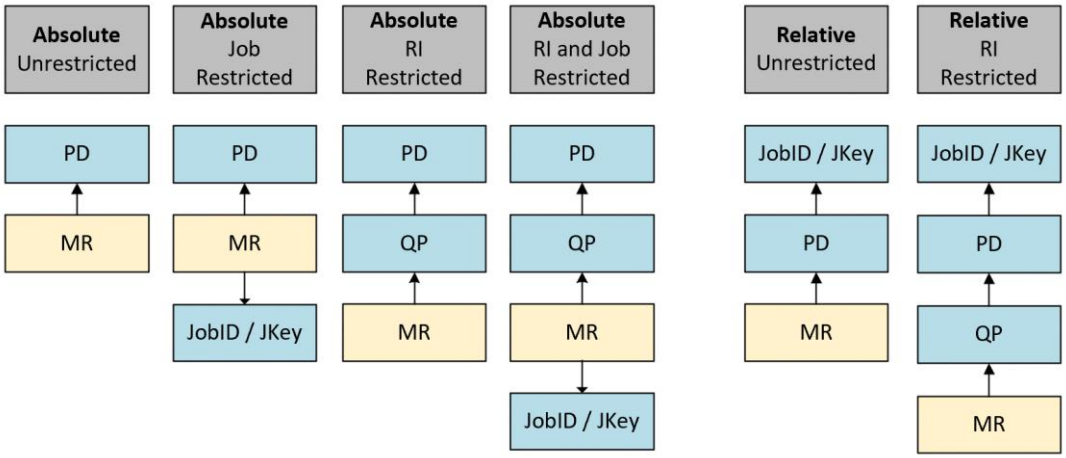


Figure 2-15 - Verbs Memory Region Access

Table 2-47 shows the exact access requirements used for the optimized and non-optimized RKey formats against both the relative and absolute addressing models.

Table 2-47 - Memory Region Binding Requirements

Addressing Model	Memory Key Type	Header Format	MR Access	MR Binding
Absolute Addressing	Optimized	Optimized non-matching	RI restricted	{ Device, PD, QP }
			RI job restricted	{ Device, PD, QP, JobID }

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Addressing Model	Memory Key Type	Header Format	MR Access	MR Binding
		Non-optimized	Unrestricted	{ Device, PD }
			Job restricted	{ Device, PD, JobID }
			RI restricted	{ Device, PD, QP }
			RI job restricted	{ Device, PD, QP, JobID }
	Non-optimized	Non-optimized	unrestricted	{ Device, PD }
			Job restricted	{ Device, PD, JobID }
			RI restricted	{ Device, PD, QP }
			RI job restricted	{ Device, PD, QP, JobID }
Relative Addressing	Optimized	Optimized non-matching	RI restricted	{ Device, JobID, PD, QP }
		Non-optimized	Unrestricted	{ Device, JobID, PD }
			RI restricted	{ Device, JobID, PD, QP }
	Non-optimized	Non-optimized	Unrestricted	{ Device, JobID, PD }
			RI restricted	{ Device, JobID, PD, QP }

Support for the different MR access restrictions is device-specific and exposed through device capability flags.

2.3.3.1 Absolute Addressing Model

Informative Text:
The absolute addressing model targets client/server where a server can communicate with many clients. Note that a single parallel job is considered a single client. It is anticipated that servers will

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

have their own assigned JobIDs and clients will use that JobID when sending messages to the server. Reverse traffic from a server to clients will use separate JobIDs per-client.

Figure 2-16 shows the relationships between UET verbs objects for absolute addressing. At the top level is the PD, which is associated with a PIDonFEP. Every QP and MR allocated under a PD will be associated with that PD's PIDonFEP. The server can receive a message on any JobID but can send only on job keys allocated under the PD.

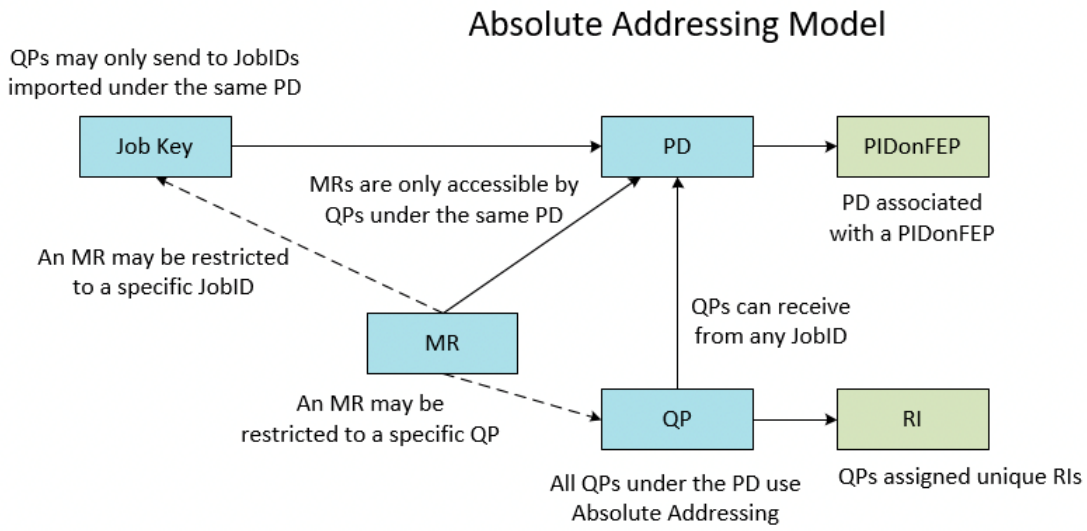


Figure 2-16 - Verbs Absolute Addressing Model

UET verbs maintains a flexible access model to MRs for absolute addressing on a server. MR access restrictions to a client can be:

- MR is unrestricted on the PIDonFEP. Any client may access it using any QP that shares the PD.
- MR is restricted to a client's JobID (i.e., job-restricted). Only that specific client may access it using any QP that shares the PD.
- MR is restricted to a QP (i.e., RI restricted). Any client may access it using only that QP.
- MR is restricted to a client's JobID and restricted to a QP (i.e., RI job restricted). Only that specific client may access it using only that QP.

2.3.3.2 Relative Addressing Model

Informative Text:
The relative addressing model targets a parallel application running under a specific JobID. Like absolute addressing, the PD is associated with a PIDonFEP. Additionally, it's associated with a JobID. All QPs will be associated to their PD's JobID and PIDonFEP. Similarly, MRs will receive only data that target the PD's JobID and PIDonFEP. Messages sent can target only job keys allocated under the PD. A job key may reference a different JobID than that associated with the PD. This enables a parallel application to communicate with a server as a client, for example.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Figure 2-17 shows the relationship between verbs objects for Relative Addressing.

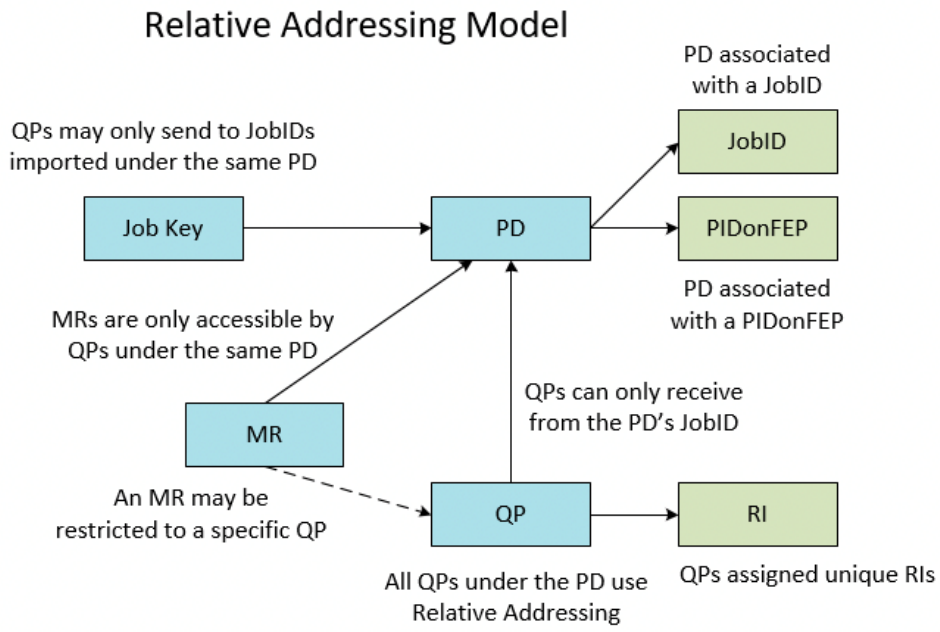


Figure 2-17 - Verbs Relative Addressing Model

UET verbs maintains a flexible access model to MRs for relative addressing. MR access restrictions can be:

- MR is unrestricted on the PIDonFEP. Any peer may access it using any QP that shares the PD.
- MR is restricted to a QP (i.e., RI restricted). Any peer may access it using only that QP.

2.3.3.3 Mapping UET SES Fields

Most SES header fields map directly to UET verbs concepts. Table 2-48 defines the conceptual mapping between SES header fields and UET verbs. Actual API details may differ slightly from that shown.

Table 2-48 - SES Field Mapping to UET Verbs

SES Field	Verbs Mapping	Note
opcode	ibv_send_wr::opcode	Message operation used for a send work request
ver	SGID entry type	The type assigned to the SGID entry implies the UET version being used
DC	ibv_send_wr::send_flags IBV_SEND_DELIVERY_COMPLETE	New send flag for a send work request
rel	QPN bit 31 – absolute/relative addressing	UET-defined format for QPN (see Section 2.3.5.1.3). Absolute addressing is enabled by setting this bit.
JobID	ibv_job and ibv_job_key	New job objects
PIDonFEP	QPN bits 11:0	UET-defined format for QPN
resource_index	QPN bits 23:12	UET-defined format for QPN

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

SES Field	Verbs Mapping	Note
buffer_offset	ibv_send_wr::remote_addr	Offset into the target buffer to write to with zero-based addressing on the memory region
initiator	ibv_qp_init_attr_ex::src_id	UET-extended QP attributes
memory_key	ibv_mr::rkey64 ibv_send_wr::rkey64	UET-extended MR and send work requests
header_data	ibv_send_wr::imm_data64 ibv_wc::imm_data64	UET-extended send work requests and work completions
request_length	ibv_send_wr::ibv_sge[]::length	Total length of the data being sent in a message (i.e., total across all SGEs)

2.3.4 Device Discovery

UET verbs can leverage the RDMA verbs infrastructure to report devices that support UET with minimal changes. This re-uses the device, device context, command fd, async event fd, and port objects.

2.3.4.1 GID Type

GIDs identify addressing information underneath the SES and PDS headers and identify the transport address used by a queue pair. UET verbs defines the following GID types.

- **UET_UDP** - Indicates that UET is layered over UDP/IP headers using the UET identified UDP port.
- **UET_IP** - Indicates that UET is layered over the IP header, plus the PDS entropy header.
- **UET_UFH** - Indicates that UET is layered over the UFH header.

Devices that support UET MUST support one or more of the UET GID types. Applications will select the UET protocol by choosing an appropriate GID entry.

2.3.5 UET Services

A single UET service is exposed through UET verbs as a queue pair object. Each allocated QP represents a PIDonFEP + Resource Index combination.

2.3.5.1 Queue Pairs (QPs)

2.3.5.1.1 QP Types

UET verbs introduces a reliable unconnected (RU) QP type. UET also supports the unreliable datagram (UD) QP type. The QP type is paired with a specific GID entry to identify the transport and network protocols. Supporting combinations are:

- UD QP + UET UDP GID
- UD QP + UET IP GID
- UD QP + UET UFH GID
- RU QP + UET UDP GID
- RU QP + UET IP GID
- RU QP + UET UFH GID

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

2.3.5.1.2 QP States

UET QPs expose a simplified QP state model to UET verbs users. Valid UET QP (UD or RU) states are:

- RST (Reset) – QP is allocated and not yet configured.
- RTS (Ready-to-Send) – QP is fully configured and may actively send and receive data transfers.
- ERR – An error has occurred on the QP, and all data transfers are disabled.

The simplified QP state machine is shown in Figure 2-18. It overlays the existing state machine apart from two new added paths. The first path that can be followed is when the QP is created. Normally when a QP is created, it starts in the RST state. For UET verbs, all the required attributes may be specified when creating the QP, and the application can directly specify the RTS state as the initial state of the QP. A second path is a direct connection from RST to RTS. This represents a modify QP call that skips over the INIT and RTR states. If an underlying error occurs on the QP, the NIC hardware may put the QP into the ERROR state, which stops all processing. An errored QP is recoverable only by recycling it back through the RST state.

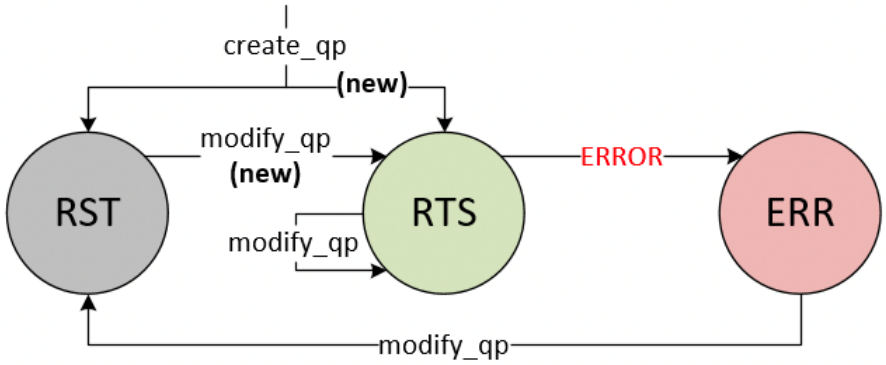


Figure 2-18 - QP State Machine

2.3.5.1.3 QP Numbers

For UET verbs, the QPN is formatted to contain additional addressing information used for targeting a UET service. The QPN is formatted as follows:

```
QPN = [ ABS/REL 1 bit | 7 bits of 0 | RI 12 bits | PIDonFEP 12 bits ]
```

The upper bit is a flag that specifies if absolute or relative addressing is being used. When set, the QP receives messages using absolute addressing. Additionally, both the Resource Index and the PIDonFEP are encoded in the lower 24 bits of the QPN. The remaining seven bits are reserved.

When creating a QP, the application may request a specific QPN if it knows what PIDonFEP and Resource Index is required for the service. Alternatively, the application may create a QP without specifying a QPN, resulting in the provider driver dynamically assigning the values.

In either case, the PIDonFEP and Resource Index values encoded in the QPN are validated by the provider kernel driver against assignments established by a privileged entity. This privileged assignment may occur at job initialization time, such as through a privileged job launcher, or through a privileged

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

provisioning process during endpoint address assignment, as described in the UE libfabric specification (see Section 2.2). The provider kernel driver MUST reject QP creation requests that specify PIDonFEP and Resource Index values already allocated or not authorized for the requesting process.

2.3.5.1.4 Extended QP Attributes

UET verbs extends RDMA QP attributes. To create a UET QP, applications specify all required QP attributes at the time of QP allocation. The returned QP is in the RTS state and is able to accept posted send and receive work requests.

UET QPs require additional attributes beyond those defined for RDMA QPs. The following extensions are identified:

- **Source QPN** – Allows the application to specify the QPN value required for the QP being created.
- **Source Identifier** – QP attributes are extended with an identifier field that maps directly to the SES initiator field.
- **QP Semantics** – Further refine message and data ordering semantics if different from the defaults as supported by the QP type.

2.3.5.2 Completion Queues (CQs)

Applications retain all the existing flexibility in verbs regarding QP+CQ allocation and management for UET verbs. QPs can be created as send only or receive only, specify separate CQs for the RQ/SQ, share a CQ between the RQ/SQ or even across QPs, etc.

When an application requires multiple Resource Indexes on a PIDonFEP be available, it must create a separate QP for each Resource Index.

Figure 2-19 shows examples of possible QP configurations.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

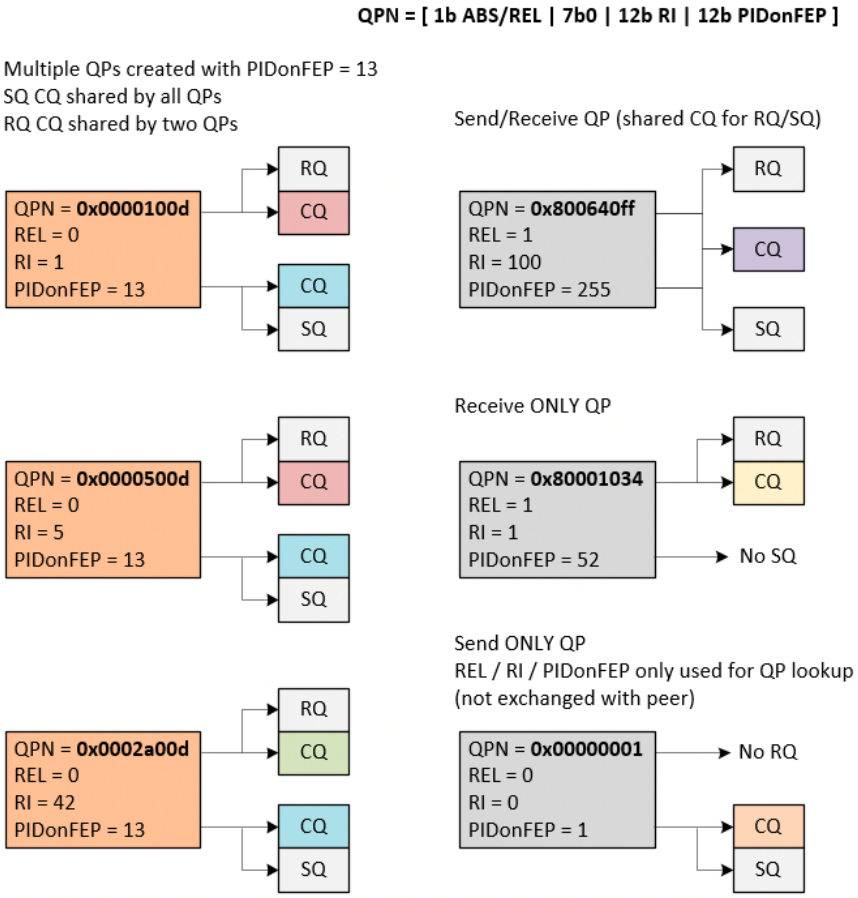


Figure 2-19 - CQ Management

2.3.5.3 Addressing

Verbs logically separates addressing data into two components. This separation is aligned with the network layering model where the addressing is used. Addressing data that resides in UET SES headers is referred to as QP-level transport addressing. Addressing information found in underlying headers (e.g., UDP/IP) are referenced as network-level addressing. Network-level addresses are described using GIDs and address handle (AH) attributes. In general, a network-level address identifies a network port. UET QP-level transport addressing includes the JobID, PIDonFEP, Resource Index, and addressing mode.

To align with RDMA verbs semantics, UET verbs specify the network-level and QP-level components separately. The network and QP components are then combined in an extended address handle to form the target QP address.

2.3.5.3.1 Extended Address Handles

An extended address handle in verbs is used to represent both network and transport information required to target a remote service. The information specified in an extended address handle:

- Network information
    - Network port

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

    - Source GID (source network address)
    - Destination GID (destination network address)
    - Traffic class
- Transport information
    - Addressing mode (absolute vs relative)
    - PIDonFEP
    - Resource Index

The transport information is specified in the extended address handle as a single remote QPN value. As previously described with the QPN encoding, the QPN contains addressing mode, PIDonFEP, and Resource Index.

2.3.5.4 Examples

Figure 2-20 and Figure 2-21 provide a more concrete example of the UET verbs object model and how multiple services can be managed by an application. Colors are used for identifying resources bound together via a JobID or job key.

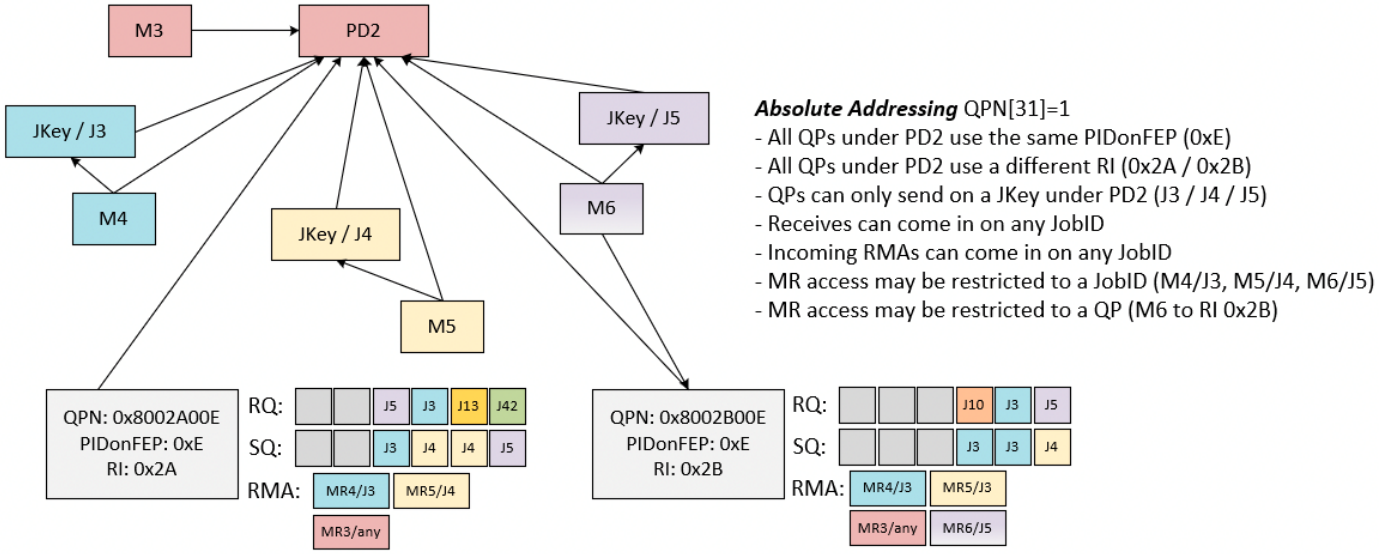


Figure 2-20 - Verbs Example Absolute Addressing

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

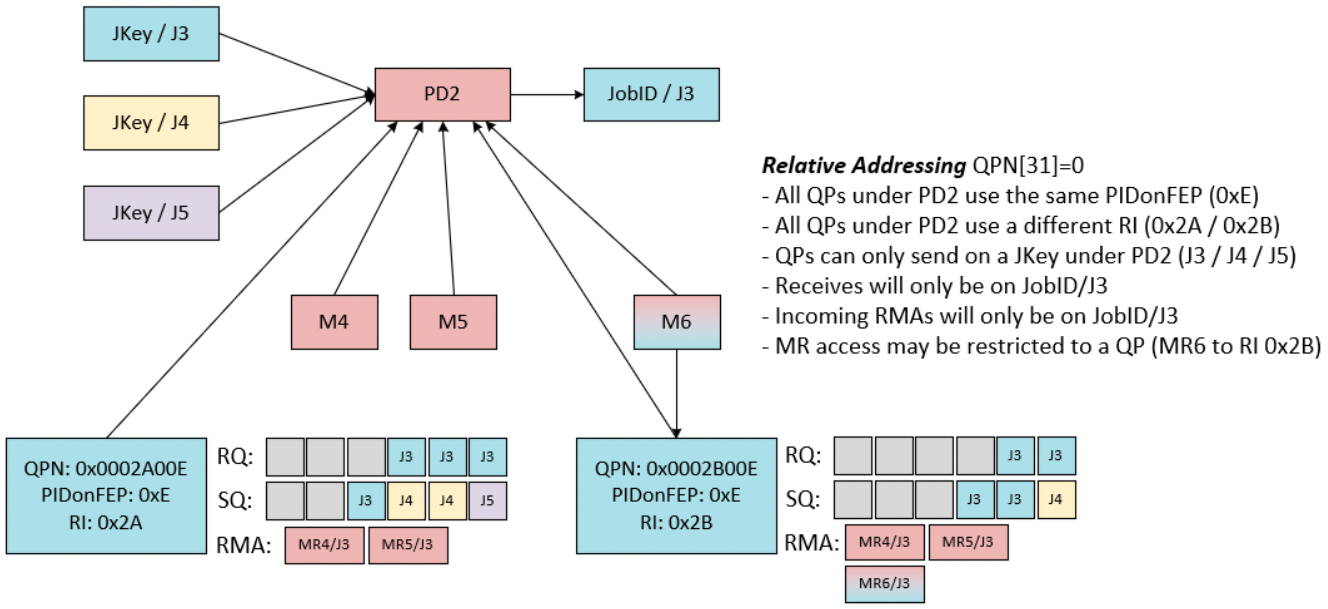


Figure 2-21 - Verbs Example Relative Addressing

2.3.6 JobIDs

Support for job-based communication introduces new verbs objects. A job object is a device-level object that may be shared between processes. The configuration of a job object is considered a privileged operation.

The maximum number of JobIDs that a device supports is advertised as an extended device attribute.

2.3.6.1 Job Attributes

A job object MUST be associated with a JobID value, port number, and source GID entry. The JobID value maps directly to the JobID carried in the SES header. The source GID entry identifies the transport protocol and source network address.

2.3.6.2 Address Table

An address table identifies a set of peer QPs belonging to a JobID. Each entry in the table contains the extended address attributes used to target a remote service. Each address inserted into the table is associated with an address index. The address index is specified when submitting send work requests to identify the target QP.

The configuration of an address table within a job object is considered a privileged operation. Operations include adding, replacing, removing, and querying an address at a specific index.

A JobID's address table is automatically shared between all processes within the same job.

Support for address tables is optional.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

2.3.7 Job Keys

Job keys functionally import a device-level job object into a specific protection domain. A job key provides a secure mapping between a user-accessible value and the corresponding transport JobID. QPs may access JobIDs only where the job key and the QP share the same PD. Job keys enable processes to share job information, including the address table, while maintaining separate memory registrations.

2.3.7.1 JKey

A JKey is a vendor-assigned value to a job key. A JKey is to a JobID as an LKey is to an MR. The JKey value is submitted as part of a work request to target a specific job. When address tables are being used, a JKey is paired with an address index to identify a target QP.

2.3.8 Work Requests

Work requests over UET QPs require additional information, including larger-sized data fields, than those provided by RDMA verbs. UET verbs defines extended QP operations to configure work requests. Extended operations are required to support:

- 64-bit immediate data
- 64-bit RKey and LKey
- JobID (i.e., JKey)
- Extended address handle
- Address table index (i.e., index into the JobID's address table when the extended address handle is not specified)

2.3.9 Immediate Data Support

A NIC may support RDMA and UET protocols. Such NICs may implement transport neutral methods for handling immediate data. To allow NIC implementations that are optimized for legacy RDMA protocols, the UET verbs implementation may expose device restrictions on immediate data used by applications. Restrictions may include:

- Preventing an RDMA write with immediate data to consume a receive buffer at the target.
- The maximum size of immediate data written to the CQ.

A new device capability is exposed that specifies if 64-bit immediate data is supported. If it is supported and the underlying transport (i.e., UET) also supports 64-bit immediate values, then the application can use 64-bit values. If not supported and the application uses a 64-bit value, the upper 32 bits will get truncated. Similarly, if the transport does not support 32-bit immediate values, yet the device advertises 64-bit support, any 64-bit value used by the application over that transport will get truncated.

2.3.10 Work Completions

UET verbs defines extended CQ operations to report larger and additional completion data. Extended operations are required to report:

- 64-bit immediate data
- JobID

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

- SourceID

2.3.11 Memory Registration

UET verbs extends RDMA verbs memory registration with UET-focused capabilities. UET verbs memory registration extensions are outlined below. Extended memory registration capabilities are aligned with UET libfabric support.

There are three main configurations of MRs used to control the granularity of access:

- **MR + PD** – This is the traditional PD model where only QPs that are part of the same PD can access the MR.
- **MR + PD + JobID** – This protects the MR even further, allowing an MR to be bound to a JobID. Only operations within the JobID targeting a QP in the same PD can access the MR.
- **MR + PD + QP** – This provides even further protection of an MR by attaching it to the QP. Only that QP has access to the MR.

The above configurations provide different levels of MR protection for incoming requests at the target. Alternatively, as an initiator, the only restriction for sending data from an MR in an outgoing request is that the QP and MR must be within the same PD. An application can send data on a QP from any MR under the same PD, to any JobID with a JKey allocated within that PD, regardless of whether the MR was bound to a specific JobID.

2.3.11.1 Attach / Detach MR

UET verbs allows a vendor to expose an MR to the network as a separate operation from its creation. A vendor can indicate this requirement through a QP usage flag. Exposing an MR to the network requires attaching an existing MR to a QP. Network access can be revoked by detaching the MR. MR attach and detach functions are UET verbs extensions to QP operations. The behavior of attaching an MR to a QP is aligned with the libfabric FI_MR_ENDPOINT requirement (see Section 3.4.2.3.6).

2.3.11.2 Job Based MRs

An MR may optionally be associated with a specific job by specifying the job key when allocating the MR (i.e., job-restricted). When a job key is specified, only incoming requests within that job will be allowed to access the MR. The QP, MR, and job key must all belong to the same PD.

2.3.11.3 Derived MRs

Derived MRs are new memory regions that are wholly contained within an existing region but may have different access rights or other attributes. A derived region must be within the same PD as the existing region. Deriving a region allows a vendor to share page mappings or other hardware resources between the two regions. UET verbs extends memory registration to support derived memory regions. Derived MRs are an optional optimization that a vendor may support. UET devices MAY support the semantics of derived regions, even if underlying optimizations are not available.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

2.3.11.4 User RKeys

UET verbs will allow for application selected RKey values. Support for user RKeys is device-specific and exposed through a device capability flag. An application that makes use of user RKeys is responsible for ensuring that the RKey value is usable for any transport that may access the MR.

2.3.11.5 64-bit LKeys and RKeys

Devices that support UET MUST return a 64-bit RKey value that adheres to the UET specified RKey format. A multi-transport device may also return a valid 32-bit RKey for use with legacy RDMA transports. A 64-bit LKey MUST be returned if a 64-bit RKey is returned. The number of LKey bits that are non-zero is vendor-specific.

2.3.12 QP Semantics

QP semantics define a set of device queryable attributes, which can be modified by the application and provided into QP creation. QP semantics define message and data ordering semantics implemented by a QP type over a specific transport. This includes various combinations of:

- [Send | Read | Write] after [Send | Read | Write]

RDMA verbs require that a developer reference RDMA specifications to understand data transfer ordering. UET verbs exposes data ordering as new QP attributes. Additionally, applications can remove unneeded ordering requirements prior to creating a QP. Vendors that are able can optimize transfers around necessary ordering semantics.

UET devices SHOULD use the QP ordering semantics to guide protocol selection between ROD or RUD.

UET verbs QP semantics are aligned with libfabric endpoint attributes.

2.3.13 References

[1] InfiniBand Trade Association, InfiniBand Architecture Specification, Volume 1, Release 2.1, Beaverton, OR, USA, 2025.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

2.4 In-Network Collectives (INC)

High-performance computing (HPC) and artificial intelligence (AI) workloads frequently require multiple accelerators to coordinate as groups of cooperating processes that accomplish computational objectives. In the message-passing interface (MPI) and most collective communication libraries (CCLs), we refer to these groups as communicators, with each process in a group called a rank. Each rank receives an integer ID, ranging from 0 to r-1 for a group of r ranks. Ultra Ethernet (UE) adopts the same abstraction for the purpose of facilitating collective operations.

Ultra Ethernet In-Network Collectives (INC) enables network switches to perform computational tasks directly for a range of group operations. These switches can accelerate key collective operations defined by the MPI and various CCLs. While this overview does not comprehensively list all such operations, the MPI specification has additional details. This INC specification focuses on a subset of collective operations from MPI and CCLs, demonstrating the adaptability and extensibility of the INC framework for high-performance computing environments.

INC can reduce the bandwidth required for large-message reduction operations by a factor of two, as established in the literature [1]. This reduction in data movement immediately decreases energy consumption at the endpoints. By executing computations within the network, INC also lessens the computational burden and alleviates memory load on accelerators located at the endpoints. Additionally, INC can significantly reduce latencies for small operations, further enhancing performance and efficiency in high-performance computing environments.

Collective operations are initiated by application software on a communicator. They are considered active from when the first call issues by a rank in the communicator until the final completion notification is received by a rank in the communicator.

INC offers support to accelerate the following collective operations in switches:

- Barrier
- Broadcast
- Allreduce
- Reduce

These basic operations can be used to accelerate more complex operations such as reduce-scatter.

2.4.1 Assumptions

UE makes several assumptions about the definition and deployment of INC operations:

1. UE INC operates on a single logical switch with a single switch endpoint (SEP) in the network. Vendors may offer solutions where this single logical switch consists of a distributed set of switches, provided it adheres to the behavior specified herein.
2. UE INC MUST operate over a lossless fabric (using PFC or CBFC) over the UET ROD transport between the FEP and the SEP. Each FEP MUST be directly connected to a port associated with a SEP.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

3. Packets carrying INC reduction operations (allreduce and reduce) MUST carry full elements of the specified datatype (i.e., a single element of the operation must be carried in one packet).
4. All FEPs in an INC communicator are associated with the same job.
5. The operands carried in the payload MUST use little-endian byte order. Operands MUST NOT be padded[1].

INC is composed of the following components:

- INC manager (IM): A daemon that facilitates collective offload.
- UET.
    - SES deferrable send.
    - PDS reserved ROD service.
    - Non-interleaved messages.
    - TFC (optional).
    - TSS (optional).
- Group(s) of UET fabric endpoints (FEPs) called communicator(s).
- One or more logical SEPs.
    - Each SEP supports the INC switch.

The INC switch requirements are defined in Section 2.4.10.

2.4.2 Theory of Operation

The IM establishes a communicator by grouping member FEPs and associating the group with a SEP, assigning a unique collective ID (CID) upon creation. The INC switch uses the CID to disambiguate communicators. The INC switch filters packets based on JobID/CID combinations.

Some applications require specific orders of operator application. Optionally, each communicator can have constraints on the order of operations.

Upon communicator creation, the CSN is set to 0. The establishment of the required transport resources is coordinated by the IM, as described in Section 2.4.12.

When a new operation is started, a rank uses the current CSN and increments it by one, wrapping from the maximum 12-bit value back to 0. To start a collective operation, a rank issues a UET_DEFERRABLE_SEND with the defined INC field (see Section 2.4.11.1) that encodes the CSN, type of operation, and necessary parameters.

The SEP receives the input data, performs the operation defined by the INC field on the group identified by the CID, and sends the output data to the recipients. The recipients of the output data depend upon the operation. For example, the allreduce collective requires the SEP to send the data to all ranks of the communicator. The reduce collective requires the SEP to send the data to only one specified root node. The broadcast collective requires the SEP to send the data to all ranks except the root node. Optionally

[1] Composite operands, such as those used for MINLOC and MAXLOC, may contain padding. No padding is added to the payload before or after any of the operands.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

the root node can also receive the output data if forced output is requested in the INC operation. INC communicator ranks MAY post a receive buffer that is identified using a tag with the **inc_info** format (see Section 2.4.11.1) before sending their input data. Ranks asynchronously issue their operations, and a switch will block an operation until all ranks have contributed their data.

The maximum number of outstanding collective operations (i.e., operations that a rank may initiate without waiting for a completion) is subject to the SEP's capabilities and MUST NOT exceed the size of the CSN space (i.e., 4096 total outstanding collective operations).

Informative Text:

While this specification does not define how to implement a logical INC switch with multiple smaller switches, one could use simple tree topologies and a similar protocol to achieve this. The fact that the collective root is carried in the matching bits simplifies that implementation.

2.4.3 INC Operations

Table 2-49 defines the operations for INC.

Table 2-49 - INC Collective Operations

Name	Opcode	Req	Description
UET_INC_REDUCE	0x00	MUST	Reduces values to the root rank
UET_INC_ALLREDUCE	0x01	MUST	Reduces values to all ranks
UET_INC_BCAST	0x02	Note[1]	Distributes values to all ranks
UET_INC_BARRIER	0x03	MUST	Synchronizes all ranks
UET_INC_EXSCAN	0x04		Exclusive scan
UET_INC_SYNCBCAST	0x05	Note[1]	Synchronizes all ranks and distributes values to all ranks
Reserved	0x06-0xEF		Reserved
Vendor defined	0xF0-0xFF		Vendor-specific operations

Note:

1. An implementation MUST support at least one broadcast implementation, either UET_INC_BCAST (where non-root FEPs do not synchronize with the SEP, enabling pipelining of multiple BCAST operations) or UET_INC_SYNCBCAST (where non-root FEPs synchronize with the SEP using a zero-sized message before starting the operation).

Informative Text:

The UET_INC_SYNCBCAST operation may lend itself to a simpler switch implementation because FEPs typically post receive buffers before the broadcast arrives, eliminating the need for the deferrable send to be deferred. In contrast, the UET_INC_BCAST operation allows more flexible use of the operation by not forcing a synchronization point.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

2.4.4 Datatypes

The supported atomic datatypes are shown in Table 3-28.

2.4.5 Reduction Operators

A subset of the operations defined in Section 3.4.7.2 is available for INC reductions. The encoding of the INC reductions is kept consistent with the atomic operation encoding while replacing some codepoints that do not apply to INC (See Table 3-27).

Table 2-50 - INC Reduction Operations

Name	Constant	Description
UET_INC_MIN	0x00	Minimum: Output = MIN(Input1, Input2)
UET_INC_MAX	0x01	Maximum: Output = MAX(Input1, Input2)
UET_INC_SUM	0x02	Sum: Output = Input1 + Input2
UET_INC_SUM_SCALE	0x03	Sum and divide by a 32-bit unsigned integer factor configured by IM for communicator
UET_INC_PROD	0x04	Product: Output = Input1 * Input2
UET_INC_LOR	0x05	Logical OR: Output = Input1 || Input2
UET_INC_LAND	0x06	Logical AND: Output = Input1 && Input2
UET_INC_BOR	0x07	Bitwise OR: Output = Input1 | Input2
UET_INC_BAND	0x08	Bitwise AND: Output = Input1 & Input2
UET_INC_LXOR	0x09	Logical XOR: Output = Input1 xor Input2
UET_INC_BXOR	0x0A	Bitwise XOR: Output = Input1 ^ Input2
UET_INC_MINLOC	0x0B	Minimum value and location (on paired types)[1]
UET_INC_MAXLOC	0x0C	Maximum value and location (on paired types)[1]
Reserved	0x0D-0xEF	Reserved
Vendor specific	0xF0-0xFE	Vendor-defined reduction operation
Reserved	0xFF	Reserved

Note:

1. The location is UET_TYPE_UINT32. If the value and the location are of different sizes, the smaller of the two is padded to match the other's size. The padding is added after the element. It is set to zero by the sender and ignored by the receiver.

Informative Text:

The sum-and-scale operation is a generalized version of average, in that the factor by which the final division is performed is configurable. Setting said factor to the number of ranks in the group makes sum-and-scale work the same as average.

The required combinations of datatypes and operations are defined in Table 2-51.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Table 2-51 - Required Datatype Combinations

Name	UET_INC_MIN	UET_INC_MAX	UET_INC_SUM	UET_INC_LOR	UET_INC_LAND	UET_INC_BOR	UET_INC_BAND	UET_INC_LXOR	UET_INC_BXOR
UET_TYPE_INT32	MUST	MUST	MUST	MUST	MUST	MUST	MUST	MUST	MUST
UET_TYPE_UINT32	MUST	MUST	MUST	MUST	MUST	MUST	MUST	MUST	MUST
UET_TYPE_BF16	MUST	MUST	MUST						
UET_TYPE_FP16	MUST	MUST	MUST						

Informative Text:

This specification does not mandate an internal accumulator length. It is recommended to select a sufficiently large datatype for accumulation.

2.4.6 Operator Ordering Modes

INC implementations MUST support at least one of the following operator ordering modes when configured through the IM:

- **No ordering**: No guarantee of the reduction order.
- **Consistent ordering in the same communicator**: For the same communicator, the switch always applies the same order of reduction operations.
- **Consistent ordering across different communicators**: For different communicators with identical attributes in the same job, the switch always applies the same order of reduction operations.
- **Consistent ordering across different communicators and jobs**: For different jobs with identical node allocations and different communicators with identical attributes in those jobs, the switch always applies the same order of reduction operations.
- **Natural ordering**: The switch reduces in the order of ranks beginning from 0 to p-1. For example p=4: sum(sum(sum(0,1),2),3); note that this mode limits parallelism and may lead to lower performance.

2.4.7 Operator Rounding Modes

INC implementations MUST support the following floating-point rounding modes:

- Round to nearest, ties to even.

INC implementations MAY support any of the following floating-point rounding modes:

- Round to nearest, ties away from zero.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

- Round toward 0.
- Round toward plus infinity.
- Round toward minus infinity.

2.4.8 INC Libfabric

The creation of a communicator is initiated by the ranks calling *fi_join_collective()* with matching arguments. Figure 2-22 shows the sequence of exchanges between the IM, an exemplary rank, and an INC switch during INC communicator creation.

After all ranks send a *CommunicatorJoinRequest*, the IM allocates fabric resources for the operation of the INC communicator, subject to immediate availability and, optionally, any vendor-specific policies.

The IM configures each switch individually and sends a *CommunicatorJoinReply* to the communicator ranks with the information required to communicate with their adjacent SEPs.

The rank then allocates an initiator PDS and performs a UET_INC_BARRIER collective, which forces the establishment of both the initiator and target PDCs.

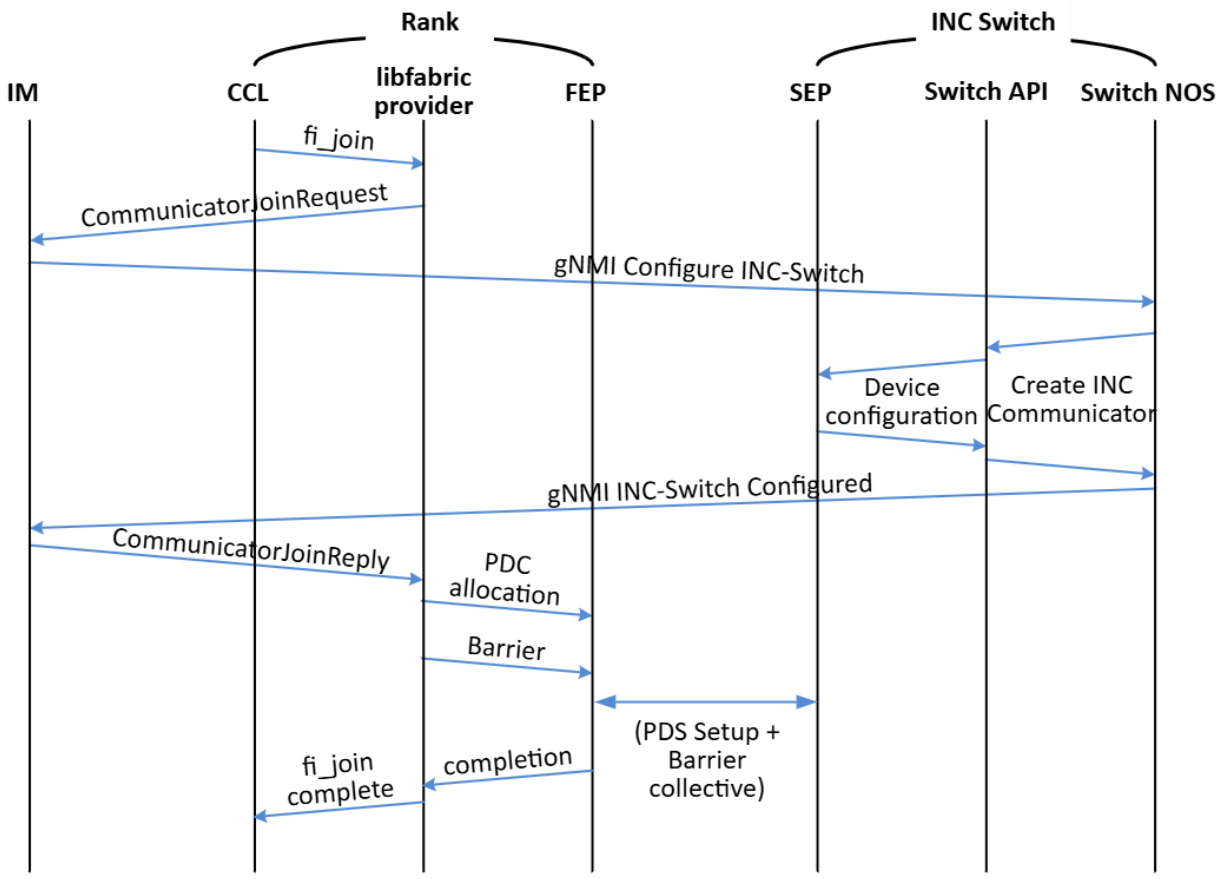


Figure 2-22 - Example INC Communicator Creation

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

The communication between the libfabric provider and the IM is specified in Section 2.4.15. The communication between the IM and the switch is specified in Section 2.4.17.

After all collective operations of the communicator have completed, the ranks request that the IM tear down the INC communicator. Figure 2-23 depicts the sequence of steps taken by the IM, a rank, and an INC switch.

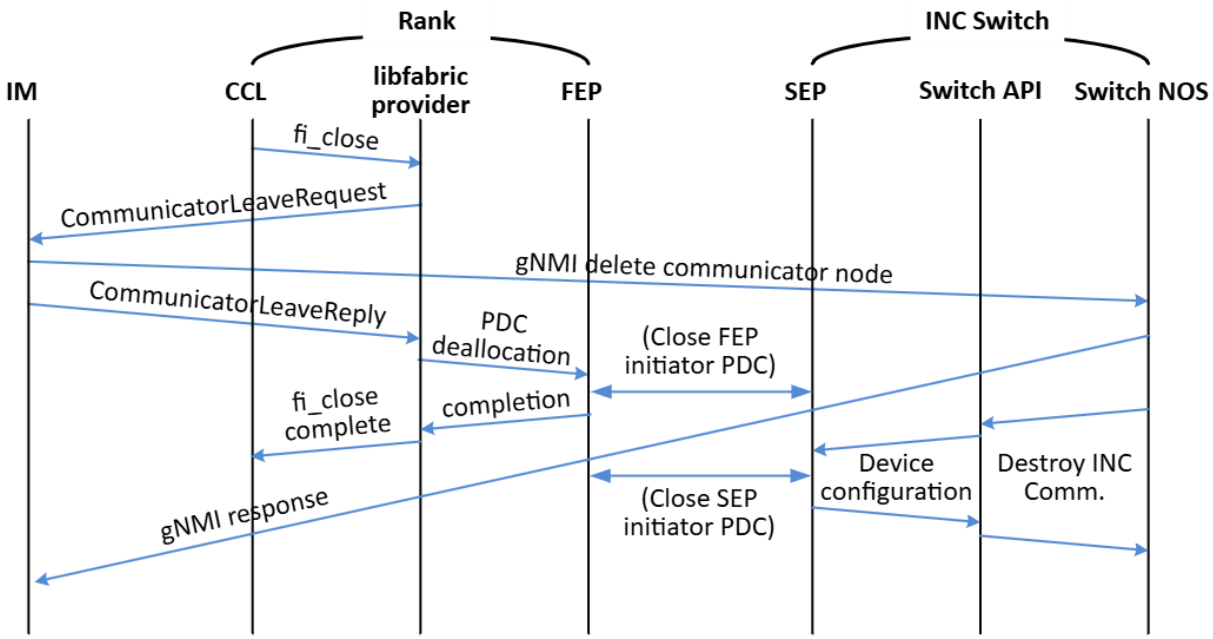


Figure 2-23 - Example INC Communicator Teardown

The IM first waits for all ranks to produce a *CommunicatorLeaveRequest* to allow for all collectives to complete. Afterward, it starts independent procedures at both the rank and the switch.

The rank receives a *CommunicatorLeaveReply*, whereupon the initiator PDC is closed.

The INC switch is directed to terminate the INC communicator. Following this, the switch adjusts its internal configuration, disabling INC offload for the communicator, and closes the SEP's initiator PDC.

The libfabric provider MAY implement a mechanism to release target PDCs that the switch failed to close. Such a mechanism is beyond the scope of this document.

The IM can also tear down an INC communicator for any of the following reasons:

- A rank signals an error.
- Conditions are detected that preclude continued INC communicator operation (e.g., link failures).
- Administrative action or policy.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

2.4.8.1 Composite Collective Operations

The libfabric provider decomposes an offloaded collective operation into multiple INC operations in the following cases:

1. Allgather operations (*fi_allgather()*) are performed as a series of broadcast operations where each member is designated as the source of the input data for one of the broadcast operations.
2. Reducescatter operations (*fi_reduce_scatter()*) are performed as a series of reduce operations where each member is designated as the destination for the output data for one of the reduce operations.

2.4.9 Libfabric Provider Extensions

An application utilizing INC collectives provides a 16-byte unique ID when the address vector (AV) set is created, by passing a structured communication key (*comm_key*) parameter of the AV set attributes. The communication key is passed in the *fi_av_set()* API call within the *attr* parameter.

The structure of the *comm_key* for INC collectives is:

```
/*
 * UET INC comm_key
 */
struct uet_inc_comm_key {
    uint8_t version;
    uint8_t unique_id[16];
} __attribute__((packed));
```

For INC collectives, the INC *comm_key* MUST be used and the *comm_key_size* parameters MUST be set to the key's size (i.e., *sizeof(struct uet_inc_comm_key)*). The communication key MUST be filled as follows:

- *version*: set to 1.
- *unique_id*: set to a Unique ID chosen for the communicator.

2.4.10 INC Switch Required and Optional Capabilities

A switch MUST support the following capabilities to be INC switch compliant:

- SES functions:
  - Header types: UET_HDR_REQUEST_STD, UET_RESPONSE
  - Opcodes: UET_DEFERRABLE_SEND, UET_DEFERRABLE_RTR
- PDS functions:
  - Reserved ROD PDCs
    - ROD_REQ, ACK, NACK, CP (close commands)
  - UET CRC

A switch MAY support the following capabilities:

- CMS functions: TFC

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

- TSS functions

2.4.11 INC Transport

INC leverages existing UET services and introduces a compact extension field to a semantic sublayer header to carry INC-specific information. This extension field conveys fields such as the communicator ID, collective sequence number, and operation type.

2.4.11.1 Semantic Operation

INC uses UET_DEFERRABLE_SEND and UET_DEFERRABLE_RTR operations for data exchange. All necessary information for the collective operation is encoded in the 64-bit **ses.inc_info** field (see Figure 2-24).

The UET_HDR_REQUEST_STD (Figure 3-11 and Figure 3-12) is used for deferrable send operations. In UET_HDR_REQUEST_STD packets, the **ses.match_bits** field is replaced by the 64-bit **ses.inc_info** field. While the **ses.match_bits** fields in traditional send operations inform the destination where to place data and match messages, the **ses.inc_info** field informs the SEP how to process the data.

Informative Text:
The **ses.inc_info** field is overlayed upon the **ses.match_bits** field and it is difficult for an independent third party (e.g., a wireshark packet capture) to disambiguate the contents of these fields. A switch understands how to process the **ses.inc_info** because it is aware of the {FA, JobID, PIDonFEP, RI} tuple in the request.

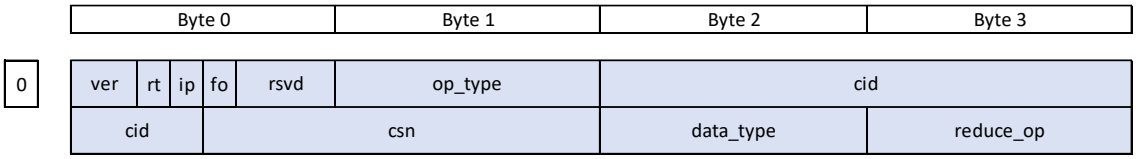


Figure 2-24 - INC Operation ses.inc_info Encoding

Table 2-52 - INC Operation ses.inc_info Encoding

Field Name	Size (bits)	Description	Reference
ver	2	INC protocol version number (set to 0x00).	
rt	1	Used only for operations with a single output (e.g., UET_INC_REDUCE) or single input (e.g., UET_INC_SYNCBCAST). The rank that solely receives the output, or solely provides the input, sets this bit to one; all other group members set it to zero.	
ip	1	When set to one, the operation is in place, and the send and receive buffers at the FEP are the same. This flag indicates that once output data has been delivered to at least one rank, any duplicate messages carrying the corresponding input data may no longer contain valid data.	

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Field Name	Size (bits)	Description	Reference
fo	1	Forced output. When set to 1, BCAST and SYNCBCAST root ranks expect output data.	
rsvd	3	Reserved.	
op_type	8	INC operation type.	Table 2-49
cid	20	Communicator ID.	
csn	12	Collective sequence number. Starts at 0; increments by one every operation.	
data_type	8	Datatype. Set to 0x00 if the collective operation does not perform reduction.	Table 3-28
reduce_op	8	Reduction operation. Set to 0x00 if the collective operation does not perform reduction.	Table 2-50

The CSN is incremented by broadcast collective operations even if no message is received, ensuring global synchronization. Non-root FEPs in a broadcast collective operation increment the expected CSN without receiving a message, ensuring subsequent CSNs align. The CSN is incremented by all ranks for each new collective operation.

If data arrives at the receiving FEP and a buffer is not available, the FEP generates a response indicating the event (see Section 3.4.5.4).

Because deferrable send is used, packets are retransmitted with new **pds.psn** values when delivered to SES and acknowledged. A NACK is not generated when a message is deferred; the defer is communicated via the SES response in an ACK.

Sending the RTR message optimizes latency as compared to waiting for a pre-configured time. If the buffer is posted more quickly than the 'buffer-not-ready' timer, the RTR message will trigger retransmission faster than waiting for the configured time period. The timer becomes a backup to the RTR message, as not all FEPs support generation of UET_DEFERRABLE_RTR.

SEPs receiving a message with an unknown **ses.inc_info.ver** MUST ignore the rest of **ses.inc_info**, reject the message and return RC_UNSUPPORTED_OP (see Table 3-25).

2.4.11.2 Packet Delivery and Congestion Management

INC uses ROD PDCs, which provide in-order delivery of requests and responses. Reserved PDCs are used to simplify the SEP. Each INC communicator is allocated an individual PDC.

INC PDCs may optionally use UET transport flow control (TFC–See Section 3.6.15), allowing the receiving SEP or FEP to constrain in-flight data on a PDC based on buffer availability.

No special PDS or CMS features are required to support INC. UET profiles supporting INC MUST support reserved PDCs.

2.4.11.3 Transport Security

No special TSS features are required to support INC.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

2.4.12 INC Data Plane Overview

This section specifies the INC data plane protocol. The IM controls the initial creation phase of the INC communicator. Once the INC communicator is created, collective operations may be issued repeatedly without additional setup. The IM also controls the destruction of the INC communicator. Refer to Section 2.4.8 for an overview of the command flows.

2.4.12.1 Data Plane Sequences

Once the communicator is active, with all PDCs established, collective operations can be performed. Figure 2-25 shows an example of executing a CCL allreduce operation. Each packet carries an integer number of elements and sufficient information for the SEP to perform the operation autonomously. The state required at the SEP is primarily the INC communicator membership and root FEP for reduce operations.

TFC may be used if the switch has limited buffering and needs to limit the amount of data each member FEP sends. In some deployments, CBFC may be adequate and TFC is not necessary.

The initialization phase is not shown in Figure 2-25.

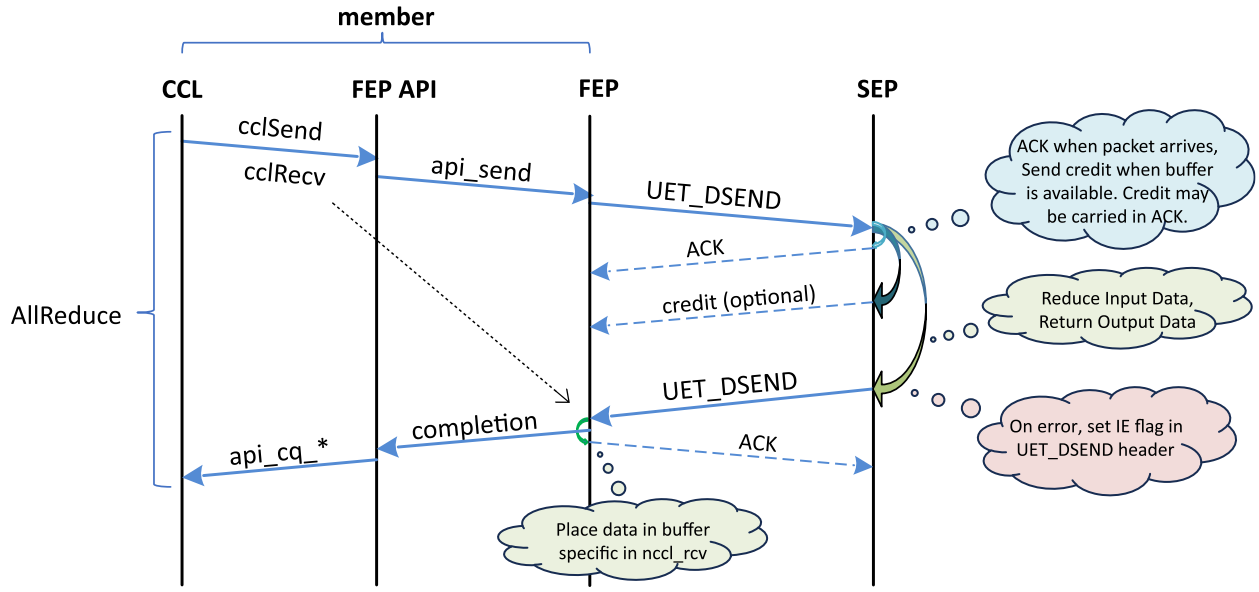


Figure 2-25 - Allreduce Sequence

In an allreduce operation, every member FEP receives output data. Other collective operations, such as reduce, may produce output data only for a designated root FEP. In those cases, the acknowledgement from the SEP that all input data has been received indicates completion of the collective operation.

Figure 2-26 shows the flow when a member FEP receive buffer is not available. This condition causes all traffic on the INC communicator to pause.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

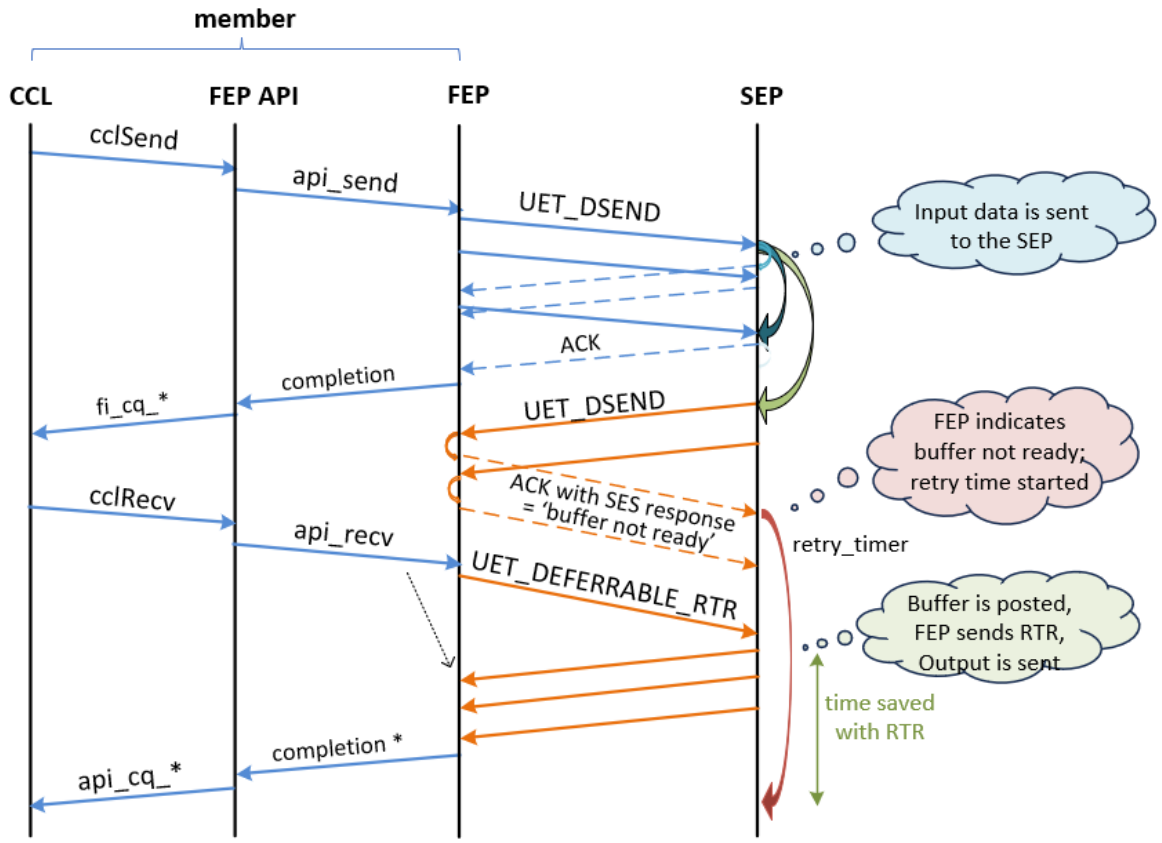


Figure 2-26 - Deferrable RTR Sequence

Table 2-53 describes how ranks participate in the data plane of each collective. When UET_DEFERRABLE_SEND packets carrying data are expected, the cell reads “Data”. When an empty UET_DEFERRABLE_SEND is expected, the cell reads "Empty”. When no UET_DEFERRABLE_SEND is to be performed, the cell reads “Nothing”.

Table 2-53 - Dataplane Participation

Collective	Input, root	Input, non-root	Output, root	Output, non-root
UET_INC_REDUCE	Data	Data	Data	Empty
UET_INC_ALLREDUCE	N/A	Data	N/A	Data
UET_INC_BCAST	Data	Nothing	Empty, if fo = 0 Data, if fo = 1	Data
UET_INC_BARRIER	N/A	Empty	N/A	Empty
UET_INC_SYNCBCAST	Data	Empty	Empty, if fo = 0 Data, if fo = 1	Data
UET_INC_EXSCAN	N/A	Data	N/A	Data

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

2.4.13 Error Handling

INC errors fall into the following categories:

- Transmission error (e.g., unrecoverable packet loss or link loss)
- Format error (e.g., unexpected CSN, datatype mismatch, unsupported data type, etc.)
- Member failure (e.g., timeout)
- SEP failure (e.g., timeout)
- IM failure (e.g., timeout)

In most cases, transmission errors are avoided using a lossless transport alongside FEC and LLR. Errors that are not recovered by FEC and LLR are recovered by PDS retransmission.

Link loss can be avoided using redundant links or other mechanisms, which are out of scope for this specification.

Uncorrectable errors encountered by the ranks or switches are reported to the IM. In such a case, the IM destroys the INC communicator and notifies all ranks and switches involved.

2.4.14 INC Manager

The INC manager (IM) coordinates the setup and teardown of collective offload to INC switches. It uses switch capability information to allocate switch resources for collective offload.

The IM requires network connectivity to the ranks of a job and INC switches. It can operate in the same network as the INC data plane, or in a separate management network.

The IM exposes a gRPC server to the ranks of a job, which act as clients. The ranks are configured with a URL to the IM's gRPC server. The IM, in turn, uses gNMI as a client to communicate with the INC switches.

2.4.15 UET Libfabric Provider and IM

The IM exposes a gRPC server supporting a UE-defined protocol that interacts with a UET libfabric provider on behalf of an application that intends to use INC.

The protocol supports the following operations:

- Capability query
- Communicator creation/destruction
- Communicator status query

2.4.15.1 The gRPC API

The full .proto file specifying the API exposed to the UET provider can be found at the following URL: <https://github.com/ultraethernet/uec-inc/blob/main/proto/uec-uet-inc-im.proto>

It is reproduced below:

```
syntax = "proto3";
```


Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

```
import "google/protobuf/empty.proto";

package uec_uet_inc_im.v1;

service INCMembership {
  // Register a rank with the IM.
  rpc Register(RegistrationRequest) returns (RegistrationReply);
  // De-register a rank from the IM.
  // Also leaves any communicators that the rank is part of.
  rpc Deregister(RegistrationRequest) returns (google.protobuf.Empty);

  // Query the INC capabilities of the fabric.
  // Until a specific communicator is formed, the final capabilities may not
  // be known if the fabric consists of switches with different
  // capabilities.
  rpc GetCapabilities(google.protobuf.Empty) returns (CapabilitiesResponse);

  // Join a communicator.
  // A communicator is established after all ranks have attempted to join.
  // Communicator creation MUST fail if the required network resources and/or
  // capabilities is not provided.
  // This RPC is idempotent and safe to call again in case of a timeout.
  // The rank must be registered before calling this.
  rpc CommunicatorJoin(CommunicatorJoinRequest) returns
(CommunicatorJoinReply);
  // Leave a communicator.
  // This is irrevocable and kills the communicator.
  // The rank must be registered before calling this.
  rpc CommunicatorLeave(CommunicatorLeaveRequest) returns
(CommunicatorLeaveReply);

  // Fetch the list of failed communicators.
  // The rank must be registered before calling this.
  rpc CommunicatorStatus(AddressingInfo) returns (CommunicatorStatusReply);
  // Fetch the list of failed communicators and receive streaming updates as
  // communicators fail.
  // Leaving the stream open is the suggested way of generating heartbeats.
  // The rank must be registered before calling this.
  rpc StreamingCommunicatorStatus(AddressingInfo)
    returns (stream CommunicatorStatusReply);
}

// Local addressing information for one entity participating in INC
message AddressingInfo {
  uint32 job_id = 1;
  //The IP address of the FEP. Can be either v4 or v6.
  bytes ip = 2;
  uint32 pid_on_fep = 3;
  uint32 resource_index = 4;
  uint32 initiator_id = 5;
}

enum Collective {
  BARRIER = 0;
  BCAST = 1;
  SYNCBCAST = 2;
```


Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

```
  ALLREDUCE = 3;
  REDUCE = 4;
  EXSCAN = 5;
}

enum DataType {
  INT8 = 0;
  UINT8 = 1;
  INT16 = 2;
  UINT16 = 3;
  INT32 = 4;
  UINT32 = 5;
  INT64 = 6;
  UINT64 = 7;
  INT128 = 8;
  UINT128 = 9;
  FLOAT = 10;
  DOUBLE = 11;
  FLOAT_COMPLEX = 12;
  DOUBLE_COMPLEX = 13;
  LONG_DOUBLE = 14;
  LONG_DOUBLE_COMPLEX = 15;
  BF16 = 16;
  FP16 = 17;
  FPE4M3 = 18;
  FPE5M2 = 19;
}

enum ReductionOperator {
  MIN = 0;
  MAX = 1;
  SUM = 2;
  SUM_SCALE = 3;
  PROD = 4;
  LOR = 5;
  LAND = 6;
  BOR = 7;
  BAND = 8;
  LXOR = 9;
  BXOR = 10;
  MINLOC = 11;
  MAXLOC = 12;
}

message Operation {
  DataType data_type = 1;
  ReductionOperator operator = 2;
}

enum Ordering {
  NONE = 0;
  // For the same communicator, the switch will always apply the same order
  // of reduction operations
  CONSISTENT_COMMUNICATOR = 1;
  // For different identical communicators in the same job, the switch will
  // always apply the same order of reduction operations
  CONSISTENT_JOB = 2;
```


Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

```
  // For different jobs with identical node allocations and different
  // identical communicators in those jobs, the switch will always apply the
  // same order of reduction operations
  CONSISTENT_GLOBAL = 3;
  NATURAL = 4;
}

enum RoundingMode {
  // Round to nearest, ties to even
  NEAREST_TIE_EVEN = 0;
  // Round to nearest, ties away from zero
  NEAREST_TIE_AWAY = 1;
  // Round toward 0
  TRUNCATION = 2;
  // Round toward plus infinity
  CEILING = 3;
  // Round toward minus infinity
  FLOOR = 4;
}

enum ProtocolType {
  DEFERRABLE_SEND = 0;
}

// Indicates what INC functionality can be provided by the fabric
message Capabilities {
  repeated Collective collectives = 1;
  // Operations
  repeated Operation operations = 2;
  optional Ordering ordering = 3;
  repeated ProtocolType protocol_types = 4;
  // The maximum payload MTU
  optional uint32 payload_mtu = 5;
  // The bandwidth for collectives
  optional uint32 bandwidth = 6;
  repeated RoundingMode rounding_modes = 7;
}

message FlowControlInfo {
  // Cell size
  // Initial credit is equal to the cell size if non-zero, or 1, otherwise.
  uint32 cell_size = 1;
  // Packet overhead
  uint32 packet_overhead = 2;
}

message RegistrationRequest {
  // The rank's addressing information.
  AddressingInfo rank_addr = 1;
  // The payload MTU supported by the rank
  uint32 payload_mtu = 2;
  optional FlowControlInfo flow_control_info = 3;
}

message RegistrationReply {
}
```


Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

```
message CapabilitiesResponse {
  // The INC capabilities that are available to any potential communicator
  Capabilities guaranteed_capabilities = 1;
  // The best-case capabilities that could be available to an arbitrary
  // communicator
  // This is a superset of guaranteed_capabilities.
  Capabilities max_capabilities = 2;
}

message CommunicatorJoinRequest {
  // The rank's addressing information.
  AddressingInfo rank_addr = 1;
  // A 0-indexed integer indicating the rank's position in a communicator
  uint32 rank_id = 2;
  // A 16-byte Unique ID (UID)
  // Also used to refer to the communicator for the remainder of the job's
  // lifetime.
  // All ranks must supply the same value.
  // The UID must not be reused for the duration of the job.
  // This field was sized such that it can house a *CCL Unique ID.
  bytes unique_id = 3;
  // The number of ranks in the communicator
  uint32 rank_count = 4;
  // The capabilities required for communicator operation.
  // Communicator creation MUST fail if these can not be provided.
  optional Capabilities required_capabilities = 5;
  // A hint for the IM indicating which capabilities are desired for
  // communicator operation
  // The IM may or may not honor this request.
  optional Capabilities desired_capabilities = 6;
  // The scale factor by which sum-and-scale data is divided
  // Must be non-0.
  uint32 scale_factor = 7;
}

message CommunicatorJoinReply {
  // The peer's addressing information
  AddressingInfo peer_addr = 1;
  // The communicator ID
  uint32 communicator_id = 2;
  // The maximum number of active collectives
  uint32 in_flight_ops = 3;
  // The INC capabilities available to the communicator
  // Contains exactly one protocol type.
  Capabilities capabilities = 4;
  // The peer's flow control info
  optional FlowControlInfo flow_control_info = 5;
}

enum LeaveReason {
  NO_ERROR = 0;
  GENERIC_ERROR = 1;
}

message CommunicatorLeaveRequest {
  // The communicator's Unique ID
  bytes unique_id = 1;
```


Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

```
  // The rank's addressing information
  AddressingInfo rank_info = 2;
  // The reason for leaving
  LeaveReason reason = 3;
  // A vendor-specific description of the reason for leaving the communicator
  string message = 4;
}

message CommunicatorLeaveReply {
}

message CommunicatorStatusReply {
  // The list Unique IDs of failed communicators that the rank has not
  // explicitly left
  repeated bytes failed_communicator_uids = 1;
}
```

2.4.15.2 Semantics of the *Capabilities* Fields for the *CommunicatorJoin* RPC

During the *CommunicatorJoin* RPC, the *Capabilities* message appears in the *CommunicatorJoinRequest* as both *required_capabilities* and *desired_capabilities*, and in the *CommunicatorJoinReply* as *capabilities*. The fields in the *Capabilities* message have the following semantics based on where they are located:

- *collectives*, *operations*, which contain items that are non-exclusive:
  - Required: the list of required items
  - Desired: the list of desired items, in addition to the required items
  - Reply: list of supported items
- *protocol_types*, *rounding_modes*, which contain items that are exclusive on a per-group basis:
  - Required: the list of items in order of preference, any of which are considered sufficient
  - Desired: the list of items in order of preference, if none are required
  - Reply: the item chosen by the IM for the communicator
- *ordering*, *payload_mtu*, *bandwidth*, which contain the minimum values:
  - Required: the minimum required value, if any
  - Desired: the minimum desired value, if any
  - Reply: the value chosen by the IM for the communicator

2.4.16 IM Management

An IM management interface is required for the job launcher to configure the IM at job launch and termination. A control protocol is also required to establish INC communicators and configure components for collective operations. Vendors can implement a control protocol of their choosing; this document provides an example gRPC API.

2.4.16.1 The gRPC API [Informational]

The full .proto file specifying the suggested API can be found at the following URL:
<https://github.com/ultraethernet/uec-inc/blob/main/proto/uec-uet-inc-mgmt.proto>

It is reproduced below:

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

```
syntax = "proto3";

package uec_uet_inc_im.v1;

service INCManagement {
  // Configures the IM for operation within a new job.
  rpc CreateJob(JobCreationRequest) returns (JobCreationReply);
  // Destroys a job.
  // All gRPC streams to the ranks are also closed.
  rpc DestroyJob(JobDestructionRequest) returns (JobDestructionReply);
}

message JobCreationRequest {
  // The Job ID.
  uint32 job_id = 1;
  // A list of X.509 certificates used by the ranks for authentication.
  // If empty, no authentication is required. Otherwise, ranks must
  // authenticate.
  repeated bytes certificates = 2;
}

message JobCreationReply {
}

message JobDestructionRequest {
  // The Job ID.
  uint32 job_id = 1;
}

message JobDestructionReply {
}
```

2.4.17 IM and INC Switches

The IM communicates with INC switches to perform the following operations:

- Query INC capabilities.
- Allocate and free switch resources for an INC communicator.
- Query the configuration that is in place.
- Monitor the status of an INC communicator.
- Collect statistics for troubleshooting.

For this purpose, each switch exposes a gNMI server to the IM. The mechanisms by which the IM discovers the INC switches and establishes connectivity to them are beyond the scope of this document.

2.4.17.1 YANG Schema

The YANG model of the data communicated between the IM and INC switches is made up of two modules:

- *uec-uet-inc*, and
- *uec-uet-inc-types*.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

The modules can be found at the following URL: <https://github.com/ultraethernet/uec-inc/tree/main/yang>.

The schema of the main module, *uec-uet-inc*, is depicted below:

```
module: uec-uet-inc
  +--rw communicators
  |  +--rw communicator* [job-id communicator-id]
  |     +--rw job-id              -> ../config/job-id
  |     +--rw communicator-id     -> ../config/communicator-id
  |     +--rw config
  |     |  +--rw communicator-id?   inc:communicator-id
  |     |  +--rw payload-mtu?       inc:payload-mtu
  |     |  +--rw admin-status?      enumeration
  |     |  +--rw dscp-low?          inet:dscp
  |     |  +--rw dscp-high?         inet:dscp
  |     |  +--rw job-id?            inc:job-id
  |     |  +--rw scale-factor?      uint32
  |     |  +--rw protocol-type?     identityref
  |     |  +--rw bandwidth?         uint32
  |     |  +--rw ordering-mode?     inc:ordering-mode
  |     |  +--rw rounding-mode?     identityref
  |     +--ro state
  |     |  +--ro (same as ../config/)
  |     |  +--ro oper-status?           enumeration
  |     |  +--ro failure-description?   string
  |     |  +--ro counters
  |     +--rw peers
  |        +--rw peer* [id]
  |           +--rw id        -> ../config/id
  |           +--rw config
  |           |  +--rw id?                      uint32
  |           |  +--rw remote-fabric-address?   inet:ip-address
  |           |  +--rw role?                    identityref
  |           |  +--rw flow-control-info
  |           |  |  +--rw enabled?           boolean
  |           |  |  +--rw cell-size?         uint16
  |           |  |  +--rw packet-overhead?   uint16
  |           |  +--rw remote-pid-on-fep?       inc:pid-on-fep
  |           |  +--rw remote-resource-index?   inc:resource-index
  |           |  +--rw interface?
  |           |  |     -> /oc-if:interfaces/interface/name
  |           |  +--rw subinterface?
  |           |  |     -> /oc-if:interfaces/
  |           |  |        interface[oc-if:name=current()/../interface]/
```


Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

```
|           |  |        subinterfaces/subinterface/index
|           |  +--rw local-fabric-address?    inet:ip-address
|           |  +--rw local-pid-on-fep?        inc:pid-on-fep
|           |  +--rw local-resource-index?    inc:resource-index
|           |  +--rw assoc-index?             uint16
|           +--ro state
|              +--ro (same as ../config/)
|              +--ro counters
+--ro capabilities
   +--ro max-groups?                        uint32
   +--ro max-rank-bandwidth?                uint32
   +--ro supported-operations* [data-type reduction-operator]
   |  +--ro data-type             identityref
   |  +--ro reduction-operator    identityref
   +--ro max-payload-mtu?                   inc:payload-mtu
   +--ro max-interfaces?                    uint32
   +--ro max-interfaces-per-communicator?   uint32
   +--ro max-peers-per-interface?           uint32
   +--ro max-peers?                         uint32
   +--ro max-peers-per-communicator?        uint32
   +--ro max-communicators-per-interface?   uint32
   +--ro available-protocol-types*          identityref
   +--ro max-outstanding-collectives?       uint32
   +--ro supported-ordering-modes*          inc:ordering-mode
   +--ro supported-rounding-modes*          identityref
   +--ro supported-collectives*             identityref
   +--ro flow-control-info
      +--ro enabled?           boolean
      +--ro cell-size?         uint16
      +--ro packet-overhead?   uint16
```

2.4.17.2 Determining a Switch's Eligibility

The *capabilities* node holds information regarding switch capabilities. The IM uses these capabilities to determine if an INC switch is eligible to be used for INC offload for a particular communicator. The IM reads the switch capabilities and performs the following:

- The IM ensures that at least one of the protocol types included in the *available-protocol-types* node is supported by all the other switches used by the communicator.
- The IM verifies that the set of capabilities provided by the switch matches the set of capabilities required by the ranks. The capabilities are listed in Table 2-54.
- The IM verifies that the resources listed in Table 2-55 are available on the switch.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Table 2-54 - Fabric Capabilities

gRPC field (*Capabilities::…*)	YANG leaf (*capabilities/…*)	Description
bandwidth	*max-rank-bandwidth*	The maximum aggregate bandwidth of input or output data streams.
operations	*supported-operations*	The list of {data type, reduction operator} tuples.
collectives	*supported-collectives*	The supported collectives.
payload_mtu	*max-payload-mtu*	The data Payload MTU.
ordering	*supported-ordering-modes*	A single-switch setup provides corresponding ordering to the communicator as a whole.
protocol_types	*available-protocol-types*	The list of available data plane protocol types.
rounding_modes	*supported-rounding-modes*	The supported rounding modes.

Table 2-55 - Switch INC Resources

YANG leaf (*capabilities/…*)	Description
max-communicators	The number of communicators that can concurrently be active on the switch.
max-interfaces	The maximum number of interfaces that can be employed across all communicators.
max-interfaces-per-communicator	The maximum number of interfaces that can be used in the operation of a communicator.
max-peers-per-interface	The maximum number of peers with which a local interface can peer.
max-peers	The maximum number of peers that can take part in collective operations at the same time.
max-peers-per-communicator	The maximum number of peers that can be part of a communicator.
max-communicators-per-interface	The maximum number of communicators that can operate on an interface concurrently.

2.4.17.3 Configuration of an INC Communicator

This section describes the process of configuring an INC switch for an INC communicator. To perform this process the following information is produced by or provided to the IM:

- The CID.
- The data plane protocol[2].
- The set of involved SEPs and their fabric addresses.
- The peering relationships of the SEPs.
- The data MTU size to be used by the communicator.
- The scalar integer by which the results are divided when performing sum and scale operations.
- The TC_low and TC_high traffic classes used by the UET protocol.
- The bandwidth at which collectives are to be run.
- The operator ordering mode for the communicator.

[2] This specification defines a single data plane protocol (see section 2.4.11); however, other vendor-specific protocols are possible.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

- The floating-point rounding mode for the communicator.

The series of steps makes no assumption about the atomicity of how gRPC requests are handled. An IM implementation performs the following steps.

The INC communicator node is created as follows:

- The *admin-status* is initialized to *DOWN*.
- The *communicator-id* is set to the CID.
- The *job-id* is set to the INC communicator's JobID.
- The *payload-mtu* is set to the Payload MTU to be used for the INC communicator.
- *dscp-low* and *dscp-high* are set to the codepoints mapped to the low and high traffic classes to be used for INC traffic.
- *scale-factor* is set to the number by which data for sum and scale is divided.
- *protocol-type* is set to the protocol type selected for the communicator.
- *ordering-mode* is set to the chosen operator ordering mode for that communicator.
- *rounding-mode* is set to the chosen rounding mode for that communicator.
- The *bandwidth* node is set to the rank bandwidth allocated by the IM for the INC communicator. The *interface* and *subinterface* nodes reference the chosen IETF interface and subinterface, respectively.
- The local addressing information, namely the values for *local-pid-on-fep* and *local-resource-index*, are chosen arbitrarily by the IM.
- The *peers* list node is populated with the interfaces participating in the communicator.

A *peer* node is created as follows:

- The *id* is set to a value that is unique within the list.
- The *remote-fabric-address* is set to the peer's fabric address.
- The *role* is set to *LEAF*.
- The *flow-control-info* node includes the cell-size and packet-overhead nodes. This information was provided as part of the rank's *RegistrationRequest::flow_control_info*.
- If natural operator ordering mode is used, the *assoc-index* node indicates the position in the natural order taken up by the peer's input data.
- The remote addressing information, namely the values for *remote-pid-on-fep* and *remote-index*, are extracted from the rank's *RegistrationRequest::rank_addr*, namely *pid_on_fep* and *index*, respectively.

The IM MUST set the INC communicator's *admin-status* to *UP* only after all of the data structures have been initialized.

Once the *admin-status* is set to *UP*, a switch that cannot or will not comply with the initialized data structures MUST set the INC communicator's *oper-status* node to *FAILED*.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

The switch otherwise waits for the necessary INC resources to become ready for use and sets *oper-status* to *UP*.

The IM considers the switch configuration to be finished when the communicator's *oper-status* is set to *UP*. It is RECOMMENDED that the IM set a gNMI subscription to the *oper-status* node at this point.

2.4.17.4 Monitoring INC Communicator Status

Switches report communicator failures by setting the *oper-status* node to *FAILED*. The IM monitors the *oper-status* node and reports the INC communicator as failed if any switch reports a failure. It is RECOMMENDED that the IM set a gNMI subscription to the *oper-status* node.

When an INC communicator's *oper-status* node changes to *FAILED*, the *failure-description* leaf SHOULD contain a human-readable description of the failure.

2.4.17.5 INC Communicator Destruction

The IM can destroy an INC communicator by deleting the communicator node.

2.4.18 References

[1] R. L. Graham, D. Bureddy, P. Lui, H. Rosenstock, G. Shainer, G. Bloch, D. Goldenerg, M. Dubman, S. Kotchubievsky and V. Koushnir, "Scalable hierarchical aggregation protocol (SHArP): A hardware architecture for efficient data reduction," in First International Workshop on Communication Optimizations in HPC (COMHPC), 2016.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

3 UE Transport Layer

The Ultra Ethernet Transport (UET) layer is designed to handle the most challenging application scale, deliver packets reliably and securely, manage and avoid congestion within the network, and react to contention at the endpoints. Its goals are minimal tail latency and highest network utilization. At the same time, UET is designed to enable simple hardware and software implementations—such as what might be required for accelerator-integrated endpoints. UET can be programmed through the OFI libfabric standard interface. It sets out to address the shortcomings of RoCEv2 [1], specifically its semantics, transport layer, wire operations, implementation complexities, and scale limits.

UET leverages semantics and reliability techniques from HPC to enable extreme scale while providing advanced congestion management that employs the breadth of techniques and telemetry rising from hyperscale datacenters for traditional Ethernet environments. The transport supports up to millions of endpoints (NIC ports) with the ability to address up to billions of processes. Scalability is built into every aspect of its design—from the state required in the semantic layer down to the way encryption keys are managed and the way reliability and ordering are achieved.

To enable simplified implementations, subsets of the capabilities are defined in specific profiles that support specialized use cases in AI domains. The HPC profile offers the most comprehensive functionality. Some of it is optional to implement; two of the AI profiles define subsets of HPC. The AI Full and AI Base profiles enable full or partial offload of *CCL-like messaging functions to hardware. The AI Full profile offers deferrable sends, which save up to 1 RTT compared to the AI Base profile in the case when the receive buffer is not ready. The AI Full profile also provides exact match offload to identify a specific receive buffer. The AI Base profile requires a prior message exchange to communicate a desired target buffer, which enables slimmer AI Base profile implementations. The Unified Local profile extends UET to support local scale-up networks. Optimized for wire efficiency, small transactions, and compact implementation, the Unified Local profile supports write/read/atomic operations.

This overview section has been used by the working group to define the goals and the overall architecture of UET as well as to design the specifics of the sublayers. It contains an overview of the architecture and each sublayer as well as some historic context to ease the understanding of readers. It does not contain normative text outside of Section 3.3.

3.1 UET Scope, Scale, and Reach

This section outlines the original design guidelines for UET to limit its scope and enable optimized design choices. This background information helps readers understand decisions and architectural choices.

UET focuses on the backend scale-out and local scale-up networks. UET opportunistically considers support for frontend networks while not preventing applications in the scale-up network. UET does not define details of the hardware interface to the endpoints.

UET is designed to operate within one administrative domain. This means that, where necessary, a single provisioning system is presumed for the assignment of identity and policy.

The workload and use cases are focused on generic AI, HPC, and storage traffic.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Single applications are expected to span the entire system, single-node jobs filling up the system, and the full spectrum in between. The objective of UET is to serve the breadth of all those use cases with a single transport.

3.1.1 Virtualization

Virtualization hides the details of the physical infrastructure from tenants and tenants from each other. Although there are “bare metal” instances, host virtualization is common, and network virtualization is almost universal in cloud deployments. This level of virtualization presents a stark contrast between traditional HPC and clouds: Traditional HPC deployments typically focus on performance first with a more trusted user base.

Most virtualization technology sits in direct tension with performance objectives. For example, traditional network virtualization requires O(N) state per endpoint, which UET tries to avoid. Additionally, using network tunnel techniques introduces substantial packet overheads. In many environments, however, the added security and isolation provided by virtualization is a fundamental requirement. UET assumes the following with respect to virtualization:

1) Network tunnel techniques that are used today work in the context of UET, because UET uses IP packets. UET packets can be encapsulated within VXLAN and similar tunnels; similarly, UET packets can carry packets of tunneled protocols. Detailed implementations of virtualization techniques are currently beyond the scope of UET. If UET packets are encapsulated in other protocols, care must be taken to support UET’s signaling.
2) Host virtualization can be accomplished using traditional techniques—such as SR-IOV, SIOV, unique fabric addresses (IP) per tenant, or others.
3) Deployments focused on the largest scales leverage techniques that simplify the tunnel logic (e.g., structured addressing techniques).
4) Deployments that need to focus on network packet efficiency may choose to not use encapsulation.

Nonetheless, some basic support is needed to support tunneling in a network using UET. For example, systems carrying UET packets inside a tunnel would provide congestion information, such as explicit congestion notification (ECN), to the encapsulated packets. Solutions such as IETF RFC 6040 [2] are available to provide this functionality. Furthermore, such systems would copy the entropy value of encapsulated UET packets to the encapsulating packet to ensure ordering and load balancing. UET does not currently define details for tunneling and leaves it up to the system administrator to ensure correct encapsulation and decapsulation.

3.2 UET Layers, Components, and Capabilities

The Ultra Ethernet transport layer is built on four largely independent sublayers: semantics, packet delivery, congestion management, and security. See Figure 3-1. Each piece is designed to be separable from the others; however, implementations may choose to build and integrate the functionality in any way.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

The semantics sublayer (SES) defines addressing, authorization, message types, protocols, and semantic header formats between endpoints. SES works at the level of transactions, such as messages or remote memory accesses (RMA), and breaks those into multiple packets to be transmitted by the packet delivery sublayer. The packet delivery sublayer (PDS) transports a stream of packets reliably to the destination FEP and passes them to the target's SES layer for processing. It generates and interprets ACK and NACK packets to ensure reliable transmission and uses ephemeral state to track outstanding packets in the network. The congestion management sublayer (CMS) ensures that the packets are transmitted at highest rate while minimizing network congestion. The transport security sublayer (TSS) defines scalable encryption and authentication mechanisms for peer-to-peer as well as client-server communications.

Figure 3-1 shows an overview of the overall UE stack and the UET sublayer structure. The key architectural features of each sublayer are summarized, including how they are composed to form UET. Full details of the specification are defined in the respective sections of this document.

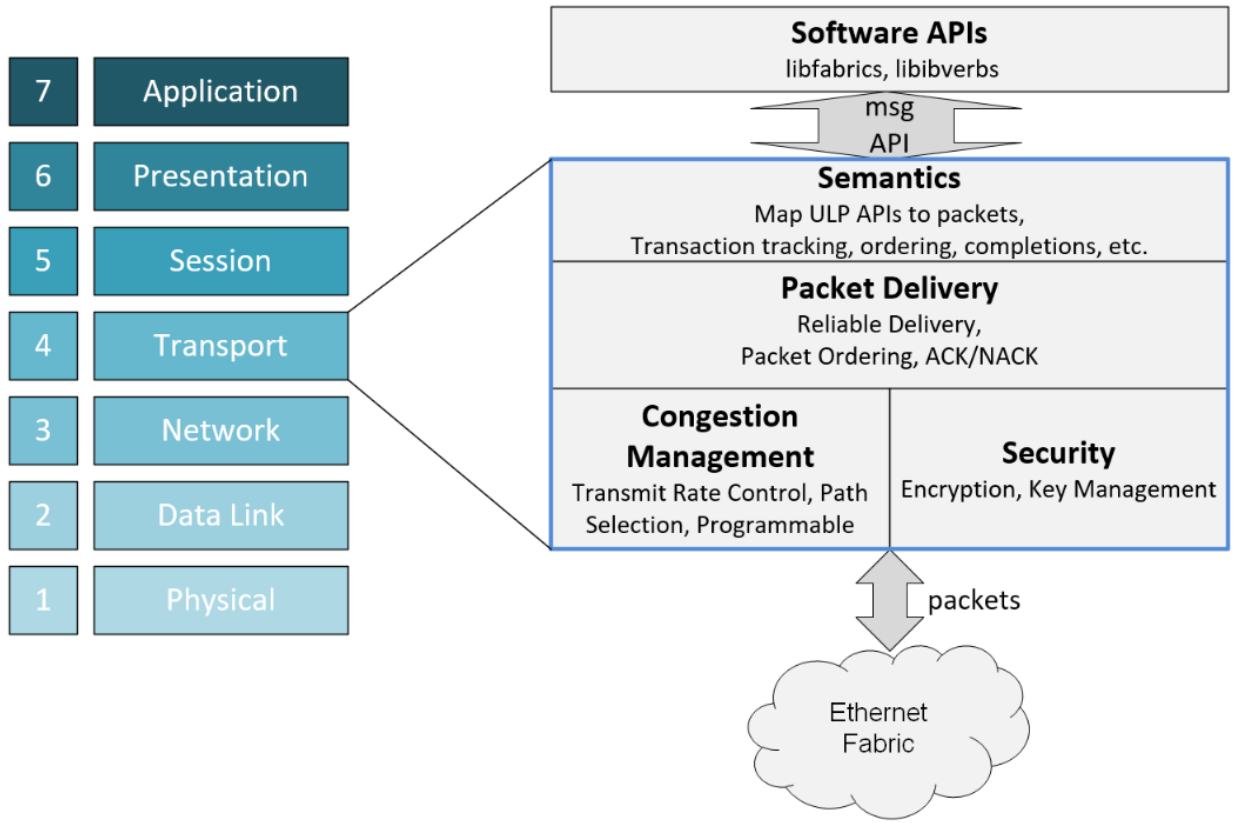


Figure 3-1 - Overview of UET

Each sublayer uses specific header fields in the packet. The UET payload is the data portion of a UET packet beginning immediately after the SES header fields and of length specified by the length field of the SES header. The length field of the SES header does not include the UET trailer if present. The total

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

size of a UET packet is the sum of UET header, UET payload, and UET trailer. The UET packet structure is shown in Figure 3-2.

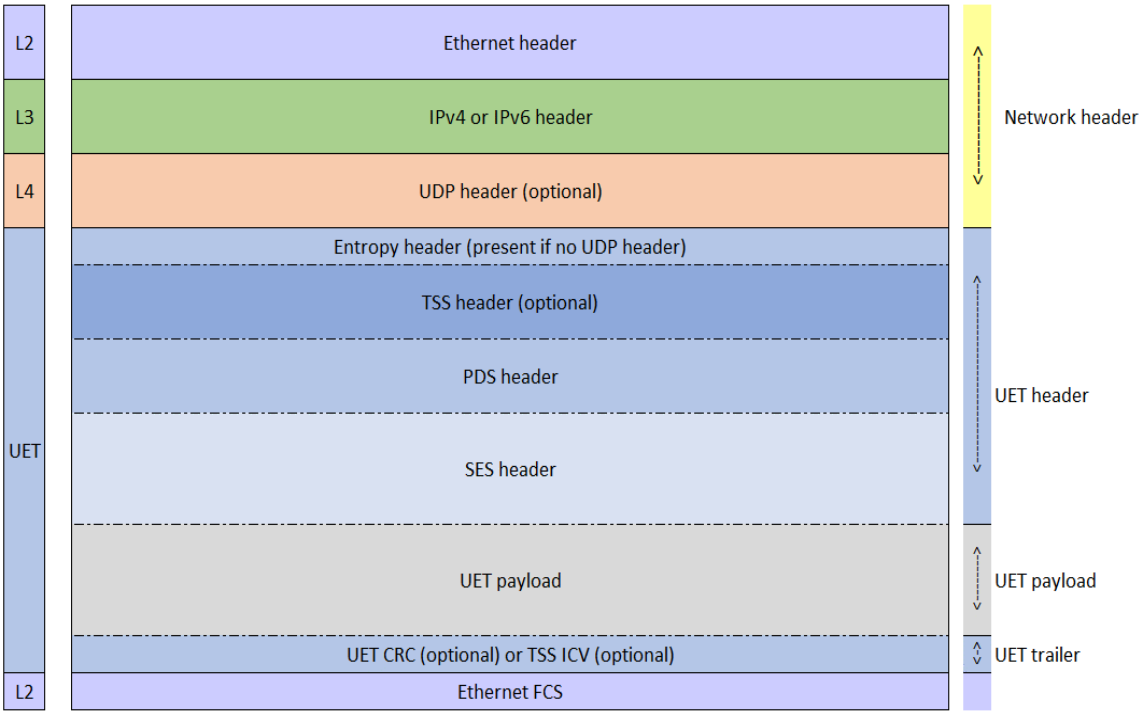


Figure 3-2 - UET Packet Structure

3.2.1 Semantic Sublayer (SES)

The objective of UET's semantic sublayer is to provide high-performance and highly scalable messaging to enable specialized AI and fully featured HPC deployments. The SES bridges between the user-facing libfabric API and the PDS by mapping libfabric API calls to a set of UET communication operations, such as tagged and untagged send/receive, RMA read/write, and atomics. Utilizing libfabric inherits benefits from a wider ecosystem in which user-facing libraries using libfabric already exist and are portable even beyond UE devices. SES is optimized for common end-user-facing APIs—from *CCL to MPI to OpenSHMEM. It provides optional message ordering and various optional initiator or target completion notifications (e.g., global observability). SES supports libfabric's connectionless API to allow the underlying hardware to support a large number of endpoints.

SES translates libfabric communication calls into messages that transmit data from or to buffers at the initiator or target process, respectively. It transmits message transactions by utilizing PDS functionality, packetizing messages and mapping packets into messages at the destination. PDS interactions include the packets of the message but also control packets for reliability and status exchanges.

SES defines two protocols for (large) message transmission: rendezvous and deferrable send. The rendezvous protocol is used for messages that exceed the available temporary eager buffer limit at the

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

target. The target waits (“rendezvous”) until the receiving process has posted a matching receive and then triggers a read from the source. The deferrable send protocol simply sends messages of any size, and a target that cannot receive it yet sends a message to stop the sender. Later, once the buffer is posted, the target sends a resume message to continue the transaction. The main difference is that with the rendezvous protocol, a sender decides before sending the message whether it is sending it using an eager or a rendezvous transaction, while the deferrable send behaves the same at the sender, and the receiver reacts dynamically and defers the send if the receive has not been posted and the message cannot be buffered.

SES supports two fundamental addressing types: relative addressing for peer-to-peer communication in large compute jobs and absolute addressing for client-server connections. Once a message endpoint is identified, a buffer is selected with optional matching criteria (based on a packet-carried initiator ID, e.g., an MPI rank and additional match bits).

OFI libfabric is an open-source ecosystem with wide adoption—from ISV-certified MPI implementations to ML communication libraries like *CCL. UEC does not specify how user-facing libraries use libfabric, but many mappings are natural. For example, *CCL semantics map naturally to *fi_tagged(send/recv)* with exact matching semantics, MPI-1 semantics map to *fi_tagged(send/recv)*, and MPI-3 RMA as well as OpenSHMEM semantics map to *fi_rma(read/write)* in libfabric. Collective operations can map to *fi_collective()* for acceleration. Of course, it is always possible to implement any messaging semantics (e.g., *CCL) over any lower-layer semantics (e.g., *fi_rma()*), but this may add additional software overhead and prevent full offload.

3.2.2 Packet Delivery Sublayer (PDS)

PDS implements reliable packet transmission for the SES. It receives packets from the SES at the initiator and delivers packets to the SES at the target, which the SES resolves to messages that may update target memory. It also delivers SES return codes and return data (read and atomics) back to the initiator. The PDS offers various ordering modes for packet delivery. All packets of a UET message, excluding the last packet, are of size MTU.

Achieving a scalable reliability solution requires that the state retained in the NIC be based on the number of simultaneously active (concurrent) communications—not the total number of endpoints in the application. The reliability layer is designed to cover three key requirements:

1. Extreme scalability
2. Ordered delivery of some packets
3. Unordered delivery of some packets—particularly of bulk payload

Those requirements enable efficient support for packet spraying. PDS defines three packet types: request, response (aka ACK/NACK), and control packets. PDS request packets flow from the initiator of a message to the target (reads are an exception). PDS responses flow in the opposite direction. Request packets usually carry data; ACKs carry SES codes to complete the message or defer it. As an optimization, PDS responses may also carry small read data directly. Larger reads are handled differently. The SES response code for a read request indicates “read accepted,” and a separate SES

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

response with data is sent using a PDS request from the target to the initiator. Control packets are used for management purposes, e.g., to request the status of a request packet or to probe a path's congestion situation.

In support of the ordering objectives and the use of multiple, concurrent, independent paths, the reliability layer is designed with five packet delivery modes. Those modes support the semantic needs of both HPC and AI, while enabling highly optimized, highly scalable implementations. In some use cases, the same message may utilize more than one packet delivery mode. For example, an MPI implementation might use an ordered mode to ensure header ordering together with an unordered delivery mode for the payload. The five packet delivery modes are:

1. **Reliable, unordered delivery for operations (RUD)**: The RUD packet delivery mode is designed to enable operations that are passed to the semantic sublayer only once but can tolerate reordering in the network (e.g., atomic add operations). This allows all bulk data to be routed unordered across the network. The reliability sublayer detects duplicate packets so that each packet is delivered to the semantic sublayer only once (i.e., operates on host memory only once).
2. **Reliable, ordered delivery (ROD):** The ROD packet delivery mode maintains the order for all packets between two endpoints. It is designed for applications that require message ordering, e.g., MPI's match ordering or OpenSHMEM put-with-signal semantics. As an example, MPI can transfer a header using an ordered delivery protocol and then transfer the body of the message using a different protocol. This provides ordering at the message level and unordered bulk data. The reliability sublayer detects replays and ensures that the operation within each packet interacts with host memory only once.
3. **Reliable, ordered delivery for local networks (RODL):** The RODL packet delivery mode is a variation of the ROD packet delivery mode that is optimized for networks using the Unified Local profile. RODL allows implementations to omit PDS-layer retransmission for efficiency, relying on mechanisms such as LLR for reliability. The header format is optimized for wire efficiency.
4. **Reliable, unordered delivery for idempotent**[3] **operations (RUDI)**: The RUDI packet delivery mode supports special applications where very small messages need to be delivered between an extreme number of endpoints. It takes advantage of the fact that some data—like bulk payload delivery—can be written into memory multiple times up until the final message completion is delivered at the initiator. RUDI packets can be reordered in the network and replayed due to loss, leading them to be delivered more than once to the semantic sublayer and in any order. Because end-user semantics can be complex, this protocol may not be appropriate for all use cases; yet, due to its stateless nature at the receiver, it is the most scalable of the delivery modes because it requires no state at the receiver.
5. **Unreliable, unordered delivery (UUD)**: As with most transport layers, UET provides an unreliable packet delivery mode. Unreliable packets (or datagrams) can be used if guaranteed delivery is not required.

[3] Idempotent operations give the same result with the no additional side effects when performed multiple times.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Both RUDI and UUD traffic are not subject to UET congestion control and therefore it is not recommended for their traffic to share the same traffic class with either ROD or RUD traffic.

Informative Text
UET does not use connections but defines various packet transport modes. UET first distinguishes between reliable and unreliable packet transmission, then between ordered and unordered delivery. The reliable unordered mode differentiates between idempotent (RUDI) and non-idempotent (RUD) operations, where the former can be applied multiple times in the context of the same message transmission (e.g., write or read). Examples for non-idempotent operations are atomic operations and operations that have other side effects at the target (e.g., delivering header data to a completion queue). The ability to apply a packet operation multiple times has implementation benefits. Idempotent packet operations may not change the message transmission or matching state when they are applied multiple times.

UET supports in-order message delivery while allowing the majority of the associated bulk data placement to be out of order. This includes ordered buffer addressing. For example, many AI applications rely on in-order buffer matching semantics in *CCL libraries. This can be implemented with UET using user-level message sequence numbers as matching tags. This way, incoming multi-path RUD messages can be matched at the receiver in the order they were issued at the sender. If wildcard matching is required by the user code (e.g., in MPI), a combination of ROD and RUD can be used to deliver rendezvous messages: The initial part of the message would be delivered in order through ROD, and the remainder could be delivered as part of rendezvous or deferrable send through RUD.

Extreme scalability is also supported by using dynamically created packet delivery contexts (PDCs). This means that reliability state is required only between peers with ongoing communications. PDCs can be created as part of normal packet transmission without incurring additional round trips. Depending on resource availability, PDCs can be kept alive for extended periods of time (up to 2^{31} packets) as well, and the mechanism supports cache-like PDC management.

3.2.3 Congestion Management Sublayer (CMS)

UET's congestion management involves the combination of mechanisms for window-based congestion control and load balancing that is performed at each FEP. UET defines various congestion-control algorithms to enable interoperability between vendors. Specific load balancing techniques and mechanisms to ensure fairness of congestion control in the presence of multiple FEPs are outside of the scope of UET, as they are not required for FEP-FEP interoperability. By necessity, different vendor solutions operating in the same fabric plane are required to co-exist. UET exemplifies various possible algorithms.

CMS reacts to endpoint contention (e.g., incast) and network congestion (e.g., in oversubscribed scenarios). CMS defines two fundamental congestion control algorithms: Network-signal based congestion control (NSCC) that runs primarily at the sender and receiver-controlled congestion control (RCCC) that runs primarily at the receiver. Either can be used in isolation, or deployments can combine

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

both. NSCC is designed as a generic stand-alone algorithm supporting arbitrary deployments and traffic patterns. It relies only on round-trip time and ECN, while RCCC explicitly monitors incoming flows at the receiver. RCCC is expected to work best on fully provisioned (nonblocking) fat trees and can be complemented with NSCC in oversubscribed fat trees. UET also specifies optional packet trimming support for more rapid congestion information provided to both algorithms.

Both algorithms limit the amount of pending data (in unacknowledged packets) in the network. UET defines a congestion control context (CCC) that contains the state required to implement the congestion control algorithm(s). The CCC state controls the window size that limits the number of data packets in the network for the one or more PDCs associated with the CCC. This efficient scheme supports multiple PDCs between the same two FEPs that share the same traffic class (TC). CCCs may be helpful to coordinate multiple PIDonFEP endpoints, for example, or different logical types of traffic sharing the same TC. Such a set of PDCs is called a PDC group and is matched to a single CCC such that all traffic on the same TC between the same pair of FEPs is coordinated. Multiple CCCs may exist between the same FEPs to enable co-existing RUD and ROD flows.

Path load balancing algorithms choose which path to utilize for a specific packet. UET does not mandate any path selection or load balancing algorithm to enable vendor differentiation. It provides examples where path selection is implemented using equal-cost multi-pathing (ECMP) and is controlled through changing the entropy in the packet headers. Those example algorithms aim to reduce hash collisions and congestion and improve performance. The UDP source port is used as the common field in the packet headers to specify entropy. ECMP forwarding behavior of switches is assumed to guarantee (in the absence of failures) that packets with the same entropy use the same path.

Ultra Ethernet offers programmable congestion management (3.6.18) allowing administrators to define their own congestion control and load balancing algorithms in software.

3.2.4 Transport Security Sublayer (TSS)

Security is a first-class citizen in UET and designed in from the start. UET's security solution provides the option to encrypt and authenticate all data payload and most of the transport headers while being designed to enable packet spraying. The security solution provides a scalable solution using a single key across an entire parallel application and no per-peer security state for large parallel jobs. It also provides a scalable mechanism to secure many clients that communicate with a server using key derivation. The PDS is designed in conjunction with the TSS to enable the detection of replays. TSS is robust against known attacks on other low-level transport schemes, such as various exhaustion attacks [3].

3.2.5 Layering Summary

UET can run on top of IP/UDP or experimentally on top of IP directly; implementations may be configured with a protocol number as described in IETF RFC 3962 [4]. When wire efficiency is important, UET can run directly over UFH. Detailed packet header formats can be found in the SES, PDS, CMS, and TSS sections of this specification. Packet formats include an entropy value field in the UDP source port to enable compatibility with existing switches and common ECMP path selection approaches.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Because UET uses a standard IP packet carried over Ethernet, it is implicitly possible for UET to be tunneled using standard technologies such as VXLAN and NVGRE.

Within a UET packet, there is an optional encryption header followed by a reliability header and then a semantic header. The encryption and reliability headers both leverage a next-header encoding so as not to prohibit the UET definition from evolving to allow other traffic—including RoCE or TCP—to be layered within the encryption and reliability mechanisms. While the initial design of UET does not consider such encapsulations, the packet layering is provisioned to enable this as a future option.

3.2.6 Sublayer Interfaces

The sublayers of UET are modularized to specify roles and responsibilities of functionality rather than indicate implementation requirements. Actual implementations are free to choose whatever modularity is appropriate, as long as the external observable behavior of the implementation is compliant with the normative directives in the specification. Interfaces between the UET components and the users of UET, both above and below, are described in the specification to further delineate the roles and responsibilities of the modular functionality.

Figure 3-3 depicts the UET components and component interfaces. The UET libfabric provider is the primary application-facing layer of UE. It is responsible for mapping libfabric API calls to the UET semantic sublayer to support the communication needs of the libfabric endpoint. Send and completion queues are the primary means of interfacing between the UET libfabric provider and the semantics layer.

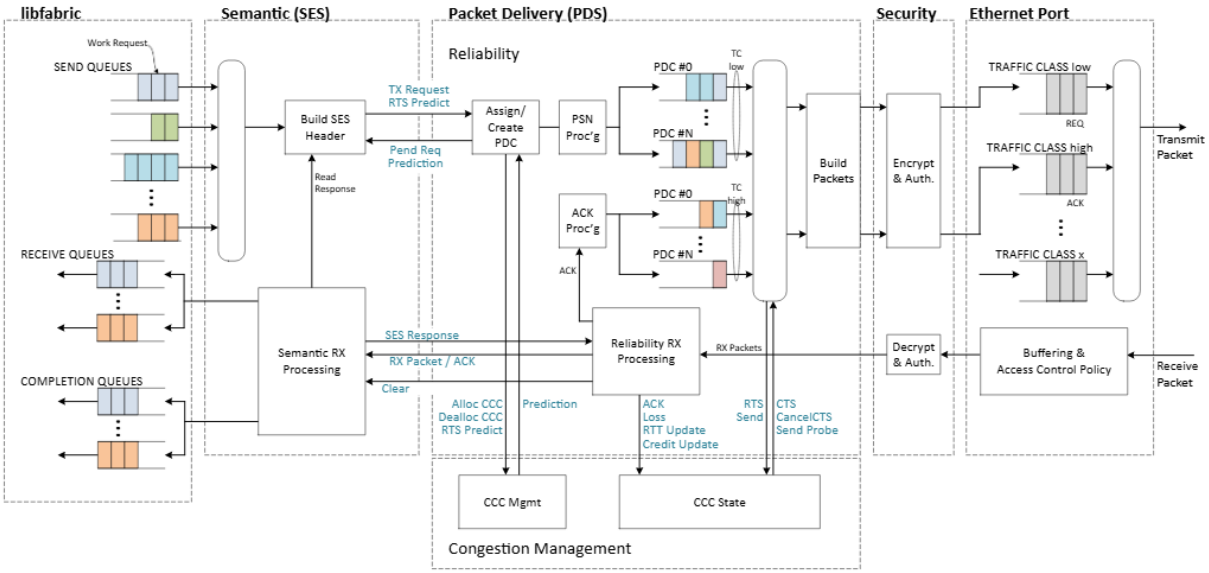


Figure 3-3 - Component Interface Overview

SES connects the UET libfabric provider to the PDS. SES packetizes messages at the source of the data and deposits the received data into memory at the target. In addition, it implements various higher-level protocols, such as rendezvous or deferrable sends, to synchronize initiator and target. SES in turn makes

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

transmission requests to the reliability component of PDS, which (if needed) creates and assigns a packet delivery context (PDC) and a congestion control context (CCC) for those packets. PDS is responsible for delivering packets from sender to receiver. PDS implements reliability and is designed to operate on both lossless and best-effort networks. CMS is optimized to support best-effort networks. It manages the assignment of packets to traffic classes. The PDS interfaces with the congestion management sublayer on each packet to determine whether new packets can be sent into the network, as well as an entropy value to be used. Send completion notifications and congestion signals from received packets and ACKs are passed to the congestion management sublayer to manage transmission windows. The TSS performs encryption and authentication at the sender, and decryption and authentication at the receiver, of each packet. The data is then forwarded to the output ports of the fabric interface.

3.2.7 Error Handling

Each layer may generate errors that are either to be passed to the application through the libfabric bindings (see libfabric mapping Section 2.2.5.6) or to the system administrator through logging or other alert mechanisms. In UET, only the SES layer interfaces to the application directly; other layer errors are either passed through SES to libfabric or logged in a vendor-specific form.

3.3 Profiles and Capabilities [normative]

UET covers various application use cases utilizing different communication libraries and services with differing communication requirements for both HPC and AI. Not all use cases require all semantics, and simplified semantics offer opportunities to specialize and optimize hardware. Thus, UET defines four profiles called HPC, AI Base, AI Full, and Unified Local respectively. The HPC profile supports full-fledged HPC semantics and supports a wide range of applications. The AI Base profile is specialized to support *CCL and unreliable datagram communication focusing on minimal implementation complexity. The AI Full profile is intended to support all AI training and AI inferencing requirements to serve that emerging market most effectively. The Unified Local profile defines an efficient header format for high efficiency over the wire in local scale-up networking contexts. A specific endpoint implementation can support any single profile or any combination of the four profiles. However, communication between two endpoints requires that those endpoints both support the same profile.

Within each profile, specific features of the protocol may be defined as being optional. For example, all profiles define encryption as optional. This means that devices choosing to implement a profile would implement the upper layers of semantics with or without encryption, and then a deployment could decide whether to enable encryption. A device that does not implement encryption can be compliant to a profile, and deployments using that device could not enable encryption. Yet, communication between devices of the same profile shall be possible through the least common denominator feature set.

The following specifies the UET features included in each of the profiles. The keyword “MUST” indicates that a feature must be implemented. If a cell is empty, it means that the feature does not have to be supported, but vendors are free to support it in enhanced profiles (e.g., AI Base + Read). In this case, vendors should be sure to track all dependencies. Optional features may be disabled at runtime. That means that an implementation declaring support for AI Base may implement all features (which would

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

make it also HPC and AI Full compliant). The following tables assume familiarity with the respective UET subsections. All requirements apply to both best-effort and lossless deployments unless otherwise specified.

3.3.1 SES Transactions

Table 3-1 defines the requirements for supporting various transactions to be compliant with each profile. Multi-packet transactions such as UET_WRITE, UET_READ, and UET_SEND MUST use a consistent Payload MTU size to transport the SES payload. Therefore, a deployment MUST use a consistent Payload MTU size across all FEPs communicating in a fabric.

Table 3-1 - Profile Requirements for Supporting UET Transaction Types

Transaction	Description	AI Base	AI Full	HPC	Unified Local
NO_OP	Null message as defined by the UET_NO_OP opcode in SES Section 3.4.7.2	MUST	MUST	MUST	MUST
SEND	Send a message that uses the buffer at head of the receive queue	MUST	MUST	MUST	
SEND size	Required maximum msg size	1 MTU	4GB-1	4GB-1	
DATAGRAM SEND	Used for UUD	MUST	MUST	MUST	
TAGGED SEND (EM)	Exact tag match		MUST	MUST	
TAGGED SEND (WC)	Wildcard tag match			MUST	
WRITE	RMA write	MUST	MUST	MUST	MUST
WRITE IMM	RMA write with immediate	MUST	MUST	MUST	
WRITE size	Required maximum message size	4GB-1	4GB-1	4GB-1	16KB-1
READ	RMA read		MUST	MUST	MUST
READ	Required maximum message size	4GB-1	4GB-1	4GB-1	16KB-1
Non-fetching ATOMIC	Atomics support	MUST	MUST	MUST	MUST
Fetching ATOMIC (including CAS)	Atomics support		MUST	MUST	MUST
Tagged Atomics	Both fetching and non-fetching			MUST	
DEFERRABLE SEND	Send operations that can be deferred by the target until the corresponding receive buffer is posted		MUST		
DEFERRABLE TSEND	Deferrable send with matching		MUST		
RENDEZVOUS	with unexpected msg support			MUST	
SYNC WRITE SYNC WRITE IMM Non-Fetching ATOMIC	Sync group support				

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Transaction	Description	AI Base	AI Full	HPC	Unified Local
Note: 1. The libfabric provider advertises the max SEND message size **2.** Atomic operation support is determined via libfabric capability discovery (see SES Section 3.4.2.5.4).					

3.3.2 Buffer Addressing Mechanisms

Implementations of all profiles MUST conform to all basic buffer addressing schemes (described in SES Section 3.4.2.3) except for matching (described in SES Section 3.4.2.3.5). Implementations of all profiles MUST support the use of memory keys for UET_WRITE operations. All implementations MUST support the use of memory keys for UET_READ and UET_ATOMIC operations if RMA read is supported. Exact matching support may be implemented using wild-card matching.

Table 3-2 - Addressing Requirements for Implementations of Profiles

Operation	Description	AI Base	AI Full	HPC	Unified Local
Relative Addressing	Select libfabric endpoint using: {FA, JobID, PIDonFEP, RI[range]}	MUST	MUST	MUST	
Absolute Addressing	Select libfabric endpoint using: {FA, PIDonFEP, RI[range]}	MUST	MUST	MUST	
Exact tag match	Exact tag match for send operations		MUST	MUST	
Wildcard tag match	Wildcard tag match operations			MUST	
Direct Memory Addressing	Select target memory using: {FA, JobID}				MUST

3.3.3 Authorization

All profiles MUST implement the authorization semantics (SES Section 3.4.2.4). All profiles MUST be able to insert the JobID according to SES Section 3.4.2.4.1. The profile implementation MUST check the JobID before allowing buffer access. All profiles MUST support at least one JobID per FEP.

3.3.4 Buffer Behavior

The profiles MUST support the buffer behaviors shown in Table 3-3.

Table 3-3 - Profile Buffer Behavior Requirements

Transaction Type	Notes	AI Base	AI Full	HPC	Unified Local
Use-Once RMA	Use-once applicable to RMA operations			MUST	
Memory Key	Full range (standard header)	MUST	MUST	MUST	
	Limited range (optimized header)			MUST	

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Transaction Type	Notes	AI Base	AI Full	HPC	Unified Local
Direct Memory Addressing	Applies to RODL operations using UET_HDR_REQUEST_LCL				MUST
Tagged Operations: Exact Match	Applicable to tagged send and deferrable tagged send and rendezvous tagged send (if supported)		MUST	MUST	
Tagged Operations: Wild Card Match	Applicable to tagged send and rendezvous tagged send			MUST	
Multi receive	Support FI_MULTI_RECV				
Sync groups	Sync group support for writes with header data and non-fetching atomic completing actions				

3.3.5 Packet Formats

Table 3-4 lists the SES header formats (SES Section 3.4.3) that implementations of each profile MUST support. All fields and flags in the headers MUST be supported unless specifically called out as an exception in the table.

Table 3-4 - Profile Summary—SES header formats

Header Format	Description	AI Base	AI Full	HPC	Unified Local
Standard	44 B full header	MUST	MUST	MUST	
Deferrable Send	44 B header with restart token		MUST		
Deferrable Send RTR	44 B header with restart token		MUST		
Deferrable Send as Send	Target should treat arriving deferrable sends as send	MUST	MUST	MUST	
Optimized Non-matching	20 B header without match, initiator ID, header data, and message ID fields (single packet)			MUST	
Optimized Small Message	32 B header without message ID and with either header data OR offset (single packet)			MUST	
Optimized Local Transaction	12 B header without message ID, with support for packing messages. 8 B header using addressing relative to the previous message's address [UET_HDR_REQUEST_LCL]				MUST
Rendezvous Extension	24 B header with eager length and address information for read			MUST	
Atomic Extension	4 B header with opcode, datatype, control	MUST	MUST	MUST	MUST
Compare-and-Swap	Two operand header format		MUST	MUST	MUST

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Header Format	Description	AI Base	AI Full	HPC	Unified Local
Header					
Response	16 B header with return code	MUST	MUST	MUST	
Response with Data	Variable size header with 20 B response header + data		MUST	MUST	
Optimized Response	8 B compact header			MUST	
Optimized Local Response	4 B compact header				MUST
Message packing	FEP MUST accept and process up to 16 messages in one SES payload (one packet)				MUST
Delivery Complete (GO)	Delivery complete (DC) as implemented with global observability (GO)	MUST	MUST	MUST	MUST

3.3.6 PDS Ordering Modes

Table 3-5 lists the PDS ordering modes and network behavior that implementations of each profile MUST support.

Table 3-5 - Profile Summary—PDS ordering modes

Service	Description	AI Base	AI Full	HPC	Unified Local
Reliability and Ordering Modes	Reliable, unordered delivery (RUD)	MUST	MUST	MUST	
	Reliable, ordered delivery (ROD)	MUST	MUST	MUST	
	Reliable, ordered delivery for local networks (RODL)				MUST
	Reliable, unordered delivery for idempotent operations (RUDI)			MUST	
	Unreliable, unordered delivery (UUD)	MUST	MUST	MUST	

3.3.7 CMS Congestion Control Algorithms

CMS specifies two complementary congestion control algorithms, NSCC and RCCC, and supports programmable congestion management (PCM) for developing additional or experimental congestion control algorithms. For the specified algorithms, the following tables specify the implementation requirements in each profile. Independent requirements are provided for best-effort and lossless networks. Each implemented algorithm must support a mechanism to disable it at deployment time such that either NSCC or RCCC can run in isolation or together if the implementation supports both.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Table 3-6 - Profile Summary—Best Effort

Service	Description	AI Base	AI Full	HPC	Unified Local
NSCC	Network-signal-based congestion control	MUST	MUST	MUST	
RCCC	Receiver-controlled congestion control				

Table 3-7 - Profile Summary—Lossless

Service	Description	AI Base	AI Full	HPC	Unified Local
NSCC	Network-signal-based congestion control				
RCCC	Receiver-controlled congestion control				

3.3.8 INC Support

INC is an optional feature in UE. All features related to INC are to be implemented only on NICs with INC support.

Table 3-8 - Profile Summary—INC

Service	Description	AI Base	AI Full	HPC	Unified Local
Reserved PDCs	Reserved PDCs for INC				
TFC	Transport flow control				

3.3.9 Encapsulation

Many of the transport components of all profiles inherently rely on consistent lower-level protocol encapsulation (e.g., IPv4 vs. IPv6 as network layer encapsulation, UFH_16 vs. UFH_32 for the Unified Local Profile, and native UET vs. UDP encapsulation at the transport layer). For example, TSS uses confidentiality and encryption offsets for the secure domain specific to the encapsulation type, and CMS uses a nominal packet size to calculate bandwidth. An implementation SHOULD use consistent encapsulations on all packets on a FEP, PDC, and TSS security domain.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

3.4 Semantics Sublayer (SES)

UET specifies a semantic sublayer (SES) with support for a range of use cases. The first use case is traditional scale-out networking for AI and HPC using the AI Base, AI Full, and HPC profiles. A subset of these semantics is adapted and extended to support local scale-up use cases as part of the Unified Local profile.

3.4.1 Semantic Use Cases

The UET SES supports multiple use cases. These include:

- The OFI libfabric API
- Networks using the Unified Local profile.

3.4.1.1 Support for the OFI libfabric API

UET specifies a semantic sublayer (SES) that supports the OFI libfabric API. To support libfabric, the semantic sublayer uses concepts that have been deployed in HPC networking products as part of libfabric implementations. This specification defines optimized profiles for AI and HPC deployments that allow seamless interoperability between the common features of the profiles. Mappings of the popular send/receive semantics in *CCL to the proposed AI semantics are discussed in Section 3.4.10.

The products that have seeded this initial effort are inspired (to varying degrees) by the semantics exposed through the Portals 4.x API [5]. In particular, various pieces of nomenclature are similar to what is found in Portals 4, as is much of the addressing and authorization model. Portals 4 is mentioned here as a resource for additional insight into some of the concepts.

Informative Text:
SES uses the terms “initiator” and “target” in various places. Whereas many implementations use “sender” and “receiver” respectively, those terms are often difficult when talking about transactions like “read” or even more complex transactions like rendezvous. For example, in a rendezvous transaction, an initial request is sent from the initiator to the target. The target then issues a “read” to pull the data from the initiator.

Because libfabric was modeled after Portals 4, a strong semantic match remains between the two. However, libfabric does not define how to handle various challenges for a wire protocol that lie beneath the API (e.g., unexpected messages). Portals does not define a wire protocol either, and many of the Portals semantics relate to NIC or software behavior that is beyond the scope of UET (e.g., event delivery). By necessity, UET defines some behaviors of network hardware and software, but only to the extent necessary to build compatible devices. The UET on the wire protocol does not intend to achieve compliance with any existing semantics (neither IBTA nor Portals 4) and interoperates only with a peer UET device.

SES defines the behavior of processing the wire protocol only to the extent that is necessary to achieve interoperability between implementations of the libfabric API over the wire. FEP semantics and implementation details are left to individual implementation choices, and the split of the

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

implementation between hardware and software is not defined. Several details of the libfabric implementation requirements are not defined by the wire protocol. The libfabric mapping specification covers some additional requirements for achieving interoperable providers.

3.4.1.2 Support for Local Scale-up Networking

To support local scale-up networking within a consistent framework, SES leverages concepts, header fields, and opcodes from the semantics defined to support libfabric. Specific adaptations are incorporated to reduce header overhead while meeting the semantic requirements of the Unified Local profile. UE does not specify any particular API above SES to support the Unified Local profile.

Local scale-up use cases typically require much lighter-weight endpoints than scale-out deployments. This is often because they leverage endpoints incorporated into processing units to deliver substantially higher bandwidth. To support lightweight endpoints, scale-up use cases typically sacrifice multi-tenancy and scale (operating in the hundreds or thousands of endpoints rather than tens or hundreds of thousands). Throughout the SES definition, semantics concepts are defined in the context of libfabric and then specialized for the Unified Local profile.

3.4.2 Definition of Semantic Concepts

This section articulates the semantics provided by the UET to support AI and HPC applications. Fields within the header are articulated in this section, and the proper semantic handling of those fields is described; additional normative text is found in Section 3.4.4. The section starts by describing addressing: how is a buffer selected. This is followed by a discussion of authorization: the process of determining whether a message is allowed to access a buffer. In addition, this section covers the network operation types, how the ordering modes interact with the semantics provided, and network protocol security.

Informative Text:
Throughout the semantic section, the text refers to “buffers” or “data buffers.” In this specification, the term buffer is used generically to refer to an addressable section of memory. This includes a libfabric “memory region” (i.e., the target of an RMA operation) as well as the pointers passed into an *fi_tsend()/fi_trecv()* that refer to memory to send from or receive into. It includes areas described by a scatter/gather list as well.

3.4.2.1 Operations, Messages, and Transactions

A “transaction” is composed of all of the packets needed to eventually deliver the payload desired by the user and implement the libfabric request. A “message” consists of a set of packets sharing a single message ID. A transaction consists of one or more messages and supporting packets. As an example, a 32 KB libfabric *fi_send()* would be transmitted in eight packets, all sharing the same message ID that identifies the send transaction. A send transaction would include the acknowledgments and semantic responses to the send message packets. A more complex example would be a large *fi_send()* using rendezvous (3.4.5.3). This would consist of one or more packets as part of an eager message plus one or more packets as part of the read message (a second message). Each of those messages would have

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

associated responses and acknowledgments. All of these together would make up the rendezvous transaction.

The term “operation” is reserved for the behavior implemented at the endpoint. The operation is encoded in the opcode (3.4.7.2) and specifies operations such as reads and writes to memory, sending of messages, and atomic operations on memory. Operation is also used in a handful of cases when referring to libfabric concepts (e.g., a receive operation).

3.4.2.2 Services and Resources

A “service” is a libfabric-level construct that encapsulates all of the resources associated with a higher-level library (e.g., *CCL, MPI). Resources can include constructs such as buffers, completion queues, and completion counters. A “Resource Index” is an addressing construct used to select an addressable set of resources within a service. Three Resource Index spaces exist, corresponding to three operation types: RMA, SEND, and TAGGED. That is, Resource Index 0 for an RMA opcode (e.g., UET_WRITE) has a different meaning from Resource Index 0 for a SEND opcode (e.g., UET_DEFERRABLE_SEND). Similarly, TAGGED opcodes (e.g., UET_RENDEZVOUS_SEND) have a third meaning for Resource Index 0.

Informative Text:
Tagged and untagged messages inherently require two different “lists” at the target, which requires two different addressing points. The original mechanism considered was to have a single opcode (e.g., UET_PUT) for all, with the target resources addressed based on the configuration of the target resource. An alternative mechanism considered was to have each resource index be configured for what type of operation could be performed on it and reject operations of other types.

3.4.2.3 Addressing

Addressing describes the entire process of selecting a target and identifying a data buffer at the target. This may be the destination of data (e.g., for a Send (3.4.2.5.1)– or a Write (3.4.2.5.2), the source of the data (e.g., for a Read (3.4.2.5.3), or both a source and destination of data (e.g., for atomic operations). There are three types of addressing: relative addressing, absolute addressing, and direct memory addressing. The first two types rely on a Fabric Address (FA), PIDonFEP (3.4.2.3.2), and set of Resource Index (3.4.2.3.3) values to select a libfabric endpoint. They differ in how a PIDonFEP is interpreted. In relative addressing, the PIDonFEP is relative to the JobID (0). In absolute addressing, the PIDonFEP is interpreted without a JobID. The libfabric endpoint has a variety of resources that can include such things as a receive queue for send/receive operation, a set of matching buffers, RMA resources, and completion delivery—both completion queues and counters. This list does not attempt to be exhaustive, and the exact types and nature of the resources are not intended to be prescriptive.

For each matching, nonmatching, or RMA operation that an implementation supports, separate resources are identified by the combination of a Resource Index and operation type. Buffers provided for different operation types are addressed independently, as illustrated throughout the following examples.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

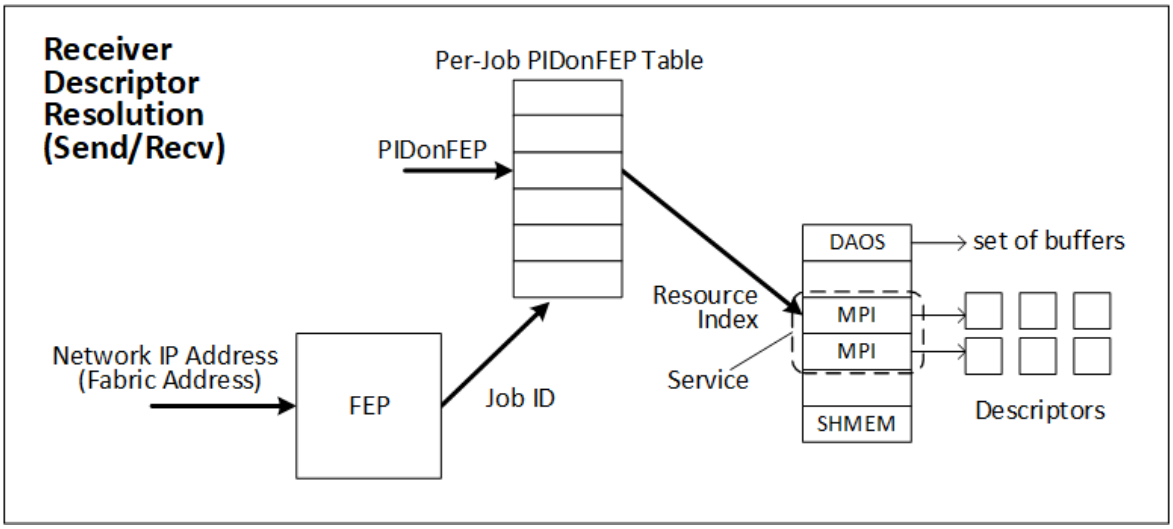


Figure 3-4 - Relative Addressing

The addressing hierarchy begins with a Fabric Address. An FA selects a fabric endpoint (FEP). Once a FEP is selected, two addressing modes are defined. Relative addressing is intended to enable scalable addressing for parallel communication within a distributed application, which is called a job. Absolute Addressing is intended to enable scalable addressing for client/server operations where the server is not required to be part of the application. In both cases, the initiator of a transaction is assigned a JobID (0) that is inserted in each packet in a trusted way (3.4.8.1). The JobID is an identifying property of the initiator only, and then is used as part of addressing and authorization (3.4.2.4) at the target.

In relative addressing, the FEP uses JobID to define the scope of the PIDonFEP. Within the PIDonFEP, a Resource Index (RI) is associated with a "service". A service can correspond to a specific use case or library. In libfabric, a libfabric endpoint is opened for the service, and the number of Resource Index values used depends on the service. The details of the usage of the Resource Index is hidden beneath the libfabric endpoint[4] that was opened in association with the service. The mapping of services to Resource Index values is covered in the libfabric mapping specification [6]. Within a service, each Resource Index creates a unique addressing space where a send operation, tagged send operation, or RMA operation can select a buffer. This is highlighted in Figure 3-4. Figure 3-4 specifically illustrates the operation of send/receive and tagged send/receive, where there is a pool of buffers behind a Resource Index. For send/receive, the first buffer is consumed from the queue by an incoming message. For tagged send/receive, the buffer is selected using the matching criteria. An initiator identifier is provided as part of the matching criteria (3.4.2.3.4). Note that send/receive and tagged send/receive have a different pool of buffers—even when they use the same Resource Index.

While Figure 3-4 provides an overview of how the addressing might be used in a complete HPC software stack—where multiple complex applications might share a node—it is also possible to implement a simplified version of this hierarchy for more purpose-built use cases (e.g., an integrated FEP in an AI

[4] Implementation of various libfabric features (e.g., scalable endpoints) can lead to a limited level of application level visibility of index values.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

accelerator). As illustrated in Figure 3-5, a FEP may choose to support only one JobID at a time to effectively remove the first level of indirection. In this model, the FEP is expected to validate that the received JobID is the same as the one JobID using the FEP. In addition, while the PIDonFEP plus Resource Index space is drawn hierarchically, the total number of bits dedicated to this addressing (24 bits) is consistent with implementation as a flat address space, if desired. PIDonFEP and Resource Index select a receive queue at the target, which is a resource that scales based on the messaging patterns rather than the total size of the system (in endpoints or ranks). Thus, similar solutions for direct mapping to resources could be employed. Finally, UET defines return codes and recovery mechanisms for how to cleanly handle transactions that go beyond the resources implemented by a particular device.

Informative Text:
A libfabric implementation associates multiple types of resources with a libfabric endpoint. This includes things such as completion queues, memory regions, and receive buffers. Because a PIDonFEP combined with a Resource Index is how an endpoint is addressed, this effectively selects a set of resources—some of which are directly addressable from the network (e.g., a memory region) and others which are not (e.g., a completion queue). Within a space selected by a PIDonFEP and Resource Index, a memory key can also address a memory region. Each resource may have an (implementation-defined) limited range.

Absolute addressing differs from relative addressing in that the PIDonFEP is an absolute number that spans the process space on the target node instead of being relative to a JobID. This allows a service to exist that is not associated with any job and allows addressing of those services within the UET framework. An absolute address facilitates a client directly addressing a process that is not part of its job, and the authorization model (3.4.2.4.1) enables scalable authorization using the JobID. For example, a client can use this approach to access a service residing at a well-known PIDonFEP (or a PIDonFEP obtained through some form of address resolution) on a server node. The JobID at that point is used

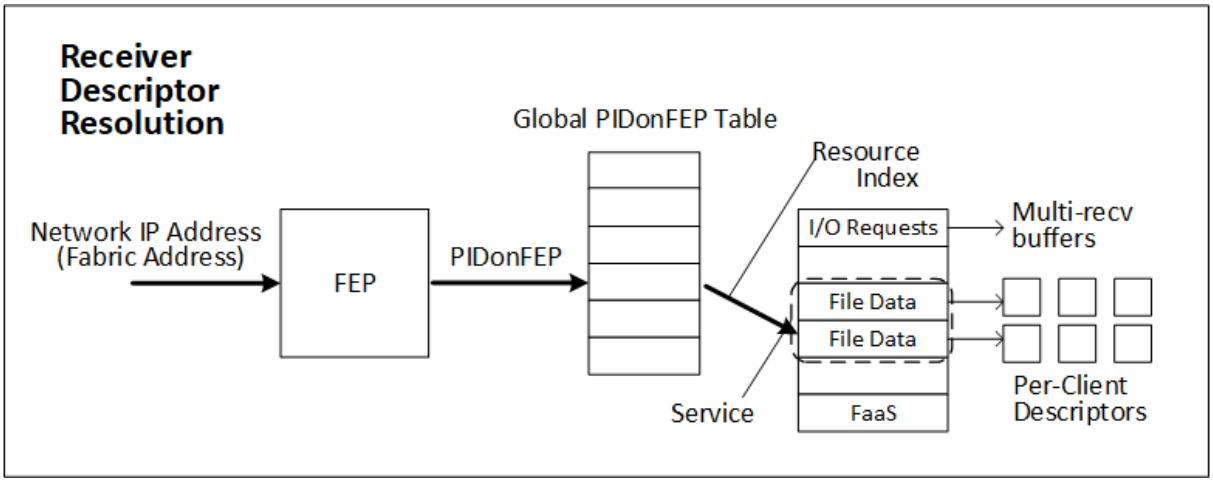


Figure 3-5 - Absolute Addressing

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

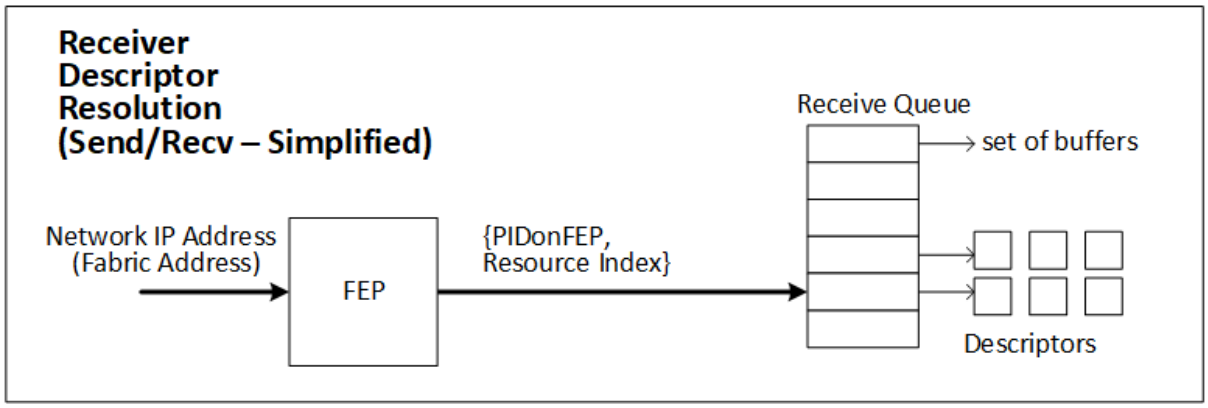


Figure 3-6 – Simplified Absolute Addressing

only as part of authorization to access a buffer (3.4.2.4.1). Buffer authorization in this scenario is expected to occur on a per-buffer basis (e.g., per memory region or per receive buffer).

RMA operations (e.g., *fi_read()* and *fi_write()*) use a Resource Index space independent from send/receive (and tagged send/receive) operations. Target memory regions are identified using a memory key. The namespace for a memory key is scoped by the Resource Index within a PIDonFEP or within the PIDonFEP independent of the Resource Index. A memory region that uses a Resource Index namespace within a PIDonFEP is referred to as an RI restricted memory region. Figure 3-7 illustrates this concept. A given Resource Index (e.g., RI 0) selects a service (e.g., SHMEM). Within the selected service, a memory key is mapped to a descriptor of a memory region using a vendor-defined mechanism. A provider uses a combination of FA, JobID , PIDonFEP, Resource Index, and memory key to select and access a memory region.

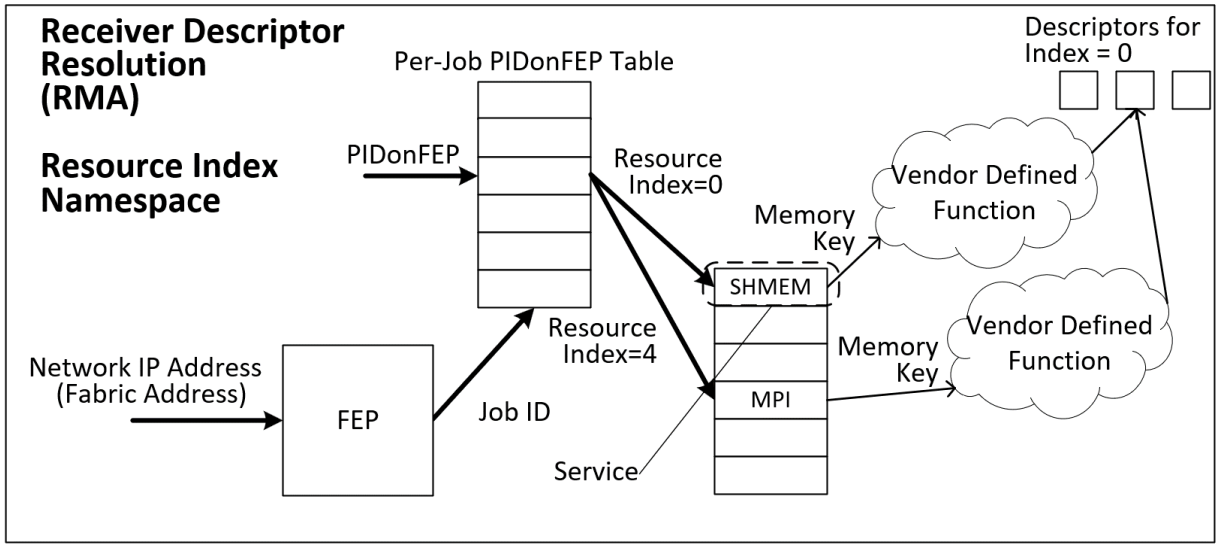


Figure 3-7 - Relative Addressing for RMA Operations using Resource Index Namespace

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

When the memory key namespace is the PIDonFEP, the memory key namespace is common for all services within the PIDonFEP and referred to as an unrestricted memory region. Figure 3-8 illustrates an unrestricted memory region. A provider uses a combination of FA, JobID, PIDonFEP, and memory key to select and access a memory region. When the memory region is unrestricted, any Resource Index within that PIDonFEP can access the memory region.

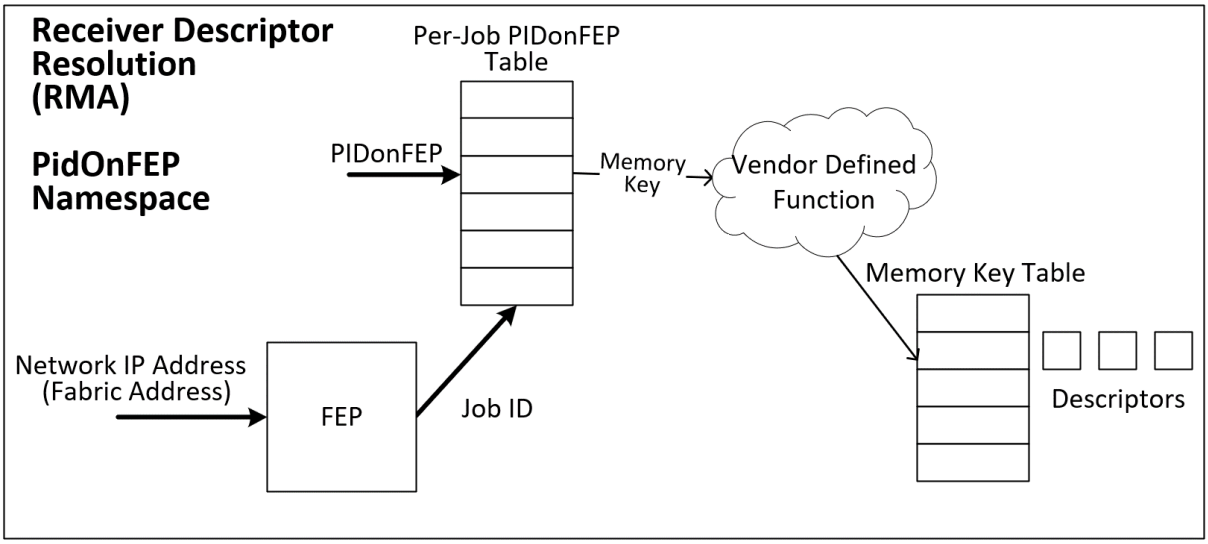


Figure 3-8 - Relative Addressing for RMA Operations using PIDonFEP Namespace

Both namespaces MUST comply with the memory key structure definition in Section 2.2.5.3.4.1 and the binding requirements in Table 2-47. When memory region mapping is not found or unallowed, a RC_BAD_MKEY error MUST be returned.

Implementation Note:
A memory region describes contiguous virtual memory space, sometimes called a buffer, at the target FEP. The memory key is a handle given to the sender and used to select this buffer. The sender is expected to use the allocated memory key when addressing the receiving FEP resource. Therefore, the mapping of a memory key to a memory region is solely a receiving FEP function. An implementation may choose to use PIDonFEP, Resource Index, or both namespaces.

Figure 3-9 illustrates the case where optimized headers (Figure 3-15) that do not include a memory key can be used. Here, a single buffer is associated with a Resource Index. A Resource Index is one of many associated with a service (e.g., SHMEM); thus, the Resource Index can directly select a buffer.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

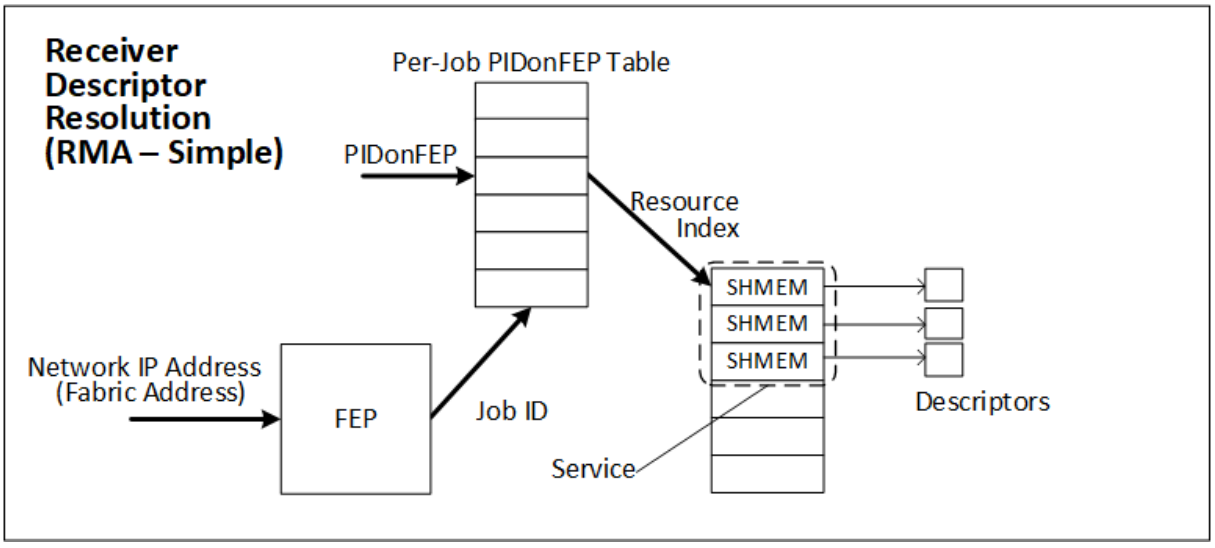


Figure 3-9 - Overview of Relative Addressing for RMA Operations Using Optimized Headers

Direct memory addressing is associated only with transactions using the Unified Local profile. These transactions do not carry PIDonFEP, Resource Index, memory key / match bits, or initiator fields; hence, they cannot use these as part of addressing or resource separation. Instead, a single network-visible address space is associated with the entire JobID at the target FEP. System-level software is responsible for managing addresses in this context. Translations (or, lack thereof) and protection mechanisms other than the JobID are outside the scope of UET. The target FEP MUST utilize the JobID for isolation. It may do this in one of two ways:

1. Allowing only a specific job to access memory at a given FEP.
2. Making the address relative to a base address associated with the JobID.

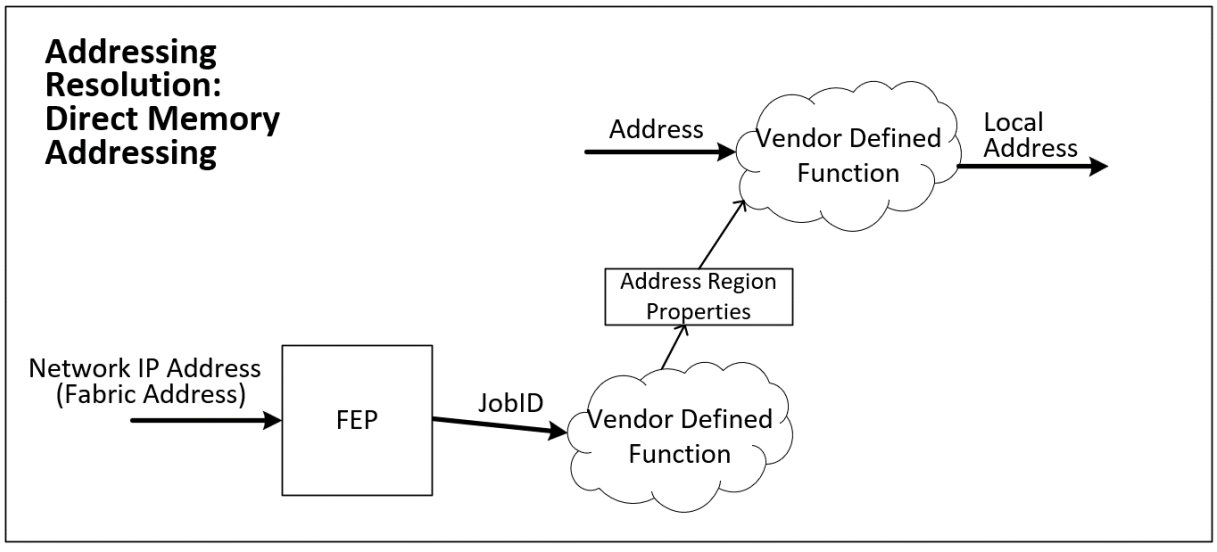


Figure 3-10 - Overview of Direct Memory Addressing

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

As illustrated in Figure 3-10, the lookup of the properties of the address region and the validation and translation of the address in the semantic header are vendor-defined functions.

3.4.2.3.1 Job Identifiers (JobID)

A JobID identifies an application (spanning one or more FEPs) that a communicating process belongs to (e.g., within a distributed parallel application). The JobID is assigned to the initiating process and is part of both addressing and buffer access authorization. In relative addressing for a FEP, it defines the scope of the PIDonFEP within the FEP. In absolute addressing for a FEP, JobID is used only for buffer access authorization. See 3.4.2.4.1 for an elaboration on the use of JobID for authorization. In all addressing modes, the JobID is populated with the JobID of the initiating application.

Job identifiers are useful in a traditional parallel computing environment (e.g., both HPC and AI) to segregate different user applications. They also provide a scalable authorization field when communicating in a client/server environment. A JobID is assigned by a provisioning system—much like a VXLAN ID would be. The numerical relationship of the JobID to the security domain is not defined by UET. The JobID MUST be unique for all jobs that are concurrently executing within the reachable network, and it could potentially utilize an existing ID if that is how the provisioning system is designed. Phrased differently, two entities that can reach each other across the network MUST have different JobIDs unless they are allowed to communicate. A distributed parallel application MAY be associated with more than one JobID.

Informative Text:
An example of a parallel application being associated with more than one JobID is the case where two separately launched distributed parallel applications need to communicate. These two jobs may be launched at somewhat different times such that the first job has no knowledge of the details of the second job at launch time. Multiple mechanisms can achieve this. The two historical solutions are: 1) Create a spanning JobID where each process uses relative addressing in two different jobs, and 2) Use absolute addressing. In either case, separate resources (e.g. separate sets of receive buffers and completion queues) are used for communicating between the two jobs.

Informative Text:
JobID assignment is conceptually similar to VXLAN ID assignment. The relationship of the JobID to a VXLAN ID (or other virtualization ID) is not defined by UET. In either case, the JobID could potentially be the same as a virtualization ID or a security domain, but that is not a requirement.

3.4.2.3.2 Process Identification (PIDonFEP)

UET provides a process identification field in the SES header called **ses.PIDonFEP**, which is used to select a set of resources associated with a specific process. Because operating systems tend to evolve independently of network protocols, the UET PIDonFEP is decoupled from the OS process ID as well as the PCI Express PASID. The PIDonFEP-to-OS process mapping is many-to-one. That is, each OS process using the network stack MUST have a different PIDonFEP, but more than one PIDonFEP may be assigned

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

to an OS process. A PIDonFEP MUST have either a scope that is relative to a JobID (relative addressing) or a scope that is relative to a Fabric Address (absolute addressing).

Informative Text:
A PIDonFEP used in relative addressing mode can be thought of as a "rank on node." Relative addressing mode is designed to allow hierarchical, algorithmic addressing so that a "rank"—such as for MPI or *CCL – can be expressed as a logical node and rank on that node. Thus, a CCL or MPI rank would be decomposed into "node X and PIDonFEP Y"—such as an application with 8 ranks per node translating Rank 33 into node 4 and PIDonFEP 1. In this case, PIDonFEP 1 says that Rank 33 is the second rank on node 4.

Informative Text:
Multiple PIDonFEPs have been bound to a single OS process in historical implementations to expand the flexibility of resource management. In most hardware architectures, this is "harmless" since the hardware must map PIDonFEP to some address translation context and typically uses structures that are agnostic to N:1 mappings. For example, if the Resource Index space proves to be too small for multi-programming model applications, it may be possible to map a service to an alternate PID to give it the full range of Resource Index values.

3.4.2.3.3 Separation of Services within a Process (Resource Index)

Within a process, it is often necessary to separate different usages of the network stack. For example, a user-level networking stack—such as MPI or *CCL—would want a separate set of communication resources from a user-level object storage system (e.g., Distributed Asynchronous Object Storage (DAOS) [7]). A specific set of resources within a process is a libfabric endpoint. A libfabric endpoint may acquire more than one Resource Index—as defined by the service that was used to open the endpoint. A communication context within that endpoint (e.g., a receive queue) is selected by the Resource Index. Different operation types (e.g., UET_SEND, UET_READ, and UET_TAGGED_SEND) have different Resource Index spaces. When used with a tagged operation on the wire, a Resource Index refers to a set of buffers using tag matching for buffer resolution (accessed using TSEND opcodes). The same numerical value of Resource Index used with a SEND opcode accesses a nonmatching queue. One or more RMA resources (accessed using UET_READ, UET_WRITE, UET_ATOMIC, and others) can be associated with a Resource Index. An RMA operation using an optimized header (Figure 3-15) MUST select exactly one RMA resource using the Resource Index. An RMA operation using UET_HDR_REQUEST_STD (Figure 3-11) MAY target a Resource Index that has exactly one RMA resource. This configuration is part of the Resource Index configuration at the target FEP.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Informative Text:
The text above notes that send/receive, tagged send/receive, and RMA opcodes have three different Resource Index spaces. That is, a Resource Index using send/receive selects different resources (completion queue, set of target buffers) from a tagged send/receive using that same Resource Index value. Similarly, an RMA opcode using that same numeric value accesses a third set of resources. Whether that is thought of as three separate Resource Index tables or one table with three sets of resources selected by the operation type is an implementation detail beyond the scope of this spec.

3.4.2.3.4 Initiator Identifiers

The **ses.initiator** field is included in the UET header and is part of the matching criteria (3.4.2.3.5). The **ses.initiator** field contains the job level "rank" of the initiating process. An initiator has meaning only within the context of a JobID (i.e., it is a rank numbered from 0 to N-1, where N is the total number of ranks in the application). As such, it MAY be assigned by a user-controllable resource. In many messaging APIs—including various CCL libraries and MPI—the receive operation (e.g., *MPI_Irecv()*) specifies the rank that a message must come from in order to match to that receive. This **ses.initiator** field is intended to carry that rank—or a proxy for that rank (e.g., the source rank could be translated to a global rank rather than the communicator-specific rank used in MPI). This is expressed in libfabric through the FI_DIRECTED_RECV portion of the API.

Informative Text:
The **ses.initiator** field is currently defined as the initiator ID and is relative to the JobID in both relative addressing and absolute addressing. The initiator ID does not currently have a separate trust model or addressing model for absolute addressing, because there is not a currently known use case for the initiator ID in absolute addressing mode. Future extensions of the specification may include updates to the initiator ID definition for absolute addressing if a well-defined use case emerges.

3.4.2.3.5 Matching and Nonmatching Operations

The UET header defines a set of fields to enable either matching or nonmatching operations. When the message indicates it is a tagged send (e.g., UET_TAGGED_SEND, UET_RENDEZVOUS_TSEND), the "matching criteria" (the 64 **ses.match_bits** field and the **ses.initiator_id** field) are used for buffer selection. When matching is not requested (e.g., UET_SEND, UET_RENDEZVOUS_SEND), a network message selects the next buffer provided by the Resource Index. The matching (if present) and nonmatching resources associated with a Resource Index are logically separated.

Matching operations exist in various forms. The traditional form of matching as defined through the libfabric *fi_trecv()* API uses matching criteria—specifically the **ses.match_bits** and the **ses.initiator** fields of the message—to select the buffer. In this approach (as exemplified in *fi_trecv()* and other historical software APIs), the matching criteria can provide ignore bits for each of the 64 match bits and/or be set to receive from "any source" (i.e., using FI_ADDR_UNSPEC—see *fi_trecv()*). Support for MPI matching requires wildcard matching for both a tag field and a source field, and this is accomplished through a combination of ignore bits and FI_ADDR_UNSPEC. For *CCL implementations that support send/recv, their matching behaviors are typically a subset of this behavior, where the matching criteria may not be wildcarded; however, that is a property of the libfabric endpoint and not the wire protocol. When

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

wildcard matching is used with ROD, the message MUST consume the entries in the FIFO order provided by software amongst all buffers that match the matching criteria; thus, for a given {FA, JobID, PIDonFEP, Resource Index} matching is performed in the order that buffers were added to that Resource Index. When RUD is used, the matching process SHOULD consume the entries in the FIFO order provided by software amongst all buffers that match the matching criteria—whether exact matching or wildcard matching is used; however, because RUD is unordered, the fabric is allowed to reorder packets. Entries are allowed to be consumed in any order when using RUD.

The match bits MAY be used as part of the RMA operations for libfabric. In this use case, the target of an RMA operation configures the resources identified by a {JobID, PIDonFEP, Resource Index} tuple in an RMA operation (e.g., *fi_write(), fi_read(), fi_atomic()*) as expressed as an RMA opcode (e.g., UET_WRITE, UET_READ, UET_ATOMIC, UET_FETCH_ATOMIC) for access using a "key" provided as a parameter of the *fi_write()* or *fi_read()* API. The implementation MUST map the key into the match bits on the wire. On some implementations, this key (as carried in the match bits) MAY have a very limited range and simply index a table of memory regions affiliated with a {JobID, PIDonFEP, Resource Index} tuple. On other implementations, some values of this key MAY use a hash table to identify a buffer associated with that {JobID, PIDonFEP, Resource Index, key} tuple. On still other implementations, the hardware MAY be simply configured to use the matching logic used to support tagged messaging. Each of these modes is enabled through the libfabric mapping because the libfabric API allows implementations to control the actual values of a memory key and how those values are used. Interoperability between various implementation strategies for the use of match bits is discussed in the libfabric mapping specification.

Informative Text:
The design of SES always provides an **ses.initiator** field to be part of the matching criteria. This enables libfabric to support FI_DIRECTED_RECV. An endpoint could be opened without FI_DIRECTED_RECV. In this case, the target side endpoint would ignore the **ses.initiator** field. This is a topic for coverage in the libfabric mapping specification but is noted here for the sake of the reader.

3.4.2.3.6 Memory Addressing

The UET header provides an offset within the buffer selected through the above processes. The offset is zero based (with respect to the buffer start) and matches the default memory region behavior for libfabric (formerly known as FI_MR_SCALABLE). Implementations MAY support the FI_MR_VIRT_ADDR option. FI_MR_ENDPOINT MAY be set by the user of the libfabric provider.

3.4.2.3.7 Addressing Summarized

A message that uses relative addressing selects a set of resources associated with a libfabric endpoint using a JobID, PIDonFEP, operation type, and Resource Index combination. For an untagged operation, a given {JobID, PIDonFEP, Resource Index} tuple MUST be used to select one buffer. If the message uses a tagged operation, the FEP uses the match bits as part of selecting a buffer. In a matching operation, a given {JobID, PIDonFEP, Resource Index, Match Criteria} tuple MUST be used to select one buffer; however, the FEP MAY use one or more mechanisms to uniquely identify that buffer (as defined in Section 3.4.2.3.5). The mechanism used MAY depend on the {JobID, PIDonFEP, Resource Index} tuple. An RMA operation MAY include a memory key in the same header field used for match bits (i.e.,

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

ses.memory_key). If a memory key is not used, the {JobID, PIDonFEP, Resource Index} tuple MUST uniquely select one buffer. Note that the Resource Index space used for untagged operations, tagged operations, and RMA operations are three separate spaces.

A message that uses absolute addressing selects a set of resources associated with a libfabric endpoint using a PIDonFEP and Resource Index combination. Within this set of resources, addressing proceeds in the same way as it does for relative addressing.

A message that uses direct memory addressing carries only the JobID and address. A single address space is visible per JobID on a target FEP. This address space is not required to be dense and linear, though implementations may choose to do that.

3.4.2.3.8 Addressing and libfabric [Informative]

This section is informational and provides the design intent for the addressing modes and how they are used in libfabric. Addressing for libfabric utilizing a UET provider is standardized in the libfabric mapping specification (see Section 2.2.5.1).

A libfabric endpoint is part of one or more JobIDs. It exists only in the context of one OS process, and it would typically exist on one PIDonFEP. A libfabric endpoint may allocate one or more values of Resource Index.

A libfabric endpoint is likely to utilize one or more {JobID, PIDonFEP, Resource Index} tuples. At least one motivation is to provide a control channel within the provider. Here, a control channel is defined as a mechanism through which two instances of a provider in different processes can exchange control information.

In libfabric, the *fi_addr_t* (e.g., *dest_addr*) determines whether relative addressing or absolute addressing is used.

For an *fi_send()*, the {JobID, PIDonFEP, Resource Index} tuple selects a set of resources in a libfabric endpoint that is similar to a shared receive queue.

For an *fi_tsend()*, the {JobID, PIDonFEP, Resource Index} tuple selects a construct similar to a shared receive queue in a libfabric endpoint. Within the resources selected by that tuple, the matching criteria (**ses.match_bits** plus **ses.initiator**) select a buffer based on the corresponding matching criteria provided in the *fi_trecv()*. It is expected that libfabric will be extended to allow an endpoint to specify that it supports only exact matching (i.e., that all of the ignore bits must be 0). Exact matching is a property of the libfabric endpoint—particularly the target libfabric endpoint—and not the wire protocol itself. The exact matching semantic is defined as not allowing a wildcard to be applied to the match field. It is intended to be used with exactly one copy of the matching criteria (**ses.match_bits** plus **ses.initiator**) in flight at any given time on the wire and exactly one copy of the match bits in use in the tagged receive list; however, that is not required. If these criteria are violated, then the order in which buffers are matched is undefined.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

In libfabric, RMA operations (e.g., *fi_write(), fi_read()*) utilize a 64-bit memory key. The wire formats provide two encodings of the key information for these operations. In one mechanism, only the {JobID, PIDonFEP, Resource Index} tuple is available in the UET header, and this chooses exactly one buffer. In this mechanism, the key pass through the libfabric API uses an encoding that is compatible with an abbreviated encoding on the wire—essentially placing all key information needed for addressing into the Resource Index itself. In the second mechanism, the wire protocol provides 64 bits for transporting the memory key (i.e., **ses.memory_key**). The {JobID, PIDonFEP, Resource Index} tuple selects a set of resources associated with a libfabric endpoint where the key has meaning and selects the completion queue for delivering libfabric completion data (i.e., the **ses.header_data** field in the transport header that is analogous to the immediate data for an RDMA Write-with-Immediate). The libfabric mapping specification fully defines the memory key encoding and how that encoding is utilized to create the relevant transport header formats.

Informative Text:
In libfabric, a memory key uniquely identifies a memory region that is the target of an RMA operation (e.g., *fi_read()*). The memory key in libfabric may be used to encode various information. The memory key is controlled by the provider implementation, which is how various usages and structures of the memory key achieve interoperability.

3.4.2.4 Authorization

After addressing a buffer, access to that buffer MUST be authorized as specified in 3.4.2.4.1 and 3.4.2.4.2. The security mechanisms outlined in the security chapter assist in establishing the identity of the sending device and authenticity of the header fields. Beyond that, the JobID plays a key role in buffer access authorization.

3.4.2.4.1 Job Identifier (JobID) and Authorization

The JobID MUST be assigned by or validated by a trusted layer within the initiating node (3.4.8.1). This can be within the FEP or OS—if the OS is part of the network trust boundary for a given deployment. The JobID can be assigned in any way that the provisioning system sees fit.

For every access to a buffer, the implementation MUST apply a mechanism to validate that the JobID is allowed to access the buffer. This mechanism can be as simple as the JobID being a required part of the mechanism to address the buffer (e.g., in relative addressing). In absolute addressing, this can require checking the JobID for each buffer. Implementations MUST support at least one JobID, and an implementation MAY support only one JobID and support only relative addressing or direct memory addressing; such an implementation can simply check the JobID of every inbound transaction. However, an implementation that simply checks that a JobID is one of a set supported by the FEP, without further applying a mechanism (e.g., relative addressing or a per-buffer check) to validate that JobID can access the buffer, is not sufficient.

3.4.2.4.2 Encryption and Authorization

Encryption is optionally supported and described in the transport security subsystem. The primary purpose of encryption in UET is to secure data and authenticate headers. Enforcement of fabric

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

addresses and JobIDs is covered in Section 3.4.8.1. To prevent server state scalability issues, UET does not require server-side mappings between a secure domain identifier (SDI) and JobID. The JobID is part of the identity of the initiating process in absolute addressing and is part of the shared identity of the initiating process and target process in relative addressing.

When encryption is enabled, a secure domain is logically mapped to a service from an application perspective. Only members 'bound' to the domain are allowed to use that secure domain. This is achieved by first selecting the endpoint using the {FA, JobID, PIDonFEP, Resource Index} tuple for relative addressing mode, the {FA, PIDonFEP, Resource Index} tuple for absolute addressing mode, or the {FA, JobID} for direct memory addressing mode. When encryption is enabled, the FEP MUST validate that the SDI is allowed to access that endpoint. If this validation fails, the message MUST be dropped and reported as defined in the transport security sublayer (TSS) Section 3.7.4. An SES response MUST NOT be sent in response to this failure. A PDS NACK MUST NOT be sent in response to this failure.

Implementation Note:
The SES is allowed to respond with UET_NO_RESPONSE before determining that the SDI validation for the endpoint failed. This can cause PDS to later indicate a UET_DEFAULT_RESPONSE for a given packet, but a message that fails SDI validation will never successfully complete. The PDC will not survive this and will eventually be torn down due to the lack of responses.

Informative Text:
UET is designed to avoid an amount of state that scales with the number of communicating peers (e.g., in a client/server environment, the number of clients of one server). As an example, the security specification includes a mode to allow all clients of a server to use a single SDI to minimize the required key storage state at the server—to make it independent of the number of clients. Having an SDI to JobID mapping would then require that the SDI maintain a list of all client JobID values that could use the SDI, which is an amount of state that scales with the total number of clients.

3.4.2.5 Network Transaction Types

Network transactions include various forms of tagged and untagged sends, RMA operations, and atomic operations. Not all transaction types may be supported on all networks. Graceful behaviors are defined for when an unsupported network transaction type is received (3.4.6.4.2). In general, all network transaction types work in conjunction with all variations of relative addressing and absolute addressing. Direct memory addressing is limited to read, write, atomic, and fetching atomic operations.

Network transactions that carry bulk payload (i.e., reads, writes, sends, tagged sends) MAY transfer up to 2^{32}– 1 bytes (4GB – 1) in a single message. This message size limit is defined by the maximum value that can be represented in the request length in the standard header format. Some header formats are defined that further constrain the size of an individual network transaction based on a narrower request. Individual implementations MAY limit the size of RMA operations and send operations separately. Such a limit is exposed through the max message size capabilities in libfabric. Implementations MUST break data into packets containing payload sizes conforming to a Payload MTU

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

(3.4.2.11). All packets in a message—other than the last packet—carry exactly Payload MTU bytes of data.

Informative Text:
Where implementations choose different maximum message sizes to export, that decision will impose requirements on middleware software (e.g., MPI or *CCL implementations). An initiator that fails to honor the limits of the target FEP will receive a return code indicating a failure to deliver the message. It is the responsibility of the middleware to coordinate the maximum message size limits across the environment. In practice, this is not typically a problem, since implementations have traditionally chosen from a small number of maximum message sizes (either the largest the transport supports or a single MTU), and the middleware is typically coded to choose from those options.

3.4.2.5.1 Sends—Tagged and Untagged

All sends cross the wire with addressing information—JobID, PIDonFEP, Resource Index, an optional offset into the target buffer, matching criteria[5], header data, and a payload. The target of the operation (e.g., the receiving process where the data will be deposited) then has some control over the exact handling of that message and the resulting change in the state of the buffer that addressing information specifies (3.4.2.6). Sends select the head of the untagged receive queue addressed by the {JobID, Process ID, Resource Index} and tagged sends use the matching criteria to choose one of the buffers associated with the tagged receive queue selected by {JobID, Process ID, Resource Index}. Send operations that arrive at a receive queue that does not have a posted receive buffer are handled using unexpected header processing (3.4.4.6.1). Similarly, tagged sends that do not find a matching entry follow the unexpected header processing procedures.

3.4.2.5.2 Writes

A write crosses the wire with addressing information—JobID, PIDonFEP, Resource Index, an offset into the target buffer, an optional memory key[6], header data (immediate data) and a payload. Direct memory addressing carries only JobID and an offset into the target buffer.

Writes select a single buffer based on the addressing criteria. The target of the operation (e.g., the memory region within the target process where the data will be deposited) then has some control over the exact handling of that message and the resulting change in the state of the buffer that addressing information specifies (3.4.2.6). For example, a target buffer could be defined as use-once or could be defined as read-only. For any access violation (e.g., the memory key is not valid, a write accesses a read-only buffer, etc.), a return code (Table 3-25) is generated that indicates the error.

3.4.2.5.3 Reads

A read utilizes the same addressing information as a write. Reads select a single buffer based on the addressing criteria. The target of the operation (e.g., the memory region within the target process from

[5] Matching criteria is currently included in the packet for all types of sends, but it is used only as part of buffer selection for tagged sends.
[6] RMA addressing may work in either of two formats—one with and one without a memory key on the wire.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

which the data will be retrieved) then has some control over the exact handling of that message and the resulting change in the state of the buffer that addressing information specifies (3.4.2.6). For example, a target buffer could be defined as use-once or could be defined as write-only. For any access violation (e.g., the memory key is not valid, a read accesses a write-only buffer, etc.), a return code (Table 3-25) is generated that indicates the error.

3.4.2.5.4 Atomic operations

Atomic operations are defined within UET. Both fetching atomic and non-fetching atomic operations are included. Atomics utilize the same addressing semantics as RMA read and write. Fetching atomic operations are limited to a single element of a datatype. Non-fetching atomic operations (or, just atomic operations) may be as large as a message—up to the full request length. Atomicity is limited in scope to a single element granularity and limited to the scope of a single {FA, JobID, PIDonFEP, Resource Index, Memory Key}. Variants of atomic operations using tagged addressing semantics are supported on profiles that support tagged sends, in which case atomicity is limited in scope to a {FA, JobID, PIDonFEP, Resource Index, Matching Criteria}. JobID is not part of determining the scope of atomics when absolute addressing is used.

3.4.2.5.4.1 Atomic Operations and Datatypes

A broad range of atomic datatypes and operations is supported within the UET definition. These types and operations target traditional HPC as well as AI/ML workloads. An enumeration of supported operations and datatypes is found in Section 3.4.7.4.

3.4.2.5.4.2 Atomic Operation Control Fields

Atomic operations are tightly tied to the memory model exposed by the network (3.4.9). As such, controls (Table 3-29) to convey the semantics required for the operation are provided as part of the atomic header definition.

3.4.2.5.5 Synchronized Operations

Synchronized operations are a class of operations designed to provide an aggregate completion across a collection of unordered RMA write operations. Synchronized operations utilize the same addressing information as writes (3.4.2.5.2) and atomics operations (3.4.2.5.4). In addition, they add a synchronization group field to track which operations are completed together and a synchronization count field to track the number of operations in the group.

Synchronized operations come in two forms: synchronized write operations and synchronized completing operations. Synchronized write operations behave the same as standard writes (3.4.2.5.2) but carry a sync group ID that identifies the write as part of a synchronization group. Synchronized completing operations are: synchronized write with header data (immediate data) and synchronized atomic operations. In each synchronization group, the synchronized completing operation is guaranteed to be performed at the target after the associated synchronized writes have been received and processed by SES.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

3.4.2.5.6 Rendezvous Send Transactions

A rendezvous send transaction is defined as an operation that sends a request from an initiator to a target. On successfully identifying a buffer, the target then "pulls" the data from the initiator. The initial request MAY have an "eager" portion of data that is transferred with it. Eager transfers are payload transfers before the buffer has been identified at the target. In a rendezvous transaction, the target controls how the eager portion of the transfer is handled. If the matching buffer is not found, it may be buffered, or it may be discarded and requested from the initiator later.

In UET, a rendezvous transaction begins with a rendezvous request (send or tsend). The rendezvous request includes information about a buffer at the initiator. This information is a full set of addressing information to enable the read to later retrieve the remainder of the full message payload. The read is semantically identical to other read operations in UET in all ways—including their packetization and implementation options as described in the packet delivery sublayer (PDS); however, the read for a rendezvous transaction requires completion tracking at the target beyond the typical scope of *fi_read()*. See 3.4.5.3 for additional discussion and for an implementation note on rendezvous implementations.

3.4.2.5.7 Deferrable Send Transactions

A deferrable send is a separate type of send transaction (for send or tsend) that can be deferred by the target of the send. Deferrable sends can then be resumed later. A deferrable send begins with a deferrable send request (send or tsend—see Table 3-23 for a list of opcodes) that includes a restart token containing a 32-bit portion allocated by the initiator and a 32-bit portion that will be filled in by the target. The target responds with a semantic response indicating whether the message was accepted or not. Accepted messages can be accepted in their entirety or partially. If the message was not accepted in its entirety, a later request from the target to the initiator—a ready to restart—uses a token to indicate to the initiator that the original message can be restarted. The ready-to-restart request can include an offset to allow it to start somewhere other than the start of the message. This allows the implementation to capture the initial portion of the message in an unexpected buffer and copy it when the matching receive (e.g., *fi_trecv()*) is posted.

Informative Text:
A deferrable send could be implemented with an "eager" limit; however, this is a matter of device architecture. In this context, an eager portion does not go into the wire protocol, because it does not impact the behavior of the target. The eager size is not part of the libfabric API.

Informative Text:
The deferrable send transaction provides a wire protocol implementation and optimization of an "eager long" protocol [8]. Conceptually, the round-trip time required for a rendezvous operation has a negative impact on the performance of moderate sized messages when the message is expected. Eager long protocols optimize for the expected message case. Deferrable sends go one step further by providing a wire protocol that reduces (though does not eliminate) the wasted bandwidth associated with unexpected messages using an eager long protocol. Deferrable sends also include a wire transaction that does not require implementing a read operation to pull the remainder of the data.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

3.4.2.5.8 Responses

Two types of responses are defined. The first is a response to send, tagged send, write, read (for large reads), and non-fetching atomic operations, which indicates the semantic result of that operation (3.4.4.3). The second is a response with payload, which is returned as the result of a read or fetching atomic operation.

3.4.2.6 Target Operation Types: Supported Buffer Behavior for Scale-out Networks

UET attempts to minimize the requirements placed on FEPs to maximize the flexibility afforded to implementations. Nevertheless, it is important that FEPs—and their associated software stacks—implement certain behaviors. Minimum requirements for hardware are discussed in Section 3.4.8. FEPs have expected behaviors for use-once, multi-receive, tagged, and RMA operations as described below. Direct memory addressing does not leverage any of the buffer behavior described here.

3.4.2.6.1 Use-Once Operation

One important buffer behavior for implementing libfabric is use-once behavior. This behavior means that once a single message has targeted the buffer, the buffer MUST NOT be accessible by other messages. The buffer MUST continue to be available for the one operation that started on it. The buffer MUST NOT be released back to the application (e.g., through a completion notification) until the operation targeting it has completed. Target-side FEP implementations MUST support use-once behavior, since this is the defined behavior for *fi_send()*/*fi_recv()*. If tagged messaging is supported, target-side FEP implementations MUST support use-once behavior as part of the tagged API implementation.

Target-side FEP Implementations of RMA (*fi_read()*/*fi_write()*) MAY implement a use-once semantic. Use-once can be a useful feature for building more secure implementations, as it minimizes the exposure window of a buffer. It can also be helpful for software to not have to explicitly tear down exposed buffers.

Informative Text:
Use-once behavior for RMA operations requires libfabric extensions. This behavior can be an important building block for secure protocols—especially in client/server operation. Many attack surfaces for unencrypted protocols depend on “guessing” the value of fields such as the **ses.memory_key**. Use-once behavior minimizes the lifetime of the exposure of the buffer, which reduces the success probability of these attacks. Use-once behavior is also a useful (though not strictly necessary) building block for the rendezvous protocol, since it eliminates the overhead of tearing down the buffer that was exposed by software to enable the target to issue a rendezvous read.

3.4.2.6.2 Multi-Receive

Target-side FEP Implementations SHOULD implement support for a multi-receive capability (i.e., the FI_MULTI_RECV option as specified for API calls such as *fi_recv()* in libfabric). Multi-receive allows a

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

single buffer to accept multiple requests with a locally managed offset “until it is full.” This applies to send and tagged send operations (and their rendezvous variants).

3.4.2.6.3 Tagged Operations

The target side of an implementation of libfabric over the HPC profile for UET MUST include support for tagged operations. Tagged operations MAY be accelerated with special mechanisms (e.g., dedicated hardware support); however, implementations that comply with the minimum requirements in Section 3.4.8 are also acceptable. Implementation of tagged operations MUST use the matching criteria (**ses.match_bits** plus **ses.initiator**) provided in the UET header.

3.4.2.6.4 Memory Key Size and Range

Target-side FEP implementations MUST support using the match bit field of a packet as a memory key for an RMA operation (e.g., *fi_write()*). The structure of the memory key is not defined as part of the transport definition and is defined in the libfabric mapping specification. Implementations have significant freedom to choose the exact usage of the memory key while complying with the libfabric definition.

3.4.2.7 Ordering

Ordering behavior is divided between the ordering semantics defined for RMA and messaging in support of libfabric and the ordering semantics associated with scale-up classes of accesses. The former is coupled to the ordering controls exposed through libfabric, while the latter is closely tied to the implementation.

3.4.2.7.1 Libfabric Ordering

Two basic forms of ordering are defined in libfabric; libfabric message ordering and payload ordering. The first is the ordering between libfabric messages—where a libfabric message here is loosely defined as any type of network operation — and the second is payload ordering which is the order in which bytes targeting a single address are delivered to memory at the target. In contrast, packet ordering is a property of the packet delivery sublayer (PDS) and is used to implement the ordering requested at the libfabric layer. The ordering requested at the libfabric layer passes through SES to select a PDS mode (e.g., ROD, RUD, RUDI) and how that PDS mode is used. Libfabric message ordering—defined as the order in which message headers are used to select a buffer—is distinct from payload ordering when utilizing underlying PDS modes. Payload ordering (3.4.2.7.1.2) refers to the order in which two different transactions operating on the same target address are applied.

Refer to Section 3.4.2.7.2 for information on ordering in the Unified Local profile.

3.4.2.7.1.1 Libfabric Message Ordering

Message ordering as defined by libfabric (e.g., send-after-send, send-after-write) requires that message headers be resolved at the target (e.g., matched or consume a receive queue entry) in the order that the messages were provided by software at the initiator. That is, message ordering begins with the order in which API calls (e.g., *fi_send()*) are made from software at the initiating libfabric endpoint and ends when a buffer has been selected (e.g., tag matching has concluded and the buffer marked as a use-once buffer is no longer available for other messages to match). Message ordering requirements apply to a

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

specific {initiator FA, target FA, initiator PIDonFEP, JobID, target PIDonFEP, Resource Index, traffic class} tuple only. To meet the libfabric message ordering semantics for send-after-send (and other message orderings), operations MUST use a ROD PDC to transport information required for resolving to a buffer at the target. This means using a ROD PDC for *fi_send()* or *fi_tsend()* when making a request—including a rendezvous request. Payload may be delivered out of order by using rendezvous transactions (3.4.5.3) or deferrable send transactions (3.4.5.4). In these transactions, the initial message (e.g., eager portion) is sent using a ROD PDC, and the remainder (e.g., the rendezvous read) is a separate message that may use a RUD PDC. Section 3.4.10 discusses the mapping of *CCL ordered send/receive over a tagged interface using a RUD PDC.

Informative Text:
Most implementations are likely to place multiple independent streams of ordered traffic onto a single PDC.

The mixture of RMA operations (e.g., *fi_write()*) with messaging (e.g., *fi_send()*) that requires message ordering (e.g., send-after-write) MAY use a ROD PDC. If it does not use a ROD PDC, mixed RMA and messaging MUST use source-side fencing to force the required ordering between messages. Source-side fencing refers to the action of waiting for the prior operations to complete—conceptually similar to a memory fence—and then initiating the subsequent transactions. This action would be applied at the semantic sublayer, if it were implemented.

3.4.2.7.1.2 Payload Ordering

Payload order refers to the ordering that two different operations from the same initiating PIDonFEP from a given initiator FEP perceive when accessing the same buffer associated with a single target FEP. Common ordering modes (i.e., typical CPU memory model ordering modes that are also defined in libfabric) include write-after-write (WAW), write-after-read (WAR), and read-after-write (RAW). RMA operations that request payload ordering MAY use a ROD PDC. This would effectively order the delivery of payload data such that the order in which it is presented to the target FEP would match the order of the requests issued by software. Even then, WAR ordering is likely to have a limited size as defined by the target implementation (i.e., as expressed through the max_order_war_size attribute in libfabric). To implement WAR ordering, targets using a ROD PDC to achieve ordering MUST buffer the result of the first read until an indication has been received from PDS that the result has been received by the initiator. This is indicated by PDS transmitting a Clear packet for the result. Typical implementations limit the size of data they buffer (e.g., to the maximum payload size of fetching atomic operations). Even with these mechanisms in place, implementations need to take care in how the host bus interface (e.g., PCI Express) semantics interact with the chosen ordering mode.

A second alternative for implementing payload ordering is to use initiator-side fencing in conjunction with a RUD, or even RUDI, PDC. Initiator-side fencing implies that after one operation is performed, the initiator waits for that operation to complete, and then the next operation is performed.

3.4.2.7.1.3 Order of Generation of Packets Within a Message

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Packets of a single message are generated with packet sequence numbers (PSNs) and offsets within the message that are ascending through the message. That is, the first packet of the message MUST have the first sequence number (e.g., PSN=A) used for the message and MUST have the **ses.som** bit set. The Nth packet MUST have PSN=A + N-1; thus, in the logical interface from SES to PDS, packets are provided in order. Similarly, the first packet of a message MUST have an offset within the message of 0. The Nth packet MUST carry an offset of (N-1) * Payload MTU in the **ses.header_data** field. The final packet of the message MUST have the **ses.eom** bit set. Packets from a single message MUST be contiguous in the sequence number space. That is, two messages MUST NOT be interleaved in their delivery to PDS for a single PDC.

For the ROD protocol, packets MUST be delivered to the wire in the order of their sequence number. For the RUD protocol, packets MAY be delivered to the wire in any order.

From the perspective of generating packets within a message, deferrable sends can behave like one message (i.e., when they are not deferred), in which case they follow single message packet generation ordering rules. Deferrable sends can also behave like two (or more) messages (i.e., when the deferrable send is deferred), in which case the two (or more) messages follow the ordering and interleaving rules of the PDCs that are used. The first message carries an **ses.som** bit on the first packet and generates M packets from the N packets that the overall payload of the deferrable send. It concludes with **ses.eom** set on the last packet of the original message. The second (and subsequent) message(s) also carries an **ses.som** bit on the first packet. Each message generates up to N packets, where N is the number of packets needed to transfer the entire message. Each message that is part of the deferrable send is treated as an independent message for the purposes of the above rules.

3.4.2.7.1.4 Completion Ordering

SES does not directly provide completion ordering. Completion ordering is a function of the FEP—and the hardware/software interface of the FEP. When using RUD or RUDI, two operations are not guaranteed to have any particular relationship in the completion of the network portion of the transaction—either at the initiator or target. Additional discussion of ordering (3.4.9.1) and completion ordering (3.4.9.1.2) is provided in the memory model description.

3.4.2.7.2 Unified Local Ordering

The Unified Local semantic layer is defined to run over an ordered reliability layer. This places all requests—reads, writes, and atomics—on the wire in the order they were generated and provides ordering to the target endpoint. Ordering is preserved only on a per-peer-FEP basis through the network. At the target, the ordering semantics depend on the memory semantics of the target device.

Implementation Note:
RODL on its own is insufficient to achieve per-address write-after-read (WAR) ordering. WAR ordering can be achieved only if a) RODL is used with the option to disable retransmits, or b) each read is marked as guaranteed delivery.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

3.4.2.7.3 Ordering of Response Data

For the ROD and RODL protocols, a read response (UET_RESPONSE_W_DATA) MUST be generated in the order that the read requests were received for a given PDC.

3.4.2.7.4 Synchronized Completion Ordering

Synchronized completion ordering is like completion ordering (3.4.2.7.1.4) in that there is no guarantee on completion ordering. Synchronized completing operations result in either a completion notification to the host or an atomic operation to host memory. Synchronized write with header data (immediate data) operations deliver a completion notification at the target after all synchronized operations have completed. Synchronized atomic operations perform an atomic operation after all synchronized operations have completed. Completion notifications from synchronized completing operations are not ordered relative to other synchronized completing operations. Atomic operations that result from synchronized atomic operations are not performed in any defined order relative to any other atomic operations. There are no changes to sender completion processing.

3.4.2.8 Protocol Isolation Mechanisms

SES provides mechanisms for isolation between jobs on a system. This consists of controlling access to buffers using device-enforced identities. A FEP is validated as being within a given security domain using either cryptographic methods (see Section 3.7.4) or other site-specific methods beyond the scope of UET. In a client/server environment, however, this does not indicate which buffers a message may access. Buffer access authorization depends on the identify information in a UET header (3.4.2.4) being properly enforced (3.4.8.1).

3.4.2.9 Header Data

Libfabric defines completion data that can have a size up to 64 bits. UET supports this with a field called **ses.header_data** that is 64 bits in size. In libfabric, completion data is delivered as part of the completion queue entry and does not consume an entry from the receive queue. Header data is provided only in the start of message packet, and only if the **ses.hd** bit indicates that it was provided by software.

3.4.2.10 Additional Control Fields

Some additional bits of control are included in the UET header (Table 3-9). There are two bits of version information (**ses.ver**). Multi-packet messages include a **ses.message_length** field that describes the entire length of the payload to be delivered. One bit (**ses.dc**) is included to request that the semantic response indicates that the packet has been made globally observable (3.4.9.3). One bit is used to indicate that the message encountered an error at the source (**ses.ie**) and should terminate the message in error. As an example, a PCIe transaction may fail in the middle of a packet in the middle of a message. Similarly, an address translation may fail for a packet in the middle of a message. One bit indicates relative or absolute addressing (**ses.rel**).

For transactions using the Unified Local profile, a control bit (**ses.mm**) enables multiple semantic messages to be placed in a single packet. A second control bit (**ses.ac**) identifies semantic headers

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

utilizing address compression. An additional control field (**ses.ehl**) allows up to 16 bytes of implementation specific fields per message.

3.4.2.11 Packet Sizes Based on Payload MTU

Packets for SES utilize a programmable Payload MTU. The Payload MTU is the amount of payload (in bytes) that can be carried in a maximum-sized packet. The Payload MTU is configured (out-of-band) to a value of 1024, 2048, 4096, or 8192. FEP implementations MUST support a Payload MTU of 4096 bytes. FEP implementations SHOULD support Payload MTU of 1024 bytes, 2048 bytes, and 8192 bytes. The default for Payload MTU is 4096 bytes, and this size is used for various architectural decisions.

The Payload MTU MUST be set such that the resulting packet fits within the limits defined by the Ethernet MTU along the entire path. The “don’t fragment” bit MUST be set in the IPv4 header when running over IPv4. The source of a message MUST put exactly Payload MTU bytes into every packet of the message—except the final packet. A generated packet MUST NOT be further fragmented by the FEP.

For packets leveraging the **ses.mm** control bit, the total payload length must not exceed the Payload MTU.

3.4.2.12 Zero-Byte Operations

Zero-byte operations are supported for read, write, send, tagged send, and the atomic put and atomic get operations. Zero-byte reads and atomic gets issue a memory read at the target using the specified offset. Zero-byte writes, sends, tagged sends, and atomic puts do not write memory but can generate a completion at the target. Zero-byte writes, sends, tagged sends, and atomic puts also perform whatever operation is necessary to honor the delivery complete setting for the message. Zero-byte operations undergo the same checks (e.g., whether the buffer being written is writeable, whether the address is valid, etc.) and generate the same return codes as all other operations. Similarly, a zero-byte send or tagged send consumes the corresponding buffer at the target—using the same target-side controls as a one-byte send or tagged send would. A zero-byte send or tagged send resolving to a multi-receive buffer consumes no space in the buffer. Zero-byte operations use the same header formats—both PDS and semantic headers—as a corresponding one-byte operation would use. The only difference is that the length field is set to 0. Because the payload length of a zero-byte response is 0, it inherently passes the PDS test of “payload_len <= PDS_MAX_ACK_DATA”.

3.4.2.13 Interaction of Semantics with Reliability Modes

The packet delivery sublayer (PDS) has five delivery modes. Throughout the semantic specification, differences in behavior between reliable ordered delivery (ROD) and reliable unordered delivery (RUD) are specifically enumerated. Unless otherwise stated, all functionality is available for both ROD and RUD delivery modes. PDS also supports an unreliable unordered datagram (UUD) mode. The only semantic provided with UUD is the opcode UET_DATAGRAM_SEND. UUD is provided to enable applications to use a datagram transport (e.g., like UDP) without having to use a different communication library. UET_DATAGRAM_SEND utilizes the same addressing mechanisms as ROD and RUD. As an untagged send, it does not utilize matching. UET_DATAGRAM_SEND supports only single-packet messages.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

The RUDI mode—reliable unordered delivery for idempotent messages—has specific constraints in the way it is utilized. With RUDI, PDS provides reliable delivery but does not provide deduplication. This means that certain scenarios (e.g., lost acknowledgements) can lead to the duplicate delivery of packets at the target. RUDI is designed for semantics that are idempotent. For example, *fi_read()* and *fi_write()* are idempotent when they do not deliver a target completion. Semantics that have side effects are not "safe" when using RUDI. This includes send and tagged send, since both consume a buffer at the target. Similarly, *fi_atomic()* would be exposed to delivering the operation to the target twice.

The RODL delivery mode is a variant of ROD optimized for lossless service andnetworks using the Unified Local profile. RODL supports only the semantic header formats (UET_HDR_REQUEST_LCL and UET_HDR_RESPONSE_DATA_LCL) and associated operations designated in the Unified Local profile.

3.4.2.14 Message Identifiers and Message Construction

Each multi-packet message contains a message identifier (**ses.message_id**). Single-packet read request (UET_READ) messages using the standard header (Figure 3-11) MUST include a message ID for the **ses.message_id** field. Other single-packet requests that have the **ses.message_id** field MUST include a message ID or set the value to 0 if no message ID is provided. The message ID for a UET_DATAGRAM_SEND must be set to 0. The message ID MUST be unique across all messages in flight from one initiator FEP to a given target FEP using a given target PDC. The message ID MAY be globally unique for a given initiator FEP.

If a message ID was present in the request, the read responses (UET_RESPONSE_W_DATA) carry the original message ID from the read request in the **ses.read_request_message_id** field. Otherwise, the **ses.read_request_message_id** field is set to 0. These responses also carry their own response message ID (**ses.response_message_id**). The response message ID does not have the same uniqueness requirements as the message ID in the request.

Informative Text:
The initiator can track all the information needed to implement read transactions without requiring uniqueness in **ses.response_message_id**. This frees the target implementation to leverage the response message ID in whatever way it desires. This includes the full range of implementation options, from always populating the **ses.response_message_id** with 0 to inserting a random value to having a unique response message ID that is used for all responses to a given read.

3.4.2.15 Original Request PSN

Certain optimized header formats for RMA operations do not carry a message ID. These formats use the PSN from the original PDS Request packet to identify the original request state at the initiator. This allows an initiator to associate return data with the original SES read request when the original message ID is not present.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Implementation Note:
The standard SES header carries a **ses.message_id** that is carried back to the initiator in the SES "response with data header" (i.e., with read response data). Some optimized SES headers do not carry a message ID. When the message ID is not present, the original PSN is used to identify the transaction at the initiator. PDS passes each PSN to SES. The original request PSN is needed only for the optimized response with data header (Figure 3-27). In these cases, SES passes it back to the initiator in a special header field.

3.4.3 Semantic Header Formats

Several header formats are needed to implement UET. Those formats are shown here in both illustrated and tabular forms. Some header fields (e.g., **ses.opcode**) use enumerated types defined in Section 3.4.7. All reserved fields MUST be set to 0 and ignored upon receipt. Header formats are selected based on the guidelines in 3.4.3.8.

3.4.3.1 Standard Header Format

The standard header format shown in Table 3-9 is used for most operation types. Table 3-9 illustrates the fields present when **ses.som** is set to 1, and Table 3-10 presents the slight variation on the header for when **ses.som** is set to 0. Implementations MUST support using the UET_HDR_REQUEST_STD format for any request operation in the AI Base, AI Full, and HPC profiles. It is used for any multi-packet operation and for any operation that requires matching criteria—including those that need a full memory key as part of implementing *fi_write()* or *fi_read()*. Either the UET_HDR_REQUEST_STD or the UET_HDR_REQUEST_MEDIUM header in Figure 3-16 MUST be used when completion data is needed. The field names and sizes are summarized in Table 3-9.

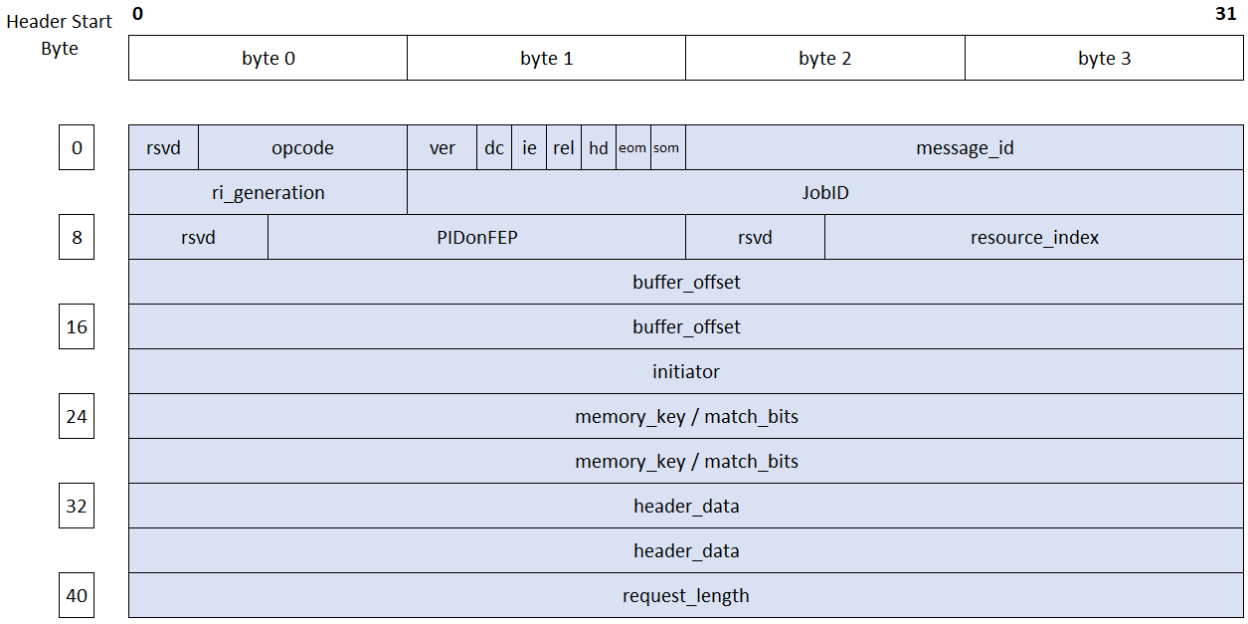


Figure 3-11 - Standard Header Format when ses.som is 1 (UET_HDR_REQUEST_STD)

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Table 3-9 - Standard Header Format Fields when ses.som is 1

Field	Size	Description	Section Ref
rsvd	2	Reserved. MUST be 0.	
opcode	6	The operation being performed for this packet. (Table 3-23)	3.4.7.2
version (ver)	2	Semantic protocol version; Set to 0 in the initial version.	
Delivery Complete (dc)	1	Defer the semantic response for this until the packet has been made globally observable (3.4.9.3). This matches the FI_DELIVERY_COMPLETE option in libfabric.	3.4.9.3
Initiator Error (ie)	1	Indicates this packet encountered an error at the initiator. Initiator Error prevents the packet from being written at the target. Initiator Error should be set only for packets that MUST cause the message to complete in error. This is primarily designed for messages consisting of more than one packet.	3.4.6.4.1
Relative (rel)	1	This packet uses relative addressing.	3.4.2.3
header data present (hd)	1	Header data was provided for this message.	3.4.2.9
End of Msg (eom)	1	Indicates the last packet of the message. **ses.eom** MUST be set on the last packet of a message.	3.4.2.7.1.3
Start of Msg(som)	1	Indicates this is the first packet of a message. Impacts the interpretation of header data.	3.4.2.7.1.3
message_id	16	Message identifier: Assists in associating different packets to one message at the target. Also assists in reverse-mapping responses to message. The value of 0 is reserved to indicate that the message ID is not valid.	3.4.2.14
ri_generation	8	Resource Index Generation	3.4.4.7.3.2
JobID	24	JobID used for relative addressing and for buffer access authorization. Note: Matches size of VXLAN VNI.	3.4.2.3.1
rsvd	4	Reserved. MUST be 0.	
PIDonFEP	12	The PIDonFEP value to be used at the target.	3.4.2.3.2
rsvd	4	Reserved. MUST be 0.	
resource_index	12	Resource Index field.	3.4.2.3.3
buffer_offset	64	Offset within the target buffer used for 0 based addressing. The first memory access of the first packet in a message begins at Buffer Offset bytes from the base of the memory region selected. In multi-packet messages, Buffer Offset is the same across packets, so that the Buffer Offset and Request Length can	

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Field	Size	Description	Section Ref
		be used together to determine if the message fits in the target buffer. The access address of a given packet is Buffer Base Address (from the memory region) + Buffer Offset + the offset within the message (taken from the header data field for packets on which **ses.som**=0). Special use case for deferrable send requests: carries the restart token. All active restart tokens must be unique at the initiating FEP. The upper 32 bits of the restart token are allocated by the initiator, and the lower 32 bits are set to 0. Usage of FI_MR_VIRT_ADDR: Some implementations may support the setting of FI_MR_VIRT_ADDR in the libfabric API. In these cases, the buffer offset field of the packet carries the absolute virtual address where the payload is delivered.	
initiator	32	Initiator ID used as part of matching criteria.	3.4.2.3.4
match_bits / memory_key	64	Used for tagged matching or as a memory key, depending on the opcode being used. In ready-to-restart requests (UET_DEFERRABLE_RTR), this field carries the upper 32 bits of the restart token that was part of the deferrable send request as well as 32 bits allocated by the target in the lower 32 bits.	3.4.2.3.5
header_data	64	This is the completion data to deliver at the target when this operation completes when **ses.som**=1. If **ses.hd**=0, this field is ignored. See Table 3-10 for usage when **ses.som**=0.	3.4.2.11
request_length	32	Length of the payload to be transferred (in bytes). 0 is a legal transfer size (0 byte write/read). Maximum size is 2^{32}-1. The request length field MUST be populated both when **ses.som**=1 and **ses.som**=0.	

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

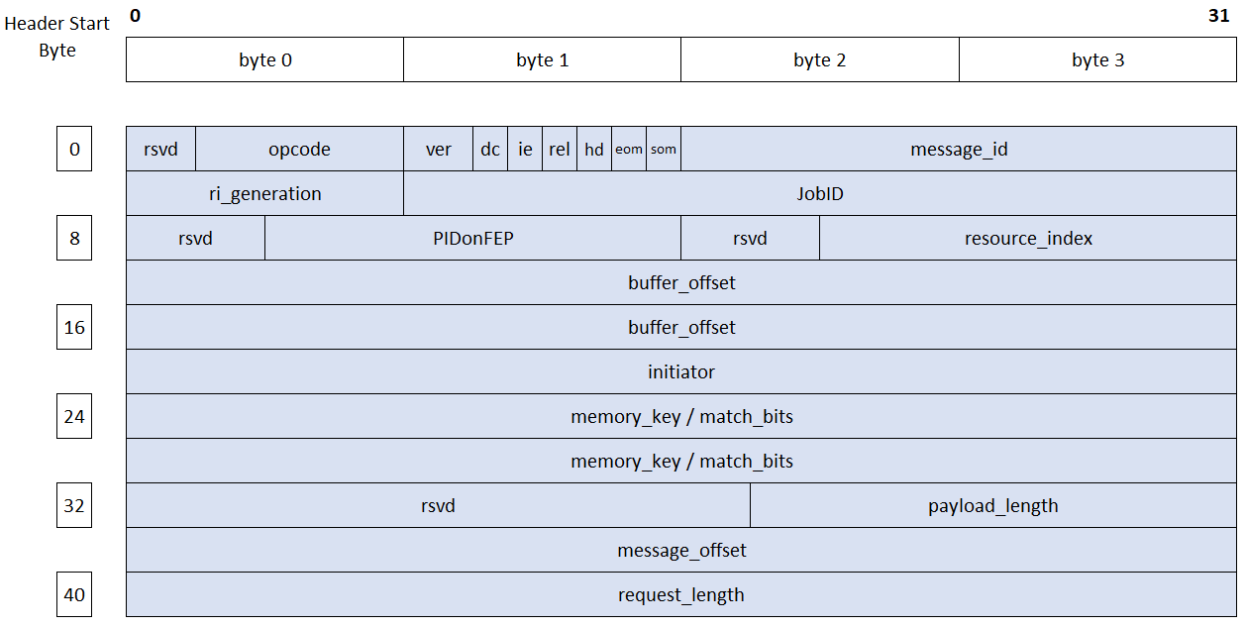


Figure 3-12 - Standard Header Format when ses.som is 0 (UET_HDR_REQUEST_STD)

Table 3-10 - Standard Header Format Fields when ses.som is 0

Field	Size	Description	Section Ref
rsvd	2	Table 3-9	
opcode	6	Table 3-9	3.4.7.2
version (ver)	2	Table 3-9	
Delivery Complete (dc)	1	Table 3-9	3.4.9.3
Initiator Error (ie)	1	Table 3-9	3.4.6.4.1
Relative (rel)	1	Table 3-9	3.4.2.3
header data present (hd)	1	Table 3-9	3.4.2.9
End of Msg (eom)	1	Table 3-9	
Start of Msg(som)	1	Table 3-9	
message_id	16	Table 3-9	3.4.2.14
ri_generation	8	Table 3-9	3.4.4.7.3
JobID	24	Table 3-9	3.4.2.3.1
rsvd	4	Table 3-9	
PIDonFEP	12	Table 3-9	3.4.2.3.2
rsvd	4	Table 3-9	
resource_index	12	Table 3-9	3.4.2.3.3
buffer_offset	64	Table 3-9	
initiator	32	Table 3-9	3.4.2.3.4
match_bits	64	Table 3-9	3.4.2.3.5
rsvd	18	Reserved. MUST be 0.	
payload_length	14	Length (in bytes) of the payload portion of this packet.	

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Field	Size	Description	Section Ref
message_offset	32	32-bit offset (in bytes) from the start of the message.	
request_length	32	Table 3-9. The request length field MUST be populated both when **ses.som**=1 and **ses.som**=0.	

A special case form of the UET_HDR_REQUEST_STD format is used for deferrable sends (Figure 3-13). Deferrable sends are designed for *CCL-style messaging where unexpected messages may occur, and the sequence of messages in a deferrable send is illustrated in 3.4.5.4. Deferrable sends always deliver data starting at the first byte of the receive buffer; thus, deferrable sends do not require a buffer offset field. This allows the offset in UET_HDR_REQUEST_STD to be replaced by a restart token, which is broken into an initiator restart token and a target restart token. The upper half of the restart token is entirely defined by the initiating FEP. This allows implementations to choose how the bits are populated and how they are encoded. The lower half is set to zero in the initial request and contains the target restart token when the operation is restarted using a ready-to-restart message.

The ready-to-restart (RTR) message (Figure 3-14) in the deferrable send sequence uses a similar special case. In the ready-to-restart message, the restart token is placed in the match bits. This consists of an echo of the initiator restart token as well as a restart token allocated by the target. The target is not required to allocate a restart token. If the target does not allocate a restart token, it MUST populate the target restart token field with 0. If the target allocates a restart token, it MUST accept the restarted deferrable send. If the target does not allocate a restart token, it MAY defer the deferrable send again. An offset from the start of the buffer originally being sent is placed in the buffer offset field. This buffer offset is limited to the range 0 to 2^{32}-2 and is used to select the portion of the transfer that the target did not capture when it was first transmitted. When the deferrable send restarts, it creates a new message using the format in Figure 3-13 and carries the full restart token (initiator restart token and target restart token) that was provided in the RTR message. As with all other send operations, the offset relative to the start of this new message is carried in the header data field. When the deferrable send restarts, it MUST set the **ses.som** bit. A deferrable send can only be restarted if a corresponding buffer is not found at the target. A deferrable send MUST be restarted using an RTR message containing a target restart token value other than 0 exactly once and MUST NOT be deferred after being restarted by an

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

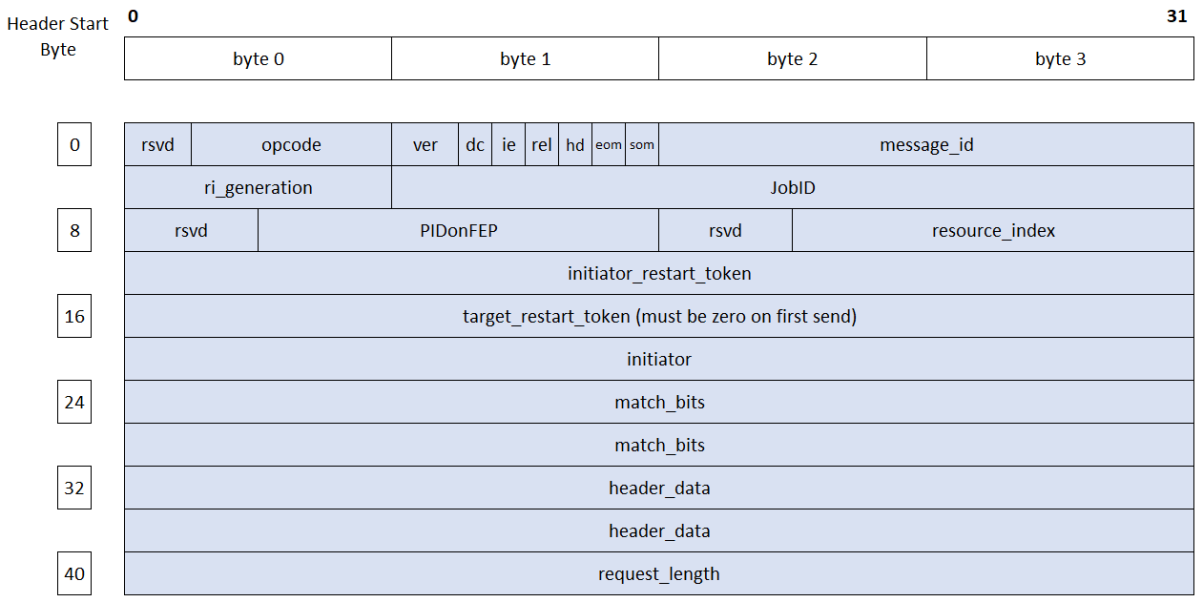


Figure 3-13 - Standard Header Format as Used for Deferrable Sends

RTR message containing a target restart token value other than 0. Any deferrable send that contains a target restart token value of 0 MAY be deferred.

Each time the deferrable send is restarted, it MUST carry the addressing fields and header data from the original request. This allows for implementations where the deferrable send is restarted multiple times without reserving a buffer at the target. The request length in the restarted deferrable send MUST be the request length indicated by the target in the RTR message.

Informative Text:
Deferrable sends can utilize the various resource exhaustion sequences described in 3.4.4.6.1. These sequences do not count as deferrals of the deferrable send.

Implementation Note:
It should be noted that the restarted deferrable send cannot carry a starting offset for delivering the payload; thus, if the target has buffered a portion of the payload and the restart of the deferrable send begins at a non-zero offset into the buffer, the target is required to remember this.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

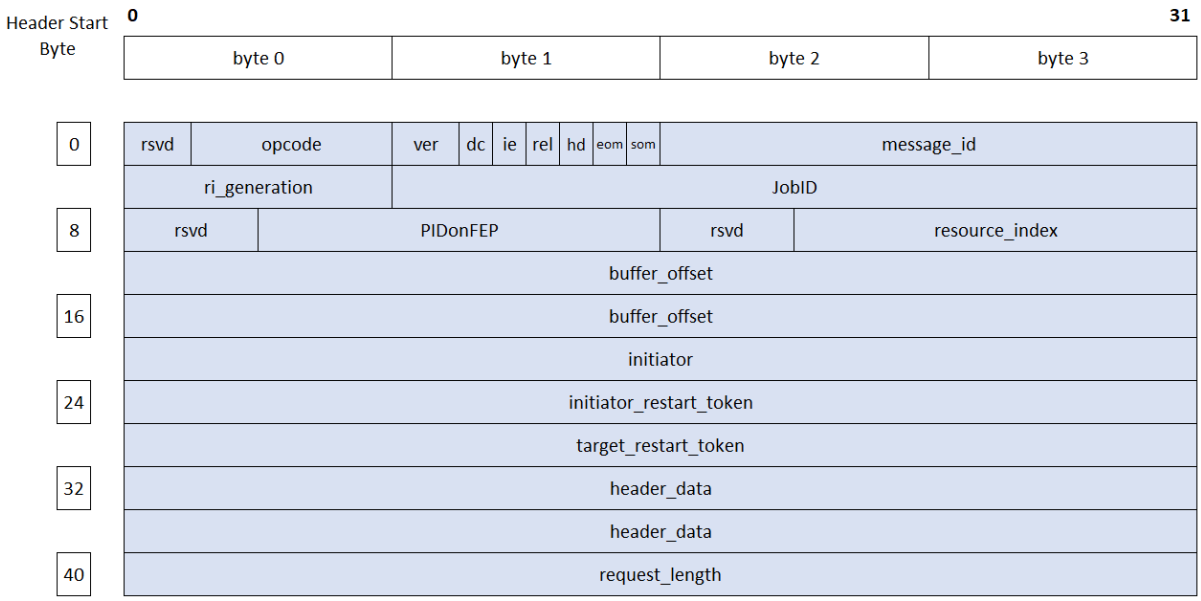


Figure 3-14 - Standard Header Format as Used for Ready-to-Restart Requests

3.4.3.2 Optimized Header Formats for Scale-out

Two use cases of small transfers motivate creating a set of optimized semantic headers. The first of these is the case of non-matching transfers that do not require header data. These single packet messages can eliminate the **ses.match_bits** and **ses.initiator** fields (i.e., all matching criteria) as well as the **ses.header_data** and **ses.message_id**. They carry an abbreviated (14-bit) **ses.request_length** field that enables single packet payloads up to 8192 bytes. It should be noted that the packet sequence number (PSN) from the packet delivery context (PDC) is needed within the semantic implementation to resolve responses back to the original request since a message ID is not carried in the optimized header format. Single-packet messages using the format in Figure 3-15 set both **ses.som** and **ses.eom**.

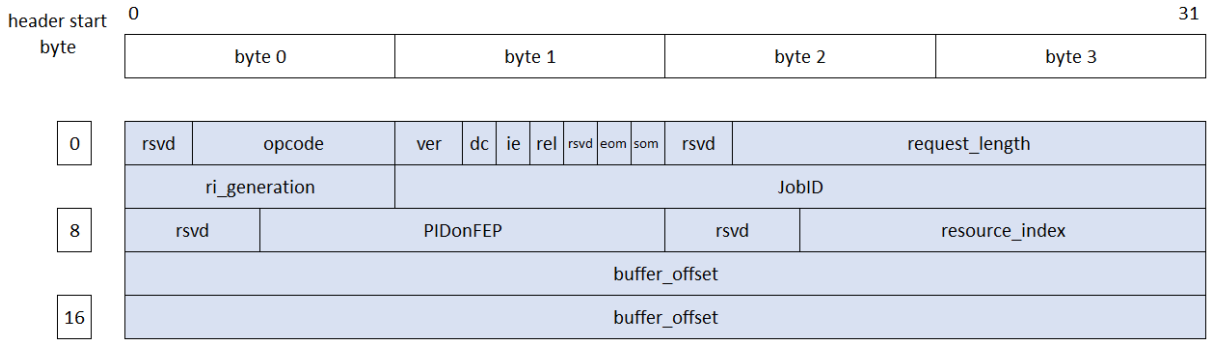


Figure 3-15 - Optimized, Non-Matching Format (UET_HDR_REQUEST_SMALL)

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Informative Text:
The 14-bit abbreviated **ses.request_length** field is focused on the purpose of the optimized header formats: efficiency at small transfer sizes. The overhead of UET_HDR_REQUEST_STD (relative to an optimized header) for messages larger than 8 KB is nominal.

Table 3-11 - Optimized Header Format Fields

Field	Size	Description	Section Ref
rsvd	2	Table 3-9	
opcode	6	Table 3-9	3.4.7.2
version (ver)	2	Table 3-9	
delivery complete (dc)	1	Table 3-9	3.4.9.3
initiator error (ie)	1	Table 3-9	3.4.6.4.1
relative (rel)	1	Table 3-9	3.4.2.3
rsvd	1		
end of msg (eom)	1	Table 3-9	
start of msg (som)	1	Table 3-9	
rsvd	2		
request_length	14	Table 3-9	
ri_generation	8	Table 3-9	3.4.4.7.3
JobID	24	Table 3-9	0
rsvd	4	Table 3-9	
PIDonFEP	12	Table 3-9	3.4.2.3.2
rsvd	4	Table 3-9	
resource_index	12	Table 3-9	3.4.2.3.3
buffer_offset	64	Table 3-9	

A second use case for an optimized transfer is a single-packet message with matching criteria. Most implementations of this scenario still utilize header data but do not need an offset. A third use case for an optimized transfer is a single-packet RMA operation with immediate data or extended (memory key) addressing. Both use cases carry an abbreviated (14-bit) request length and share the format shown in Figure 3-16. In this format, **ses.som** and **ses.eom** must be set.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

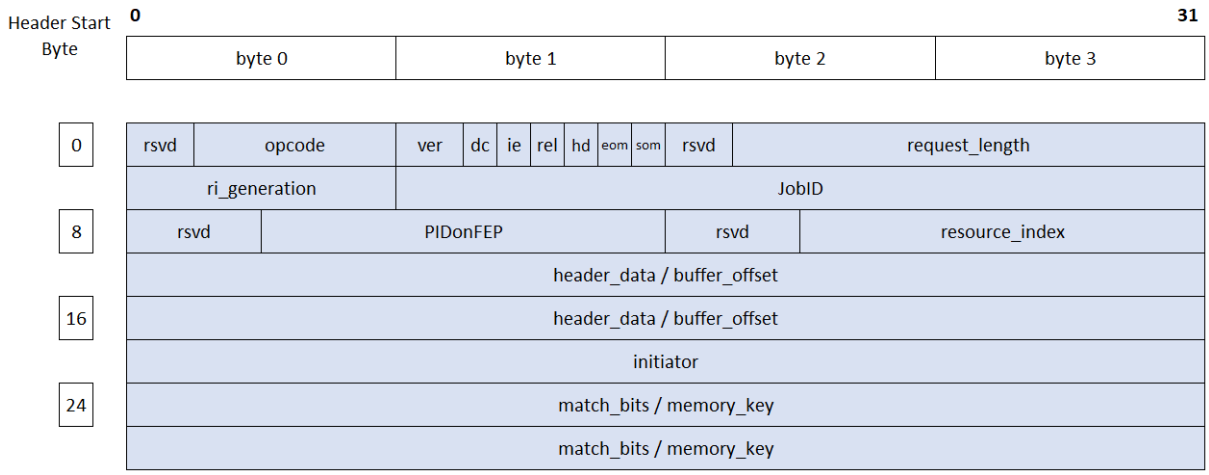


Figure 3-16 - Small Message/Small RMA Format (UET_HDR_REQUEST_MEDIUM)

3.4.3.3 Optimized Header Formats for the Unified Local Profile

Two request header formats are used in conjunction to optimize the wire efficiency of the Unified Local profile transport. These formats target single packet transfers with a payload size up to 16K-1 bytes and have a request length that is sized appropriately. The payload size is limited by the configured Payload MTU, which has a maximum 8 KB. Optimizing beyond the format selected by UET_HDR_REQUEST_SMALL, these two formats support only the direct memory addressing mode; thus, they do not have the **ses.PIDonFEP**, **ses.resource_index**, or **ses.ri_generation** fields. The **ses.request_length** field is shortened to optimize for shorter transfer lengths, and flags that are not relevant to single-packet direct memory addressing are removed (i.e., **ses.rel, ses.som**, and **ses.eom**).

UET_HDR_REQUEST_LCL provides the option to carry implementation specific data per message. This is done by setting **ses.ehl** to one for 4 B, two for 8 B or three for 16 B. The extension header is supported explicitly when **ses.ac** = 0. If **ses.ac** = 1, the **ses.ehl** field is not present. In this case, **ses.ehl** is taken from the previous message in the same packet with **ses.ac = 0**

Multiple optimized semantic headers for the Unified Local profile can be combined into a single PDS packet. The first optimized semantic header in a PDS packet MUST use the format shown in Figure 3-17 and described by Table 3-12. In this format, the address compression bit (**ses.ac**) MUST be set to 0. If only one semantic header is used per PDS packet, then **ses.ac** MUST be set to 0 and **ses.mm** (more messages) MUST be set to 0.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

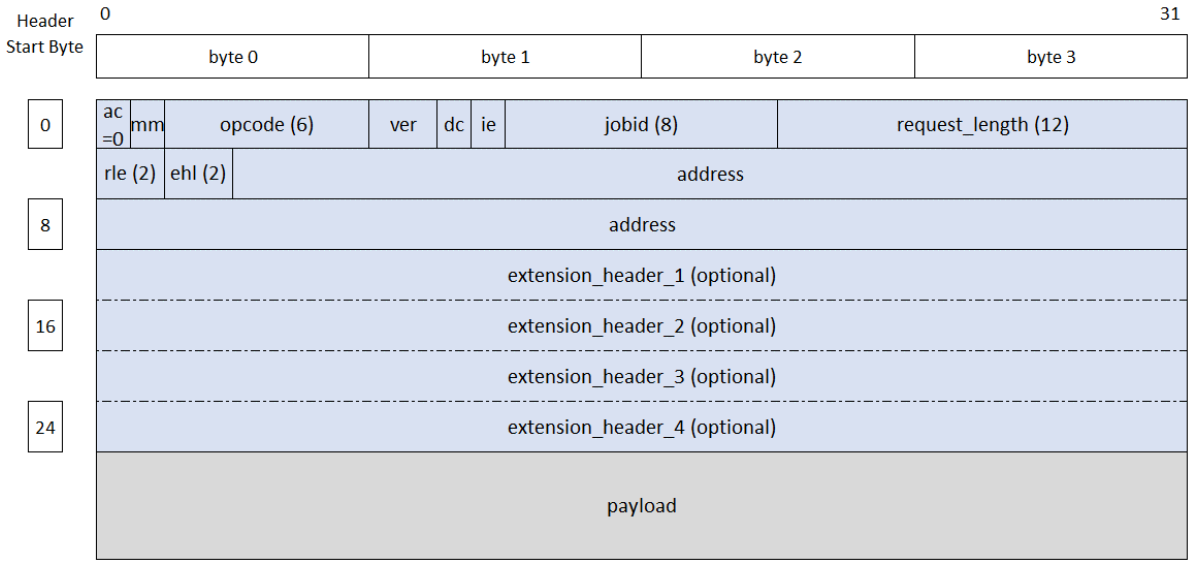


Figure 3-17 - Unified Local Semantic Header with ac=0 (UET_HDR_REQUEST_LCL)

Table 3-12 - Unified Local Semantic Header with ac=0 (UET_HDR_REQUEST_LCL)

Field	Size	Description	Section Ref
Address compression (ac)	1	When set, utilizes a compressed address format that is an offset from the prior operation.	3.4.1.10
More messages (mm)	1	When set, indicates additional semantic headers follow the payload associated with this one.	3.4.1.10
opcode	6	See Table 3-8	
Version (ver)	2	See Table 3-8	
Delivery Complete (dc)	1	See Table 3-8	
Initiator Error (ie)	1	See Table 3-8	
JobID	8	JobID used for direct memory addressing and authorization	3.4.1.3.1
request_length	12	Length of the payload to be transferred (in bytes). 0 is a legal transfer size. Concatenated with **ses.rle** to achieve a maximum size of 16KB-1 bytes but the transfer size is limited by the configured Payload MTU.	
Request length extension (rle)	2	most significant bits of **ses.request_length**	
Extension header length (ehl)	2	Extension header length • 0 = 0 B • 1 = 4 B • 2 = 8 B • 3 = 16 B	

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Field	Size	Description	Section Ref
address	60	Offset into the region identified by the JobID.	

Implementation Note:
As an example of the implementation of **ses.dc**, when **ses.dc** is 1, a target using AXI would need to wait for the AXI completion before generating the SES response.

When additional semantic headers are placed in a single packet, the **ses.mm** bit MUST be set to 1 for all semantic headers in the packet other than the last one. The **ses.mm** bit MUST be set to 0 for the last semantic header in the packet. Figure 3-21 illustrates a packed SES payload with four messages and a corresponding packed SES response. This example uses **ses.ehl** = 0.

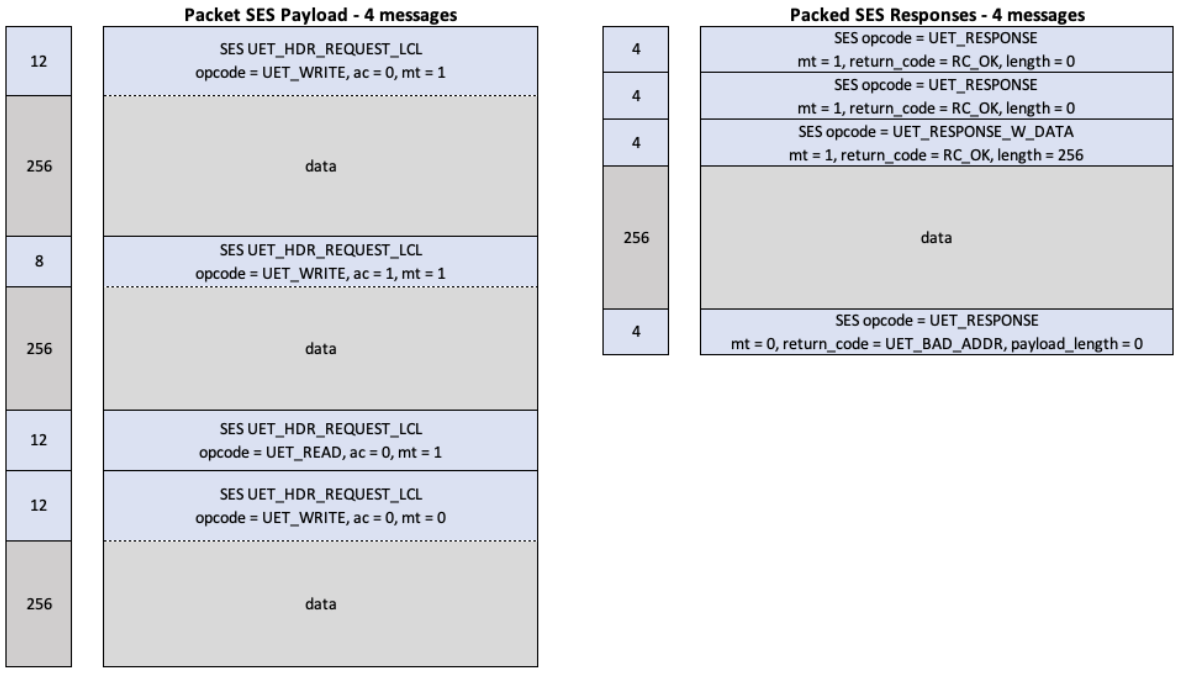


Figure 3-18 - Packed messages using ses.mm=1

When packing messages, if the payload of a message is not a multiple of four bytes, that message is padded out to the next multiple of four bytes. The next SES header in a packed payload always starts on the same 4-byte boundary as the previous SES header. For example, if the **ses.request_length** is 53 bytes, then the next SES header starts 56 bytes after the start of the previous message's payload. The padding bytes are set to zero. When present, the UET CRC directly follows the last message (no padding before the UET CRC).

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

The maximum size of the SES payloads (i.e., the sum of all **ses.request_length** fields) MUST NOT exceed the configured Payload MTU. If the requests include read or fetching atomic messages, the sum of the SES payloads in the responses (i.e., the sum of all **ses.payload_length** in the response packet) MUST NOT exceed the configured Payload MTU. The maximum size of the packet with all network headers and payloads MUST NOT exceed the network configured MTU.

SES aggregates multiple messages into a single PDS payload, setting the **ses.mm** flag and **ses.ac** flag as needed. The multiple messages appear as a single SES payload to the PDS layer.

The SES at the destination FEP generates a corresponding packed response. The responses MUST be packed in the same order as the request messages. Association of response with request is done implicitly using the order. If the Payload MTU of the response is exceeded, SES returns a single response with **ses.opcode** = UET_RESPONSE and **ses.return_code** = RC_UNSUPPORTED_SIZE. This will result in all messages in the associated packed request failing.

If all messages use **next_hdr** = UET_DEFAULT_RESPONSE with **ses.return_code** = RC_OK, a single response may be used. The source FEP maintains the necessary state to associate the responses in a packed response with the appropriate message. If a single UET_DEFAULT_RESPONSE with RC_OK is received, then all messages in the packed payload were successful.

The **ses.ac** bit MUST be set to 0 for the first semantic header in a packet, and MAY be set to 0 or 1 for any subsequent header in the packet. When **ses.ac** is set to 1, the format used is shown in Figure 3-19, and the compressed address field is treated as an offset relative to the address used for the immediately prior semantic header. The presence and size of the extension header is a function of the previous message in the packet with **ses.ac** = 0.

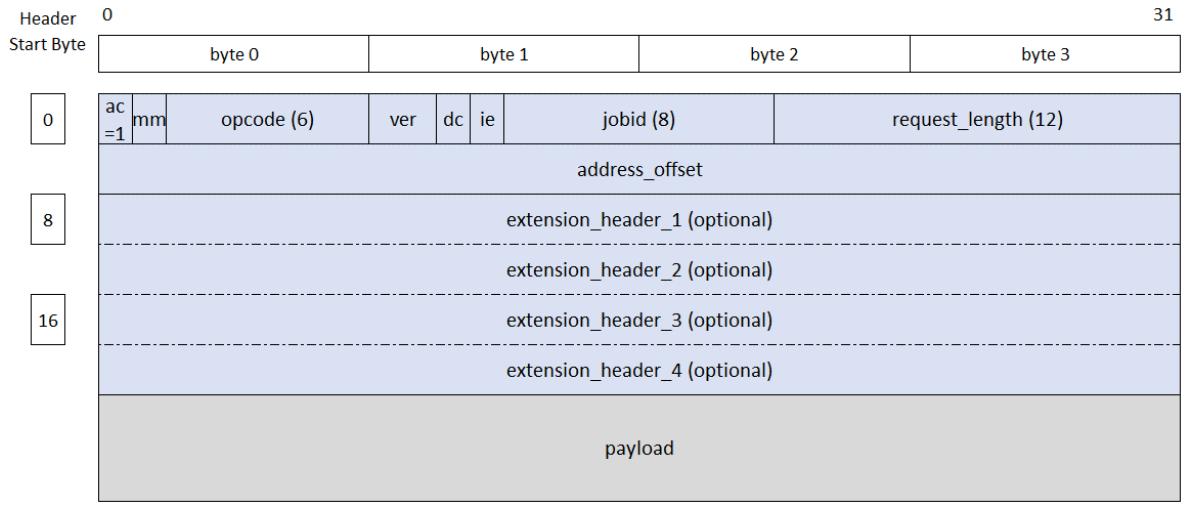


Figure 3-19 - Unified Local Semantic Header with ac=1 (also uses UET_HDR_REQUEST_LCL)

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Table 3-13 - Unified Local Semantic Header with ac=1 (UET_HDR_REQUEST_LCL)

Field	Size	Description	Section Ref
Address compression (ac)	1	See Table 3-12	
More messages (mm)	1	See Table 3-12	
opcode	6	See Table 3-8	
Version (ver)	2	See Table 3-8	
Delivery Complete (dc)	1	See Table 3-8	
Initiator Error (ie)	1	See Table 3-8	
JobID	8	See Table 3-12	
request_length	12	Length of the payload to be transferred (in bytes). 0 is a legal transfer size. Maximum size is 4KB-1 bytes, but limited by the configured Payload MTU	
address_offset	32	Compressed address expressed as a 2s-complement offset relative to the prior operation	

Address compression uses two's complement math to calculate the address offset. "Chaining" is used so each message uses the target address calculated for the previous message. Table 3-14 shows the target address and the address offset in the SES header when **ses.ac** = 1.

Table 3-14 - Target Address and Offset when ses.ac=1

ac	Target address (decimal)	address_offset (decimal)	Target address (hex)	Address_offset (hex)
0	8590996480	-	0x0000_0002_0010_3400	-
1	8590996736	256	0x0000_0002_0010_3500	0x0000_0100
1	8590995968	-768	0x0000_0002_0010_3200	0xFFFF_FD00
1	8623489024	32493056	0x0000_0002_0200_0000	0x01EF_CE00

When UET_HDR_REQUEST_LCL is used for atomic operations, if the extension header is present, it directly follows the 12-byte base header and the atomic header follows the extension header, as illustrated in Figure 3-20.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

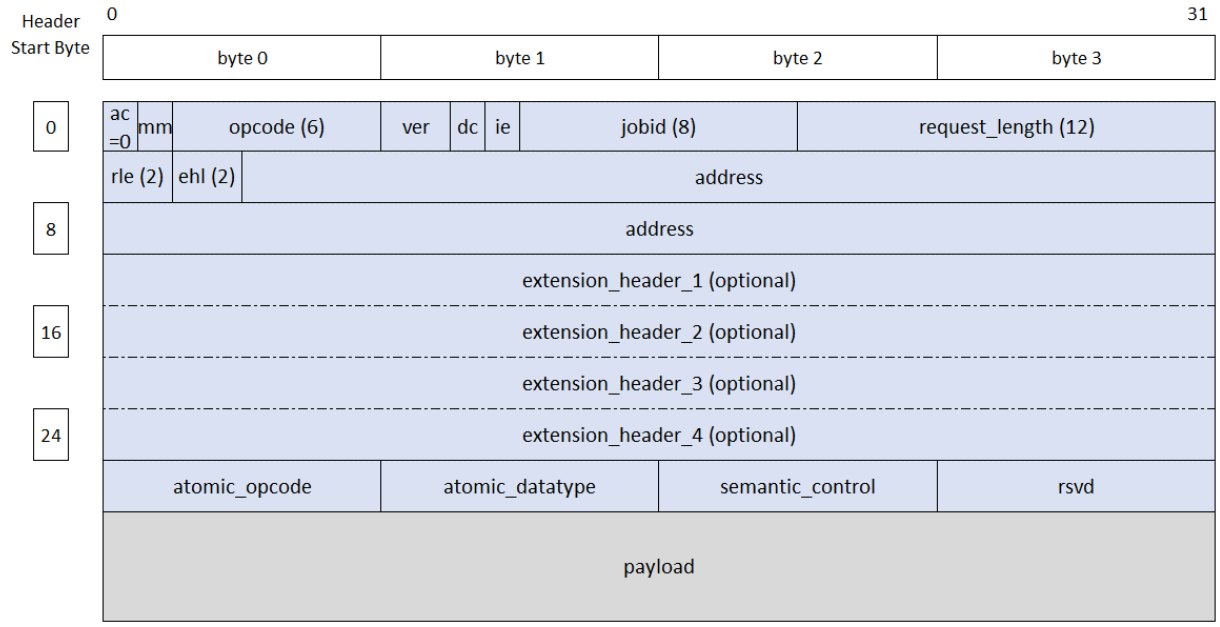


Figure 3-20 - Unified Local Semantic Header with ac=1, ext_hdr > 0, and atomic operation

3.4.3.4 Rendezvous Extension Header Format

The wire protocol includes the option for rendezvous transactions. Rendezvous transactions leverage an extension header shown in Figure 3-21. This extension header includes a 32-bit eager length. The eager length indicates how much message payload is being pushed with the request. The remaining fields correspond exactly to the addressing information needed to issue a read operation. Details of how these fields are used to generate a read are discussed in Section 3.4.4.5.

The rendezvous extension header is used immediately following Figure 3-11. It is used when a rendezvous send or Tsend opcode is used. Rendezvous does not require any additional headers or operation types. A rendezvous extension header should be placed on every packet using a rendezvous opcode (UET_RENDEZVOUS_SEND, UET_RENDEZVOUS_TSEND).

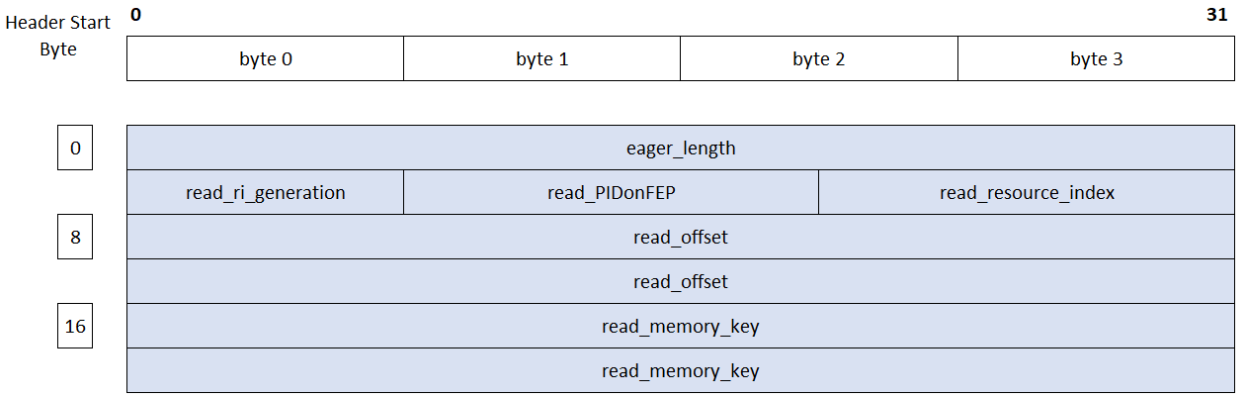


Figure 3-21 - Rendezvous Extension Header Format

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

3.4.3.5 Atomic Operation Extension Header Format

Atomic operations include an atomic header to describe the atomic operation. The memory model for atomic operations is discussed in Section 3.4.9. Atomic headers—shown in Figure 3-22—are carried with every packet of an atomic operation. Atomic headers are used with any of the optimized header formats or the standard header formats. Atomic headers MUST NOT be combined with the rendezvous header format. For non-fetching atomic operations, the number of elements in an atomic operation is determined by the request length of a message (equivalently, the payload length of a packet in the optimized header formats) divided by the size of the atomic datatype. The payload part of messages containing atomic operations should be an integral multiple of the atomic datatype size. In messages where this is not true, an implementation of the target FEP MUST truncate the operation to 0 bytes (i.e., a message with an incorrect length is not performed for any of the bytes in the message).

The atomic operation extension header introduces three new types: an atomic opcode, an atomic datatype, and semantic control. Semantic control (Table 3-29) provides additional information for handling of atomic operations. Payload for all atomic operations follow the atomic header.

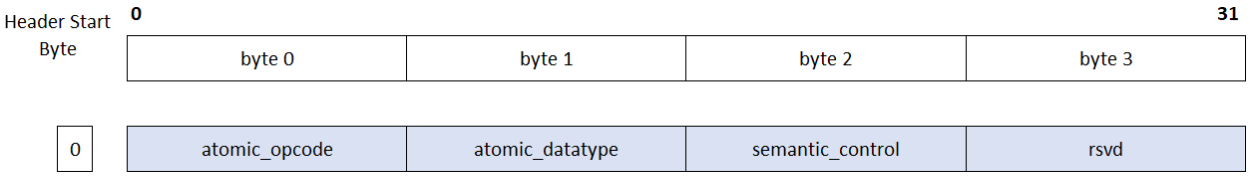


Figure 3-22 - Atomic Operation Extension Header Format

Fetching atomic operations are more limited than general atomic operations. Because fetching atomic operations impose the need to buffer data at the target (i.e., the original data), fetching atomic operations MUST NOT operate on more than one element. That element is of the size of the atomic datatype and follows the atomic header in the packet.

A special type of fetching atomic operation is a dual operand fetching atomic. This exists as a compare-and-swap or swap-under-mask operation. Dual operand fetching atomics use the packet format shown in Figure 3-23, with the compare or mask value first and the swap value after. The largest supported compare and swap (or masked swap) is 16 bytes. The payload length for a compare and swap operation is exactly 32 bytes, and the atomic datatype defines the total size of the operation. Note that, in the optimized header formats, this means that the request length is set to 32 bytes and the actual length of the operation is inferred from the datatype. The compare/mask values and swap values start in the low order bytes of the respective fields.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

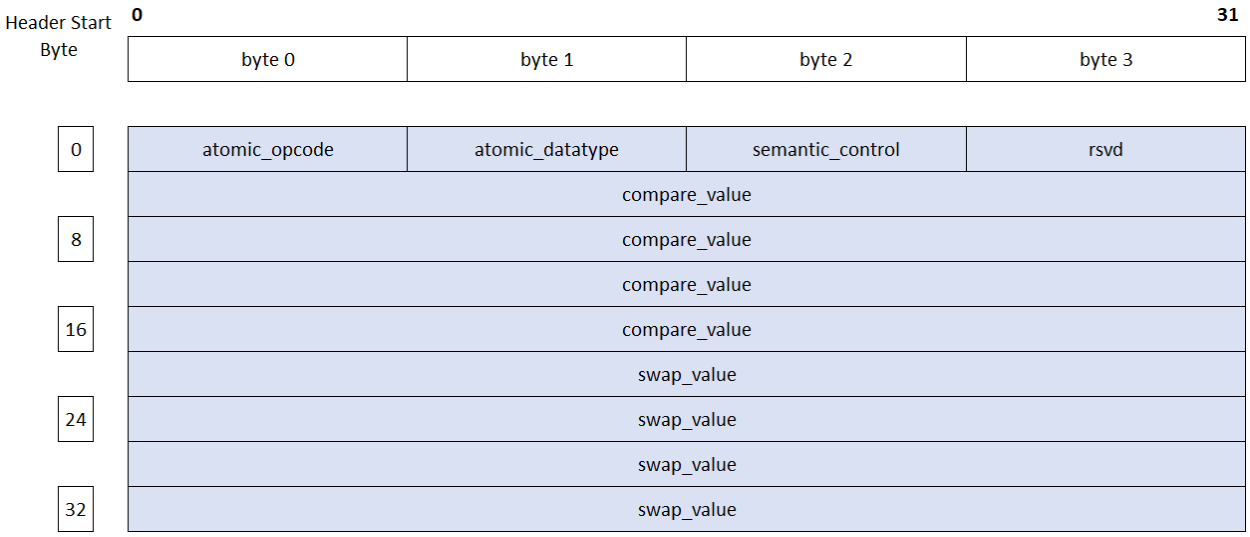


Figure 3-23 - Compare and Swap Operation Atomic Header and Payload Format

3.4.3.6 Synchronized Operation Extension Header Format

Two opcodes are defined for synchronized operations: UET_SYNC_WRITE and UET_SYNC_ATOMIC (see Table 3-23). All synchronized operations utilize the extension header format shown in Table 3-15 and Figure 3-24. This extension header follows the base header type and may be used with any base header type supported by the profile, including UET_HDR_REQUEST_SMALL (Figure 3-15), UET_HDR_REQUEST_MEDIUM (Figure 3-16), and UET_HDR_REQUEST_STD (Figure 3-11).

When UET_SYNC_WRITE is used as a synchronized completing operation, it MUST set **ses.hd**. When UET_SYNC_ATOMIC is used, the maximum **ses.request_length** supported is 8 bytes, and the message MUST NOT set **ses.hd**. The UET_SYNC_ATOMIC opcode MAY be used with any data type that is supported by the device that is 8 bytes or less. The UET_SYNC_ATOMIC opcode MAY be used with any atomic operation opcode supported by the device that is non-fetching. UET_AMO_READ, UET_AMO_MSWAP, and all variations of UET_AMO_CSWAP are prohibited.

The extension header contains two fields: **ses.sync_group** and **ses.sync_count**. The **ses.sync_group** field is the group number that the message is associated with. At the target FEP, the **ses.sync_group** field is scoped to the initiator FEP. That is, the tuple {**ip.src_addr, ses.sync_group**} is used to lookup the context for the synchronized group. At the initiator, the scope MAY be global to the FEP or scoped per target FEP. The **ses.sync_count** field is valid only for completing operations encoding the count of the total number of operations in the synchronized group, including the synchronized completing operation. The **ses.sync_count** MUST be set to zero for all sync write operations.

Table 3-15 - Synchronized Operation Extension Header Format

Field Name	Size (in bits)	Description
sync_group	16	The synchronized group number this message is associated with.
sync_count	16	This has two encodings: Non-completing operations (synchronized write): MUST be 0.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Field Name	Size (in bits)	Description
		Synchronized completing operations: The total number of messages in the synchronized group.

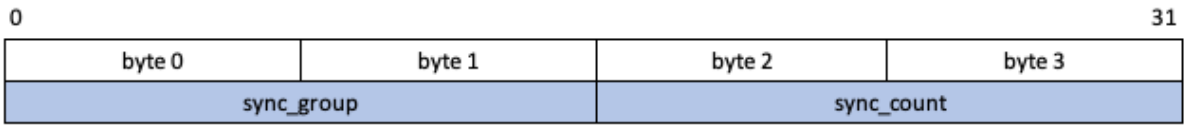


Figure 3-24 - Synchronized Operation Extension Header Format

3.4.3.7 Semantic Response Header Formats for Scale-out Networks

Standard semantic responses are carried in PDS acknowledgements and use the format shown in Figure 3-25. Semantic response headers are used for semantic acknowledgements (e.g., a semantic response to a Send). The response header includes a field to indicate the modified length. The modified length in a semantic response indicates the amount of payload that will be delivered as part of processing the message. Some use cases require message truncation at the target (e.g., the MPI unexpected message sequence). In other cases, the modified length is used as part of the rendezvous sequence as described in Section 3.4.4.5.

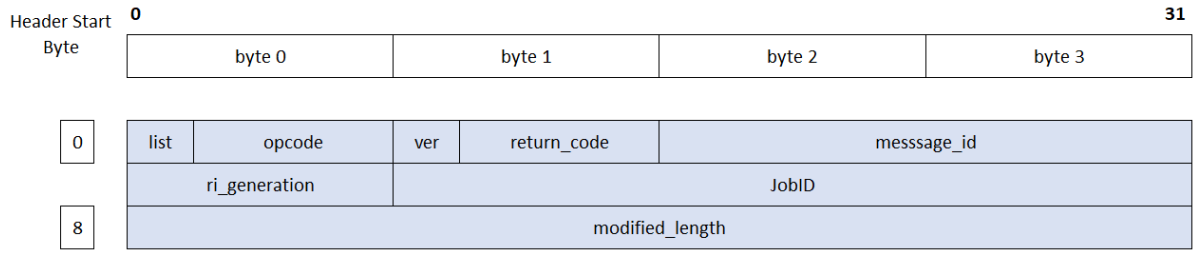


Figure 3-25 - Semantic Response Header Format (UET_HDR_RESPONSE)

Table 3-16 - Response Header Fields

Field	Size	Description	Section Ref
list	2	Indicates if the payload was delivered to the expected or unexpected list.	3.4.7.3
opcode	6	Indicates type of response (e.g., default response, response with payload, etc.). Table 3-24.	3.4.7.2
version (ver)	2	Semantic protocol version: Set to 0 in the initial version.	
return_code	6	Indicates success conditions and some types of error conditions detected at the semantic sublayer.	3.4.7.3
message_id	16	Message ID of the original request	3.4.2.14
ri_generation	8	Contains the new index generation on a generation mismatch response.	

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Field	Size	Description	Section Ref
JobID	24	JobID of the original request	3.4.2.3.1, 3.4.2.4.1
modified_length	32	Indicates the number of bytes of the target buffer that will be modified by this transaction. For example, some message may be truncated because no buffer is available.	

Two variations on the semantic response with data are used. The first (Figure 3-26) mimics the basic semantic response and adds a payload length—to indicate the number of bytes in the packet—and a message offset—to indicate where within the message this packet falls. The message ID is used to identify the original read request. At the semantic level, all operations—including read operations—are packetized based on the Payload MTU. A multi-packet *fi_read()*, for example, will use the same message ID in each packet and use that to issue a single completion at the initiator. This format MUST be used for responses with data to requests that include a request length larger than the Payload MTU. This format MUST NOT be used to respond to optimized request headers (Figure 3-15), because various fields are not available in that request to generate this response.

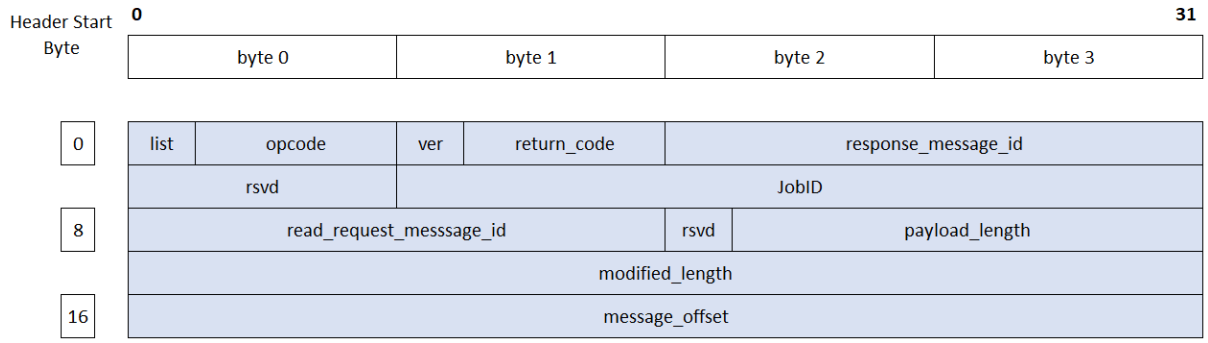


Figure 3-26 - Semantic Response with Data Header Format (UET_HDR_RESPONSE_DATA)

Table 3-17 - Response with Data Header Fields

Field	Size	Description	Section Ref
list	2	Indicates if the payload was delivered to the expected or unexpected list.	3.4.7.3
opcode	6	Indicates type of response (e.g., semantic ACK, response with payload, etc.).	3.4.7.2
version (ver)	2	Semantic protocol version: Set to 0 in the initial version.	
return_code	6	Indicates success conditions and some types of error conditions detected at the semantic sublayer.	3.4.7.3
response_message_id	16	Message ID of the response.	3.4.2.14
rsvd	8	Reserved	
JobID	24	JobID of the original request.	3.4.2.3.1, 3.4.2.4.1

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Field	Size	Description	Section Ref
read_request_message_id	16	Message ID used in the original read request (or of the original fetching atomic operation request).	3.4.2.14
rsvd	2	Reserved	
payload_length	14	Length of the payload in this specific packet for a response with data. Note: These bits are reserved in a semantic response without payload.	
modified_length	32	Indicates the total number of bytes of the initiator buffer that will be modified by this transaction. For example, a message may be truncated because no buffer is available or the buffer it was targeting is too small.	
message_offset	32	Indicates the relative position in the message that this payload corresponds to.	

The second response with data header format (Figure 3-27) is used for responses with data where the original operation consisted of an optimized request header (Figure 3-15) with a request length that described a total payload that could be carried in a single Payload MTU. Here, the payload length is also the modified length and a message ID is not needed, because PDS can associate the response with the original request using the packet sequence number (PSN) echoed in this packet format. This MAY be used for small reads and for fetching atomic operations. The payload length in this format is placed where the message ID is carried in the full format, and the modified length and message offset are omitted.

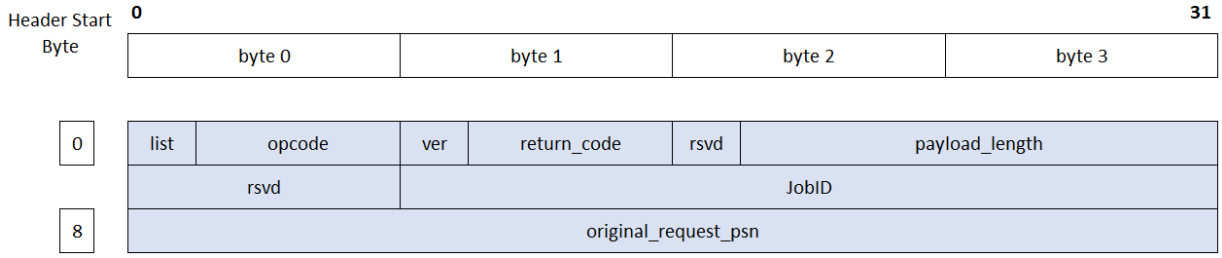


Figure 3-27 - Optimized Response with Data Header Format (UET_HDR_REQUEST_DATA_SMALL)

Table 3-18 - Optimized Response with Data Header Fields

Field	Size	Description	Section Ref
list	2	Indicates if the payload was delivered to the expected or unexpected list.	3.4.7.3
opcode	6	Indicates type of response (e.g., semantic ACK, response with payload, etc.).	3.4.7.2
version (ver)	2	Semantic protocol version: Set to 0 in the initial version.	

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Field	Size	Description	Section Ref
return_code	6	Indicates success conditions and some types of error conditions detected at the semantic sublayer.	3.4.7.3
rsvd	2	Reserved	
payload_length	14	Length of the payload for a response with data. This field serves at the modified length as well. For two-operand atomics, this value contains the length of the datatype in the typical case. Note: These bits are reserved in a semantic response without payload.	
rsvd	8	Reserved	
JobID	24	JobID of the original request.	0 3.4.2.4.1
original_request_psn	32	The PSN of the original request (either fetching atomic or read) that yielded this return data.	3.4.2.15

Informative Text:
The original request PSN is provided for the case where the returned data is larger than the *Max_ACK_Data_Size* in PDS. In these cases, the data is returned on the response direction channel using a separate sequence number space. As such, the original request PSN is needed to reconcile the optimized response header with the original request.

3.4.3.8 Semantic Response Header Formats for the Unified Local Profile

In addition to optimizing the request header formats for the Unified Local profile, an optimized header is provided for responses as shown in Figure 3-28 and Table 3-19. If any semantic request that is packed into a single PDS packet has a semantic response that is not UET_DEFAULT_RESPONSE, then each semantic request from that packet MUST have a corresponding semantic response in the PDS acknowledgement. See Section 3.4.4.4 for additional details.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

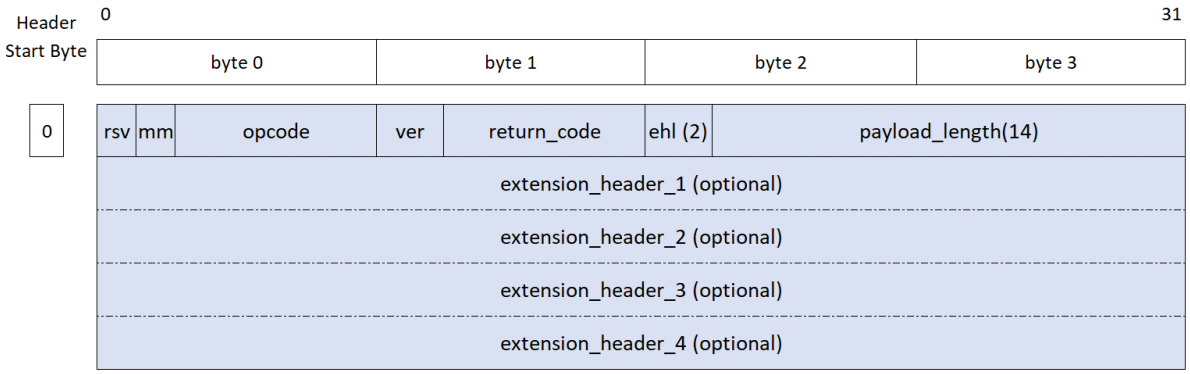


Figure 3-28 - Unified Local Semantic Header Response Header (UET_HDR_RESPONSE_DATA_LCL)

Table 3-19 - Unified Local Semantic Header Response Header (UET_HDR_RESPONSE_DATA_LCL)

Field	Size	Description	Section Ref
rsvd	1	Reserved. MUST be 0	
More messages (mm)	1	See Table 3-12	
opcode	6	See Table 3-11	
Version (ver)	2	See Table 3-11	
return_code	6	See Table 3-11	
elh	2	See Table 3-11	
payload_length	14	See Table 3-12	

3.4.3.9 Header Parsing Guide

Table 3-20 provides a summary of how to parse the semantic header. The leftmost column contains the **pds.next_hdr** enumeration. For each header enumeration, the middle column enumerates the opcodes or opcode types that are used with the next header definition. The rightmost column cross-references to the relevant header formats that are used for that opcode or opcode type. In many cases, the header consists of a base header and extension header. As shown, a parser can look at the **pds.next_hdr** field to determine the base header structure and the **ses.opcode** field to determine whether an extension header is present and its type. The base header used for deferrable send and ready-to-restart operations has the same size and overall structure as the standard request header.

Table 3-20 - Parsing Guide

PDS next_hdr	opcode type	Format
UET_HDR_REQUEST_SMALL	Non-Atomic Opcodes	Figure 3-15
	Atomic Opcodes	Figure 3-15 + Figure 3-22
	Two Op Atomics	Figure 3-15 + Figure 3-23
	Sync Group Writes	Figure 3-15 + Figure 3-24
	Sync Group Atomics	Figure 3-15 + Figure 3-24 + Figure 3-22

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

PDS next_hdr	opcode type	Format
UET_HDR_REQUEST_MEDIUM	Non-Atomic Opcodes	Figure 3-16
	Atomic Opcodes	Figure 3-16 + Figure 3-22
	Two Op Atomics	Figure 3-16 + Figure 3-23
	Sync Group Writes	Figure 3-11 + Figure 3-24
	Sync Group Atomics	Figure 3-11 + Figure 3-24 + Figure 3-22
UET_HDR_REQUEST_STD	Non-Atomic Opcodes	Figure 3-11
	Atomic Opcodes	Figure 3-11 + Figure 3-22
	Two Op Atomics	Figure 3-11 + Figure 3-23
	Deferrable Send	Figure 3-13
	Ready to Restart	Figure 3-14
	Rendezvous Opcodes	Figure 3-11 + Figure 3-21
	Sync Group Writes	Figure 3-15 + Figure 3-24
	Sync Group Atomics	Figure 3-15 + Figure 3-24 + Figure 3-22
UET_HDR_REQUEST_LCL with ac=0	Non-Atomic Opcodes	Figure 3-17
	Atomic Opcodes	Figure 3-17 + Figure 3-22
	Two Op Atomics	Figure 3-17 + Figure 3-23
UET_HDR_REQUEST_LCL with ac=1	Non-Atomic Opcodes	Figure 3-19
	Atomic Opcodes	Figure 3-17 + Figure 3-22
	Two Op Atomics	Figure 3-19 + Figure 3-23
UET_HDR_RESPONSE	UET_RESPONSE UET_DEFAULT_RESPONSE UET_NO_RESPONSE	Figure 3-25
UET_HDR_RESPONSE_DATA	UET_RESPONSE_W_DATA	Figure 3-26
UET_HDR_RESPONSE_DATA_SMALL	UET_RESPONSE_W_DATA	Figure 3-27
UET_HDR_RESPONSE_DATA_LCL	UET_DEFAULT_RESPONSE UET_RESPONSE_W_DATA	Figure 3-28

Table 3-21 enumerates the legal combination of **pds.next_hdr** fields and **ses.opcode** fields. It also highlights some limitations of the formats that may not be obvious.

Table 3-21 - Header Formats and Legal Opcodes

pds.next_hdr	ses.opcode Allowed	Limitations
UET_HDR_REQUEST_SMALL	UET_NO_OP	Payload size MUST be 0. Single-packet messages only.
	UET_WRITE	Payload size MUST be less than or equal to one Payload MTU.
	UET_READ	
	UET_ATOMIC	

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

pds.next_hdr	ses.opcode Allowed	Limitations
		MUST use UET_HDR_RESPONSE_DATA_SMALL for response with data.
	UET_FETCHING_ATOMIC	
	UET_MSG_ERROR	
	UET_SYNC_ATOMIC	
	UET_SYNC_WRITE	
UET_HDR_REQUEST_MEDIUM	UET_NO_OP	Payload size MUST be 0. Single-packet messages only.
	UET_WRITE	Payload size MUST be less than or equal to one Payload MTU.
	UET_READ	
	UET_ATOMIC	
	UET_FETCHING_ATOMIC	
	UET_SEND	Payload size MUST be less than or equal to one Payload MTU.
	UET_TAGGED_SEND	
	UET_DATAGRAM_SEND	
	UET_TSEND_ATOMIC	Cannot use offset into target buffer.
	UET_TSEND_FETCH_ATOMIC	
	UET_MSG_ERROR	
	UET_SYNC_ATOMIC	
	UET_SYNC_WRITE	
UET_HDR_REQUEST_STD	UET_NO_OP	Payload size MUST be 0. Single-packet messages only.
	All Other Request Opcodes	
	UET_MSG_ERROR	
UET_HDR_REQUEST_LCL	UET_NO_OP	Payload size MUST be 0.
	UET_WRITE	Payload size MUST be less than or equal to 16KB-1 for **ses.ac**=0 and 4K-1 for **ses.ac**=1 but limited by the configured Payload MTU.
	UET_READ	
	UET_ATOMIC	
	UET_FETCHING_ATOMIC	
UET_HDR_RESPONSE	UET_RESPONSE	
	UET_DEFAULT_RESPONSE	
	UET_NO_RESPONSE	
UET_HDR_RESPONSE_DATA	UET_RESPONSE_W_DATA	
UET_HDR_RESPONSE_DATA_SMALL	UET_RESPONSE_W_DATA	Cannot be used for responses to messages larger than one Payload MTU
UET_HDR_RESPONSE_DATA_LCL	UET_DEFAULT_RESPONSE	Must be used in response to a UET_HDR_REQUEST_LCL. Payload size must be less than or equal to Payload MTU.
	UET_RESPONSE_W_DATA	

3.4.4 Semantic Processing

This section contains the normative requirements for how semantic headers are processed. Semantic processing is expected to be implemented by a combination of hardware and software. The division between hardware and software is an implementation decision.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

The semantic processing definition contained here does not override the profile definition or any discussion of which features are optional elsewhere in this document. Instead, the semantic processing defines how operations are performed when they are supported.

3.4.4.1 Buffer Selection

For packets using relative addressing, the tuple {JobID, PIDonFEP, Resource Index} MUST identify a unique set of one or more buffers. Buffers associated with the Resource Index MUST be configurable to support either use-once or persistent (i.e., not use-once) semantics. This configuration is associated with the individual buffers. Packets using absolute addressing have a similar requirement: the pair PIDonFEP and Resource Index MUST identify a unique set of one or more buffers. Implementations of direct memory addressing select a region based exclusively on the JobID.

3.4.4.1.1 Send/Receive Operation

The requirements in this section apply only to *fi_send()*/*fi_recv()* in cases where tag matching is not used. Where the behavior varies based on ordering, that is noted in the requirement.

Messages arriving on a RUD PDC MAY consume receive buffers in any order. For implementations that support ordering, messages arriving on a ROD PDC MUST consume the first receive buffer associated with the receive queue of the Resource Index that is addressed. All packets associated with one message MUST be delivered to the same receive buffer. This is accomplished using the message ID in the packet. If no receive buffer is available on a given index when a message arrives, the implementation MUST use an unexpected message handling procedure described in Section 3.4.4.6.1.

3.4.4.1.2 RMA Operation

Some profiles support the UET_HDR_REQUEST_SMALL format for RMA operations (Figure 3-15). In this format, an RMA opcode used with a {JobID, PIDonFEP, Resource Index} tuple MUST map to exactly one buffer. Implementations indicate through the provider when this format can be used following the guidelines developed for the libfabric mapping (see Section 2.2.5.3.4.1).

An implementation MUST support the UET_HDR_REQUEST_STD format for RMA operations. In this format, an RMA opcode used with a {JobID, PIDonFEP, Resource Index, Memory Key} tuple MUST map to exactly one buffer.

The libfabric mapping specification defines a memory key format that enables interoperability. Much of the detail of how the memory key resolved to an actual memory region is implementation-specific. As an example, the memory key MAY use some of the bits as the memory region lookup and populate the remaining bits with a random number that is checked before the memory region is accessed.

Informative Text:
The standard way in which memory keys are utilized is for an application to create a memory key and then share that memory key with the peer that is going to use it. In this way, a memory key is an opaque token for selecting a memory region. The libfabric mapping spec includes a minimal set of rules for the construction of the overall memory key that facilitates interoperability while leaving extensive flexibility for implementations in terms of how a memory key is constructed and used.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

3.4.4.1.3 Matching Operation

The requirements in this section apply only to implementations that support matching. Where the behavior varies based on ordering, that is noted in the requirement. Where the behavior varies based on wildcarding, that is noted in the requirements.

Messages arriving on a RUD PDC MAY attempt matching in any order, or concurrently across all posted buffers. This is independent of whether the target uses exact matching or wildcard matching.

If an implementation supports wildcard matching and a message arriving on a RUD PDC matches more than one buffer that is posted, the implementation SHOULD choose the oldest buffer posted; however, the implementation MAY choose any matching buffer.

For implementations that only support exact matching, arriving messages MAY attempt matching in any order—regardless of whether a ROD or RUD PDC was used. The implementation MUST select a matching buffer if a matching buffer is available. If more than one match is found, the implementation MAY select any matching buffer. This specification places no bias on which buffer should be selected if multiple buffers match. If no match is found, the unexpected message procedures are used (3.4.4.6.1).

Informative Text:
The exact matching semantic was created as a way to enable simplified hardware implementations (e.g., a CAM) to implement matching. While the requirement was relaxed to allow duplicate matches, the desire was to preserve hardware simplicity. Achieving historical definitions of match ordering with multiple matches in a CAM is challenging; thus, no preference in match order is defined for cases where exact matching would return more than one result. Choosing the oldest buffer is marked as a SHOULD, because it typically simplifies the management of software resources if the buffers are consumed in order.

For implementations that support message ordering and wildcard matching, messages arriving on a ROD PDC:

1. MUST attempt[7] to match buffers in the order the buffers were provided by the application through the libfabric API for a given {JobID, Target PIDonFEP, Resource Index} tuple.
2. MUST process headers from a given {JobID, Initiator} tuple in the order that they were sent from the initiating PIDonFEP for a given {JobID, Target PIDonFEP, Resource Index} tuple on a given target FEP.

Implementations MAY perform optimizations that preserve the appearance of this ordering. Implementations MAY implement stronger ordering than required at a target. For example, all messages MAY be ordered when attempting matching at a given FEP.

[7] "Attempt," because matching can fail to find a corresponding buffer.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

3.4.4.1.4 Accesses with Direct Memory Addressing

RMA operations in header formats designed for the Unified Local profile carry only a JobID and address. The JobID MUST select the addressing context used to interpret the address conveyed over the wire. The address MUST begin at the base associated with the addressing context. Implementations MAY support only a single JobID, in which case the addressing context corresponds to the entire address space of the local node.

3.4.4.2 Buffer Authorization

Buffer authorization is a function of the target FEP implementation, including the libfabric provider implementation associated with the target FEP. Implementations MUST validate the JobID provided in the packet to determine whether it is allowed to access the buffer found through buffer selection (3.4.4.1). Two options for buffer authorization based on JobID MUST be supported. First, implementations MUST allow an option for a buffer to be exposed for exactly one JobID where the JobID is authorized as specified in 3.4.2.4.1. Second, implementations MUST allow an option for a buffer to be exposed for "any" JobID such that the JobID check for that buffer is effectively ignored.

A buffer that is addressed using matching criteria effectively limits the access of that buffer using an initiator check.

3.4.4.3 Scale-out Response Generation

Each message MUST receive at least one semantic response. For many messages, this may take the form of a UET_DEFAULT_RESPONSE using the format in Figure 3-25. The semantic response for the last packet received MUST NOT be returned until all prior packets have completed semantic processing.

Implementation Note:
The requirement to wait until all packets have completed semantic processing before returning the semantic response for the last packet received is known to require some implementation effort; however, the congestion management sublayer (CMS) requires early acknowledgements on most packets. By guaranteeing that any semantic information will be returned no later than the acknowledgement for the last packet to arrive for the message, it is possible to allow early acknowledgements of other packets in the message.

Informative Text:
There was a trade-off between number of packets on the wire and the need to wait until all semantic processing had completed before returning the semantic response for the last packet. The congestion management sublayer needs acknowledgements to be generated quickly in many cases. Other schemes were considered to meet this goal (e.g., a fast ACK scheme at the packet delivery sublayer), but those schemes led to two acknowledgements per packet in typical scenarios.

A UET_DEFAULT_RESPONSE contains a **ses.JobID**, a **ses.message_id**, a **ses.modified_length** field that is equal to the request length, a **ses.list** field set to UET_EXPECTED, a **ses.ri_generation** field set to the generation of the request, and a **ses.return_code** set to RC_OK. Usage of UET_DEFAULT_RESPONSE

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

allows PDS to coalesce acknowledgements. Semantic processing MUST complete before setting the opcode to UET_DEFAULT_RESPONSE. A response opcode of UET_DEFAULT_RESPONSE does not need to be marked for guaranteed delivery. A message that has received responses for all packets at the initiator is presumed to be a UET_DEFAULT_RESPONSE unless another response encoding has been sent and marked for guaranteed delivery at the target. Any response that cannot use the UET_DEFAULT_RESPONSE opcode (because it does not meet the requirements above) MUST be marked for guaranteed delivery. Marking a response for guaranteed delivery prevents acknowledgement coalescing at the packet delivery sublayer, because the packet delivery sublayer cannot communicate the unique content in a guaranteed delivery response in a coalesced acknowledgement.

Informative Text:
One semantic response per message is a MUST because the transport does not have a way to signal over the wire that a semantic response is not needed.

Implementation Note:
Acknowledgement coalescing at the packet delivery sublayer is specifically enabled by the UET_DEFAULT_RESPONSE opcode and the UET_NO_RESPONSE opcode (see below). Acknowledgement coalescing leads to three quirky characteristics that implementors should be aware of. First, two UET_DEFAULT_RESPONSE semantic responses can be coalesced into a single acknowledgement. There is no guarantee that the two UET_DEFAULT_RESPONSE messages will have the same **ses.JobID** or **ses.message_id**. They are certainly not likely to have the same modified length. Nonetheless, the two can be coalesced, which means that the initiator has no mechanism to retrieve that required information other than through lookup of the PSN. Second, during coalescing, a UET_NO_RESPONSE can be coalesced with a UET_DEFAULT_RESPONSE. This effectively promotes the signal received at the initiator from UET_NO_RESPONSE to UET_DEFAULT_RESPOSE. This semantic promotion is one of the reasons that the semantic response to a deferrable send is required to wait for semantic processing to complete. Third, a UET_NO_RESPONSE can be promoted to UET_DEFAULT_RESPONSE when responding to a retransmit. This is an extreme corner case.

An implementation MAY provide multiple semantic responses per message; however, three rules apply in this case. First, an additional semantic response MUST either utilize the same opcode and return code as the first semantic response or deliver an error response. Second, the first packet with a semantic error (non RC_OK) MUST be marked for guaranteed delivery. Error responses MAY be replicated for many packets; however, subsequent errors MUST NOT be marked for guaranteed delivery. Finally, more than one semantic response MUST NOT be provided per packet.

Implementation Note:
A careful reader may note that the above rules allow for up to two different responses per message to be generated and marked for guaranteed delivery in the case of a multi-packet message — a first response that indicates RC_OK with some other condition that requires guaranteed delivery and a

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

second response that indicates an error that is marked for guaranteed delivery; however, no more than one response per packet may be marked for guaranteed delivery.

Except in the case of deferrable send, an implementation MAY acknowledge a packet before semantic processing. In this case, UET_NO_RESPONSE MUST be used as the semantic opcode for any packet acknowledged before semantic processing. The PDC at the initiator may use this to determine that a packet was delivered successfully, but that semantic processing had not concluded. This response uses the same format as UET_DEFAULT_RESPONSE and contains a **ses.JobID** and **ses.message_id**. Deferrable sends MUST include a semantic response with the response to the first packet received for that message.

Informative Text:
The semantic response is the mechanism by which deferrable sends are deferred. If a deferrable send is going to be deferred, then the implementation cannot delay the semantic response which would lead to deferring the send.

If delivery complete (**ses.dc**) is set for the message, the response to the last packet that is received for a message MUST NOT be generated until the completion semantics have been met—that is, until the entire message is globally observable (3.4.9.3). The last packet received may not be the last packet of a message (e.g., for the RUD protocol).

Responses MUST include the **ses.message_id** that was included in the request **ses.message_id** field or **ses.read_request_message_id** field (for responses with data). For request formats that do not contain a message ID, a response that has a message ID field MUST have the value 0 in the **ses.message_id** field or **ses.read_request_message_id** field (for responses with data).

The **ses.JobID** in the response MUST be the **ses.JobID** of the request—if one is present. Certain PDS control packets do not necessarily provide a JobID to use in the response.

For a response with data that contains a message offset, the **ses.message_offset** field MUST be 0 when responding to the start-of-message request and MUST be the **ses.message_offset** carried in the header data for other responses.

If a UET_DEFAULT_RESPONSE opcode is not appropriate, the response opcode MUST be UET_RESPONSE when no data is returned or UET_RESPONSE_W_DATA when data is returned (Table 3-23). UET_RESPONSE and UET_RESPONSE_W_DATA packets MUST be marked for guaranteed delivery.

The **ses.return_code** field MUST be RC_OK (Table 3-25) for successful operations. Errors from all previous packets in the message MUST be aggregated into the **ses.return_code** field using a first error model for error precedence (3.4.6.1). Here, “first error” is the first error for this message that SES encountered. Due to out-of-order packet handling, the first error may occur out of packet order.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

In normal message processing, the modified length is typically the requested length; however, target implementations are allowed to truncate the message for a variety of reasons—including, but not limited to, managing unexpected messages and server use cases that bound the size of the message that is accepted. The **ses.modified_length** field MUST indicate the entire length of the accepted operation—except in the case of deferrable send responses.

A deferrable send response indicates a **ses.modified_length** of 0 even if it buffers some of the operation in an eager buffer. The ready-to-restart message indicates how much of the message to resend.

The **ses.payload_length** field MUST encode the length of the payload (in bytes) returned in this packet.

3.4.4.4 Unified Local Profile Response Generation

Semantic responses to semantic requests that used the UET_HDR_REQUEST_LCL format use the UET_HDR_RESPONSE_DATA_LCL format. All responses to messages packed into a single PDS packet are returned in a single PDS acknowledgment. The acknowledgement to PDS can be a simple UET_DEFAULT_RESPONSE when a write transaction completes without encountering processing errors. In this case, a distinct acknowledgement packet need not be generated immediately, as a cumulative acknowledgement is sufficient.

When a PDS acknowledgment packet is generated, it MUST contain one UET_HDR_RESPONSE_DATA_LCL for each UET_HDR_REQUEST_LCL packed into the corresponding PDS request packet. Any UET_HDR_REQUEST_LCL with a UET_READ or UET_FETCHING_ATOMIC opcode MUST generate a response using the UET_RESPONSE_W_DATA opcode. If any semantic request packed into a single PDS request packet requires a UET_RESPONSE_W_DATA opcode with a non-RC_OK return code, a PDS acknowledgement packet MUST be generated, and it MUST contain one semantic response for each semantic request that was packed into that PDS request packet.

All semantic responses using ROD or RODL MUST be generated in order. A semantic response MUST NOT be generated until all required local processing has completed. Required processing includes local error detection, but excludes waiting for a completion unless **ses.dc** is asserted. For any semantic request with **ses.dc** set, the corresponding semantic response MUST NOT be generated until the completion has been received.

Implementation Note:
Examples of local completion include a PCIExpress UIO completion or an AXI completion.

3.4.4.5 Rendezvous Processing

Informative Text:
The rendezvous transaction is intended to be implemented by a combination of hardware and software that resides within the libfabric provider. To provide interoperable operation, the entire rendezvous transaction must be compatible.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

A rendezvous send is provided an eager length from software. The portion of the eager length that is sent before receiving the semantic response SHOULD NOT exceed the current size of the PDC's congestion management window at the time the eager portion is attempted to be sent. This is true even if the congestion control window is increased while the eager portion of the message is in flight. A rendezvous send operation MUST NOT send more data than the eager length before completing the rendezvous send. Stated differently, the eager length is the full length of the message as it is transferred on the wire for a rendezvous send or rendezvous tagged send. On receiving a semantic response, the rendezvous send implementation SHOULD adjust the eager transfer for this specific message to be no longer than the modified length provided in the response. If the portion of the eager length already transferred equals or exceeds the modified length, the implementation MUST send at least one more packet to mark this portion of the transfer with the **ses.eom** bit set. This additional packet MAY be 0 bytes, Payload MTU bytes, or the remaining fragment of the eager portion. Implementations MAY treat packets in the process of being constructed as if they had already been sent. The last packet of the eager portion of the message MUST have **ses.eom** set.

The text above covers the sequence for data delivery when the message is expected. In the unexpected case, the above scenario would typically have a modified length smaller than the eager length (often 0). In addition, the unexpected message solution MUST use one of the mechanisms described in Section 3.4.4.6.1 to complete the data transfer. When using a RUD PDC, an implementation of rendezvous SHOULD implement a buffered header solution (3.4.4.7.2.1) and MUST implement a back-off and retry (3.4.4.7.1.1) for when the header space is exhausted. When using a ROD PDC, an implementation of rendezvous MUST implement a buffered header solution, and MUST implement a resource exhaustion solution (3.4.4.7.3).

The rendezvous send message contains sufficient information to retrieve the remainder of the payload using a read operation. The contents of the rendezvous extension header are provided at the initiator by software or hardware. The rendezvous extension header fields for **ses.PIDonFEP**, **ses.resource_index**, and **ses.match_bits** MUST be used “as is” for the construction of the read operation that completes the rendezvous transfer. The **ses.read_offset** field in the rendezvous extension header MUST be an offset that can be used to retrieve the entire message. The target of the operation MAY buffer up to the eager length of the payload if it implements a buffered unexpected message solution (3.4.4.7.2.2). The read operation uses either the offset provided or the amount of eager payload that was buffered to increment the offset used in the read.

3.4.4.6 Deferrable Send Processing

A deferrable send from the initiator SHOULD NOT send more data than the current congestion management window size for the PDC that the deferrable send uses before it receives a semantic response from the target. On receiving a semantic response with RC_OK, if the **ses.modified_length** is 0, the deferrable send implementation MUST send at least one additional packet that has **ses.eom** set. This additional packet MAY be 0 bytes, Payload MTU bytes, or the remaining fragment of the eager portion. The deferrable send implementation MUST send the entire payload if the **ses.return_code** is RC_OK and the modified length is equal to the request length.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

In some cases, the target may not want to accept the entire deferrable send operation. For example, it may be an unexpected message in a buffered unexpected header implementation (3.4.4.7.2.1). In such cases, deferrable sends stop the transfer of payload. An implementation of deferrable sends SHOULD implement a buffered header solution (3.4.4.7.2.1) and MUST implement a back-off and retry (3.4.4.7.1.1) for when the header space is exhausted.

The deferrable send message contains a token to restart the message transfer that consists of a 32-bit **ses.initiator_restart_token**—allocated entirely by the initiator—and a 32-bit **ses.target_restart_token**, allocated entirely by the target. In the initial request, the **ses.target_restart_token** MUST be set to 0. When the target is ready to restart a deferred send, it sends a ready-to-restart (RTR) message containing the **ses.initiator_restart_token** to the initiator and restarts the message transfer. The target MAY allocate a target restart token and include it in the RTR message. If the target does not allocate a target restart token, the target MUST set the **ses.target_restart-token** field to 0. If the target allocates a target restart token, it MUST guarantee that the restarted deferrable send will match a buffer. Otherwise, the target MAY defer the deferrable send again. If a deferrable send received a semantic response with a **ses.modified_length** of 0, the implementation MUST NOT restart the message until the corresponding RTR is received. Such a response is a request to defer the deferrable send. The restart MUST send the portion of the original message requested by the RTR message. An implementation that uses the back-off and retry method MUST NOT send an RTR message for the corresponding deferrable send. In contrast with the **ses.modified_length** of 0 response above, the back-off and retry scheme uses an RC_NO_MATCH response that indicates that no header was captured and that an RTR will never be generated.

Implementation Note:
The usage of the restart token is defined to directly signal the state of the transaction to implementations on both ends. An initiator can know whether the deferrable send may be deferred again, and the target can know whether resources have been allocated.

Informative Text:
Rendezvous transactions and deferrable send transactions are semantically similar operations with different optimization points and different implementation implications. Rendezvous is optimized to not utilize bandwidth when messages are unexpected, which is common in HPC. Deferrable send is optimized for latency in the expected message case. An implementation of rendezvous could set eager length to the request length, then truncate the eager portion of the message to the size of the modified length when the semantic response returned. In this case, the difference between rendezvous and deferable send would be whether the remainder of the data was transferred using a read message or by restarting a prior transmit message.

3.4.4.6.1 Supporting Deferrable Send as Send

For implementations that do not support deferrable send, it is possible for a target device to implement deferrable send operations (UET_DEFERRABLE_SEND) as send operations (UET_SEND) by using a subset of the functionality defined above (and below). This applies to tagged sends (UET_DEFERRABLE_TSEND /

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

UET_TSEND) as well. A target can accept a deferrable send and treat it as a send by ignoring the initiator and target restart tokens. When deferrable send is treated as a send, the target MUST respond with either a **ses.return_code** of RC_NO_MATCH and a **ses.modified_length** of zero, or a **ses.return_code** of RC_OK and a **ses.modified_length** that is equal to the requested length. This is used to cover the case of unexpected messages and expected messages, respectively.

3.4.4.7 Unexpected Message Handling

In HPC, unexpected messages are common. Unexpected messages may also be encountered in *CCL implementations. This section documents the unexpected message handling requirements when implementing *CCL over unordered, tagged send/receive operations in libfabric. It also covers the more challenging aspects related to the MPI ordered matching cases. As a preface, the traditional way to handle unexpected messages is using two mechanisms: For short messages, the payload is captured as eager payload in an unexpected message buffer at the target. For long messages, a rendezvous protocol is used so that the payload can be retrieved when the receive is posted. Exact matching simplifies the implementation of these traditional sequences. The objective of this section is to articulate the requirements for implementations so that a range of implementations are possible (with different levels of complexity), while still allowing for interoperability.

3.4.4.7.1 Unexpected Messages over RUD PDC

With unordered packet delivery in general, there is no guarantee of the ordering of resolving a message to a buffer. This dramatically simplifies the handling of unexpected messages—even in the case of wildcard tag matching.

Informative Text:
Properly mixing a RUD PDS and wildcard tag matching requires great care; however, it can be done. A server may have an RPC queue, for example, where the order of arrival of RPCs does not matter. In such a case, it may accept messages from any source or may use a subset of match bits to divide the RPCs into types. As long as software has expressed FI_ORDER_NONE, the techniques in this section can be used.

Three schemes are possible for unexpected message handling with unordered messaging: back-off and retry (3.4.4.7.1.1), buffered headers (3.4.4.7.2.1), and buffered unexpected messages (3.4.4.7.2.2). Buffered unexpected messages do not require any unique support at the initiator. Target FEPs MUST use at least one of the three schemes and all three MAY be used in combination. For example, an implementation can utilize buffered unexpected messages but degrade to buffered headers when the unexpected message buffering is exhausted. Or an implementation can operate in either buffered headers mode or buffered unexpected messages mode and degrade directly to back-off and retry.

Informative Text:
A combination mode of operation has been field tested on existing hardware that implements semantics similar to Portals 4. The exact wire details are different from what is described here, but the concepts have been deployed for unordered traffic with wildcard matching.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

3.4.4.7.1.1 Back-off and Retry

One solution point for unexpected messages is for the target to respond with a semantic response that indicates that the message could not be matched and was dropped (**ses.return_code** = RC_NO_MATCH). If RC_NO_MATCH is set, the **ses.modified_length** field MUST be set to 0. If a semantic response has an RC_NO_MATCH return code, the initiator MUST retransmit the message. This retransmit can happen in hardware or can be performed in software (e.g., inside the provider implementation). The appropriate back-off time is unknowable, and aggressively short retry times (a few RTT or less) will substantially increase wasted network bandwidth.

All implementations MUST support the back-off and retry mechanism with RUD PDCs for all types of send and tagged send operations.

3.4.4.7.2 Unexpected Messages with Buffering

Buffering at the target can be used to implement unexpected messages for both RUD and ROD PDCs. Buffering effectively creates the illusion that messages arrived after the receive call from libfabric instead of before it; thus, buffering of unexpected headers or messages is expected to conform to the ordering semantics of the underlying PDC. When buffers are exhausted, RUD PDCs can easily fallback to a back-off and retry scheme; however, a ROD PDC requires more complex handling (3.4.4.7.3).

3.4.4.7.2.1 Buffered Headers

In situations where the payload is recoverable, an implementation SHOULD choose to buffer semantic information associated with message headers that arrive and drop the payload when unexpected messages are encountered. The response in this case MUST be RC_OK with a modified length of 0. If an implementation chooses this mechanism, these message headers MUST be compared against new receive operations—that is, calls to *fi_recv()* or *fi_trecv()*—that were provided. There are two times when this can occur: when sends utilize rendezvous sends or deferrable sends.

When rendezvous sends are used with a buffered header mechanism, the buffered header MUST include the information needed to issue the read. When a new receive matches the buffered header, the rendezvous operation MUST be completed by issuing the read using the information provided in the rendezvous send. When deferrable sends are used with a buffered header mechanism, the buffered header MUST include the initiator restart token. When a new receive matches the buffered header, the ready-to-restart message MUST be generated using the initiator restart token.

Without buffered headers, an implementation cannot generate an RTR and cannot generate a rendezvous read; thus, failing to support buffered headers substantially defeatures both schemes (this is inherent to the definition of these operations). This scheme MUST ONLY be used in conjunction with rendezvous or deferrable sends.

3.4.4.7.2.2 Buffered Unexpected Messages

An implementation MAY choose to buffer both the message header and the message payload when unexpected messages are encountered. The response in this case MUST be RC_OK with a modified length equal to the request length for eager messages. Implementations choosing this mechanism MUST buffer the entire payload for eager messages. If an implementation chooses this mechanism, the

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

buffered message headers MUST be compared against new receive operations that were provided. If a new receive operation matches the unexpected buffered header, the implementation MUST provide the buffered payload to the target user application. This is accomplished via hardware or software. If an implementation has chosen this mechanism, it MAY degrade to the back-off and retry mechanism at any point (e.g., if the buffered header space is exhausted).

Rendezvous sends and deferrable sends[8] MAY be used with a buffered unexpected message scheme. If a rendezvous end uses buffered unexpected messages, the implementation MUST return a modified length less than or equal to the eager part of the rendezvous request. If a deferrable send uses buffered unexpected messages, the implementation MUST return a modified length of either zero or the request length. A modified length of zero is the signal in a response to a deferrable send that indicates to the initiator that the operation was deferred. Note, however, that an implementation MAY combine a buffered unexpected message scheme for standard sends with a buffered header scheme for rendezvous sends and deferrable sends.

3.4.4.7.3 Unexpected Messages and Resource Exhaustion over ROD PDC

This section applies to both untagged messages over a ROD PDC and tagged messages using wildcard matching over a ROD PDC and covers limitations of the resource exhaustion scheme when using ordering as well as how ordering interacts with resource exhaustion. Handling unexpected messages over an ordered PDC while using wildcard matching can be a challenging problem. Typically associated with flow control in MPI, the combination of a strong ordering requirement with the possibility of matching any receive buffer that has been posted has always been difficult to implement efficiently. Solutions like back-off and retry do not work well, because a wildcard receive can be posted just after a message has been rejected as unexpected. For untagged messages, any receive is effectively a wildcard receive. A subsequent message in flight from the same initiator could match the wildcard receive out of order because no other mechanism is available to establish ordering. Per-peer sequence numbering at the MPI level does not work, because one receive is allowed to specify that it wants to receive from MPI_ANY_SOURCE. The back-off and retry scheme MUST NOT be used for unexpected messages using wildcard matching over a ROD PDC or for untagged messages over a ROD PDC.

Because these solutions are hard, the general solution implemented is buffered unexpected messages. Buffered headers with rendezvous sends is also part of the typical solution. Implementations providing wildcard matching over a ROD PDC MUST implement buffered unexpected messages or buffered headers as a solution.

Buffered unexpected messages and buffered headers can still run into scenarios where the buffering resources are exhausted. These scenarios are relatively uncommon but happen deterministically in some applications. The libfabric over UET definition MUST include a buffer exhaustion solution. Scalable solutions to buffer exhaustion have long proven challenging. To cover a breadth of use cases, two solutions are provided in sections 3.4.4.7.3.1 and 3.4.4.7.3.2.

[8] In this paragraph, “sends” is intended to capture both tagged and untagged sends.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Informative Text:
Why not receiver not ready (RNR)? Verbs uses an RNR to address this problem. In RNR, a destination QP enters an RNR state. Recovery uses the process-to-process QP sequence numbering to restore message ordering. To enhance scalability, UET does not have process-to-process sequence numbers. Hypothetically, the PDC sequence numbers could be used instead; however, that has several undesirable characteristics. First, it couples the reliability logic and semantic logic at the target in a way that makes it hard to achieve high message rates on a single PDC. Second, it couples the reliability logic and semantic logic at the initiator in a way that stalls the PDC until the semantic resource issue is resolved. Finally, resource exhaustion on one process using a PDC stalls all other users of that PDC.

3.4.4.7.3.1 Scalable Parallel Application Deployments

For the most scalable applications, a mechanism is provided to allow a low-state recovery from resource exhaustion events while still preserving message ordering (i.e., while using a ROD PDC). It is preferable not to require any per-target tracking at the initiator; thus, the client/server mechanism is less preferred. The scalable mechanism is simple at a transport level. When resources are exhausted, the associated target {FA, JobID, PIDonFEP, Resource Index} tuple is disabled. RC_DISABLED is returned for the target {FA, JobID, PIDonFEP, Resource Index} tuple. All subsequent operations to that target tuple MUST receive an RC_DISABLED response. That tuple remains in a disabled state until re-enabled by software. Before being re-enabled, software MUST guarantee that no operations are in flight (i.e., on the wire) to guarantee that ordering is not violated. Once this condition is met, software MUST re-enable the tuple at the target. After the target is re-enabled, all initiators MUST retransmit all operations that received an RC_DISABLED response in their original order.

Informative Text:
In Portals 4, a flow control solution was proposed where the equivalent of a libfabric endpoint (a portal table entry) is placed into a disabled state. Subsequent operations to that portal table entry would complete with a bad return code at the initiator until the portal table entry was re-enabled. Before a portal table entry could be re-enabled, the target with the disabled portal table entry must first confirm that there were no initiators that were in the middle of an ordered stream. That is, it must first confirm that no initiators had a message that had been rejected and yet still had other messages in flight. The proposed solution was to use a barrier amongst all participating processes. This solution is modeled after that proposal.

3.4.4.7.3.2 Client/Server Buffer Exhaustion

In client/server interactions, the server cannot trust the client to check-in after receiving an RC_DISABLED response. UET introduces the concept of a “generation” for a Resource Index as a separate field carried in the header when using tagged or untagged messaging. At the target, a Resource Index MAY be configured to use a generation. If it is not configured to use a generation, the generation MUST be zero. A message that arrives at the target MUST contain the correct generation, or the message MUST receive a semantic response indicating a generation mismatch (RC_BAD_GENERATION).

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

This response MUST contain the current generation. A tagged or untagged message using a ROD PDC that encounters resource exhaustion at the semantic level that prevents accepting the message MUST disable the target Resource Index. The message MUST receive a semantic response indicating that the message encountered a disabled index (RC_DISABLED_GEN). The index MUST remain disabled until additional resources are available and the generation number is incremented.

The implementation—whether in the provider layer or hardware—MUST retransmit messages that encounter a disabled index. When the message is retransmitted, the implementation SHOULD increment the index generation before retransmitting the message; however, the implementation MAY retransmit using the same index until it receives a generation mismatch response. If the implementation chooses to retransmit the message with an incremented generation, it MUST retransmit only one message (the oldest message needing retransmission) until it has received a successful semantic acknowledgement. This avoids a race where the generation could be incremented between the first retransmitted message and second retransmitted message reaching the target. A message encountering a disabled index SHOULD defer retransmit using an exponential back-off scheme.

The implementation MUST retransmit messages that encounter a generation mismatch. The retransmitted message MUST use the new generation returned in the generation mismatch response. A message encountering a generation mismatch MAY be retransmitted immediately.

Informative Text:
This mechanism is named “client/server buffer exhaustion” because it requires a state per-peer process. Most of the specification goes through great lengths to avoid per-peer-process state in the fast path of messaging libraries for compute applications.

Implementation Note:
RC_DISABLED_GEN is a scenario similar to traditional resource exhaustion concepts (e.g., receiver not ready). Because the target cannot easily track the full list of initiators that have encountered RC_DISABLED_GEN, initiators can rely only on traditional retransmit heuristics to determine when to retransmit the message.

3.4.4.8 Synchronized Operation Processing

Each group of synchronized operations consists of up to 2^{16}-2 write messages using the UET_SYNC_WRITE opcode. Each group of synchronized operations MUST include exactly one synchronized completing operation—either a UET_SYNC_WRITE with the **ses.hd** bit set or a UET_SYNC_ATOMIC. After sending the synchronized completing operation, the initiator MUST wait for all messages in the group to complete—that is, receive all acknowledgements at the initiator—before reusing the **ses.sync_group** value.

The target MUST track each in-progress sync group until it completes. The target MUST honor the ordering rules defined in 3.4.2.7.4 and 3.4.9.6. The payload associated with a UET_SYNC_WRITE MUST

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

NOT be delayed for the purpose of being ordered behind other UET_SYNC_WRITE messages when RUD is used.

The target SHOULD associate each **ses.sync_group** with the **ses.JobID** that initiated it, and SHOULD verify the **ses.JobID** against the stored value for a **ses.sync_group**.

Implementation Note:
Implementations should be mindful of the need to cleanup synchronized operation tracking resources in failure scenarios. For example, when a sync group that is occupying a resource does not complete at the target.

3.4.5 Semantic Protocol Sequences

This section describes the general semantic sequences in the protocol. Ladder diagrams are provided for the various semantic operations to illustrate the relevant concepts. For simplicity, the diagrams do not illustrate out-of-order packet delivery. PDS-level packets are not shown unless they are relevant to the semantic processing. Not all information carried on a packet is illustrated. Only those fields that are needed to illustrate the concept are included. In this section, lossless networks (as noted in the PDS specification) use TC_request for green and purple arcs, while the blue and orange arcs use TC_response. Similarly, in best-effort operation, the solid arcs use TC_low while the dotted arcs use TC_high. Usage of trimming (and the corresponding TC_med) is not covered in the semantic figures. Section 3.6.4.7 specifies the mapping of UET to traffic classes and DSCP values. In some figures, two PDCs are used: a forward-direction PDC (green requests with blue responses) and a reverse-direction PDC (purple requests with orange responses). Green request/completion arcs are shown in the interaction between libfabric and SES for initiator to target messages, and purple request/completion arcs are shown in the interaction between libfabric and SES for target to initiator messages.

3.4.5.1 Requests with Payloads

Requests with payloads consist of sends, tagged sends, writes, and atomics. In the figures here, send operations are used as a general illustration.

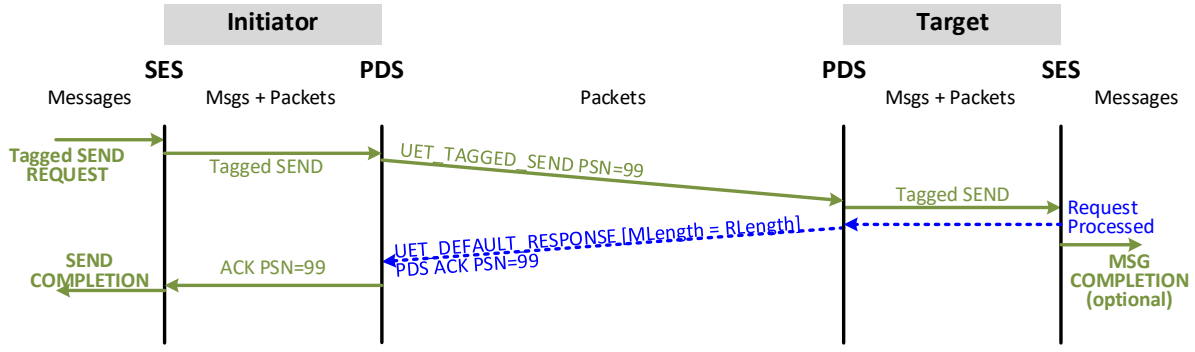


Figure 3-29 - Single-Packet Request, Expected Message

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

3.4.5.1.1 Single-Packet Requests

As illustrated in Figure 3-29, a single-packet request has a single-packet response (UET_DEFAULT_RESPONSE) where the semantic ACK is combined with the PDS ACK. The PSN from the PDS ACK is used to identify the original request. Optimized header formats that may be used for a single packet message do not contain the message ID field; thus, they MUST use the PSN as the mechanism to identify the original request. This enables SES to deliver a completion at the initiator. The modified length is equal to the requested length indicating that the entire message is delivered. The semantic ACK also indicates that the message was captured in the expected list. This can then be used to build a lightweight *MPI_Ssend()*. In this sequence, UET_DEFAULT_RESPONSE SHOULD be used (as illustrated), but UET_RESPONSE MAY be used (not shown). SES MUST NOT return the semantic response to PDS before the packet has been processed through SES. If delivery complete is set (DC=1), the semantic response MUST NOT return until the payload has been made globally observable. The single-packet sequence illustrated applies to a variety of opcodes, including UET_TAGGED_SEND (shown), UET_WRITE, UET_ATOMIC, UET_ SEND, and UET_TSEND_ATOMIC. The UET_TAGGED_SEND shown in Figure 3-29 can use either UET_HDR_REQUEST_STD or UET_HDR_REQUEST_MEDIUM (Figure 3-16). The choice of header has some impact on the addressing operations but does not change the overall sequence.

3.4.5.1.2 Multi-Packet Requests

Figure 3-30 illustrates some additional concepts that are relevant to multi-packet transactions. For the transaction shown, the first PDS ACK includes a semantic response with the UET_NO_RESPONSE opcode. This indicates that the packet has been received but that the semantic processing is not yet complete. It is allowable for many packets to generate a UET_NO_RESPONSE opcode. In conformance with the requirement that at least one semantic response be generated, a later packet—arriving after semantic processing has completed—generates a UET_DEFAULT_RESPONSE. This is an expected message that has

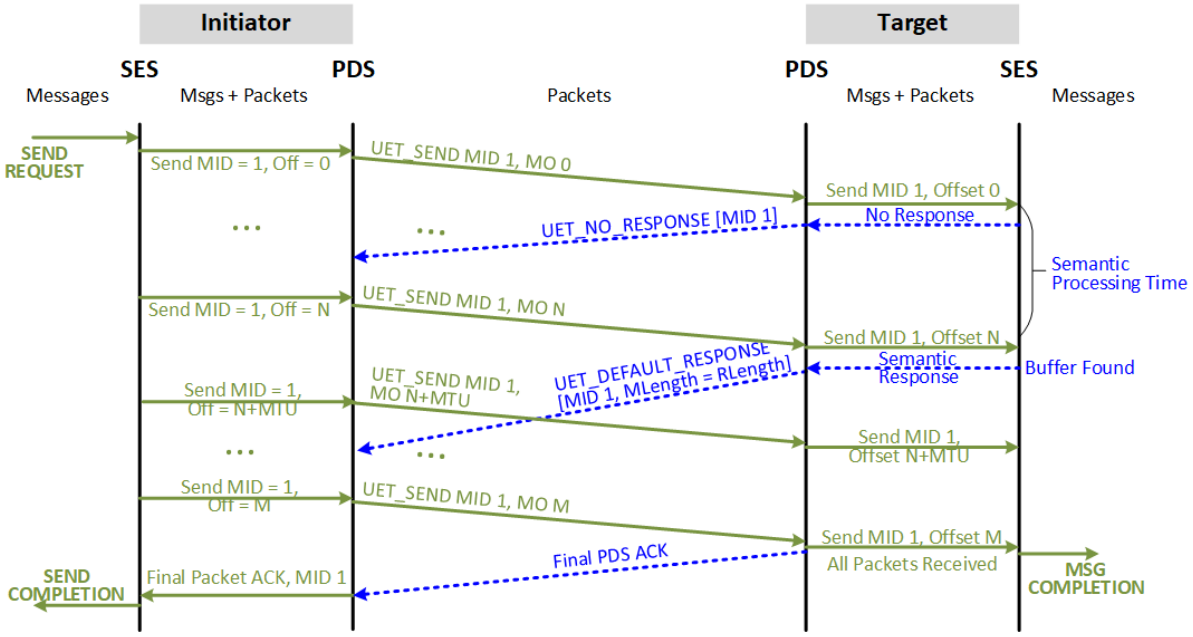


Figure 3-30 - Multi-Packet Request, Expected Message

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

found a buffer and will be delivered in its entirety. Note that every response in most sequence diagrams carries a semantic header. Some arcs, such as the final arc in Figure 3-30, are not labeled with the semantic response opcode to keep the diagram simple, since many options are possible.

3.4.5.2 Requests with Responses That Have Payloads

Read requests (UET_READ) generate payload data in the return direction. There are two mechanisms for returning data—as illustrated in Figure 3-31 and Figure 3-32. (Note: Neither figure illustrates PDS acknowledgements.) Both figures illustrate the multi-packet case. For a read request with a request length that is larger than one Payload MTU, SES breaks the read request into one packet (UET_READ) per Payload MTU worth of response data. Each read packet associated with one message contains the same message ID, and each read packet requests one Payload MTU of data—except for the last read request, which can request the final fragment. The sequence is the same for single-packet read requests. The choice between the two sequences is based on the value of PDS_MAX_ACK_DATA. When PDS_MAX_ACK_DATA is greater than the request length or PDS_MAX_ACK_DATA is greater than the Payload MTU, then the sequence in Figure 3-32 is used. Otherwise, the sequence in Figure 3-31 is used. When a fetching atomic request (UET_FETCHING_ATOMIC) is issued, the sequence in Figure 3-32 is always used. A broader discussion of these sequences can be found in the PDS specification 3.5.19. The remainder of this section focuses on the semantic sublayer characteristics of these sequences.

When the data that can be carried in an acknowledgment is small, the target issues new messages containing the read payload data. These messages are technically responses; thus, the UET_RESPONSE_W_DATA opcode is used. Reads (optionally) complete at the target when all of the

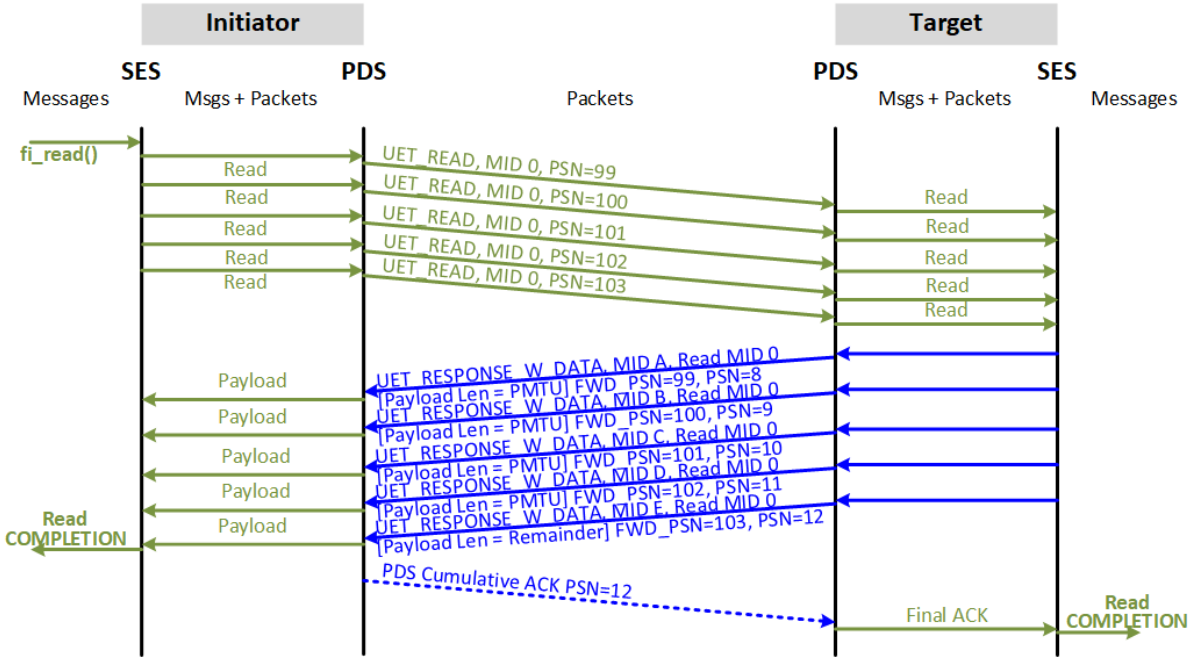


Figure 3-31 - Multipacket Read Request – Standard

response packets for the read have been acknowledged. Note that the response payload packets

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

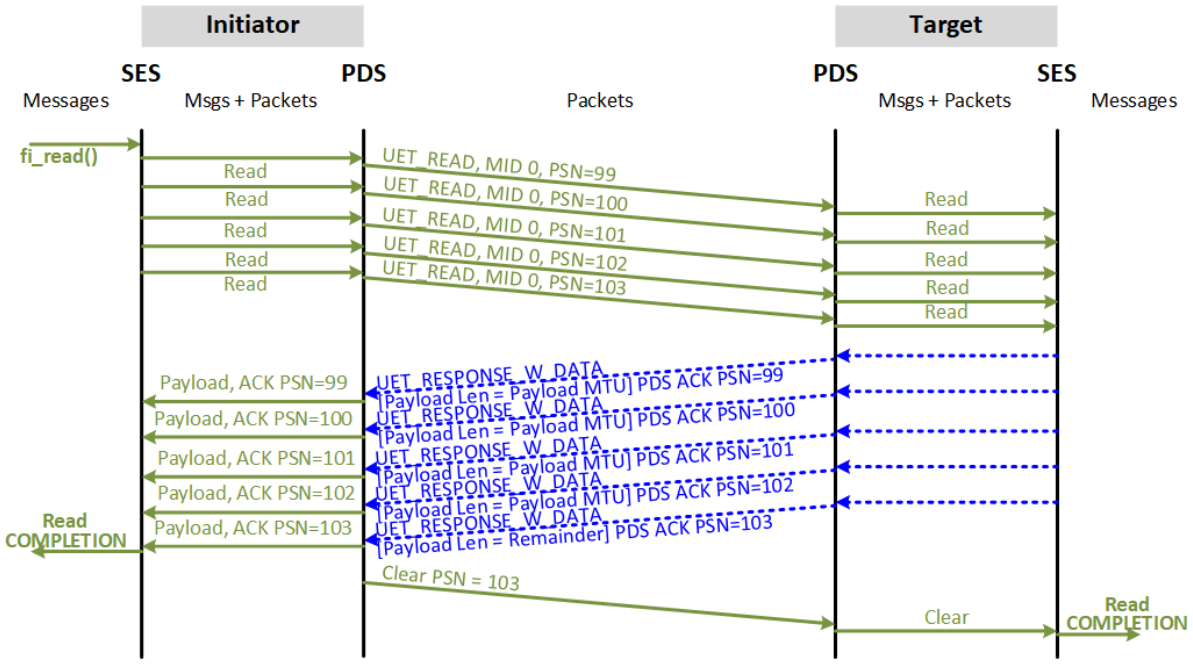


Figure 3-32 - Multi-Packet Read Request – Large PDS_MAX_ACK_DATA

contain the original message ID of the request and the original sequence number. The message ID from target to initiator is only reflected to the target in the acknowledgements; thus, it is entirely within the purview of the target how they are assigned. Specifically, all responses to a single request message are not required to use a single-message ID.

As illustrated, Figure 3-31 allows for a semantic response to the UET_RESPONSE_W_DATA carrying the read payload. This makes it possible to carry failure information from the initiator to the target; however, Figure 3-32 does not have this ability to carry failure information. A careful review of Table 3-25 shows that very few error responses could possibly apply when a read response returns to the initiator of the read. Implementations of the standard read sequence (Figure 3-31) MUST provide a semantic response to the UET_RESPONSE_W_DATA; however, that semantic response MAY be UET_DEFAULT_RESPONSE in all cases. Indeed, this is what is illustrated in the figure where there is a single cumulative acknowledgement. Implementations of the target MUST be prepared to receive an error response, which implies a guaranteed delivery response, and MUST clear packets that were marked for guaranteed delivery. Target implementations SHOULD deliver any error response provided in the read completion.

Informative Text:
Errors related to a UET_RESPONSE_W_DATA can occur when the payload attempts to write memory that does not have a valid translation available. This can occur due to a programming error or due to the application terminating.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

In Figure 3-32, payload data is carried in the acknowledgement. Read request formation is the same as in the flow used in Figure 3-31, but response data is carried in the acknowledgements. Because of this, the read completion is delivered after all of the response packets are cleared.

The choice of protocol for reads impacts the mapping of the flows onto TCs. Acknowledgements are mapped to the response TC (as illustrated in Figure 3-32). This is the primary reason that bulk data reads are expected to typically use the sequence in Figure 3-31. It should be noted, however, that lossless networks map the bulk data onto a response TC in both cases. In addition, deadlock-free operation on a lossless network may require additional care with resource management for the sequence shown in Figure 3-31. In both sequences, the initiator MUST guarantee that all response data can be accepted without generating any wire transactions from initiator to target.

In Figure 3-31, each UET_READ is acknowledged by a PDS acknowledgement. That PDS acknowledgement carries a semantic response of either UET_NO_RESPONSE, UET_DEFAULT_RESPONSE, or UET_RESPONSE. SES failures can be carried by the UET_RESPONSE opcode. Errors detected later in semantic processing are returned with the data. Read request packets that receive a non-RC_OK return code as part of the PDS acknowledgement do not receive a UET_RESPONSE_W_DATA. In Figure 3-32, the semantic response return code is returned as part of the UET_RESPONSE_W_DATA.

3.4.5.3 Rendezvous Transactions

Rendezvous transactions have two parts. The first part consists of the eager send (e.g., UET_RENDEZVOUS_SEND). The eager send has 0 or more bytes of payload as indicated by the eager size in the rendezvous header. The second part is a rendezvous read message (UET_READ) that completes the transfer by pulling the data. The rendezvous read is an ordinary read transaction using the fields provided in the rendezvous header. Figure 3-33 illustrates how this works in the expected message case. A rendezvous send message (i.e., UET_RENDEZVOUS_SEND) is initiated and consists of one or more send packets carrying eager payload (illustrated with UET_RENDEZVOUS_SEND MID1, Message Offset (MO) 0 through N). Notionally, the eager size is set to cover the round-trip bandwidth delay product, so that the first packet of the read request arrives just in time to start the read payload transfer[9]. There are two completion points at the initiator of the transaction: the completion of the rendezvous send and the completion of the read. These are delivered as one completion queue entry to software (i.e., through the libfabric API). Similarly, completion at the target is based on both the arrival of the eager part and the arrival of the read payload. Due to variations in network timing, the completion of the eager send and the completion of the read can happen in any order; thus, both points are illustrated here. Nonetheless, both completion points must be reached before the message completion (e.g., completion queue entry) can be returned through the libfabric API.

Note the interaction with PDS for read completion at the initiator of the rendezvous transaction. The PDS provides reliability for read responses; thus, the completion of the read at the initiator of the rendezvous occurs when all of the read response packets have been acknowledged.

[9] There are a variety of reasons why the eager size may be set smaller. For example, the initial rendezvous send may be ordered while the read response payload may be unordered.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Figure 3-34 illustrates the unexpected message variation for the rendezvous transaction. The difference between the two sequences is limited to two features. First, the modified length in the response is intended to truncate the eager portion of the message to the part that is buffered at the target. In the figure, it is assumed that the target buffer is matched to the size of the eager portion; however, a modified length could be shorter (often 0 bytes). In that case, the UET_RENDEZVOUS_SEND can be completed with a single additional packet—just like the deferrable send transaction. Second, the start location of the read transaction differs from the expected case. When the message is unexpected, the read starts from the point in the message that has been buffered at the target.

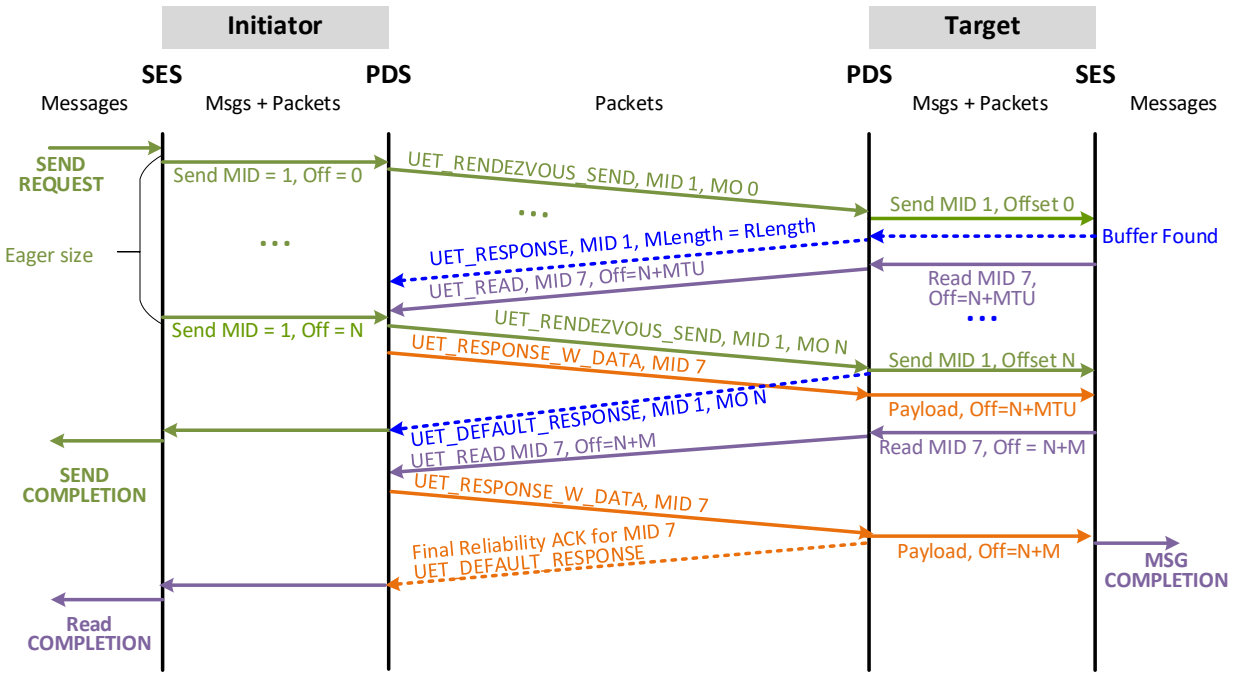


Figure 3-33 - Rendezvous Transaction, Expected Message Case

When implementing a rendezvous transaction, a target MAY read more data than is strictly necessary. Any offset within the bounds of the original message is a legal starting point for the read that is part of the overall rendezvous transaction.

Informative Text:
Reads of more data than strictly necessary can enable implementation optimizations (e.g., for performance).

Implementation Note:
Rendezvous transactions have two completion points. One is when the eager send(UET_RENDEZVOUS_SEND) has completed, and the other is when the read (UET_READ) has completed. The eager send and read are two independent messages. As such, for a variety of reasons,

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

they may complete in either order at the initiator or the target. Implementation-specific mechanisms are used to reconcile the two completion points and deliver one completion queue entry to the user application.

Figure 3-33 illustrates a specific two completions at the initiator. In this example, an implementation would use the fields of the read—the Resource Index and the Memory Key—to encode information that would allow the read completion to be mapped to the original transaction. This is fully supported in the Memory Key definition by using a provider defined key. Whatever bits are chosen for encoding such information would need to be delivered in a read completion between the device and the libfabric provider.

Read completions can be implemented by delivering a completion after every read response has been acknowledged. Unacknowledged read responses are retransmitted by PDS until they are acknowledged.

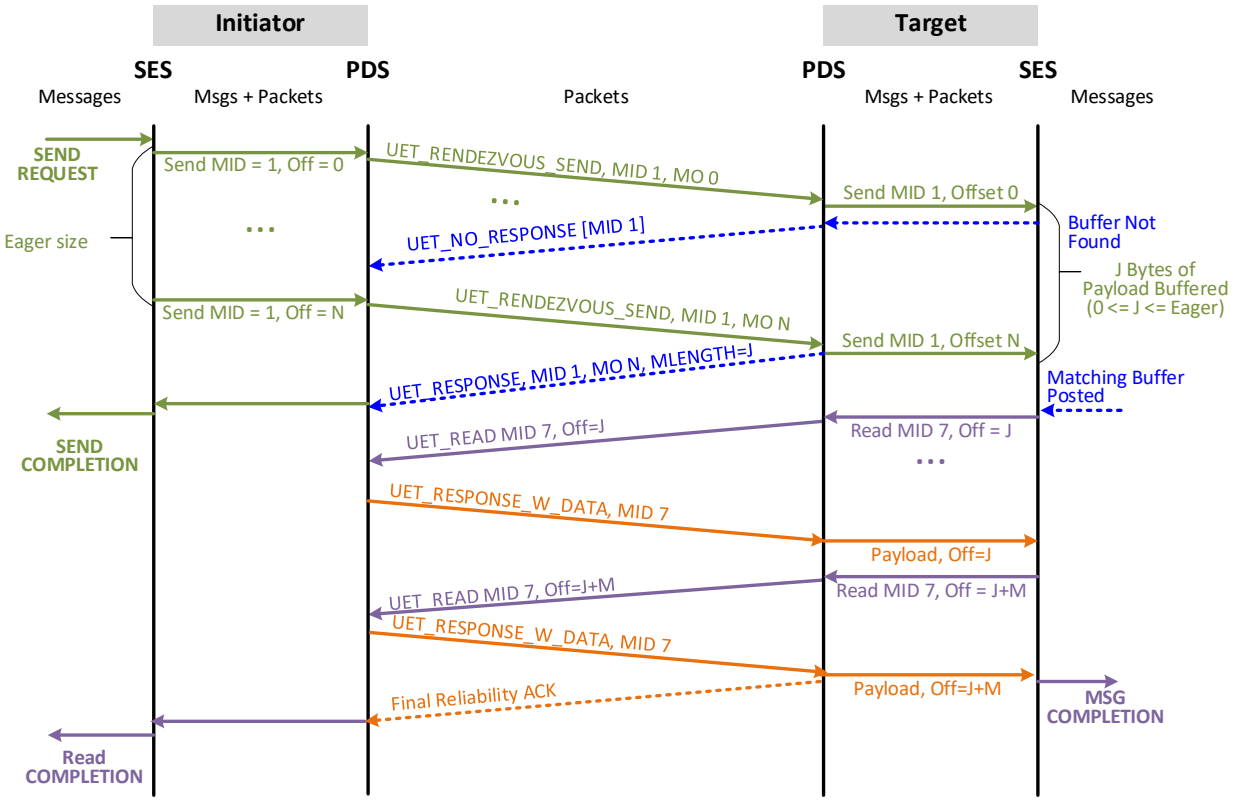


Figure 3-34 - Rendezvous Transaction, Unexpected Message Case

3.4.5.4 Deferrable Send Transactions

Deferrable send transactions are a similar concept to rendezvous sends. Deferrable sends begin a transfer of the message. The congestion management sublayer is expected to limit the outstanding payload based on the current congestion state. This means that approximately one BDP of data at the

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

current achievable bandwidth should be outstanding when a response returns from the target. In Figure 3-35, the deferrable send (UET_DEFERRABLE_SEND) finds that the matching buffer is available at the target. This causes a semantic response (UET_DEFAULT_RESPONSE) indicating RC_OK and a modified length equal to the request length. The deferrable send proceeds as any normal send.

The goal for deferrable sends is that the expected cases (seen in Figure 3-35) will return the semantic response just in time to keep the wire fully saturated. In conjunction with the congestion management sublayer, this should be achievable. In this case, there is little difference for any of the processing from a simple send transaction.

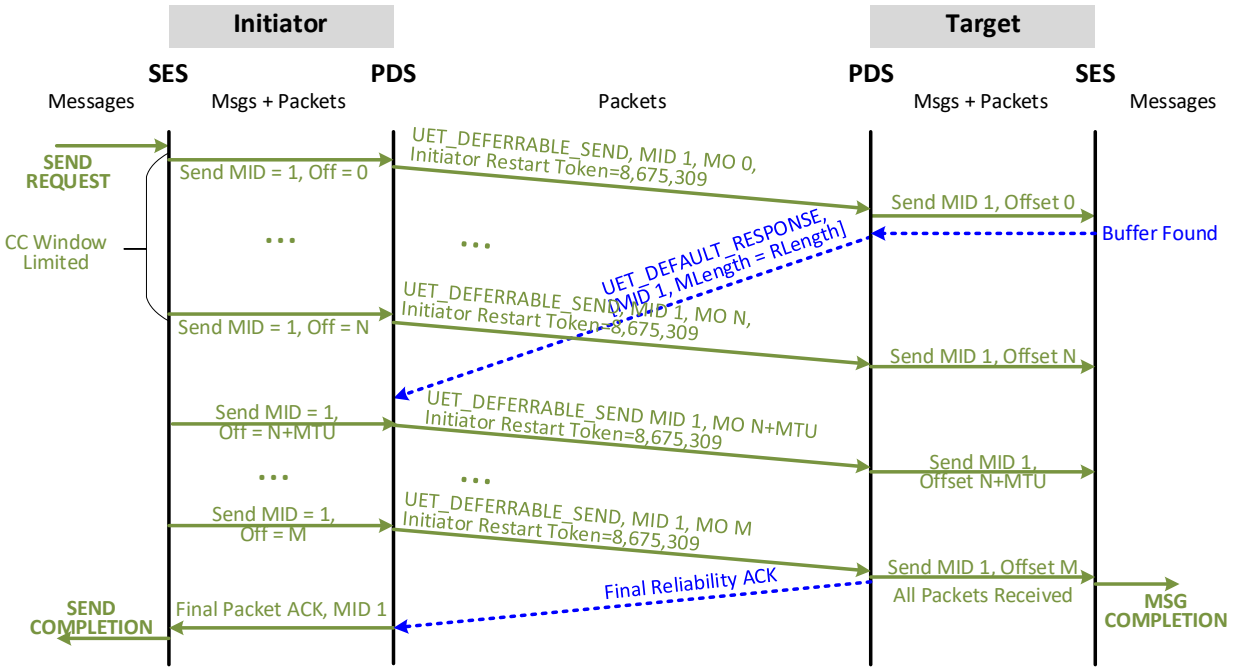


Figure 3-35 - Deferrable Sends, Expected Message Case

In the transaction illustrated in Figure 3-36, the semantic ACK (UET_RESPONSE) indicates that the message is to be deferred by setting the **ses.return_code** field to RC_OK and the modified length to 0. This response is marked for guaranteed delivery, since it does not use the default set of response values. Later, a restart token (8675309) is used to restart the message using UET_DEFERRABLE_RTR. In the illustrated example, the target has chosen to buffer J bytes of the original message. Thus, the ready-to-restart (RTR) message indicates that the restart should start at message offset J. The restarted message MUST be delivered to the matching buffer that was provided. The target MUST remember J using some mechanism (see Implementation Note), since it is not carried in the restarted message.

A second notable feature in Figure 3-36 is that the first message for the unexpected deferrable send MUST be completed using a packet with **ses.eom** set. This allows the target to deallocate state associated with that message.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

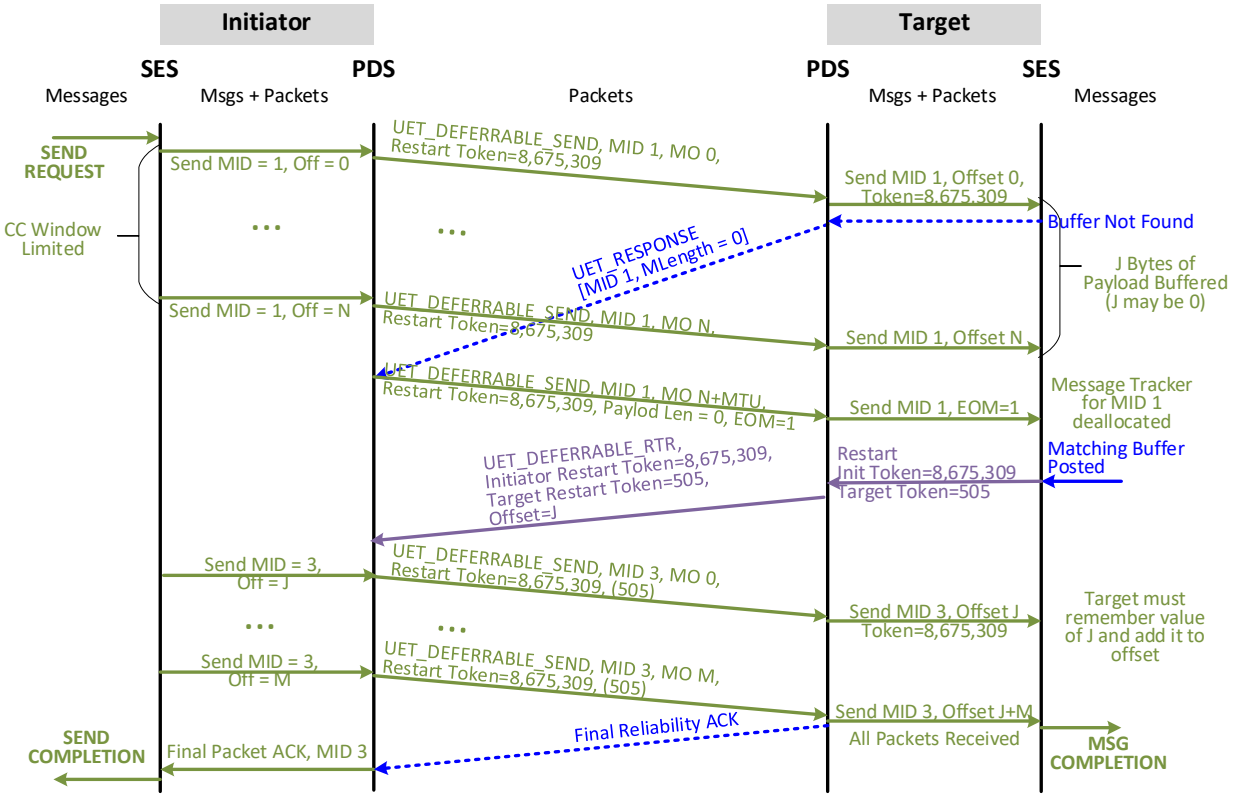


Figure 3-36 - Deferrable Sends, Unexpected Message Case

A final variation on deferrable sends is illustrated in Figure 3-37. In this scenario, the target does not reserve the buffer when a matching buffer is posted. This means that a deferrable send can be started, deferred, restarted, and redeferred. This sequence can occur an indefinite—even infinite—number of times. The lack of a reserved buffer is indicated by the passing of a target restart token of 0. A target restart token of 0 MUST NOT be used for a reserved target buffer. In Figure 3-37, message ID 1 is reused for the third transmit of the deferrable send, since that message ID was no longer in use for an in-flight message.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

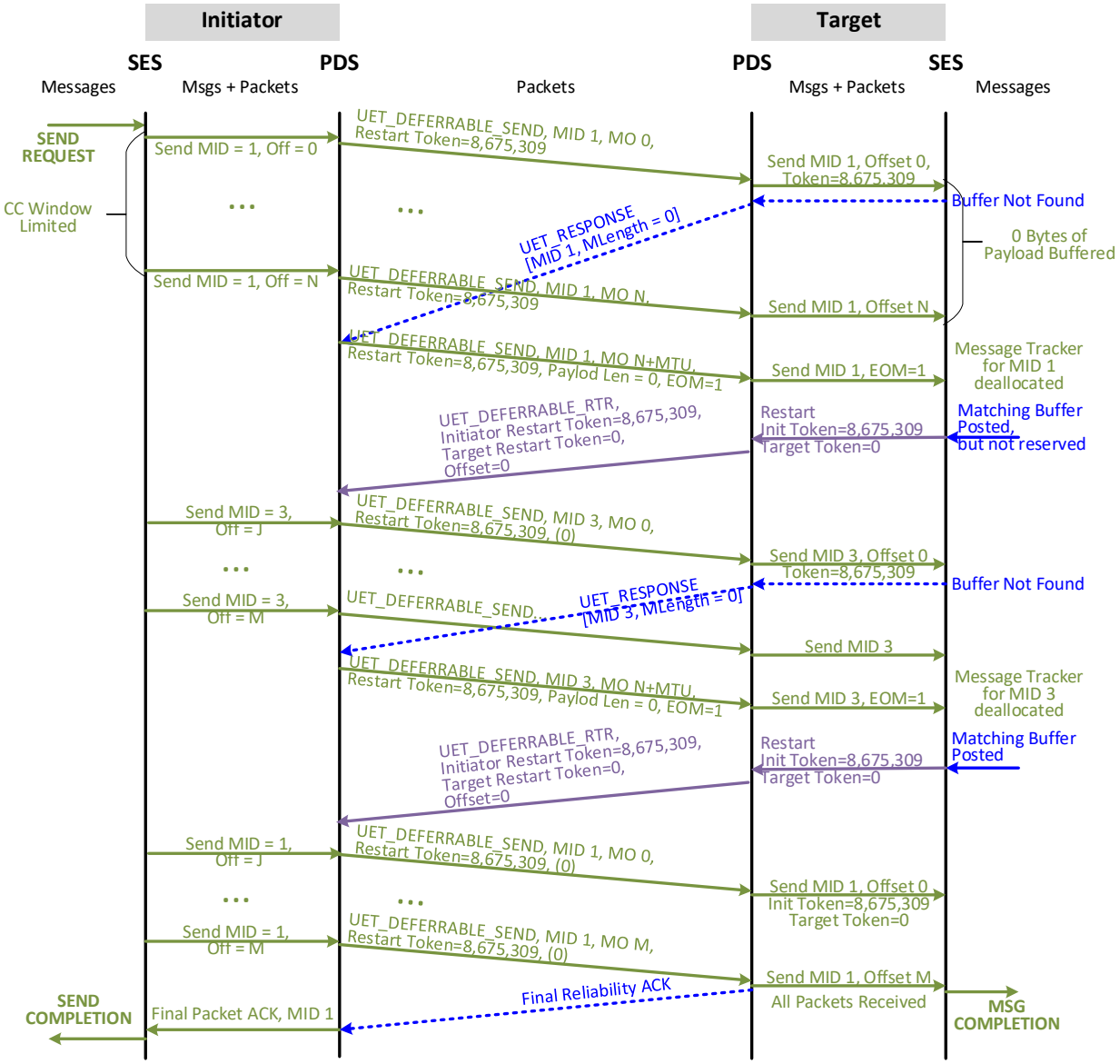


Figure 3-37 - Deferrable Sends, Unexpected Message Case, No Reserved Buffer

Implementation Note:

Implementation of the expected message case for deferrable sends is relatively straightforward. The unexpected message case—where the deferrable send is deferred—requires a little more care. While there are many choices an implementation can make, this note provides guidance for an implementation that could be used. It is necessary to achieve several things with an implementation:

1. The first packet of the deferrable send to arrive at the target must be identified as a new message.
2. Subsequent packets of the deferrable send must be correlated with the first transmission of the message.
3. It must be possible to identify the first packet of the restarted deferrable send as a restart at the target.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

4. It must be possible to reconcile the restarted deferrable send to the original message information.
5. Subsequent packets of the restarted deferrable send must be reconciled to the ongoing message.
6. Because of inherent race conditions in the sequence, the original deferrable send and multiple restarts of the deferrable send could hypothetically have packets in the network concurrently.

These operations can be accomplished using various mechanisms. To establish (1), the FEP can examine the target restart token and note that it is 0. This indicates this is not a restarted message, or, if it is a restarted message, it does not have a reserved buffer at the target that it must select. Because this is the first packet of a deferrable send that does not have a reserved buffer at the target, the FEP can proceed with semantic processing. Semantic processing indicates that the message is to be deferred, and a tracker is associated with the MID = 1. Packets associated with MID = 1 will be discarded, and the tracker will be discarded when a packet for MID = 1 with **ses.eom** set is received.

Later, a restart is initiated. Before sending the restart, the target FEP can allocate a table entry indexed by 505. 505 is then placed in the target restart token in the ready-to-restart message. The restarted message would contain (505) with the initiator restart token 8675309. This table could indicate where the restarted message would land in memory. For example, if the base address of the matching buffer was 42, then the start address of the restarted transfer would be 42+J. Alternatively, when the restart is initiated, the target FEP may not allocate a table entry. In this case, it populates the target restart token with 0 so that the restarted message is handled like a new message.

When the restarted message arrives at the target FEP, if it finds that the target restart token was not zero, it uses the target restart token to index the table. The restarted message always has a different message ID (e.g., MID = 3) from prior in-flight iterations of this deferrable send. It would then find that the restart token table contained 8675309 at entry 505. It would use this information to establish a tracker for MID = 3 and start delivering payload into the target memory. Subsequent packets of MID = 3 would find this MID tracking entry.

3.4.5.5 Synchronized Operation Sequences

Synchronized operations group write operations together with a completing action. The completing action can be either a write completion with header data (immediate data) or a non-fetching atomic operation. When all writes in the sync group have been placed, the completing action is performed. It is important to note that the sync group is finished only when the completing action is finished. No ordering between sync groups or writes within a group is maintained.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Figure 3-38 illustrates two sync groups (42 and 2112) associated with 8 and 6 write operations respectively. The messages are logically posted in order at the initiator. When the first packet arrives at the target, a new sync group is allocated in a local sync group tracking resource (implementation dependent). Subsequent write messages for sync group 42 increment the counter allocated to sync group 42. When the synchronized write immediate operation arrives, the final count (**ses.sync_group**) and header data (immediate data) are stored in the tracker. The same sequence of messages is done for sync group 2112. Once the operation count is reached for the sync group, the completion using the stored header data is posted. Note that in this example sync group 2112 completes before sync group 42 even though it was logically posted first.

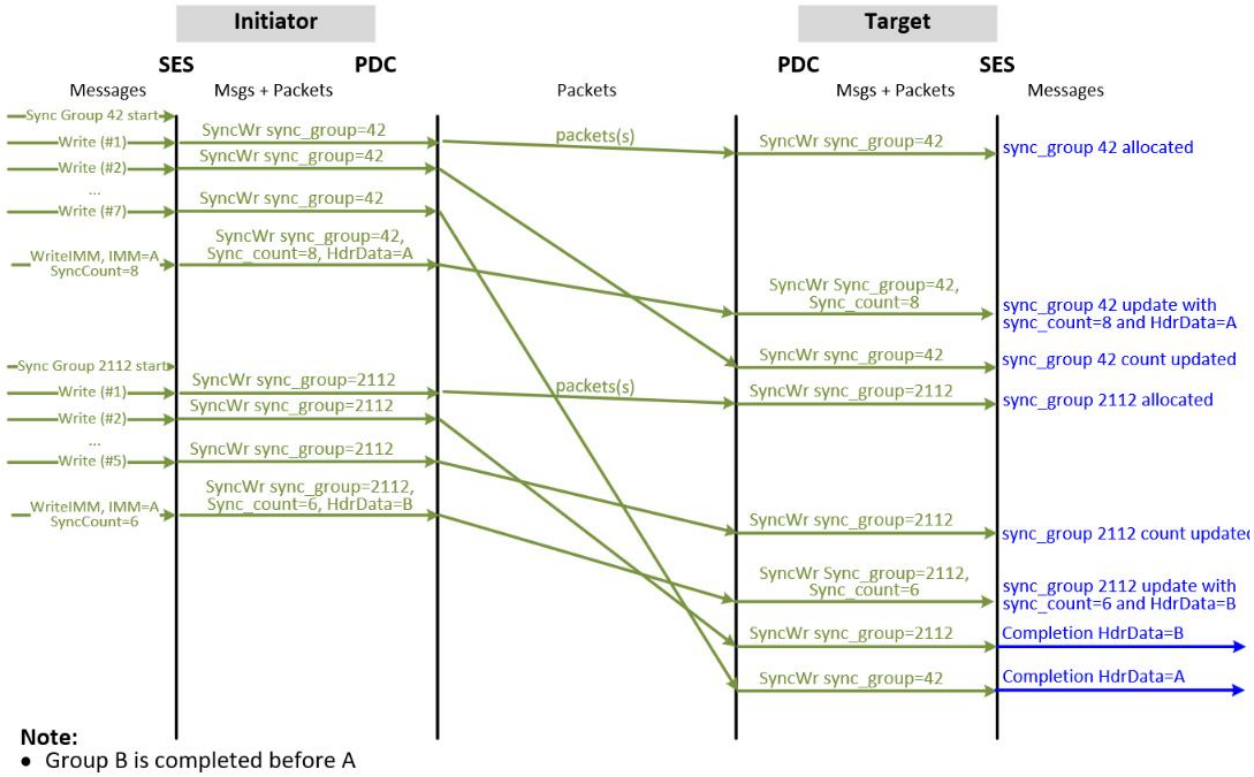


Figure 3-38 - Synchronized Operation Sequence with Write with Data

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

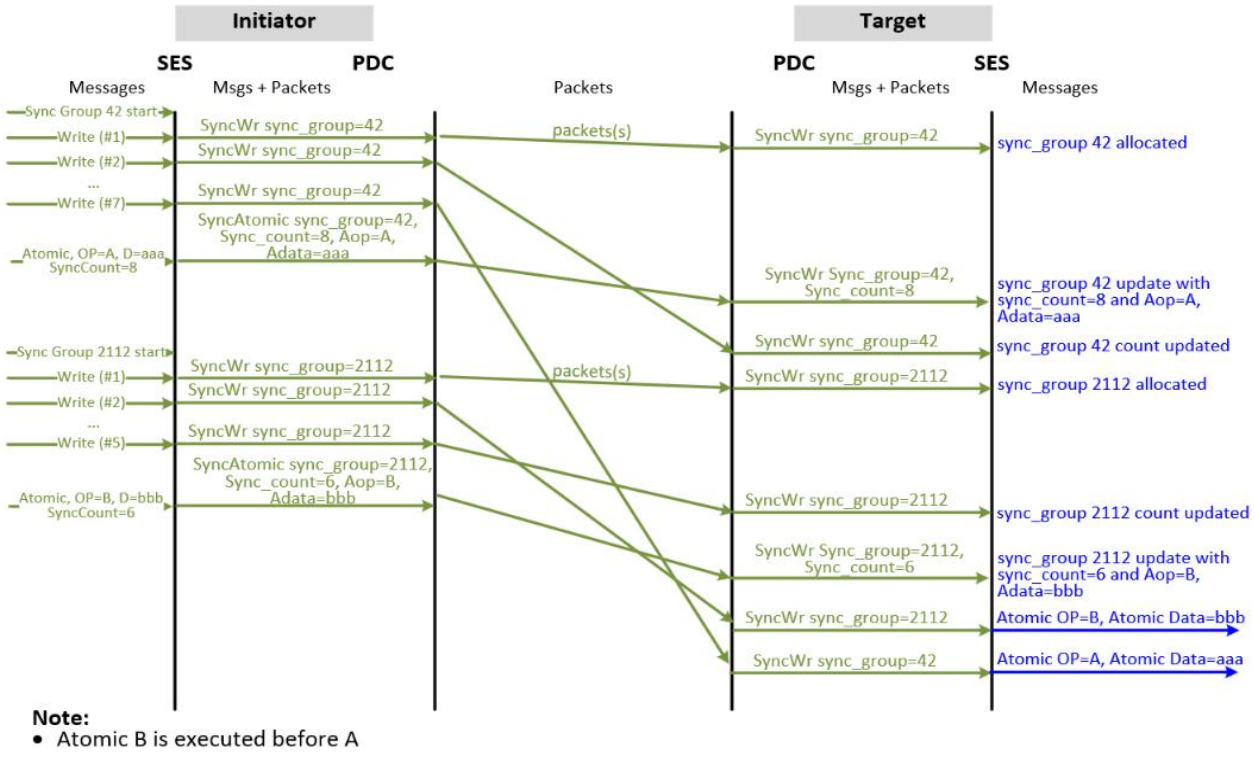


Figure 3-39 - Synchronized Operation Sequence with Atomic

Figure 3-39 illustrates the use of a non-fetching atomic as the synchronized completing action. The group is allocated in the same fashion as the write header data case. When the atomic operation is received for the sync group, the atomic operation and atomic data are stored instead of the write header data. When all operations for the sync group are received, the non-fetching atomic stored for the sync group is executed. Note that in this example the atomic sync group 2112 is executed before sync group 42 even though it is logically posted second.

3.4.5.6 Additional Unexpected Message Sequences

Unexpected messages are common in both AI and HPC and are a requirement as part of the libfabric provider. To achieve interoperability, various compatible unexpected message sequences are provided. This includes mechanisms supporting the RUD and RUDI protocols that enable a simple back-off and retry approach (Figure 3-40) (3.4.4.7.1.1) as well as mechanisms that can be used with the ROD protocol to preserve ordering requirements (3.4.4.7.2.1 and 3.4.4.7.2.2). Further mechanisms are defined to support recovering from resource exhaustion with the ROD protocol while preserving ordering (Figure 3-41) (3.4.4.7.3). The buffering schemes do not change the wire behavior, other than the insertion of the list field; thus, they are not illustrated.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

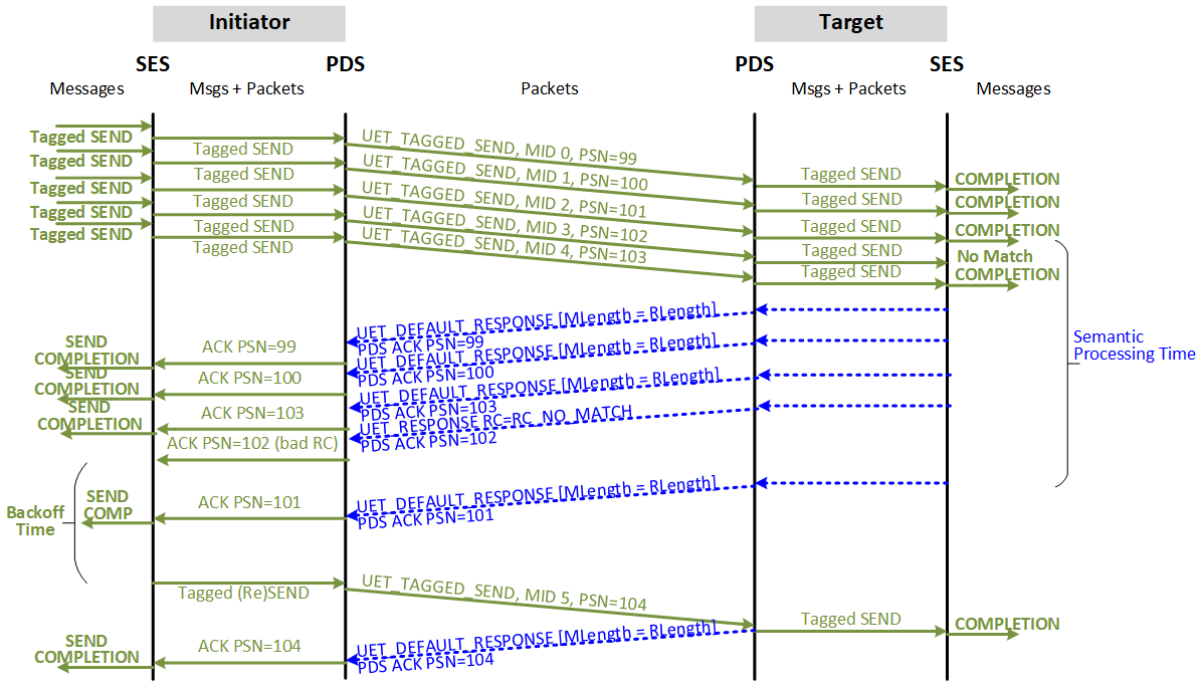


Figure 3-40 - Single Packet Messages using Backoff and Retry

Figure 3-40 illustrates a few features of the RUD transport when implementing a back-off and retry scheme. Single-packet messages are illustrated, and a message in the middle of the stream (message ID 3) does not find a match at the target. This returns a UET_RESPONSE with a **ses.return_code** of RC_NO_MATCH. This indicates to the initiator that the message was dropped at the target because no match was found. The initiator is then responsible for retrying the message at a later time. Note that when the message associated with MID=3 and PSN=102 receives the RC_NO_MATCH response, it is complete from the perspective of PDS and from the perspective of the message ID allocation. When this message is retransmitted after the back-off time, it MUST allocate a new PSN and MAY allocate a different message ID. Figure 3-40 also illustrates how semantic processing can have very different processing times; thus, the response to PSN 101 is somewhat delayed.

Implementation Note:
SES does not specify the backoff time or the implementation mechanisms for the backoff and retry. For example, the backoff and retry could be implemented as part of the libfabric provider implementation.

While Figure 3-40 illustrates only single-packet messages, multi-packet messages have similar characteristics. If the message associated with MID=3 and PSN=102 had been a multi-packet message, then the diagram would look much the same. While it is expected that the initial send of a message would complete before the back-off time expired, this is not a strict requirement. For example, a

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

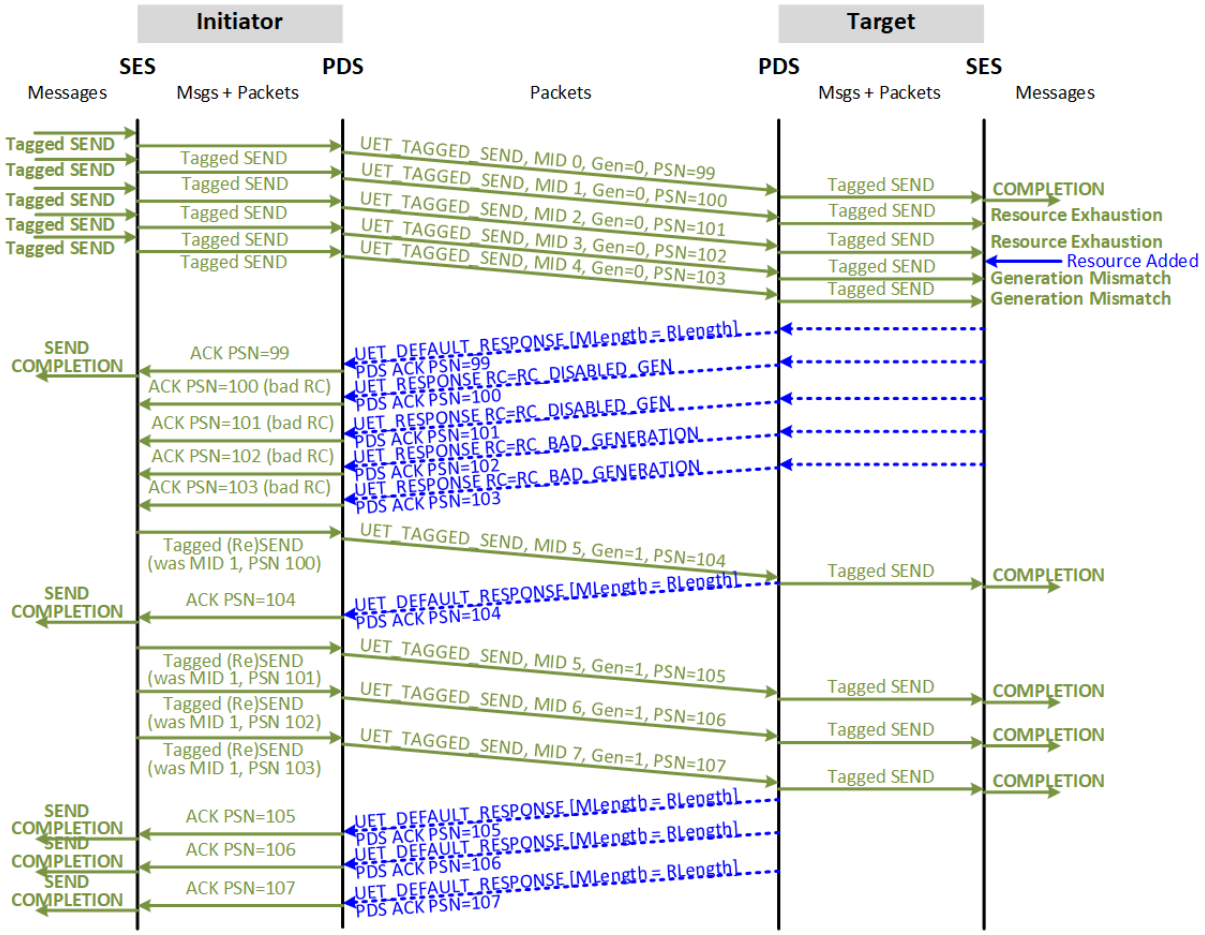


Figure 3-41 - Messages using Resource Index Generation

message using the RUD transport would be allowed to initiate an SES retransmit on a different PDC from the initial transmission. This allows for the PDC to be torn down and re-established between the first (unsuccessful) transmit of the message and the retransmit of the message. It also allows for both the original message and the retransmit of the message to be in progress concurrently.

For simplicity, one additional feature of the back-off and retry sequence is not illustrated. In the figure, a single back-off is shown with a successful retry. In practice, a back-off and retry sequence could have many failed attempts with successive retries of the message.

Like Figure 3-40, Figure 3-41 uses single-packet messages to illustrate the mechanics of a different unexpected message approach that applies specifically to ROD PDCs and uses the concept of a Resource Index generation (see Section 3.4.4.7.3.2). Because ROD PDCs require that ordering be maintained, a message that encounters resource exhaustion creates a challenge. If resources were replenished while subsequent messages were in flight, ordering could be violated if care was not taken. To preserve the ephemeral nature of PDCs, this resource exhaustion scenario is handled at the semantic sublayer rather than injecting state into the PDC itself to recover.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Figure 3-41 illustrates several aspects of the resource exhaustion protocol. First, once a resource is exhausted (e.g., an unexpected message buffer), the next message and all subsequent messages fail with an RC_DISABLED_GEN return code—until resources are added. Second, when resources are replenished, the generation number is atomically updated with the addition of resources. Third, messages that were already in flight encounter a generation mismatch and are dropped with an RC_BAD_GENERATION return code being generated. Fourth, to maintain ordering, the first dropped message is retransmitted with an updated generation number, but subsequent messages are not retransmitted until a successful return code is received. Once the first message is accepted by the target semantic sublayer, subsequent messages may be retransmitted with the new generation number.

Some features of this protocol are not illustrated. For example, a third FEP could send a message and encounter a generation mismatch. It would follow the same recovery procedure as illustrated. Note that, with multi-packet messages, it is still necessary to wait until the entire message has completed before starting the next message. This is because PDS can promote a UET_NO_RESPONSE to a UET_DEFAULT_RESPONSE as part of acknowledgement coalescing. It is not until the entire message has been acknowledged that the initiator can know for sure that the message was accepted at the target.

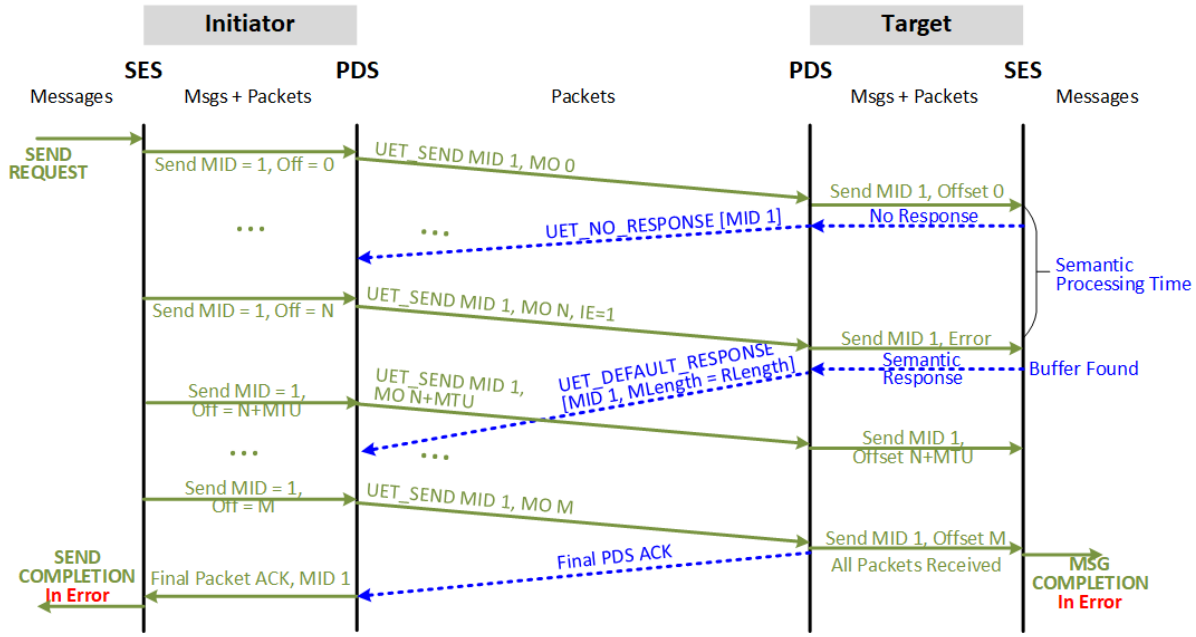


Figure 3-42 - Multi-Packet Request, Expected Message, with Initiator Error

3.4.5.7 Errors Indicated by the Initiator

Two types of error sequences are illustrated in Figure 3-42 and Figure 3-43. Errors that are detected at the target are delivered to the initiator using return codes in a response using guaranteed delivery. Errors detected at the initiator can be signaled to the target in two different ways. An error can be signaled by setting the initiator error (**ses.ie**) bit in the header as shown in Figure 3-42. This type of error signaling only marks a packet as bad and results in a message completion in error at the target. The

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

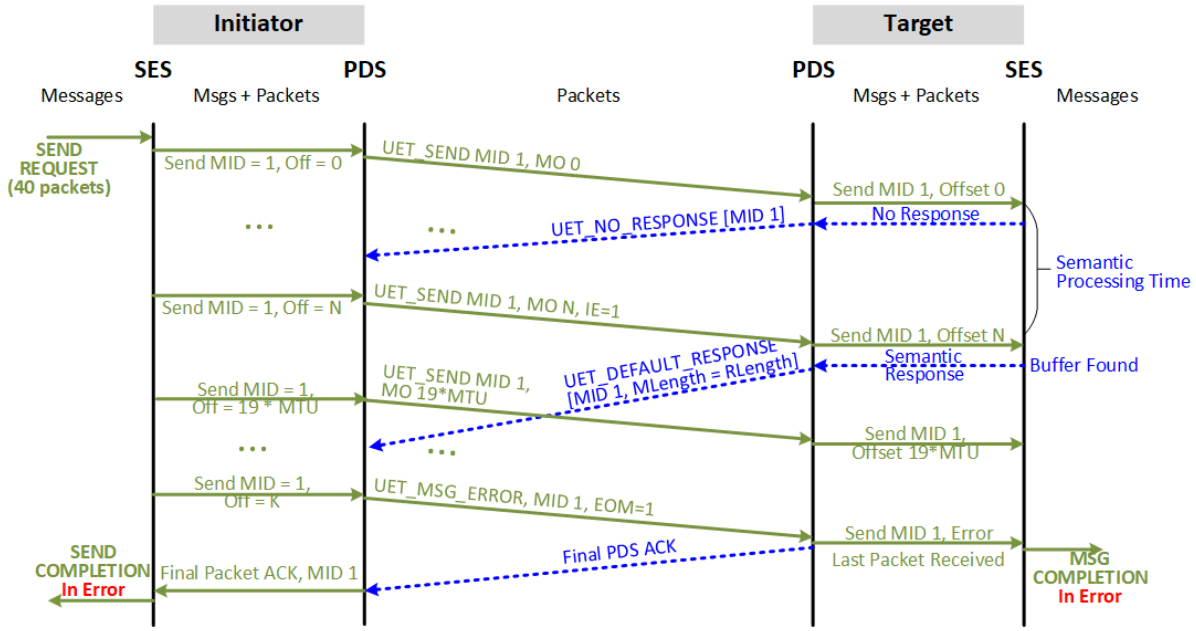


Figure 3-43 - Multi-Packet Request, Expected Message, with Message Error

illustration shows an error being delivered to SES for the packet that is in error. It is expected that the message completion at the initiator would also signal an error in this case.

In other cases, an error at the initiator may need to terminate the message that is in progress. In this case, the sequence in Figure 3-43 is used. Here, a UET_MSG_ERROR with **ses.eom** set is sent using the message ID of the message that is being terminated. As indicated in the figure, a message using this mechanism may be terminated before delivering all of the packets in the message.

3.4.5.8 Unified Local Transactions

At the semantic layer, Unified Local transactions are largely similar to their counterparts in AI Base, AI Full, and HPC; however, the supported operations are much narrower. Three examples are illustrated: UET_WRITE without delivery complete (Figure 3-44), UET_WRITE with delivery complete (Figure 3-45), and UET_READ (Figure 3-46). Error flows are the same as RMA error flows with error signaling in-band as

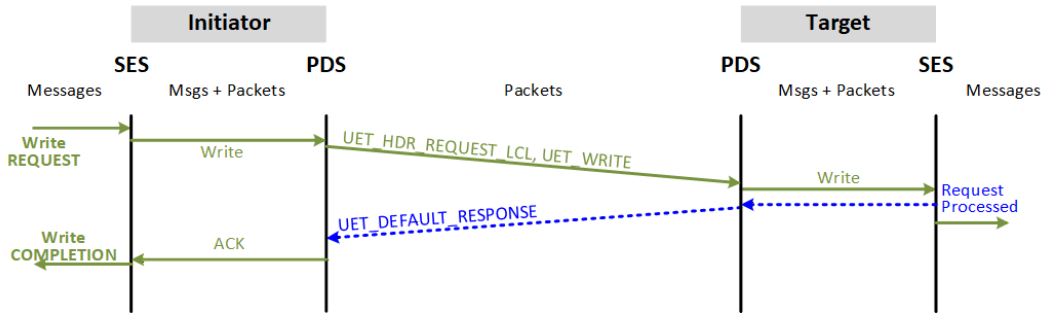


Figure 3-44 - Unified Local UET_WRITE with ses.dc=0

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

part of the completion (i.e., a non-RC_OK return code) or occasionally as part of the request (i.e., by setting ses.ie) and signaled with implementation-defined mechanisms at the endpoint.

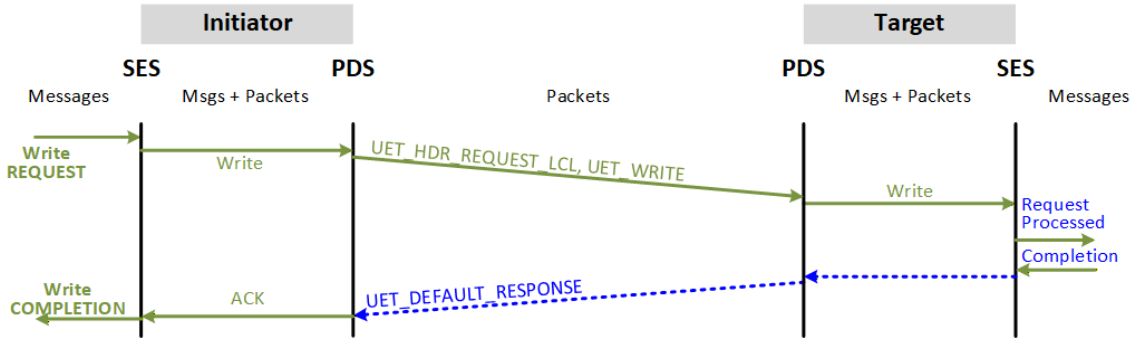


Figure 3-45 - Unified Local UET_WRITE with ses.dc=1

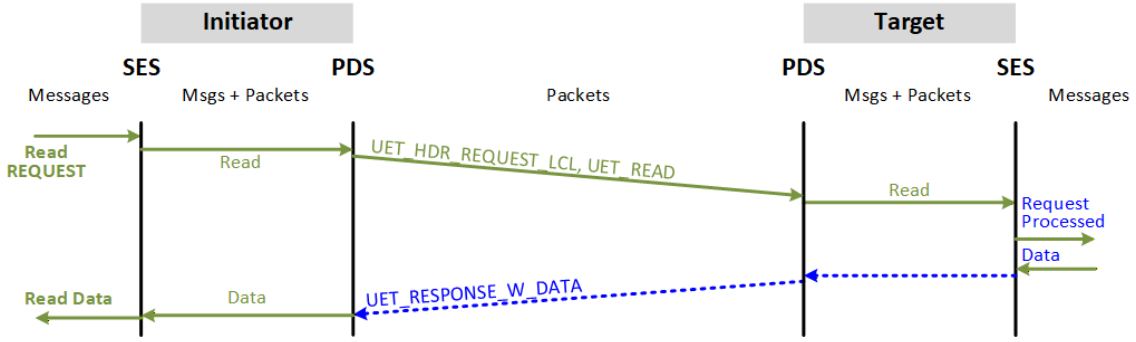


Figure 3-46 - Unified Local UET_READ

3.4.6 Error Handling

Errors in UET have three possible scopes: message level, PDS level, or device level. Error conditions fall into three broad categories: recoverable errors, unrecoverable errors, and informational errors. Within each category, errors are further subdivided into synchronous and asynchronous errors. The UET Semantic specification covers only errors that are communicated by the semantic sublayer and have a message-level scope. PDS level errors MAY have message-level scope; however, PDS-level errors are documented in the PDS specification. Device-level errors are beyond the scope of this specification.

3.4.6.1 Error Precedence

In general, UET uses a first-error model of error precedence. That is, the first error encountered is the error reported. If multiple errors are encountered for a message (i.e., different packets encounter different errors), it is the return code (RC) associated with the first error that is returned in the semantic response. If a single packet encounters multiple errors, the error contribution of that packet can be any of the return codes associated with an error the packet encountered. The first error encountered may or may not occur on the first packet (in packet sequence number order) of a message, since packets may be received out of order.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Informative Text:
A first-error model is used because errors can often propagate into more severe errors that are a direct result of the first error.

3.4.6.2 Error Scopes

A message-level error scope impacts one message. It may be recoverable, unrecoverable, or informational. If it is unrecoverable, a message-level error MUST impact only one message and its associated transaction.

A PDS-level error scope impacts an entire PDC. Unrecoverable PDS-level errors MUST terminate the PDC. All transactions in flight on a PDC that terminates MUST terminate in error to the application through libfabric. In other words, an unrecoverable PDS-level error becomes an unrecoverable message error for any transaction that has not already successfully completed.

Device-level and device-specific errors are beyond the scope of this document. Each vendor implementation is responsible for mapping device errors to message-level, PDS-level, or local node-level in scope. Only device-level errors and PDS-level errors may be asynchronous. That is, only device-level errors and PDS-level errors are ever delivered in any way other than as part of a message completion.

Informative Text:
Many PDS-level errors also cause erroneous completion of a message; however, some PDS-level errors may report only through out-of-band mechanisms (e.g., to a driver or to a management system).

3.4.6.3 Recoverable Errors

Recoverable errors may be retried by an intermediate portion of the communication stack (e.g., the libfabric provider). Retries of recoverable errors are expected to succeed eventually. The number of retry attempts MAY be limited by implementation-specific controls. If the number of retry attempts is limited, then messages that exceed the maximum number of retry attempts MUST terminate in error.

3.4.6.3.1 Handling of Recoverable Errors Detected at the Initiator

Recoverable errors detected at the initiator before a sequence number is allocated SHOULD be handled by the initiating FEP without allocating a sequence number. Recoverable errors detected at the initiator after allocating a sequence number MUST poison the FCS in the packet. An example of this type of error is a parity error on an intermediate buffer that the packet passes through. Recoverable errors detected at the initiator MUST NOT set the **ses.ie** bit.

3.4.6.3.2 Handling of Recoverable Errors Detected at the Target

Recoverable errors detected at the target all fall into the class of resource exhaustion errors. These scenarios are generally recoverable. Handling of the recoverable resource exhaustion occurrences MUST utilize the mechanisms described in sections 3.4.4.6.1 and 3.4.4.7. If an initiator receives a semantic response indicating a recoverable error from the target while a multi-packet message is still in progress,

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

the initiator MAY terminate the message early using the UET_MSG_ERROR mechanism as described in Section 3.4.6.4.1.

3.4.6.4 Unrecoverable Errors

Unrecoverable errors are reported as failed messages. Message-level failures are not fatal to the connection and are not fatal to the device. Device-fatal errors are beyond the scope of this specification. Errors that tear down a connection are defined as part of the PDS specification.

3.4.6.4.1 Handling of Unrecoverable Errors Detected at the Initiator

A variety of errors may be detected at the initiator that are fatal to the message. Such errors range from hardware failures (e.g., a PCI Express read completion timeout) to programming errors (e.g., a user process initiates a message with an address that cannot be translated) to other errors that make data unavailable (e.g., a process terminates after starting a message). Each of these cases is unrecoverable and may occur while a message is in flight. Two mechanisms are provided for signaling of uncorrectable initiator errors to the target.

The initiator error (**ses.ie**) bit in the semantic header is used to indicate that the header was properly constructed, but that the payload cannot be used. Implementations MAY use the **ses.ie** bit to indicate errors encountered at the initiator, but implementations are not required to have any class of error that causes the **ses.ie** bit to be set. Packets with the **ses.ie** bit set MUST NOT access memory at the target. Messages containing packets with the **ses.ie** bit set MUST use a non-RC_OK **ses.return_code**. If the initiator error is the first error, the return code MUST be RC_INITIATOR_ERROR. Messages containing packets with the **ses.ie** bit set MUST indicate an error at the target if a completion queue entry is provided or if a counter is used to indicate completion. Because the **ses.ie** bit indicates that the header was properly constructed, any completion delivered at the target MUST include the completion queue data (i.e., header data from the wire) if the **ses.hd** bit was set to indicate that such data is available.

The UET_MSG_ERROR opcode is used to terminate messages without transmitting all of the constituent packets. This mechanism MAY be used when an unrecoverable error is detected at the initiator, or as a general mechanism for early termination when the initiator receives a recoverable error indication from the target. The UET_MSG_ERROR opcode MAY be sent at any time—including the packet with **ses.som** set. When using a RUD PDC, the initiator MAY wait for all outstanding packets of the message to complete, then issue the final packet of the message with the UET_MSG_ERROR opcode; however, this is not required, and target implementations MUST NOT depend on it. All header fields within a UET_MSG_ERROR MUST be set to the value that the packet would have used if it had not been in error—except for the payload length. The payload length MAY use any size between 0 and the size the original packet would have taken. Only one packet per message may indicate UET_MSG_ERROR. A UET_MSG_ERROR packet MUST set **ses.eom**. Packets with the UET_MSG_ERROR opcode MUST NOT access memory at the target. Messages terminating with the UET_MSG_ERROR opcode MUST use a non-RC_OK return code. If UET_MSG_ERROR is the first error indication, the return code MUST be RC_INITIATOR_ERROR. Messages terminating with the UET_MSG_ERROR opcode MUST indicate an error at the target if a completion queue entry is provided or if a counter is used to indicate completion.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

UET_MSG_ERROR is not used with UET_HDR_REQUEST_LCL or UET_HDR_RESPONSE_DATA_LCL as part of the Unified Local profile.

Informative Text:
The PDS treats a packet using the UET_MSG_ERROR opcode just like any other packet. UET_MSG_ERROR does not change any packet reliability guarantees.

3.4.6.4.2 Handling of Unrecoverable Errors Detected at the Target

A broad set of unrecoverable errors may be detected at the target of a message. These errors range from programming exceptions (intentional or deliberate) such as permission violations (RC_PERM_VIOLATION) to uncorrectable hardware errors of an unspecified type (RC_UNCOR). Unrecoverable semantic errors are still limited in scope to a single transaction and do not impact the state of a PDC, for example. Uncorrectable errors deliver an appropriate return code using a first error model of error precedence, and that return code MUST be marked for guaranteed delivery.

Responses with data that return with an RC indicating an error MUST NOT write to memory at the initiator.

3.4.6.4.3 Unrecoverable Errors and Rendezvous

Rendezvous transactions are relatively unique in their composition within UET. There is a eager portion of the transaction originating from the initiator of the rendezvous using a UET_RENDEZVOUS_SEND (or TSEND) message and a read portion of the transaction originating from the target of the rendezvous. Unrecoverable errors in each sub-transaction are delivered with the completion notifications for those sub-transactions. For example, the eager portion of a rendezvous transaction delivers errors as if it were a send (or tagged send). The read portion of the rendezvous transaction delivers errors as if it were a read transaction. These errors are aggregated through the provider to be delivered to software.

3.4.6.5 Informational Errors

A handful of return codes are marked as informational. These errors (e.g., a floating-point underflow) may be safely ignored by some applications in some use cases. Like other errors, these errors MUST be provided back to the application (if requested). Informational errors deliver an appropriate return code using a first error model of error precedence among informational errors. Uncorrectable errors have a higher error precedence. An implementation that encounters an informational error and then a subsequent uncorrectable error MUST deliver the uncorrectable error. The return code MUST be marked for guaranteed delivery.

3.4.6.6 Return Codes

Return codes are defined for a variety of recoverable, unrecoverable, and informational errors. These codes are standardized for certain well-established error scenarios to provide additional information for debugging purposes. In addition, portions of the return code space are available for vendor-specific return codes. The return codes are enumerated in Table 3-25 along with a designation for their scope and whether they are recoverable (R), unrecoverable (U), or informational (I).

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

3.4.7 Enumerated Types Used in Headers

This section enumerates field definitions for various header types. This includes opcodes, return codes, and NACK types.

3.4.7.1 PDS Next Header Enumerations

The next header field used by the TSS header and the PDS header is enumerated in Table 3-22. The next header field indicates the size and format of the semantic header following the PDS header (see Section 3.5.11.2).

Informative Text:
Future iterations of the specification may include the ability to encapsulate and encrypt other types of traffic in the UET Security Protocol. This would increase the usage of the next header enumeration within the encryption header.

Table 3-22 - Next Header Enumeration

Mnemonic	Constant	Description
UET_HDR_NONE	0x00	No header follows this header.
UET_HDR_REQUEST_SMALL	0x01	The semantic header following the PDS header is the one illustrated in Figure 3-15.
UET_HDR_REQUEST_MEDIUM	0x02	The semantic header following the PDS header is the one illustrated in Figure 3-16.
UET_HDR_REQUEST_STD	0x03	The semantic header following the PDS header is the one illustrated in Figure 3-11, Figure 3-12, Figure 3-13, or Figure 3-14.
UET_HDR_RESPONSE	0x04	The semantic header following the PDS header is the one illustrated in Figure 3-25.
UET_HDR_RESPONSE_DATA	0x05	The semantic header following the PDS header is the one illustrated in Figure 3-26.
UET_HDR_RESPONSE_DATA_SMALL	0x06	The semantic header following the PDS header is the one illustrated in Figure 3-27.
UET_HDR_REQUEST_LCL	0x07	The semantic header following the PDS header is the one illustrated in Figure 3-17 and Figure 3-19
UET_HDR_RESPONSE_DATA_LCL	0x08	The semantic header following the PDS header is the one illustrated in Figure 3-28
UET_HDR_CP_ACK	0x09	The header following the PDS header is the one illustrated in Figure 3-87.
Reserved	0x0A-0x0E	
UET_HDR_VENDOR_DEFINED	0x0F	The semantic header following the PDS header is vendor-defined.

3.4.7.2 Opcode Enumerations

The opcode field in the header is enumerated for requests in Table 3-23 and for responses in Table 3-24. Each opcode space reserves encodings for vendor innovation.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Table 3-23 - Supported Request Messages (Opcode)

Mnemonic	Constant	Description
UET_NO_OP	0x00	SES message that performs no operation. MUST be a single-packet message. A FEP is not required to identify a buffer or completion queue at the target for a UET_NO_OP.
UET_WRITE	0x01	RMA write: Used to support *fi_write()*.
UET_READ	0x02	RMA read: Used to support *fi_read()*.
UET_ATOMIC	0x03	
UET_FETCHING_ATOMIC	0x04	Includes compare and swap.
UET_SEND	0x05	(non-matching) send message.
UET_RENDEZVOUS_SEND	0x06	Incorporated to allow send over ROD (for ordering) with bulk payload over RUD.
UET_DATAGRAM_SEND	0x07	Only valid when used with a UUD delivery mode.
UET_DEFERRABLE_SEND	0x08	A send message where the payload transfer may be deferred by the target.
UET_TAGGED_SEND	0x09	A tagged send message using match bits for buffer selection.
UET_RENDEZVOUS_TSEND	0x0A	A rendezvous version of the tagged send.
UET_DEFERRABLE_TSEND	0x0B	A deferrable version of the tagged send.
UET_DEFERRABLE_RTR	0x0C	A deferred send is ready to restart.
UET_TSEND_ATOMIC	0x0D	Atomic operations with tagged send addressing semantics.
UET_TSEND_FETCH_ATOMIC	0x0E	Fetching atomic operations (including compare and swap) using tagged send addressing semantics.
UET_MSG_ERROR	0x0F	Used to terminate an in-progress message ID. Can be sent as an early final packet of a message for an in-flight message that encounters an an initiator error or rejected by the target with a recoverable error. Invalid for the UUD delivery mode.
UET_SYNC_WRITE	0x10	Synchronized RMA write that is part of a synchronization group, including writes with immediate that terminate the group.
UET_SYNC_ATOMIC	0x11	Synchronized non-fetching atomic operation that terminates a synchronization group.
Reserved	0x12-0x2F	
UET_VENDOR_DEFINED[15]	0x30-0x3E	An encoding space to allow vendor extensions for experimentation and differentiation.
UET_OP_EXTENDED	0x3F	This opcode is reserved as an opcode space escape to an extended opcode location to be defined at a later time.

Table 3-24 - Supported Response Messages (Opcode)

Mnemonic	Constant	Description
UET_DEFAULT_RESPONSE	0x00	A default response, where the return code is RC_OK, the modified length is the requested length, and the list is UET_EXPECTED.
UET_RESPONSE	0x01	A response other than a default response.
UET_RESPONSE_W_DATA	0x02	A response carrying data.
UET_NO_RESPONSE	0x03	An indication that no semantic response is available at this time.
Reserved	0x05-0x2F	
UET_VENDOR_DEFINED[16]	0x30-0x3F	An encoding space to allow vendor extensions for experimentation and differentiation.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

3.4.7.3 Return Codes

SES can encounter a variety of errors—ranging from errors that may be only informational (I) (e.g., RC_AMO_FP_UNDERFLOW) to recoverable errors (R) (e.g., RC_BAD_GENERATION) to errors that are fatal to the message (U) (e.g., RC_AMO_UNSUPPORTED_OP) to errors that may require more drastic actions to correct (U) (e.g., RC_UNCOR). SES return codes are listed in Table 3-25.

Table 3-25 - Defined Semantic Return Codes

Mnemonic	Constant	Scope/ Recover	Description
RC_NULL	0x00	NA/NA	The RC status of this transaction is unknown.
RC_OK	0x01	Msg/NA	The transaction completed successfully at the target.
RC_BAD_GENERATION	0x02	Msg/R	The generation in the request did not match the generation at the target index.
RC_DISABLED	0x03	Msg/R	The targeted resource is disabled. Disabled resource has precedence over RC_NO_MATCH.
RC_DISABLED_GEN	0x04	Msg/R	The targeted resource is disabled and supports the index generation. Disabled resource has precedence over RC_NO_MATCH.
RC_NO_MATCH	0x05	Msg/R	The message could not be matched at the target and was dropped. This is returned for matching, nonmatching, and RMA transactions that fail to find a buffer.
RC_UNSUPPORTED_OP	0x06	Msg/U	Unsupported network message type.
RC_UNSUPPORTED_SIZE	0x07	Msg/U	The message was larger than the supported size.
RC_AT_INVALID	0x08	Msg/U	Invalid address translation context.
RC_AT_PERM	0x09	Msg/U	Address translation permission failure.
RC_AT_ATS_ERROR	0x0A	Msg/U	ATS translation request resulted in either unsupported request or completer abort.
RC_AT_NO_TRANS	0x0B	Msg/U	Unable to obtain a translation.
RC_AT_OUT_OF_RANGE	0x0C	Msg/U	Virtual address is out of range and unable to attempt translation.
RC_HOST_POISONED	0x0D	Msg/U	The host read (e.g. PCIe) indicated the access was poisoned.
RC_HOST_UNSUCCESS_CMPL	0x0E	Msg/U	The host read (e.g. PCIe) indicated an unsuccessful completion.
RC_AMO_UNSUPPORTED_OP	0x0F	Msg/U	Unsupported AMO message type.
RC_AMO_UNSUPPORTED_DT	0x10	Msg/U	Invalid datatype at the target.
RC_AMO_UNSUPPORTED_SIZE	0x11	Msg/U	The AMO operation was not an integral multiple of the datatype size.
RC_AMO_UNALIGNED	0x12	Msg/U	The AMO operation address was not natively aligned to the datatype size.
RC_AMO_FP_NAN	0x13	Msg/I	An AMO operation generated a NaN and signaling is enabled.
RC_AMO_FP_UNDERFLOW	0x14	Msg/I	An AMO operation generated an underflow and signaling is enabled.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Mnemonic	Constant	Scope/ Recover	Description
RC_AMO_FP_OVERFLOW	0x15	Msg/I	An AMO operation generated an overflow and signaling is enabled.
RC_AMO_FP_INEXACT	0x16	Msg/I	An AMO operation generated an inexact exception and signaling is enabled.
RC_PERM_VIOLATION	0x17	Msg/U	Message processing encountered a permissions violation (e.g., a mismatch in the JobID).
RC_OP_VIOLATION	0x18	Msg/U	An operation violation occurred. This includes a read attempting to access a buffered configured as write only, a write attempting to access a buffered configured as read only, or an atomic attempting to access a buffer that does not have both read and write permissions.
RC_BAD_INDEX	0x19	Msg/U	An unconfigured index was encountered.
RC_BAD_PID	0x1A	Msg/U	PID was not found at the target node (within the JobID for relative addressing, or at all for absolute addressing).
RC_BAD_JOB_ID	0x1B	Msg/U	JobID was not found at the target node.
RC_BAD_MKEY	0x1C	Msg/U	The specified memory key does not map to a buffer.
RC_BAD_ADDR	0x1D	Msg/U	Invalid address (not covered elsewhere) (e.g., an offset that extends beyond the length of the configured memory region).
RC_CANCELLED	0x1E	Msg/U	Response indicating the target cancelled an in-flight message.
RC_UNDELIVERABLE	0x1F	Msg/U	Message could not be delivered.
RC_UNCOR	0x20	Msg/U	An uncorrectable error was detected. The error is not likely to be rectified without corrective action.
RC_UNCOR_TRNSNT	0x21	Msg/R	An uncorrectable error was detected. The error is likely to be transient.
RC_TOO_LONG	0x22	Msg/U	The message was longer than the buffer it addressed. The target was configured to reject a message that was too long rather than truncate it.
RC_INITIATOR_ERROR	0x23	Msg/U	This RC echoes back the initiator error field from the incoming packet.
RC_DROPPED	0x24	Msg/R	Message dropped at the target for reasons other than those enumerated elsewhere.
RC_RESERVED	0x25-0x2F		
RC_VENDOR_DEFINED[0-7]	0x30-0x37	Vendor/ Vendor	
RC_RESERVED_WITH_DATA	0x38-0x3D		This opcode space is reserved for use with response-with-data messages. Responses without data MUST NOT infringe on this space, since they can easily define an extended RC space using the reserved bits in the header.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Mnemonic	Constant	Scope/ Recover	Description
RC_EXTENDED	0x3E		Used to extend the RC space. The format for this extension will be defined when it is needed.
RC_RESERVED	0x3F		

When buffered payload schemes are used for unexpected message handling, the initiator cannot determine whether the message was expected or unexpected at the target. The libfabric provider—often even the hardware—knows which of those things happened; thus, this information is returned in the semantic response. By returning this information to the initiator, it significantly simplifies the synchronous send implementation (*MPI_Ssend()*).

Table 3-26 - List Where the Message was Delivered

Mnemonic	Constant	Description
UET_EXPECTED	0x0	Message matched the expected list.
UET_OVERFLOW	0x1	An unexpected header was tracked for this message (3.4.4.7.2.1 or 3.4.4.7.2.2). Message payload may have been (3.4.4.7.2.2)
UET_VENDOR_DEFINED[0-1]	0x2-0x3	

3.4.7.4 Atomic Memory Operations (AMO) Enumerations

Table 3-27 enumerates the atomic opcode encodings, while Table 3-28 enumerates the encodings for datatypes. A broad set of operations and datatypes are defined based on the datatypes that are defined in libfabric, which was in turn inherited from the MPI definition. Two types beyond the MPI definition (two 16-bit floating-point types) are specifically included for AI. Space is reserved in both encodings to allow for vendor innovation. The UET_AMO_INVAL operation can be used in conjunction with the delivery complete bit to make the cached item globally observable.

Table 3-27 - Atomic Operation Opcodes

Mnemonic	Constant	Description
UET_AMO_MIN	0x00	Minimum: Target = MIN(Target, Initiator)
UET_AMO_MAX	0x01	Maximum: Target = MAX(Target, Initiator)
UET_AMO_SUM	0x02	Sum: Target = Target + Initiator
UET_AMO_DIFF	0x03	Diff: Target = Target – Initiator
UET_AMO_PROD	0x04	Product: Target = Target * Initiator
UET_AMO_LOR	0x05	Logical OR: Target = Target || Initiator
UET_AMO_LAND	0x06	Logical AND: Target = Target && Initiator
UET_AMO_BOR	0x07	Bitwise OR: Target = Target | Initiator
UET_AMO_BAND	0x08	Bitwise AND: Target = Target & Initiator
UET_AMO_LXOR	0x09	Logical XOR: Target = (Target && !Initiator) || (!Target && Initiator)
UET_AMO_BXOR	0x0A	Bitwise XOR: Target = Target ^ Initiator

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Mnemonic	Constant	Description
UET_AMO_READ	0x0B	Atomic read: Initiator = Target
UET_AMO_WRITE	0x0C	Atomic write: Target = Initiator
UET_AMO_CSWAP	0x0D	Compare and swap if equal
UET_AMO_CSWAP_NE	0x0E	Compare and swap if not equal
UET_AMO_CSWAP_LE	0x0F	Compare and swap if less than or equal
UET_AMO_CSWAP_LT	0x10	Compare and swap if less than
UET_AMO_CSWAP_GE	0x11	Compare and swap if greater than or equal
UET_AMO_CSWAP_GT	0x12	Compare and swap if greater than
UET_AMO_MSWAP	0x13	Swap masked bits: Target = (Target & Mask) ^ Initiator
UET_AMO_INVAL	0x14	If the location at the target is cached, invalidate the cache.
Reserved	0x15-0xDF	
UET_AMO_VENDOR[0-30]	0xE0-0xFE	Vendor-defined AMO
Reserved	0xFF	

Table 3-28 - Supported Atomic Datatypes

Mnemonic	Constant	Description
UET_TYPE_INT8	0x00	8-bit signed integer
UET_TYPE_UINT8	0x01	8-bit unsigned integer
UET_TYPE_INT16	0x02	16-bit signed integer
UET_TYPE_UINT16	0x03	16-bit unsigned integer
UET_TYPE_INT32	0x04	32-bit signed integer
UET_TYPE_UINT32	0x05	32-bit unsigned integer
UET_TYPE_INT64	0x06	64-bit signed integer
UET_TYPE_UINT64	0x07	64-bit unsigned integer
UET_TYPE_INT128	0x08	128-bit signed integer
UET_TYPE_UINT128	0x09	128-bit unsigned integer
UET_TYPE_FLOAT	0x0A	Single-precision floating-point value[1]
UET_TYPE_DOUBLE	0x0B	Double-precision floating-point value[1]
UET_TYPE_FLOAT_COMPLEX	0x0C	Pair of floats (real, imaginary)
UET_TYPE_DOUBLE_COMPLEX	0x0D	Pair of doubles (real, imaginary)
UET_TYPE_LONG_DOUBLE	0x0E	Double-extended precision floating-point value
UET_TYPE_LONG_DOUBLE_COMPLEX	0x0F	Pair of long doubles (real, imaginary)
UET_TYPE_BF16	0x10	16-bit floating-point value (bfloat 16)[2]
UET_TYPE_FP16	0x11	16-bit floating-point value (FP16 format)
UET_TYPE_FPE4M3	0x12	FP8 encoded as E4M3[3]
UET_TYPE_FPE5M2	0x13	FP8 encoded as E5M2[3]
Reserved	0x14-0xDF	
UET_TYPE_VENDOR[0-30]	0xE0-0xFE	Vendor-defined types
Reserved	0xFF	

Note:

1. As specified in IEEE 754-2019: <https://standards.ieee.org/ieee/754/6210/>
2. Industry specification (de facto): <intel.com/content/dam/develop/external/us/en/documents/bf16-hardware-numerics-definition-white-paper.pdf>

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Mnemonic	Constant	Description
3. As specified in the OCP 8-bit Floating Point Specification (OFP8), Revision 1.0: <https://www.opencompute.org/documents/ocp-8-bit-floating-point-specification-ofp8-revision-1-0-2023-12-01-pdf-1>		

Informative Text:
Typical programming interfaces (e.g., MPI, SHMEM) are silent on the details of handling NaN in floating-point arithmetic and define it as platform-specific.

Table 3-29 - AMO Semantic Control

Field	Size	Description
UET_AMO_CTRL_CACHEABLE	1	This AMO operation may be cached by the target device.
UET_AMO_CPU_COHERENT	1	This operation should be performed in a way that is coherent with CPU accesses.
Reserved	3	Reserved
VENDOR_DEFINED[0-2]	3	Vendor-defined encoding space.

The matrix of supported operation types versus datatype is shown in Table 3-30. The tables use ● to denote a supported operation on a given datatype and ○ to indicate an optional operation. Blank cells indicate an unsupported operation. Notable trends in the tables are that product is optional across all datatypes, and "hard" operations (e.g., 128-bit integer sum) are left as optional. Similarly, only the most basic operations are required on the complex floating-point formats. Nonsensical operations (e.g., logical operations on floating-point numbers) are excluded.

Table 3-30 - Valid Combinations of Operations and Datatypes (Alternative)

	Min	Max	Sum	Diff	Prod	LOR	LAND	BOR	BAND	LXOR	BXOR	READ	WRITE	CSWAP	CSWAP_NE	CSWAP_LE	CSWAP_LT	CSWAP_GE	CSWAP_GT	MSWAP
UET_TYPE_INT8	●	●	●	●	○	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
UET_TYPE_UINT8	●	●	●	●	○	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
UET_TYPE_INT16	●	●	●	●	○	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
UET_TYPE_UINT16	●	●	●	●	○	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
UET_TYPE_INT32	●	●	●	●	○	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
UET_TYPE_UINT32	●	●	●	●	○	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
UET_TYPE_INT64	●	●	●	●	○	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
UET_TYPE_UINT64	●	●	●	●	○	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
UET_TYPE_INT128	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
UET_TYPE_UINT128	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
UET_TYPE_FLOAT	○	○	○	○	○							○	○	○	○	○	○	○	○	
UET_TYPE_DOUBLE	○	○	○	○	○							○	○	○	○	○	○	○	○	

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

	Min	Max	Sum	Diff	Prod	LOR	LAND	BOR	BAND	LXOR	BXOR	READ	WRITE	CSWAP	CSWAP_NE	CSWAP_LE	CSWAP_LT	CSWAP_GE	CSWAP_GT	MSWAP
UET_TYPE_FLOAT_COMPLEX			○	○	○							○	○	○	○					
UET_TYPE_DOUBLE_COMPLEX			○	○	○							○	○	○	○					
UET_TYPE_LONG_DOUBLE	○	○	○	○	○							○	○	○	○	○	○	○	○	
UET_TYPE_LONG_DOUBLE_COMPLEX			○	○	○							○	○	○	○					
UET_TYPE_BF16	○	○	○	○	○							○	○	○	○	○	○	○	○	
UET_TYPE_FP16	○	○	○	○	○							○	○	○	○	○	○	○	○	

Informative Text:
Datatype use in AI applications is a rapidly evolving field. Many modern types (e.g., int4 and smaller, FP4 and FP6, etc.) are not useful for summation across multiple nodes due to the limited range of the representation. Similarly, some formats (e.g. MX formats) have variable definitions that are harder to map well into meaningful network transactions.

3.4.8 Device Expectations

To help ensure high-performance interoperability and a consistent level of isolation between processes and users, this section provides guidelines for device implementors.

3.4.8.1 Header Field Integrity Enforcement

The JobID MUST be protected within a privileged context that is at the highest level of privilege within the reachable network. This MAY be achieved by having the privileged context insert the JobID or by having the privileged context check the JobID at the initiator to confirm that the initiating process is allowed to use that JobID. One initiating process MAY be part of more than one JobID, and the number of JobIDs it is allowed to utilize is implementation-defined (a number greater than or equal to 1).

The source FA MUST be protected within a privileged context that is at the highest level of privilege withing the reachable network. This MAY be achieved by having the privileged context insert the source FA or by having the privileged context check the FA.

Implementation Note:
It may be difficult to achieve a high-performance implementation if the privileged context performing JobID and FA validation for each message is privileged software running on the host.

In virtualized environments, the JobID and source FA MUST be protected within a configuration state that is controllable only by the HyperVisor/VMM. In configurations that support out-of-band

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

provisioning systems, the JobID and source FA SHOULD be enforced by resources that are controlled by only the provisioning system to protect against HV/VMM escalation attacks.

3.4.8.2 SDI Assignment to Applications

The secure domain identifier (SDI) is used as part of the selection of an encryption key. Devices MUST limit how the SDI is utilized by applications. An application MUST NOT be allowed to use an SDI that is not assigned to it. A device MUST support the pairing of SDI and JobID at the initiator. That is, the device MUST provide a mechanism to guarantee that a specific SDI is used only with JobIDs that are allowed to use that SDI.

One JobID MAY be allowed to use more than one SDI. The number of SDIs that are usable by a single JobID is implementation-defined (a number greater than or equal to 1).

3.4.9 UE Transport Semantics: Memory Model

The AI Base, AI Full, and HPC profiles provide a memory model that is consistent with options that are available through the libfabric API. This memory model includes a set of minimal requirements that is expected for all implementations as well as how the memory model is controlled to be stronger for implementations that choose to do so.

The memory model supporting libfabric covers ordering, consistency, atomicity, global observability, and idempotency. The memory model for the Unified Local profile is largely implementation-dependent; supported options and capabilities are detailed in section 3.4.9.5.

3.4.9.1 Ordering

Ordering consists of two components: message ordering and data ordering. Both message ordering and data ordering are tightly tied to the underlying transport layer.

3.4.9.1.1 Message Ordering

Message ordering applies to the set of send operations (e.g., send, tagged send). The RUD and RUDI protocols do not make any message ordering guarantees.

Implementations of the ROD protocol MUST provide message processing ordering for a given “flow” between SES on an initiator FEP and SES on a target FEP. On ROD, at minimum, a flow is defined as the traffic from one PIDonFEP at the initiator to a specific Resource Index associated with a PIDonFEP at a target FEP. One or more flows may be mapped onto a single PDC.

SES MUST process messages in the order presented to it at a target FEP for a specific {JobID, PIDonFEP, Resource Index} tuple when using ROD.

3.4.9.1.2 Completion Ordering

If requested, implementations MUST deliver completion notifications at the target after all data has been placed. Messages marked with the delivery complete bit MUST NOT deliver the target completion until the corresponding global observability has been achieved (3.4.9.3).

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

The wire protocol does not provide direct support for fully ordered completions at the target. For example, rendezvous transactions do not use the same sequence number space for payload delivery as they do for initial request processing. As another example, the fully ordered completion order for deferrable sends is based on the order in which matching receives are provided to the implementation. This is beyond the scope of the transport definition.

Implementation Note:
The transport definition does not preclude libfabric provider implementations that deliver stronger completion ordering; however, such implementations are vendor-defined. One historical completion model—where send-after-write ordering is guaranteed—delivers the completion for a send transaction only after all preceding writes have completed. This requires leveraging the information from the sequence number space to determine when all prior writes have completed.

3.4.9.1.3 Data Ordering

Data ordering refers to the order in which data for a given target location is placed. Specifically, if two messages access a single memory address from a single source, data ordering describes the order in which the accesses become globally observable. Ordering between bytes in a given message is not defined. That is, the last byte of a message MAY become globally observable in host memory before the first byte. Similarly, the last byte of a message MAY be the last byte of a message deposited in memory. Implementations MAY provide stronger ordering semantics, but those are beyond the scope of this specification and are not likely to be portable.

The RUD and RUDI protocols do not make any data ordering guarantees.

Implementations of the ROD protocol that support atomic operations MAY provide an option to achieve data ordering (RAW, WAW, WAR) for ATOMIC and FETCHING_ATOMIC operations on data that is 16 bytes in size or less. The granularity of such ordering is limited to the size of a single element of the size of the datatype. Implementations MAY combine atomic operations on the target FEP so that not every update is globally observable at the target.

Informative Text:
The node and bus architecture that a FEP is attached to are beyond the scope of this specification; however, the data ordering model is purposefully relaxed based on known limitations in contemporary (ca. 2025) architectures and implementations.

3.4.9.2 Consistency and Atomicity

If atomic operations are supported, atomicity MUST be guaranteed at the granularity of a single element of the size of the datatype.

If atomic operations are supported, concurrent atomic operations MUST be consistent at the target at the granularity of a single element of the size of the datatype. Concurrent operations are defined as two operations targeting the same {FA, JobID, PIDonFEP, Resource Index, Memory Key/Match Bits, Offset}

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

tuple where one or more operations start concurrently with or after one or more other operations that have not completed. Concurrent operations can be initiated from the same FEP or from more than one FEP. This includes consistency between ATOMIC and FETCHING_ATOMIC operations. This includes consistency between UET_AMO_ WRITE, UET_AMO_ READ, and other atomic operations.

Implementation Note:
Concurrent access is scoped to a {FA, JobID, PIDonFEP, Resource Index, Memory Key/Match Bits, Offset} basis because aliasing to a single physical memory location behind those constructs may not be detectable by the FEP (e.g., when a host is using an IOMMU).

If atomic operations are supported, implementations MAY support consistency between concurrent send, write, read, and atomic operations; however, it is not required. Users needing write or read semantics that are consistent with atomic operations MUST use the atomic versions of those operations. Concurrent operations are defined as two operations that have not completed back to the initiator.

Informative Text:
The classical definition of consistency is used. That is, consistency is defined as having the operations appear as if they executed in some order. It is a common limitation in network APIs that consistency between send and an atomic to the same memory location is not guaranteed. The UET atomic operations include atomic read and atomic write operations to allow upper-level APIs to achieve consistency, where needed.

Implementations MAY support stronger atomicity. The UET_AMO_CPU_COHERENT semantic control allows peers to request atomic operations that are coherent with (and atomic relative to) CPU accesses. If the requested semantics cannot be provided, the target FEP returns RC_AMO_UNSUPPORTED_OP.

Implementation Note:
There are many ways to provide coherent atomics relative to the CPU. As an example, an implementation could utilize PCI Express atomic operations to implement the corresponding network atomic operations.

3.4.9.3 Global Observability

The default for global observability is that global observability is not guaranteed. A FEP MUST NOT indicate that a message has completed until all the packets have been acknowledged by PDS. In addition, the FEP MUST wait until a semantic response is received before indicating that the message is complete. This is sufficient for implementing the “transmit complete” semantic in libfabric.

An implementation MAY support stronger global observability. If the DC (delivery complete) bit is set in a message, an implementation MAY indicate with a semantic failure that the transaction is not supported. If an implementation does not indicate that the transaction is not supported, then the implementation MUST defer the semantic response until it can guarantee that the data is globally observable at the target.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Informative Text:
Global observability can be implemented by issuing a flushing read on PCI Express after issuing a data write.

While target completion delivery falls largely outside the scope of the semantic specification, libfabric compliance suggests that target completions should consistently indicate global observability. For example, in many PCI Express hierarchies, this can be accomplished merely by setting RO=0 for completion notifications from the FEP.

3.4.9.4 Idempotency

Operations performed using the ROD or RUD protocol MUST appear in memory at the target and at the initiator as if they were performed exactly once. This creates specific requirements for FETCHING_ATOMIC operations, which return the "old" data value and then modify the memory at the target. If a target implements FETCHING_ATOMIC operations, the target MUST buffer the old data to handle the case where the response is lost. This is true for the ROD and RUD protocols.

3.4.9.5 Memory Model for the Unified Local Profile

Much like the libfabric semantics, the transport for the Unified Local profile is coupled to the semantics of the transport layer. Transactions across the wire are ordered for ROD or RODL delivery modes and unordered for the RUD delivery modes. The data ordering model is defined by the implementation.

The transport for the Unified Local profile follows the same consistency and atomicity guarantees described in Section 3.4.9.2; however, the definition of concurrent operations is narrowed to two operations targeting the same {FA, JobID, Offset}. Similarly, the global observability semantics leverage the DC bit (delivery complete) to (optionally) offer a global observability guarantee. The transport for the Unified Local profile is defined only over the RODL delivery mode; thus, it guarantees idempotency.

3.4.9.6 Synchronized Operation

Synchronized operations add only nominal ordering requirements beyond the ordering requirements discussed in 3.4.9.1. Specifically, synchronized operations add no additional ordering expectations to the delivery of payload between two UET_SYNC_WRITE messages—even if one of the two UET_SYNC_WRITE messages is a synchronized completion operation (i.e., has **ses.hd** set).

Synchronized operations do not change the ordering of completion notifications with respect to other completion notifications. The completion notification (if requested) for a UET_SYNC_WRITE with **ses.hd** set MUST follow the payload delivery for every UET_SYNC_WRITE message in the **ses.sync_group**—including the payload included in the synchronized completing operation. That completion notification has no defined ordering with respect to other completion notifications.

Synchronized completing operations leveraging UET_SYNC_ATOMIC MUST perform the atomic operation after the payload delivery for every UET_SYNC_WRITE message in the sync group.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

3.4.9.6.1 Sync Group Error Cases

There are several error cases that can occur when using sync groups. Table 3-31 describes the behavior under various conditions.

Table 3-31 - Synchronized Operation Error Cases

Case	Error Case	Target Behavior
1	One or more messages of the sync group failed at target due to error.	The failure of one message does not impact the other messages of the sync group. The failed message is still counted as part of the sync group count.
2	Fewer than (X + 1) messages are received by the target	Target ages out incomplete sync groups. If a completing operation message was received it is not completed or executed.
3	Completing operation is received on a sync group where the current count is greater than **ses.sync_count**.	Operation fails with SES Error (RC _OP_VIOLATION)
4	Initiator sends X write messages, completing the operation with **ses.sync_count** = (X + 1), then Y write messages on same sync group without waiting for appropriate completions.	All X operations are not impacted. All Y operations fail with SES error (RC_OP_VIOLATION).
5	Initiator sends multiple completing operations on same sync group.	First operation will be processed normally. Subsequent completing operations will fail with RC_OP_VIOLATION.

3.4.10 Mapping of *CCL Send/Receive to Proposed Semantics [Informative]

The text in this section is informative and not normative. The section describes two approaches for mapping *CCL send/receive APIs to the proposed libfabric/UET semantics, where the notation *CCL is used to denote a generic Collective Communication Library (CCL). Both approaches aim to maintain the current *CCL buffer usage semantics in conjunction with UET's reliable connectionless delivery modes. Other approaches beyond those described in this section are also possible.

The section assumes *CCL APIs of the form shown below:

- *ccl_send: API to send a message with parameters that identify:
  - The buffer containing the message and its size.
  - The peer rank that the message is destined to.
  - A communication context for the message.
- *ccl_recv: API to post a receive buffer with parameters that identify:
  - The address and size of the buffer.
  - The peer rank that the buffer is being posted for.
  - A communication context for the message.

In one of the approaches described, the *CCL plugin that implements the *ccl_send/*ccl_recv APIs uses the libfabric *fi_tagged()* APIs, while the other approach uses the libfabric *fi_rma()* APIs.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Mapping *CCL send/receive concepts to the scale-up semantics is not specified.

3.4.10.1 Tag-Based *CCL Mapping

The tag-based *CCL mapping leverages the libfabric *fi_tagged()* API beneath the untagged send and receive operations within a typical *CCL API. The *CCL user API is unchanged, but the tagged API implementation allows the ordered send/receive pair to use an unordered protocol underneath for better network efficiency. To simplify the underlying implementation, libfabric will be extended to allow the libfabric user to specify that only “exact match” operations will be supported for a given Resource Index. In this model, libfabric is initialized with FI_ORDER_NONE. This allows the implementation to choose a RUD PDC.

A *CCL send must match the corresponding *CCL receive based on the order in which they were issued. To achieve this while using the RUD protocol, the *CCL implementation maintains the following state per communication context:

1. A send message sequence number (MSN) per-peer rank
2. A receive MSN per-peer rank

The tag passed through the *fi_tagged()* API is a communication context identifier concatenated with the appropriate MSN for that peer. For example, an *fi_tsend()* uses the send MSN combined with the communication context identifier. The source rank is part of the local endpoint and is placed in the initiator field of the packet by the provider. The *dest_addr* field of the *fi_tsend()* comes from the destination in the *CCL send. Each time a *CCL send is called, the send message MSN for the corresponding rank is incremented. Correspondingly, the *fi_trecv()* uses the receive MSN combined with the communication context identifier. The source rank is used to populate the *src_addr* field. Each time a *CCL receiver is called, the receive MSN for the corresponding rank is incremented.

The use of the MSN tables is illustrated in Figure 3-47.

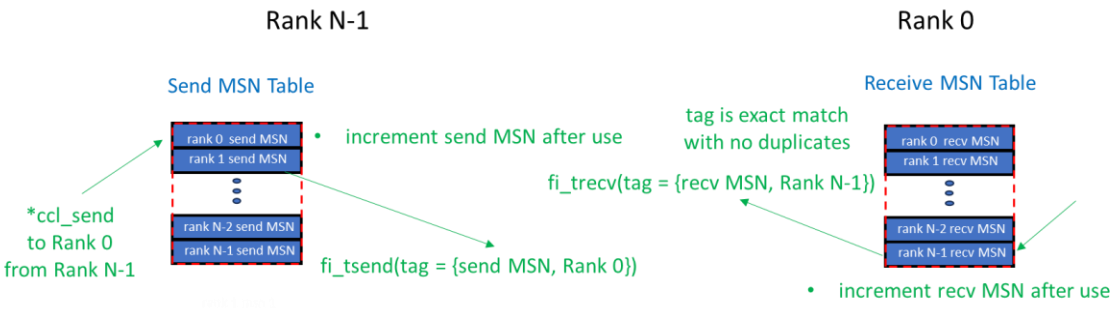


Figure 3-47 - Use of MSN Tables

A sequence diagram for tag-based *CCL send and receive operations is shown in Figure 3-48.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

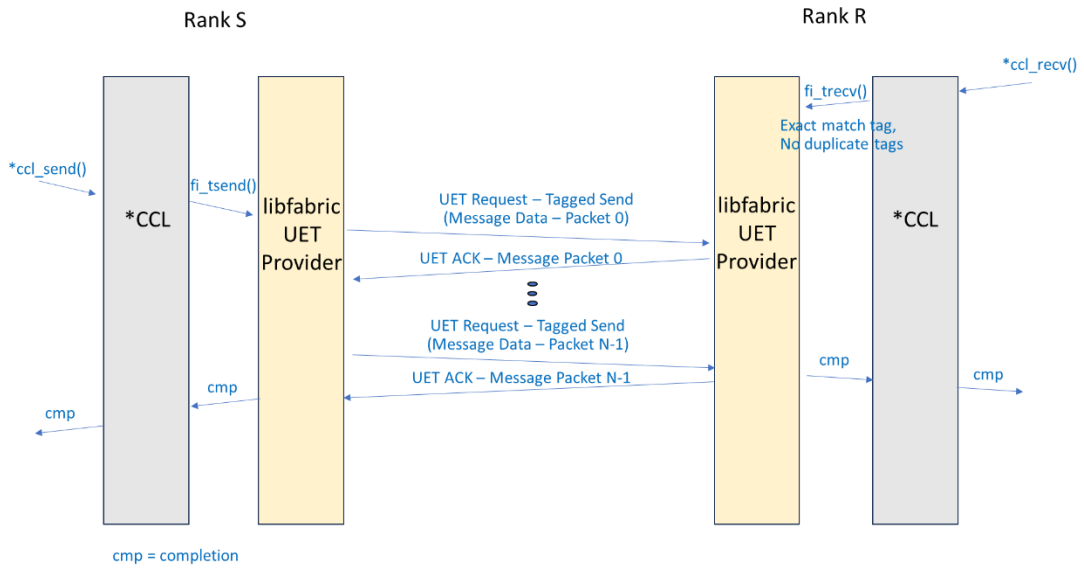


Figure 3-48 - Tag-Based Sequence for *CCL Send and Receive

The sequence above illustrates the expected message sequence and does not distinguish the specific type of tagged send message. The rendezvous tagged send (3.4.4.5) may also be used. Unexpected messages can be particularly challenging in tagged environments; however, the use of an unordered PDC (a RUD PDC) simplifies unexpected message handling. The implementation may use any of the unexpected message handling mechanisms outlined in Section 3.4.4.6.1.

3.4.10.2 RMA-Based *CCL Mapping

The RMA-based mapping requires RMA write and RMA write with immediate functionality, both of which are provided by UET.

The RMA-based mapping uses a rendezvous queue data structure that is illustrated in Figure 3-49.

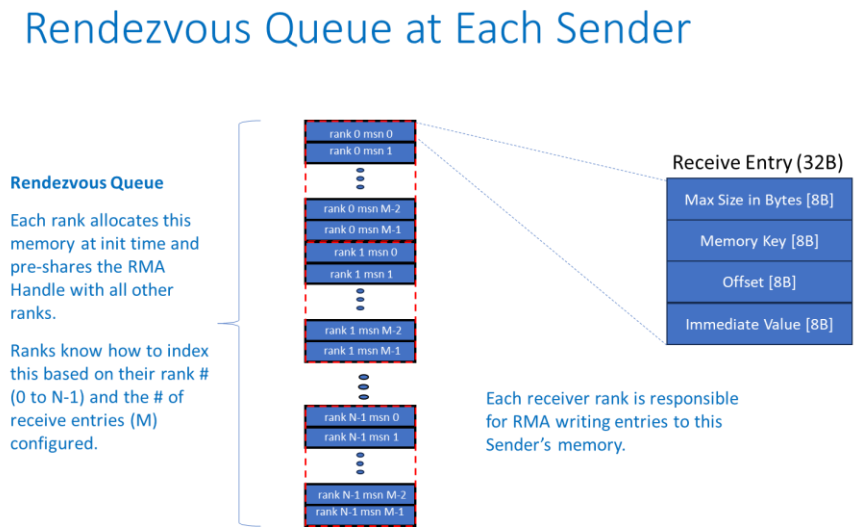


Figure 3-49 - Rendezvous Queue Data Structure

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

The rendezvous queue works as follows:

- The rendezvous queue is a set of N descriptor rings, one per rank.
  - Each ring contains entries for sending to a particular rank.
  - The entries are produced by receivers and read by senders in FIFO order.
  - The sender maintains a tail index for each ring that is used for consuming entries.
  - The receiver maintains a head index for each rank that is used for producing entries.
  - The receiver is also responsible for managing the case where a ring is full.
- When a *CCL receiver of rank R posts a receive buffer for a given sender of rank S:
  - The receiver writes a receive entry for the buffer to the rendezvous queue of rank S.
  - The entry is written to the ring associated with rank R using the head index for rank S that is maintained at rank R.
- When a *ccl_send API is called by rank S to send to rank R:
  - The sender uses the ring associated with rank R.
  - The sender accesses the ring entry at the tail index.
  - If there is a valid receive entry at the tail index, thenthe sender uses the information in the entry to write the message data to the receiver's buffer.
  - If there is not a valid receive entry at the tail index, then the sender waits for valid receive entry to be written at the tail index.

A sequence diagram for write-based *CCL send and receive operations is shown in Figure 3-50.

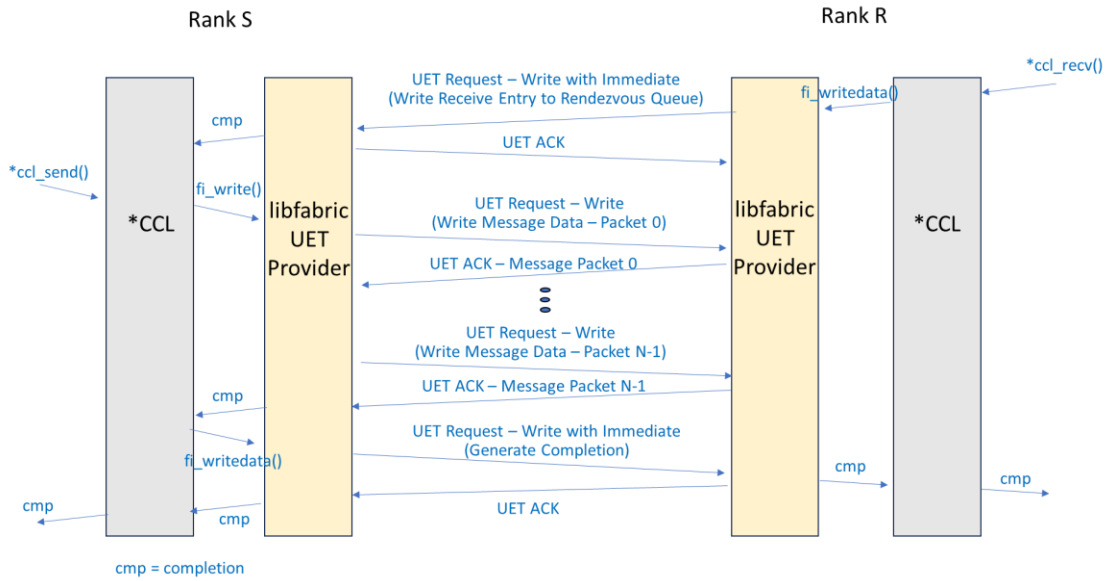


Figure 3-50 - Write-Based Sequence for *CCL Send and Receive

The write-based sequence proceeds as follows:

- An application call is made to the **ccl_recv()* API by rank R to post a receive buffer for rank S.
- The *CCL at rank R consults the rendezvous queue data structures and calls the libfabric *fi_writedata()* API to generate an RMA write with immediate transaction for the purpose of writing a receive entry for the posted buffer to rank S.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

- The completion for the write immediate informs the *CCL at rank S that the receive entry is available, which is useful when there is a pending send transaction for rank R.
- An application call is subsequently made to the **ccl_send()* API by rank S to send a message to rank R.
- The *CCL at rank S consults the rendezvous queue data structures and calls the libfabric *fi_write()* API to generate an RMA write for the purpose of writing the message to the buffer posted by rank R.
- The completion for the *fi_write()* is assumed to inform the *CCL at rank S that the message has been delivered to the posted buffer and is globally observable.
- The *CCL at rank S then calls the libfabric *fi_writedata()* API to generate an RMA write with immediate for the purpose of generating a completion to the *CCL at rank R.
- The *CCL then delivers completions to the application for the **ccl_recv()* at rank R and the **ccl_send()* at rank S.

The above sequence is for the case when a valid receive entry is available at the time **ccl_send()* is called. If a valid receive entry is not available, the *CCL saves the request in a send pending queue. When the completion of an RMA write immediate indicates that a receive entry has been posted for rank R, the send pending queue is checked, and if there is a pending send request for rank R, that request is removed from the queue and serviced using the posted receive entry. In this manner, the proposal does not require receiver not ready (RNR) functionality from UET.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

3.5 Packet Delivery Sublayer (PDS)

The packet delivery sublayer is the part of the Ultra Ethernet Transport (UET) protocol responsible for delivering packets over IP/Ethernet networks. The packet delivery sublayer offers reliability and ordering capabilities. PDS exists as a sublayer in the UET, between the semantic sublayer (SES) and the congestion management sublayer (CMS), as illustrated in Figure 3-51.

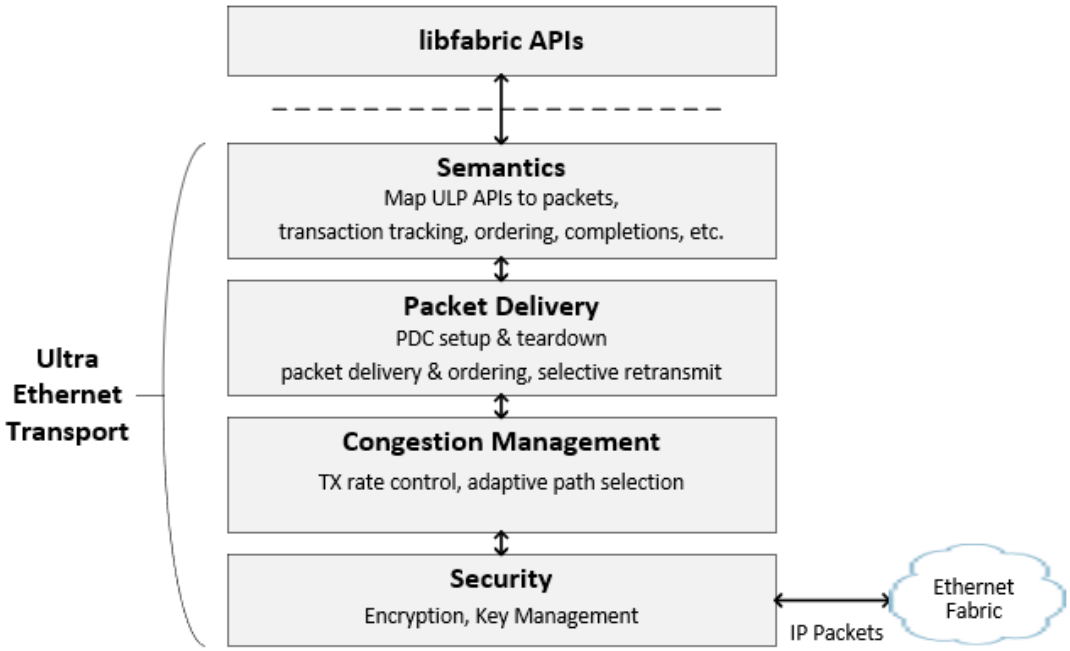


Figure 3-51 - PDS High-Level Architecture Diagram

3.5.1 PDS Terminology

Table 3-32 contains a summary of PDS terminology. The terms defined here are used in this section and other Transport layer sections. Some of these terms are previously defined in the UE Specification frontmatter, Section 1.2.2, but are listed here again to provide additional context.

Table 3-32 - PDS Terminology

Term	Description
ACK	Acknowledgement
CC	Congestion control (aka congestion management).
CCC	Congestion control context • Used to control traffic congestion in one direction for RUD/ROD.
Clear	Use to acknowledge an acknowledgement (ACK) when the ACK requires guaranteed delivery; that is, the ACK is carrying semantic state that must be delivered—such as error information or return data.
CP	A control packet type used for RUD, ROD, and RODL delivery modes.
Deferrable Send	Send with option for the target to indicate the receive buffer is not yet available with a NACK and later send a restart transmission request (RTR) packet when the receive buffer is available.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Term	Description
DEF_RESP	Default response, shorthand for UET_DEFAULT_RESPONSE; this SES Response format is used for ACKs when multiple PSNs are coalesced into a single ACK and if PDS is re-creating an SES Response that was not guaranteed delivery.
Destination	FEP to which a transmitted packet is sent.
DPDCID	Destination PDCID • PDCID assigned by peer FEP that is the destination of packet (i.e., the FEP on the other end of the PDC).
Duplicate packet	Refers to a packet that is received at the destination a second time, e.g., if the source retransmits a packet that was successfully received. For RUD/ROD/RODL, duplicate packets are not passed to SES and may be acknowledged.
Entropy value (EV)	The entropy used to load balance packets depending upon the encapsulation used. For UE packets in native IP encapsulation the EV is taken from the **pds.entropy** header field. For UE packets encapsulated in UDP the EV is taken from the **udp.src_port** field.
Forward Direction	The direction used by PDS request packets from initiator FEP to target FEP. ACKs for PDS Request packets in the forward direction flow from target FEP to initiator FEP. See Figure 3-52.
Forward PSN	PSN assigned to packets (initiator requests) on the forward direction; may be carried with return data on return path.
GTD_DEL	Guaranteed delivery; Tthis identifies SES responses that are guaranteed to be delivered from target to initiator—that is, the response is stateful. Example stateful responses are error events, use of unexpected list, and fetching atomic responses (data).
Initiator	FEP that initiates establishment of a PDC by sending a packet to another FEP.
IPDCID	Initiator PDCID • Assigned by FEP that initiates PDC establishment.
MID	Message identifier: Assigned by SES, treated as opaque by PDS; this acronym is used to clarify ordering implications in the packet sequences figures. • A message is a group of one or more packets using the same message ID.
MO	Message offset: The packet number within an SES message; this acronym is used to clarify ordering implications in the packet sequences figures.
MP_Range	Maximum PSN range: Defines the maximum number of packets (PSNs) at a destination that can being tracked on the PDC based on available resources. This value is carried in compressed format for in PDS ACKs using the **pds.mpr** field.
NACK	Negative acknowledgement
Highest PSN	Refers to the highest PSN value, noting that when the PSN space wraps, the highest PSN may have a lower numerical value than older PSNs.
OOR	Out of Resources
PDC	Packet delivery context including both a forward direction and return direction. The PDC is ephemeral with dynamic establishment and close.
PDS ACK	Generated by PDS and transmitted over the Ethernet fabric to a PDS on another FEP; these carry SES responses that are delivered to SES at destination.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Term	Description
PDCID	Packet delivery context identifier.
PDS	Packet delivery sublayer.
PDS Request	Generated by PDS and transmitted over the Ethernet fabric to a PDS on another FEP; these PDS requests carry an SES Request or SES Response with data (in the return direction) to be delivered to the destination SES.
PSN	Packet sequence number.
Return Data	Data sent from the target to the initiator in response to an SES Request (i.e., read response). This may be carried in a PDS ACK in the forward direction or in a PDS Request in the return direction. Generated by SES and passed to PDS to request delivery of a packet carrying read response data on a specific RUD, ROD, or RODL PDC in the return direction.
Return Direction	The direction used by PDS Request packets from target FEP to initiator FEP. ACKs for PDS Request packets in the return direction flow from initiator FEP to target FEP. See Figure 3-52.
Return PSN	PSN assigned to packets (target return data) on the return direction.
RTR	Restart transmission request—this SES Request is used to restart a deferrable send that was deferred. It uses a separate PDC, as the target for the original deferrable send acts as an initiator to send the RTR.
ROD	Reliable ordered delivery.
RODL	Reliable ordered delivery for local networks.
RTO	Retransmission time out—event when a timer expires before an ACK or NACK is received for a transmitted PDS Request or CP
RUD	Reliable unordered delivery.
RUDI	Reliable unordered delivery of idempotent operations.
SACK	Selective acknowledgement.
SES	Semantic sublayer.
SES Request	Generated by SES and passed to PDS to request delivery of a packet with specified ordering and reliability service; only the initiator may issue these. These are relayed to SES at the destination using a PDS Request.
SES Response	Generated by SES at receiver and passed to PDS in response to receipt of an SES Request—these may be carried in PDS ACKs or PDS requests. Refer to the description following Figure 3-52 for more information.
SES RTR	Deferrable send ‘restart transmission request’; refer to semantic section.
Set / Clear	When referring to fields, set means one, 0b’1, and clear means zero, 0b’0.
Source	FEP that transmits the packet.
SPDCID	Source PDCID • PDCID assigned by FEP that transmitted the packet (i.e., the locally assigned PDCID for the PDC)
Target	FEP that is the destination for a packet from another FEP; establishes a PDC in response to the initiator.
TC	Traffic class.
TPDCID	Target PDCID • Assigned by FEP that is the target of a PDC establishment request.
UUD	Unreliable unordered delivery
Note: • Request—with a capital ‘R’—is used to refer to refer to SES and PDS header types (SES Request). • request—with a lower case ‘r’—is used to refer to SES and PDS actions (SES request to send a packet).	

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

3.5.2 Illustration of PDS Terms

Initiator and target terms are related to a specific PDC. All FEPs may operate simultaneously as both initiator and as target.

Figure 3-52 uses the following basic sequence with numbers in the ovals matching the following:

1. SES Request generated by FEP-A SES.
2. Initiator generated PDS Request carries SES Request to target.
3. SES Request passed to FEP-B SES.
4. SES Response to this request generated by FEP-B SES.
5. Target generated PDS ACK carries SES Response to Initiator → this ACK is part of forward direction.
6. SES Response passed to FEP-A SES.

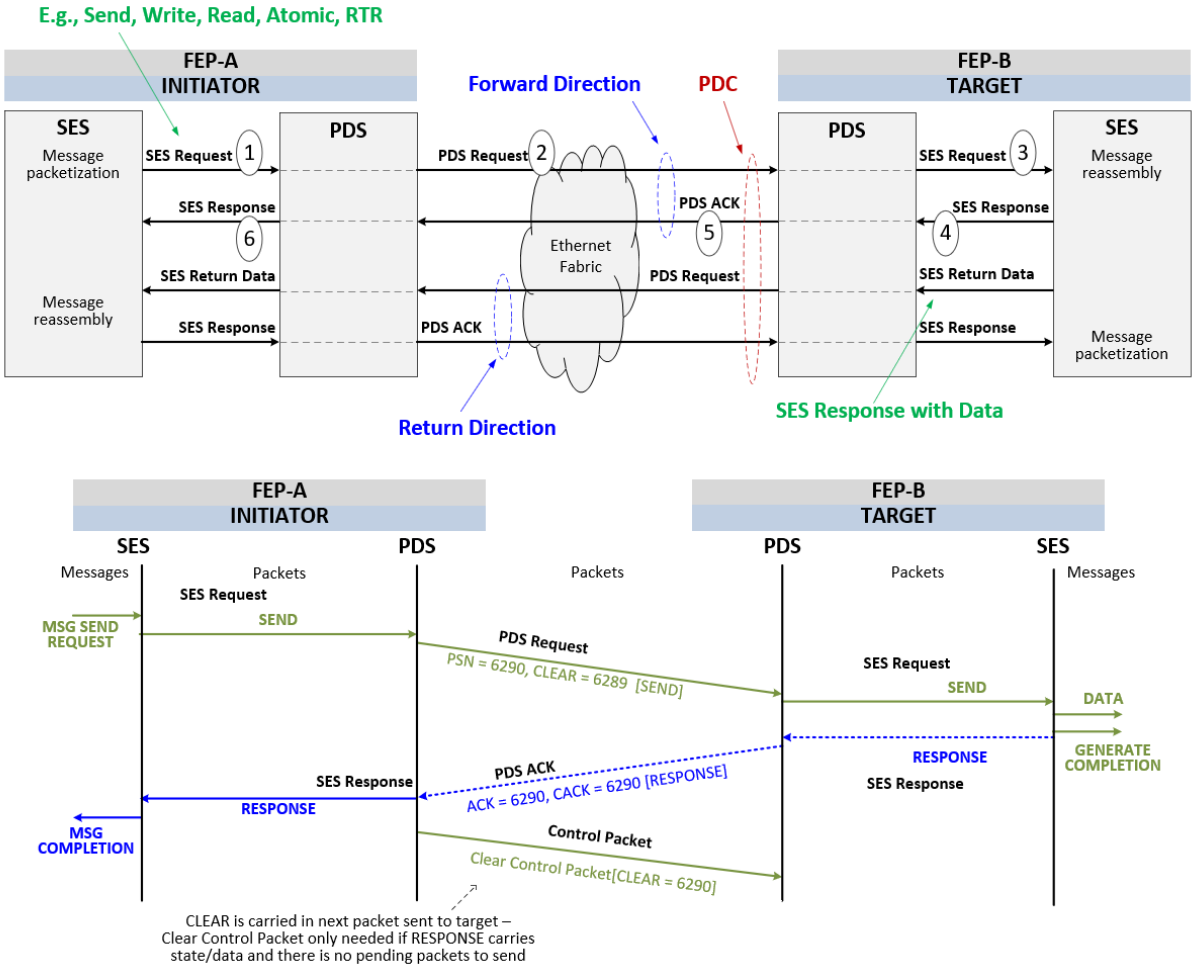


Figure 3-52 - Illustrated PDS Terms

The lower arrows depict the same sequence in the return direction for read responses. Congestion control is not applied to PDS ACKs. The return direction is used to carry large read response packets—

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

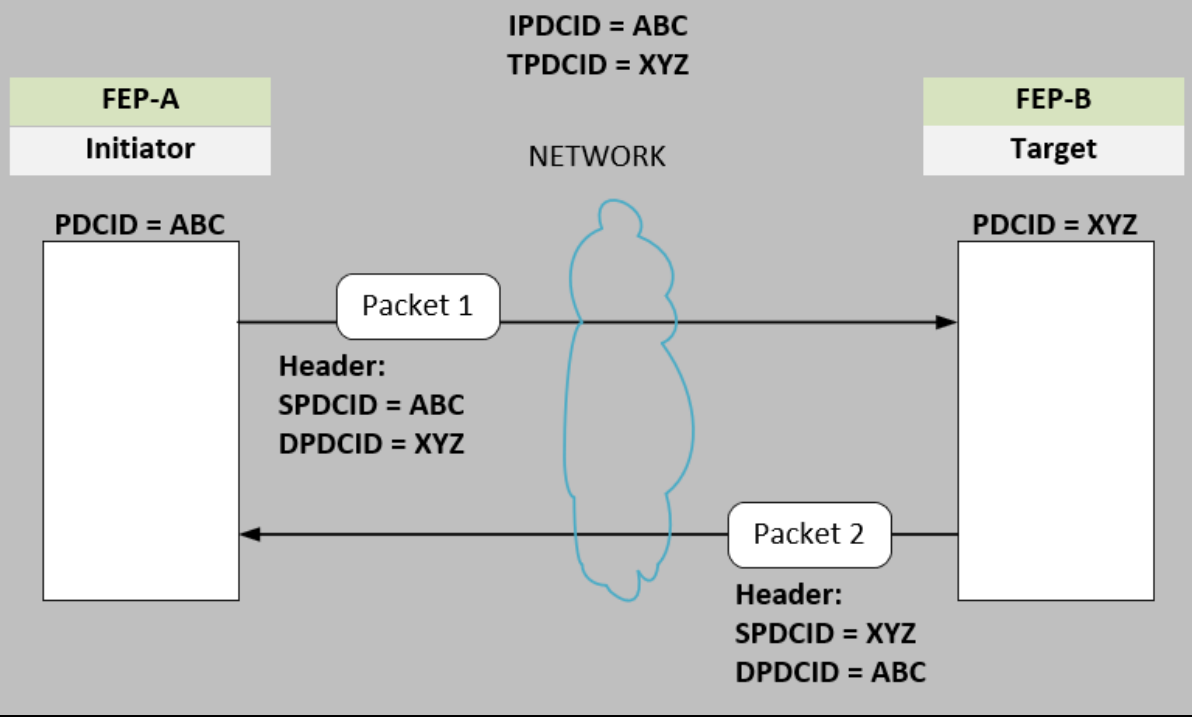
this creates a path from target to initiator where congestion control is applied. Smaller read response data may be carried in a PDS ACK as described in Section 3.5.12.1.

Note that larger read response data is carried in a PDS Request in the return direction. These packets, labeled 'SES Return Data' in Figure 3-52, are technically SES responses and use a **pds.next_hdr** field (e.g., UET_HDR_RESPONSE_DATA or UET_HDR_RESPONSE_DATA_SMALL). PDS requests carry both SES Requests (in forward direction) and SES responses with data (in return direction). PDS ACKs carry only SES responses. Throughout this section the term 'SES request' refers to a request by SES to transmit a packet using a PDS Request. The term 'SES response' refers to a response from SES to be relayed using a PDS ACK. The lower case is used to indicate the PDS function and not the SES next header type.

Certain PDCs enable bidirectional traffic, meaning both the initiator and target can transmit messages (e.g., send or write operations). This is limited to the RODL delivery mode described in Section 3.5.21.

Informative Text:
PDCIDs are referred to using two different perspectives. Every PDC has an initiator FEP and a target FEP. These are fixed during the life of the PDC. There is also a source FEP and destination FEP. These are relative to the FEP that transmitted the packet. That is:

- *IPDCID* refers to the PDCID assigned by the initiating FEP
- *TPDCID* refers to the PDCID assigned by the target FEP
- *SPDCID* refers to the PDCID assigned by the FEP that generated and is transmitting a packet
- *DPDCID* refers to the PDCID assigned by the FEP that is the destination of a packet
- *SPDCID* and *DPDCID* are carried in the PDS headers as **pds.spdcid** and **pds.dpdcid** respectively



Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

3.5.3 Packet Delivery Services

The packet delivery sublayer services include:

- Delivery of requests and responses sourced by the local SES to the destination SES.
    - PDS processes SES packets—not messages.
        - SES requests PDS to deliver a packet with specified ordering and reliability.
    - PDS processes SES responses.
        - SES generates a response for each delivered packet, nominally 'success' or 'fail', and optionally including a small amount of state (e.g., fail reason) and/or data.
        - An SES Response may carry data (e.g., fetching atomic or read data).
        - SES indicates which responses are guaranteed to be delivered.
        - The SES Response is carried in a PDS ACK packet.
- Receiving and forwarding requests and responses from destination SES to local SES.
- Providing reliability and ordering guarantees for packets received from SES and transmitted on the wire as well as packets received from the wire and passed to SES.
    - PDS offers both reliable and unreliable packet delivery modes.
    - PDS offers both ordered and unordered packet delivery modes.
- Setting up and tearing down dynamic connections between fabric endpoints (FEPs).
    - PDS provides a connectionless interface to SES; using dynamically established ephemeral connections, referred to as packet delivery contexts (PDCs), between FEPs on an as-needed basis.
    - PDS also provides connectionless reliable delivery for idempotent operations; a separate delivery mode (RUDI—see Section 3.5.7.4) is used for this that does not establish any connection.
    - PDS provides connection-oriented services to SES, using reserved PDCs configured by the user.

PDS defines packet delivery modes that provide varied ordering and reliability services. Each UET profile defines which packet delivery modes are available within that profile and which congestion management schemes are available. UET profiles are defined in the UET overview section. UET packet delivery modes are defined in this PDS section—refer to Section 3.5.6.

The UET PDS is designed in close conjunction with SES and CMS. SES is responsible for:

- Segmenting messages into packets provided to PDS for transmission
- Reassembling packets received from PDS into messages
- Generating a response for every request packet received

PDS is unaware of the SES transaction type. Data returned in response to a read request is called 'return data'. This is carried in a PDS ACK or in a PDS Request on the return direction as described in Section 3.5.12.1.

CMS services include:

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

- Monitoring telemetry signals to detect network and endpoint congestion.
- Carrying congestion state between sources and destinations.
- Generating signals to control the transmission characteristics of PDS connections. The transmission characteristics are referred to as transmit controls and are based on the following:
    - The amount of data that can be outstanding on a connection or group of connections (i.e., the window size and/or credit).
- Supporting multipath delivery with congestion aware path assignment.

UET breaks larger reads at the semantic sublayer into multiple read request packets where a single read request packet is limited to requesting a maximum of one MTU. This is done by the semantic sublayer (SES). Refer to SES Section 3.4.5.2 for details.

UET also supports packing multiple messages into a single packet for local networks. This is done in SES and treated by PDS as a single packet. Refer to Section 3.4.3.3.

This specification provides a logical interface between PDS and SES that is used to define PDS behavior. The actual interface between PDS and SES is implementation specific.

Informative Text:
Implementations may choose to merge PDS and SES.

3.5.4 PDS-SES Logical Interface

This section contains non-normative text that describes an example of the SES-PDS interface from a logical perspective to illustrate the information that crosses the interface. The interface between PDS and SES is implementation specific. Table 3-34 summarizes an example PDS-SES interface using C function signatures. The example interface is depicted visually in Figure 3-53. This example is presented to provide a framework for better understanding the overall architecture but does use some terminology that is not introduced until subsequent sections.

The following defines logical structures and commands passed between SES and PDS. This is intended to describe the concepts without establishing requirements for implementation. The interface includes source FEP fields, which allows multiple FEPs; a single FEP instance would not necessarily pass the source FEP fields with each call.

```
struct uet_ep *src_fep    # ptr to struct with source address, etc.
struct uet_ep *dst_fep    # ptr to struct with dest address, etc.
uint32_t jobid            # SES passed through JobID
uint32_t tss_context      # Transport Security Sublayer context (e.g., SDI),
                          #   used to limit pkts on PDC to a common SDI
uint8_t mode              # delivery mode = {RUD, ROD, RUDI, UUD}
uint16_t rod_context      # identifies a ROD send queue, used to keep packets
                          #   from a send queue on same PDC
uint16_t rsv_pdc_handle   # used to keep pkts in same reserved PDC;
                          #   0 = not to a reserved PDC
uint16_t tx_pkt_handle    # SES assigned packet handle at source
uint16_t msg_id           # SES assigned message identifier at source
```


Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

```
void *pkt                  # ptr to packet
uint16_t pkt_len           # packet length in bytes
void *rsp                  # ptr to response
uint16_t rsp_len           # response length in bytes
uint8_t tc                 # traffic class
uint8_t next_hdr           # controlled by SES, used to determine the type of
                           #   header in the encapsulated UET payload
bool som                   # TRUE => start of message
bool eom                   # TRUE => end of message
bool lock_pdc              # TRUE => do not close this PDC until SES indicates
                           #   the lock can be lifted (separate function)
bool return_data           # TRUE => packet must use PDC in orig_pdcid,
                           #   set for read responses
uint16_t orig_pdcid        # PDCID from Read request in fwd direction
                           # [local ID identifying a specific PDC]
bool orig_psn_val          # TRUE => include orig PSN field in PDS Request hdr
uint32_t orig_psn          # PSN from Read req or Def Send in fwd direction
bool gtd_del               # TRUE => SES Response needs guaranteed delivery
bool ses_nack              # SES indication to send a PDS NACK
uint16_t eager_id          # SES identifier for eager estimate request
uint32_t eager_size        # size in bytes of eager data
uint16_t rx_pkt_handle     # PDS assigned packet handle at destination
bool pdc_pause             # TRUE => SES stops sending RUD/ROD packets to PDS
bool rudi_pause            # TRUE => SES stops sending RUDI packets to PDS
enum pds_error             # enum of reasons for PDC reset
```

Contexts are used to group or isolate packets for PDCs or CCCs. Contexts are assigned by SES and used by PDS in assigning packets to PDCs and/or CCCs.

Table 3-33 - Packet Contexts

Context	Usage
tss_context	Used to limit all packets on a PDC to a common SDI; that is, all packets on a PDC must have the same *tss_context*. SDI, secure domain identifier, is defined in the TSS Section 3.7.4.
rod_context	Used to keep all packets with common *rod_context* on the same PDC. Multiple *rod_contexts* may map to the same PDC. If used with security, this includes the SDI, and *tss_context* is ignored.
rsv_pdc_handle	This handle is assigned when a reserved PDC is allocated. It is used to assign group packets to a common PDC; that is, only packets passed to the PDS with this handle will use the associated PDC. If used with security, this handle encompasses the SDI, and *tss_context* is ignored.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

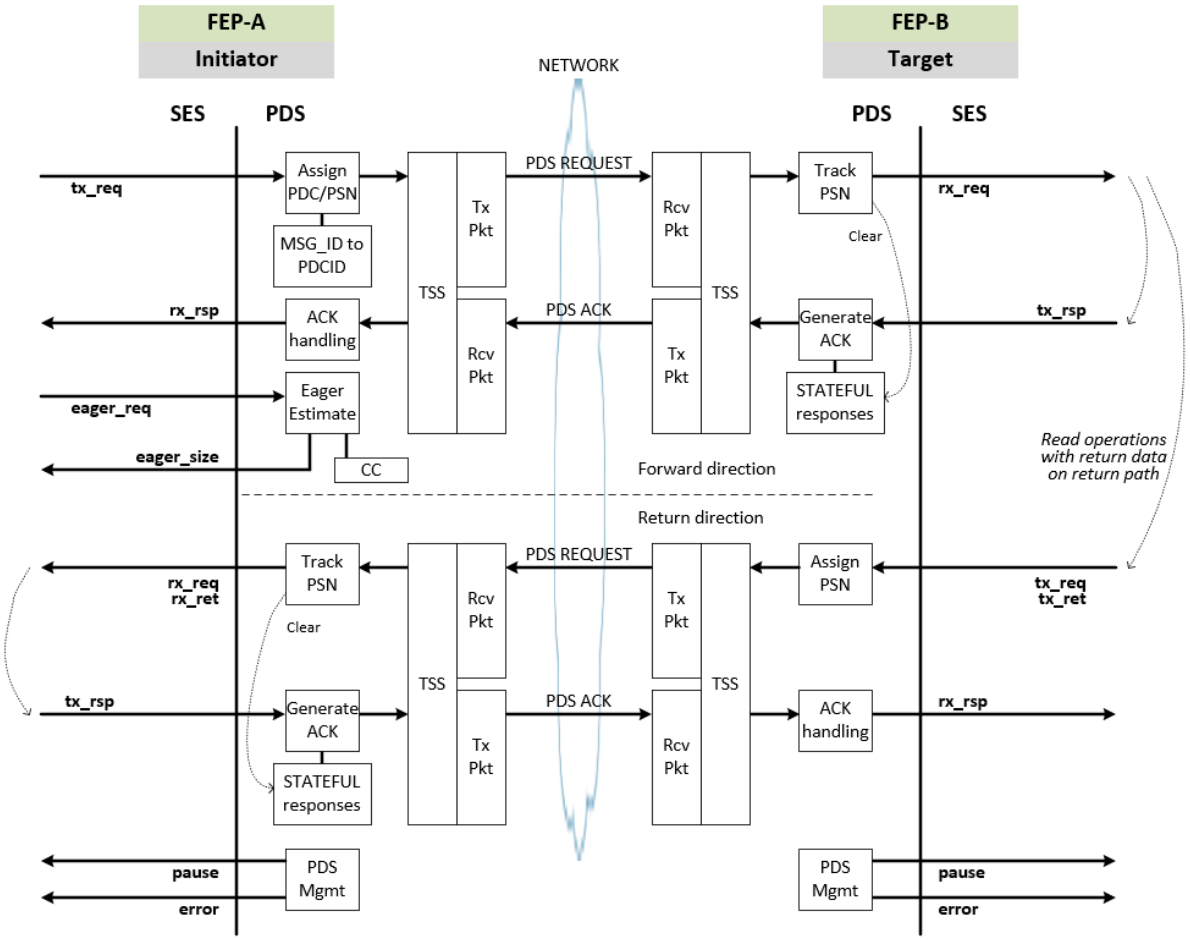


Figure 3-53 - Example PDS-SES Interface

Table 3-34 - Summary of Function-Based PDS-SES Interface Example

Function Name	Direction	Description and Parameters
ses_pds_tx_req()	SES to PDS at initiator	**SES request for PDS Transmission** *src_fep, dst_fep, jobid, mode, rod_context, next_hdr, tc, som, eom, lock_pdc, tx_pkt_handle, pkt, pkt_len, msg_id, tss_context, rsv_pdc_handle* • *msg_id* is used to keep all packets of a message on same PDC, e.g., PDS maintains *msg_id* to *pdcid* mapping from *som* to *eom*, or longer if *lock_pdc* = TRUE. • *lock_pdc* prevents PDS from tearing down a PDC, see the description in *ses_pds_unlock()* for more information. • If *rsv_pdc_handle* is non-zero, the packet is directly mapped to a reserved PDC. These PDCs are used for a dedicated service (e.g., INC) where the PDC remains established until closed by software.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Function Name	Direction	Description and Parameters
pds_ses_rx_req()	PDS to SES at target	**PDS reception of SES request** *rx_pkt_handle, pkt, pkt_len, next_hdr, orig_pdcid, orig_psn* • *orig_psn* is the PSN from the received packet that is used by SES for some read responses; this is only valid for packets received on the forward direction.
ses_pds_tx_ret()	SES to PDS at target	**SES return data for PDS Transmission** *src_fep, dst_fep, jobid, mode, next_hdr, tc, som, eom, tx_pkt_handle, pkt, pkt_len, msg_id, return_data, orig_pdcid, rsv_pdc_handle* • When *return_data* is TRUE, this packet is a read response, and the packet must use the same PDC as the associated read request (*orig_pdcid*). • An implementation may integrate this into a single *ses_pds_tx_req()* function.
pds_ses_rx_ret()	PDS to SES at initiator	**PDS reception of SES return data** *rx_pkt_handle, pkt, pkt_len* • An implementation may integrate this into a single *pds_ses_rx_req()* function.
ses_pds_tx_rsp()	SES to PDS	**PDS transmission of SES Response** *src_fep, dst_fep, rx_pkt_handle, gtd_del, ses_nack, rsp, rsp_len*
pds_ses_rx_rsp()	PDS to SES	**PDS reception of SES Response** *tx_pkt_handle, rsp, rsp_len*
ses_pds_unlock()	SES to PDS	**SES unlocks the PDC, allowing the PDC to be torn down** *msg_id* • The *lock_pdc* parameter to the *ses_pds_tx_req*() function indicates that the PDC may not be closed until the return data for the read request has been returned on the PDC. The PDC maintains a counter that is incremented each time a *ses_pds_tx_req()* function is invoked with the *lock_pdc* parameter set. The *ses_pds_unlock()* function decrements that counter. If the counter is non-zero, the PDC cannot be closed.
ses_pds_eager_req()	SES to PDS	**SES request for estimate of appropriate eager size at Initiator** *src_fep, dst_fep, mode, tc, eager_id* • Eager data is used only for rendezvous. • PDS reliability determines if a CCC exists for this *dst_fep, tc*, and *mode* tuple. If yes then CMS calculates the eager size for that CCC. Otherwise it returns the default eager value.
pds_ses_eager_size()	PDS to SES	**PDS response with estimate of appropriate eager size at Initiator (return for ses_pds_eager_req)** *eager_id, eager_size* • Estimated the eager size value.
pds_pause()	PDS to SES	**PDS indication to SES that resources are temporarily not available** *pdc_pause, rudi_pause* • No RUD or ROD request packets are passed to PDS while *pds_pause* is TRUE. No RUDI packets are passed to PDS while *rudi_pause* is

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Function Name	Direction	Description and Parameters
		TRUE. PDS calls this with *pdc_pause* = TRUE or *rudi_pause* = TRUE and later calls again with *pdc_pause* = FALSE or *rudi_pause* = FALSE. • An implementation MAY allow responses when *pds_pause()* is invoked with *pds_pause* = TRUE or *rudi_pause* = TRUE.
pds_ses_error()	PDS to SES	**PDS unrecoverable Error** *pds_error, tx_pkt_handle or rx_pkt_handle* • If a PDC has an unrecoverable error, PDS returns a *pds_ses_error* for every outstanding, unacknowledged SES request packet associated with the error event—this may be all outstanding packets if PDS is torn down in error.

PDC processing at the initiator and target is largely symmetrical, accepting requests from SES to transmit a packet and returning the corresponding SES Response. At the target, SES requests are limited to responses to SES read requests (i.e., return data).

Only the initiator issues a query for an eager window size as only the initiator can transmit Send messages. These sequences of related functions are shown in the high-level diagram in Figure 3-54 and are described in more detail ahead.

Figure 3-54 shows how these example APIs would interact between the SES-PDS at both the initiator and target. The thicker arrow around the (Request/ACK) and (Return Data/ACK) illustrates the independent PSN spaces used on the forward and return direction. PSNs are described in Section 3.5.8.

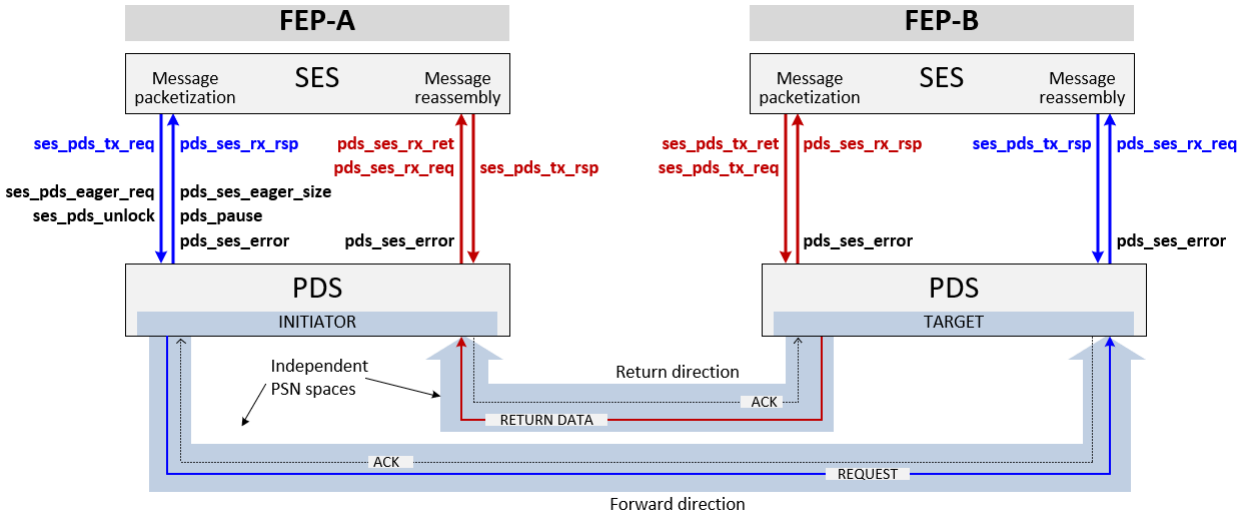


Figure 3-54 - Example PDS-SES Interface between Initiator and Target

3.5.5 PDS Configuration Parameters

Table 3-35 contains a summary of the PDS configuration parameters. At least one set of the PDS configuration parameters MUST be supported for each FEP. If reserved PDCs are supported, a second set of these parameters MUST be supported for each FEP. The reserved pool, discussed in Section 3.5.18, is used for special services.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

The two pools are distinguished by adding a three-letter acronym to the start of the name. GEN refers to the general pool. RSV refers to the reserved pool. For example, *Gen_Max_ACK_Data_Size* and *Rsv_Max_ACK_Data_Size*.

These parameters are configured with the specified units. The required range is described in Table 3-35 and larger ranges may be supported. The indicated quanta is shown and finer-grain quanta may be supported. The default values shown are for reference for a best-effort network using UET-CC with packet trimming. Table 3-36 MUST be supported by each FEP.

Table 3-35 - PDS Configuration Parameters

Name	Required Range	Description
UET_Over_UDP	Boolean Default: 1	This determines if UET runs over UDP (when *UET_Over_UDP* is set) or directly over IP (when *UET_Over_UDP* is cleared).
UDP_Dest_Port	0—2^{16}-1 Default: 4793	When UET is running over UDP, this number in the UDP destination port indicates the following protocol is UET.
IP_Proto_Nxt_Hdr	0—255 Default: N/A[1]	This value is used when UET is run directly over IP. Until an IP protocol number is assigned for UET, an experimental number (253–254) can be used[2].
UET_Data_Protect	0—3 Default: 1	Global configuration – across an entire fabric domain 0 = Neither CRC nor TSS enabled 1 = CRC enabled 2 = TSS enabled 3 = Reserved Refer to Section 3.5.25.7
Limit_PSN_Range	Boolean Default: TRUE	When set, a PDC will close when the PSN reaches *Start_PSN* + 2^{31}. This is an optional security feature. Refer to Section 3.5.8.2.2.
Default_MPR	1—255 Default: 8	The default MPR assumed when creating a PDC. Zero is not valid for the default setting; if set to 0, use 1. Refer to Section 3.5.12.4.4.
Max_ACK_Data_Size	0—8 KB Unit: bytes Quanta: 16 B Default: 16 B	Maximum amount of return data that can be carried with a PDS ACK sent in response to a PDS Request sent in the forward direction (i.e., from target to initiator in response to a read request from the initiator). Refer to Section 3.5.12.1
Trimmable_ACK_Size	0—10 KB Units: bytes Quanta: 16 B Default: 10 KB	ACK packets carrying read response data which are larger than this size use a trimmable DSCP. Support for trimmed ACK packets is optional. If not supported, set this to a value larger than the maximum ACK MTU.
ACK_On_ECN	Boolean Default: TRUE	TRUE: reception of a packet that is ECN marked triggers an ACK generation. FALSE: ACK generation is not based on whether a received packet is ECN marked or not.
Enb_ACK_Per_Pkt (optional—required if coalesced ACKs is supported)	Boolean Default: FALSE	FALSE: Use coalesced ACKs. TRUE: Use ACK per packet.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Name	Required Range	Description
		When TRUE, sources set **pds.flags.ar** field in every PDS request. Support for coalesced ACKs is optional. Refer to Section 3.5.12.2.
ACK_Gen_Trigger (optional—required if ACK coalescing is supported)	0—32 KB Unit: bytes Quanta: 256 B Default: 16 KB	Configured value in bytes, when ACK_GEN_COUNT reaches this threshold, an ACK is generated. See Section 3.5.12.4.1.
ACK_Gen_Min_Pkt_Add (optional—required if ACK coalescing is supported)	0—2 KB Unit: bytes Quanta: 64 B Default: 1 KB	Minimum number of bytes added to ACK_GEN_COUNT when a packet is received at a PDC. See Section 3.5.12.4.1.
RTO_Init_Time	0—8 sec Unit: 128 ns Quanta: 128 ns or less Default: n/a	Retransmit packet after this amount of time if no ACK or NACK is received within this configured time (timer at source). Default depends on CC mode, fabric scale, etc.
Max_RTO_Retx_Cnt	0—15 Unit: retry count Default: 5	Max number of retransmissions for a single packet before declaring a failure event; maximum setting indicates infinite retry. Retransmissions based on time out (RTO) events are counted and retransmission based on NACKs may be included in the count. Refer to Section 3.5.12.7.
NACK_Retx_Time	0—8 sec Unit: 128 ns Quanta: 128 ns or less Default: n/a	This set of configured times is used to determine how long to delay the retransmission of a NACK'd packet. At least one configurable value is required. A set of four values should be provided, where the value is selected based on the NACK code. That is, each NACK code is mapped to one of these times or to no delay. Refer to Section 3.5.12.7.
Max_NACK_Retx_Cnt	0—31 Unit: retry count Default: 5	Optional: This counter can be used to set a separate, possibly higher, threshold for retransmissions based on NACK packets. Maximum setting indicates infinite retry. Refer to Section 3.5.12.7.
New_PDC_Timeout_Thresh	0—64K Unit: timeout count Default: 1024	This threshold defines when an error is reported indicating a potential DoS attack. It is used with NEW_PDC_TIMEOUT_CNT. Refer to Section 3.5.8.2.1.
New_PDC_Time	0—100 ms Unit: 100 μs Default: n/a	If TSS is enabled, this defines the time allowed for a PDC initiator to establish a PDC. When this expires, the PDC is closed in error and NEW_PDC_TIMEOUT_CNT is incremented.
PDS_Clear_Time	0—100 ms Unit: 128 ns Default: n/a	Optional: This configured time is used to trigger generation of a Clear Command CP when an ACK is received that advances *CLEAR_PSN* and there is no PDS Request pending to carry the updated *CLEAR_PSN*. Refer to Section 3.5.11.4.3.
Close_REQ_Time	0—100 ms Unit: 128 ns Default: n/a	Optional: This configured time is used to constrain the amount of time an initiator is allowed to take to

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Name	Required Range	Description
		respond to a request to close with a Close Command CP. Refer to Section 3.5.8.3.
Tail_Loss_Time	0—100 ms Unit: 128 ns Default: n/a	Optional: See Section 3.5.15.
Max_Tail_Loss_Retx	0—15 Unit: retry count Default: 5	Optional: See Section 3.5.15.
Override_MPR_Zero_Value	0 – 127 Default: 0	This configuration parameter defines the interpretation of the value of 0 in the **pds.mpr** field. When set to 0, this parameter indicates that a **pds.mpr** setting of 0 should be interpreted as meaning that the *MP_Range* is ignored. Otherwise, the value of 0 in the **pds.mpr** field means that the *MP_Range* is set to this CSR value.
Syn_Retx_Safety_Time	$0 – 2^{16}$-1 Unit: 1 μs	If a packet with **pds.flags.syn**=1 is an inferred loss (e.g., based on timeout), a FEP MUST wait *Syn_Retx_Safety_Time* between the original transmit of that packet and the closing of that PDC (see Section 3.5.8.3). The required range is a 24-bit value in microseconds. *Syn_Retx_Safety_Time* SHOULD be set to 0 if encryption is enabled or if another anti-replay mechanism is enabled.

Note:

1. As of this specification's publication, IANA has not assigned a native IP protocol number. This field remains configurable, allowing users to set experimental or locally determined values.
2. The IETF does not allow experimental numbers to be set as default values. Implementations are required to provide a means of configuring the *IP_Proto_Nxt_Hdr* parameter. Deployments may use an experimental value (e.g. 253, 254) until the UEC obtains an IP protocol number.

Table 3-36 - PDS Status and Error Indications

Name	Field Type	Description
NEW_START_PSN_TO_ERR	Boolean FALSE: no error	This is set when the NEW_PDC_TIMEOUT_CNT crosses the configured threshold, *New_PDC_Timeout_Thresh*
NEW_PDC_TIMEOUT_CNT	Bit or counter	This is used when TSS is enabled. Incremented when *New_PDC_Timer* expires.
CC_TYPE_EVENT	16-bits	Optional: Should be clearable. Bit is set when an ACK_CC arrives with **pds.cc_type** corresponding to the bit, e.g., if **pds.cc_type** = 3 then bit 3 in the field is set.
CCX_TYPE_EVENT	16-bits	Optional: Should be clearable. Bit is set when an ACK_CCX arrives with **pds.ccx_type** corresponding to the bit, e.g., if **pds.ccx_type** = 4 then bit 4 in the field is set.
NCCX_TYPE_EVENT	16-bits	Optional: Should be clearable.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Name	Field Type	Description
		Bit is set when a NACK_CCX arrives with **pds.nccx_type** corresponding to the bit, e.g., if **pds.nccx_type** = 0 then bit 0 in the field is set.
UET_CRC_ERR_COUNT	32-bits	Optional: Incremented when a packet with a CRC error is received.
PDC_CLOSE_IN_ERR	Bit or counter	Optional: This is used to indicate (or count) when a PDC is closed in error.
PDS_TYPE_INVALID	1-bit to 16-bit counter	This counter is incremented when a packet is received with a **pds.type** field that is not recognized. 1-bit counter is effectively an event flag.
PDS_CTL_TYPE_INVALID	16-bit counter	This counter is incremented when a CP is received with a **pds.ctl_type** field that is not recognized.
PDC_REQ_ERR	Bit	Optional: This indicates a packet with **pds.flags.req** = 3 was received.
OUT_OF_WINDOW_PSN	Bit or counter	Optional: This indicates a packet with PSN outside of the expected window was received.
UNEXPECTED_TRIM	Bitmap Counter	Optional: This counter is incremented every time a trimmed packet that should not be trimmed arrives. Bitmap indicates which packet types [ACK, NACK_CP]

3.5.6 Reliability and Ordering

Five combinations of reliability and ordering define the packet delivery modes. Each combination is referred to as a delivery mode:

- Reliable unordered delivery (RUD) (Section 3.5.7.1)
- Reliable ordered delivery (ROD) (Section 3.5.7.2)
- Reliable ordered delivery for local networks (RODL) (Section 3.5.7.3)
- Reliable unordered delivery of idempotent operations (RUDI) (Section 3.5.7.4)
- Unreliable unordered delivery (UUD) (Section 3.5.7.5)

The reliable delivery modes RUD, ROD, and RODL are defined in the context of a dynamically established PDC. PDCs are described in Section 3.5.8. The RUDI and UUD modes do not use PDCs.

When a PDC is established, the PDC delivery mode remains the same for all packets on that PDC until the PDC is closed. The delivery mode selected for a given SES request is determined by SES. How packets are assigned to a PDC is described in Section 3.5.8.1. More than one PDC between a pair of FEPs is supported. The criteria for deciding when to establish multiple PDCs between a pair of FEPs is implementation specific. There is no ordering guarantee across packets associated with different PDCs.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Packets of a single SES message are sent on the same PDC. Some SES transactions—such as rendezvous and deferrable send—may use multiple messages that may be placed on separate PDCs. Applications that wish to allow distribution of large data transfers across multiple PDCs must split data transfers into multiple messages above SES, as multiple messages may be mapped across different PDCs. When using rendezvous, the eager portion of the message may use a different PDC than the rest of the message. When using deferrable send, the RTR will use a different PDC (initiated by the target of the deferrable send) and data transfer may use a different PDC than the original deferrable send. Handling of these two cases is implementation specific.

Implementation Note:
Because there may be multiple PDCs between two FEPs, PDS needs to track the PDC on which the first packet of a message is sent. Additional packets for the same message are mapped to the same PDC. PDS tracks the **ses.som** and **ses.eom** fields to assure packets of the same message use the same PDC and to assure a PDC is not closed in the middle of a message.

Once a deferrable send is deferred and restarted via RTR, it is considered a new message and may use a different PDC.

Packets from different messages are not interleaved on a single PDC. Specifically, all packets of a message are on the wire before a packet from another message is put on the wire. The entire message is transmitted on a PDC and then the next message is transmitted. PDS may establish an additional PDC between two FEPs to avoid delaying short messages.

PDS ACKs carrying SES Responses that are marked for guaranteed delivery are always delivered (see Section 3.5.17). Specifically, when an SES Response on a RUD, ROD, or RODL PDC is marked as requiring guaranteed delivery, PDS is responsible for assuring that response is delivered and cleared. Refer to the Semantics Section 3.4.4.3 for details on which responses are guaranteed. More details on guaranteed delivery and clear of ACKs are provided in Section 3.5.11.

There is no acknowledgement for the unreliable delivery mode.

3.5.7 Packet Delivery Modes Overview

3.5.7.1 Reliable Unordered Delivery (RUD)

The RUD delivery mode guarantees that each packet is delivered to the target SES once and only once. Packets are delivered to the semantic sublayer in the order they arrive from the network. This mode uses selective retransmission capabilities and enables semantic processing and direct data placement out of order. Direct data placement refers to writing of data arriving at the Ethernet network port directly into system memory without CPU intervention. RUD relies on sequence numbering to identify lost and duplicate packets.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

3.5.7.2 Reliable Ordered Delivery (ROD)

The ROD delivery mode guarantees that each packet is delivered to the target SES once and only once in the order the packets are sent from SES to PDS at the initiator. A ROD PDC requires packets to be transmitted in the same order in which SES sends the packets to PDS. The packets on a ROD PDC are transmitted in order over the network port using a single network path (i.e., using a single entropy value) and arrive at the target network port in the same order (excluding error scenarios).

ROD uses GoBackN loss recovery. GoBackN drops all packets that arrive out of order, requiring the source to retransmit all packets starting from the first missing PSN. That is, the missing PSN is referred to as 'N' and the source goes back to 'N' and retransmits.

ROD ACKs are transmitted in the same order as the packets they ACK and read responses are transmitted on the wire in packet order. There are no ordering requirements between PDS ACKs on the forward direction and read responses on the return direction. An ACK may be sent and arrive out of order when that ACK is requested, e.g., based on receiving a duplicate packet or ACK request CP. Refer to Section 3.5.21.1.

3.5.7.3 Reliable Ordered Delivery for Local Networks (RODL)

The RODL delivery mode has the same requirements as ROD with the following exceptions:

- RODL uses bidirectional PDCs. This allows the target FEP to initiate transactions.
- RODL includes the option to exclude PDS retransmissions.
- RODL over the UFH_16 header compresses the PDS header, specifying a smaller PDCID space, smaller clear state, and compressed flags.
- When using RODL over the UFH_16 header, the 32-bit expected PSN is included in the CRC calculation and not explicitly included in the header once the PDC is established.
- RODL carries ACK state in the PDS Request for the reverse direction.
- RODL over the UFH_16 header is optimized for fabrics with a single tier of switching.

3.5.7.4 Reliable Unordered Delivery for Idempotent Operations (RUDI)

Idempotent means a packet can be processed by the target SES multiple times and the result is the same. For example, multiple RMA Writes to a memory address or multiple RMA Reads from a memory address.

The RUDI delivery mode guarantees that each packet is delivered to SES at least once. Packets are delivered in the order they arrive from the network and every RUDI packet received from the network is delivered to SES without attempting to remove duplicate packets. PDS assigns unique identifiers per packet for RUDI. The RUDI packet identifiers for different packets need not be related to each other (i.e., the RUDI packet identifiers are not required to be based off an incrementing counter). RUDI is optimized to enhance scalability and minimize endpoint state, relative to RUD, by exploiting the relaxed requirements associated with being able to deliver a packet more than once. RUDI requires no PDS or SES state at the target, uses a RUDI response per packet, and supports direct data placement out of order; thus, RUDI is very efficient at the PDS and SES levels.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

RUDI does not use ACKs. The response to each RUDI request is a RUDI response. RUDI uses NACK for early detection of packet loss.

The intended use for RUDI is 'do lots of RMA writes/RMA reads to multiple destinations and then barrier'. For example, an application can use RUDI to exchange data among participating processes (e.g., send a series of write and read operations) and, after all of these are completed, perform another send that generates a completion at the target(s) to indicate all data movement is done.

PDS has no knowledge of idempotency and relies on SES to identify when RUDI may be used. RUDI is never used for non-idempotent operations.

When using UET congestion control (UET-CC) on a FEP and assigning RUDI to the same traffic class as RUD/ROD, it is the responsibility of the application to avoid creating congestion in the network. RUDI is not controlled by UET CC. If RUDI is used for large data transfers, then it SHOULD NOT use the same traffic class as RUD/ROD traffic that is using UET-CC, as it may impact performance of the RUD/ROD traffic.

3.5.7.5 Unreliable Unordered Delivery (UUD)

The UUD delivery mode is a basic datagram service. An unreliable datagram service enables best-effort delivery. UUD avoids the need for applications to use a different semantic sublayer (and network API) such as UDP to leverage unreliable datagrams.

When using UET-CC on a FEP and assigning UUD to the same traffic class as RUD/ROD, it is the responsibility of the application to avoid creating congestion in the network. UUD is not controlled by UET-CC. If UUD is used for large data transfers then it SHOULD NOT use the same traffic class as RUD/ROD traffic that is using UET-CC, as it may impact performance of the RUD/ROD traffic.

Informative Text:
Unreliable ordered delivery is not specified. The four modes provided map to all common fielded use cases.

3.5.7.6 Ordered Packets vs. Ordered Messages

PDS offers ordered and unordered packet delivery modes. PDS manages packet sequence number (PSN) spaces for each PDC to provide ordered delivery. PDS has no knowledge of message ordering. SES is responsible for message ordering.

3.5.8 Packet Delivery Contexts (PDC)

A packet delivery context (PDC) is a dynamically established FEP-to-FEP connection that provides the context needed to implement reliability, ordering, duplicate packet elimination, and congestion management.

A PDC is used to send requests from an initiator to a target with PSNs that are returned in ACK responses by the target. Packets are transmitted in both directions. A PDC uses TX state on the initiator (next sequence number, ACK bitmap, etc.) and RX state (received sequence number state, ACK state

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

including guaranteed delivery indication, etc.) at the target to provide the PDS service in the forward direction. Each PDC requires a second set of TX and RX state for the return direction. The aggregate of all this state is referred to as a PDC.

A single PDC MUST be limited to a single mode—RUD, ROD, or RODL—and a single traffic class for PDS requests.

Each PDC uses two PSN spaces that advance independently. One PSN space is for the forward direction with requests transmitted by the initiator. The second PSN space is used to carry return data messages from the target to the initiator. Each PSN MUST monotonically increase by one based on packet transmission and reception. Return direction PDS requests are acknowledged in the same manner as forward-direction PDS requests.

The following sub-sections apply only to RUD, ROD, and RODL. RUDI and UUD do not use PDCs.

3.5.8.1 PDC Selection and Sharing

This section defines how packets are mapped to RUD, ROD, and RODL PDCs, and how a PDC can be shared among packet flows between a pair of FEPs, where a packet flow is defined to be a group of related packets between two processes (PIDonFEP).

At the initiator, a packet flow is mapped to a PDC based on the mapping tuple described below. When supporting reserved PDCs (3.5.18), an implementation may provide an API with a PDC handle that allows the user to specify the reserved PDC to use. This handle supersedes the mapping tuple. The mapping tuple is built from network header fields and control information provided by SES. The initiator mapping tuple need not be consistent across FEPs.

When using reserved PDCs, packets are assigned based on a handle passed through the SES rather than the mapping tuples. Refer to Section 3.5.18.

The mapping tuple determines which sets of packets can be assigned to a common PDC. All packets sharing a PDC have the same value for all fields in the tuple. If a packet matches the mapping tuple, it may use the designated PDC or may be mapped to another PDC between the same pair of FEPs. All packets of a single message MUST use the same PDC. Packets from different messages MUST NOT be interleaved on a single PDC. A message is completely transmitted before transmitting a packet from a different message.

ROD traffic has a restriction that packets from the same {JobID, source FA, destination FA, source PIDonFEP, destination PIDonFEP, Resource Index, TC} MUST use the same PDC.

Each PDC carries the traffic of exactly one delivery mode (RUD, ROD, or RODL). The destination FA and TC MUST be included in all tuples.

Two initiator tuples are defined as required—one each for when TSS is disabled or enabled. Optionally, multiple tuples may be supported for RUD, ROD, and/or RODL PDCs. It is beyond the scope of this specification to define how the tuple is selected for each packet when multiple tuples are supported.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

The following initiator PDC mapping tuples MUST be supported:

- {source FA, destination FA, TC, RUD/ROD/RODL}.
- {SDI, source FA, destination FA, TC, RUD/ROD/RODL} only if TSS is supported.
    - tss_context is an example name used in the logical SES-PDS interface for a field passed to PDS by SES which can be used to identify the security domain (SDI).
- The ability to include any combination of JobID, destination PIDonFEP, source PIDonFEP, and Resource Index in the tuples when using RUD/ROD.
- Separate default tuples for RUD, ROD, and RODL traffic.
- If TSS is enabled, include SDI in the tuples.
- It may be beneficial to use multiple ROD or RODL PDCs between a pair of FEPs if there is substantial ROD/RODL bandwidth, as ROD PDCs use a single network path.
    - Mapping of traffic across multiple RODL PDCs to a common destination FEP is left to the implementation.

Reserved PDCs are allocated under software control (see Section 3.5.18).

The following target tuples MUST be supported for RUD/ROD/RODL delivery modes:

- {source FA, destination FA**, pds.spdcid**} tuple identifies the PDC
    - The **pds.spdcid** field is sufficient versus including traffic class and PDC mode, as a FEP uses unique PDCIDs across traffic classes and PDC modes.
    - When **pds.spdcid** is used to associate received packets with a PDC, the **ip.src_addr** field is verified to match the expected value.
    - Note that the destination FA is a constant within a single FEP and may be omitted in implementations instantiating a single FEP (i.e., {source FA**, pds.spdcid**} is sufficient for a single FEP).
    - If TSS is enabled, **tss.sdi** MUST be included in the tuples.

The following CCC_ID mapping tuples MUST be supported:

- {destination FA, **ip.dscp**}
- Assign ROD PDCs unique CCC_IDs (cannot be mixed with RUD CCC_IDs)

A target generating a return data packet in response to a read request MUST use the same PDC the associated read request arrived on. That is, read response data must use the return direction of the same PDC that carried the read request.

The initiator PDS assigns an initiator PDC Identifier (*IPDCID*) that is carried on the wire. When the target accepts the PDC, it assigns a target PDC Identifier (*TPDCID*) that is also carried on the wire. PDC Identifiers are described in Section 3.5.11.5.

Establishment of multiple PDCs between two FEPs using the same tuple and traffic class is supported to enable both RUD and ROD. Establishing multiple RUD PDCs between two FEPs using the same tuple and

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

traffic class SHOULD be supported to reduce head-of-line blocking that may occur when one or more large messages are posted to a PDC.

When UET-CC is used, all RUD PDCs between a pair of FEPs sharing a TC MUST be mapped to the same CCC except for RUD PDCs from the reserved pool, which may use independent CCCs.

When multiple PDCs are established between two FEPs, all packets from a single message MUST use the same PDC. RUD mode allows different messages to be distributed across multiple PDCs. In ROD mode, all messages between {JobID, source PIDonFEP, destination PIDonFEP, RI, TC} MUST use a single PDC.

3.5.8.2 PDC Establishment

A PDC is created dynamically on demand. The protocol is defined to allow PDC creation to occur without incurring a round-trip delay (i.e., with zero startup time). The initiator assigns an *IPDCID* used as part of PDC creation. When the target establishes the PDC, it assigns a *TPDCID* that is returned to the initiator.

PDCID = 0 is reserved and MUST NOT be used to establish a PDC. PDCID = 0 is used in NACK messages when a valid PDCID is not available.

PDC creation refers to allocating PDC resources and transmitting packets. PDC establishment refers to both initiator and target allocating and using PDCIDs. The target moves directly from created to established, while the initiator waits until a response packet is received with the target's PDCID. Trimmed packets MUST NOT be used to establish a new PDC. A NOOP CP or Negotiation CP may establish a new PDC.

PDCIDs are intended to be locally unique at the FEP level to enable efficient implementation. An implementation MUST allocate PDCIDs such that {source FA, destination FA, PDCID} is globally unique. For example, the PDC mode (RUD, ROD) cannot be used to uniquely identify a PDC. This is done at the initiator and target FEPs. When using UET-CC, a CCC_ID is assigned such that RUD PDCs share a CCC when possible and ROD PDCs each use their own CCC.

An initiator requests PDC establishment by setting **pds.flags.syn** in a PDS Request packet. This PDS Request packet may also contain an SES Request. The **pds.flags.syn** MUST be set in all packets sent on the PDC until a packet is received for the PDC from the target. An initiator SHOULD NOT send more than *MP_Range* - 1 packets with **pds.flags.syn**=1. If a PDC is still not established, an initiator SHOULD establish a connection with control packets before sending any more data packets. *MP_Range* is defined in Section 3.5.11.13. This provision facilitates recovery when all packets with **pds.flags.syn**=1 are lost. Before the first PDS packet is received, the *TPDCID* is not known by the initiator, as it is assigned by the target. The PDS Request packet sent by the initiator overloads the **pds.dpdcid** field to allow the target to determine the mode of the PDC and the starting initiator PSN number:

- The initiator MUST set *DPDCID* = {**pds.pdc_info, pds.psn_offset**} when the initiator generates packet with **pds.flags.syn** set to 1

The **pds.pdc_info** field is defined in Section 3.5.11.6. The **pds.psn_offset** field is the offset of the PSN from the starting PSN for the PDC, as illustrated in Table 3-37. The initiator starting PSN MUST be a

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

random or pseudo-random number at least 2^{16} distance from the last PSN used on that PDC. This is done to reduce the likelihood of accepting delayed and stale packets in the network. When UET encryption is enabled, follow the requirements in Section 3.5.8.2.1.

Informative Text:
PDC connections are ephemeral with all state cleared on close except the last PSN used, which is saved for use in selecting a new starting PSN for unencrypted PDCs. That is, the random new *Start_PSN* is independent from the destination—one 'last used PSN' is stored for each PDC.

Table 3-37 - PSN Offset Field

Packet # on PDC	PSN	PSN_OFFSET
First	Starting PSN + 0	0
Second	Starting PSN + 1	1
Third	Starting PSN + 2	2
etc.	Starting PSN + N	N

The PDC establishment procedure MUST be supported as follows:

1. If a PDS Request is received and there is not sufficient packet buffer to accept the packet, the packet is dropped and a NACK packet SHOULD be transmitted.
    a. Set **pds.nack_code** = NO_PKT_BUF.
        i. If the PDC has not yet been established and a NACK is transmitted, **pds.spdcid** is set to 0x0. This indicates a PDC was not allocated at the target.
    b. Drop the packet and exit processing of the PDS Request packet.
2. Check if the packet is trimmed—if yes:
    a. Send a NACK with **pds.nack_code** = UET_TRIMMED or UET_TRIMMED_LASTHOP
        i. See Section 3.5.15.1.
    b. If the PDC has not yet be established, the **pds.spdcid** in the NACK is set to 0x0 if the NACK is generated.
    c. Drop the received trimmed packet and exit processing of the PDS Request packet.
3. If **pds.flags.syn** = 1:
    a. **pds.flags.syn** is set until the ACK reaches the initiator; thus, many packets may arrive with the **pds.flags.syn** bit set.
    b. Determine if the PDC already exists using the {**ip.src_addr, pds.spdcid**} mapping tuple—if yes:
        i. Skip to the processing at step d for **pds.flags.syn** = 0.
    c. Check if there are PDC resources available—if no:
        i. Send NACK with **pds.nack_code** = UET_NO_PDC_AVAIL, UET_NO_CCC_AVAIL, UET_NO_BITMAP, UET_NO_PKT_BUFFER, UET_NO_GTD_DEL_AVAIL, UET_NO_SES_MSG_AVAIL, UET_NO_SES_SYNC_AVAIL, or UET_NO_RESOURCE
        ii. Set **pds.spdcid** in the NACK to 0x0.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

iii. Drop the received packet and exit processing of the PDS Request packet.

d. Verify the PDC meets establishment criteria — invalid starting PSN check:
   i. When operating without encryption, there are no checks on the starting PSN.
   ii. When operating with encryption, refer to Section 3.5.8.2.1—if this check fails, allocate the resources, send a NACK packet as described in the next step, drop the received packet and go to the PENDING state and exit processing of the PDS Request packet.
      - NACK packet uses **pds.nack_code** = UET_NEW_START_PSN, **pds.payload_start_psn** = *Start_PSN* and **pds.spdcid** set to allocated *SPDCID*.
   iii. Continue to establish the PDC.

e. Check if the PDC is ROD or RODL and if the received packet has the *Start_PSN*—if no:
   i. Allocate the resources, send NACK, drop the received packet and exit processing of the PDS Request packet.
      - NACK uses **pds.nack_code** = UET_ROD_OOO and **pds.spdcid** set to allocated PDCID.

f. If the establishment criteria are met based on the above checks:
   i. Allocate the PDC resources and *TPDCID*.
   ii. Set *Start_PSN* for return direction to same *Start_PSN* as the forward direction; The PSNs in each direction are independent but start at the same value.
   iii. Return *TPDCID* to initiator in ACK/NACK and exit processing of the PDS Request packet. The *TPDCID* is returned as the **pds.spdcid**.

4. If **pds.flags.syn** = 0, perform the following checks
   a. If **pds.dpdcid** does not match an active PDCID, send a NACK with **pds.nack_code** = UET_INV_DPDCID.
   b. Confirm the network header matches the expected source FA (**ip.src_addr**); if **ip.src_addr** doesn't match then send a NACK with **pds.nack_code** = UET_PDC_HDR_MISMATCH
   c. Confirm the delivery mode (RUD/ROD/RODL) matches; if not, drop the packet, record error event and send a NACK with **pds.nack_code** = UET_PDC_MODE_MISMATCH.
   d. Check the PSN (**pds.psn**)—if outside expected window (Section 3.5.12.2), generate a NACK with **pds.nack_code** = UET_PSN_OOR_WINDOW.
      i. If the PDC type is ROD or RODL, check if the **pds.psn** matches the *Expected_PSN*—if the PSN does not match, transmit a NACK with **pds.nack_code** = UET_ROD_OOO (See Section 3.5.21.1).
   e. If the PSN was already received and the **pds.flags.retx** flag is not set, drop the packet
      i. A NACK is not generated; this event is counted and reported if a configured threshold is exceeded.

A NACK MUST NOT be used to create a PDC except when the target sends a NACK with **pds.nack_code** = UET_NEW_START_PSN or UET_ROD_OOO. In those cases, a PDC is created when a packet arrives with **pds.flags.syn** set and no PDC yet exists.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

The target either verifies the tuple on every received packet or relies on the **pds.dpdcid** once received packets no longer have the **pds.flags.syn** set. The source FA (**ip.src_addr**) and **pds.spdcid** MUST be verified on every packet. When the tuple {**ip.src_addr, pds.spdcid**} is checked and does not match the expected fields, the packet MUST be dropped and an error recorded, UET_PDC_HDR_MISMATCH.

Figure 3-55 illustrates the startup sequence. The **pds.spdcid** is the source PDCID, unique at the FEP that sent the packet. The **pds.dpdcid** is the destination PDCID. The initiator/target labels consistently refer to specific FEPs based on how the PDC is created. The source/destination labels vary depending on the specific task/packet being discussed. For example, in Figure 3-55 both FEP-A and FEP-B may be a source or destination.

Once a PDC is established, packets with **pds.flags.syn** set are accepted for PSNs up to the *Start_PSN* plus *MP_RANGE* (as defined in Section 3.5.11.13). If a packet is received with a PSN higher than *Start_PSN* + *MP_RANGE* and **pds.flags.syn** is set, the packet MUST be NACK'd with **pds.nack_code** = UET_INVALID_SYN.

The source receives an ACK or NACK indicating whether the target established the PDC. If the **pds.spdcid** field in a received NACK is zero, then the PDC was not created. In this case, all traffic (except control packets) on the PDC MUST be paused until a NACK retransmit timer (Section 3.5.12.7) expires or a subsequent ACK or NACK is received with a non-zero **pds.spdcid**.

If the **pds.spdcid** in the ACK/NACK is non-zero, then the PDC is established. The source uses that **pds.spdcid** from the ACK/NACK as the **pds.dpdcid** for all subsequent PDS requests and CPs sent on the PDC, including any retransmissions that originally had **pds.flags.syn** = 1.

When encryption is not enabled, an initiator SHOULD NOT retransmit a packet with **pds.flags.syn**=1 unless an explicit NACK is received for that packet. Instead, for packets with **pds.flags.syn**=1 and are determined to be lost, the initiator SHOULD wait until the PDC is established and retransmit lost packets with **pds.flags.syn**=0. If any packet with **pds.flags.syn**=1 is determined to be lost, the initiator MAY use a Negotiation CP at any time before the Close Command CP is sent to determine if the target will accept responsibility for tracking *Syn_Retx_Safety_Time* when the PDC is ready to close. The Negotiation CP uses the negotiation type (see Table 3-78) of PDS_NEG_ON_OFF with **SYN_RETX_TRANSFER** feature bit of the **pds.net_on_off_mask** field set (see Table 3-80).

If all packets with **pds.flags.syn**=1 are lost, the initiator SHOULD establish the PDC by sending a Negotiation CP (see 3.5.16.7). The Negotiation CP is used to establish the PDC, avoiding the need to send additional payload packets with **pds.flags.syn**=1. An initiator MAY use the Negotiation CP to negotiate that the target assume responsibility for tracking *Syn_Retx_Safety_Time*.

When encryption is not enabled, if a packet with **pds.flags.syn**=1 is considered lost, an initiator MUST capture a timestamp. The timestamp captured SHOULD be the local time that the packet was originally transmitted, but MAY be the local time that it was determined to be lost. This time MUST be kept as part of the initiator PDC state. The initiator PDC needs to track only a single captured timestamp—the

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

one associated with the most recently transmitted packet with **pds.flags.syn**=1 that was determined to be lost.

Informative Text:
Implementations MAY choose to allow the anti-replay mechanisms described in 3.5.8.2.1 to be enabled independently of encryption. A deployment that guarantees that anti-replay mechanisms are enabled on all nodes MAY safely set *Syn_Retx_Safety_Time* to 0.

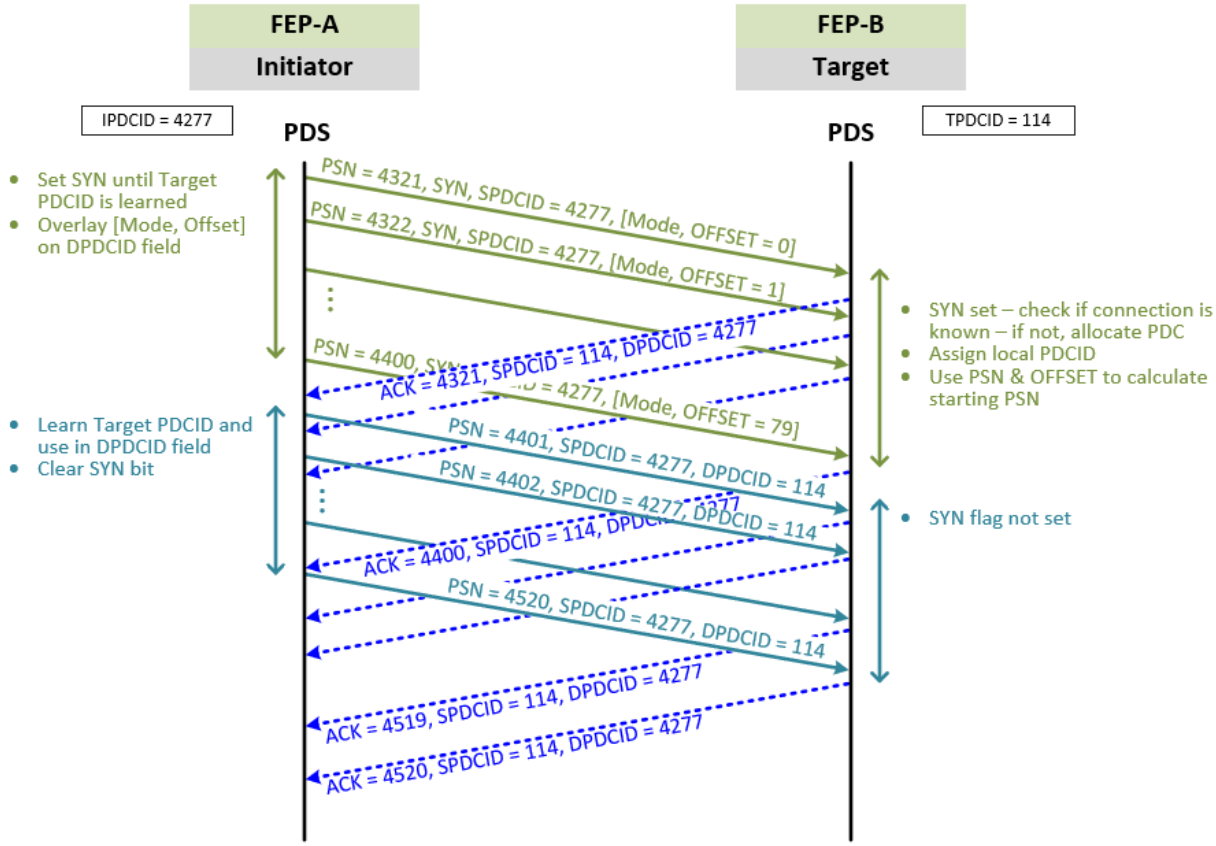


Figure 3-55 - PDC Startup using SYN Flag

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

The simplified PDC Establishment State Machine is shown in Figure 3-56. Refer to Section 3.5.9 for a more detailed example of the processing sequence.

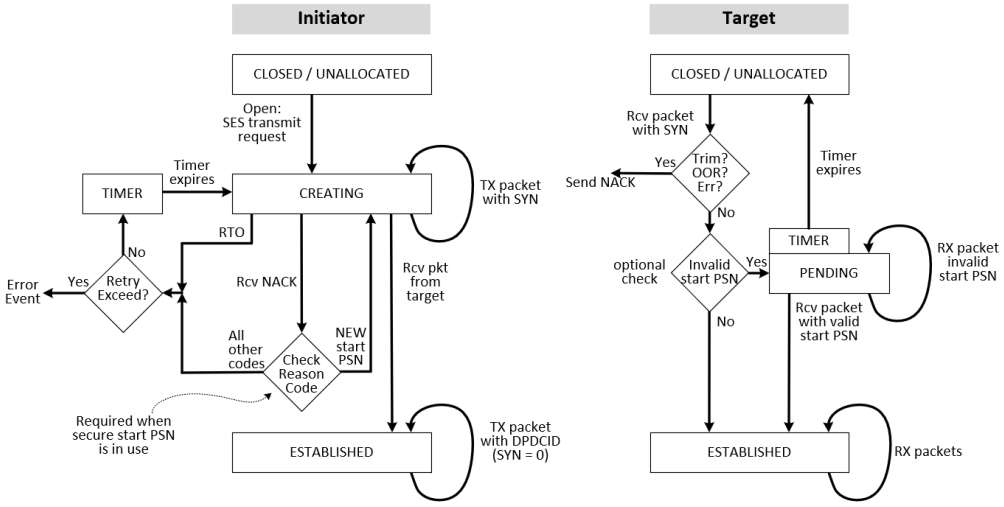


Figure 3-56 - PDC Establishment State Machine

3.5.8.2.1 PDC Establishment with Encryption

When TSS is enabled, PDC establishment packets are subject to replay attack. When using TSS, the *Start_PSN* MUST be validated—using either the **RANDOM_1RTT_START** or **EXPECTED_0RTT_START** methods—before fully establishing an encrypted PDC. The two methods interoperate with one another, falling back to target assignment of the *Start_PSN* at the cost of one additional RTT, when the two endpoints are using different methods. The two methods are not independent options. The **RANDOM_1RTT_START** method is a mandatory base capability and the **EXPECTED_0RTT_START** method an optimization layered upon it.

Two methods are defined to establish a secure *Start_PSN*. In this context, the term *Start_PSN* is used to represent different values. These values are further described as follows:

- *Initiator_Start_PSN*: The initiator's per-SDI base value for the establishing the next PDC. This value is stored in the SDKDB.
- *Target_Start_PSN*: The target's per-SDI minimum *Start_PSN* the target will accept representing the anti-replay floor. This value is stored in the SDKDB.
- *Random_Start_PSN*: A random value selected by the target in the **RANDOM_1RTT_START** method.
- *PDC_Start_PSN*: The *Start_PSN* retained in per-PDC state by both ends once the PDC is established.

Both the *Initiator_Start_PSN* and *Target_Start_PSN* are initialized to zero when the key is installed. The *PDC_Start_PSN* is a snapshot of the current *Initiator_Start_PSN* when the new PDC is established. Both the initiator and target maintain a copy of the *PDC_Start_PSN* for the lifetime of the PDC.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

3.5.8.2.1.1 RANDOM_1RTT_START

The **RANDOM_1RTT_START** method on the target ignores the initiator's start PSN and always assigns a random value. This method always incurs one additional RTT. The following sequence describes the use of the **RANDOM_1RTT_START** method:

1. An initiator begins the process of opening a PDC with a target. The initiator sends a data packet with **pds.flags.syn** = 1 and **pds.psn** = N (see Section 3.5.8.2). Alternatively, the initiator uses a NOOP CP to open the PDC (see Section 3.5.16.1).
2. The target allocates a PDC and puts it in the PENDING state. The target generates a *Random_Start_PSN* and sends a NACK with **pds.nack_code** = UET_NEW_START_PSN, **pds.spdcid** = 0, and **pds.payload.start_psn** = *Random_Start_PSN*.
3. The initiator re-initiates the PDC open process using the **pds.payload.start_psn** value received in the NACK.
4. The target accepts the PDC and moves the PDC state to ESTABLISHED.

Because this method uses a random *Start_PSN* supplied by the target, the closing ACK does not include a new *Start_PSN* when the PDC is closed.

3.5.8.2.1.2 EXPECTED_0RTT_START

The **EXPECTED_0RTT_START** method on the initiator uses a known value for the *Start_PSN* with the expectation of avoiding the extra RTT incurred by the **RANDOM_1RTT_START** method. The following sequence describes the use of the **EXPECTED_0RTT_START** method:

1. The initiator sends a data packet with **pds.flags.syn** = 1 and **pds.psn** = *Initiator_Start_PSN*.
2. A target using the **EXPECTED_0RTT_START** method will perform either of the following actions:
    a. Accept the opening of the PDC if **pds.psn** >= *Target_Start_PSN*
    b. Fallback to the **RANDOM_1RTT_START** method by allocating a PDC in the PENDING state and transmitting a NACK with **pds.nack_code** = UET_NEW_START_PSN, **pds.spdcid** = 0, and **pds.payload_start_psn** = *Target_Start_PSN*.

When the PDC is closed, the following actions are taken at the target and initiator

- Target: if the closing PDC's *PDC_Start_PSN* >= *Target_Start_PSN*, then the targets updates *Target_Start_PSN* to *PDC_Start_PSN* + 1 and returns a new *Start_PSN* value in the CP ACK payload field of the ACK to the Close Command CP (see Section 3.5.16.4.1 and Figure 3-90). The new *Start_PSN* value returned is greater than or equal to the new anti-replay floor, which is either the *Target_Start_PSN* or the highest *PDC_Start_PSN* across all open PDCs in the same SDI.
- Initiator: If the returned *Start_PSN* value from the CP ACK payload field (Figure 3-90) of the closing CP ACK is greater than *Initiator_Start_PSN*, then the initiator updates *Initiator_Start_PSN* to the received *Start_PSN*. The initiator MAY update *Initiator_Start_PSN* with any value greater than the received *Start_PSN*.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

3.5.8.2.1.3 Method Interoperability

Endpoints using different methods can still establish PDCs by falling back to target assigned *Start_PSN* values, at the cost of an extra RTT. If the initiator is using **EXPECTED_0RTT_START** and the target is using **RANDOM_1RTT_START**, the target assigns a *Random_Start_PSN* and the initiator adopts it. If the initiator is using **RANDOM_1RTT_START** and the target is using **EXPECTED_0RTT_START**, the target rejects the initial *Start_PSN* and returns the *Target_Start_PSN* to retry.

Both miss-matched combinations incur an extra RTT for PDC establishment. If the initiator was speculatively sending data before PDC establishment, a potential full BDP of traffic can be dropped and require retransmission. This loss of data can be avoided by an initiator using a NOOP control packet to establish the PDC.

Implementations SHOULD use either a single method across all FEPs or provide a configuration option permitting a slow start option (e.g., using NOOP CP) for **RANDOM_1RTT_START** initiators.

3.5.8.2.1.4 Method Applicability

The two methods are not independent implementation options. The **RANDOM_1RTT_START** method assures the target will transmit a NACK with **pds.nack_code** = UET_NEW_START_PSN, requiring the initiator to re-initiate PDC establishment. The **EXPECTED_0RTT_START** method is an optional optimization layered upon the mechanisms of the **RANDOM_1RTT_START** method.

An initiator adopts the *Start_PSN* received in the NACK **pds.payload.start_psn** field and use it to re-initiate PDC establishment. The NACK may arrive from a target using either method. A target can reject the initial PDC establishment and assign a new *Start_PSN* to the initiator using the **pds.payload.start_psn** field of a transmitted NACK. A target implementing the **EXPECTED_0RTT_START** method requires this ability for anti-replay protection since rejecting stale *Start_PSN* values uses the same NACK exchange.

A consequence of these requirements is that the mechanisms of the **RANDOM_1RTT_START** method are present in every conforming implementation. When using TSS and implementation MUST support the mechanisms of the **RANDOM_1RTT_START** method. The initiator's behavior is identical for both methods. In both cases, the initiator adopts the returned *Start_PSN* and re-initiates PDC establishment, regardless of whether that value was random or reflects the target’s anti-replay floor. Note that an implementation that supports only **RANDOM_1RTT_START** is compliant and simpler to implement because it does not require SD state or closing-ACK feedback.

3.5.8.2.1.5 Anti-Replay Scope and Shared-SD Behavior

The *Start_PSN* validation specified in Section 3.5.8.2.1 protects only the establishment of a new PDC. Once a PDC is ESTABLISHED, replay is prevented by the normal per-PDC PSN receive window, which discards duplicate PSNs, together with the encryption’s per-packet integrity. *Target_Start_PSN* defines the new interval that becomes available once a PDC closes. By advancing the floor as the PDC is released, the target ensures that no recorded establishment can be replayed to recover a retired *Start_PSN* that lies below the floor.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

A single SD may be shared by many peers. Because *Initiator_Start_PSN* and *Target_Start_PSN* are maintained per-FEP, and *PDC_Start_PSN* is per-PDC, the design operates as follows. A target FEP maintains one floor (*Target_Start_PSN*) plus the stored *PDC_Start_PSN* for each open PDC. The floor only rises—each PDC close advances it beyond that PDC's base—so a retired base is never accepted again from any initiator. An initiator FEP maintains one base (*Initiator_Start_PSN*) that rises only by adopting the largest floor returned by any peer. Establishment proceeds without an additional RTT whenever the initiator's base is at or above the target's current floor. If the target's floor has since been raised by other initiators on the shared SD, the attempt is rejected, and the NACK re-synchronizes the base.

3.5.8.2.1.6 Protection Against Resource Exhaustion

A PENDING PDC reserves resources, so replayed packets can exhaust the PDC table with half-open entries. A timer MUST be used to bound how long the PDC waits for a valid *Start_PSN*. This timer is set to *New_PDC_Time* when the first packet is sent. If the timer expires, the event MUST be counted by incrementing NEW_PDC_TIMEOUT_CNT. When this count exceeds the configured threshold, *New_PDC_Timeout_Thresh*, the error MUST be reported to the provider as NEW_PDC_TIMEOUT_ERR.

Informative Text:
UET errors are reported to the libfabric provider or management interface. Details on these will be provided in a future version.

3.5.8.2.2 PDS Lifecycle with Encryption

In order to avoid potential replay attacks, when the PSN of a PDC reaches *Start_PSN* + 2^{31}, the PDC MUST close. This allows a single PDC to carry over 2 billion packets before closing. A new PDC may then be established using one of the methods in Section 3.5.8.2.1.

If *Limit_PSN_Range* is configured FALSE, the PDC is allowed to continue beyond *Start_PSN* + 2^{31}.

The close follows the same process of completing any in-process messages.

3.5.8.3 PDC Close

A PDC is closed for multiple reasons that include a resource shortage or remaining idle for a configured amount of time. Closing a PDC is controlled by the initiator PDS but may be triggered by either the initiator or the target. When the target PDS determines the PDC should be closed, the target sends either a PDS ACK with the **pds.flags.req** field set or sends a Close Request CP. CPs are described in Section 3.5.16.

The following covers the normal scenario. If either an initiator or target FEP fails to respond or if a NACK with a fatal error is received, the peer FEP may close the PDC in error. At the initiator, failure to respond is determined when the maximum number of retry attempts is reached for any packet. At the target, the max retry attempts reached on the return direction or a local timer on the close processing may lead to PDC close in error.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Once an initiator begins the PDC closing process, it MUST NOT transmit new SES messages for that PDC. If additional messages need to be sent between the same pair of FEPs, a new PDC is established. This enables fairness when PDC resources are stressed. If the PDC being closed is a RUD PDC, the additional SES requests can be sent on another PDC immediately (either existing or newly opened). If the PDC being closed is a ROD or RODL PDC, the additional SES requests MUST be deferred until the original ROD/RODL PDC is fully closed. This maintains the ordering of messages.

Closing a PDC is triggered by either the initiator deciding to close (local FEP decision) and sending a Close Command CP or the target FEP sending a Close Request CP to the initiator.

The PDC close procedure MUST be supported as follows:

1. If the initiator decides to close the PDC, the PDC close procedure at the initiator is:
    a. Stop accepting new message requests for this PDC.
        i. A clean closing point on an SES message boundary is identified—the initiator does not send a Close Command CP in the middle of a message.
    b. Complete all SES messages that are in progress.
        i. An SES message remains in progress until all SES packets associated with the message have been received, transmitted, and responded to, including all necessary PDS acknowledgements, clears and recovery from any NACKs that arrive.
        ii. This includes receiving return data (or error) for all outstanding read requests.
    c. When all PDS ACKs for in-progress SES messages have arrived, the initiator checks the PDC for lost packets that have **pds.flags.syn**=1. If packets with **pds.flags.syn**=1 were determined to be lost, the initiator MUST choose one of the following approaches:
        i. Wait until at least *Syn_Retx_Safety_Time* has elapsed since transmission of the most recent lost packet with **pds.flags.syn** = 1, and then send a Close Command CP.
        ii. Or, if the target has assumed responsibility for tracking *Syn_Retx_Safety_Time*, immediately send a Close Command CP with the remaining time to be tracked encoded in the command payload.
            1. Determining if the target accepts responsibility for tracking the *Syn_Retx_Safety_Time* can be accomplished using a Negotiation CP at any time between the determination of loss (i.e., a loss detected by a timeout or inferred by other mechanisms) and the issuing of the Close Command CP.
    d. Wait for PDS ACK of the Close Command CP packet to arrive.
        i. Close the PDC and free resources when this ACK is received.
        ii. If this does not arrive after *Max_RTO_Retx_Cnt* retransmit, close the PDC in error.
2. The PDC close procedure at the target is:
    a. Receive a Close Command CP.
        i. The Close Command CP implicitly clears all PSNs.
        ii. There should be no active PSNs when this is received—no packets outstanding on the return direction; if the PDC is not idle then a PDC_CLOSE_IN_ERR SHOULD be recorded for diagnostic purposes.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

b. If a Close Command CP was already received and a packet with a higher PSN is received, transmit a NACK with **pds.nack_code** = UET_CLOSING.
c. Transmit a PDS ACK for the Close Command CP packet.
    i. If using UET encryption and the **EXPECTED_0RTT_START** method for *Start_PSN*, return the *Expected_PSN* with the closing ACK (See Section 3.5.8.2.1).
    ii. If the target negotiated support for tracking *Syn_Retx_Safety_Time* and the *Remaining_Time* in the Close Command CP is non-zero, the target tracks the remaining *Syn_Retx_Safety_Time*. All arriving packets with **pds.flags.syn** = 1 and the {source FA, SPDCID} associated with *Syn_Retx_Safety_Time* are NACK'd with **pds.nack_code** = UET_CLOSING.
    iii. If the target did not negotiate support for tracking *Syn_Retx_Safety_Time* or if the *Remaining_Time* in the Close Command CP is zero, no timer is required.
    iv. Close the PDC and free resources not needed for tracking *Remaining_Time* (if *Remaining _Time* is non-zero).

The Close Command CP is allocated a unique PSN—the next PSN in the sequence—and the initiator retransmits the packet if an ACK is not received. If the target receives a Close Command CP for a PDC that is already closed, the target MUST transmit a PDC NACK to the initiator.

If a target decides to close the PDC, it sends a Close Request CP. A timer—*Close_REQ_Timer*—may be used to constrain the time the initiator takes to send a Close Command CP. If the initiator does not issue a Close Command CP in response to the Close Request CP within the configured time (*Close_REQ_Time*), the target may issue a NACK with **pds.nack_code** = UET_CLOSING_IN_ERR. The PSN is set to be one larger than the highest PSN that was received, acknowledged, and cleared. The PDC is then closed.

The simplified state machine for closing a PDC is shown in Figure 3-57. At the initiator, once a PDC moves to QUIESCE state, new messages are not started. If a new message is received it is either pushed to another PDC or, if resources are stressed, it may be pushed to a pending queue (either in HW or SW) that holds packets waiting for PDC resources. Handling this is implementation specific.

In the example above, 'close events' are determined locally within a FEP and include:

1. Receiving a Close Request CP from the target.
2. Identifying limited PDC resources at the initiator and deciding to free a PDC by closing it.
3. Expiring a PDC lifetime timer.

The initiator does not need to wait for a clear from the target for any read response unless the read response is stateful. The initiator MUST wait to receive all outstanding read responses before issuing a Close Command CP.

The Close Command CP is an implicit clear of all PSNs on forward direction and ACK for all PSNs on the return direction. If a Close Command CP arrives at a target and that target has any outstanding operations, these are all terminated and the PDC is closed. If the target detects an error—such as an incomplete message—then the PDC is closed in error and the event reported.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Refer to Section 3.5.9 for a more detailed example state machine.

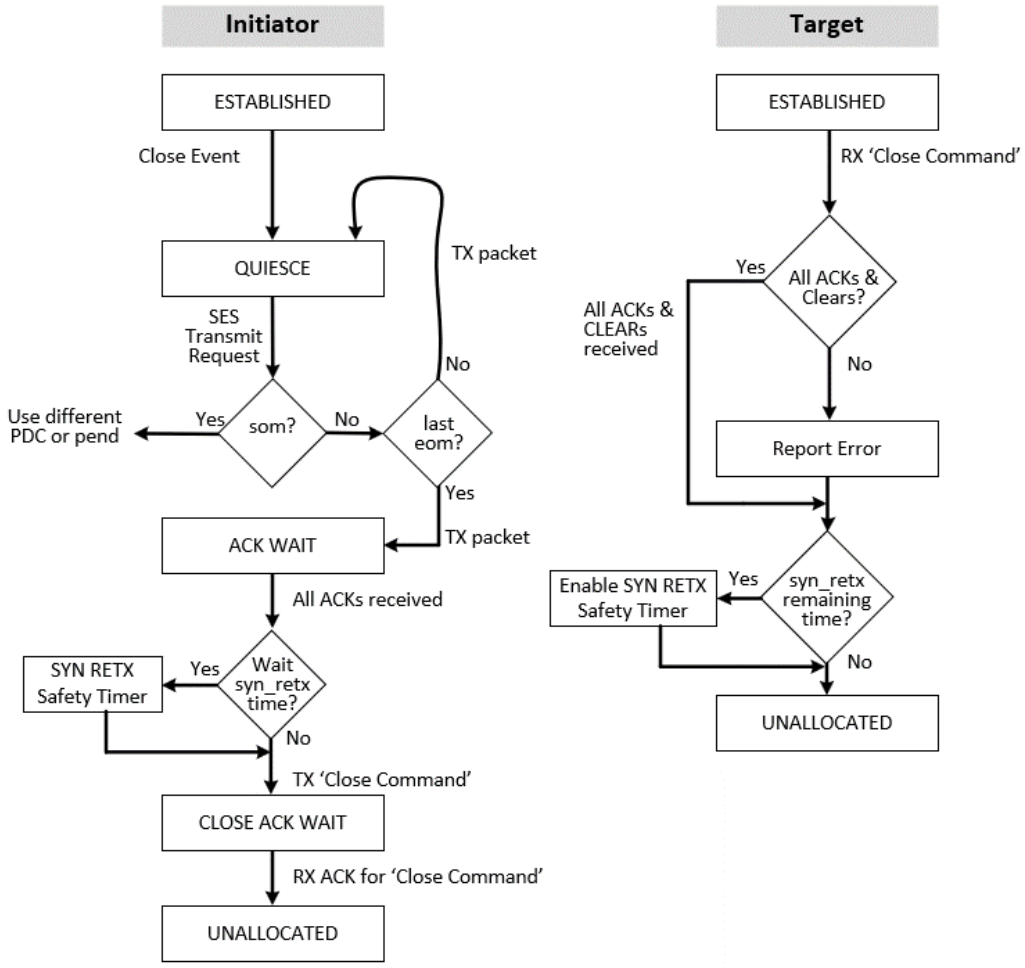


Figure 3-57 - Single PDC Close State Machine

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

3.5.8.4 PDC Lifetime

The lifetime of a PDC depends on the use case of the network. An implementation may choose to provide configuration parameters to control PDC lifetime, but these are not specified here. A PDC may exist for a very brief time—as short as the time to send a single packet and receive the associated acknowledgement plus one RTT. This could occur when there is pressure on the number of PDCs supported by an implementation. Alternatively, a PDC may exist for an extended period, as long as the life of an application (or more).

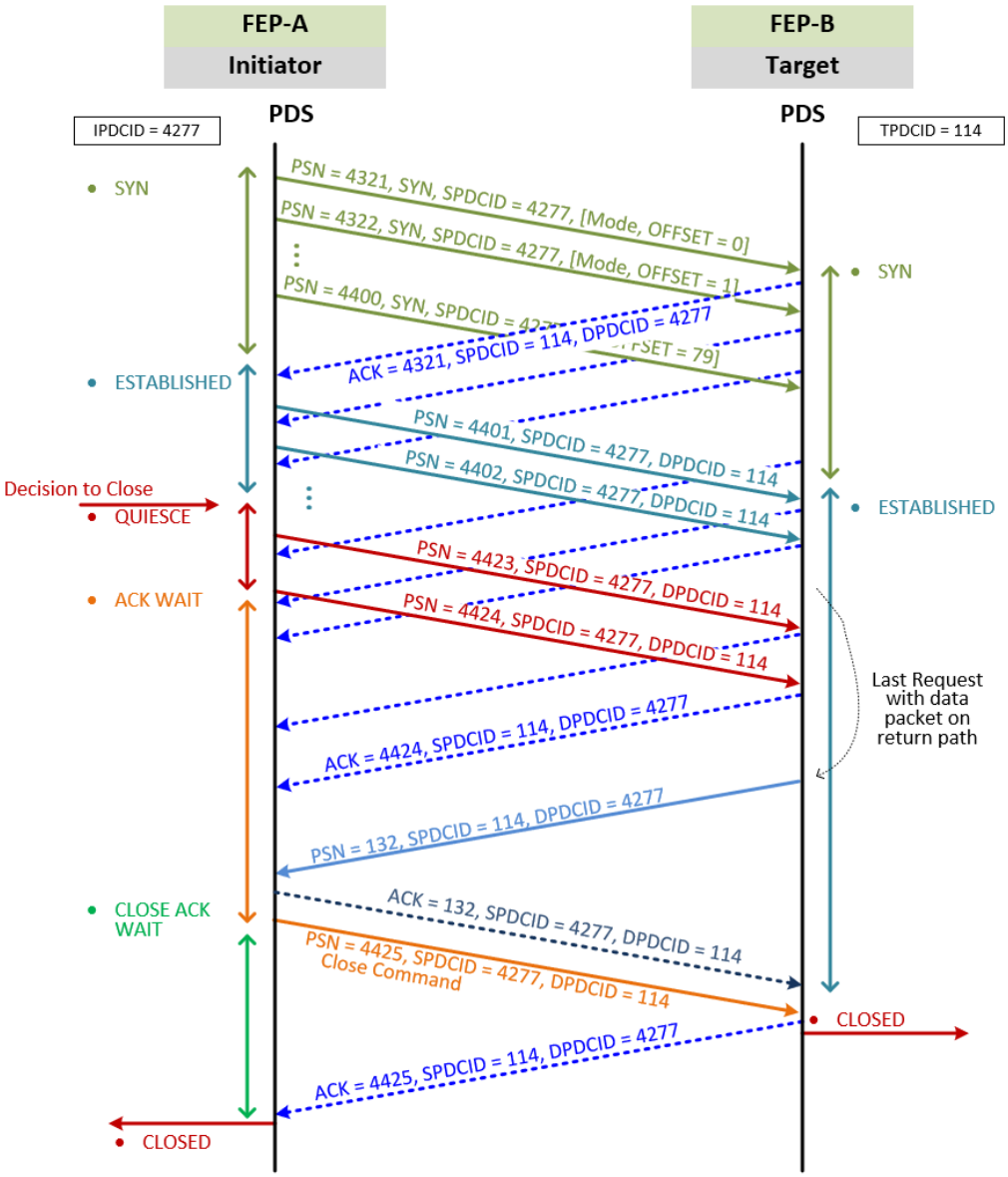


Figure 3-58 - Sequence for PDC Setup and Teardown

An example sequence for PDC setup/teardown is shown in Figure 3-58. Note the Close Command CP is sent directly after the ACK = 132, so these packets may arrive at the target in either order.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

3.5.9 PDS Event State Machine

This section describes the function of PDS as a set of state machines. The RUD protocol is the focus of these state machines. The implementation described is not required.

A state is represented as a block with associated processing within the block and arrows showing the possible next steps. Arrows from IDLE or ESTABLISHED are labeled with the event that triggers the move to the next functional block. Arrows from a functional block are labeled with the condition that determines which path to take. Some arrows use an unconditional transition designated as UCT.

Following a state machine figure are descriptions of the notable processing within each block. The abbreviations req and rsp are used for request and response, respectively. PDS behavior is described by the following four state machines:

- Top Level - Shows how the state machines are connected
- Manager State Machine - Controls PDC allocation and closing, resource management, and any errors that are not associated with an active PDC
- Initiator State Machine - PDC state machine when acting as an initiator
- Target State Machine - PDC state machine when acting as a target

The initiator and target state machines are illustrated primarily as processing sequences from the ESTABLISHED state—the largest rectangle in the center of the diagram. From there, events occur that lead to a series of processing steps and back to the ESTABLISHED state. Each line is labeled with the event that triggers the processing.

The PDC establishment state machines in Section 3.5.8 show the states through which a PDC moves. The PDC event state machines in this section shows how events are handled. Most of these events occur in all of the PDC establishment states shown in the PDC establishment state machines. Therefore, the PDC state is reflected as a variable (*State*) in these event state machines.

3.5.9.1 PDS Top Level

Figure 3-59 illustrates the relationship between the PDS manager state machine and individual PDC state machines, including the interface events to PDS from SES and TSS. The per-PDC state machines shown in Figure 3-59 include the initiator and target state machines described in sections 3.5.9.3 and 3.5.9.4.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

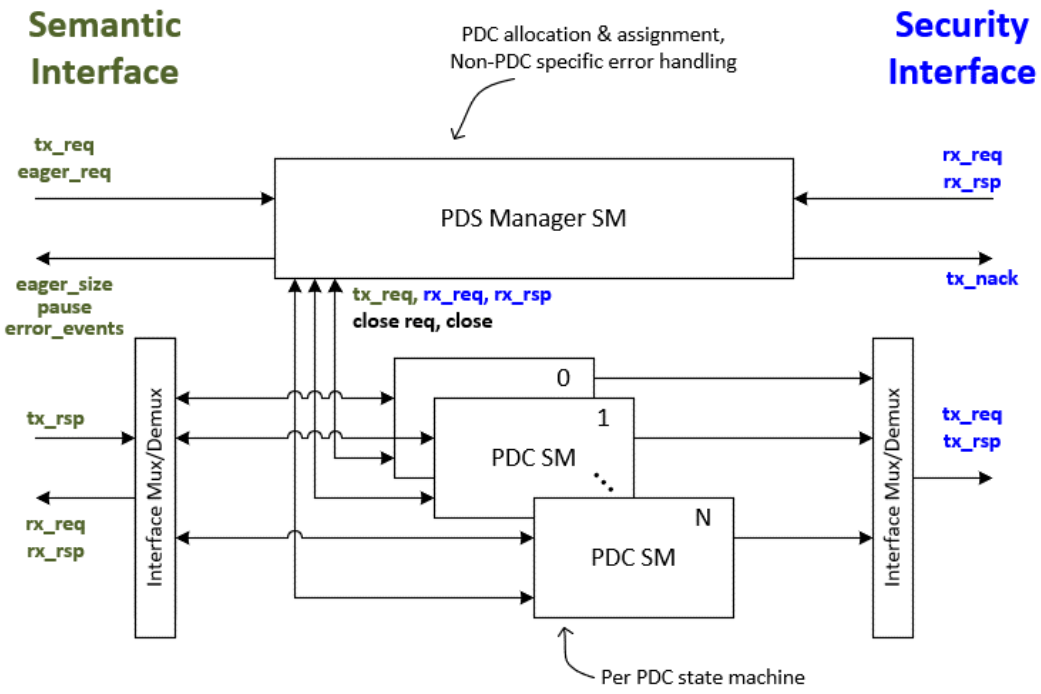


Figure 3-59 - PDS Top Level State Machines

3.5.9.2 PDS Manager

Within PDS, some functions are general and others are specific to individual PDCs. Examples of general functions include the allocation of PDCs, the handling of error events that are not associated with a specific PDC, the assignment of SES packets to PDCs, etc. These services are provided by the PDS manager state machine. Figure 3-60 illustrates the services within the PDS manager.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

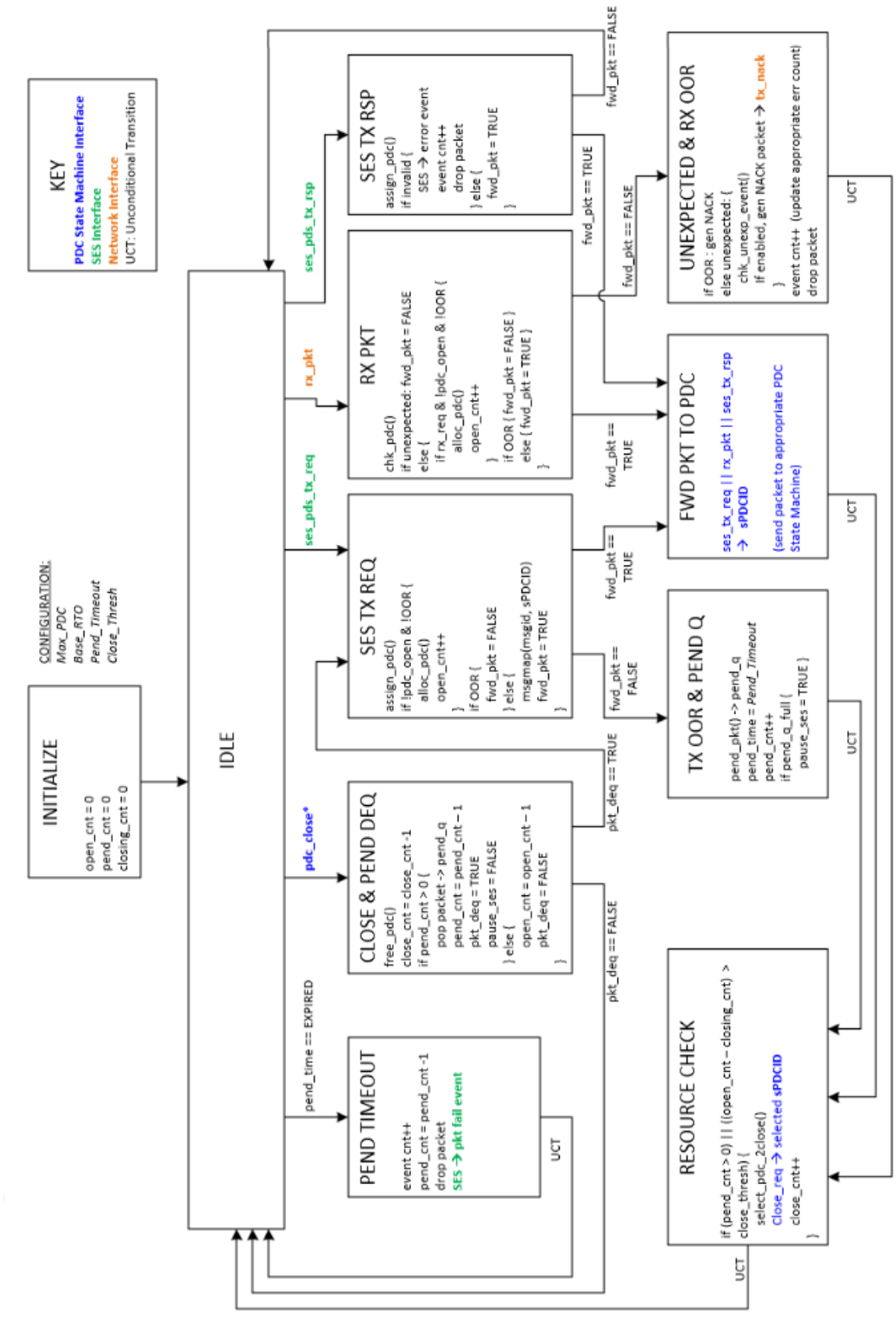


Figure 3-60 - PDS Manager State Machine

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

The PDS manager pre-processes packets from SES and the network and assigns each to a PDC state machine, or it handles the packet locally if the packet is not associated with a specific PDC. Its functions include:

- assign_pdc()
  - Checks the fields used to assign packets to PDCs (e.g., {JobID, destination FA, traffic class, delivery mode}).
  - If a PDC already exists, then forward the packet to that PDC.
  - If a PDC does not exist, then if this is a tx_req then allocate a PDC or if this is a tx_rsp then report the error to SES.
- alloc_pdc()
  - Checks if a PDC is available, allocates a new PDC, and forwards the packet to that PDC state machine for further processing.
  - If no PDC is available because PDS is out of resources (OOR), pass SES-generated packets to a pending queue or transmit a NACK for packets received over the network.
- free_pdc()
  - When a PDC is closed, reset all state except PSN as follows:
    - If not using UET security, save last PSN for use in calculating *Start_PSN* the next time a PDC is allocated (i.e., to assure the next randomly assigned PSN is at least 2^{16} distance from the last PSN as described in Section 3.5.8.2).
    - If using UET security and using **EXPECTED_0RTT_START** for new *Start_PSN*s, then follow the procedure described in Section 3.5.8.2.1.
    - If using UET security and using **RANDOM_1RTT_START** for new *Start_PSNs*, no need to save state, as a random new *Start_PSN* will be generated during the next PDC creation.
- msgmap()
  - Associates each message ID with a PDC on start of message (*som*).
  - All packets in a message go to the same PDC; remove mapping on end of message (*eom*) or, if locked (e.g., while waiting for read return data), when the unlock event is received.
- pend_pkt()
  - Implements a pending queue that holds up to N packets waiting for PDC resources.
  - When a packet is pushed to the pending queue, start closing a PDC to allow the PDC to be reallocated; there are many ways this can be implemented.
- select_pdc_2close()
  - Selects a PDC to be closed when there are packets in the pending queue, thus freeing resources to transmit the pending packets.
  - Ideally an idle PDC is closed, otherwise choose randomly or use an implementation specific policy.
  - When the number of unallocated PDCs reaches a configured threshold, this function is called to begin closing PDCs. The recently closed PDCs create a pool of available PDCs for packets arriving from SES to new destinations.
- chk_pdc()

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

    - Verifies that packets received from the network are valid, checks for unexpected events that are not associated with a specific PDC (e.g., NACK_NO_CONN or NACK_INV_DPDCID).
    - If invalid, then do not forward the packet to a PDC state machine.
- chk_unexp_event()
    - Checks the type of unexpected event and determine whether to transmit a NACK.

3.5.9.3 PDC Initiator State Machine

Figure 3-61 provides an illustrative example skeleton of a PDS state machine that manages the initiator side of a PDC. The states shown in the simplified PDC establishment and close state machines of Figure 3-56 and Figure 3-57 are represented by the *State* variable shown in Figure 3-61.

A PDC initiator state machine handles packets from SES to the target and packets arriving from the target. Its functions include:

- chk_tx_bitmap()
    - Checks the distance between the lowest unacknowledged PSN and the highest acknowledged PSN.
    - If the distance reaches the maximum PSN range (MP_RANGE), then pause the PDC until the lower PSN(s) are acknowledged.
- update_ccc()
    - Pass the received bytes, round-trip time, etc. to the CCC.
    - The *send_req()* process is constrained by CC scheduling.
- update_tx_psn_tracker()
    - Update the PSN bitmap to reflect which packets have been acknowledged.
    - Determine if a Clear Request or ACK Request CP should be transmitted; or if a request packet should be retransmitted (note that this is not called out in the state machine).
- process_rx_req()
    - Process a read response at the initiator.
- update_rx_psn_tracker()
    - Update the PSN bitmap to reflect which packets have been received.
    - Determine if an ACK should be transmitted.
- gen_ack()
    - Check the criteria for generating an ACK; if TRUE, then transmit an ACK packet.
- send_close()
    - Send a Close Request CP to the target. This is done by sending a Close Command CP.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

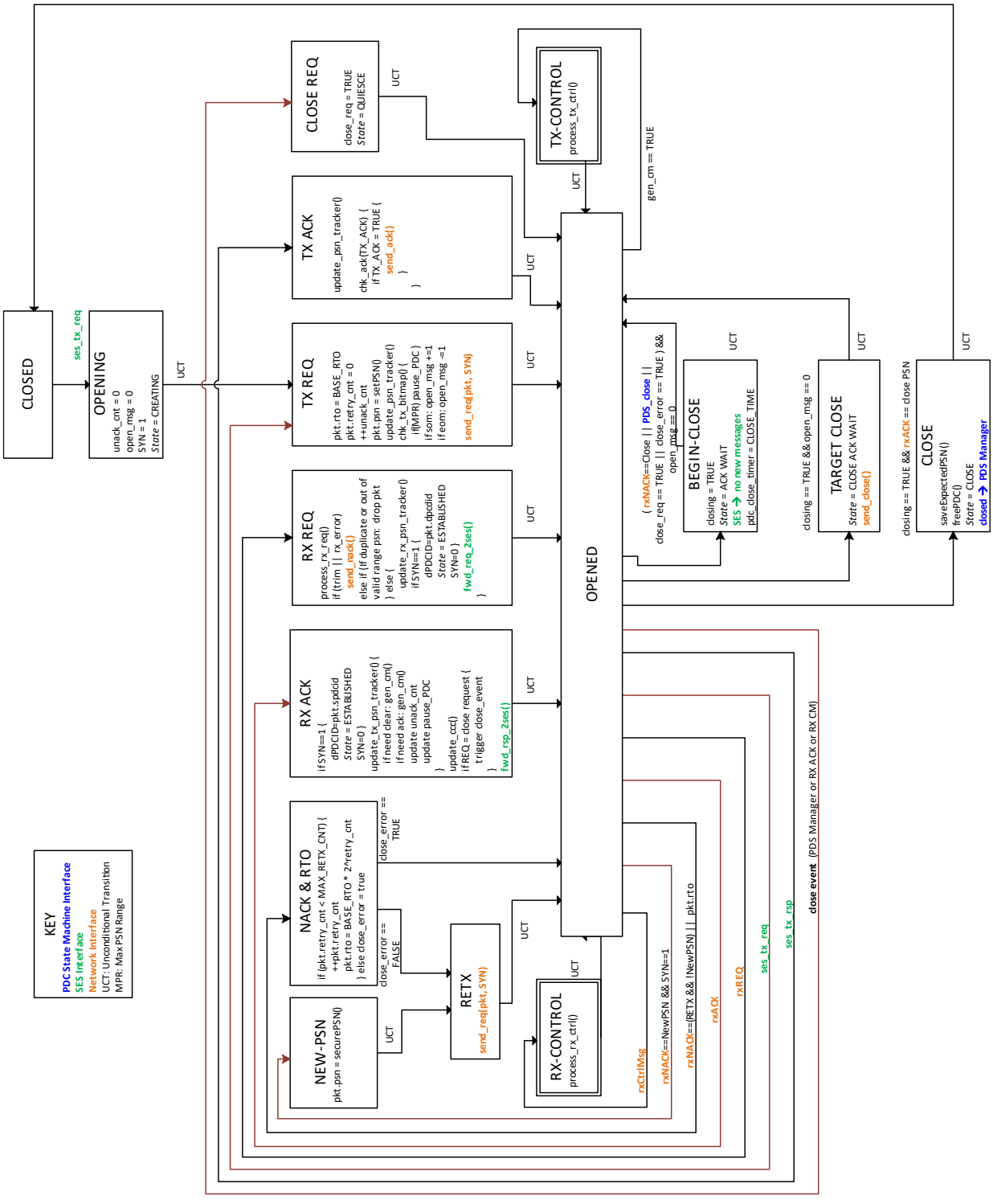


Figure 3-61 - PDC Initiator State Machine

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

3.5.9.4 PDC Target State Machine

Figure 3-62 illustrates example processing at the target on a per-PDC basis. Most state machine processes are to the same as the initiator state machine processes with the following exception:

- chk_secure_psn()
  - If secure PSN is enabled, this process checks if the *Start_PSN* from the initiator is acceptable; if invalid, a NACK with the alternate *Start_PSN* is transmitted.

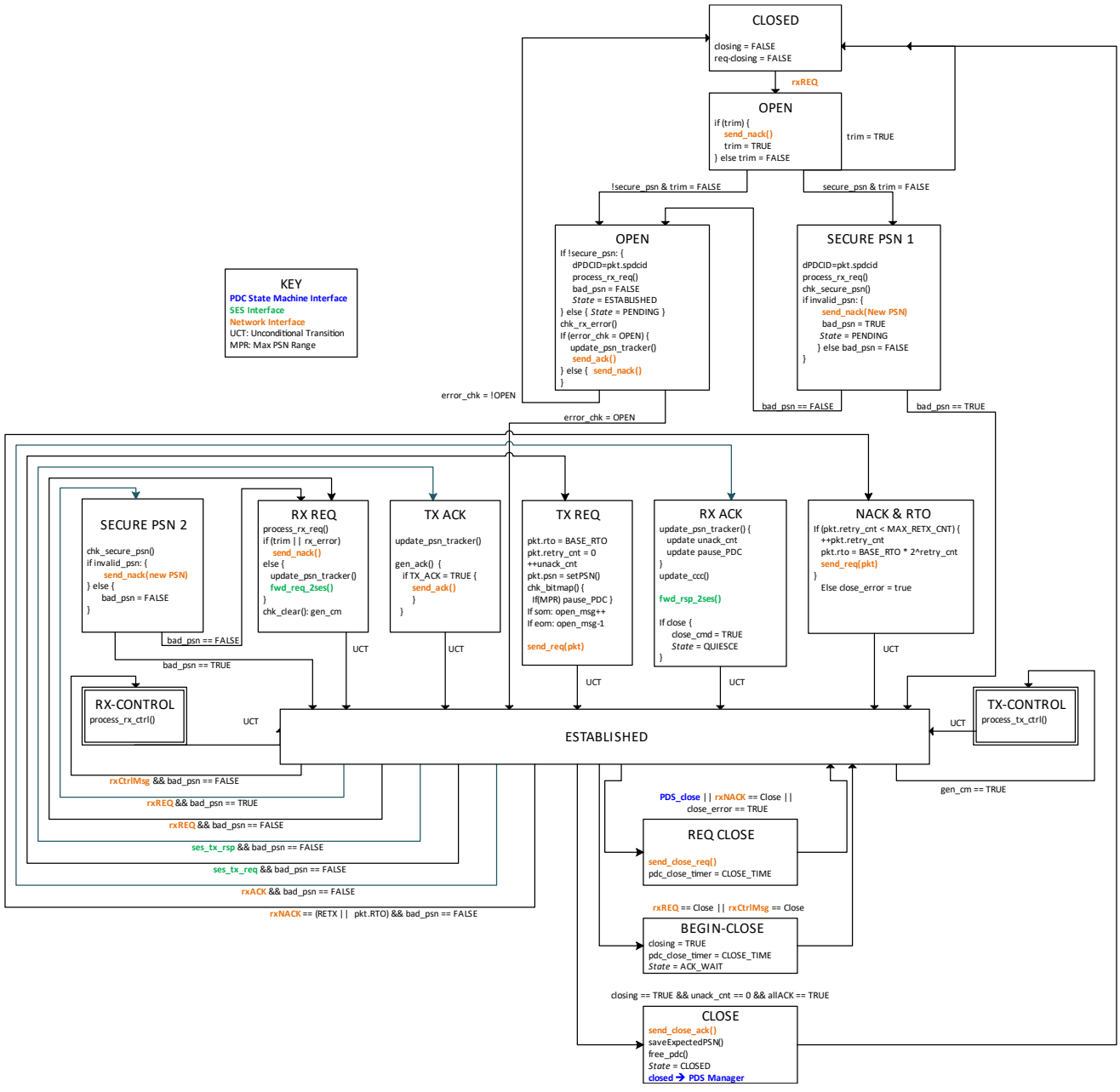


Figure 3-62 - PDS Target State Machine

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

3.5.10 Header Formats

This section specifies the PDS header formats. All UET packets MUST use the header formats defined in this section.

Formats are defined for the following headers:

- Entropy
- RUD/ROD Request
- RODL Request
- Acknowledgement
- ACK with CC
- ACK with extended CC
- CP
- RUDI Request/Response
- Negative Acknowledgement
- UUD Request

In all PDS headers, reserved fields MUST be set to zero on transmission and ignored on reception.

Section 3.5.10 presents the header fields in a compact tabular format with a brief description of each field. Section 3.5.11 provides more detailed descriptions of each field.

3.5.10.1 UET Entropy Header

The UET entropy header is used when running UET directly over IPv4 or IPv6, without UDP. It appears directly after the IP header. The format of the UET entropy header is shown in Figure 3-63. The header fields are described in Table 3-38.

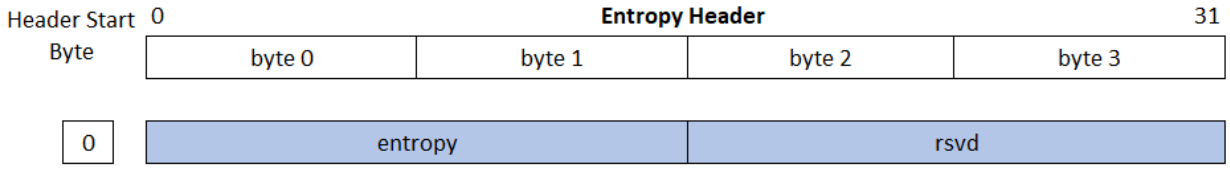


Figure 3-63 - UET Entropy Header Format

Table 3-38 - Fields of UET Entropy Header

Field Name	Size (in bits)	Field Description
entropy	16	Entropy value used — typically by fabric switches — for path selection • Positioning at this location places the entropy value in the same packet location as the UDP source port when UDP is not present
rsvd	16	Reserved

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Native IPv4, native IPv6, and UDP encapsulations are specified. The PDS header formats are the same for all encapsulations. UET FEPs MUST support generating and accepting UET packets using IPv4, IPv6, and UDP encapsulation. Deployments may restrict which encapsulation type(s) are enabled.

When UDP encapsulation is used, the **udp.src_port** field is used as the entropy. The UET entropy header MUST NOT be present when using UDP. When UDP is used, the **udp.checksum** MUST be set to 0 on send and ignored on receive.

When native IPv4 or IPv6 encapsulation is used, the UET entropy header MUST be used. Implementations MUST allow the **ip.protocol** field in the IP header to be configured.

IP fragmentation is not supported when using UET. FEPs MUST NOT fragment a packet. When using IPv4, the **ipv4.flags.df** bit (don't fragment) is set, and the **ipv4.flags.mf** bit (more fragments) and **ipv4.fragment_offset** field is cleared to zero on send.

3.5.10.2 UET UFH Entropy Header

The UET UFH entropy header is used when running UET directly over the UFH_32 header. The format of the UET UFH entropy header is shown in Figure 3-64. The header field is the same as described in Table 3-38; however, the location of the field does not align with the location of a UDP source port and instead is located directly after the EtherType field where the EtherType is 0xE790 identifying UET.

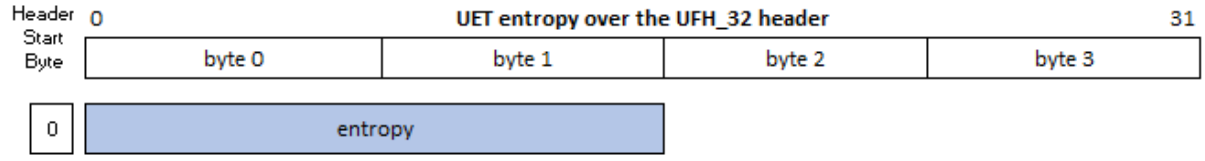


Figure 3-64 - UET UFH Entropy Header

The UET UFH entropy header MUST be present when running UET directly over the UFH_32 header. If using any layer 2 tags such at VLAN or CSIG, these are located before the EtherType indicating UET.

3.5.10.3 PDS Prologue

The first two bytes at the beginning of all PDS Headers except RODL uses the format shown in Figure 3-65. The header fields are described in Table 3-39.

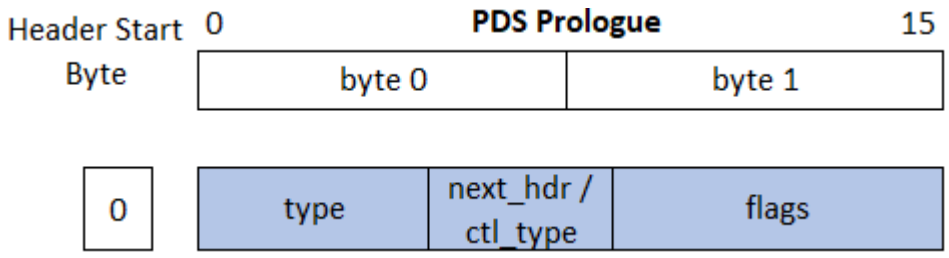


Figure 3-65 - PDS Prologue Format

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

This section defines the **pds.type** encodings.

Table 3-39 - Fields of PDS Prologue

Field Name	Size (in bits)	Field Description
type	5	Encodings are: • 0 => Reserved • 1 => UET Encryption Header (TSS) • 2 => RUD Request (RUD_REQ) • 3 => ROD Request (ROD_REQ) • 4 => RUDI Request (RUDI_REQ) • 5 => RUDI Response (RUDI_RESP) • 6 => UUD Request (UUD_REQ) • 7 => ACK • 8 => ACK_CC • 9 => ACK_CCX • 10 => NACK • 11 => CP (Subtype encoded in CTL_TYPE field) • 12 => NACK_CCX • 13 => RUD_CC_REQ • 14 => ROD_CC_REQ • 15 => RODL Request (RODL_REQ) • 16-31 => Reserved
next_hdr or ctl_type	4	Type-specific field • When **pds.type** = CP, this field is ctl_type • When **pds.type** != CP, this field is next_hdr as passed in from SES to PDS (e.g., in the example *ses_pds_tx_req()* API in Section 3.5.4)
flags	7	Type-specific flags See Section 3.5.11.8.

Encodings for the **pds.next_hdr** field are defined in the semantic sublayer Section 3.4.3.8. One codepoint, 0x00, is reserved for PDS packets that do not carry an SES header and UET payload. Encodings of the **pds.ctl_type** field are defined in Section 3.5.10.9.

3.5.10.4 RUD/ROD Request

The PDS header format for a RUD or ROD Request packet is shown in Figure 3-66. The header fields are described in Table 3-40.

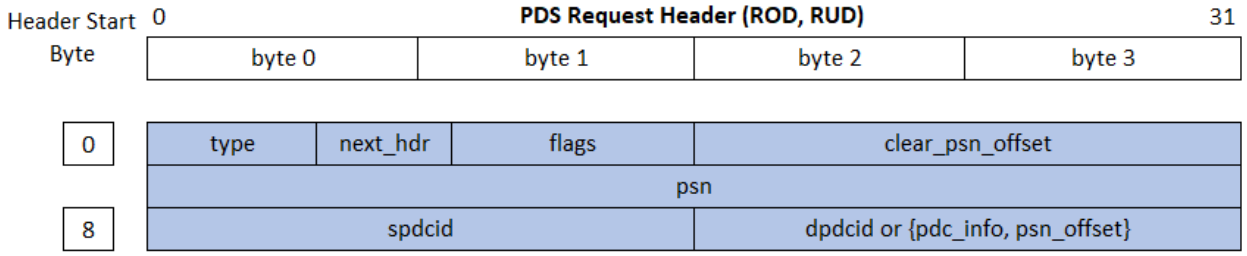


Figure 3-66 - RUD/ROD Request Header Format

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Table 3-40 - Header Fields for RUD/ROD Request

Field Name	Size (in bits)	Field Description
type	5	Packet Type = ROD Request or RUD Request
next_hdr	4	Encoding identifying UET Semantic type (Refer to SES Section 3.4.3.8)
flags (7 bits) • rsvd • retx • ar • syn • rsvd	 2 1 1 1 2	**pds.flags** = [rsvd, retx, ar, syn, rsvd] • Reserved • 1 => This packet is a retransmit (Refer to Section 3.5.11.8.4) • 1 => ACK Request, when set an ACK is requested to be sent for this packet (Refer to Section 3.5.12.2) • 1 => PDC establishment request • Reserved
clear_psn_offset	16	Encoding of *CLEAR_PSN* relative to PSN This field is a sequence number used to acknowledge reception of PDS ACKs
psn	32	Packet sequence number assigned to the PDS Request
spdcid	16	Source PDCID; PDCID assigned by FEP that is source of the packet
{dpdcid} or {pdc_info, psn_offset}	16 4 12	This field is overloaded as specified in Section 3.5.8.2 • When **pds.flags.syn** = 0: ○ Destination PDCID assigned by FEP that is the destination of the packet • When **pds.flags.syn** = 1: {pds.pdc_info, pds.psn_offset} Encoding for **pds.pdc_info** = [rsvd, rsvd, rsvd, use_rsvd_pdc] • **pds.pdc_info.reserved**(3) • **pds.pdc_info.use_rsvd_pdc** 1 = use PDC from reserved pool 0 = use PDC from global shared pool pds.psn_offset is the numerical difference between the PSN in this packet and the *Start_PSN* on the PDC; refer to Section 3.5.8.2

3.5.10.5 RODL Request

The PDS header format for a RODL Request packet is shown in Figure 3-67. The header fields are described in Table 3-41.

When using RODL over the UFH_16 header, for maximized wire efficiency, the RODL header is compressed to the format shown in Figure 3-68. The header fields are described in Table 3-42.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

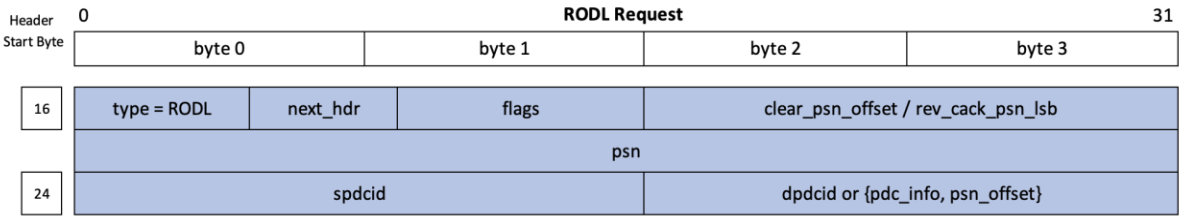


Figure 3-67 - RODL Header Format - General

Table 3-41 - RODL Request Fields

Field Name	Size (in bits)	Field Description
type	5	Packet Type = RODL
next_hdr	4	Encoding identifying UET Semantic type
flags (7 bits) • rsvd • cack • retx • ar • syn • rsvd	 1 1 1 1 1 2	**pds.flags** = [rsvd, retx, ar, syn, rsvd] • Reserved • 1 => **pds.rev_cack_psn_lsb** replaces **pds.clear_psn_offset** • 1 => This packet is a retransmit (Refer to Section 3.5.11.8.4) • 1 => ACK Request, when set an ACK is to be sent for this packet (Refer to Section 3.5.12.2) • 1 => PDC establishment request • Reserved
clear_psn_offset or rev_cack_psn_lsb	16	If **pds.flags.cack** = 0: **pds.clear_psn_offset** is the same as ROD/RUD If **pds.flags.cack** = 1: **pds.rev_cack_psn_lsb** are the lower 16 bits of the *CACK_PSN* for the reverse direction.
psn	32	Same as ROD/RUD
spdcid	16	Same as ROD/RUD
dpdcid or {pdc_info, psn_offset}	16 16	Same as ROD/RUD

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

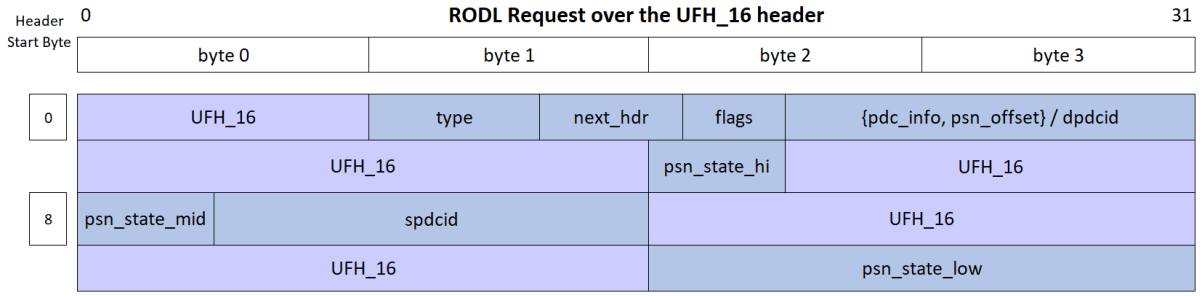


Figure 3-68 - RODL Header Format over UFH_16

Table 3-42 - RODL Request Fields over UFH_16

Field Name	Size (in bits)	Field Description
type	5	Packet Type = RODL
next_hdr	4	Encoding identifying UET Semantic type
flags (3 bits): over UFH_16 • retx • ar • syn	 1 1 1	**pds.flags** = [**retx, ar, syn**] • 1 => This packet is a retransmit • 1 => ACK Request, when set an ACK is requested to be sent for this packet • 1 => PDC establishment request
spdcid	12	Source PDCID; PDCID assigned by FEP that is source of the packet
dpdcid or {pdc_info, psn_offset}	12 12	This field is overloaded • When **pds.flags.syn** = 0: ○ PDCID assigned by FEP that is the destination of the packet • When **pds.flags.syn** = 1: **{pds.pdc_info(2), pds.psn_offset(10)}** **pds.pdc_info:** **pds.pdc_info.rsvd(1)** **pds.pdc_info.use_rsv_pdc** 1 = use PDC from reserved pool 0 = use PDC from global shared pool **pds.psn_offset** is the numerical difference between the PSN in this packet and the *Start_PSN* on the PDC. The 10-bit size limits the number of packets sent using **pds.flags.syn** = 1 to 1022.
psn_state	24	The content of this field depends on **pds.flags.syn**. When **pds.flags.syn** = 1, *PSN* = {0x00, psn_state } When **pds.flags.syn** = 0, **pds.clear_psn_offset**, **pds.rev_cack_psn_lsb}** are present. In the UFH_16 header: **pds.psn_state** = { **psn_state_hi, psn_state_mid, psn_state_low** }

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Field Name	Size (in bits)	Field Description
		with **psn_state_low** incrementing with every new *PSN* (refer to Section 3.5.25.9)

3.5.10.6 RUD/ROD Request with CC State

The PDS header format for a RUD or ROD Request packet with congestion control state is shown in Figure 3-69. The header fields are described in Table 3-43.

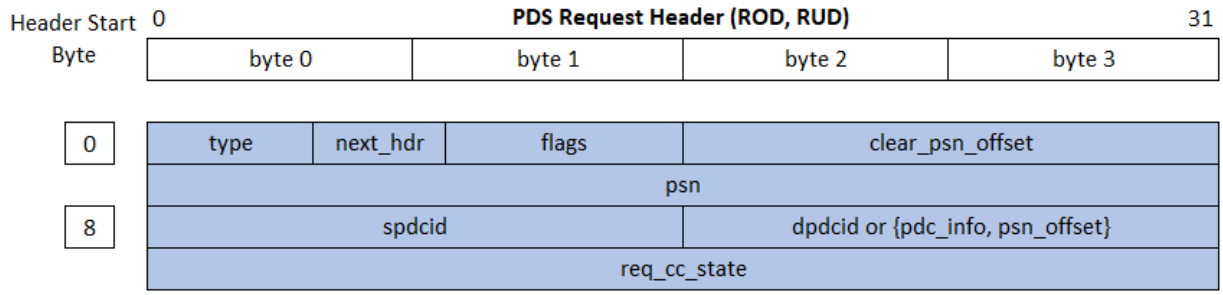


Figure 3-69 - RUD/ROD Request Header with CC State Format

Table 3-43 - Header Fields for RUD/ROD Request with CC State

Field Name	Size (in bits)	Field Description
type	5	Packet Type = ROD Request or RUD Request with CC state
next_hdr	4	Same as RUD or ROD Request
flags (7 bits)	7	**pds.flags** = [rsvd, retx, ar, syn, rsvd] • Same as RUD or ROD Request
clear_psn_offset	16	Same as RUD or ROD Request
psn	32	Same as RUD or ROD Request
spdcid	16	Same as RUD or ROD Request
{dpdcid} or {pdc_info,psn_offset}	16	Same as RUD or ROD Request
req_cc_state	32	The contents of this field are defined in the Section 3.6.9.1.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

3.5.10.7 ACK

The PDS header format for a RUD/ROD acknowledgment packet is shown in Figure 3-70. The header fields are described in Table 3-44.

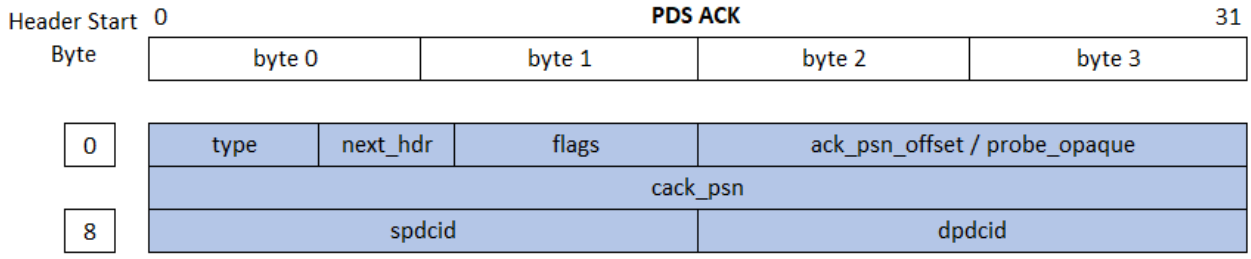


Figure 3-70 - ACK Header Format

Table 3-44 - Header Fields for ACK

Field Name	Size (in bits)	Field Description
type	5	Packet Type = PDS ACK
next_hdr	4	Encoding identifying UET Semantic type (Refer to SES Section 3.4.3.8)
flags (7 bits) • rsvd • m • retx • p • req • rsvd	 1 1 1 1 2 1	**pds.flags** = [rsvd, m, retx, p, req, rsvd] • Reserved • 1 => Associated request packet was ECN marked • 1 => This is an ACK for a packet with **pds.flags.retx** = 1 • 1 => This ACK is for a Probe CP; **pds.ack_psn_offset** and **pds.cack_psn** are ignored • Requests a clear or close; see Section 3.5.11.8.6 • Reserved
ack_psn_offset	16	Signed representation of the offset from *CACK_PSN* to *ACK_PSN*. This field encodes the PSN of the packet that triggered the generation of the ACK packet. Done to reduce the size of the field from 32 bits to 16 bits.
probe_opaque	16	Used when an ACK is generated for a Probe CP, copied from the Probe CP into the ACK; see Section 3.5.16.4.2.
cack_psn	32	Cumulative acknowledgement packet sequence number—all PDS requests with PSN prior and including this PSN are acknowledged. On the forward direction, this includes all PSNs prior, and including this PSN, that require a clear have been cleared.
spdcid	16	Source PDCID assigned by FEP that is the source of the packet
dpdcid	16	Destination PDCID assigned by FEP that is the destination of the packet

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

3.5.10.8 ACK_CC

The ACK with congestion control format is used to carry information for the congestion management sublayer. When NSCC is used, the ACK_CC and/or ACK_CCX are used.

The ACK_CC format is shown in Figure 3-71. The ACK_CCX fields are described in Table 3-45.

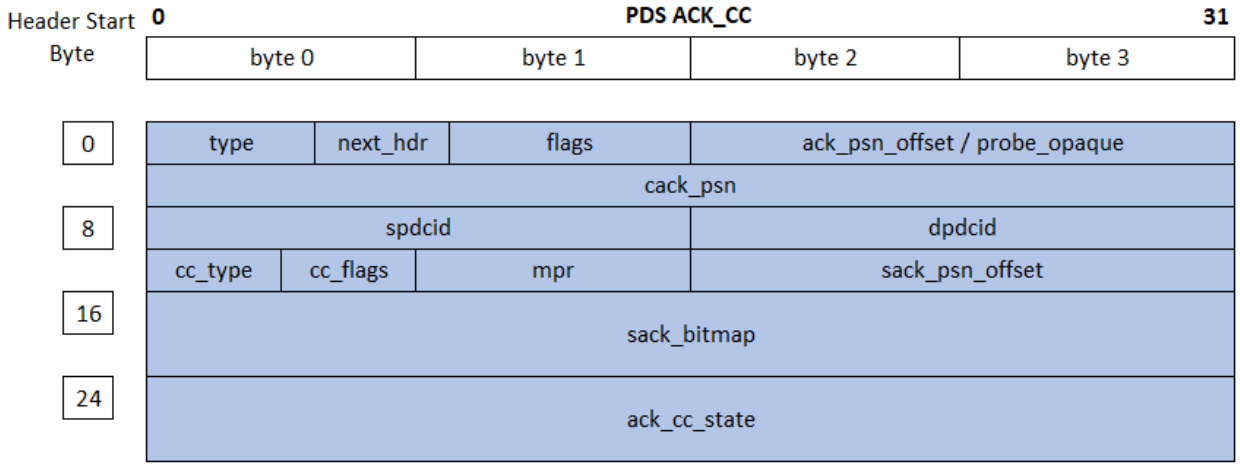


Figure 3-71 - ACK_CC Format

Table 3-45 - Header Fields for ACK_CC

Field Name	Field Size (in bits)	Field Description
First 12 bytes	96	Identical to PDS ACK header but with **pds.type** set to ACK_CC
cc_type	4	This field defines the contents of **pds.ack_cc_state** field that is used to support CC algorithms; defined in CMS Section 3.6.9
cc_flags • rsvd	4	**pds.cc_flags** = [rsvd] • Reserved
mpr	8	Maximum PSN range—used to set a maximum window on source PDC; defines the maximum number of outstanding (not yet cleared) packets tracked at destination
sack_psn_offset	16	Signed representation of the offset from *CACK_PSN* to *SACK_PSN*. Done to reduce the size of the field from 32 bits to 16 bits. Base *SACK_PSN* defines the lowest PSN in the SACK bitmap.
sack_bitmap	64	Selective ACK Bitmap, 1 => ACK, each bit in the map represents one PSN in the range: [*SACK_PSN* … *SACK_PSN*+63].
ack_cc_state (64b)	64	Congestion control state generated at the destination and carried to the source; content defined by **pds.cc_type** field.

3.5.10.9 ACK_CCX

The extended congestion control ACK format is one of the tools used to enable flexibility in the congestion management schemes as those schemes evolve in the future. When NSCC is used, at least one of ACK_CC or ACK_CCX is used.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

The ACK_CCX (congestion control eXtended) format is shown in Figure 3-72. The header fields are described in Table 3-46.

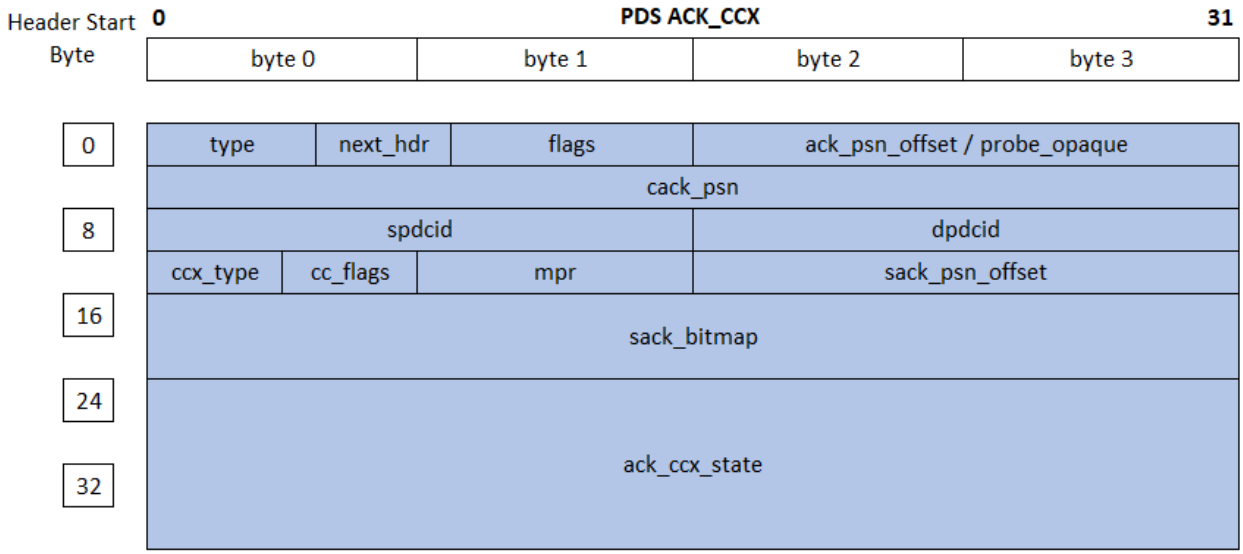


Figure 3-72 - ACK_CCX Format

Table 3-46 - Header Fields for ACK_CCX

Field Name	Field Size (in bits)	Field Description
First 12 bytes	96	Identical to PDS ACK header but with **pds.type** set to ACK_CCX
ccx_type	4	This field defines the contents of **pds.ack_ccx_state** field used to support multiple CC algorithms; defined in CMS Section 3.6.9.
cc_flags • rsvd	4	**pds.cc_flags** = [rsvd] • Reserved
mpr	8	Same as ACK_CC
sack_psn_offset	16	Same as ACK_CC
sack_bitmap	64	Same as ACK_CC
ack_ccx_state (128 b)	128	Congestion control state generated at the destination and carried to the source; content defined by the **pds.ccx_type** field. The **pds.ack_ccx_state** field is defined for future extensibility.

3.5.10.10 Control Packet

The PDS header format for a control packet is shown in Figure 3-73. The header fields are described in Table 3-47.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

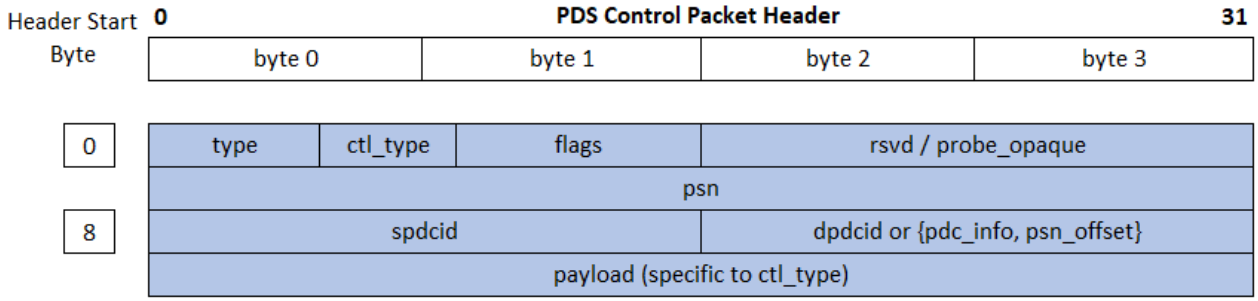


Figure 3-73 - Control Packet Header Format

Table 3-47 - Header Fields for Control Packet

Field Name	Size (in bits)	Field Description
type	5	Packet Type = CP
ctl_type	4	Identifies the type of control packet: (See Section 3.5.16) • 0 => NOOP • 1 => ACK Request ○ Source requesting an ACK for a specific PSN • 2 => Clear Command ○ From initiator to target to clear guaranteed delivery PDS ACK state at target • 3 => Clear Request ○ Target requests source to send clear • 4 => Close Command ○ Initiator indicating the PDC is being closed • 5 => Close Request ○ Target request to initiator to close the PDC • 6 => Probe ○ Source to destination to request PDS ACK • 7 => Credit ○ Destination to source carrying congestion control credit • 8 => Credit Request ○ Source to destination requesting credit • 9 => Negotiation • 10-15 => Reserved
flags (7 bits) • pdc_type • retx • ar • syn • rsvd	 2 1 1 1 2	**pds.flags** = [pdc_type, retx, ar, syn, rsvd] • 00 => RUD 01 => ROD 10 => RODL 11 => Reserved • 1 => This CP is a retransmit • 1 => ACK Request • 1 => PDC establishment request • Reserved
probe_opaque	16	Same as ACK

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Field Name	Size (in bits)	Field Description
psn	32	Packet sequence number assigned to the PDS CP. Some CPs consume a new PSN; refer to Section 3.5.16 for specifics
spdcid	16	Same as ACK
{dpdcid} or {pdc_info, psn_offset}	16 4 12	Same as ACK
payload	variable	Refer to Section 3.5.16.9

3.5.10.11 RUDI Request/Response

The PDS header format for a RUDI Request/Response packet is shown in Figure 3-74. The header fields are described in Table 3-48.

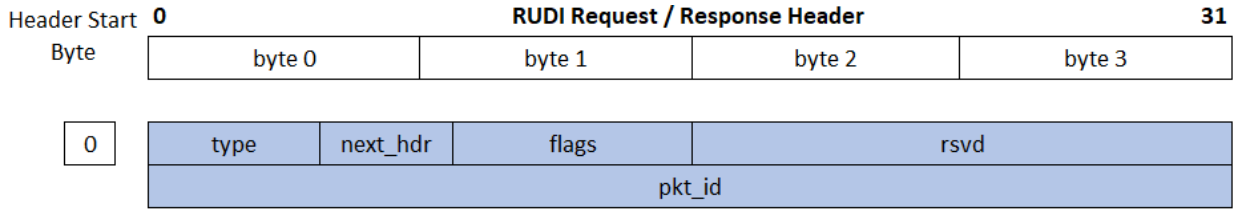


Figure 3-74 - RUDI Request Header Format

Table 3-48 - Header Fields for RUDI Request /Response

Field Name	Size (in bits)	Field Description
type	5	Packet Type = RUDI Request/Response
next_hdr	4	Encoding identifying UET Semantic type
flags (7 bits) • rsvd • rsvd / m • retx • rsvd	 1 1 1 4	**pds.flags** = [rsvd, rsvd/m, retx, rsvd] • Reserved • 1 => Associated request packet was ECN marked (Reserved for RUDI Requests, **pds.flags.m** for RUDI Responses) • 1 => This is a request/response for a retransmitted packet • Reserved
pkt_id	32	RUDI Packet Identifier: Locally unique on source • This is not a sequence number, see Section 3.5.22

3.5.10.12 NACK

The header format for a NACK packet is shown in Figure 3-75. The header fields are described in Table 3-49.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

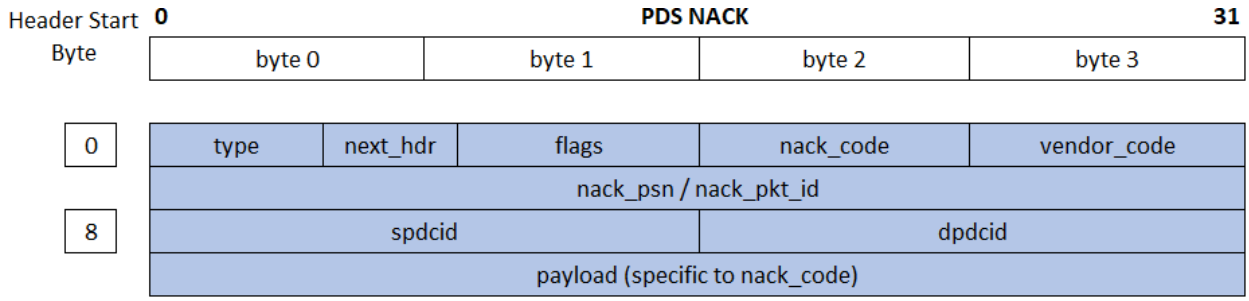


Figure 3-75 - NACK Header Format

Table 3-49 - Header Fields for NACK

Field Name	Size (in bits)	Field Description
type	5	Packet Type = PDS NACK
next_hdr	4	Encoding identifying UET Semantic type (Refer to SES Section 3.4.3.8) • Always set to UET_HDR_NONE for NACKs
flags(7 bits) • rsvd • m • retx • nt • rsvd	 1 1 1 1 3	**pds.flags** = [rsvd, m, retx, nt, rsvd] • Reserved • 1 => Associated request packet was ECN marked • 1 => The NACK'd packet was a retransmit (Refer to Section 3.5.11.8.4) • 0 => RUD/ROD, 1 = RUDI [NACK TYPE] • Reserved
nack_code	8	Field indicating why the NACK was transmitted Enumerated in Section 3.5.12.7
vendor_code	8	Vendor-specific field, no processing requirements and no interop required—intended to allow vendors to collect statistics on event types, etc. This field may be ignored. Interpretation of this field is outside the scope of this document.
nack_psn or nack_pkt_id	32	Packet sequence number associated with the received packet that triggered the generation of the NACK packet. If **pds.flags.nt** is set, this field is **pds.nack_pkt_id**—the NACK'd packet was a RUDI packet. The **pds.spdcid** and **pds.dpdcid** fields are not valid.
spdcid	16	Source PDCID assigned by FEP that is the source of the packet • If **pds.nack_code** indicates PDC establishment failure (e.g., out of PDC resources), this field is cleared to 0 and is not valid. Cleared to 0x0 if **pds.flags.nt** is set.
dpdcid	16	Destination PDCID assigned by the FEP that is the destination of the packet. Cleared to 0x0 if **pds.flags.nt** is set.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Field Name	Size (in bits)	Field Description
payload	32	Specific to **pds.nack_code**

3.5.10.13 NACK_CCX

The header format for a NACK_CCX packet is shown in Figure 3-76. The header fields are described in Table 3-50. Similar to ACK_CCX described in Section 3.5.10.8, NACK_CCX provides flexibility to enable evolution of congestion management schemes in the future.

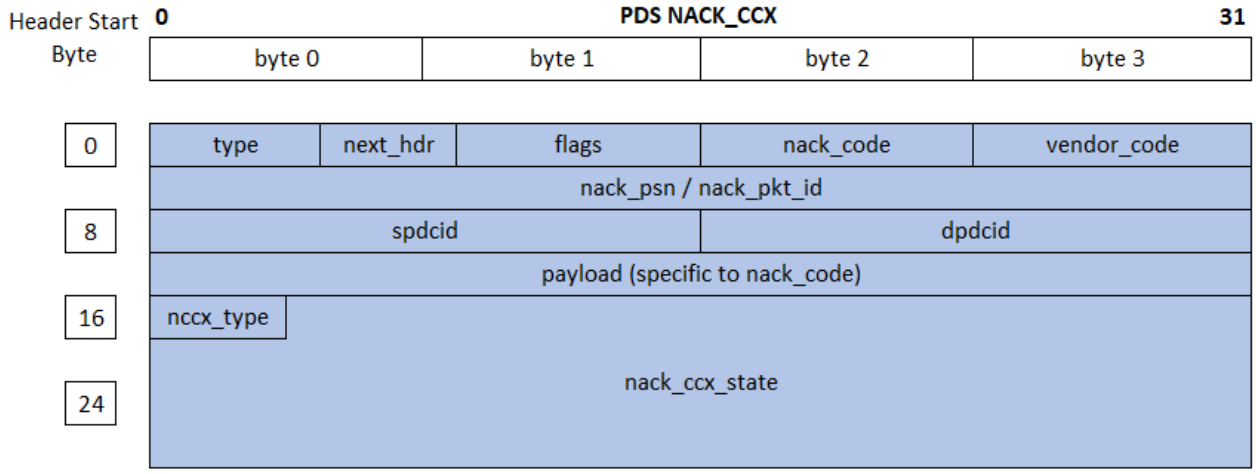


Figure 3-76 - NACK_CCX Header Format

Table 3-50 - Header Fields for NACK_CCX

Field Name	Size (in bits)	Field Description
First 12 bytes	96	Identical to PDS NACK header but with **pds.type** set to NACK_CCX
nccx_type	4	Encoding identifying contents of **pds.nack_ccx_state** • Refer to Section 3.5.11.12.
nack_ccx_state	124	Congestion control state generated at the destination and carried to the source; content defined by **pds.type** and **pds.nccx_type** fields. The **pds.nack_ccx_state** field is defined for future extensibility.

3.5.10.14 UUD Request

The PDS UUD Request Header, shown in Figure 3-77, uses a compact 4-byte format. The header fields are described in Table 3-51.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

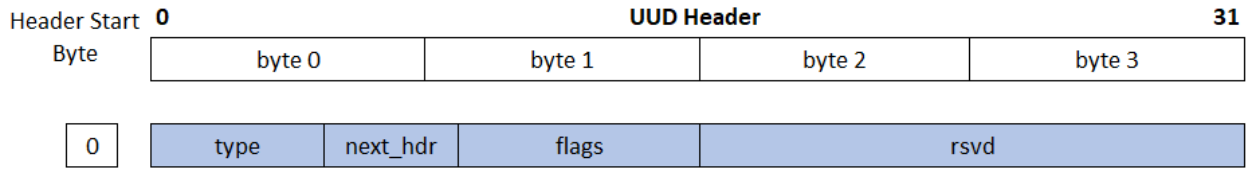


Figure 3-77 - UUD Header Format

Table 3-51 - Header fields for UUD Request

Field Name	Size (in bits)	Field Description
type	5	Packet Type = UUD Request
next_hdr	4	Encoding identifying UET Semantic type
flags	7	No flags are defined for UUD packets. Set to 0x0 on transmission and ignored on reception.
rsvd	16	Reserved

3.5.10.15 RUD/ROD Default Response SES Header

The format of the default SES response header is shown in Figure 3-78. This section describes how PDS formats a UET_DEFAULT_RESPONSE. The SES discussion of UET_DEFAULT_RESPONSE is in Section 3.4.4.3. When a packet is received with a PSN that has already been received, and when an ACK Request CP is received for a PSN that has already been received, an ACK is generated. If that PSN has a guaranteed delivery SES response, that response is sent. If that PSN has a default SES response, the ACK is generated with the SES header shown in Figure 3-78. The header fields are described in Table 3-69 with additional details in Section 3.5.17.

Default SES headers alleviate the need to store the SES response for every packet until the associated clear arrives.

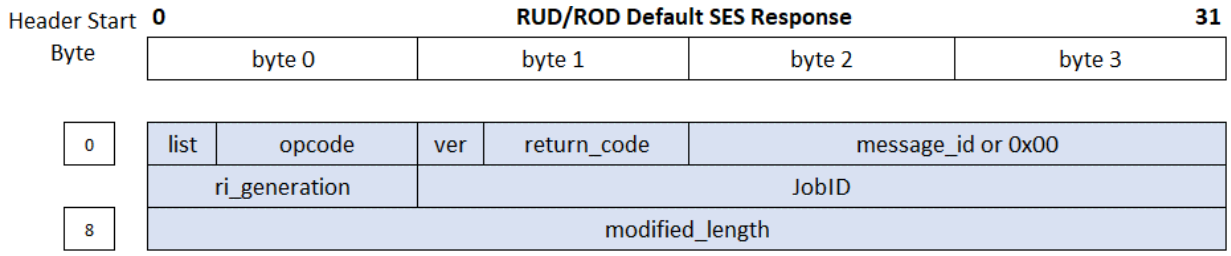


Figure 3-78 - RUD/ROD Default SES Header Format

3.5.11 Header Fields

This section provides additional description of the header fields specified in Section 3.5.10.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

3.5.11.1 pds.type

The UET **pds.type** field indicates whether the following header is a TSS or a PDS header. If the header is for PDS, the **pds.type** fields defines the reliability mode (RUD, ROD, RODL, RUDI, UUD) and PDS packet type (Request, Response, ACK, NACK, CP) as defined in Section 3.5.10.

If a packet is received with a **pds.type** value that is not recognized, the error counter PDS_TYPE_INVALID MUST be incremented and the packet MUST be dropped without generating a response.

3.5.11.2 pds.next_hdr

The **pds.next_hdr** field is reserved for use by SES. This field is passed to PDS by SES. When a PDC creates a packet autonomously (i.e., not at the behest of SES, such as an ACK or CP), this field is set to UET_HDR_NONE. There is no interaction with SES in this case.

3.5.11.3 pds.ctl_type

PDS CPs carry a control type, **pds.ctl_type**, indicating the type of CP. CPs are defined in Section 3.5.16.

If a CP arrives that uses a **pds.ctl_type** field that is not recognized, an error counter, PDS_CTL_TYPE_INVALID, MUST be incremented. The packet MUST be dropped without generating a response packet. The PDC state is not affected.

Unknown types are dropped rather than NACK'd, because some CPs use PSN = 0 and NACK relies on a valid PSN.

3.5.11.4 Packet Sequence Numbers

RUD, ROD, and RODL reliable delivery modes use PSNs—packet sequence numbers—to uniquely identify packets, assure delivery, eliminate duplicate packets, and establish packet ordering. PDS uses multiple PSNs to track packets, including a unique *PSN* per PDS Request, an acknowledgement PSN (*ACK_PSN*), a cumulative acknowledgement PSN (*CACK_PSN*), and a clear PSN (*CLEAR_PSN*).

Informative Text:
The PSNs and associated offsets denoted in italic font (e.g., *ACK_PSN, CACK_PSN, CLEAR_PSN*) are referencing the local working variable versions of the PSNs used by each PDC. When the PSN fields of the packet are referenced, they are denoted in bold font (e.g., **pds.psn, pds.clear_psn_offset**)

The full 32-bit range of packet sequence numbers is used. At 800 Gbps, a 4194 B Ethernet packet with takes roughly 42 nsec to transmit. A 32-bit PSN will wrap around in 180 seconds. Minimum size frames could lead to wrap in under 4 seconds. The 32-bit size enables future increases in link speed beyond 800 Gbps without wrapping within a 100 millisecond RTT.

In the above example, the packet size is determined as follows: Ethernet (14 B) + IPv4 (20 B) + UDP (8 B) + PDS (12 B) + SES (44 B) + UET payload (4096 B) = 4194 B.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

3.5.11.4.1 pds.psn

The **pds.psn** field refers to the packet sequence number assigned to RUD/ROD/RODL PDS Request packets and certain CPs. The **pds.psn** field is monotonically increased by one for each packet on each PDC direction except for specific CPs (as described in Section 3.5.16). The *PSN* increases independently on the forward and return directions.

3.5.11.4.2 pds.psn_offset

The **pds.psn_offset** field is present during PDC creation, when the **pds.flags.syn** bit is set. This field allows the destination to determine the *Start_PSN* of the PDC as described in Section 3.5.8.2.

3.5.11.4.3 CLEAR_PSN

The clear sequence number, *CLEAR_PSN*, MUST be supported when using RUD/ROD/RODL. *CLEAR_PSN* indicates that all PSNs up to and including this PSN have been cleared. Clearing a PSN involves confirming the receipt of the PDS ACK for a packet. A *CLEAR_PSN* of X indicates that acknowledgements for the packet with PSN = X and all packets with earlier PSNs have been received.

The purpose of *CLEAR_PSN* is to enable guaranteed delivery of certain SES responses. When SES indicates that an SES Response requires reliable delivery, PDS stores the response until a clear is received. This is done to support retransmitting the associated ACK if necessary. When PDS receives a clear, any SES Response state associated with a PSN equal to or lower than *CLEAR_PSN* is deleted.

CLEAR_PSN is indirectly carried in the RUD/ROD Request header using the **pds.clear_psn_offset** field. The **pds.clear_psn_offset** field is a 16-bit signed integer in twos-complement format that is added to **pds.psn** to determine *CLEAR_PSN*. The msb of **pds.clear_psn_offset** is extended to create the 32-bit signed value added to **pds.psn**. See the examples below in the description for *ACK_PSN*. This offset is used to reduce the size of the field from 32 bits to 16 bits.

CLEAR_PSN is reduced in size for RODL over the UFH_16 header. Refer to Section 3.5.10.5 and Section 3.5.22.1.

CLEAR_PSN is directly carried in PDS CPs as part of the **pds.payload** field. Refer to Section 3.5.16.3

CLEAR_PSN MUST be initialized to the *Start_PSN* minus 1 on the forward and return directions.

Informative Text:
The **pds.clear_psn_offset** field will never be a positive number, as *CLEAR_PSN* is never higher than *PSN*. The 16-bit signed integer format is used to align the processing of this field with other sequence numbers that are carried as offsets.

When an SES response includes data (e.g., a small read response carried in a PDS ACK), the data is not stored. A lost PDS ACK that carried read response data is handled by retransmitting the original read request packet.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

CLEAR_PSN is maintained as the highest clear sequence number received in each direction on a PDC. The *CLEAR_PSN* will always monotonically increase. The *CLEAR_PSN* state on the forward and return directions are independent.

When SES indicates that an SES Response requires guaranteed delivery, PDS requests a clear by setting **pds.flags.req** to REQ_CLEAR in the associated PDC ACK.

PDS provides a *CLEAR_PSN* in response to receiving an ACK with **pds.flags.req** set to REQ_CLEAR or receiving a Clear Request CP. PDS provides the *CLEAR_PSN* either by carrying **pds.clear_psn_offset** in an outgoing PDS Request or, if there are no requests pending, by sending a Clear Command CP.

When no requests are pending, the Clear Command CP MUST be sent either immediately or after a local timer expires to allow *CLEAR_PSN* to progress at the destination. A timer SHOULD be used and initialized to *PDS_Clear_Time*.

3.5.11.4.4 CACK_PSN

The cumulative ACK PSN (*CACK_PSN*) MUST be supported for RUD/ROD/RODL PDCs. The **pds.cack_psn** field is carried in full 32-bit format in the PDS ACK header field on RUD/ROD. *CACK_PSN* is reduced in size on RODL packets. See Section 3.5.10.5 and Section 3.5.12.4.4.

CACK_PSN is defined such that all earlier PSNs up to and including *CACK_PSN* have been successfully received, and all corresponding guaranteed delivery SES responses have been cleared. The *CACK_PSN* MUST NOT advance beyond the lowest PSN requiring guaranteed delivery that has not been cleared by CLEAR_PSN. This is described in more detail in Section 3.5.12.4.4.

CACK_PSN MUST be initialized to the PDC's *Start_PSN* minus 1 when a PDC is created.

3.5.11.4.5 ACK_PSN

The acknowledgement PSN, *ACK_PSN*, identifies the packet that triggered the generation of an ACK.

ACK_PSN is indirectly carried in the RUD/ROD/RODL ACK header using the **pds.ack_psn_offset** field. The **pds.ack_psn_offset** field is a 16-bit signed integer in twos-complement format that is added to *CACK_PSN* to determine *ACK_PSN*. The msb of **pds.ack_psn_offset** is extended to create the 32-bit value added to **pds.psn**. See the examples in Table 3-52.

Table 3-52 - Example PSN OFFSET Calculation

ACK_PSN_OFFSET	CACK_PSN	ACK_PSN
0x0024	0x62231120	0x62231144
0xFFDC (-36)	0x62231120	0x622310FC
0x0010	0xFFFFFFF3	0x00000003
0xFEE2 (-286)	0x00000079	0xFFFFFF5B

3.5.11.4.6 SACK_PSN

SACK_PSN is the base or lowest PSN in the **pds.sack_bitmap** field of a RUD/ROD ACK_CC and ACK_CCX header, as described in Section 3.5.12.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

SACK_PSN is indirectly carried in the RUD/ROD ACK_CC and ACK_CCX headers using the **pds.sack_psn_offset** field. The **pds.sack_psn_offset** field is a 16-bit signed integer in twos-complement format that is added to *CACK_PSN* to determine *SACK_PSN*. See the examples in the *ACK_PSN* description in Table 3-52.

3.5.11.4.7 New_Start_PSN

New_Start_PSN is used by the target to send the initiator a valid starting PSN (*Start_PSN*). It is used with encryption to negotiate a *Start_PSN* during creation of a new PDC. Refer to Section 3.5.8.2.

3.5.11.4.8 pds.pkt_id

The RUDI delivery mode uses a unique identifier in the PDS header called **pds.pkd_id** that is not required to be monotonically increasing and may be assigned in any order. This field is assigned to each RUDI request by the source and is reflected back in the RUDI response.

3.5.11.5 PDC Identifiers

Both the initiator and target allocate PDC Identifiers. Refer to the PDS terminology in Section 3.5.2 and the informative text box at the end of that section for additional information on PDCIDs.

- Initiator PDC Identifier (*IPDCID*): Allocated by initiator
- Target PDC Identifier (*TPDCID*): Allocated by target
- Source PDC Identifier (*SPDCID*): Carried in the **pds.spdcid** field. This is *IPDCID* or *TPDCID* depending on which FEP was source and which is destination for the packet
- Destination PDC Identifier (*DPDCID*): Carried in the **pds.dpdcid** field. This is *IPDCID* or *TPDCID* depending on which FEP was source and which is destination for the packet

A PDCID is intended to be locally unique for each FEP. A source FEP can reuse the same PDCID for different destination FEPs. A source FEP does not simultaneously use the same PDCID to the same destination FEP even if the delivery mode or traffic class is different.

3.5.11.6 pds.pdc_info

When a PDC is first created and until it reaches the ESTABLISHED state (see Section 3.5.8.2), the **pds.dpdcid** field is overloaded with {**pds.pdc_info, pds.psn_offset**}. The **pds.pdc_info** field is used to allow the target to determine the properties of the PDC. This 4-bit field is defined as follows:

pds.pdc_info = [rsvd, rsvd, rsvd, use_rsvd_pdc]

- **rsvd (3)** Reserved for future use
- **use_rsvd_pdc** when this bit is set, the PDC resources are taken from the reserved pool; when this bit is zero, the PDC resources are taken from the general pool

The reserved pool is described in Section 3.5.18. The **pds.pdc_info** field is reduced to two bits in RODL.

3.5.11.7 pds.req_cc_state

Refer to the CMS Section 3.6.9.1 for a description of the **pds.req_cc_state** field.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

3.5.11.8 pds.flags

This field is **pds.type** specific and includes control bits that indicate the start of a PDC, the presence of optional fields such as congestion control information, etc. UET implementations parse associated header fields based on the state of these flag bits.

3.5.11.8.1 pds.flags.pdc_type

This field is used in the NOOP CP and Negotiation CPs to indicate if the associated PDC is RUD, ROD, or RODL. This field is ignored in all other CPs.

- 'b00 RUD
- 'b01 ROD
- 'b10 RODL
- 'b11 reserved

This flag is ignored in all other CPs.

3.5.11.8.2 pds.flags.syn

This flag in the RUD/ROD/RODL Request header is set during PDC establishment. Refer to Section 3.5.8.2. Once a PDC is established, the target PDCID is learned by the initiator. This flag is cleared once the target PDCID is learned.

3.5.11.8.3 pds.flags.ar

If the ACK Request CP field (**pds.flags.ar**) is set in the RUD/ROD/RODL Request header, the source requests the destination to transmit a PDS ACK. Upon receiving a PDS header with the **pds.flags.ar** bit set, the destination transmits a PDS ACK with the **pds.ack_psn** field set to the **pds.psn** field in the received packet.

The **pds.flags.ar** bit SHOULD be set on the last packet transmitted when the PDC has no additional work (i.e., no packets pending for transmission). The congestion control algorithm determines other conditions for setting the **pds.flags.ar** bit.

3.5.11.8.4 pds.flags.retx

The **pds.flags.retx** flag MUST be set in the RUD/ROD/RODL Request header of a retransmitted packet. This flag triggers the generation of a PDS ACK. A destination should prioritize accepting retransmitted packets.

The **pds.flags.retx** flag is used by congestion control algorithms. Refer to the CMS sections 3.6.12.3 and 3.6.13.5 for more information.

3.5.11.8.5 pds.flags.m

The **pds.flags.m** flag MUST be set if and only if the packet that triggered the response was ECN-marked. This indicates that the triggering packet experienced congestion. Refer to the CMS sections 3.6.10.3 and 3.6.16.4 for details on how the M flag is used at a source for congestion control and path-aware multipath spraying.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

When enabled by configuration parameter *ACK_On_ECN*, ECN marked packets always trigger an ACK or NACK. In the case of an ACK, the ACK packet cannot be coalesced.

3.5.11.8.6 pds.flags.req

This 2-bit **pds.flags.req** field in the RUD/ROD/RODL ACK is used to indicate the target has a request for the initiator. An initiator always sets the **pds.flags.req** field to zero. The field is defined as:

Table 3-53 - REQ Field Definition

REQ State	Mnemonic	Description
'b00	NO_REQUEST	No request
'b01	REQ_CLEAR	Target requires a clear for this PSN • Set when ACK carries a guaranteed delivery SES Response
'b10	REQ_CLOSE	Target requests the initiator to close the PDC
'b11	Reserved	Invalid, drop this packet and report event, PDC_REQ_ERR

3.5.11.8.7 pds.flags.p

The probe (**pds.flags.p**) flag is used in an ACK message to indicate the ACK was triggered by a Probe CP. The *ACK_PSN* value is not valid when the **pds.flags.p** flag is set.

3.5.11.8.8 pds.flags.nt

The NACK type (**pds.flags.nt**) flag in the NACK header indicates the type of packet being NACK'd. If **pds.flags.nt** is cleared, the NACK is for RUD, ROD, or RODL. If **pds.flags.nt** is set, the NACK is for a RUDI packet.

3.5.11.8.9 pds.flags.cack

The **pds.flags.cack** field is used in the RODL general request format (3.5.10.5) to indicate that the **pds.rev_cack_psn_lsb** field is present and represents the lower 16 bits of the *CACK_PSN* for the reverse direction. If the flag is cleared, the **pds.clear_psn_offset** is present and operates the same as with ROD/RUD.

3.5.11.8.10 Reserved Flags

Reserved flags are reserved for future use. Reserved flags are set to zero on transmission and ignoreded on receipt.

3.5.11.8.11 Mapping pds.flags Fields to pds.type

Table 3-54 and Table 3-55 show the definition of sub-fields of the **pds.flags** field in the UET headers dependent upon the setting of the **pds.type** field. A dash indicates the bit is reserved (i.e., not used).

Table 3-54 - pds.flags by pds.type

Type	PDS Type							
0	Reserved	rsvd	rsvd	rsvd	rsvd	rsvd	rsvd	rsvd
1	TSS encryption	Refer to TSS Section 3.7.11						
2-3	RUD/ROD Request	rsvd	rsvd	retx	ar	syn	rsvd	rsvd
4	RUDI Request	rsvd	rsvd	retx	rsvd	rsvd	rsvd	rsvd

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Type	PDS Type							
5	RUDI Response	rsvd	m	retx	rsvd	rsvd	rsvd	rsvd
6	UUD Request	rsvd	rsvd	rsvd	rsvd	rsvd	rsvd	rsvd
7-9	RUD/ROD ACK	rsvd	m	retx	p	req		rsvd
10	NACK	rsvd	m	retx	nt	rsvd	rsvd	rsvd
11	CP	pdc_type		retx	ar	syn	rsvd	rsvd
15	RODL – General	rsvd	cack	retx	ar	syn	rsvd	rsvd

Table 3-55 - pds.flags for RODL over UFH_16

Type	PDS Type			
15	RODL over UFH_16	retx	ar	syn

3.5.11.9 pds.entropy

The **pds.entropy** field is used by network switches to select a path through the network. UET provides two methods for carrying this entropy field depending on the encapsulation:

- **udp.src_port** when UDP encapsulation is used.
- A UE entropy header (**pds.entropy**) that immediately follows an IPv4 or IPv6 header when native IPv4 or IPv6 encapsulation is used or immediately follows the EtherType 0xE790 when UET runs directly over Ethernet.

The congestion management function assigns the **pds.entropy** field used in each request packet. ACK packets use the **pds.entropy** field from the packet that triggered the ACK generation. Refer to the CMS Section 3.6.16 for more information on how entropy is selected.

3.5.11.10 NACK pds.payload

The NACK **pds.payload** field carries NACK specific information. The contents are dependent on the **pds.nack_code** field.

Table 3-56 - NACK Payload Contents

NACK_CODE	NACK Payload
UET_NEW_START_PSN	**pds.payload.start_psn** This carries the new *Start_PSN* as described in Section 3.5.8.2.1.
All other NACK_CODEs on ROD PDC	**pds.payload.expected_psn** The next *Expected_PSN* on this ROD PDC
All others	0x0
Note: • When the NACK **pds.spdcid**=0, the **pds.payload** is set to 0x0	

3.5.11.11 pds.cc_type and pds.ccx_type

The **pds.cc_type** and **pds.ccx_type** fields are used to determine the contents of the **pds.ack_cc_state** and **pds.ack_ccx_state** fields in the ACK_CC and ACK_CCX packets, respectively. The two defined formats are shown in Table 3-57 and Table 3-58.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Table 3-57 - CC_TYPE Format

ACK Type	CC_TYPE	MNEMONIC	Usage
ACK_CC	0	CC_NSCC	NSCC
ACK_CC	1	CC_CREDIT	TFC, RCCC
ACK_CC	2—13	-	Reserved for future use
ACK_CC	14—15	-	Reserved for proprietary extensions

Table 3-58 - CCX_TYPE Format

ACK Type	CCX_TYPE	MNEMONIC	Usage
ACK_CCX	0	NSCC_CSIG	NSCC with CSIG
ACK_CCX	1	RCCC_CSIG	RCCC with CSIG
ACK_CCX	2—13	-	Reserved for future use
ACK_CCX	14—15	-	Reserved for proprietary extensions

Refer to the CMS Section 3.6.9 for details on the defined formats.

3.5.11.12 NCCX_TYPE

The NCCX_TYPE field is used to define the contents of the **pds.nack_ccx_state** field in the NACK_CCX packet.

Table 3-59 - NCCX_TYPE Contents

NACK Type	NCCX_TYPE	MNEMONIC	Usage
NACK_CCX	0	CSIG	CSIG Reflection Header (common format for both NSCC and RCCC)
NACK_CCX	0—13	-	Reserved for future use
NACK_CCX	14—15	-	Reserved for proprietary extensions

3.5.11.13 pds.mpr (Maximum PSN Range)

The **pds.mpr** field specifies the available PSN tracking state at a destination given its currently available resources. It is set to a default value of *Default_MPR* when a PDC is created. The maximum PSN range supported is 255, which specifies the maximum *MP_Range* of 32640 (see Table 3-60). The destination may update this value via the ACK_CC **pds.mpr** field based on locally available resources.

The **pds.mpr** field specifies the *MP_Range* by using the following formula:

```
if (pds.mpr == 0)
    MP_RANGE = Override_MPR_Zero_Value
else
    MP_RANGE = pds.mpr *128 packets
```


Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Table 3-60 - PSN Tracking Resources per PSN Range

pds.mpr	Maximum PSN tracking resources (*MP_Range* in packets)
0	if (*Override_MPR_Zero_Value* == 0) Ignore *MP_Range*; else *MP_Range* = *Override_MPR_Zero_Value*
1	128
2	256
3	384
…	
255	32640

When *Override_MPR_Zero_value* is zero and **pds.mpr** is zero, the *MP_range* is ignored. When *Override_MPR_Zero_value* is non-zero and **pds.mpr** is zero, the *MP_Range* is set to *Override_MPR_Zero_value*. When **pds.mpr** is non-zero, the source MUST accept the received **pds.mpr,** multiply the received value by 128 and update the local *MP_Range*. This limits the range of outstanding PSNs such that packets with a PSN greater than *CACK_PSN* + *MP_Range* are not transmitted. If the current outstanding PSN range exceeds this, the PDC does not transmit packets outside of the new range. For example, if *MP_Range* = 1024 and a new **pds.mpr** = 512 arrives:

- If *CACK_PSN* = 1000 and highest *PSN* sent = 1600, when *MP_Range* gets changed to 512, the PDC stops sending new PSNs until *CACK_PSN* reaches 1089; at this point *PSN* = 1601 can be sent.
- If a packet with *PSN* greater than 1512 (*CACK_PSN* + 512) is NACK'd or times out, the retransmission should wait until the *PSN* is within *CACK_PSN + MP_Range.*

The source MUST NOT accept a **pds.mpr** from a packet with a *CACK_PSN* that is lower than or equal to the *CACK_PSN* of the packet that last updated the MPR, to account for ACK packets arriving out of order. When *Override_MPR_Zero_Value* is non-zero, all encodings of **pdc.mpr** are valid. If *Override_MPR_Zero_Value* is zero, a destination that sets **pdc.mpr** on a PDC to a non-zero value MUST set **pds.mpr** to a non-zero value on that PDC for all following packets. There are no restrictions on how often MPR is updated; however, an implementation should constrain the rate of change.

3.5.11.14 pds.sack_bitmap

The ACK Extension header contains a 64-bit selective ACK bitmap field (**pds.sack_bitmap**). Each bit in this bitmap indicates reception state about the packet with the corresponding PSN. When a bit is cleared (i.e., zero), it MUST NOT be interpreted that the corresponding PSN has not been received at the destination.

For example, if one **pds.sack_bitmap** has the bit for PSN 729 set to one (indicating PSN 729 has been received and an SES response has been generated) and the next **pds.sack_bitmap** sets the bit for PSN 729 to zero, the resulting state indicates that PSN 729 was received. In this case the newly received SACK bitmap uses an 'or' function to update the existing PSN bitmap state.

A received **pds.sack_bitmap** of all zeros is valid and indicates no update in received PSN state. Refer to Section 3.5.12.3.1 for more information on the SACK bitmap.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

3.5.11.15 pds.ack_cc_state and pds.ack_ccx_state

The PDS ACK_CC and ACK_CCX packets carry congestion control state in the form of the **pds.ack_cc_state** and **pds.ack_ccx_state** fields respectively. The format of the **pds.ack_cc_state** or **pds.ack_ccx_state** fields is dependent on **pds.cc_type** or **pds.ccx_type** field, respectively. The formats are described in the CMS Section 3.6.9.2.

3.5.11.16 pds.nack_ccx_state

The PDS NACK_CCX packet carries congestion control state in the **pds.nack_ccx_state** field. The **pds.nack_ccx_state** format is dependent on the **pds.nccx_type** field. This field is currently undefined and left for future use.

3.5.12 Requests and Acknowledgements

This section documents various forms of the Request, ACK, and NACK packets.

3.5.12.1 PDS Requests

PDS requests carry SES requests and SES responses with data (in the return direction). PDS requests use PSNs or PKT_IDs assigned by the source to enable detection of packet loss and selective packet retransmission. PDS requests are sent on both the forward and return directions.

PDS requests in the return direction are limited to read and atomic response data in response to SES reads and atomics on RUD and ROD. The return direction uses an independent sequence number space from the forward direction. Excluding RODL, a FEP MUST NOT use the return direction of a PDC to support SES requests such as send, write, or read, and an SES Request carrying an RTR MUST use the forward direction of another, independent PDC.

SES read responses are handled in one of two ways based on the configuration parameter *Max_ACK_Data_Size*. *Max_ACK_Data_Size* refers to the size of the return data and does not include the size of the SES Response header. This is configured to be the same value in each FEP across a fabric domain.

- Read responses with data size ≤ *Max_ACK_Data_Size*
  - Read return data MUST be included in the ACK
- Read responses with data size > *Max_ACK_Data_Size*
  - Read return data MUST be included in a PDS Request
  - Read return data MUST use the target-to-initiator PSN space on the return direction

When *Max_ACK_Data_Size* is set to MTU or higher, all return direction read data is included in ACKs. This configuration may be used for lossless networks or for applications that have few large read transactions. Reads are limited to 1 MTU regardless of *Max_ACK_Data_Size*.

When supporting RODL PDCs, the *Max_ACK_Data_Size* field is not used. All responses, including read and atomic response data, are carried in ACK packets. RODL is designed to operate over a lossless fabric so requests and responses MUST be mapped to different traffic classes.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

For best-effort networks, the recommended value of *Max_ACK_Data_Size* is 16 B. This allows atomics and small software messages to be carried in ACKs, while preventing ACKs—which are not CC-controlled—from interfering with the ability of CMS to properly manage congestion. The associated traffic class mapping is specified in Section 3.5.9.

3.5.12.2 PDS Acknowledgements

Acknowledgements (ACKs) are used to indicate the successful receipt of a packet by the destination PDS, to carry SES responses, and to carry congestion control and load balancing state. ACKs are transmitted only in response to a received packet. This can be a delayed response while PDS waits for SES processing before generating an ACK. ACKs MUST set **pds.ack_psn** to the **pds.psn** of the request packet or the CP that triggered the generation of the ACK unless the ACK is triggered by a Probe CP. Refer to Section 3.5.16.4.2 for information on building ACKs for Probe CPs.

ACK packets triggered by arriving control packets are described in 3.5.16. This type of ACK uses the same base ACK with a CP-type-specific payload.

Load balancing and congestion control algorithms expect quick ACK responses. A potential issue is that SES processing may take a significant amount of time compared to an RTT. Therefore, SES supports the opcode UET_NO_RESPONSE which allows SES to respond to PDS quickly when processing multi-packet messages. The 'no response' concept, described in the SES Section 3.4.4.3, is based on the following:

- Most packets are processed successfully, and the initiator needs to know only 'success' for these.
- If an error occurs on a multi-packet message, it is necessary to report that error to the source on only one of the packets. That is, sending the error response for every packet is not strictly required.
- Clear functionality is used to guarantee an error response is received.

When SES gets a packet that is part of a multi-packet message and that packet is not the last packet in the message to arrive, and the packet is not marked for delivery complete (**ses.dc**), then SES can immediately generate a UET_NO_RESPONSE, allowing PDS to transmit an ACK sooner, before SES completes its processing. Note, if SES later determines there is an error related to the packet or message, that stateful SES response is carried in the ACK for the packets in the message arriving after the error is detected.

The general structure of an ACK packet is illustrated in Figure 3-79. The ACK type used is determined by the congestion mechanisms in use. Typically, this should be consistent across the set of FEPs in a fabric. An implementation may provide a configuration option to enable/disable each type, e.g., ACK_TYPE_ENB.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

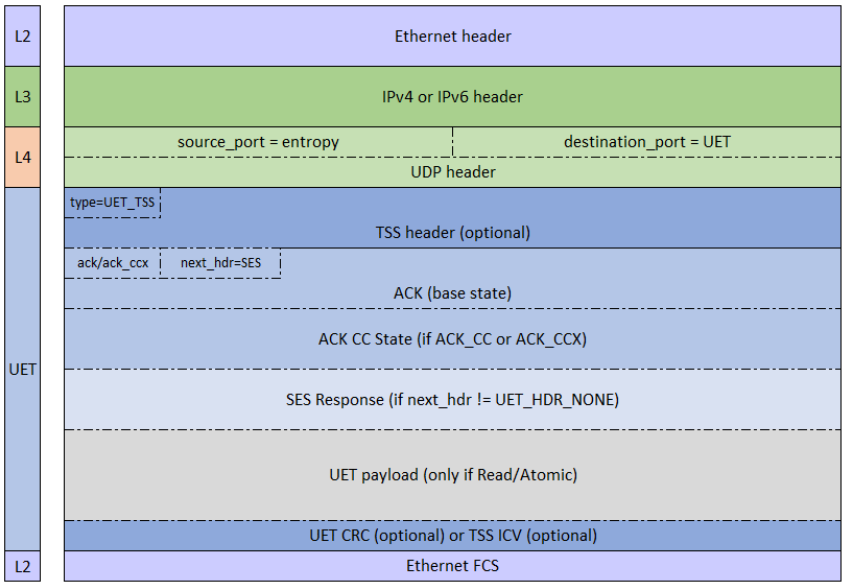


Figure 3-79 - Example ACK Packet

The ACK, ACK_CC, and ACK_CCX contents can be split into three sections, as illustrated in Figure 3-80. The base state is the same for all ACK types and SHOULD be processed when a valid ACK, ACK_CC, or ACK_CCX is received. The **pds.mpr** field SHOULD be supported. On RUD PDCs, the **pds.sack_bitmap** field MUST be processed when *Enb_ACK_Per_Pkt* is not set and SHOULD be processed when *Enb_ACK_Per_Pkt* is set. On ROD PDCs, use of the **pds.sack_bitmap** field is optional except for Probe CPs. The **pds.sack_bitmap** field MUST be populated in the ACKs generated in response to a Probe CP.

The **pds.ack_cc_state** or **pds.ack_ccx_state** fields SHOULD be processed if the corresponding **pds.cc_type** or **pds.ccx_type** field is supported. Setting **pds.cc_type** to 0 is required for NSCC. An implementation may provide a configuration option to enable/disable each type (e.g., CC_TYPE_ENB and CCX_TYPE_ENB).

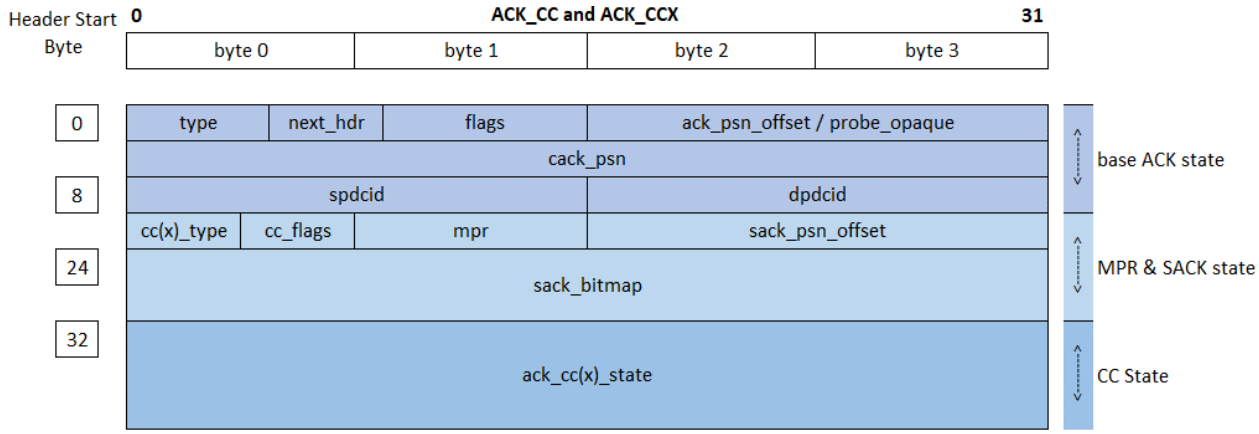


Figure 3-80 - ACK State Sections

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

For diagnostic purposes, the arrival of an ACK_CC or ACK_CCX packet should be reported based on **pds.cc_type** or **pds.ccx_type**, respectively. This state, provided as CC_TYPE_EVENT and CCX_TYPE_EVENT, includes a 16-bit field where a set bit indicates a packet with that **pds.cc_type** or **pds.ccx_type** arrived. For example, if **pds.cc_type** = 0 then bit 0 in the CC_TYPE_EVENT field is set.

PDS ACKs MUST use the entropy from the packet that triggered the ACK generation. If UET is running over UDP, the **udp.src_port** field of the packet that triggered the ACK is used as the **udp.src_port** for the ACK. If UET is running directly over IP, the 16-bit **pds.entropy** field in the UET entropy header from the packet that triggered the ACK is copied to the **pds.entropy** field for an ACK of the associated PSN.

ACKs may be transmitted for every packet or using coalescing based on the configuration parameter, *Enb_ACK_Per_Pkt*. The following two sections describe each of these two modes. ACK per packet MUST be supported.

Destinations receiving PDS requests with **pds.flags.ar** set MUST transmit an ACK regardless of the configuration of *Enb_ACK_Per_Pkt*. Destinations may transmit an ACK when **pds.flags.ar** is cleared.

The **pds.next_hdr** field MUST be set to UET_HDR_NONE when acknowledging CPs.

CACK and ACK processing is the same regardless of whether ACK per packet or ACK coalescing is in use.

If an ACK arrives with a PSN that is out of range, e.g., lower than *CACK_PSN* or higher than the highest PSN sent, the ACK MUST NOT be used to update PDC state.

If an ACK Request CP or duplicate packet is received while PDS is still processing a packet with the same PSN as the ACK Request CP or duplicate packet, the packet is NACK'd with **pds.nack_code** = UET_RCVD_SES_PROCG.

Informative Text:
When running with RCCC or TFC congestion control, the base ACK may be used rather than ACK_CC or ACK_CCX. This requires the use of credit CPs and ACK per packet.

3.5.12.3 ACK per Packet

When *Enb_ACK_Per_Pkt* is set, the **pds.flags.ar** MUST be set in every PDS request.

When an ACK is lost, the corresponding SES Response MUST be inferred to be the UET_DEFAULT_RESPONSE if *CACK_PSN* is higher than the PSN that triggered the missing ACK.

3.5.12.3.1 SACK bitmap—ACK per Packet

The SACK bitmap is transmitted in the **pds.sack_bitmap** field and MUST be supported as follows. The **pds.sack_bitmap** field is 64 bits in length. When a bit is set, it indicates the corresponding PSN was received, passed to SES for processing, and SES provided a response. When a bit is not set (zero), there is no information about the PSN. The purpose of the **pds.sack_bitmap** field is to provide a redundant indication that packets have been received. A bit in the **pds.sack_bitmap** field MUST NOT be set until the corresponding PSN is received at the destination.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

If an ACK is lost, the SACK bitmap and *CACK_PSN* allow the source to determine if the lost ACK was a default response or guaranteed delivery response, as described in Section 3.5.12.4.4. The **pds.sack_bitmap** field alone cannot determine default responses.

Because an ACK is transmitted for every packet in the ACK per packet mode, the **pds.sack_bitmap** is anchored using the PSN that triggered the ACK. The **pds.sack_psn_offset** field is then calculated relative to *CACK_PSN*. See an example in Figure 3-81.

3.5.12.4 Coalesced ACKs

Support for coalesced ACKs is optional. If supported and *Enb_ACK_Per_Pkt* is not set, the source MUST set the **pds.flags.ar** field based on the criteria list in Table 3-61.

Table 3-61 - Triggers for Setting pds.flags.ar in PDS Requests when Using Coalesced ACKs

Trigger Event	Definition
CMS control	CMS indicates AR is to be set on a packet
Retransmission	If the packet is a retransmission
Last packet	If the packet is the last packet on the PDC such that there is no additional pending work on the PDC

When using coalesced ACKs on RUD and ROD PDCs, ACKs MUST be generated based on the criteria list in Table 3-62, and the ACK_CC or ACK_CCX type MUST be used. Coalesced ACKs may acknowledge multiple packets. The *ACK_PSN* indicates which PSN triggered the ACK. The SES Response carried in the packet MUST be the SES Response for the *ACK_PSN*. All other PSNs acknowledged by the ACK are determined to have a UET_DEFAULT_RESPONSE indicating success.

When using coalesced ACKs on RODL, ACKs MUST be generated according to the criteria in Table 3-62, excluding packets marked with ECN when using RODL over the UFH_16 header (which does not include the **pds.flags.m** flag) and excluding the “last packet before idle” condition (which applies only to RUD out-of-order arrival). The packet/byte-threshold trigger may be simplified to a packet-count threshold. RODL supports sending reverse ACK state, which can be used to replace the generation of an ACK packet as described in section 3.5.12.4.4.

PDS infers that a UET_DEFAULT_RESPONSE has occurred when *CACK_PSN* is higher than the PSN of a packet waiting for an SES response. The **pds.sack_bitmap** field may indicate a packet was received at the destination. The **pds.sack_bitmap** MUST NOT be used to determine a UET_DEFAULT_RESPONSE.

Informative Text:
Inferred responses indicate success when all packets in a message have SES responses. PDS must not pass SES an inferred response for a packet until *CACK_PSN* is higher than the packet’s PSN. Any PSN higher than CACK may have an associated guaranteed response, and that guaranteed response must be evaluated by SES before the message is considered completed.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Table 3-62 - Triggers for Generating an ACK

Trigger Event	Description
First packet is received that creates/allocates a new PDC	New PDC so let the initiator know and provide CC state as soon as possible.
ECN marked packet is received	Send EV congestion state to load balancing function as quickly as possible; see the **pds.flags.m** flag in Section 3.5.11.8.5. This trigger is enabled when *ACK_On_ECN* is set and disabled when *ACK_On_ECN* is cleared. See Section 3.5.11.8.5.
Byte/packet threshold is reached (*ACK_Gen_Trigger*)	Cumulative trigger indicating a number of packets/bytes have arrived so an ACK will be transmitted, see Section 3.5.12.4.1.
AR flag is set in the arriving request packet	The **pds.flags.ar** flag is set based on CC control. Certain CPs set this flag—see Section 3.5.16.9.
SES generates a guaranteed delivery response	Send the SES Response.
Last packet before the source goes idle	The packet may be the last packet sent for a while; see Section 3.5.12.4.2.

3.5.12.4.1 Packet/Byte Threshold ACK Trigger—Coalesced ACKs

When using coalesced ACKs, this packet/byte threshold is used to trigger an ACK every several packets. A counter is maintained such that when a configured number of packets or bytes is received, an ACK is transmitted. The following configuration parameters and local variables are used:

Table 3-63 - ACK Configuration Parameters

Name	Type	Definition
ACK_GEN_COUNT	Local variable (counter)	Counter that is incremented when packets arrive and cleared when an ACK is generated
ACK_Gen_Min_Pkt_Add	Configuration parameter	Defines the smallest value added to ACK_GEN_COUNT when a packet is received
ACK_Gen_Trigger	Configuration parameter	Defines the threshold at which ACK_GEN_COUNT triggers an ACK generation

It operates as follows:

```
if pkt arrives {
    length = uet_payload_length(pkt)
    if length < ACK_GEN_MIN_PKT_ADD
        length = ACK_GEN_MIN_PKT_ADD
    ACK_GEN_COUNT += length
}

if (ACK_GEN_COUNT >= ACK_Gen_Trigger) {
    Transmit ACK
}

if (ACK is transmitted) {
    ACK_GEN_COUNT = 0
}
```


Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Informative Text:
ACK_Gen_Min_Pkt_Add defaults to 1 KB and *ACK_Gen_Trigger* defaults to 16 KB. This will trigger an ACK at least every 16 KB or 16 packets. If 4 KB packets are used, an ACK is transmitted every 4 packets (16 KB/4 KB = 4).
This is not required to be particularly accurate. The UET payload may be truncated to a convenient precision to minimize processing requirements.

3.5.12.4.2 Last Packet ACK Trigger—Coalesced ACKs

When coalesced ACKs are used, it is important to trigger an acknowledgment of the last packet received once the PDC goes idle. Therefore, the last packet in the message that leaves the sending PDC empty has the **pds.flags.ar** field set to request an acknowledgement of the packet.

Because there are out-of-order arrivals, packets may arrive after the packet with **pds.flags.ar** set. This may lead to spurious retransmissions, as there may be no trigger to transmit an ACK. Generation of an ACK for this scenario is implementation specific. An example method is provided in Section 3.5.12.4.2.1.

3.5.12.4.2.1 Example Method for Last Packet ACK Trigger

The variables in Table 3-64 may be used to identify this scenario and transmit an ACK.

Table 3-64 - Local Variables for Last Packet ACK Trigger - Coalesced ACKs

Local Variables (per PDC)	Definition
Prev_AR_PSN	Highest PSN that arrived with **pds.flags.ar** set
Max_Rcvd_PSN	Highest PSN that arrived (also used in SACK bitmap processing)

Last Packet ACK Trigger Method:

```
if (pds.flags.ar is set) {
    Prev_AR_PSN = max(Prev_AR_PSN, rcvd_pkt.pds.psn)
}
if (Prev_AR_PSN == Max_Rcvd_PSN) {
    Transmit an ACK
}
```

Alternatively, an implementation may generate an ACK whenever a packet arrives and updates the received PSN state such that there are no missing packets. Another option is to use a GEN_ACK_TIMER timer. This timer is reset every time an ACK is transmitted and triggers ACK generation if it expires.

3.5.12.4.3 SACK Bitmap—Coalesced ACKs

The SACK bitmap is transmitted in the **pds.sack_bitmap** field and MUST be supported in all profiles. The **pds.sack_bitmap** field is 64 bits in length. When a bit is set, it indicates the PSN was received and passed to SES for processing. When a bit is not set (zero), there is no information about the PSN. Zero does not indicate the packet was not received. The intent of the **pds.sack_bitmap** field is to efficiently indicate which packets have been received with redundancy. If an ACK is lost, the SACK bitmap and *CACK_PSN* will allow the source to infer if the lost ACK was a default response or guaranteed delivery response.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

On ROD PDCs, the **pds.sack_bitmap** field may be set to all zeros as *ACK_PSN* and *CACK_PSN* provide the required information.

On RUD PDCs, multiple PSNs may arrive between transmissions of an ACK such that it is not possible to represent all of the received PSNs in a single **pds.sack_bitmap**. The goal is to efficiently represent as many PSNs as possible while biasing toward sending acknowledgements for lower PSNs, to allow older resources to be freed. The following pseudo-code provides an example of the state maintained to assign the SACK bitmap to cover the newly received low PSN while iterating over the *Expected_PSN* range. For clarity, the case of the PSN wrapping at 0xFFFFFFFF is not described.

Table 3-65 - Local Variables for SACK Bitmap - Coalesced ACKs

Local Variables	Definition
Max_Rcv_PSN	Highest PSN to arrive on PDC (wraps at 0xFFFFFFFF)
CACK_PSN	Cumulative ACK PSN: All PSN lower than this PSN are received, processed, and cleared
SACK_Base	Defines PSN to be used as the lowest bit in SACK bitmap when generating an ACK.
SACK_Base_Track	Tracks the *SACK_Base* value used when generating an ACK in response to PDS requests. It is updated based on the PSNs received between ACKs.

Processing upon PDS Request or ACK Request CP acceptance

```
// If lowest PSN not yet received & cleared is advanced, advance CACK_PSN
update CACK_PSN

if (SACK_Base_Track < CACK_PSN) {
        // CACK_PSN will ACK all packets up to CACK_PSN, update SACK_BASE
        SACK_Base_Track = CACK_PSN} else {
    If (CACK_PSN < rcvd_pkt.pds.psn) && (rcvd_pkt.pds.psn < SACK_Base_Track)
{
        // Lower PSN arrives, favor ACK'ing these so update SACK_BASE
        SACK_Base_Track = rcvd_pkt.pds.psn
    }
}
```

Processing for ACK transmission

```
if (ACK is triggered by Probe CP) {
    SACK_Base = max{probe.payload, CACK_PSN}
    generate_ack()
}

if (ACK is triggered by PDS Request or ACK Request CP) {
    SACK_Base = SACK_Base_Track
    generate_ack()
    // shift the tracker up 64 to iterate over PSNs if no low PSNs arrive,
    // unless it would go past Max_Rcv_PSN
    if ((SACK_Base_Track + 63) < Max_Rcv_PSN) {
        SACK_Base_Track = SACK_Base_Track + 64
    }
}
```


Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

In general, this logic is intended to prioritize acknowledging lower PSNs when they arrive. If a lower PSN is received, update *SACK_Base* to that PSN. If a higher PSN arrives, rather than jumping to it directly, constrain *SACK_Base* so it increases by 64 — so that the SACK bitmap gradually covers all PSNs from *CACK_PSN* up to *Max_Rcv_PSN*.

SACK_Base represents the PSN of the least significant bit in the SACK bitmap. The bitmap is always started from a PSN that has the three least significant bits equal to zero. This is done to reduce the shift logic needed to generate the **pds.sack_bitmap** field.

The **pds.sack_psn_offset** field is calculated as:

```
pds.sack_psn_offset = SACK_Base - CACK_PSN
```

Figure 3-81 provides an example SACK bitmap:

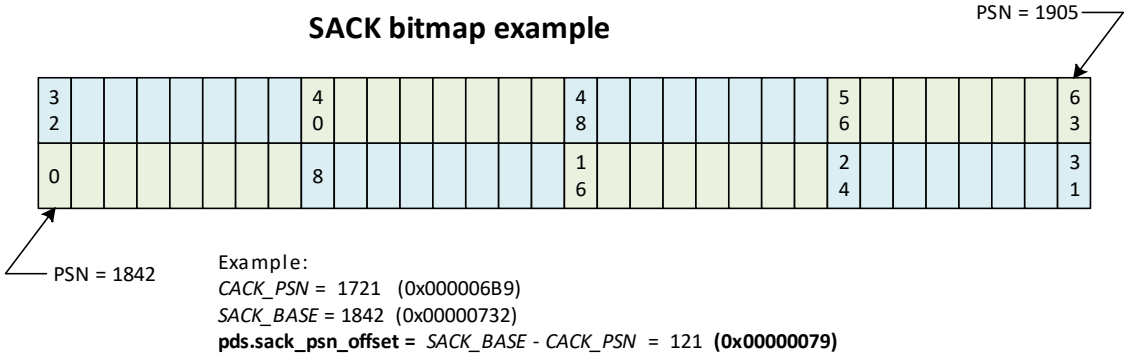


Figure 3-81 - SACK Bitmap

3.5.12.4.4 RODL Reverse ACK State

RODL uses bidirectional PDCs, which allow the acknowledgment state to be carried in reverse direction PDS requests. When using RODL over the UFH_16 header, each PDS Request includes the lower 12 bits of the cumulative acknowledgment PSN (*CACK_PSN*) for the opposite direction in the **pds.rev_cack_psn_lsb** field. Otherwise, RODL carries the **pds.rev_cack_psn_lsb** as a 16-bit field.

RODL (excluding over the UFH_16 header) alternates sending the **pds.clear_psn_offset** and **pds.rev_cack_psn_lsb**. If one of these has not advanced but the other has, the same field may be sent multiple times.

When write and non-fetching atomic operations are being performed without error on a bidirectional PDC, all SES responses are of type UET_DEFAULT_RESPONSE and carry RC_OK return codes. These responses can be messaged to the source with **pds.rev_cack_psn_lsb**, removing the need for ACK packets while traffic is active in both directions.

Stateful responses MUST be transported in an ACK packet.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Both initiator and target RODL FEPs MUST maintain a full 32-bit CACK_PSN, as in ROD. The **rev_cack_psn_lsb** field carries the 12 or 16 lowest bits of the locally maintained CACK_PSN. The PDS receiving a RODL packet is required to monitor when the CACK_PSN field wraps and update the local CACK_PSN accordingly by incrementing the upper 20 bits.

Similarly, both initiator and target MUST maintain full 32-bit PSN and CLEAR_PSN.

If there is no pending SES request or response and a PDS needs to update either CLEAR_PSN or CACK_PSN, an implementation MUST send an ACK or control packet to exchange this state. The rules used to trigger generating ACK packets when using coalescing on a RODL PDC are described in Section 3.5.12.4.

3.5.12.5 Cumulative ACK (CACK) Rules

CACK_PSN is defined such that:

1. All earlier PSNs up to and including *CACK_PSN* have been successfully received.
2. All corresponding guaranteed delivery SES Response results have been sent and cleared by the peer PDC.

When a guaranteed delivery SES Response is transmitted, the *CACK_PSN* MUST NOT advance up to or past the PSN associated with the guaranteed delivery SES Response until that response is cleared. To enable efficient PDS implementations, specifically in the per-PDC state, two optimizations are used:

- Default SES responses are defined, allowing the destination PSN to not store/hold the common SES Response of RC_OK.
- The SACK bitmap acknowledges that a PSN has arrived successfully at the destination (i.e., the packet was not trimmed and was accepted, passed to the SES, and SES generated a response), allowing the source PDC to discard state required to retransmit a packet (except in the case of a read request of size $\leq$ *Max_ACK_Data_Size*—see Section 3.5.16.2).

CACK_PSN plays a role in enabling these optimizations. When an ACK is received, *CACK_PSN* is used to determine which PSNs have been delivered and successfully processed. The processing order is important.

- If the **pds.next_hdr** field is not set to UET_HDR_NONE, the ACK is carrying an SES Response for *PSN* = *ACK_PSN*. This is processed by passing the SES Response to the local SES and marking the PSN as done.
- When processing acknowledgment state in an ACK packet, the *ACK_PSN* MUST be processed first, as it may carry a non-default response. Then, for each PSN indicated in the SACK bitmap that is less than or equal to *CACK_PSN* and not yet marked as done, a UET_DEFAULT_RESPONSE is sent to SES and the PSN marked as done.
- Done indicates that all required processing for that PSN is complete

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Informative Text:
PSN state is implementation specific. The following example intends to clarify PSN tracking state and is not optimized.

Example PSN tracking states at the source: {pkt_sent, pkt_rtx, pkt_rcvd_at_dest, done}

- pkt_sent: packet was sent at least once
- pkt_rtx: the packet is pending retransmission
- pkt_rcvd_at_dest: destination received the packet (i.e. bit set in SACK bitmap or based on CACK), source can delete retransmit state
- done: SES response provided to local SES

Example PSN tracking states at the destination: {not_rcvd, rcvd_at_ses, gtd_del, done}

- not_rcvd: packet not received yet
- rcvd_at_ses: packet received waiting on SES
- gtd_del: SES processing done and ACK needs guaranteed delivery (SACK bitmap bit is set)
- done: done with this PSN, if gtd_del then cleared (SACK bitmap bit is set)

Refer to example PSN states in Section 3.5.13. This is not an exhaustive list of all state maintained per PSN.

3.5.12.6 ACK Window

The destination MUST NOT transmit an ACK for any packet with a PSN outside the maximum PSN window of (*CACK_PSN – MP_RANGE* to *CACK_PSN + MP_RANGE*) with the exception of packets with PSNs between *CLEAR_PSN* and *CACK_PSN*. It is possible that *CLEAR_PSN* is lower than *CACK_PSN – MP_RANGE* during the transition to a smaller MPR. Any packet arriving with **pds.psn** less than or equal to *CLEAR_PSN* is not acknowledged. This constrains the range over which responses to duplicate PSNs are generated as a security feature.

MP_RANGE defines the maximum range of PSNs that can be outstanding on a particular PDC based on resource limitations at the destination, as described in Section 3.5.11.13.

The source MUST NOT transmit packets with a **pds.psn** greater than *CACK_PSN + MP_RANGE*. When the destination reduces *MP_RANGE* via the ACK_CC **pds.ack_cc_state** field, the source may temporarily exceed *CACK_PSN + MP_RANGE* until the *CACK_PSN* advances. No new packets are transmitted on that PDC during that time (retransmissions are sent).

At the destination, acknowledgements for packets older than *CACK_PSN – MP_RANGE* are not transmitted except for PSNs between *CLEAR_PSN* and *CACK_PSN*. The PDS_MPR concept is illustrated in Figure 3-82 and captured in example code below. When a packet is received outside of the *Expected_PSN* range, the OUT_OF_WINDOW_PSN event counter SHOULD be incremented. This indicates a potential security attack.

```
if ((PSN ≤ CLEAR_PSN) || (PSN > CACK_PSN + MP_RANGE))
  // drop the packet with no ACK and increment OUT_OF_WINDOW_PSN counter
```


Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

```
else if PSN was already received
  // transmit ACK – either guaranteed or default SES Response
else pass to SES for processing
```

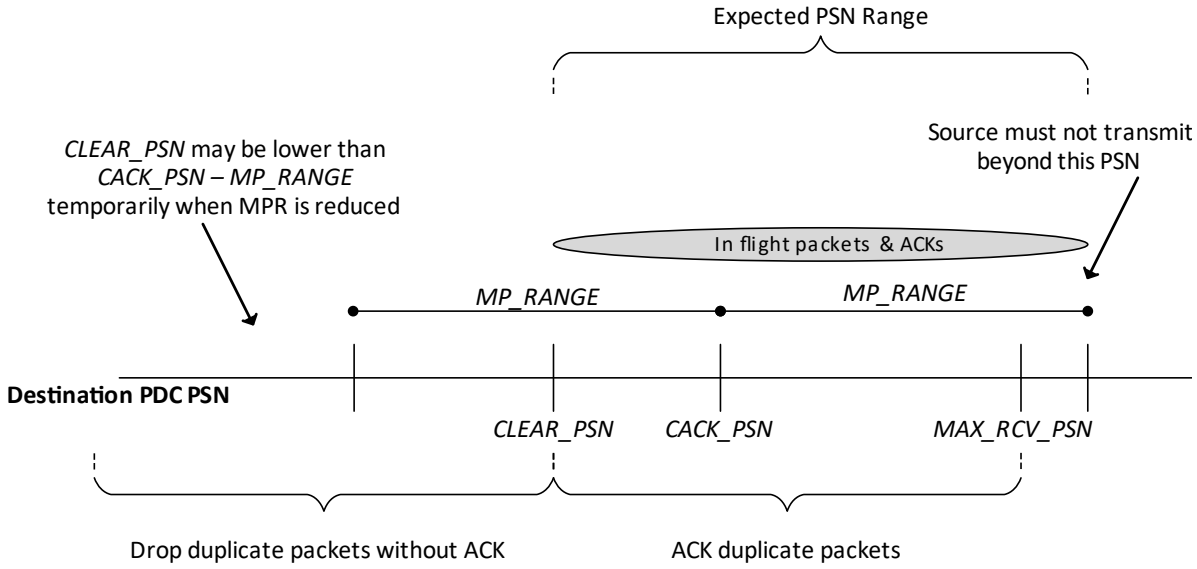


Figure 3-82 - PDS_MPR Concept

3.5.12.7 PDS Negative Acknowledgements

NACK packets are used to provide explicit indications that a packet was dropped. The format is illustrated in Figure 3-83. NACKs MUST be generated according to the events and requirements described in Table 3-66. If a PDS Request with the **pds.flags.retx** flag set is received and a NACK is transmitted in response, the NACK MUST set the **pds.flags.retx** flag. NACKs MUST use the EV from the packet that triggered the NACK.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

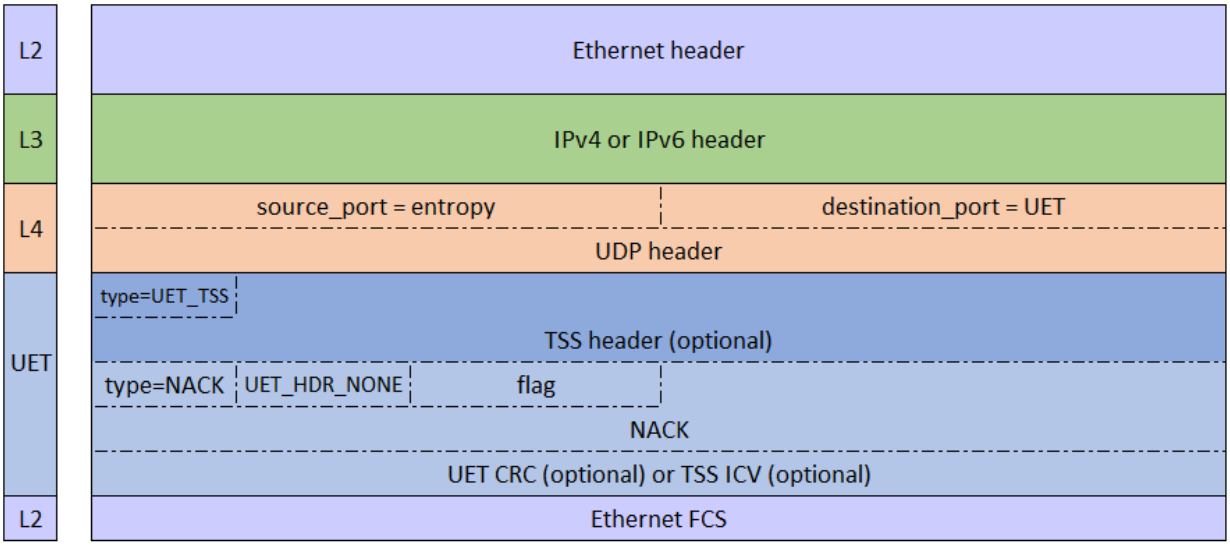


Figure 3-83 - PDS NACK Format

Table 3-66 - Triggers for Generating a NACK

Event	Requirement	Description
Local packet drop	MUST	Valid packet is received and is dropped due to local resource limitations (e.g., lack of receive packet buffer)
Trimmed packet is received	MUST	Trimmed packet is received
ACK Request CP is received	MUST	CP is received and the requested PSN has not been received
Invalid UET header or unexpected event	MUST	If a UET header check or PSN state usage or other unexpected event results in an error; see list in Table 3-68
Packet is received outside *Expected_PSN* window	MUST NOT	If a PSN is received outside the valid ACK window described in Section 3.5.12.6, the packet MUST be dropped and the event recorded
Invalid packet is received	MUST NOT	If any network header check fails (outside of UET headers), the packet MUST NOT result in a NACK (e.g., invalid IP header checksum)

Informative Text:
An implementation may support a limited NACK packet generation rate, in which case it may not be possible to transmit NACKs at the same rate as received events shown in Table 3-66. In this scenario, the packet may be dropped without generating a NACK.

When a valid NACK or NACK_CCX is received, the base NACK state (as illustrated in Figure 3-84) is always processed. The **pds.nack_ccx_state** field SHOULD be processed if the NCCX_TYPE is supported.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

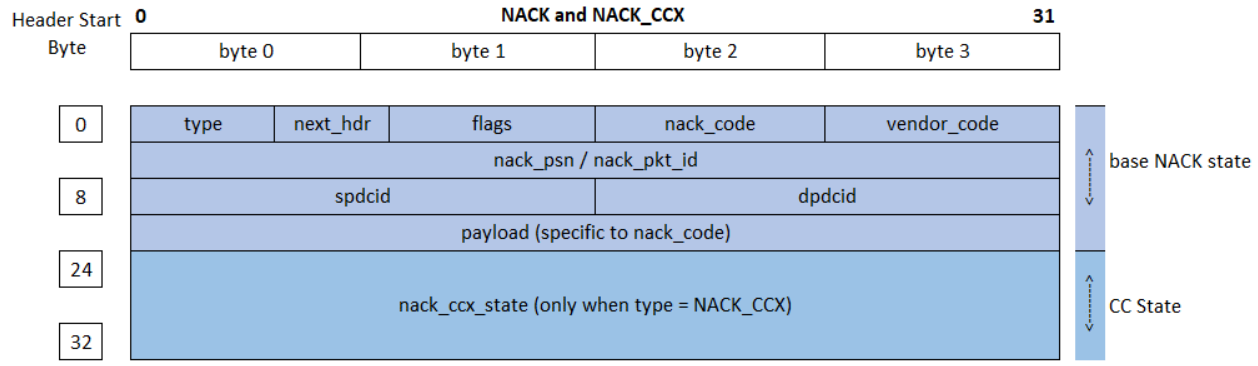


Figure 3-84 - NACK Sections

For diagnostic purposes, the arrival of a NACK_CCX packet should be reported based on **pds.nccx_type**. This state, provided in NCC_TYPE_EVENT, is a 16-bit field where a set bit indicates the arrival of a packet with that **pds.nccx_type**. For example, if **pds.nccx_type** = 0 then bit 0 in the NCC_TYPE_EVENT field is set.

PDS NACKs carry negative acknowledgement NACK codes (**pds.nack_code)**. There are two categories of NACK codes indicating either retransmit events or error events. The NACK codes are enumerated in Table 3-68. The NACK code is carried in the **pds.nack_code** field to indicate the specific type of NACK event, which may be useful for diagnostics, and to enable the source to determine if it should wait an appropriate time before retrying.

Events leading to NACK generation can be split into the following NACK error types:

- **NORMAL**: This indicates normal operation, an expected event. Typically, these NACKs occur when a packet cannot be processed when it is received but should be processed in the future. This results in a packet being retransmitted on the same PDC. When the packet would have created a PDC, but it does not, the retransmit may be done using a different PDC (i.e., different *SPDCID*).
- **PDC_ERR**: This indicates an unexpected event occurred, but the PDC remains active; for example, if a packet is received with **pds.flags.syn** set and the PDC is no longer accepting SYN packets (**pds.nack_code** = UET_INVALID_SYN).
- **PDC_FATAL**: This indicates the error is unrecoverable; for example, the associated PDCID could not be found, or the IP address does not match. The associated PDC MUST be closed. The FEP sends a NACK and closes the PDC. When a FEP receives a NACK with a PDC_FATAL code, it closes the PDC.

The NACK codes in Table 3-68 use the following terms to clarify the expected behavior at the source when a NACK is received:

- **RETX:** This indicates a transient scenario where the packet SHOULD be retransmitted on the same PDC using the same PSN. (e.g., resource exhaustion or restart with new *Start_PSN*).

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

- **RETRY:** This indicates a scenario where the packet is not associated with an established PDC and the semantic message that is carried should be retransmitted using a different PDC and/or PSN.
- **FAIL:** This indicates the packet delivery may be failed to SES.

Some PDC_ERR or PDC_FATAL events are marked as RETRY; an implementation may fail these to SES.

The local variables shown in Table 3-67 are used by the source when handling NACKs and MUST be supported.

Table 3-67 - Configuration Parameters for Handling NACK and RTO

Local Variables	Requirement	Definition
RTO_Init_Time	MUST	Initial time to wait before retransmitting a packet due to timeout at the sender. May be global across all PDCs or unique for RUD, ROD, RODL, and reserved PDCs.
NACK_Retx_Time	MUST	Set of one to four configured times used to delay the retransmission of a packet that was NACK'd. If more than one configured value, select based on NACK code. Should provide an option for no delay.
Max_RTO_Retx_Cnt	MUST	Threshold for number of retransmissions declaring failure for PSN. If *Max_NACK_Retx_Cnt* is provided, this only counts RTO retransmissions. If *Max_NACK_Retx_Cnt* is not provided, this counts RTO and NACK retransmissions
Max_NACK_Retx_Cnt		Threshold for number of retransmissions performed due to receiving NACKs before declaring failure on the PSN.

Packets are retransmitted when an RTO occurs, or a NACK packet is received. The number of times a packet is retransmitted due to RTO is limited to a configured value, *Max_RTO_Retx_Cnt*, before declaring failure. The number of times a packet is retransmitted due to receiving a NACK may be limited to a configured value, *Max_NACK_Retx_Cnt*, before declaring failure. Implementing *Max_NACK_Retx_Cnt* is optional. If omitted, NACK retransmissions are included in the RTO counter.

Retransmission due to RTO MUST be supported as follows:

1. Retransmit any packet that has not been acknowledged or NACK'd in the RTO time period
2. Each time a packet is retransmitted because of an RTO, the RTO time period is exponentially increased as follows:
    a. *RTO_TIMER* = *RTO_INIT_TIME* * 2^{retry_count}
    b. Where retry_count is the number of times a packet has been retransmitted due to RTO
3. If a NACK is received:
    a. Retransmit the packet after waiting *NACK_Retx_Time*, based on **pds.nack_code**
    b. Reset the *RTO_TIMER* and, optionally, set retry_count to zero
4. If a Probe CP is sent, the *RTO_TIMER* may be paused until the ACK for the Probe CP is received.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

For example, if the base time period is 30 µsec and zero retries have previously occurred, then the timer is set to 30 µsec. If this times out, then retry_cnt = 1 and the timer is set to 30* 2^{retry_cnt} = 60 µsec. If this times out again, then retry_cnt = 2 and the timer is set to 120 µsec, and so on.

Refer to Section 3.5.21 for NACK processing specific to ROD.

The processing based on error type is defined in the error model in Section 3.5.24. The events leading to the NACK generation are described in Section 3.5.8.2, unless otherwise indicated in the description. The description 'fail to SES' means the source PDS will indicate to SES that the packet was not successfully delivered. SES will generally complete the associated operation in error.

NACK packets MUST have the **pds.nack_code** field set to a valid value as specified in Table 3-68. Multiple NACK codes are defined to allow tracking the details of what triggered the NACK. For example, an implementation may send **pds.nack_code** = UET_NO_PDC_AVAIL whenever a PDC is not created. Or it may set **pds.nack_code** to UET_NOCCC_AVAIL, UET_NO_BITMAP, etc.

Upon reception of a NACK packet with valid **pds.nack_code**, the source MUST perform the source action specified in Table 3-68.

Table 3-68 - PDS NACK Codes

Name	Value	Error Type	Source Action	Description
Reserved	0x00	N/A	N/A	Reserved
UET_TRIMMED (Section 3.5.15.1)	0x01	NORMAL	Retransmit (RETX)	Packet was trimmed
UET_TRIMMED_LASTHOP (Section 3.5.15.1)	0x02	NORMAL	Retransmit (RETX)	Packet was trimmed at the last hop switch
Reserved	0x03	-	-	-
UET_NO_PDC_AVAIL (Section 3.5.8.2)	0x04	NORMAL	Retransmit (RETRY - no PDC created)	No PDC resource available; set **pds.spdcid** = 0
UET_NO_CCC_AVAIL (Section 3.5.8.2)	0x05	NORMAL	Retransmit (RETRY- no PDC created)	No CCC resource available; set **pds.spdcid** = 0
UET_NO_BITMAP (Section 3.5.8.2)	0x06	NORMAL	Retransmit (RETRY - no PDC created)	No bitmap or other PSN tracking resource available (e.g., if bitmaps are dynamically allocated and none are available); set **pds.spdcid** = 0
UET_NO_PKT_BUFFER (Section 3.5.8.2)	0x07	NORMAL	Retransmit (RETX)	No packet buffer resource available
UET_NO_GTD_DEL_AVAIL (Section 3.5.8.2)	0x08	NORMAL	Retransmit (RETX)	No SES guaranteed delivery response resource available (i.e., can't save guaranteed delivery response if needed so packet was dropped before passing to SES)
UET_NO_SES_MSG_AVAIL (Section 3.5.8.2)	0x09	NORMAL	Retransmit (RETX)	No message tracking state available

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Name	Value	Error Type	Source Action	Description
UET_NO_RESOURCE (Section 3.5.8.2)	0x0A	NORMAL	Retransmit (RETX)	General resource not available – implementation-specific resource not listed in NACKs 0x03—0x08; set **pds.spdcid** = 0 if there is no associated PDC
UET_PSN_OOR_WINDOW (Section 3.5.8.2)	0x0B	NORMAL	Retransmit (RETX if PSN >CACK)	PSN outside tracking window (e.g., beyond end of available bitmap)
reserved	0x0C	-	-	-
UET_ROD_OOO (Section 3.5.21.1)	0x0D	NORMAL	Retransmit (RETX)	A PSN arrived out of order on a ROD or RODL PDC
UET_INV_DPDCID (Section 3.5.8.2)	0x0E	PDC_FATAL	Close PDC (RETRY)	**pds.dpdcid** is not recognized and **pds.flags.syn** not set; set **pds.spdcid** = 0
UET_PDC_HDR_ MISMATCH Section 3.5.8.2)	0x0F	PDC_FATAL	Close PDC (RETRY)	The packet does not have **pds.flags.syn** set but did not match connection state (e.g., **ip.src_addr** or other field doesn't match PDC state) (This is an optional check separate from required check for a valid **pds.dpdcid**); set **pds.spdcid** = 0
UET_CLOSING (Section 3.5.8.3)	0x10	PDC_FATAL (drop packet)	Report error (RETRY)	The target PDCID is in a closed state or is in the process of being closed and a new PDS Request is received that advances the PSN
UET_CLOSING_IN_ERR (Section 3.5.8.3)	0x11	PDC_FATAL	Close PDC (no retransmit)	Timeout at target during close process (e.g., no response to Close Request CP). (*Max_RTO_Retx_Cnt* exceeded—NACK the Close Command CP)
UET_PKT_NOT_RCVD (Section 3.5.16.2)	0x12	PDC_ERR	Retransmit (RETX)	CP arrived with 'ACK Request' but packet with requested PSN was not received (Section 3.5.16)
UET_GTD_RESP_UNAVAIL (Section 3.5.16.2)	0x13	PDC_FATAL	Close PDC (FAIL) (no retransmit)	Duplicate PSN is received, state indicates there is a guaranteed delivery SES Response but that response cannot be found, and this PSN was not cleared (implementation error)
UET_ACK_WITH_DATA (Section 3.5.16.2)	0x14	PDC_ERR	Retransmit (RETX original pkt)	ACK Request CP for PSN with associated read response data
UET_INVALID_SYN (Section 3.5.8.2)	0x15	PDC_FATAL	Close PDC (RETRY)	Packet is received with **pds.flags.syn** set with **pds.psn** outside the expected range of PSNs with **pds.flags.syn**
UET_PDC_MODE_ MISMATCH (Section 3.5.8.2)	0x16	PDC_FATAL	Close PDC (RETRY)	Packet is received and delivery mode does not match (RUD/ROD/RODL)
UET_NEW_START_PSN (Section 3.5.8.2.1)	0x17	NORMAL	Retransmit (RETX)	Resend all packets with new *Start_PSN*
UET_RCVD_SES_PROCG (Section 3.5.13)	0x18	NORMAL	Retransmit (ACK may arrive	This can occur, e.g., if a delayed packet and a retransmission arrive at destination close in time

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Name	Value	Error Type	Source Action	Description
			before RETX)	
UET_UNEXP_EVENT (generic event—implementation specific)	0x19	PDC_FATAL	Close PDC (RETRY)	This is unexpected—processing requires an unsupported feature; use this if an event occurs that is unexpected (e.g., something indicating an implementation error—from which the PDC cannot be recovered)
UET_RCVR_INFER_LOSS (Section 3.5.15)	0x1A	NORMAL	Retransmit (RETX)	Destination infers a PSN was lost and effectively requests retransmission; application specific
UET_NO_SES_SYNC_AVAIL (Section 3.5.8.2)	0x1B	NORMAL	Retransmit (RETX)	No synchronization tracking state available
Reserved for UET	0x1C -0xFC			For future use
UET_EXP_NACK_NORMAL	0xFD	NORMAL	Retransmit (RETX)	Experimental code
UET_EXP_NACK_ERR	0xFE	PDC_ERR	Retransmit (RETRY)	Experimental code
UET_EXP_NACK_FATAL	0xFF	PDC_FATAL	Close PDC (FAIL)	Experimental code

Note:

- For NACK codes indicating RETX, the source action on a RUD PDC is to retransmit the original packet associated with the PSN (**pds.nack_psn**). The source action on a ROD or RODL PDC is to support GoBackN retransmission as described in Section 3.5.21.1. If an ACK arrives for a packet before it is retransmitted, retransmission is not done.

Informative Text:
SES may indicate to PDS that the packet is to be dropped and NACK'd. When this occurs, typically for some resource exhaustion event, PDS does not mark the PSN as received. Similar to a trim event, the source will retransmit the packet with the original PSN.

PDCs SHOULD map each NACK-based retransmit to a timer based on the NACK code, where the timer controls whether to delay the transmission, and, if so, by how much time. For example, a trimmed packet may be retransmitted immediately (within the constraints of the window/credit of the congestion control algorithm). Alternatively, when no PDC resource is available, it may be desirable to delay retransmission for a couple of RTTs.

NACK events, including the NACK code, are reported to CMS.

NACKs MUST NOT be marked as trimmable. Trimmed NACKs are dropped. The event SHOULD be counted for diagnostics. NACKs that arrive with a **pds.nack_psn** field that is out of range MUST NOT be used to update PDC state except for event counters — for example, if the **pds.nack_psn** field is lower than *CACK_PSN* or higher than the highest PSN sent on the PDC.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

3.5.13 Default SES Responses

As described in the previous section, guaranteed delivery SES responses are used to reduce the required state stored by a RUD/ROD/RODL PDC. When SES indicates its response does not require guaranteed delivery, PDS sends the response and then deletes it. If a duplicate packet or an ACK Request CP is received for a PSN that has no stored SES Response state, PDS generates an 'inferred' SES Response.

PDS MUST generate a UET_DEFAULT_RESPONSE when inferring an SES Response. Generating a UET_NO_RESPONSE may be supported. This requires maintaining state to differentiate between that and a successful default response.

Implementation Note:
The *RTO_TIMER* should be started when the packet being transmitted is as close to being on the network as practical. For example, as the packet reaches the Ethernet MAC or when the packet is enqueued in the transmit packet buffer.

Informative Text:
Tracking of PSN state varies by implementation. Two example implementations are shown within this information note.

The ACK state including SACK bitmap should not be updated until SES has provided a response. In implementation #1, the PSN state is not updated until SES returns a response—in this case duplicate packets are filtered using another mechanism until the SES response is provided. This allows distinguishing between UET_NO_RESPONSE and UET_DEFAULT_RESPONSE when a duplicate packet or ACK Request CP is received.

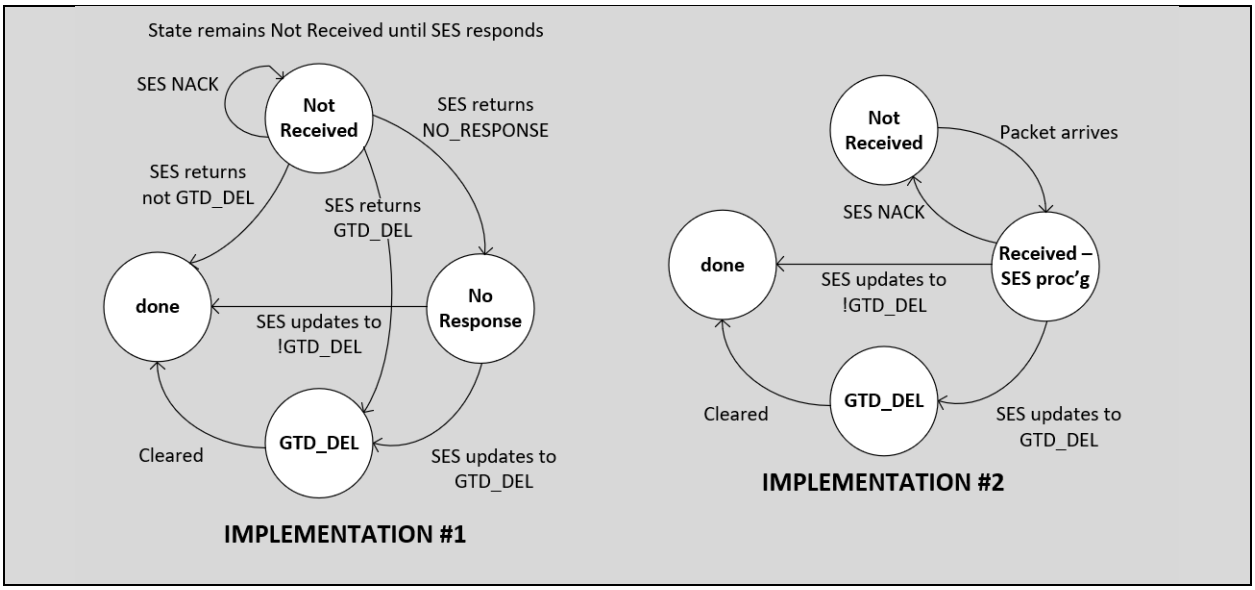
Alternatively, an implementation may choose to use a model closer to implementation #2. This option cannot distinguish UET_DEFAULT_RESPONSE versus UET_NO_RESPONSE when creating an ACK for duplicate packets and ACK Request CPs. It always generates UET_DEFAULT_RESPONSE in those cases.

The scenario when a duplicate packet or ACK Request CP is received while #2 is still in 'Received—SES processing state' may happen. If it does, the received packet is NACK'd with **pds.nack_code** = UET_RCVD_SES_PROCG.

Packets marked as UET_NO_RESPONSE may be acknowledged using a cumulative ACK in which case the source of the request will assume success for the packet. The source of the request cannot assume success for the message.

SES may NACK a packet (e.g., if out of message tracking state, in which case the packet is treated as not received).

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.



PDS MUST build a UET_DEFAULT_RESPONSE when a duplicate PSN or ACK Request CP is received—or send a NACK if SES is still processing the packet. UET_DEFAULT_RESPONSE is constructed as described in Table 3-69.

Table 3-69 - Rules for Constructing UET_DEFAULT_RESPONSE

Field Name	Size (in bits)	PDS Processing
list	2	Set to: UET_EXPECTED
opcode	6	Set to: UET_DEF_RESPONSE or UET_NO_RESPONSE
ver	2	Clear to zero
return_code	6	Set to: RC_OK or RC_NULL
message_id	16	If duplicate PSN and SES header includes **ses.message_id**, copy that field. If CP, copy from **pds.payload** (Section 3.5.16).
ri_generation	8	If present in the packet that triggered the ACK, then use that value otherwise set to zero
JobID	24	If duplicate PSN, copy **ses.JobID**. If CP, set to zero.
modified_length	32	If duplicate PSN, copy **ses.request_length**. If CP, set to zero.

3.5.14 Transmit Scheduling

PDS should avoid starving any PDCs sharing a traffic class for an extended period of time while servicing other PDCs. PDS may provide hierarchical scheduling where some PDCs are given higher priority than other PDCs in the same traffic class. This applies to PDS requests, CPs, and acknowledgements.

Higher priority SHOULD be done as weighted fair queuing (WFQ) rather than strict priority (SP), as SP scheduling may lead to blocking or starvation.

Figure 3-85 illustrates a potential transmit scheduling architecture that is well aligned with UET.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

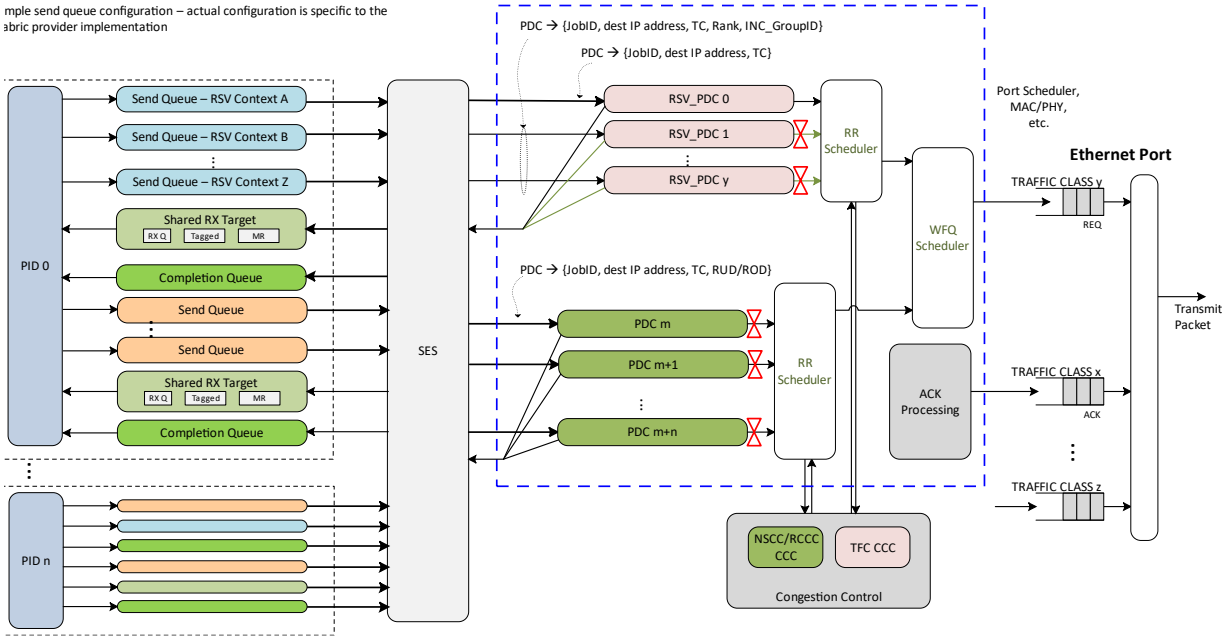


Figure 3-85 - Example Transmit Scheduler

3.5.15 Loss Detection and Recovery

PDS MUST detect the loss of packets using the methods listed in Table 3-70. The following subsections provide additional information on each method.

Table 3-70 - Loss Detection

Method	Requirement	Summary Description
Trimming	MUST	Trim indicates packet was lost—generate a NACK when trim arrives
NACK	MUST	NACK indicates a packet was lost—accept NACK and process associated PSN as lost
RTO	MUST	Timer at source that is sent on transmit and cleared on ACK
Early Loss Detection for Requests	SHOULD	Provide one or more methods to determine if a packet was lost sooner than RTO
Loss Detection for ACKs	SHOULD	Provide one or more methods to determine if an ACK was lost sooner than RTO

3.5.15.1 Trimming

Packet trimming is used by UET to provide an explicit indication of packet loss and to enhance congestion control performance. If the packet is validated using the UET CRC or authenticated using TSS and the packet was not actually trimmed because it is shorter than MIN_TRIM_SIZE, the packet can be accepted for normal processing.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

The PDS requirements are summarized below. Refer to the UE packet trimming Section 4.1 for more information.

- A trimmed packet MUST be identified by the value of its **ip.dscp** field.
- The IP length of a trimmed packet MUST NOT be verified.
- The UDP length of a trimmed packet MUST NOT be verified.
- The UET CRC may be validated if present.
- The packet may be decrypted and authenticated when the **ip.dscp** field indicates the packet was trimmed but the packet is fully intact (e.g., ICV is present).
- The packet may be partially decrypted (i.e., the portion of the packet that is received can be decrypted) and not authenticated (when the ICV is not present). In this case use of the packet is limited as described in the next paragraph.

If a trimmed packet is received which cannot be validated or authenticated, the rules for NACK generation described in Table 3-71 MUST be used or the packet may be dropped. The PDS header fields in trimmed packets are not used for any purpose beyond generating a NACK or, in the specific case of a RUDI request, the original packet is retransmitted. The NACK code depends on the PDS packet type. Trimmed CPs are dropped.

Table 3-71 - Rules for NACK Generation with Trimmed Packets

Trimmed PDS Packet Type	Generate a NACK	NACK code / Description
RUD Request	Yes	UET_TRIMMED or UET_TRIMMED_LASTHOP
RUD ACK	No	Retransmit original RUD Request
ROD Request	Yes	UET_TRIMMED or UET_TRIMMED_LASTHOP
RODL Request	Yes	UET_TRIMMED or UET_TRIMMED_LASTHOP
ROD ACK	No	Retransmit original ROD Request
RUDI Request	Yes	UET_TRIMMED or UET_TRIMMED_LASTHOP
RUDI Response	No	Retransmit the original RUDI Request
CP	No	All CP are small and SHOULD NOT use a DSCP that allows trimming. CPs that are assigned unique PSNs and are trimmed require retransmission and would need loss detection mechanism (e.g., RTO) to be recovered.

3.5.15.2 NACK Loss Detection

A destination PDC may generate a NACK for the reasons listed in Section 3.5.12.7. When a NACK is received at a source, the associated packet is considered lost.

3.5.15.3 RTO Loss Detection

A PDC MUST maintain a timer, *RTO_TIMER*, and retransmit all associated packets that have not been acknowledged when the timer expires (i.e., one timer per PDC). This timer is set to *RTO_INIT_TIME* when the packet is transmitted. See *RTO_INIT_TIME* in the PDS configuration parameters Section 3.5.5.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

A PDC SHOULD maintain a timer or timestamp per outstanding packet and retransmit an unacknowledged packet when its associated timer/timestamp expires (i.e., one timer per outstanding/unacknowledged packet). Alternatively, a per PDC timer may be used.

The time for declaring a lost packet is programmable via the *RTO_INIT_*TIME parameter and is settable in the range of 0 µsec to 8 sec. Exponential backoff (multiplicative increase) in time between retry attempts SHOULD be provided each time the packet is retransmitted until *Max_RTO_Retx_Cnt* is reached. A packet's *RTO_TIMER* is restarted to its last value without exponential increase when a NACK is received for that packet. The timer may be suspended during a Probe CP transmission. How retransmission and exponential backoff coexist with RTO and the reception of NACKs is described in Section 3.5.12.7.

3.5.15.4 Early Loss Detection — Requests

Early loss detection for PDS requests refers to determining if a packet was lost before the *RTO_TIMER* expires. This may occur if trimming is not used or if a link failure occurs. Some capability for early loss detection of PDS requests SHOULD be supported. The method(s) used are implementation specific. This section provides several example methods.

3.5.15.4.1 OOO-based Loss Detection

The OOO-based method for early loss detection involves monitoring the out-of-order count (**pds.ack_cc_state.ooo_count**) that the destination received. Table 3-72 lists the local variables used in the OOO-based loss detection method. No attempt is made to compress the fields in this example. Some fields could be reduced in size based on the maximum number of outstanding packets supported or because the field uses units of 256 bytes, or other criteria, etc. Some detection fields are required for every PDC and some are per CCC or global, as indicated in the field size column of Table 3-72.

Two variations of the approach are described. One relies on **pds.ack_cc_state.ooo_count** from the destination as carried in ACK_CC or ACK_CCX headers. Because this field is optional, the alternative is to calculate *OOO_COUNT* at the source. The **pds.ack_cc_state.rcvd_bytes** field, a count of all bytes received on a CCC, is described in the CMS Section 3.6.9.2.1.

Table 3-72 - Early Loss Detection Fields per CCC—OOO Example

Field Name	Field Size	Description
FIELDS FROM CMS		**Used in NSCC algorithm**
cwnd	32 bits (per CCC)	Number of bytes, calculated by the CC algorithm
prev_rcvd_pdc_bytes	32 bits (per PDC)	Highest **pds.ack_cc_state.rcvd_bytes** received on the PDC over previously received ACKs, multiplied by 256 for units = bytes. This can be a 24-bit count since the least significant 8 bits are always 0. Monotonically increasing count of the number of bytes received on a PDC incremented when a packet is received and wraps at max.
inflight_ccc_bytes	32 bits (per CCC)	Counter of bytes transmitted on the CCC but not yet acknowledged, across all PDCs in CCC
RECEIVED PACKET FIELDS		**Fields from the received packet @source**

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Field Name	Field Size	Description
ack.rcvd_bytes	24 bits (per PDC)	ack.rcvd_bytes refers to the **pds.ack_cc_state.rcvd_bytes** field in an ACK packet, multiplied by 256 for units = bytes
ack.ooo_count	16 bits (per PDC)	See OOO_COUNT in this table
LOCAL CONFIGURATION		**Configuration for OOO loss detection**
loss_retx_factor	8 bits (global)	Configured value used to approximate a number of RTTs
min_retx_factor	32 bits (global)	Minimum threshold in bytes
LOSS DETECTION STATE		**State for OOO_COUNT loss detection**
OOO_COUNT (@destination)	16 bits (per PDC)	Number of PSNs received at destination beyond the first missing PSN (i.e., past the first hole in the bitmap)
Dest_Lowest_Not_Rcvd_PSN (@destination)	32 bits (per PDC)	Lowest PSN not yet received
avg_tx_pkt_size (@source)	16 bits (per CCC)	Calculated average size of packet in bytes—implementation-specific calculation—or may be set to a constant if the workload is known
Src_Lowest_Not_Rcvd_PSN (@source)	32 bits (per PDC)	Lowest PSN not yet ACK'd by SACK bitmap, distinct from *CACK_PSN* in that clear is not considered
inflight_pdc_bytes (for each PDC in the CCC)	32 bits (per PDC)	Count of bytes transmitted on the PDC but not yet acknowledged—used to scale *cwnd* per PDC in these calculations
last_rtx_psn (@source)	32 bits (per PDC)	Snapshot of *Src_Lowest_Not_Rcvd_PSN* when entering loss recovery mode
recovery_psn (@source)	32 bits (per PDC)	Snapshot of highest sent PSN when entering loss recovery mode
highest_sent_psn (@source) (aka HPSN)	32 bits (per PDC)	Highest PSN that was already sent into the network
DYNAMIC FIELDS		**Temporary fields—used during calculations**
loss_threshold_bytes	32 bits (per CCC)	Calculated value (dynamic threshold) used to determine if a packet is likely lost per this method
loss_threshold_pkts	16 bits (per CCC)	Calculated using avg_tx_packet_size
cwnd_pkts	16 bits (per CCC)	Calculated using avg_tx_packet_size
newly_rcvd_bytes	24 bits (per PDC)	Number of bytes newly acknowledged in the ACK just received (i.e., the difference between the previous **pds.ack_cc_state.rcvd_bytes** and just arrived **pds.ack_cc_state.rcvd_bytes**, adjusted to units = bytes)
loss_recovery_mode	Boolean (per PDC)	Indicates if PDC is currently in loss recovery mode.

Because a CCC may include more than one PDC, stats are maintained at both the PDC and CCC level. The algorithm proportionally allocates *cwnd* to each PDC in the calculation based on the ratio of PDC bytes to CCC bytes inflight.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

The OOO-based loss detection calculations track the number of bytes that have been transmitted but not yet received at the destination on each PDC and on each CCC, as well as tracking the lowest PSN that has not been received at the destination per the SACK bitmap. The threshold used to determine if OOO-based loss detection could be triggered is calculated using:

```
loss_threshold_bytes = max(loss_retx_factor * cwnd * inflight_pdc_bytes /
                           inflight_ccc_bytes, min_retx_config)
```

where *loss_retx_factor* is roughly a number of loaded RTTs, and *min_retx_config* sets a minimum to reduce spurious retransmission. This is based, roughly, on the following concept: *cwnd* reflects how many bytes may be inflight on the CCC. This is adjusted for what percentage of those bytes are on this PDC (*cwnd * PDC bytes/CCC bytes*). Then allow for a few RTTs (*loss_retx_factor*) due to various delays. However, this can trigger when there are low rates of traffic, so include a minimum threshold (*min_retx_config*). If there are more out-of-order bytes than the calculated threshold, some packets may have been lost.

This calculated threshold is in units of bytes. It is translated to packets, which can be done by dividing the number of bytes in flight by the number of packets in flight, or using a configured number (e.g., MTU) if the workload is known. The result is *loss_threshold_pkts*. Similarly, *cwnd* needs to be translated from bytes to packets—*cwnd_pkts*.

```
loss_threshold_pkts = loss_threshold_bytes / avg_tx_pkt_size
cwnd_pkts = cwnd_bytes / avg_tx_pkt_size
```

When loss is detected, the PDS source enters the loss recovery mode and takes a snapshot of the lowest PSN not yet received at the destination based on SACK bitmap and the highest sent PSN (*HPSN*). Any unacknowledged packets between *Src_Lowest_Not_Rcvd_PSN* and *HPSN* are considered lost. PDS ACKs continue to arrive with SACK bitmaps and these may update some of these PSNs to ‘received’ state.

As each ACK is received, the *OOO_COUNT* loss recovery mechanism updates the state that was snapshot on entering loss recovery mode; the update is based on *Src_Lowest_Not_Rcvd_PSN* and the SACK bitmap. It then retransmits the packets considered lost while obeying the window limit. After all retransmissions are completed, new packets are transmitted if the window allows. While in loss recovery mode, *OOO_COUNT* is ignored.

At the destination, *OOO_COUNT* is generated. The following pseudo-code describes how *OOO_COUNT* may be calculated. The term ‘received’ here means the packet was accepted and passed to SES. The function *find_next_lowest_psn_not_received()* is similar to calculating *CACK_PSN* except *CACK_PSN* advances only when a PSN is cleared, while *Dest_Lowest_Not_Rcvd_PSN* advances as soon as a PSN is received.

```
init OOO_COUNT = 0;
if packet received {
   if PSN == Dest_Lowest_Not_Rcvd_PSN {    # advance ‘lowest PSN not rcvd’
      PREV_LOW = Dest_Lowest_Not_Rcvd_PSN;
      Dest_Lowest_Not_Rcvd_PSN = determine_new_lowest_not_rcvd_psn();
```


Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

```
      OOO_COUNT = OOO_COUNT – (Dest_Lowest_Not_Rcvd_PSN – PREV_LOW);
   } else {
      OOO_COUNT = OOO_COUNT + 1;
   }
}
```

Alternatively, *OOO_COUNT* can be calculated at the source using:

```
OOO_COUNT * avg_tx_pkt_size = ooo_rcvd_bytes =
 ((HPSN – Src_Lowest_Not_Rcvd_PSN) * avg_tx_pkt_size) – inflight_pdc_bytes
```

Refer to the informative text box at the end of this section for information on this calculation.

The count of inflight bytes per PDC and per CCC can be determined using the received **pds.ack_cc_state.rcvd_bytes** field provided in the ACKs as shown in the 'On packet transmission' section of pseudo-code below. The *inflight_ccc_bytes* is required by NSCC; *inflight_pdc_bytes* and *avg_tx_pkt_size* are added for this OOO loss detection method.

```
On packet transmission { // Increment in flight by #bytes transmitted
   inflight_ccc_bytes = inflight_ccc_bytes + (tx_packet_size)
   inflight_pdc_bytes = inflight_pdc_bytes + (tx_packet_size)
   avg_tx_pkt_size = avg_tx_pkt_size * weight + tx_packet_size * (1-weight)
}

On PDS ACK received { // ACKs out-of-order, confirm Rcvd_Bytes increased
   if (ack.rcvd_bytes > prev_rcvd_pdc_bytes){
       Calculate how many bytes destination received since last ACK,
      newly_rcvd_bytes   = (ack.rcvd_bytes – prev_rcvd_pdc_bytes) * 256;
      // Reduce number of bytes inflight by amount the destination received
      // Since there may be multiple PDCs per CCC, track each independently
      inflight_pdc_bytes = inflight_pdc_bytes – newly_rcvd_bytes
      inflight_ccc_bytes = inflight_ccc_bytes – newly_rcvd_bytes
      // Update tracker of how many bytes have been received at destination
      prev_rcvd_pdc_bytes    = ack.rcvd_bytes
     }
   if (SACK_bitmap.Src_Lowest_Not_Rcvd_PSN = 1) {
      // Track lowest not rcvd PSN based on SACK bitmap
      Src_Lowest_Not_Rcvd_PSN = determine_new_lowest_not_rcvd_psn()
   }

On PDS ACK arrival {
   if (loss_recovery_mode = FALSE) {
      // Update fields to latest state from ACK
      loss_threshold_bytes = max(loss_retx_factor * cwnd *
         inflight_pdc_bytes / inflight_ccc_bytes, min_retx_config)
      // if using OOO_COUNT from destination via ACK:
      loss_threshold_pkts = loss_threshold_bytes / avg_tx_pkt_size
      // if calculating OOO_COUNT at source:
      ooo_rcvd_bytes = ((HPSN – Src_Lowest_Not_Rcvd_PSN) * avg_tx_pkt_size) -
                        inflight_pdc_bytes

      cwnd_pkts = cwnd_bytes / avg_tx_pkt_size
   }
   // Check if criteria indicates loss recovery mode should be enabled
   // if using OOO_COUNT from destination via ACK:
```


Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

```
   if (OOO_COUNT > loss_threshold_pkts) && loss_recovery_mode = FALSE) {
   // if calculating OOO_COUNT at source:
   if (ooo_rcvd_bytes > loss_threshold_bytes) && loss_recovery_mode = FALSE){
      loss_recovery_mode = TRUE
      recovery_psn = highest_sent_psn
      last_rtx_psn = Src_Lowest_Not_Rcvd_PSN;  // First PSN considered lost
   }
   if (loss_recovery_mode) {
      // Retransmit as many lost PSN as cwnd allows
      while((last_rtx_psn < Src_Lowest_Not_Rcvd_PSN + cwnd_pkts) &&
            (last_rtx_psn < recovery_psn)) {
         enqueue last_rtx_psn for retransmission;
         last_rtx_psn = next unacked PSN between last_rtx_psn and
              recovery_psn
         if (last_rtx_psn == NULL) {
            last_rtx_psn = recovery_psn + 1;
         }
      }
   }
}  //End of ACK processing

On Retransmit {
   // Loss recovery processing already applied cwnd
   // Transmit all enqueued packets
   if (last_rtx_psn > recovery_psn) {
      loss_recovery_mode = FALSE;
   }
} //End of Retransmit processing
```

This attempts to identify all PSNs that are roughly within the *loss_retx_factor* congestion window relative to the oldest unacknowledged packet. When supported, packets are not retransmitted more than once using this loss detection mechanism

See Section 3.5.15.5.2 for how to recover lost ACKs using this loss detection mechanism.

Informative Text:

For clarity in comparing the destination and source methods for determining *OOO_COUNT*, the equations are simplified to fit in a single line by letting *pdc/ccc = inflight_pdc_bytes/inflight_ccc_byte* and excluding *min_retx_config* in this comparison. The *pdc/ccc* term is used when multiple PDCs share a CCC to scale the CCC state for each individual PDC.

Consider the calculation used in the destination method shown as equation A below. Equations A, B, and C are mathematically equivalent.

```
   A. (OOO_COUNT > loss_threshold_pkts)

   B. (OOO_COUNT > (loss_retx_factor * cwnd * pdc/ccc) / avg_tx_pkt_size

   C. OOO_COUNT * avg_tx_pkt_size > loss_retx_factor * cwnd * pdc/ccc
```


Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Average transmit packet size effectively converts *OOO_COUNT* to bytes. This is an estimate of the number of bytes that arrived out of order at the destination, *ooo_rcvd_bytes*, and can be calculated using:

```
OOO_COUNT * avg_tx_pkt_size = ooo_rcvd_bytes =
 ((HPSN – Src_Lowest_Not_Rcvd_PSN) * avg_tx_pkt_size) – inflight_pdc_bytes
```

3.5.15.4.2 EV-Based Loss Detection

Another method for early loss detection is to maintain a record of which PSN used each entropy value (EV). If an ACK is received for a higher PSN that used the same EV, the lower PSN(s) may be assumed to be lost. For N outstanding packets, this scheme requires O(N) PDS state to store EVs.

The state required can be optimized by introducing the concept of K "slots" where each slot uses the same EV, EVs are used in round-robin fashion, and thus the slots form an in-order list. Loss is detected using simple arithmetic: The next PSN expected on a slot is the last PSN on that slot + K. Optimizations could be to use part of the EV to identify the slot, or to use part of the PSN as the EV for a slot and do arithmetic on the bitmap to detect loss.

The example in Figure 3-86 provides precise early loss detection except for tail loss, which generally requires using Probe CPs as described in Section 3.5.15.4.3. The example is not optimized and is used here to provide clarity. The PDC SLOT STATE table in Figure 3-86 tracks the next *Expected_PSN* on each EV using simple addition. When an EV is changed or skipped (e.g., CMS detects congestion), an entry is added to the EV CHANGE TABLE. When all packets on the original EV are received, the EV CHANGE TABLE state is moved to the PDC SLOT STATE table and the EV CHANGE TABLE entry is deleted.

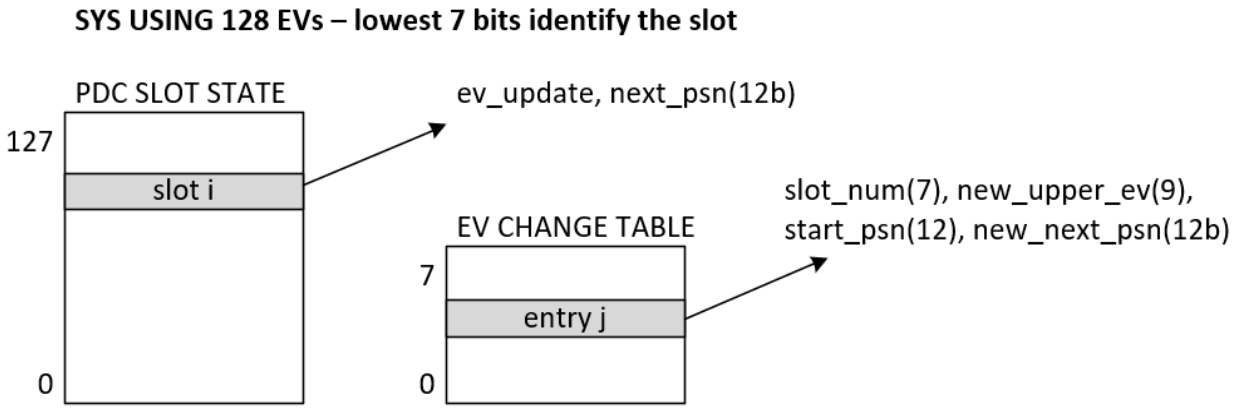


Figure 3-86 - Example PSN Tracking per EV

This example uses up to 128 EVs per PDC, where the lowest 7 bits of the EV identify the slot number. The slot state holds the next *Expected_PSN*, which is incremented by 128 when a PSN is received. If a higher PSN is received, any lower PSNs expected in that slot are determined to be lost. If the EV for the

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

slot is updated or skipped, the '*ev_update*' bit is set. Example processing code for an implementation that stores state as shown in Figure 3-86 follows:

```
SLOT_STATE = older EV state, set ev_update when EV is changed
EV_CHANGE = temporary state with new EV, moves to SLOT_STATE when all packets
on older EV are completed

if(cms_changes_EV) {
// Need to track 2 paths(EVs)during change
    add_ev_change_entry()
    SLOT_STATE.ev_update = TRUE
}
// Check if PSN used older EV
if(SLOT_STATE.ev_update = FALSE || ((rcvd_ev & 0xFF80) >> 7) !=
   EV_CHANGE.new_upper_ev) {
    // Yes, this is on older EV
    if (rcvd_psn != SLOT_STATE.next_psn) {
        // Not Expected_PSN so determine loss
        determine_lost_psn()
    }
    // Calculate next Expected_PSN
    SLOT_STATE.next_psn = rcvd_psn + 128
    if (SLOT_STATE.next_psn = EV_CHANGE.start_psn){
        // All PSN using older EV are done
        SLOT_STATE.ev_update = 0
        SLOT_STATE.next_psn = new_next_psn
        delete_ev_change_entry()
    }
} else {
    // This PSN is on the new EV, check if it's the Expected_PSN
    if (rcvd_psn != EV_CHANGE.new_next_psn) {
        // Not the Expected_PSN so determine loss
        determine_lost_psn()
    }
    // calculate the next Expected_PSN
        EV_CHANGE.new_next_psn = rcvd_psn + 128
    }
}
```

Informative Text:
When *cwnd* is reduced such that fewer packets are in flight, it may be beneficial to adjust the number of EVs in use accordingly. This improves the freshness of the EV congestion state and reduces the potential for tail loss. Probe CPs can be used to check slots that have been migrated to new EVs.

3.5.15.4.3 Tail Loss Detection

Tail loss refers to losing the last packet(s) of a message or EV such that the early loss detection mechanism doesn't trigger. For example, if using the OOO method in Section 3.5.15.4.1, then if a small number of packets is lost—below the *loss_threshold_pkts* or *ooo_rcvd_bytes* threshold —the detection method will not trigger.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

This section describes a method for using a Probe CP to detect tail loss by leveraging the loss recovery mode described in both OOO-based solutions in sections 3.5.15.4.1 and 3.5.15.4.2.

A per PDC timer, *Tail_Loss_Timer*, may be used. This timer is enabled and set to *Tail_Loss_Time* when there is any unacknowledged PDS requests on the PDC. The timer is restarted if any ACK or NACK packet is received on the PDC. The timer is disabled if there are no unacknowledged packets.

When the *Tail_Loss_Timer* expires, a Probe CP is transmitted and the timer restarted. If the timer expires again before the ACK is received, *Tail_Loss_Retx_Cnt* is incremented and the Probe CP is retransmitted. If the maximum retry count, *Max_Tail_Loss_Retx*, is reached, no additional Probe CPs are sent. RTO processing will handle the recovery or close the PDC if necessary. Alternatively, the retransmit count can be excluded and the Probe CPs transmitted until the *RTO_TIMER* expires.

The tail loss Probe CP uses the PDS Request traffic class, so it can be used to measure the network RTT. The payload is set to *SRC_LOWEST_NOT_RCVD_PSN & 0xFFFFFFF8*. The Probe CP carries a locally generated value, *PROBE_OPAQUE*, that is echoed back in the ACK in the **pds.probe_opaque** field. The EV is randomly selected. If the Probe CP is retransmitted, a different EV should be used.

When the ACK for the Probe CP is received, the RTT is compared to the target delay, *ccc.target_qdelay* (see CMS Section 3.6.13.3). If the probe-measured RTT is less than or equal to the target delay, loss recovery mode is triggered. If the probe-measured time is greater than the target delay, the timer is reset, and another Probe CP is generated. The RTO retry counter is not incremented in this case. The retry counter is cleared when any ACK is received.

If an ACK for another packet is received before the Probe CP ACK, the Probe CP ACK is ignored with respect to triggering loss recovery mode, and the *Tail_Loss_Timer* is restarted. If there are still unacknowledged PSNs, the timer remains enabled, otherwise the timer is disabled.

If there are no events pending other than waiting for ACKs to arrive to cause *CACK_PSN* to equal the highest sent PSN, do not trigger this Probe CP. Refer to Section 3.5.15.5.1.

3.5.15.4.4 Receiver Based Loss Detection

Similar to EV-based loss detection (see Section 3.5.15.4.2) and OOO-based loss detection (see Section 3.5.15.4.1), the destination may monitor local state and determine a packet was lost. A NACK with **pds.nack_code** = UET_RCVR_INFER_LOSS is transmitted that triggers the source to retransmit the packet.

3.5.15.5 Early Loss Detection of ACKs

Often lost ACK packets can be recovered by monitoring *CACK_PSN*. The exceptions are guaranteed delivery ACKs and the last ACKs when the PDC goes idle. Early loss detection for PDS ACKs refers to determining if an ACK was lost before the *RTO_TIMER* expires. This capability SHOULD be supported. The method(s) used are implementation specific. This section provides a number of example methods.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

3.5.15.5.1 Loss Detection for Guaranteed Delivery ACKs

CACK_PSN can be used as a method for inferring lost ACKs when the missing ACK carries a guaranteed delivery SES response. At the source, the following indicates the PSN has a guaranteed delivery SES response:

- The **pds.sack_bitmap** field shows a PSN arrived at the destination AND
- *CACK_PSN* is set to one less than that PSN

For example, if the received **pds.sack_bitmap** indicates PSN = N arrived and *CACK_PSN* = N-1, then PSN = N is guaranteed delivery. This is true since *CACK_PSN* would be set to N or greater if that packet didn't need a guaranteed response. Refer to Section 3.5.12.4.4.

In this scenario when using ROD or RODL PDCs, the loss of the ACK can be inferred immediately upon reception of the next ACK, since ACKs arrive in PSN order.

When using RUD, the ACKs may arrive out of order, and a mechanism such as a timer or monitoring an ACK bitmap can be used to infer the ACK loss. The time-out period should be larger than the network RTT plus an adjustment to account for SES processing at the destination.

When the source determines the ACK is lost, it MUST transmit an ACK Request CP or retransmit the original request packet. The source SHOULD generate the ACK Request CP unless the missing ACK is carrying read response data (refer to Section 3.5.16.2). I.e., it was an SES Read Request.

3.5.15.5.2 OOO-Based Loss Detection of ACKs

If using an OOO-based method as described in Section 3.5.15.4.1, when the loss recovery mode is triggered, if *Src_Lowest_Not_Rcvd_PSN* > *CACK_PSN* then it is likely the ACK associated with *CACK_PSN*+1 is guaranteed delivery and that is was lost. The source should send an ACK Request CP or, if the packet was a read request, retransmit the original packet.

If additional guaranteed delivery ACKs have been lost, this can be determined by monitoring how *CACK_PSN* advances when the first ACK (for *PSN = CACK_PSN*+1) arrives. If *CACK_PSN* advances to another PSN that is lower than *recovery_psn*, repeat this process (send an ACK Request CP or retransmit the original packet) until *CACK_PSN* is equal to or greater than *recovery_psn*.

3.5.16 Control Packet (CP)

The PDS uses CPs on RUD, ROD, and RODL PDCs. All CP types specified in Table 3-82 MUST be supported. CPs may use the traffic class of either PDS Request or PDS ACKs; refer to the CMS Section 3.6.4.7. The guidelines for assigning CPs to traffic classes are:

- CPs that are allocated a unique PSN on a ROD/RODL PDC MUST use the PDS Request traffic class.
- Refer to CMS Section 3.6.4.7 for information on mapping packet types to traffic classes.

This section specifies each type of CP and their expected use. The **pds.ctl_type** field identifies the CP, as defined in Section 3.5.10.9. The packet format is illustrated in Figure 3-88.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

A unique PSN is assigned to a CP when the packet requires guaranteed delivery. These CPs are allocated the next PSN in the PSN space in the direction the packet is sent (i.e., forward or return). When using a unique PSN, the **pds.flags.retx** bit is set when the packet is retransmitted. The **pds.flags.retx** bit may also be set when retransmitting a probe CP. Refer to the summary table at the end of this section.

Reception of a CP with unique PSN always triggers an ACK packet. The CP ACK may use the same ACK format used for PDS requests (e.g., ACK, ACK_CC or ACK_CCX). In this case **pds.next_hdr** is set to UET_HDR_NONE.

Some CP ACK packets carry a payload specific to the type of control packet. In this case, the **pds.next_hdr** is set to UET_HDR_CP_ACK. These use the ACK header with a payload that is unique to the CP type. This general CP ACK format is illustrated in Figure 3-87. The fields are defined in Table 3-73. The UET_HDR_CP_ACK next header enumeration MUST NOT be used with a PDS ACK_CC or ACK_CCX.

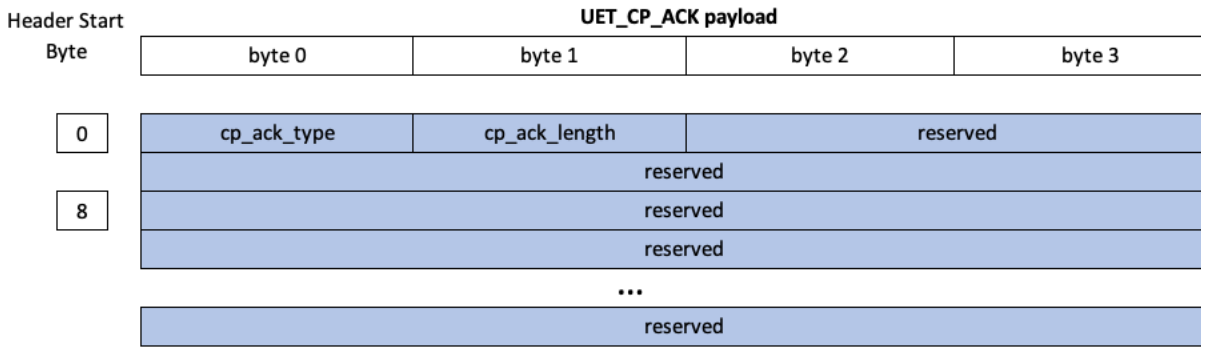


Figure 3-87 - UET_HDR_CP_ACK Payload Format

Table 3-73 - UET_HDR_CP_ACK Payload

Field Name	Size (in bits)	Description
cp_ack_type	8	Set to **pds.cp_type** in a packet that triggers the ACK
cp_ack_length	8	Length of negotiation payload in 4 B units, starting with and including **pds.cp_ack_type**
reserved	variable	Defined by the **pds.cp_ack_type**

CPs are normally transmitted over an established PDC. However, the NOOP and Negotiation CPs MAY be transmitted before a PDC is established; in these cases, the **pds.flags.syn** bit MUST be set. All other CPs MUST NOT be transmitted until the PDC has been established.

CPs MUST NOT be marked as trimmable. Trimmed CPs are dropped. The reception of a trimmed CP SHOULD be counted.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

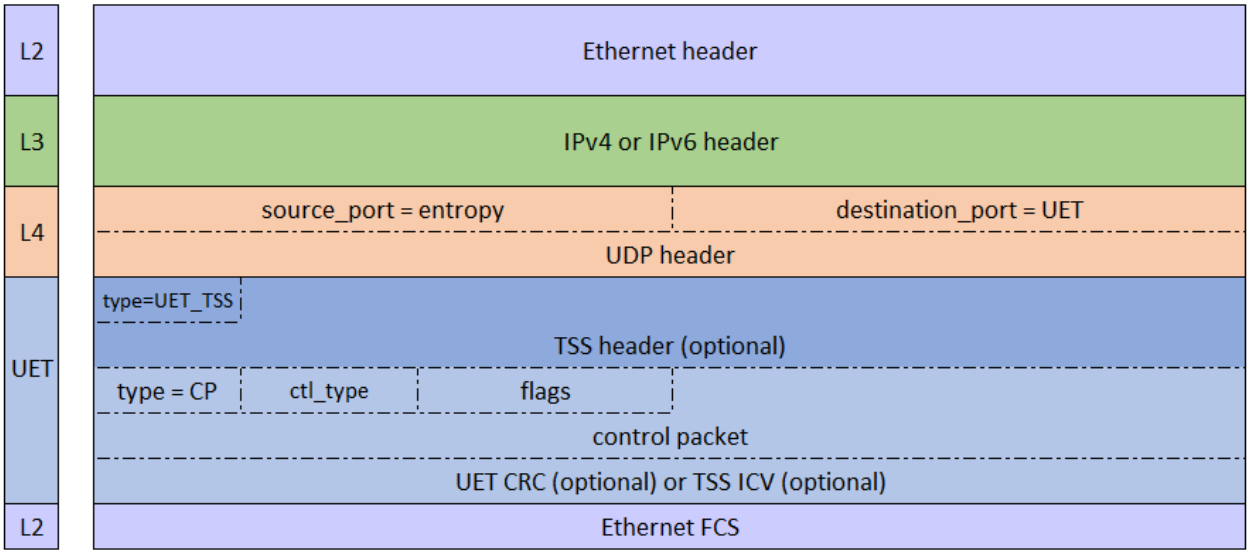


Figure 3-88 - CP Format

3.5.16.1 NOOP

The NOOP CP MUST be allocated a unique PSN, set **pds.flags.ar** to one and set **pds.flags.isrod** according to the PDC type. The reception of this CP triggers an ACK. No other processing is required at the destination. The CP **pds.payload** field is set to 0x0. The **pds.flags.ar** bit MUST be set to one, prompting the destination to generate an ACK. The **pds.flags.pdc_type** bits are set to 'b01 if the NOOP CP is sent on a ROD PDC, set to 'b00 if the NOOP CP is sent on a RUD PDC, and set to 'b10 if the NOOP CP is sent on a RODL PDC.

This CP can be used to open a PDC. Therefore, it uses the **pds.flags.syn** field and associated state used to open a PDC. Only NOOP and Negotiation CPs can be used to open a PDC. Other CPs MUST NOT be used to open a PDC.

The NOOP control packet triggers an ACK with **pds.next_hdr** = UET_HDR_NONE.

3.5.16.2 ACK Request

This CP is intended to recover lost ACKs when using ROD/RODL PDCs (as described in Section 3.5.21.3) or RUD PDCs (as described in Section 3.5.12.4.4). It requests an ACK for a specific PSN as identified by the **pds.psn** field in the CP header. The CP **pds.payload** field SHOULD carry the message ID associated with the requested packet in the first 16 bits of the **pds.payload** field. The CP **pds.payload** field is set to 0x0 if the message ID is not populated with a message ID. SES does not assign a message ID to all packets so there may not be a valid message ID. Also, an implementation may isolate SES and PDS functionality, such that PDS may not have a message ID available.

The source sends the ACK Request CP to the destination. ACK Request CPs MUST have **pds.psn** set to the PSN for which an ACK is requested and **pds.flags.ar** set to 1.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

An ACK Request CP SHOULD be generated when the SACK bitmap and *CACK_PSN* indicate a guaranteed delivery response is missing and when a trimmed ACK is received with the **pds.flags.req** field set to REQ_CLEAR. The exception is when the associated packet is a read request of size ≤ *Max_ACK_Data_Size*.

If the missing response is associated with a read request of size ≤ *Max_ACK_Data_Size*, then the original packet using that PSN MUST be retransmitted. Because the target does not save read data, the initiator reissues the read.

The destination responds as follows:

```
if received PSN is processed by SES {
    switch (SES response) {
        case not guaranteed delivery:
            send ACK with default SES response
            break
        case read response data:
            send NACK with pds.nack_code = UET_ACK_WITH_DATA
            break
        case guaranteed and response is available:
            send ACK with SES response
            break
        case guaranteed and response is not available
            send NACK with pds.nack_code = UET_GTD_RESP_UNAVAIL
            break
        default:
            fatal_error() // Not possible
            break
    }
}

If received PSN is still being processed by SES {
    Send NACK with pds.nack_code = UET_RCVD_SES_PROCG
} else {
    // The PSN was not received
    Send NACK with pds.nack_code = UET_PKT_NOT_RCVD
}
```

When an ACK is determined to be lost, implementations may choose to send an ACK Request CP or retransmit the original packet.

If an ACK Request CP is received with a PSN less than *CLEAR_PSN*, the packet MUST be silently dropped. This can happen due to network reordering where the packet is retransmitted before the ACK for the original packet arrives and then another packet carrying the clear passes the retransmitted packet. This event may be counted for diagnostic purposes.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

3.5.16.2.1 The ACK request control packet triggers an ACK with pds.next_hdr = UET_HDR_NONE.

3.5.16.3 Clear

The **pds.clear_psn_offset** field in the PDS Request header is used to calculate the *CLEAR_PSN* PCD state. *CLEAR_PSN* is used to free the state associated with ACKs at the destination. The Clear Command CP and Clear Request CP either request the cumulative *CLEAR_PSN* or carry a cumulative *CLEAR_PSN*. These are generally used when resources are tight, as processing clear events frees up guaranteed delivery state.

The Clear Command and Clear Request CPs MUST have **pds.flags.ar** set to zero and **pds.psn** set to zero, ensuring that no PDS ACK is generated.

3.5.16.3.1 Clear Command

The source sends a Clear Command CP to the destination when there is no PDS Request packet available to carry a required *CLEAR_PSN*. For example, if there are no pending packets to send and no outstanding, unacknowledged PSNs on the PDC and the **pds.flags.req** field in a received ACK indicates a clear is needed. This is also sent in response to a Clear Request CP. Not all ACKs require a clear.

When generating this CP, the CP **pds.payload** field is set to the *CLEAR_PSN*.

3.5.16.3.2 Clear Request

The destination sends a Clear Request CP to the source when there are ACKs requiring a clear. In general, the destination relies on the source to detect lost guaranteed delivery ACKs and generate an ACK Request CP (or retransmit the original request packet). Therefore, this CP may be sent based on local resource limitations (e.g., limited resources for storing guaranteed delivery responses) to free up storage of non-default responses.

When generating this CP, the CP **pds.payload** field is set to the PSN requiring the clear.

3.5.16.4 PDC Close

Two CPs are defined for closing a PDC as specified in Section 3.5.8.3: Close Command CP and Close Request CP. Only the initiator of the PDC can close a PDC. The target can request the PDC be closed. If the target requests a close and the initiator fails to acknowledge the Close Request CP within a configured time, the target MUST close the PDC in error.

The Close Command and Close Request CPs MUST be allocated a new PSN (the next PSN in the PDC PSN space) and have **pds.flags.ar** set to one, ensuring that the destination generates an ACK.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

3.5.16.4.1 Close Command

The initiator sends a Close Command CP to the target to indicate the PDC is being closed. If the target accepts a Negotiation CP to assume responsibility for tracking *Syn_Retx_Safety_Time*, then the *Remaining_Time* is placed in the **pds.payload** field as illustrated in Figure 3-89. It is encoded in units of 1 µs. Otherwise, the **pds.payload** field is set to 0x0.

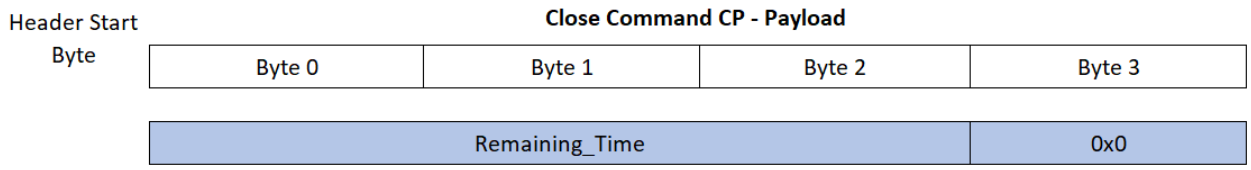


Figure 3-89 - Close Command CP Payload Format

The Close Command CP triggers an ACK with **pds.next_hdr** = UET_HDR_CP_ACK. The payload is illustrated in Figure 3-90. The fields are defined in Table 3-74.

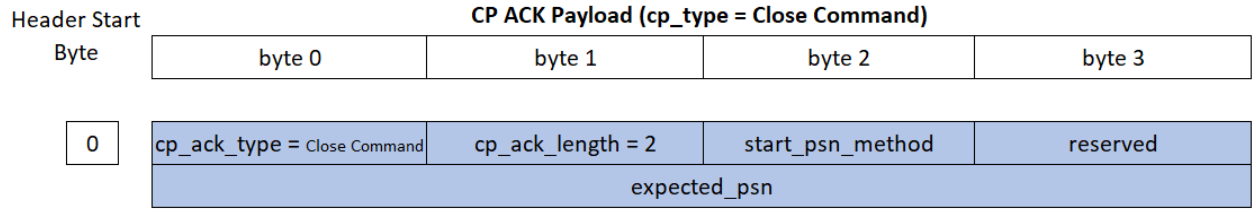


Figure 3-90 - Close Command CP ACK Payload Format

Table 3-74 - Close Command CP ACK Payload

Field Name	Size (in bits)	Description
cp_ack_type	8	Set to Close Command (see Section 3.5.16.9)
cp_ack_length	8	Payload length of 8 bytes
start_psn_method	8	0 = **pds.expected_psn** is not valid 1 = EXPECTED_0RTT_START 2 or higher = reserved for future use; **pds.expected_psn** is not valid
reserved	variable	Defined by the **pds.cp_ack_type**
expected_psn	32	Use this PSN when using **pds.start_psn_method** = EXPECTED_0RTT_START as defined in Section 3.5.8.2.1.

3.5.16.4.2 Close Request

The target sends a Close Request CP to the initiator to request the initiator close the PDC. The CP **pds.payload** field is set to 0x0.

The Close Request CP triggers an ACK with **pds.next_hdr** = UET_HDR_NONE.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

3.5.16.5 Probe

Probe CPs are generated at the source and trigger an ACK. The ACK is used to collect congestion control information. The conditions for generating a Probe CP are listed in Section 3.5.15 and in the CMS Section 3.6.13.7. A Probe CP may be generated for implementation specific reasons.

The Probe CP packet is not allocated a new PSN and MUST have the **pds.flags.ar** bit set to one and the **pds.psn** field set to zero. The **pds.probe_opaque** field in the CP header is used to carry a 16-bit opaque value that is reflected back in the associated ACK packet. The **pds.probe_opaque** field can be used to associate a received ACK with a specific Probe CP packet. The **pds.probe_opaque** field is copied from a received Probe CP packet into the **pds.ack_psn_offset** field in the corresponding ACK. The Probe CP **pds.payload** field is set to the desired selective ACK base PSN in the ACK response (when ACK_CC or ACK_CCX is in use).

When a destination receives a Probe CP, it generates an ACK because the **pds.flags.ar** is set in the received packet. The generated ACK MUST have the **pds.flags.p** bit set to indicate it is a Probe CP response and MUST NOT have the **pds.flags.req** field set to REQ_CLEAR or REQ_CLOSE.

When ACK_CC or ACK_CCX is in use, the **pds.sack_psn_offset** field is determined based on the PSN in the Probe CP **pds.payload** field. If the PSN in the Probe CP **pds.payload** field is lower than *CACK_PSN*, then *CACK_PSN* is used as the base PSN for the **pds.sack_bitmap** field in the ACK. Other fields are populated as usual.

3.5.16.6 Credit

There are two types of Credit CPs used with RCCC and TFC: one to send credit (Credit CP) and one to request credit (Credit Request CP). These packets are transmitted based on a request from CMS. Both credits CPs MUST set **pds.psn** to zero and **pds.flags.ar** to zero.

3.5.16.6.1 Credit

A Credit CP is used to carry credit allocated to sources. Credit CPs are generated at the destination and sent to the source. The conditions for triggering a Credit CP and the details of the **pds.payload.credit** field are discussed in the CMS sections 3.6.14.6 and 3.6.15.2. The Credit CP format is illustrated in Figure 3-91. The fields are defined in Table 3-75.

Credit is associated with a CCC. If multiple PDCs are sharing a CCC, the Credit CP may be sent on any PDC that is scheduled with the CCC.

An ACK_CC or ACK_CCX may be used to send credit, in which case the credit is carried in the **pds.ack_cc_state** or **pds.ack_ccx_state** fields as defined in the CMS Section 3.6.9.2. Either of these two mechanisms (Credit CP and ACK) may be used, or both may be used simultaneously, as the **pds.payload.credit** field is a monotonically increasing value.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

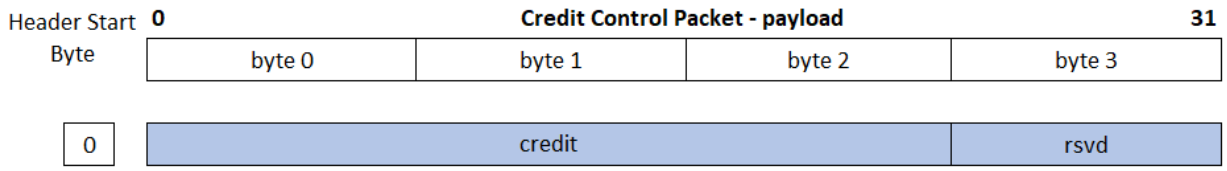


Figure 3-91 - Credit CP Payload

Table 3-75 - Credit CP Payload

Field Name	Size (in bits)	Field Description
credit	24	Credit generated by RCCC algorithm—refer to the CMS Section 3.6.9.3.1
rsvd	8	Set to zero

3.5.16.6.2 Credit Request

A Credit Request CP is used to request credit. It is generated by the source and sent to the destination. Figure 3-92 illustrates the payload format and Table 3-76 describes the fields.

Credit is associated with a CCC. The Credit Request CP may be sent on any PDC that is scheduled with the CCC.

Credit requests may also be carried in the **pds.req_cc_state** field of the PDS Request header. Either of these two mechanisms (Credit Request CP and PDS Request header) may be used or both may be used simultaneously, as the **pds.req_cc_state.credit_target** field is a monotonically increasing value.

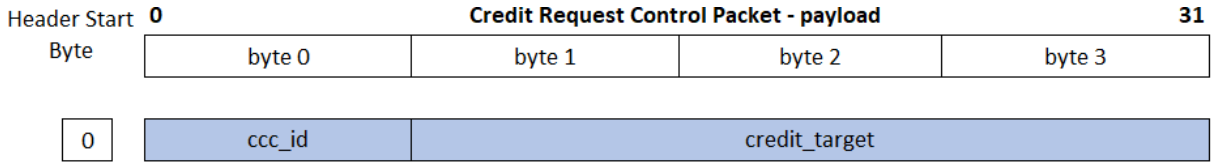


Figure 3-92 - Credit Request CP Payload

Table 3-76 - Credit Request CP Payload

Field Name	Size (in bits)	Field Description
ccc_id	8	Congestion Control Context Identifier
credit_target	24	Credit request

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

3.5.16.7 Negotiation

The Negotiation CP supports capability negotiation. If this control type is received and negotiation is not supported, then an ACK is generated as described below. This ACK may use any of the available acknowledgement formats: ACK, ACK_CC, or ACK_CCX.

Negotiation CPs MUST be allocated a unique PSN, have **pds.flags.ar** set to one, and set **pds.flags.isrod** according to the PDC type. This CP can be used to open a PDC; therefore, it uses the **pds.flags.syn** field and associated state used to open a PDC if sent during PDC creation.

The **pds.payload** field of the Negotiation CP carries an encoded payload based upon the type of negotiation. The first two bytes of the **pds.payload** are the **pds.neg_type** and **pds.neg_length** fields. The **pds.neg_type** field defines the type of negotiation being requested and the **pds.neg_length** field defines the length in increments of 4 B of the overall **pds.payload** field, including the **pds.neg_type** and **pds.neg_length** fields. The format of the remainder of the **pds.payload** beyond the **pds.neg_length** field depends upon the **pds.neg_type** field. The general format is illustrated in Figure 3-93 with field definitions in Table 3-77.

If **pds.neg_type** = PDS_NEG_NO_NEG and **pds.neg_length** = 0, the full **pds.payload** field has a fixed size of 32 bytes and each byte is set to zero. This assures backward compatibility with previous versions of the UE Specification.

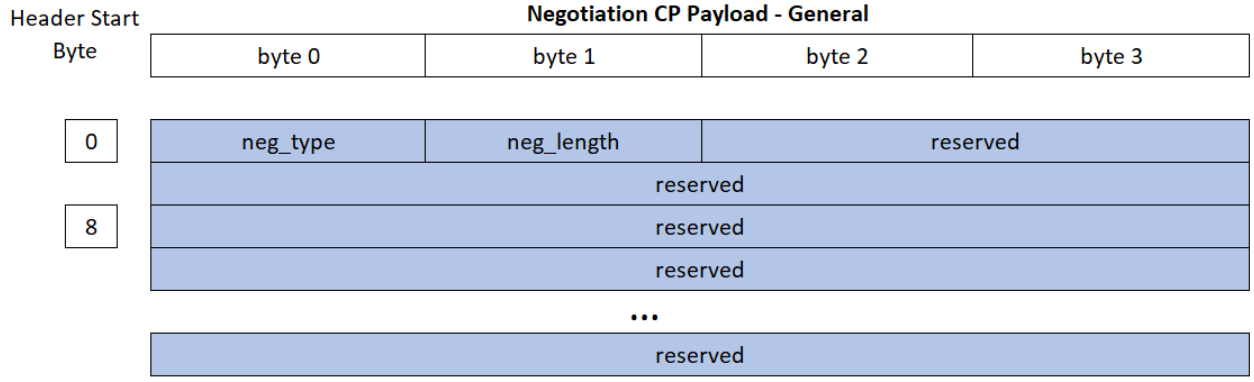


Figure 3-93 - Negotiation CP Payload Format - General

Table 3-77 - Negotiation CP Payload - General

Field Name	Size (in bits)	Description
neg_type	8	Indicates the type of Negotiation CP –defines the content and format of the rest of the **pds.payload** field.
neg_length	8	Specifies the length of negotiation payload in 4 B units starting with and including **pds.neg_type.**
Reserved	variable	Reserved

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

The negotiation types for the Negotiation CP are defined in Table 3-78. The payload field is reserved when **pds.neg_type** = PDS_NEG_NO_NEG. Section 3.5.16.7.1 describes the PDS_NEG_ON_OFF negotiation packet.

Table 3-78 - Negotiation Type (pds.neg_type)

Mnemonic	Constant	Description
PDS_NEG_NO_NEG	0x00	No negotiation request. The **pds.neg_length** field is set to 0x0, and the remainder of the **pds.payload** field MUST be 0x0. This type is always ACKed with the same ACK format used for PDS requests (ACK, ACK_CC or ACK_CCX).
PDS_NEG_ON_OFF	0x01	Simple on-off feature negotiation. A 32-bit mask of features request is provided. Only features with a simple "on or off" mechanism can use this negotiation type, and no additional negotiated parameters are included for these features.
Reserved	0x02-0xFF	Reserved

If negotiation is not supported, the Negotiation CP triggers an ACK with **pds.next_hdr** = UET_HDR_NONE.

If negotiation is supported, the Negotiation CP triggers an ACK with **pds.next_hdr** = UET_HDR_CP_ACK. The format of the payload field when **pds.cp_ack_type** = Negotiation is dependent on the **pds.neg_type** field.

3.5.16.7.1 On Off Feature Negotiation

The PDS_NEG_ON_OFF negotiation packet is illustrated in Figure 3-94. This Negotiation CP supports querying the availability of up to 32 capabilities using a 32-bit mask, **pds.neg_on_off_mask**. The list of negotiated capabilities is defined in Table 3-80.

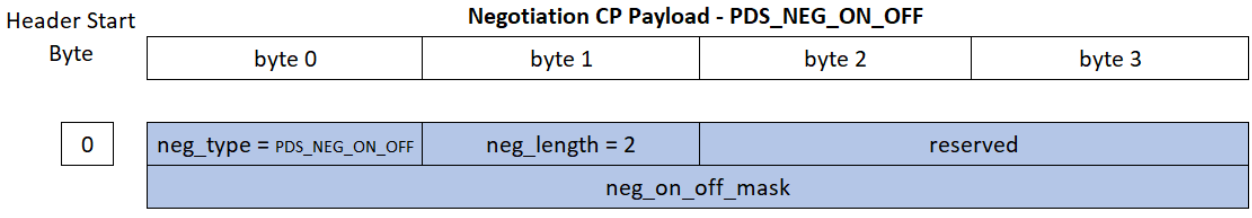


Figure 3-94 - Negotiation CP Payload Format

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Table 3-79 - Negotiation CP Payload for pds.neg_type = PDS_NEG_ON_OFF

Field Name	Size (in bits)	Description
neg_type	8	Type of this Negotiation CP = PDS_NEG_ON_OFF
neg_length	8	Set to 2 as this payload is 8 bytes
Reserved	16	Reserved
neg_on_off_mask	32	A 32-bit mask of requested features. Each bit corresponds to a single feature as defined in Table 3-80. Any number of bits may be set in a single negotiation request.

Table 3-80 - Negotiation CP Payload pds.neg_on_off_mask Format

Mnemonic	Bit Position	Description
SYN_RETX_TRANSFER	0	Request that the target allow the initiator to pass control of tracking the *Syn_Retx_Safety_Time* to the target.
Reserved	1-31	Reserved

The format of the payload field for the CP ACK to the Negotiation CP when **pds.neg_type** = PDS_NEG_ON _OFF is illustrated in Figure 3-95. Only bits defined in Table 3-81 may be set. All reserved fields are set to zero and ignored upon receipt.

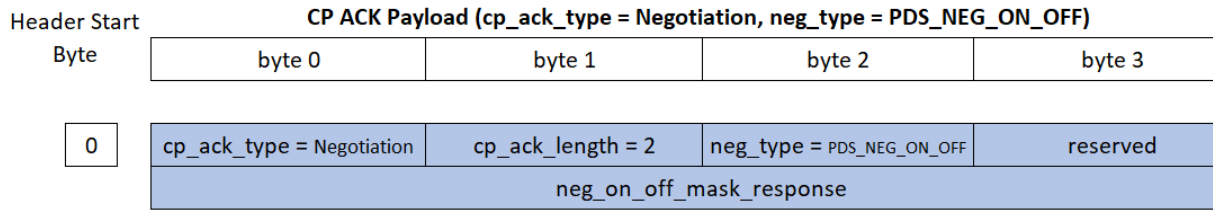


Figure 3-95 - Format of CP ACK Payload (pds.neg_type = PDS_NEG_ON_OFF)

Table 3-81 - CP ACK Payload Fields for pds.neg_type = PDS_NEG_ON_OFF

Field Name	Size (in bits)	Description
neg_type	8	8-bit negotiation type echoed from the negotiation packet.
reserved	8	Reserved
neg_on_off_mask_response	32	Bitmap field where each bit indicates that the corresponding feature request is supported. See Table 3-80. A bit is set only if the corresponding bit was set in the **pds.payload** of the Negotiation CP.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

3.5.16.8 PCM Telegram

The PCM Telegram control packet, Telegram CP, is defined to support extensible, typed control-plane messaging between UET endpoints implementing PCM. The Telegram CP provides a generic container for small, structured control messages (aka telegrams) whose semantics are identified by a type identifier and whose payload format is defined by the corresponding telegram type.

The Telegram CP is identified by a unique control packet type value (see Table 3-82). Packets are allocated a unique PSN. Packets MAY be sent in either direction on an established PDC. The **pds.flags.ar** bit MUST be set to 1.

The Telegram CP payload is carried in the payload field and has a fixed size of 32 bytes (see Section 3.6.18.6.3). This payload is submitted to the PCM subsystem for further processing. If the destination PDC is not controlled by PCM, the receiver returns a UET_UNEXP_EVENT NACK in accordance with the PDS NACK generation behavior described in Section 3.5.12.7.

3.5.16.9 CP Summary Table

Table 3-82 summarizes the fields used in CP headers by type. Source to destination means transmission from the initiator to the target on the forward direction, and from the target to the initiator on the return direction.

RODL MUST support the Close Command CP and Close Request CP. RODL does not require generating other control packets. Implementations MUST accept and respond to NOOP, ACK Request, Clear, and Clear Request CP messages. Support for Probe, Credit, Credit Request, and Negotiation CPs are not required on RODL, and these packets may be dropped without response.

Table 3-82 - PDS CP Summary

CP Type	CTL_TYPE	PSN	Usage	Control Payload	ACK
NOOP	0x00	Unique, new PSN	Either direction, can open a PDC	0x0	**pds.flags.ar** is set. Triggers an ACK; no other action.
ACK Request	0x01	PSN for which ACK is requested	Either direction	{message_id} or 0x0	**pds.flags.ar** is set. Triggers an ACK or NACK where *ACK_PSN* = **pds.psn** from CP.
Clear	0x02	0x0	Either direction	*CLEAR_PSN*	**pds.flags.ar** is cleared. Does not trigger ACK.
Clear Request	0x03	0x0	Either direction	PSN needing clear	**pds.flags.ar** is cleared. Does not trigger ACK.
Close Command	0x04	Unique, new PSN	Init -> Target on fwd direction	0x0	**pds.flags.ar** is set. Triggers an ACK.
Close Request	0x05	Unique, new PSN	Target -> Init on return direction	0x0	**pds.flags.ar** is set. Triggers an ACK.
Probe	0x06	0x0	Source to Destination	SACK bitmap base PSN	**pds.flags.ar** and **pds.flags.p** flags are set. Triggers an ACK.
Credit	0x07	0x0	Destination to Source	*Credit*	**pds.flags.ar** is cleared. Does not trigger ACK.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

CP Type	CTL_TYPE	PSN	Usage	Control Payload	ACK
Credit Request	0x08	0x0	Source to Destination	*Target_Credit*	**pds.flags.ar** is cleared. Does not trigger ACK.
Negotiation	0x09	Unique, new PSN	Either direction, can open a PDC	0x0 (32 B)	**pds.flags.ar** is set. Triggers an ACK.
Telegram	0x10	Unique, new PSN	Either direction	2 B telegram *type_id* + 30 B telegram (32 B)	**pds.flags.ar** is set. Triggers an ACK.

3.5.17 Semantic Responses

Every PDS Request triggers an associated SES Response. SES indicates to PDS if the semantic response requires guaranteed delivery, meaning it contains unique information (e.g., SES NACK code or data). SES always provides a guaranteed delivery semantic response for fetching atomics or reads if the result is carried in the ACK. SES may provide a guaranteed delivery semantic response for other requests, including when an error occurs. When SES indicates that an SES Response requires guaranteed delivery, PDS MUST request a clear by setting **pds.flags.req** to REQ_CLEAR in the associated ACK, and— with the exception of read response data—the FEP MUST maintain the state necessary to retransmit that ACK if lost.

The target PDS MUST maintain state for guaranteed delivery semantic responses until a clear is received, enabling retransmission of the response when a duplicate request packet is received. All other responses to duplicate packets use the default SES Response shown in Section 3.5.13. The destination MUST build the appropriate ACK if a duplicate request or ACK Request CP is received that is in the supported ACK window (Section 3.5.12.6).

All fetching atomic responses MUST be returned in the associated ACK packet, regardless of the *Max_ACK_Data_Size*. Fetching atomic responses up to 16 B MUST be supported when supporting fetching atomics. Fetching atomic results MUST be stored at the target and retransmitted if the associated ACK is lost. An implementation SHOULD NACK a fetching atomic operation if the response exceeds the supported size. In this case a NACK code of UET_ATOMIC_RESP_TOO_LARGE is used.

If an ACK has associated data from a read request (i.e., the read size is $\leq$ *Max_ACK_Data_Size*), the data is not stored at the target. The target marks the PSN as required delivery and generate a NACK if an ACK Request CP is received for that PSN. The initiator retransmits the original read request to recover the missing ACK in this case.

3.5.18 Reserved PDCs

Support for reserved PDCs is optional. The must statements in this section apply only if reserved PDCs are supported.

Reserved PDCs are intended to support use cases where handling a NACK due to lack of PDC or CCC resources is challenging. PDS allows reserving a pool of PDC and CCC resources that are allocated and

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

deallocated under software control. When using RUD or ROD, two reserved PDCs are required for bidirectional FEP-FEP data transfer.

PDS MUST allow reservation of a configured number of PDC and CCC resources for use by the reserved service as described in Table 3-83. The number of reserved PDCs and CCCs SHOULD be at least 16.

Reserved PDCs MUST be allocated and deallocated under the control of software and MUST NOT be closed dynamically due to resource limitations. That is, unlike normal PDCs, the PDC remains in the ESTABLISHED state until the associated context is destroyed explicitly.

Table 3-83 - Reserved PDC Configuration

Field Name	Type	Description
Num_Rsv_PDC	Configuration	Number of PDCs in the reserved PDC pool
Num_Rsv_CCC	Configuration	Number of CCCs in the reserved CCC pool

The following paragraphs use the terminology from the PDS_SES logical interface in Section 3.5.4.

When a reserved CCC is allocated for an initiator PDC, a handle is generated (e.g., *rsv_ccc_handle*). The handle may be the **ccc_id** or may be opaque.

When a reserved PDC is allocated for an initiator, a PDC handle is generated (e.g., *rsv_pdc_handle*). During allocation, the reserved PDC is associated with a reserved CCC using the *rsv_ccc_handle* if congestion control is enabled. Traffic is sent on the reserved PDC when the SES passes the *rsv_pdc_handle* with the SES transmit request.

Packets associated with a *rsv_pdc_handle* MUST use the same PDC and CCC. Only packets associated with a *rsv_pdc_handle* may use the associated PDC. During creation of a reserved PDC, the initiator MUST set the **pds.pdc_info.use_rsvd_pdc** flag. This assures a reserved PDC will be used at the target.

Target reserved PDC and CCC resources are not explicitly allocated by software. Determining the state of target PDCs requires a software query and deallocate service. This is needed to cover scenarios where the peer initiator FEP fails without closing the PDC.

If a packet arrives with the **pds.flags.syn** = 1, **pds.pdc_info.use_rsvd_pdc** = 1, and a new PDC is to be allocated, then a reserved target PDC and CCC are allocated. The CCC is only allocated if CCC is enabled for the associated traffic class. If **pds.flags.syn** = 1 and **pds.pdc_info.use_rsvd_pd**c = 0, a newly allocated PDC and CCC resource MUST NOT be taken from the reserve pool.

Transmit scheduler services for reserved PDCs are further described in Section 3.5.14.

Informative Text:
Reserved PDCs can be established by sending a UET_NO_OP or other semantic request to the destination FEP. The PDC follows the common establishment method, including supporting secure PSNs.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

3.5.19 Sequence Diagrams

The following four sections on RUD, ROD, RODL, and RUDI use packet sequence diagrams to illustrate how the protocols work. All sequence diagrams use the following key:

Table 3-84 - Sequence Diagram Key

Function	Description	Example
PDS requests	Solid line	
PDS ACKs	Dashed line	
SES headers	Fields in brackets are SES generated, and are carried in PDS Request/ACK; these are shown to provide context on the sequence, but are not used by PDS	[MID=1, RESP]
Colors	All requests/ACK in a direction are shown in the same color; a single PDC may use up to four colors: #1 Forward direction Request #2 Forward direction ACK #3 Return direction Request (Read Resp) #4 Return direction ACK	#1 #2 #3 #4

Figure 3-96 shows a sequence with text pointing out the meaning of each field and arrow. All sequence examples are informative.

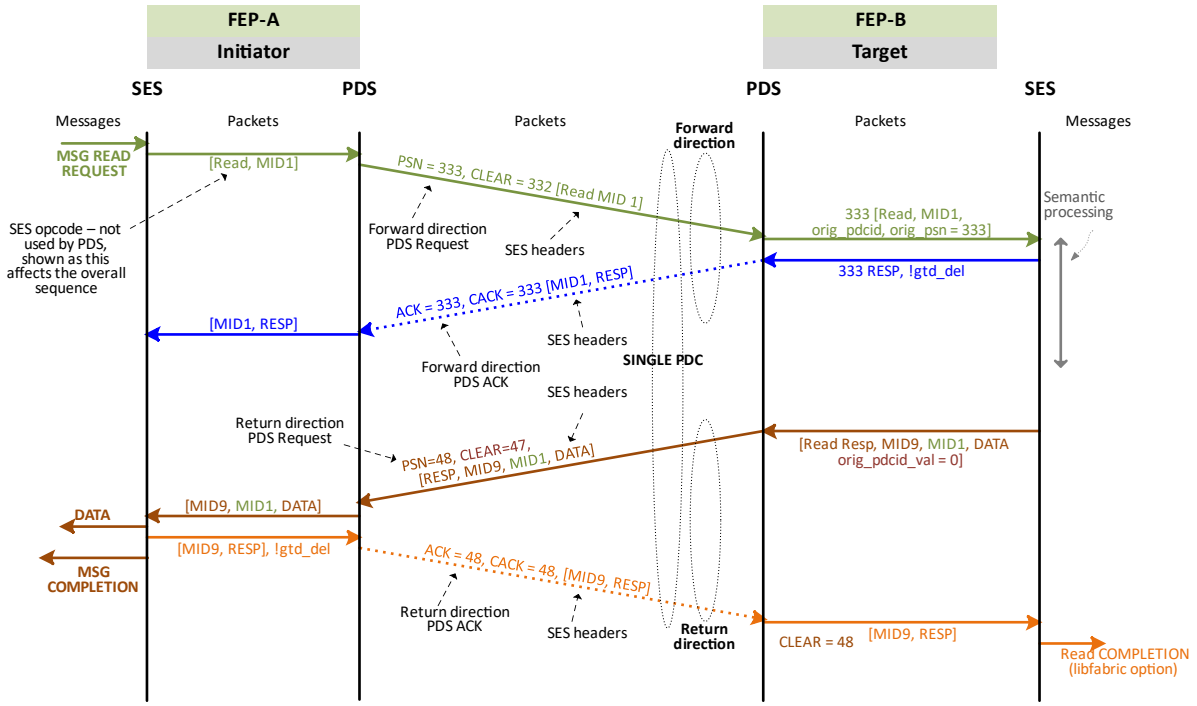


Figure 3-96 - Example Sequence with Key

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

The terms used in the sequence diagrams are abbreviated to allow them to fit in the figure and remain legible. Many of these abbreviated terms are neither the field names of the headers in the packet nor the computed values within the implementation, both of which are referenced in the text describing the sequence diagrams. For example, CLEAR in Figure 3-96 is intended to represent the *CLEAR_PSN* value that is calculated from the **pds.clear_psn_offset** and **pds.psn** fields or explicitly carried in the **pds.payload** field of a Clear Command CP.

3.5.20 Reliable Unordered Delivery

A RUD PDC is a reliable PDC established between a single initiating FEP and a single target FEP that guarantees packets are delivered once and only once. PDCs are ephemeral, established and closed on demand. Packets from multiple PIDs and/or send queues may share a PDC. All packets from a single message use a single PDC.

3.5.20.1 RUD Services

It is important to recognize that all SES Requests—send, write, read, atomic, etc.—are fundamentally the same to PDS.

1. RUD MUST transmit PDS Request packets as directed by the source SES and retransmit up to *Max_RTO_Retx_Cnt* times to attempt to deliver the packets to the destination.
2. RUD MUST deliver PDS Request packets to the destination SES once and only once.
3. RUD MUST support selective retransmission.
4. RUD MUST generate ACK packets at the destination that carry SES responses.
    a. PDS accepts an indication from SES whether each SES Response has guaranteed delivery.
    b. PDS MUST store guaranteed delivery SES responses (excluding data carried in an ACK for a read response) to enable retransmission of the associated ACKs if a valid duplicate request arrives (i.e., within the ACK window—see Section 3.5.12.6).
    c. PDS MUST construct the default SES Response to enable retransmission of the associated ACK if a valid duplicate request is received and the SES Response for the associated PSN was not guaranteed delivery.
    d. PDS MUST receive a clear indication for all PSNs with guaranteed delivery SES Responses before freeing the associated SES Response.
5. PDS MUST receive and process ACK packets at the source and pass the SES response to SES.
6. PDS MUST send *CLEAR_PSN* from source to destination in all PDS Requests. This is done by properly setting the **pds.clear_psn_offset** and **pds.psn** fields. The *CLEAR_PSN* computed value is not explicitly transmitted in PDS requests.
7. PDS MUST send *CLEAR_PSN* when a Clear Request CP is received—either in a PDS Request (as described in step 5 above) or using a Clear Command CP where *CLEAR_PSN* is loaded into the **pds.payload** field.
8. PDS MUST receive and process *CLEAR_PSN*.
    a. PDS MUST free any stored SES Response when the associated PSN is cleared.
9. PDS MUST send all read responses in the same direction for a single message.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

a. If a message is larger than PDS *Max_ACK_Data_Size*, then all responses use the return direction even if the last packet carries less than *Max_ACK_Data_Size* data

SES Read sequences operate differently depending on the configuration of *Max_ACK_Data_Size*. Refer to Section 3.5.12.1.

3.5.20.2 Standard Sequences

Reliable unordered delivery (RUD) sequences are presented in the form of ladder diagrams. This section specifies the normal sequences (i.e., PDS success cases) for RUD after a PDC is established.

3.5.20.2.1 Single-Packet Send Sequence—Non-Guaranteed Delivery

Send and write sequences are the same on best-effort and lossless networks. The standard RUD Send sequence for a single-packet message with SES Response not guaranteed delivery is shown in Figure 3-97. The sequence in Figure 3-97 proceeds as follows:

1. The PDC is idle with all previous packets successfully delivered, *PSN* = *ACK_PSN* = *CLEAR_PSN* = 332.
    a. *CLEAR_PSN* is automatically incremented when PSN has no guaranteed delivery.
2. SES at the initiator transfers a packet to PDS for transmission.
3. PDS at the initiator transmits the packet on the PDC with PSN = 333 and marks PSN = 333 as transmitted.
4. PDS at the target receives the packet, marks PSN = 333 as received, and delivers the packet to SES.
5. SES at the target provides the sematic response to PDS with no guaranteed delivery indication.
6. PDS at the target transmits an ACK packet for PSN = 333 that includes the SES Response.
    a. The ACK **pds.flags.req** != REQ _CLEAR as a clear is not required.
    b. *CACK_PSN* is incremented as SES Response is not guaranteed delivery.
    c. No state is required to be stored at the target beyond the PSN tracking state (e.g., bitmap/PSN).
7. PDS at the initiator receives the ACK packet, delivers it to SES, and frees the state for PSN = 333.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

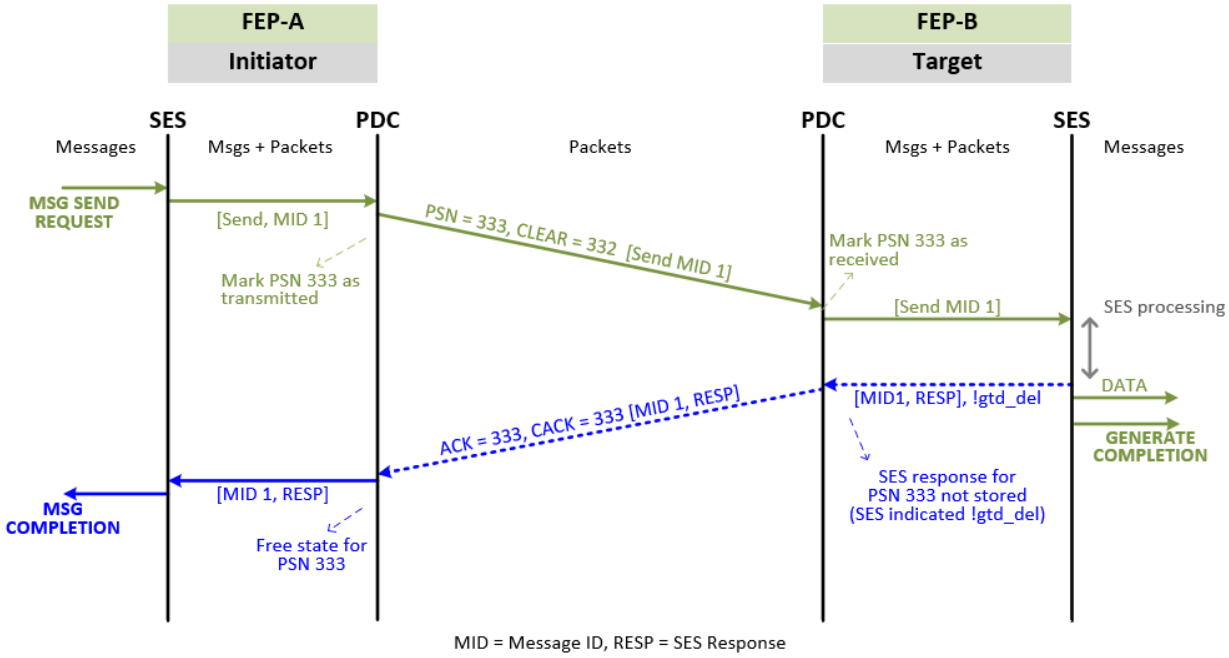


Figure 3-97 - Standard RUD Send Sequence for Single- Packet Message, Non-Guaranteed Delivery

The standard RUD Send sequence shown in Figure 3-97 also applies to non-fetching atomics. In this scenario with non-guaranteed delivery, a clear is not required. In general, PSN state may be freed once all lower PSNs are in clear state—either by receiving a *CLEAR_PSN* or because the SES Response did not require a clear.

3.5.20.2.2 Single-Packet Send Sequence—Guaranteed Delivery

PDS sequences for send and write are the same on best-effort and lossless networks. The standard RUD send sequence for a single-packet message with an SES guaranteed delivery response is shown in Figure 3-98. The sequence in Figure 3-98 proceeds as follows:

1. The PDC is idle with all previous packets successfully delivered, *PSN* = *ACK_PSN* = *CLEAR_PSN* = 332.
2. SES at the initiator transfers a packet to PDS for transmission.
3. PDS at the initiator transmits the packet on the PDC with PSN = 333 and marks PSN = 333 as transmitted.
4. PDS at the target receives the packet, marks PSN = 333 as received, and delivers the packet to SES.
5. SES at the target provides the sematic response to PDS with guaranteed delivery indication (*gtd_del*)
6. PDS at the target transmits an ACK packet for PSN = 333 that includes the SES Response.
    a. The ACK **pds.flags.req** = REQ _CLEAR, as a clear is required.
    b. *CACK_PSN* is not incremented, as SES Response is guaranteed delivery.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

c. The SES Response is stored at the target.

7. PDS at the initiator receives the ACK packet, delivers it to SES, and frees state for PSN = 333.
8. PDS at the initiator sends *CLEAR_PSN* = 333 to the target using either of the following:
    a. *CLEAR_PSN* is carried in the next PDS Request packet.
    b. A Clear Command CP may be used if there is no PDS Request packet to send; see Section 3.5.16.3 for details on when Clear CPs are generated.
9. PDS at the target frees the SES Response state for PSN = 333.
    a. *CACK_PSN* is incremented to 333 upon receiving *CLEAR_PSN*.

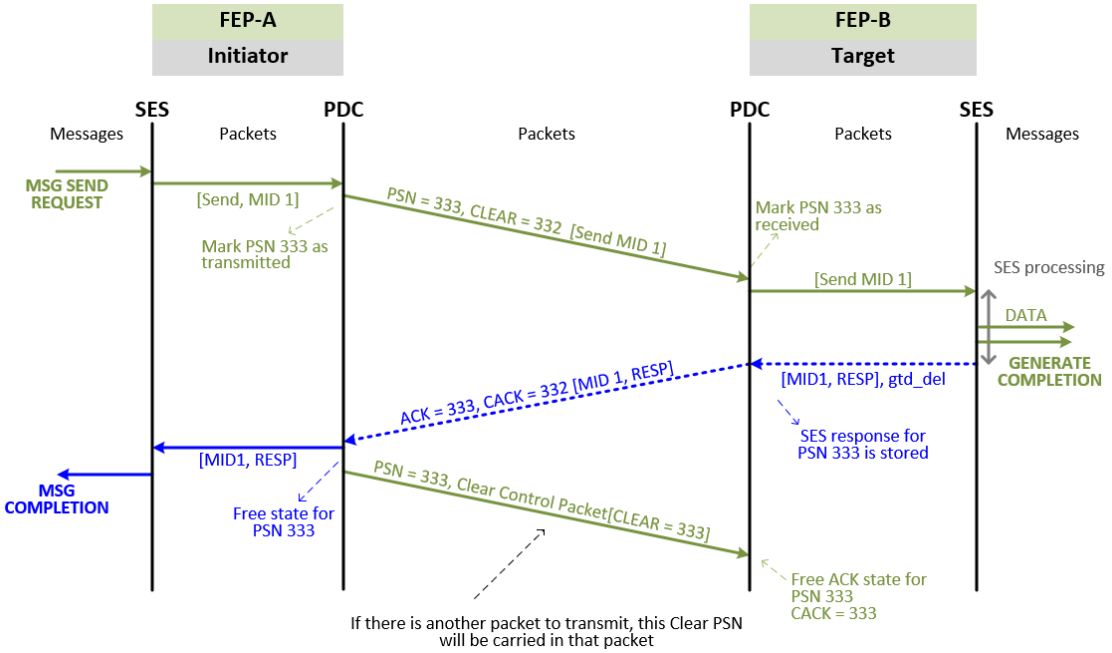


Figure 3-98 - Standard RUD Send Sequence for Single- Packet Message, Guaranteed Delivery

3.5.20.2.3 Multi-Packet Send Sequence—Non-Guaranteed Delivery

Send and write sequences are the same on best-effort and lossless networks. The standard RUD Send sequence for a multi-packet message is shown in Figure 3-99. The sequence in Figure 3-99proceeds as follows:

1. The RUD send sequence for multi-packet messages is the same as the RUD send sequence for a single-packet message.
    a. The single-packet sequence is repeated for each packet of the multi-packet message.
    b. Packets may arrive out of order at the destination.
    c. Acknowledgements may be cumulative—the figure shows an ACK every two packets; however, ACK generation events vary when cumulative ACKs are used; see Section 3.5.12.2.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

2. The PDC is idle with all previous packets successfully delivered, *PSN* = *ACK_PSN* = *CLEAR_PSN* = 332.
3. On the initiator, the PSN increases with each packet and *CLEAR_PSN* increases with *CACK_PSN* arriving from target.
    a. Because all these SES responses are not guaranteed, *CACK_PSN* increases as the generated ACKs fill in the SACK bitmap, rather than when *CLEAR_PSN* is received.
4. A single ACK arriving may ACK more than one packet; PDS may generate multiple messages to SES—the PSN ACK'd might not be consecutive.

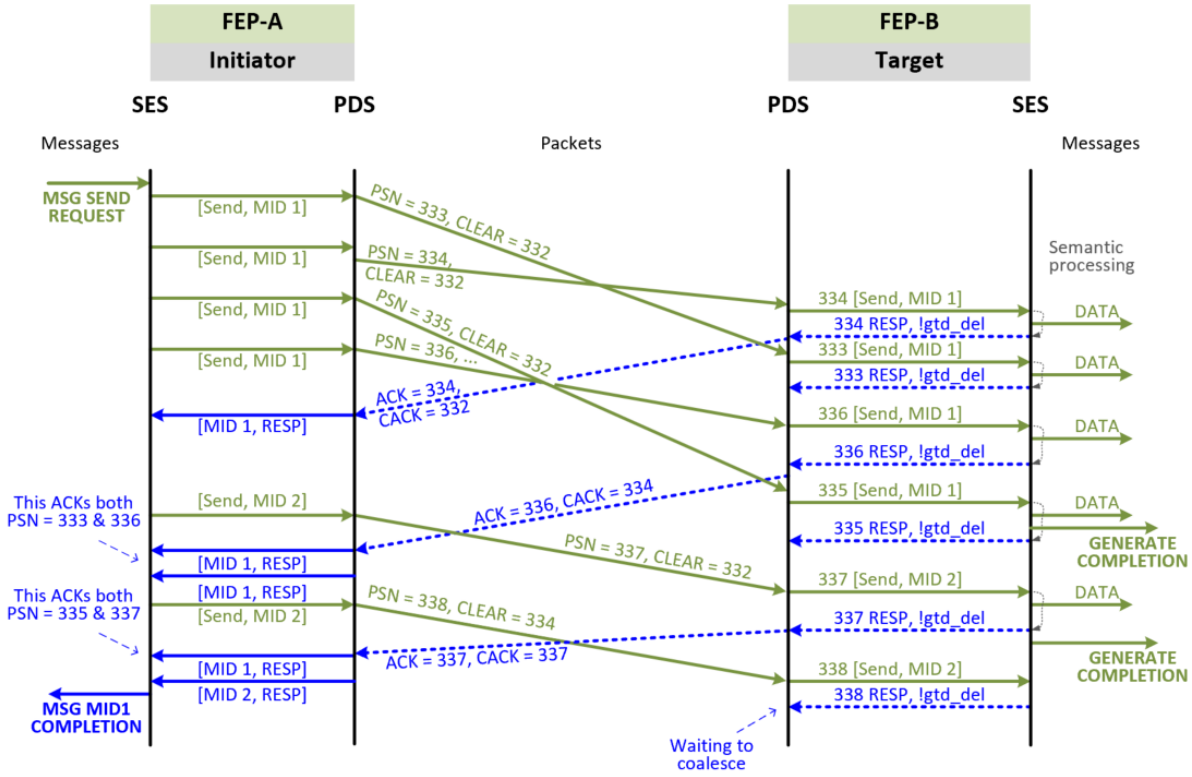


Figure 3-99 - Standard RUD Send Sequence for Multi-Packet Message, Non-Guaranteed Delivery

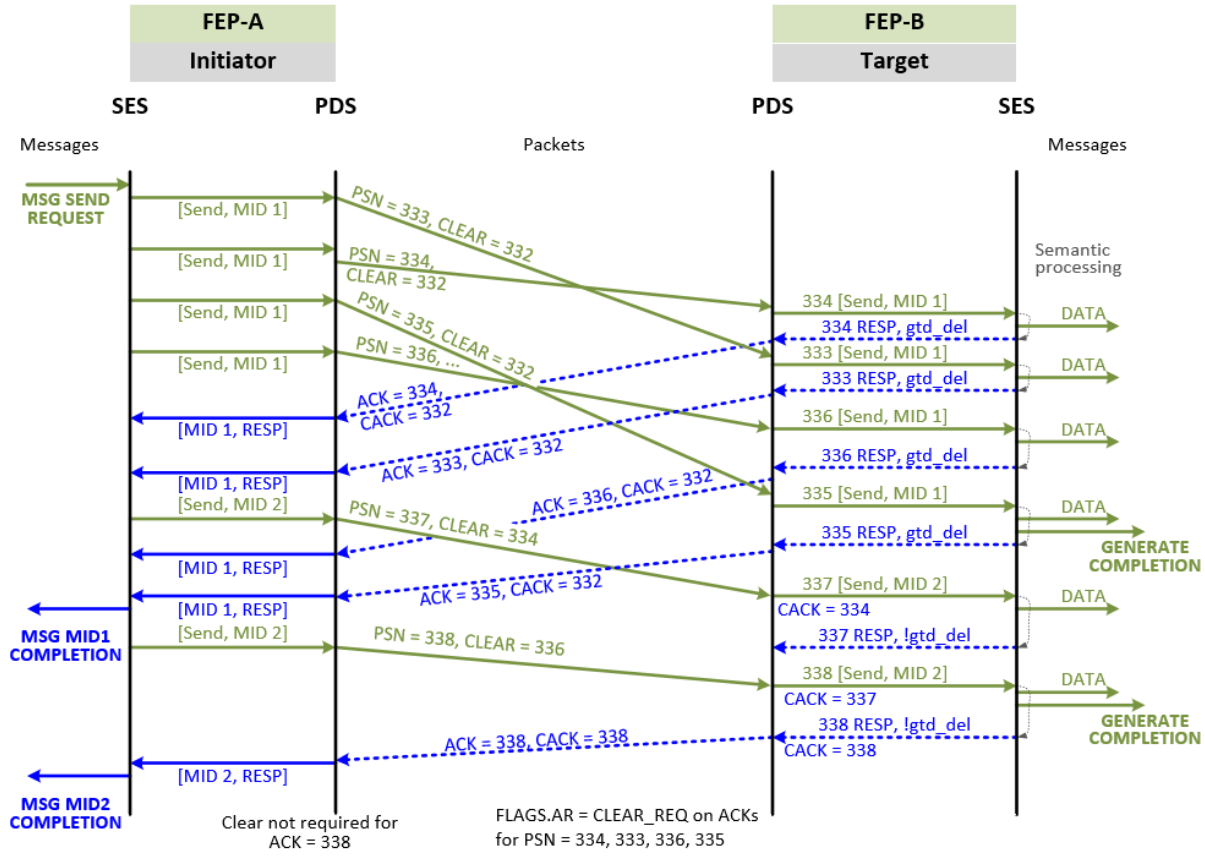
3.5.20.2.4 Multi-Packet Send Sequence—Guaranteed Delivery

The sequence for a RUD Send of a multi-packet message with a guaranteed delivery SES Response is shown in Figure 3-100. This example shows all four packets in MID = 1 marked as guaranteed delivery. If this is an error event, SES marks one packet as guaranteed delivery and may mark other packets as not requiring guaranteed delivery. This is allowed when any one packet in a message contains an error, because the message is completed in error, and it is not critical that all responses in a multi-packet message be delivered.

All PSNs with guaranteed delivery have independent ACKs (i.e., an ACK per PSN), while the PSN = 337 and 338 are coalesced into one ACK. Because CLEAR = 336 arrived before the ACK 338 and both 337 and

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

338 had *!gtd_del* responses, *CACK_PSN* advances to 338 as well. When *ACK_PSN* = *CACK_PSN*, no clear is required.



Note: *CLEAR_PSN* in the packet with PSN = 337 is set to 334 because the ACK for PSN = 335 has not yet arrived and *CACK_PSN* is less than 335.

Figure 3-100 - Standard Sequence for RUD Send of Multi-Packet Message with Guaranteed Delivery

3.5.20.2.5 Single Packet Read Sequence—SES Standard Header

Three of the types of SES headers described in the Semantics Section 3.4.3 include: standard (UET_HDR_REQUEST_STD), medium (UET_HDR_REQUEST_MEDIUM), and small (UET_HDR_REQUEST_SMALL). Only the standard header includes an **ses.message_id** field. This leads to differences in how reads are processed. The standard header sequence is described in this section; medium and small header sequences are described in Section 3.5.20.2.6. These sequences are the same for PDS regardless of SES header.

The standard RUD sequence for a single-packet read using SES standard headers on a best-effort network is shown in Figure 3-101. This shows a read with return data larger than *Max_ACK_Data_Size*, so the return direction is used to send return data.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

The sequence in Figure 3-101 proceeds as follows:

1. The PDC is idle with all previous packets successfully delivered, *PSN* = *ACK_PSN* = *CLEAR_PSN* = 332.
2. SES receives a read request from the provider and generates an SES Request with MID = 1.
    a. When using a standard SES Request header, SES assigns the read a unique message ID that allows FEP-A to associate the read response with the corresponding read request.
        i. SES medium and small headers do not include a message ID—see Section 3.5.20.2.6.
        ii. An implementation should make the message ID globally unique across all PDCs, allowing the initiator to efficiently identify read responses in the return direction (i.e., using a direct lookup on the message ID in the return direction).
        iii. Message IDs are unique per PDC.
3. The initiator transmits the SES Read to the target as a PDS Request with PSN = 333.
4. The target transmits an ACK for PSN = 333.
    a. The ACK carries the SES Response, which in this example is RC_OK — not guaranteed delivery — meaning the read request was accepted; the requested data is not in the ACK.
    b. The reception of this ACK completes the initiator SES processing of the read transaction.
5. The target SES processes the read and generates a request to PDS to send a packet back to FEP-A.
    a. This packet carries an SES UET_HDR_RESPONSE_DATA header and includes the data along with the original message ID (MID = 1 in this example) assigned by FEP-A.
    b. Optionally, the target SES may assign its own message ID. The target SES tracks the read transaction either as:
        i. A single transaction with locally assigned message ID (MID = 9 in this example), tracking when all return data packets in the transaction are sent and acknowledged; this allows a local libfabric counter to be incremented (an optional libfabric feature). A target-assigned message ID is efficient for this.
        ii. Individual packets where each packet is processed totally independently; the libfabric counter is not supported. A target-assigned message ID may or may not be used in this case; it is an implementation-specific choice.
6. The target transmits a PDS Request using PSN = 48.
    a. This packet carries the requested read data.
    b. This packet uses the return direction of the same PDC on which the associated request arrived; see Section 3.5.8.1.
    c. SES header fields carries the original message ID (**ses.read_request_message_id**) and carry a target-assigned message ID (**ses.response_message_id**) when one is assigned by the target; refer to the Semantics specification Section 3.4.2.14.
        i. The example shows a target-assigned MID = 9, which is echoed back in the ACK (when not coalesced).
7. After receiving a packet with PSN = 48, FEP-A transmits an ACK for PSN = 48.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

a. The return data is passed to the initiator SES.
    i. An example implementation option at the initiator is to maintain a packet counter for the original message ID; when the appropriate number of packets with return data arrives, the message is complete at SES; the transaction is complete when all packets are ACK'd at the target as well.
b. The read response transaction operates in the same manner as other transactions; when successful, the ACKs do not require a clear.

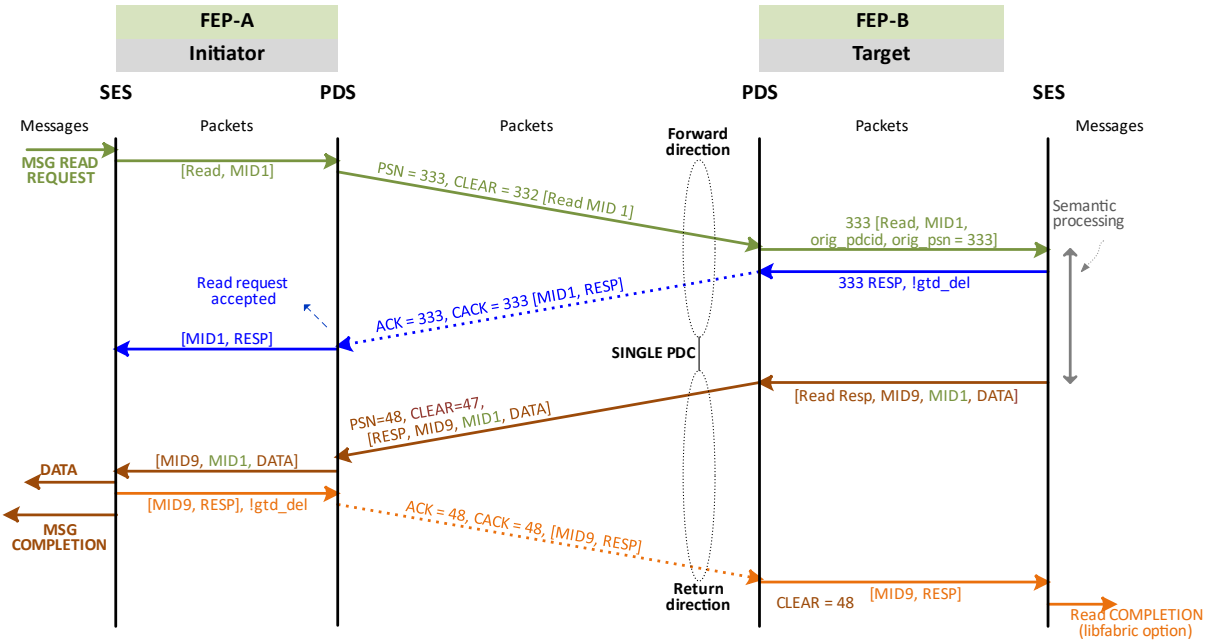


Figure 3-101 - Standard RUD Sequence for Single-Packet Read–Standard SES Header

Figure 3-101 shows how the original message identifier, MID = 1, is echoed back by SES, allowing the initiator SES to associate the data with the request.

3.5.20.2.6 Single-Packet Read Sequence—SES Medium and Small Headers

When SES uses the medium (UET_HDR_REQUEST_MEDIUM) or small (UET_HDR_REQUEST_SMALL) headers, there is no **ses.message_id** field. This leads to differences in how reads are processed. Instead of using message ID to associate the read response with the read request, the PSN of the request is carried in the SES Response header (**ses.original_request_psn**). These sequences are similar to the example in Section 3.5.20.2.5.

The standard RUD sequence for a single-packet read using SES medium and small headers on a best-effort network is shown in Figure 3-102. This shows a read with return data larger than *Max_ACK_Data_Size* so the return direction is used to send return data.

The sequence in the figure proceeds in the same manner as Figure 3-101:

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

1. The PDC is idle with all previous packets successfully delivered, *PSN* = *ACK_PSN* = *CLEAR_PSN* = 332.
2. SES receives the read request from the provider and generates an SES Request without a valid message ID.
3. The initiator transmits the SES read to the target as a PDS Request with PSN = 333.
4. The target transmits an ACK for PSN = 333.
    a. The ACK carries the SES Response
5. The target SES processes the read and generates a request to PDS to send a packet back to FEP-A.
    a. This request carries an SES UET_HDR_RESPONSE_DATA header and includes the data; the SES header carries the original PSN in the **ses.original_request_psn** field (*orig_psn* = 333 in this example) assigned by FEP-A to associate the data with the original request (since a small SES header doesn't have an **ses.message_id** field).
6. The target transmits a PDS Request using PSN = 48.
    a. This packet carries the requested read data and uses the return direction of the same PDC on which the associated request arrived; see Section 3.5.8.1.
    b. The SES Response with data header carries the original PSN in the **ses.original_request_psn** field (*orig_psn*—in this example 333) to associate the data with the original request.
7. After receiving PSN = 48, FEP-A transmits an ACK for PSN = 48.
    a. The read response transaction operates in the same manner as other transactions; when successful, the ACKs do not require a clear.

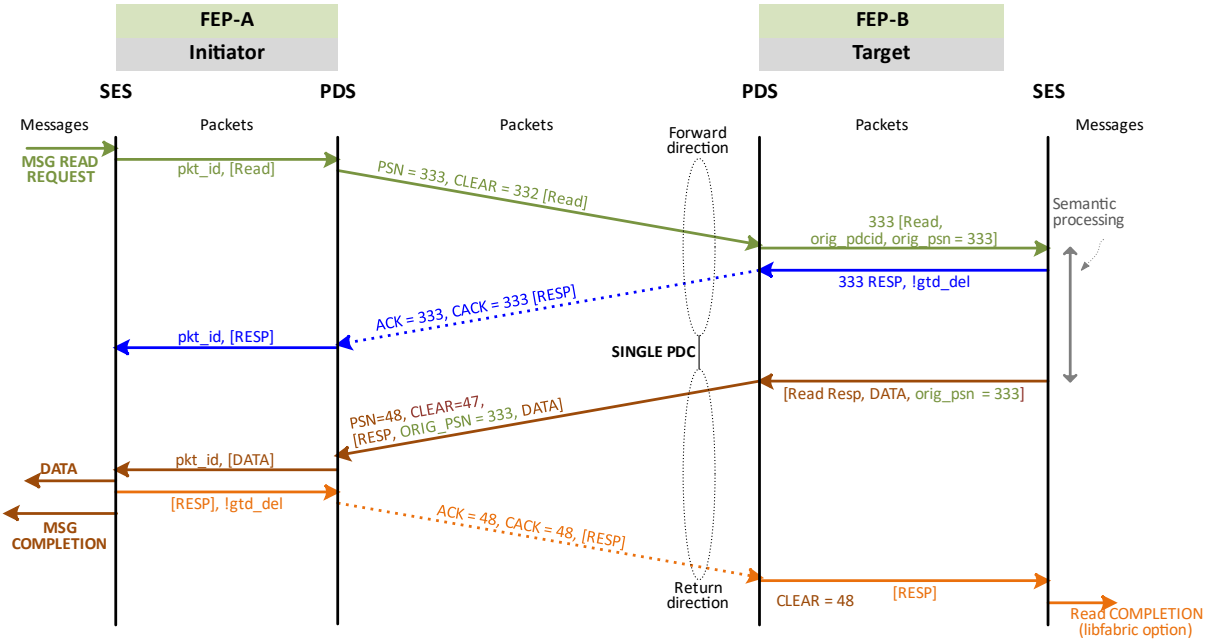


Figure 3-102 - Standard RUD Sequence for Single-Packet Read–Medium & Small SES header

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

3.5.20.2.7 Single-Packet Read Sequence Optimization

Read responses that are *Max_ACK_Data_Size* or smaller use the PDC ACK to carry the SES Response with data. This is efficient for small reads and fetching atomics. The sequence is the same as the single-packet send sequence with guaranteed delivery as described in Section 3.5.20.2.2.

3.5.20.2.8 Multi-Packet Read Sequence

Multi-packet reads (when an application requests a read larger than one MTU) are similar to single-packet reads, repeated once for each packet.

Figure 3-103 shows an example of a RUD sequence for a multi-packet read. The sequence in Figure 3-103 proceeds as follows:

1. PDC is idle with all previous packets successfully delivered, *PSN* = *ACK_PSN* = *CLEAR_PSN* = 332.
2. FEP-A SES issues three packets to PDS, where two packets are a 1 MTU read and the third packet is a small read (less than *Max_ACK_Data_Size*).
    a. These packets are for a single SES message and carry the same message identifier (= 88).
3. FEP-B generates an ACK for the first two packets arriving at the target. In this example, cumulative ACKs are used (i.e., there is no packet with ACK = 333, rather the PSN = 333 acknowledgement is determined at FEP-A using *CACK_PSN* = 334).
    a. **pds.cack_psn** = 334, **pds.ack_psn_offset** = 0.
    b. This ACK is not guaranteed delivery as it carries a default SES response.
4. FEP-B generates a second ACK for the third packet with **pds.psn** = 335.
    a. The SES response is set to RC_OK indicating the read request was accepted and read responses will be provided on the return direction. This packet is also not guaranteed delivery so CACK is advanced.
    b. **pds.cack_psn** = 335, **pds.ack_psn_offset** = 0.
5. FEP-A issues a PDS write request. This is a new message and uses a different message (= 879).
    a. This request packet encodes *CLEAR_PSN* = 335 in the **pds.clear_psn_offset** and **pds.psn** fields using **pds.clear_psn_offset** = -1 and **pds.psn** = 336.
6. FEP-B SES generates responses to the first two read requests and passes these to the PDS.
    a. These are carried in PDS Request packets using **pds.psn** = 48 and **pds.psn** = 49 in the return direction.
7. FEP-A PDS sends a cumulative ACK for the two SES responses.
    a. **pds.cack_psn** = 49 and **pds.ack_psn_offset** = 0.
8. FEP-B PDS acknowledges the write request message 879 using **pds.cack_psn** = 336 and **pds.ack_psn_offset** = 0.
9. FEP-B SES generates the return data for the third read packet and PDS sends a request with **pds.psn** = 50 and **pds.clear_psn_offset** = -3. The **pds.clear_psn_offset** is -3 because the ACK for PSN = 48 and PSN = 49 has not yet arrived at FEP-B.
10. FEP-A PDS receives the packet with pds.psn = 50 and generates an ACK that includes clearing PSN 48 through 50.
    a. **pds.cack_psn** = 50 and **pds.ack_psn_offset** = 0.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

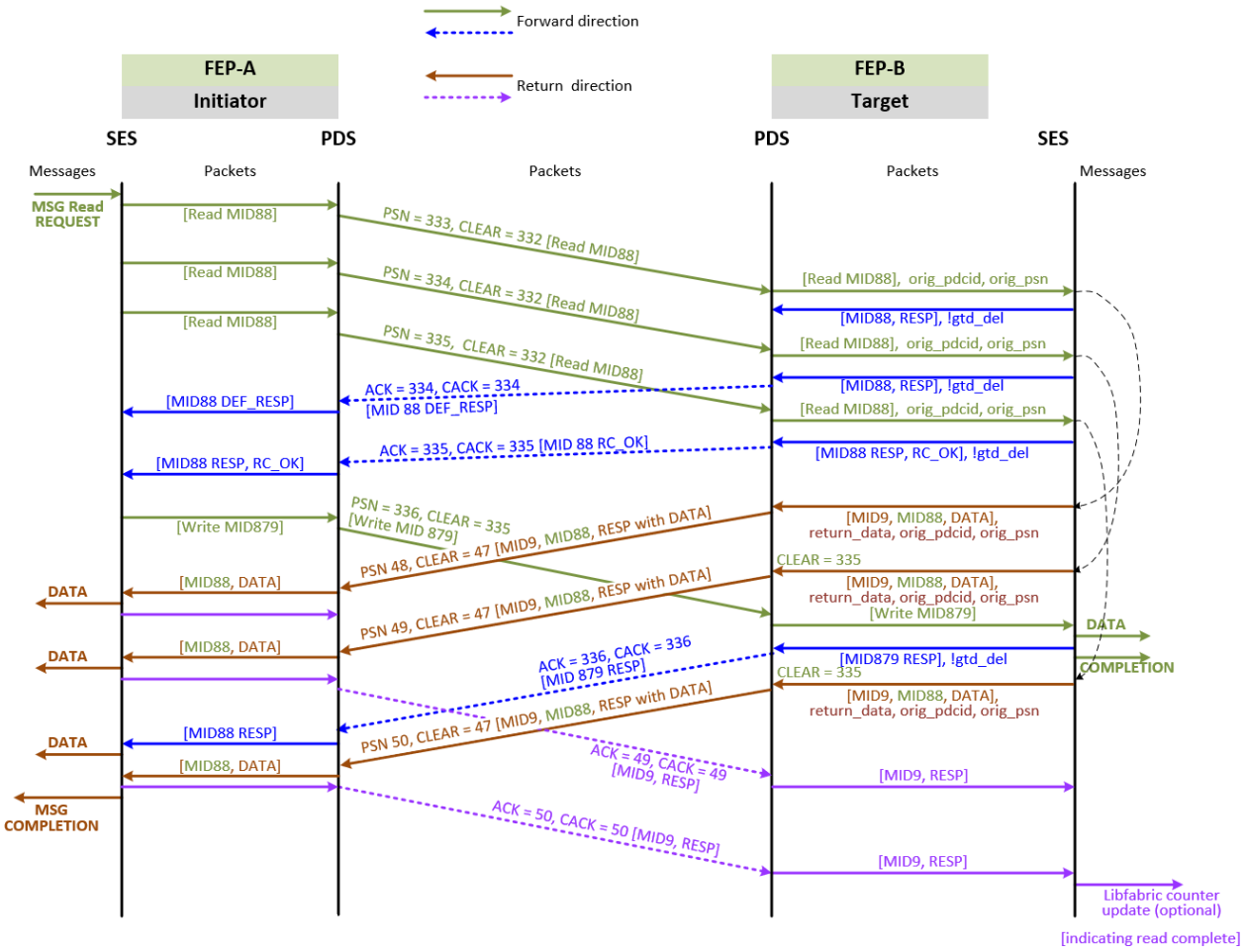


Figure 3-103 - Standard RUD Sequence for Multi-Packet Read

3.5.20.3 Negative Acknowledgements, Packet Drops, and Retries

This section specifies the RUD sequences for packet drops and other cases that cause PDS retransmissions. Negative acknowledgements (NACKs) can occur at PDS or SES. This section covers PDS NACKs. SES NACKs are not visible to PDS. SES NACKs appear as SES guaranteed delivery responses that are carried, but not interpreted, by PDS. Retry attempts at SES appear as new requests to PDS.

3.5.20.3.1 Single-Packet PDS NACK Sequence

Figure 3-104 shows the PDS NACK sequence for a RUD Send of a single-packet message. The sequence in Figure 3-104 proceeds as follows:

1. The PDC is idle with all previous packets successfully delivered, *PSN* = *ACK_PSN* = *CLEAR_PSN* = 332.
2. FEP-A generates a packet with MID = 14 to transmit, and PDS transmits the packet with PSN = 333.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

3. When a PDS Request with PSN = 333 is received at the target PDS, an out-of-packet buffer condition is encountered.
4. PDS at the FEP-B generates a NACK 333 packet and sends it to FEP-A.
    a. The PDS Request packet is dropped at FEP-B.
5. PDS at FEP-A is responsible for retransmitting PSN = 333.
6. Based on the NACK code (i.e., **pds.nack_code** field), the FAP-A waits for the configured time and retransmits PSN = 333
    a. If *Max_NACK_Retx_Cnt* is exceeded, PDS declares an error and SES completes the packet and message in error.
    b. PDS at the initiator is responsible for retransmitting PSN = 333 even if the PDS NACK is lost; in this case an *RTO_TIMER* expiration will trigger retransmission up to *Max_RTO_Retx_Cnt* times.

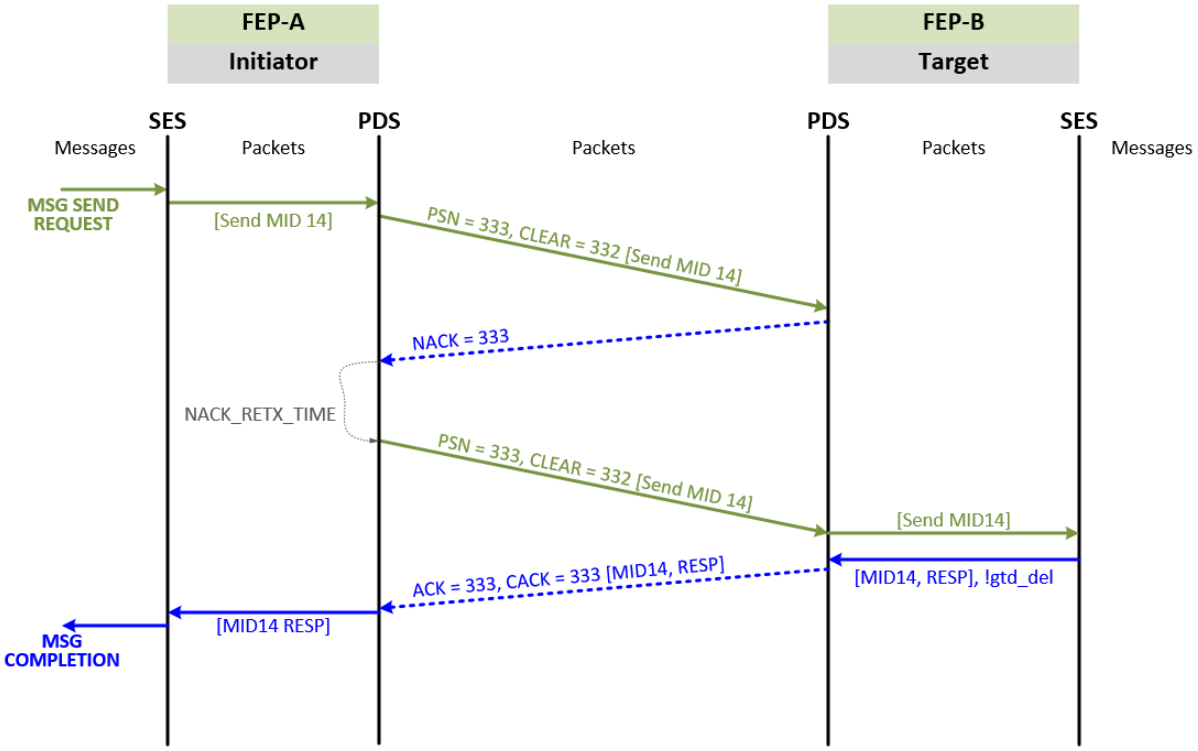


Figure 3-104 - PDS NACK Sequence for RUD Send of Single-Packet Message

The PDS NACK sequence for a RUD Send of a multi-packet message is the same as for a single-packet message. Similarly, the PDS NACK sequence for RUD Reads is the same as for RUD Sends.

3.5.20.3.2 NACK Sequence for Lost PDS Requests and ACKs

PDS at the initiator is responsible for retrying transmissions when packets are dropped in the network, including in the following situations:

- A PDS request transmitted by initiator is dropped—does not arrive at destination.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

- A PDS ACK transmitted by the target is dropped—does not arrive at source.
    - PDS determines ACK state based on *CACK_PSN*, including both successful ACKs (if *CACK_PSN* is larger than a missing ACK PSN) and guaranteed delivery ACKs (if the **pds.sack_bitmap** field indicates packet was received but *CACK_PSN* does not).

Figure 3-105 shows the PDS Request dropped sequence for a RUD Send of a single-packet message. The source is responsible for detecting the loss and retrying the transmission. In the example shown in Figure 3-105, the source retries the transmission after a timeout.

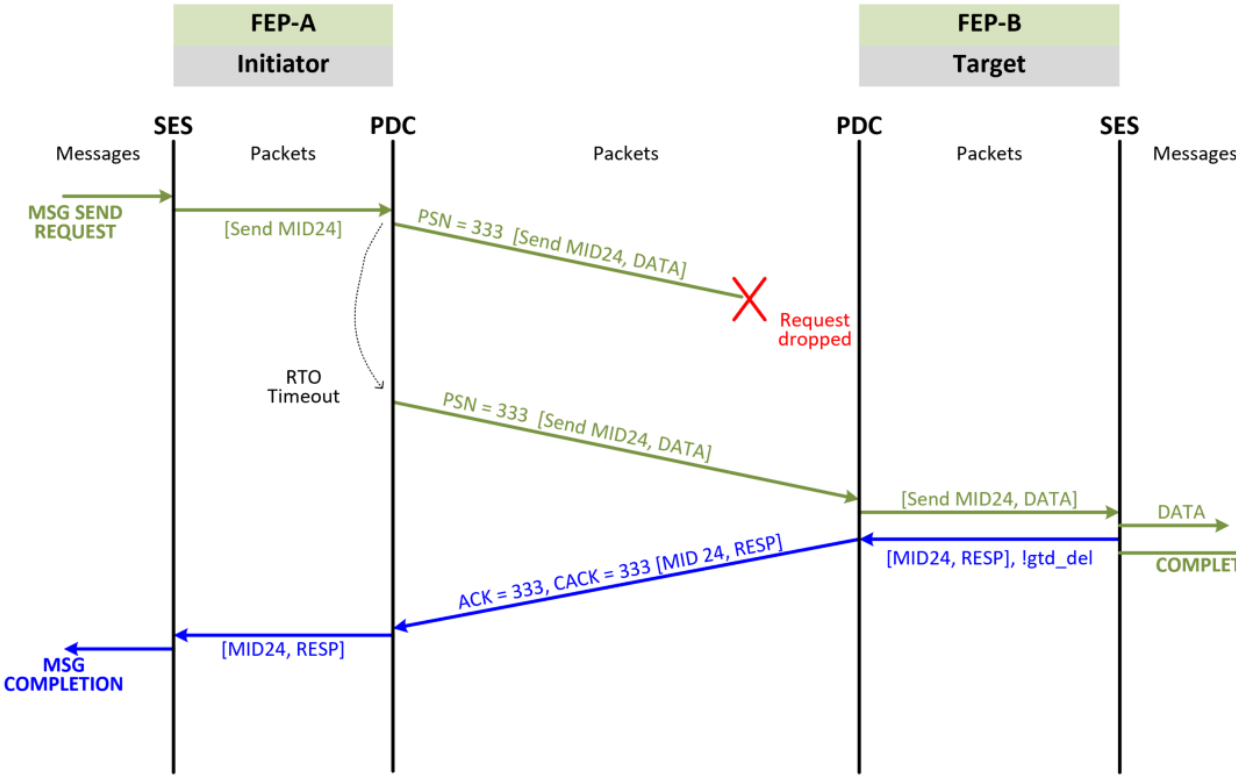


Figure 3-105 - PDS Request Dropped Sequence for RUD Send of Single-Packet Message

Figure 3-106 shows the ACK dropped sequence for a RUD Send of a single-packet message. The sequence in Figure 3-106 proceeds as follow:

1. The PDC is idle with all previous packets successfully delivered, *PSN* = *ACK_PSN* = *CLEAR_PSN* = 332.
2. FEP-A transmits a request with PSN = 333.
3. FEP-B accepts PSN = 333, marks the PSN as received, and generates an ACK with *ACK_PSN* = 333.
4. In the example, the ACK packet for PSN = 333 transmitted by FEP-B is dropped in network.
5. FEP-A retransmits the request with PSN = 333.
    a. In this example FEP-A will timeout, since the ACK does not arrive; there are multiple other methods for determining if a packet was lost (e.g., NACKs, SACK bitmap, etc.).

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

6. FEP-B processes the retransmitted packet as follows:
    a. FEP-B checks if the PSN is outside the valid PSN range (based on *MP_RANGE*)—which in this example is not true; if TRUE, drop the packet with no ACK.
    b. FEP-B checks if PSN = 333 is not marked as received, so it can accept and pass the request to SES—in this example, the request has already been received.
        i. The packet is marked as a retransmission (**pds.flags.retx** is set), which is used to trigger an ACK. However, the **pds.flags.retx** flag does not indicate whether a previous copy of the packet arrived.
    c. PSN = 333 is in the expected range and is marked as received.
        i. FEP-B determines if there is a guaranteed delivery SES Response state entry for PSN = 333.
        ii. If there is a guaranteed delivery SES Response entry for PSN = 333:
            1. FEP-B retransmits the ACK to the initiator with stored, guaranteed delivery SES Response.
            2. FEP-B maintains the SES Response, waiting for clear before freeing.
        iii. If there is not a guaranteed delivery SES response entry for PSN = 333:
            1. FEP-B retransmits the ACK with the default response; see Section 3.5.17.

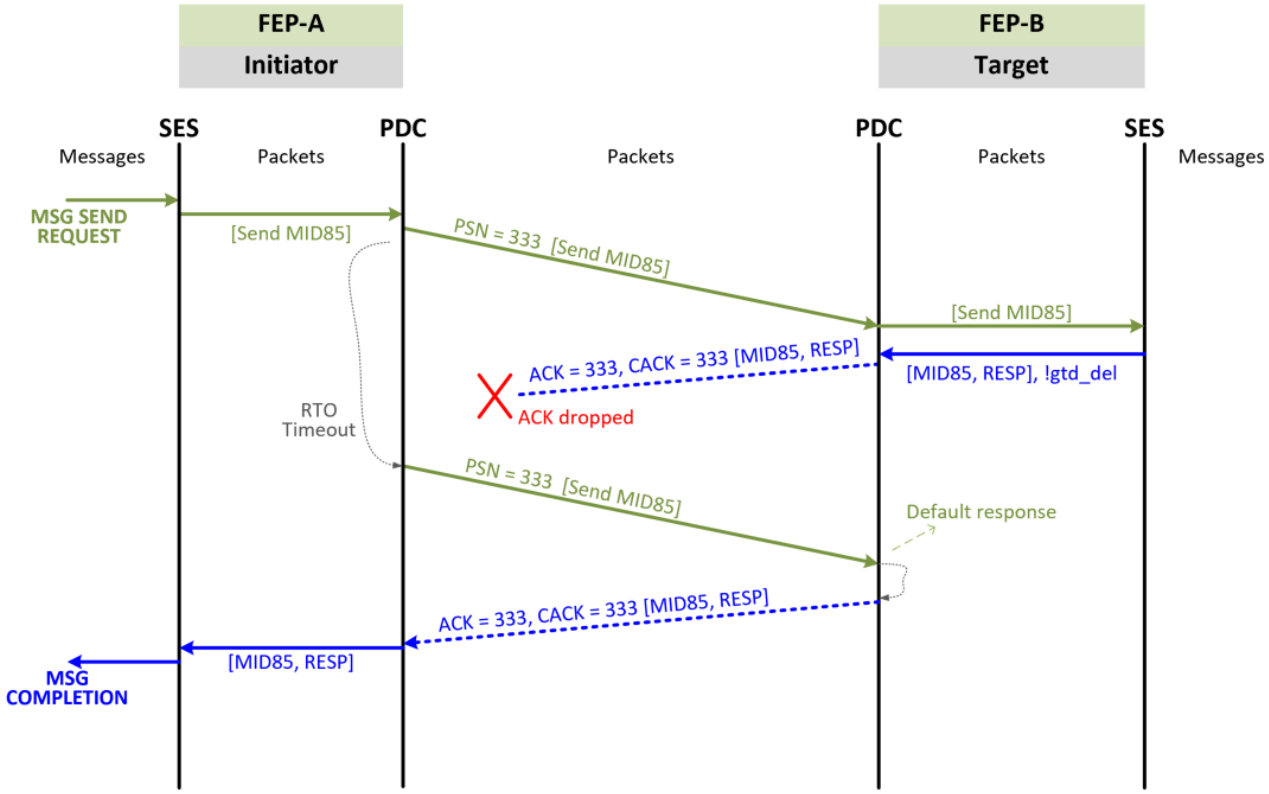


Figure 3-106 - PDS ACK Dropped Sequence for RUD Send of Single-Packet Message

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

3.5.21 Reliable Ordered Delivery

A ROD PDC is a reliable PDC established between a single initiating FEP and a single target FEP that guarantees packets are delivered at most once. PDCs are ephemeral: established and closed on demand. Packets from multiple PIDs and/or send queues may share a PDC. All packets from a single message use a single ROD PDC. Messages MUST NOT be interleaved on a ROD PDC.

3.5.21.1 ROD Services

The requirements in this section are for ROD PDS using a GoBackN retransmission protocol for all packets conveyed on the PDC. With the GoBackN retransmission protocol, UET ROD mode MUST operate on a single network path such that packets always arrive at the target in the order in which the packets were transmitted by the source. Packets within a message MUST be transmitted in order. The source MUST transmit the packets in the order that packets were posted to a common send queue. There are no ordering requirements across messages in separate send queues.

ROD implementations use the following requirements for PDS requests (both forward and return direction):

1. ROD MUST support GoBackN retransmission.
2. ROD MUST guarantee that each packet is delivered to the target SES once and only once in the order of arrival.
3. ROD ACKs MUST be transmitted in the same order as the packets they ACK.
4. ROD Read responses MUST be transmitted on the wire in the same order os the corresponding requests.
5. If a packet arrives in order and a NACK is generated for a reason other than an out-of-order PSN, the NACK carries the **pds.nack_code** field representing the event (e.g., UET_NO_PKT_BUFF or UET_TRIMMED).
    a. The NACK payload (**pds.payload** field) carries the *Expected_PSN* except in the case of **pds.nack_code** = UET_NEW_START_PSN.
    b. The **pds.nack_psn** field is set to the PSN of the packet being NACKed.
    c. This applies in both forward and return directions.
6. If a packet is received at the destination with an out-of-order PSN (PSN != last received PSN+1), the packet MUST be dropped and a NACK generated.
    a. **pds.nack_code** = UET_ROD_OOO.
    b. **pds.nack_psn** = the PSN of the out-of-order packet.
    c. **pds.payload** carries the *Expected_PSN*.
    d. This applies in the forward and return directions.
7. If a packet is received in order, the packet is accepted.
8. When a NACK is received at the source, the source determines if the NACK indicates a new packet loss or if the NACK is related to a packet loss that already triggered GoBackN processing. If new packet loss occurred, the source MUST stop transmitting higher PSN packets on that PDC and apply GoBackN retransmission. The source MUST retransmit all packets starting from the *Expected_PSN* in the NACK payload. The When a NACK arrives at a source PDC that is not

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

actively processing another NACK, the PDC stops transmitting and triggers GoBackN processing starting at PSN = **pds.payload** = *Expected_PSN*.

    a. Actively processing another NACK refers to a previous NACK arrived, and an ACK has not yet arrived after that NACK
    b. **pds.nack_psn** indicates the PSN that arrived out of order at the destination.

9. If an ACK arrives while processing a GoBackN event, the GoBackN processing for the NACK that triggered the GoBackN event is complete. Transmission continues per normal ROD processing.
10. If additional NACKs arrive prior to an ACK and those NACKs have **pds.nack_psn** > *Expected_PSN*, **pds.payload** = *Expected_PSN* and **pds.retx** = 0, these NACKs are part of the current GoBackN processing and should not be used to trigger a new GoBackN event.
11. If additional NACKs arrive prior to an ACK and those NACKs have **pds.payload** > *Expected_PSN* and **pds.retx** = 0, these NACKs indicate new packet loss and trigger a new GoBackN event.
    a. Retransmission starts at *Expected_PSN* = **pds.payload**
12. If additional NACKs arrive prior to an ACK and those NACKs have **pds.retx** = 1 then one or more of the retransmitted packets was lost
    a. If this is the first NACK with **pds.retx** = 1, trigger a new GoBackN event with retransmission starting at *Expected_PSN* = **pds.payload**
    b. If this is not the first NACK with **pds.retx** = 1, a new GoBackN event with retransmission starting at *Expected_PSN* = **pds.payload** may be triggered if **pds.payload** > *Expected_PSN*
    c. Else if not first NACK with with **pds.retx** = 1 and **pds.payload** = *Expected_PSN*, this NACK is part of te current GoBackN processing and should not trigger a new GoBackN event.
13. Handling of scenarios where a packet requires transmission more than three times (i.e., initial transmit, retransmit after a NACK with **pds.retx** = 0 and retransmit again after a NACK with **pds.retx** = 1) is implementation specific. One option is to fallback to RTO. Alternatively, additional state can be tracked to identify this scenario.

This processing is illustrated in Section 3.5.21.3.

ROD implementations use the following requirements for PDS ACKs (both forward and return direction):

- All ACKs for a ROD PDC MUST be sent in order using the entropy value of the request.
- When loss of an ACK with a non-guaranteed delivery SES Response is detected based on CACK_PSN being equal to or higher than the PSN associated with the missing ACK, the PDC MUST infer a UET_DEFAULT_RESPONSE-based *CACK_PSN* from the next arriving packet.
    1. If the ACK for the last packet is lost and there is insufficient information to infer the associated SES Response is non-guaranteed, either an *RTO_TIMER* will expire and an ACK Request CP is sent, or the original PDS Request is retransmitted.
- When loss of an ACK with a guaranteed delivery SES Response is detected, one of three actions MUST be taken:
    1. The source generates an ACK Request CP.
        - ROD PDCs SHOULD generate an ACK Request CP, unless the original packet was a read request; refer to Section 3.5.16.2.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

2. If the missing ACK is associated with a read request (i.e., carrying read response data), the source retransmits the original PDS request.
    - This will arrive out of order at the destination; ROD PDCs MUST support retransmitting a guaranteed delivery ACK when a duplicate PSN is received out of order.
3. The source may fall back to GoBackN and retransmit all the packets starting from the missing PSN; this is the least efficient option.

When using action 1 or 2 in the bullet list above, packets with a higher PSN may continue to be transmitted while the ACK Request CP or retransmitted PDS Request is outstanding. Therefore, a destination ROD PDC MUST process duplicate packets even though the PSN is out-of-order.

3.5.21.2 Standard Sequences

This section specifies the normal processing sequences (i.e., no PDS error events) for ROD.

The standard sequences for the ROD normal process are the same as the standard sequences for RUD, specified in Section 3.5.20.1 — except the packets always arrive in order at the destination. ROD PDCs track the next *Expected_PSN*, which is the last received PSN + 1.

3.5.21.3 Error Sequences

This section specifies the ROD sequences for packet drops and other error events using the GoBackN protocol. Figure 3-107 illustrates the sequence when a request packet is dropped.

- PDC is idle with all previous packets successfully delivered, *PSN* = *ACK_PSN* = *CLEAR_PSN* = 332.
- Packets with PSN = 333, 334, 335 are sent from FEP-A to FEP-B.
    - PSN = 333 is dropped in the network.
- FEP-B receives a packet with **pds.psn** = 334, which does not match the *Expected_PSN* of 333.
    - PSN = 334 is dropped at FEP-B, and a NACK is sent with **pds.nack_code** = UET_ROD_OOO, **pds.nack_psn** = 334 and **pds.payload** = *Expected_PSN* = 333.
    - All packets received on the PDC are dropped at FEP-B until a packet with PSN = 333 is received.
- FEP-B receives a packet with **pds.psn** = 335, which does not match the *Expected_PSN* of 333.
    - PSN = 335 is dropped at FEP-B, and a NACK is sent with **pds.nack_code** = UET_ROD_OOO, **pds.nack_psn** = 335 and **pds.payload** = *Expected_PSN* = 333.
- FEP-A receives a NACK with **pds.nack_psn** = 334, **pds.nack_code** = UET_ROD_OOO with **pds.payload** = 333, which triggers the retransmission of PSN = 333, 334, 335.
    - pds.retx is set to 1 for all three packets, indicating the packets are retransmits.
- FEP-A receives a NACK with **pds.nack_psn** = 335, **pds.nack_code** = UET_ROD_OOO with **pds.payload** = 333.
    - FEP-A does not trigger retransmission as it is already processing a GoBackN event starting at PSN = 333.
- The destination accepts all three packets and generates ACKs accordingly.
    - This example shows PSN = 334 and PSN = 335 using a cumulative ACK.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

- Because *CACK_PSN* = maximum PSN transmitted, there are no clears required. That is, if **pds.ack_psn_offset** = 0 then all PSN are cleared.

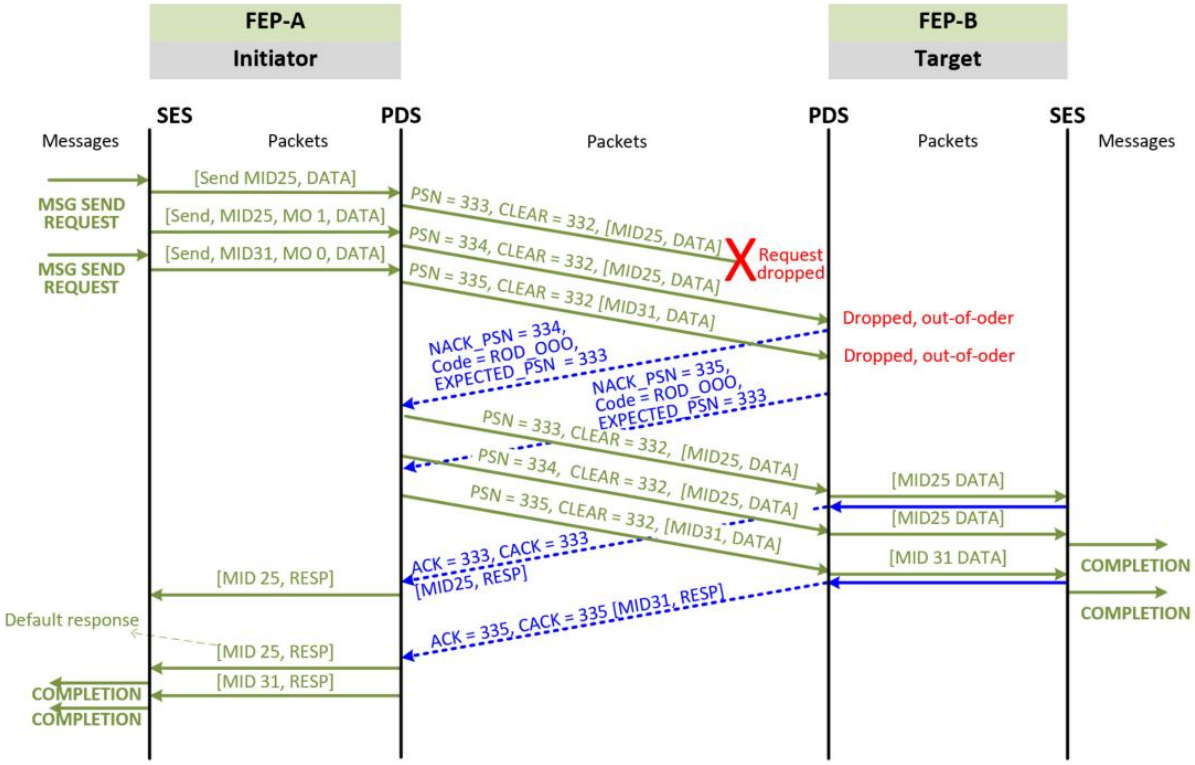


Figure 3-107 - ROD Send Sequence for Dropped PDS Request Packet

If an ACK is dropped in transit from the destination to the source, three scenarios are possible:

1. Cumulative ACKs are in use and the dropped ACK was not guaranteed delivery.
    - In this case, the next ACK to arrive will carry a **pds.sack_bitmap** field and **pds.cack_psn** field indicating the packet did arrive at the destination, and there is no stateful response.
    - No further action is needed, as the source can infer the needed information.
2. Cumulative ACKs are in use and the dropped ACK was guaranteed delivery.
    - In this case, the next ACK to arrive will carry a **pds.sack_bitmap** field indicating the packet arrived at the destination; the **pds.cack_psn** field allows the source to infer that the ACK requires guaranteed delivery.
        - In the ROD case, there is no need to track the highest acknowledged PSN; since ACKs arrive in order, the source can immediately infer the ACK was lost—then check *CACK_PSN* to determine if the lost ACK was guaranteed delivery or not.
    - The source should generate an ACK Request CP if required, or it may use GoBackN.
3. Cumulative ACKs are not in use and an ACK was dropped.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

- In this case, when the next ACK is received with the next *Expected_PSN*+1, the source will generate an ACK Request CP.

The sequence for cumulative ACKs with a lost ACK that has guaranteed delivery — item 2 in the list above — is shown in Figure 3-108 and described as follows:

1. PDC is idle with all previous packets successfully delivered, *PSN* = *ACK_PSN* = *CLEAR_PSN* = 332.
2. FEP-A sends two packets to FEP-B, PSN = 333 and 334.
3. Both arrive and are processed.
    a. The PSN = 333 SES Response requires guaranteed delivery; PSN = 334 does not.
4. ACK for PSN = 333 is dropped.
5. When ACK for a packet with PSN = 334 is received, FEP-A determines that the 333 ACK was lost and that it was guaranteed delivery.
    a. The **pds.sack_bitmap** field indicates PSN = 333 arrived.
    b. *CACK_PSN* is set to 332; because **pds.sack_bitmap** shows PSN = 333 arrived, the *CACK_PSN* indicates 333 requires a clear.
6. FEP-A generates an ACK Request CP to request an ACK for PSN = 333.
7. FEP-B receives the ACK Request CP and generates the ACK for PSN = 333.
    a. This carries the original SES Response, which was saved since it is guaranteed delivery.
8. When ACK with PSN = 333 is received, it is processed as well as any responses that were held to be passed to SES in order.
9. FEP-A then updates the clear state to PSN = 334; this is used in the next message.
10. When the *CLEAR_PSN* = 334 reaches FEP-B, *CACK_PSN* is updated — in this case to 335, since PSN = 335 is not guaranteed delivery.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

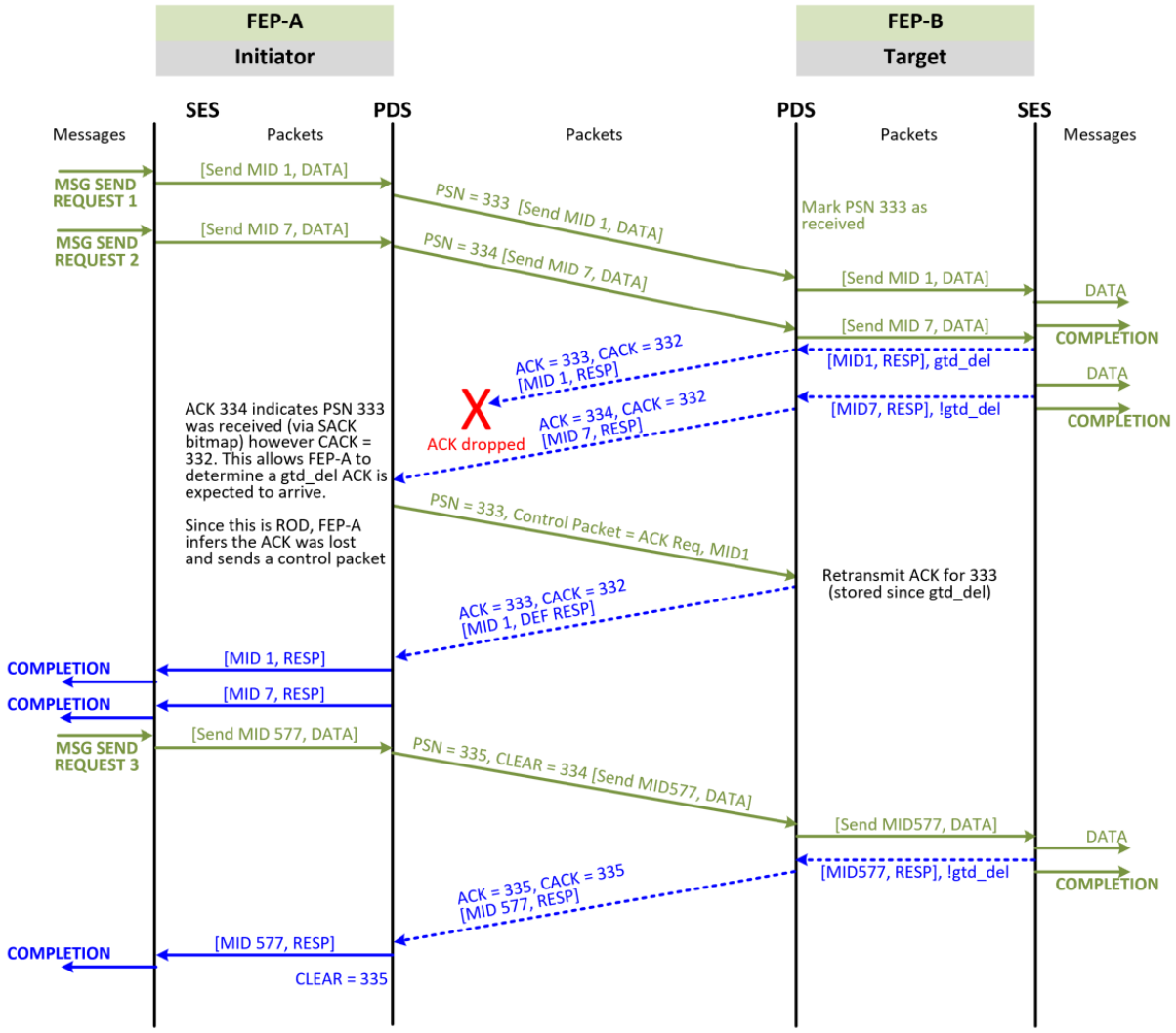


Figure 3-108 - ROD Send Sequence for Dropped ACK Packet Using ACK Request

A PDS NACK event is handled similarly to a dropped request packet. If a PDS NACK for PSN = XYZ is generated by the destination:

- The destination drops all subsequent packets with PSNs beyond XYZ, and
- The source retransmits the packets starting at PSN = XYZ.

3.5.22 Reliable Ordered Delivery for Local Networks

A RODL PDC is a reliable PDC established between a pair of FEPs that guarantees packets are delivered at most once. RODL is a variation of ROD that is optimized for scale-up networks. RODL is different from ROD in the following notable ways:

1. RODL runs over a bidirectional PDC.
    a. Targets may initiate send, write, or atomic requests on the PDC—see Section 3.5.22.1.2.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

2. RODL can be implemented without PDS retransmission.
    a. This saves state with a trade-off of packet loss leading to PDC closure
3. RODL allows sending the CACK_PSN for the reverse path in lieu of ACK packets.
    a. Since the PDC is bidirectional, PDS Requests flowing in one direction can carry the CACK_PSN for the other direction.
    b. Alternatively, normal ACK packets can be used.

The RODL header format when running over the UFH_16 header is illustrated in Figure 3-68. See Section 3.5.25.9 for a description of the RODL header mapped over the UFH_16 header. RODL also runs over UDP/IP/ETH or directly over UFH_32, as illustrated in Section 3.5.25.8.

When using RODL over the UFH_16 header, the **pds.flags** field is reduced to 3 bits and the **pds.psn_state** is reduced to 24 bits. The header is compressed in the following ways:

1. The 32-bit PSN is carried implicitly in the CRC once the PDC is established.
    a. This saves 32 bits on the wire by taking advantage of the in-order arrival of packets.
    b. During PDC establishment, the **pds.psn** field is carried in the header.
        i. RODL carries the 24 least significant bits of the full 32-bit PSN in the **pds.psn_state.psn_lsb** field.
        ii. All start PSN values have the upper 8 bits set to zero.
2. The **pds.pdc_info** field used during PDC establishment is reduced to 2 bits such that **pds.pdc_info** = [rsvd, use_rsvd_pdc].
3. The **pds.spdcid** and **pds.dpdcid** field sizes are reduced to 12 bits.
4. A 24-bit **pds.psn_state** field is defined.
    a. **pds.psn_state** carries the request PSN during PDC establishment (**pds.flags.syn** = 1).
    b. Once the PDC is established (**flags.syn** = 0), **pds.psn_state** carries {**pds.psn_state.clear_psn_offset, pds.psn_state.rev_cack_psn_lsb**}; each field is 12 bits.
    c. The PDC uses full 32-bit CLEAR_PSN and CACK_PSN, which are maintained on both sides of the PDC. The lsbs are carried in the packets. This limits the number of outstanding packets on a single RODL PDC.
5. RODL does not support NSCC or packet trimming.
    a. RODL supports ACK and NACK.
    b. RODL does not support ACK_CC, ACK_CCX, and NACK_CCX.

3.5.22.1 RODL Services

RODL uses GoBackN retransmission such as ROD when retransmission is supported. RODL can operate without retransmission. If a packet is lost on a RODL PDC that does not support retransmission, the PDC is closed in error. Refer to Section 3.5.7.3.

3.5.22.1.1 No PDS retransmission

PDS is extended to support an option to exclude PDS retransmission on RODL PDCs. When retransmission is not supported, the loss of a single packet is fatal to the PDC and the PDC MUST be closed in error.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

If retransmission is disabled and a RODL packet is determined to be lost or corrupted (e.g., CRC invalid), the PDC is closed in error and all outstanding messages after the lost packet are dropped and completed in error. PDS returns a NACK PSN for the missing PSN and every packet that arrives thereafter for all packets excluding a Close Command control packet. The **pds.nack_code** is set to UET_UNEXP_EVENT for request packets. The CACK PSN does not advance beyond the missing PSN minus one.

When a lost packet event occurs on a PDC without retransmission, the initiator sends a Close Command control packet. The target responds to the Close Command CP with an ACK packet and closes the PDC. The target accepts the Close Command CP despite the PSN being out of order. Both initiator and target ignore maximum PSN range if necessary for the Close Command CP under this error case.

3.5.22.1.2 Bidirectional PDCs and Reverse ACKs

Bidirectional PDCs reduce the amount of state required for two FEPs to exchange transactions and enable the use of reverse path ACKs. RODL PDCs are defined to be bidirectional.

Bidirectional PDCs allow the target to generate semantic requests in the return direction. The basic sequence for establishing and closing of the PDC remains the same—the role of initiator is assigned to the FEP that initiates the PDC. Only the initiator closes a PDC. The target may request the initiator to close the PDC.

The PDC close flow is modified at the target. Before the target requests that the initiator start the close sequence, it MUST wait until all transactions initiated by the target FEP on that PDC are completed, including clear processing, before sending the close request. When the initiator sends a close command, the target MUST complete all active transactions on the PDC before acknowledging the close command.

The packet flow for a Unified Local profile bidirectional PDC is illustrated in Figure 3-109. The PDC is configured for lossless operation (e.g., using CBFC or PFC and LLR) using RODL. An important factor in lossless networks is deadlock avoidance. Responses MUST use a different traffic class than requests. ACK packets are used to carry response data for read and fetching atomic operations.

For example, in Figure 3-109, the red and blue lines are PDS requests that are mapped to the same traffic class. *Max_ACK_Data_Size* MUST be set to match the Payload MTU, so all response data is carried in acknowledgements. ACKs and NACKs are mapped to a second traffic class, unique from PDS requests.

RODL is designed to run over a lossless fabric and uses the traffic class mapping defined for UET over lossless. This means PDS requests and control packets use a different traffic class than ACKs and NACKs.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

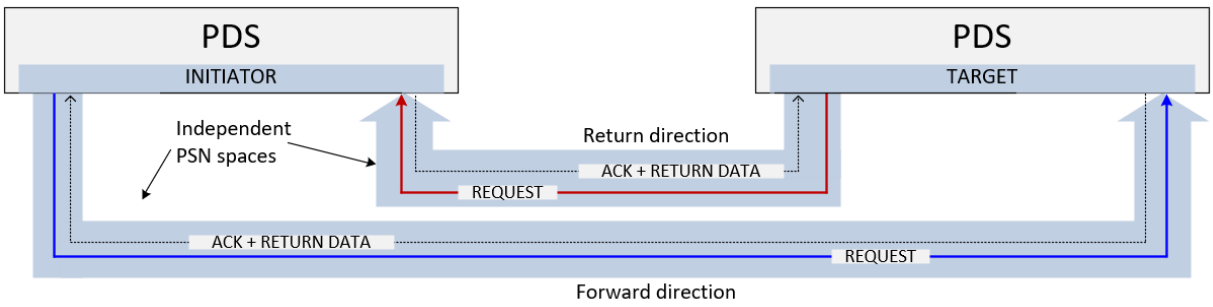


Figure 3-109 - Bidirectional PDC

If two FEPs simultaneously create PDCs between each other, both PDCs may remain open.

3.5.22.1.3 ACK and NACK Generation

ACK packets MUST be supported, as they carry stateful responses including read data, fetching atomic data, and error state. The generation of NACK packets MUST be supported. Accepting and processing received ACK and NACK packets MUST be supported.

With bidirectional PDCs, reverse acknowledgements are included. The PDS request may carry the lower bits of the cumulative acknowledgment PSN (*CACK PSN*) for the other direction in the **pds.rev_cack_psn_lsb** field.

When write and non-fetching atomic operations are being performed without error on a bidirectional PDC, all SES responses are UET_DEFAULT_RESPONSE with RC_OK return codes. These responses can be messaged to the source with **pds.rev_cack_psn_lsb**, removing the need for ACK packets while traffic is active in both directions.

Stateful responses MUST be transported in an ACK packet.

Both initiator and target maintain a full 32-bit *CACK_PSN* as in ROD with the **pds.rev_cack_psn_lsb** field carrying the lower bits of the locally maintained *CACK_PSN*. The PDS receiving a RODL packet monitors when the *CACK_PSN* field wraps and updates the local *CACK_PSN* accordingly by incrementing the upper bits.

Similarly, both initiator and target MUST maintain full 32-bit PSN and *CLEAR_PSN*.

If there is no pending SES request or response and a PDC needs to update either *CLEAR_PSN* or *CACK_PSN*, an implementation MUST send an ACK or a control packet to exchange this state. A subset of the rules used for ROD PDCs applies for generating ACK packets when using ACK coalescing as defined in Section 3.5.12.4.

3.5.22.2 Standard Sequences

The standard sequences for RODL are the same as for ROD. See Section 3.5.21.2.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

3.5.22.3 Error Sequences

RODL error sequences are the same as ROD when retransmission is enabled. See Section 3.5.21.3. When retransmission is not enabled, the dropped request results in a PDC fatal error and the PDC is closed.

RODL lost ACKs are handled in the same manner as ROD lost ACKs.

3.5.23 RUDI

RUDI sequences are specified for RMA Writes and RMA Reads. RUDI provides reliable delivery for idempotent operations. It does not use PDCs. Each packet is handled individually without maintaining sequence numbers.

The **pds.pkt_id** field in RUDI PDS headers is a locally assigned number that is echoed back in the RUDI response to allow the response to be associated with the request.

RUDI maintains no PDS or SES state at the target; all state resides at the initiator. As a result, no clear-protocol exchanges are required to free target-side state. All reliability is provided at the initiator, and SES responses are not retained for retransmission.

3.5.23.1 RUDI Services

The primary differences between the RUDI Write sequence and the RUD Write sequence are:

1. The RUDI delivery mode guarantees that each packet is delivered to SES at least once.
2. RUDI is only be used for idempotent operations.
3. The RUDI source assigns a unique **pds.pkt_id** to each transmitted packet.
4. PDS at the target waits for semantic processing to complete before transmitting the response for each packet.
    a. There is no message completion at the target.
    b. RUDI Requests are acknowledged by RUDI Response packets.
5. If a RUDI packet is retransmitted, the **pds.flags.retx** field is set.
6. If a received RUDI Request is ECN marked, the **pds.flags.m** field is set on the corresponding RUDI Response.
7. Each RUDI Request packet results in one and only one RUDI Response packet.
8. Each RUDI Response uses the **pds.pkt_id** from the corresponding RUDI Request.
9. PDS NACK event processing for RUDI includes the following:
    a. The destination transmits a NACK if a trimmed RUDI Request is received.
    b. The destination SHOULD transmit a NACK if a RUDI packet is dropped (e.g., due to limited resources).
    c. The source processes NACKs for RUDI Requests and retransmits the request.
    d. The source receives trimmed RUDI Responses and retransmits the request.
    e. RUDI Response packets are not NACK'd.
10. Loss recovery is provided by the initiator.
    a. An *RTO_TIMER* per packet is required.
11. RUDI does not use UET congestion control.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

3.5.23.2 Standard Sequences

This section specifies the normal processing sequences for RUDI. Figure 3-110 shows the standard RUDI RMA Write sequence for a multi-packet message.

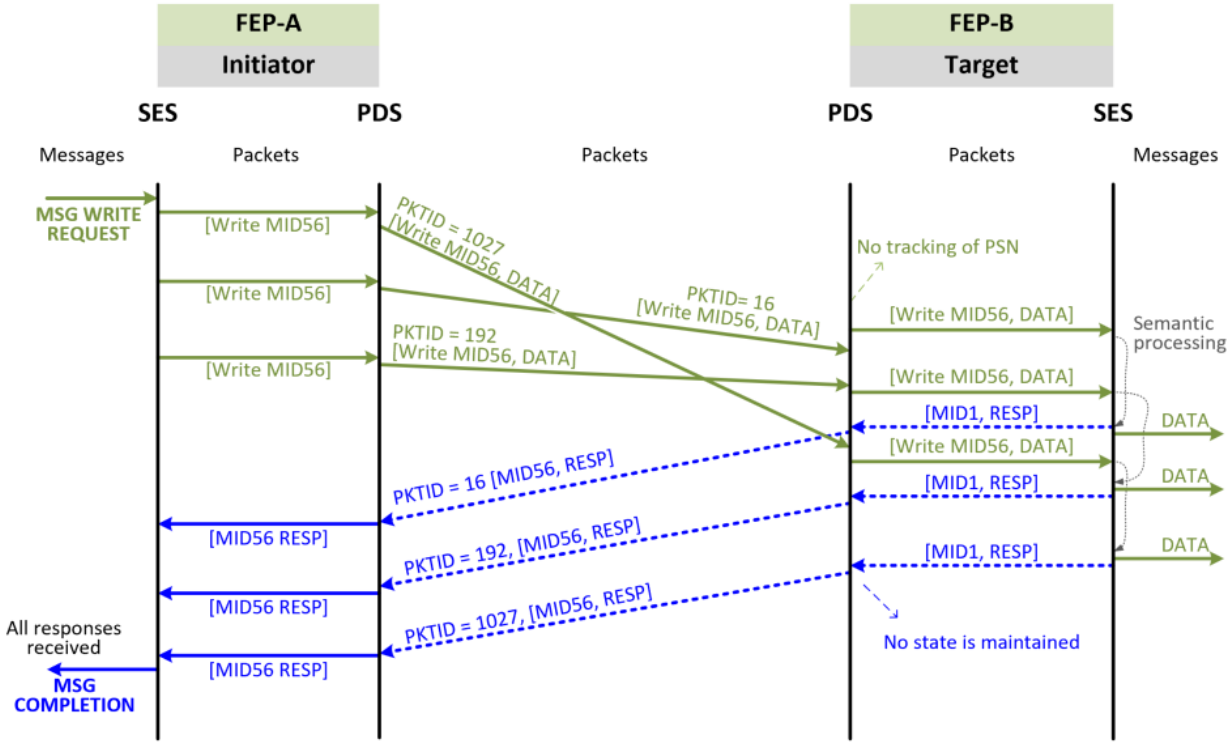


Figure 3-110 - Standard RUDI RMA Write Sequence for Multi-Packet Message

Figure 3-111 shows the standard RUDI RMA Read sequence for a multi-packet message. RUDI return data is carried in a response packet on the forward direction.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

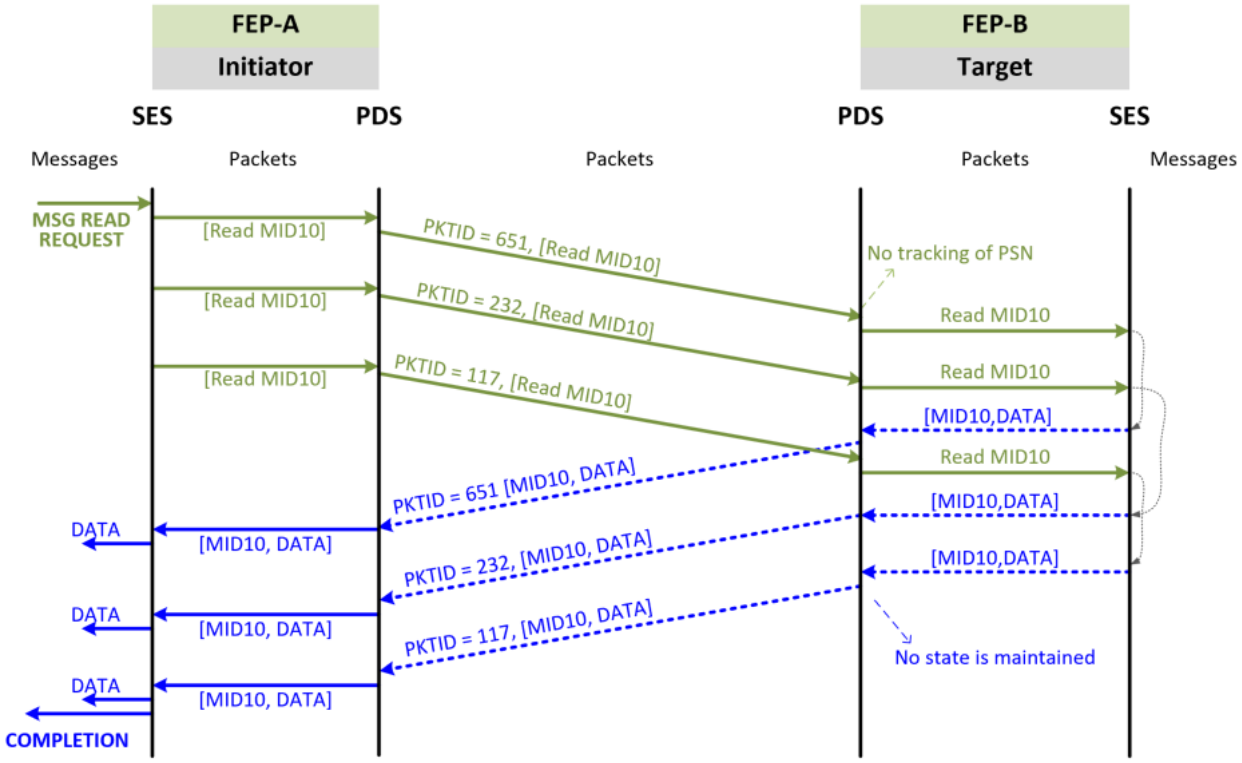


Figure 3-111 - Standard RUDI RMA Read Sequence for Multi-Packet Message

3.5.23.3 Negative Acknowledgements, Packet Drops, and Retries

This section specifies the RUDI sequences for packet drops and other cases that cause a retry.

The methods for determining packet loss are retransmit timeout (RTO) at the initiator, NACKs from target, and trimmed RUDI responses. The target transmits a NACK if a trimmed RUDI request is received and SHOULD transmit a NACK when PDS resources are not available.

The initiator retransmits the request to recover from all packet loss events. The sequence is similar to that shown for packet loss on RUD as seen in Figure 3-104 and Figure 3-105.

RUDI Responses are never NACK'd. There is no state at the target to handle a NACK for a RUDI response.

3.5.24 Error Model

This section specifies the PDS error model. There are different types of events:

- Normal events are handled without impacting the PDC state.
    - For example, sending a NACK due to trimmed packets or out of resources at the target, RTO timeout when a packet is lost in the network, etc.
- Unexpected events that occur due to implementation errata or denial of service attacks, which MUST be handled without impacting well-behaved PDCs (i.e., NACK error type of PDC_ERR).
    - For example, receiving a packet with a PSN that is out of range.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

    - These events SHOULD be counted and available as statistics (i.e., a counter for each NACK code; the counter may be a single bit wide).
- Unrecoverable errors requiring the PDC to be closed (i.e., NACK error type of PDC_FATAL).

Refer to the NACK codes listed in Section 3.5.12.7.

3.5.24.1 PDC Normal Events

These errors are expected and are handled gracefully. For example, if the necessary resources to establish a PDC are unavailable, the target responds to the establishment packet—where the **pds.flags.syn** bit is set—with a PDS NACK packet. This NACK packet includes a **pds.nack_code** field that specifies unavailable resources (i.e., PDS_NACK_PDC).

The source retries the configured number of times (*Max_RTO_Retx_Cnt*) to recover from normal operating events and then report an error if the establishment fails. All packets associated with the failed PDC are dropped, and an error is reported to SES.

3.5.24.2 Unexpected Events

NACK error types of PDC_ERR are unexpected. The events are handled gracefully to minimize the impact of implementation errors and potential denial of service attacks. These events MUST be counted at the source and SHOULD be counted at the destination. The counter may be a single bit.

If a request or CP cannot be delivered, the destination FEP MUST respond with a PDS NACK packet identifying the reason for the packet delivery failure (e.g., unknown PDCID with **pds.flags.syn** set to 0).

At the destination, all packets with unexpected events are dropped. At the source, if a NACK is received with **pds.nack_code** of type ‘unexpected’, the associated PDC is closed and the event reported. All packets associated with the closed PDC are dropped and reported to SES.

3.5.24.3 PDC Unrecoverable Events

If an unrecoverable PDC event occurs (i.e., NACK error type = PDC_FATAL), PDS MUST close the PDC with a Close Command CP or Close Request CP, release the PDC resources, and report the error. An example of a fatal PDC event is exhaustion of a retry counter. All packets associated with a failed PDC are dropped, and a failure event is provided to SES.

These events MUST be counted at the source and SHOULD be counted at the destination.

3.5.25 Full Header Format

This section shows the overall header stack, up to and including UET headers. All headers are expressed in a conventional network byte order that is consistent with IETF RFC 1700. Numbers are expressed as decimal or hexadecimal and are pictured in "big-endian" order. This convention is consistent with IETF RFC 1700.

UET provides an optional 32-bit CRC that can be used when encryption is not used. The UET CRC calculation is defined in Section 3.5.25.7.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Refer to the TSS Section 3.7.11 for the full header format when using encryption. The examples including encryption in Figure 3-114 and Figure 3-117 are simplified header formats showing only header fields that are important for packet parsing.

Not all combinations are shown in the following examples. The provided set of full-stack headers is sufficient for the implementor to visualize any missing combinations. IPv4 and IPv6 are shown using common figures, with IPv4 and IPv6 using Protocol and Next Header, respectively, to identify the next header.

3.5.25.1 UET over UDP/IP, no UET CRC, no Encryption

Figure 3-112 shows the full header stack, depicting important fields necessary for parsing when using UDP, no UET CRC and no UET encryption.

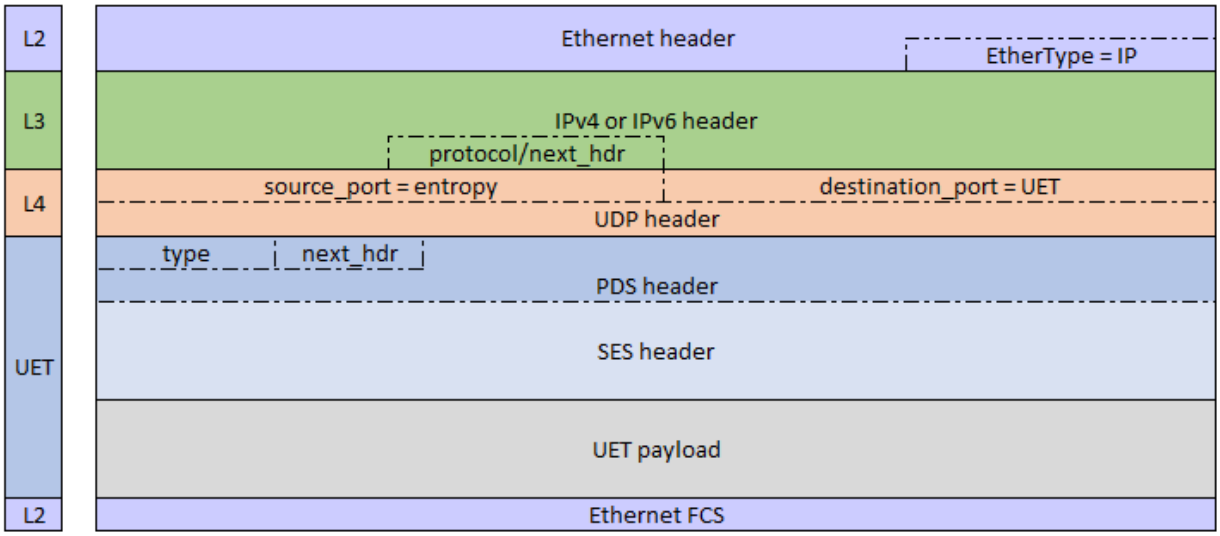


Figure 3-112 - Full Header: UET/UDP/IP – no UET CRC, no TSS

3.5.25.2 UET over UDP/IP Using UET CRC, no Encryption

Figure 3-113 shows the full header stack, depicting important fields necessary for parsing when UDP and UET CRC without UET encryption.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

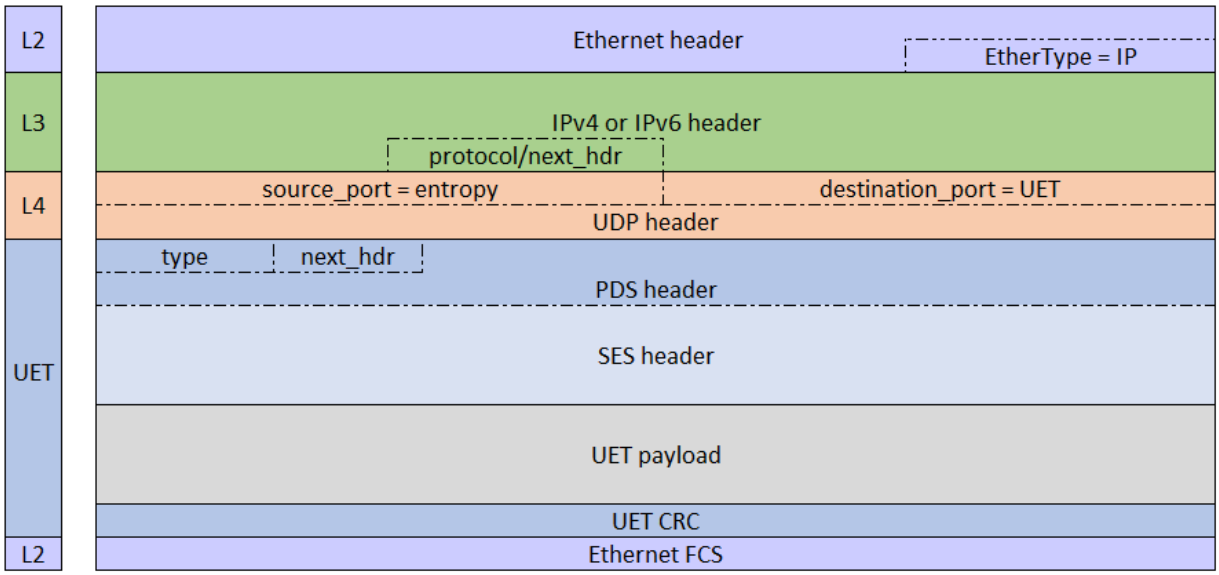


Figure 3-113 - Full Header: UET/UDP/IP with UET CRC, no TSS

3.5.25.3 UET over UDP/IP using Encryption

Figure 3-114 shows the full header stack, depicting important fields necessary for parsing when using UDP, UET CRC and UET encryption.

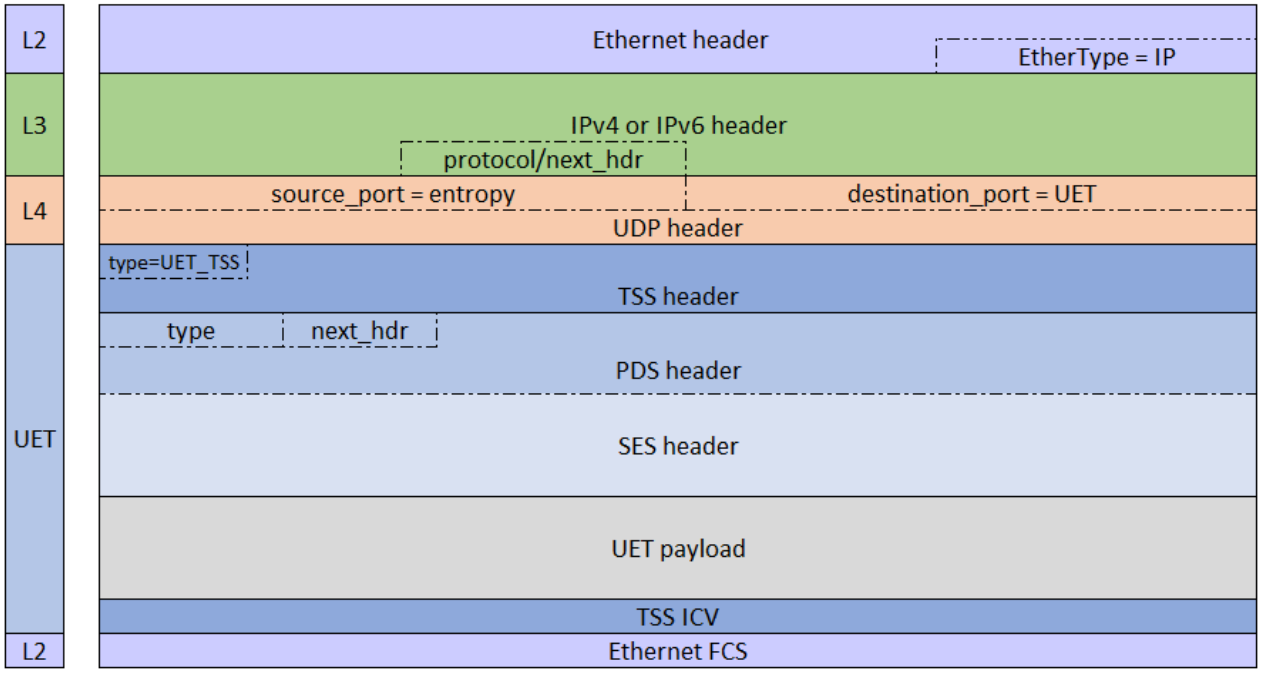


Figure 3-114 - Full Header: UET/UDP/IP with Encryption

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

3.5.25.4 UET directly over IP no UET CRC, no Encryption

Figure 3-115 shows the full header stack, depicting important fields necessary for parsing when using UET directly over IP and UET CRC and UET encryption are not in use.

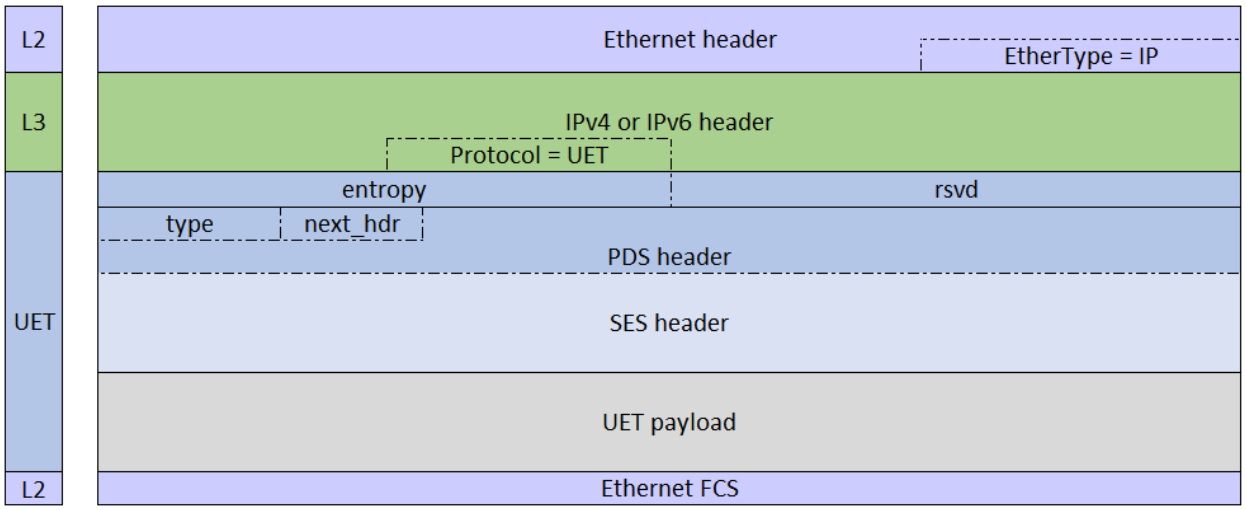


Figure 3-115 - Full Header: UET/IP no UET CRC, no TSS

3.5.25.5 UET directly over IP with UET CRC, no Encryption

Figure 3-116 shows the full header stack, depicting important fields necessary for parsing when using UET directly over IP with the UET CRC and UET encryption is not in use.

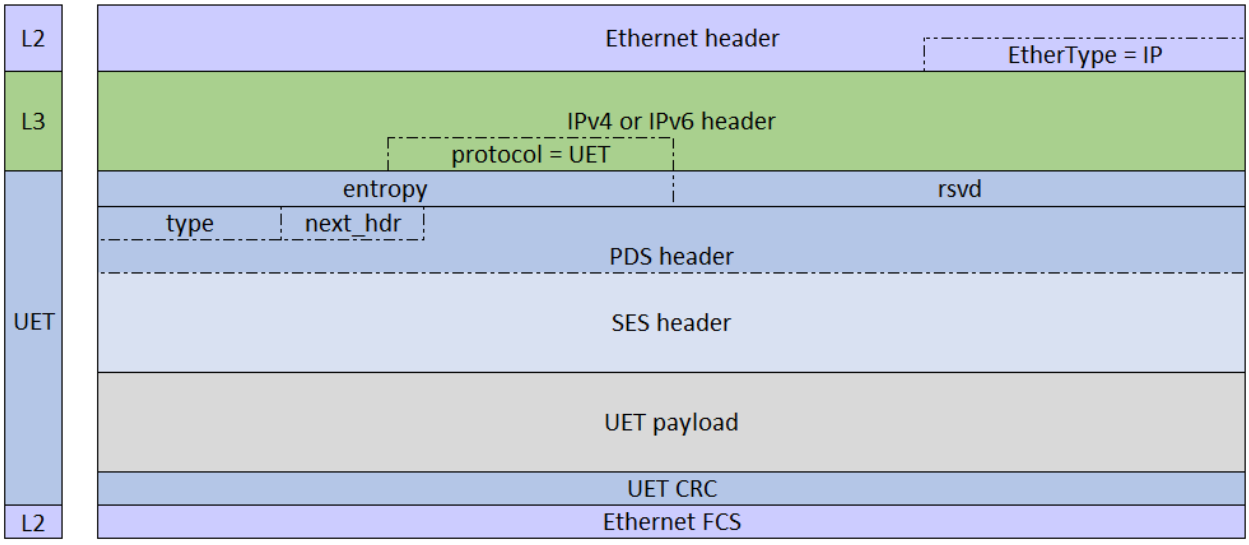


Figure 3-116 - Full Header: UET/IP with UET CRC, no TSS

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

3.5.25.6 UET directly over IP using Encryption

Figure 3-117 shows the full header stack, depicting important fields necessary for parsing when using UET directly over IP with the UET CRC and UET encryption.

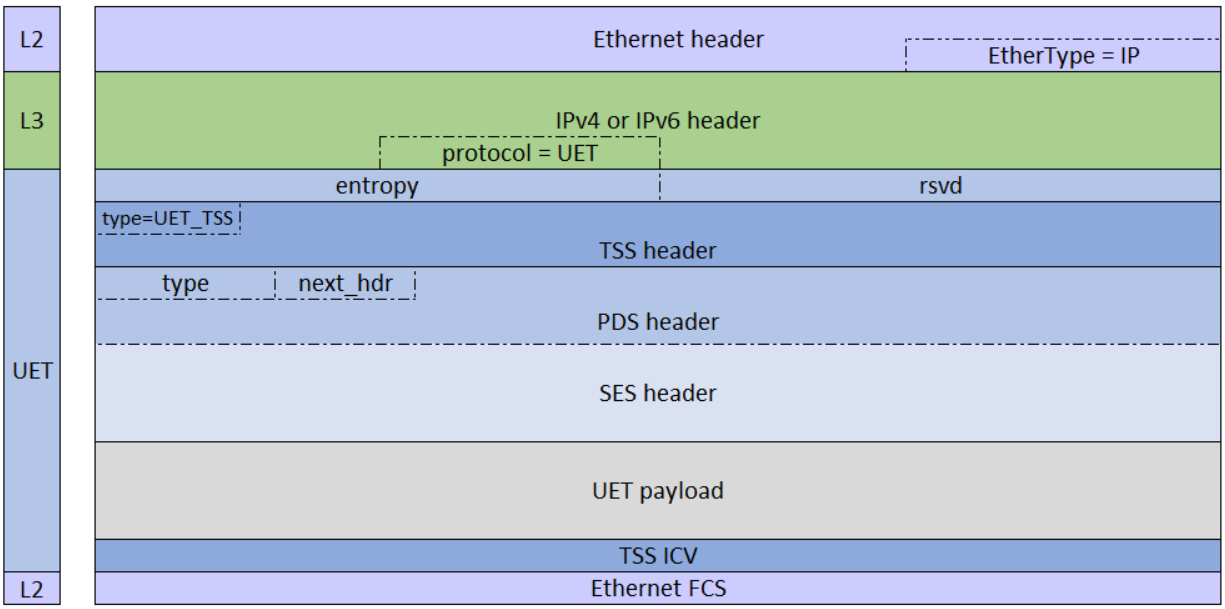


Figure 3-117 - Full Header: UET/IP with Encryption

3.5.25.7 UET over UFH_32 Header

Figure 3-118 illustrates UET over the UFH_32 header. All PDS types can be used over the UFH_32 header with the following differences noted:

- A 16-bit version of the entropy header is carried after the EtherType. The TSS or PDS header starts after the entropy field with type as the first byte.
  - The 2-byte entropy header is always present after the EtherType field.
  - The TSS or PDS header directly follows the entropy field.
- All PDS types will add pad bytes such that the UET CRC can be located directly before the Ethernet FCS—see Section 3.5.26.

This applies to all PDS packet types: RUD, ROD, RODL, RUDI, UUD, ACK, NACK, and control packets.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

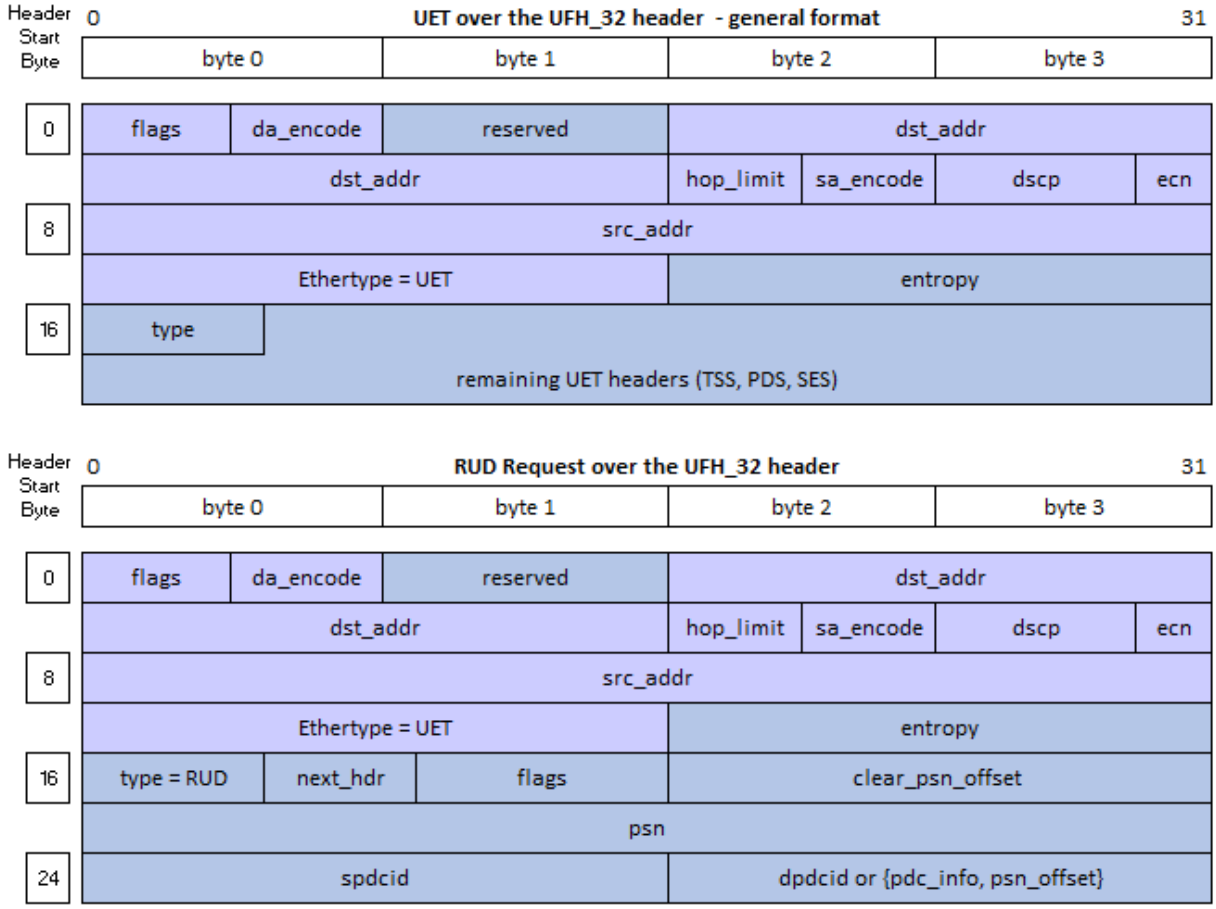


Figure 3-118 - Header Stack for General UET and RUD Requests over the UFH_32 header

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

3.5.25.8 UET RODL - General

Figure 3-119 shows UET RODL over the UFH_32 header. UET RODL also runs over a traditional Ethernet header using the same format. The RODL header, beginning at header start byte 16 in Figure 3-119, remains 12 bytes whether carried over Ethernet or over UDP/IP/Ethernet, consistent with the behavior of RUD and ROD. The **pds.entropy** field is always immediately after the UET EtherType.

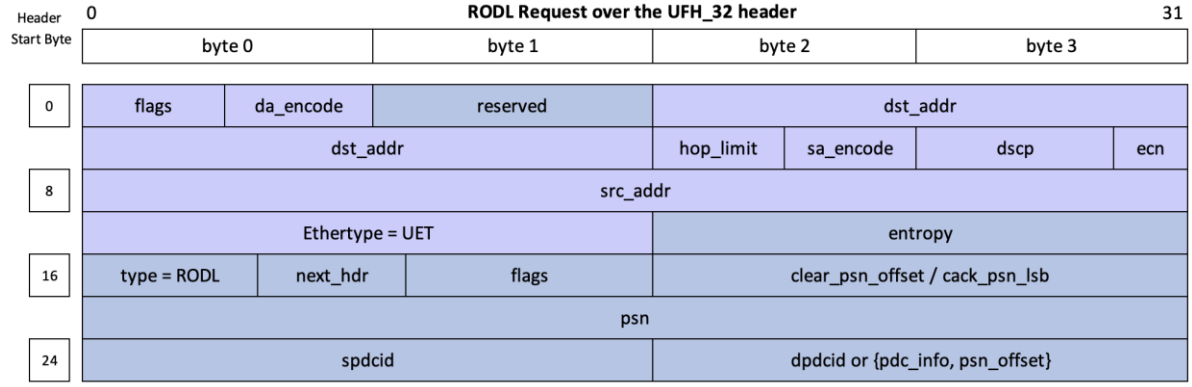


Figure 3-119 - Header Stack for UET RODL

3.5.25.9 UET RODL, ACK, NACK, and CP over the UFH_16 Header

Figure 3-120 illustrates UET over the UFH_16 header, which is optimized for a single tier of switching. Only PDS types RODL, ACK, NACK, and control packets are defined for the UFH_16 header. The illustration shows how these PDS headers are mapped into available bytes in the unified forwarding header with remaining fields following the EtherType.

TSS is not supported with the UFH_16 header.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.



Figure 3-120 - Header Format for UET over the UFH_16 Header

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

3.5.26 UET CRC

This section describes the coverage and calculation used for the UET CRC. The UET CRC is optional and used to provide additional protection beyond the Ethernet FCS. Additional protection can be achieved using TSS. It is RECOMMENDED that either the CRC or TSS is used to ensure end-to-end protection of data and CPs. The CRC is placed after the UET payload in the UET trailer field. A global user configuration (refer to Section 3.5.5), *UET_Data_Protect*, indicates if the trailer contains the CRC. This configuration is the responsibility of the operator and needs to be consistent across the entire fabric domain, as received packets cannot be properly parsed if the CRC is present on some packets and not others in the same domain.

The CRC is designed to include non-mutable portions of the IP headers shown in Figure 3-121.

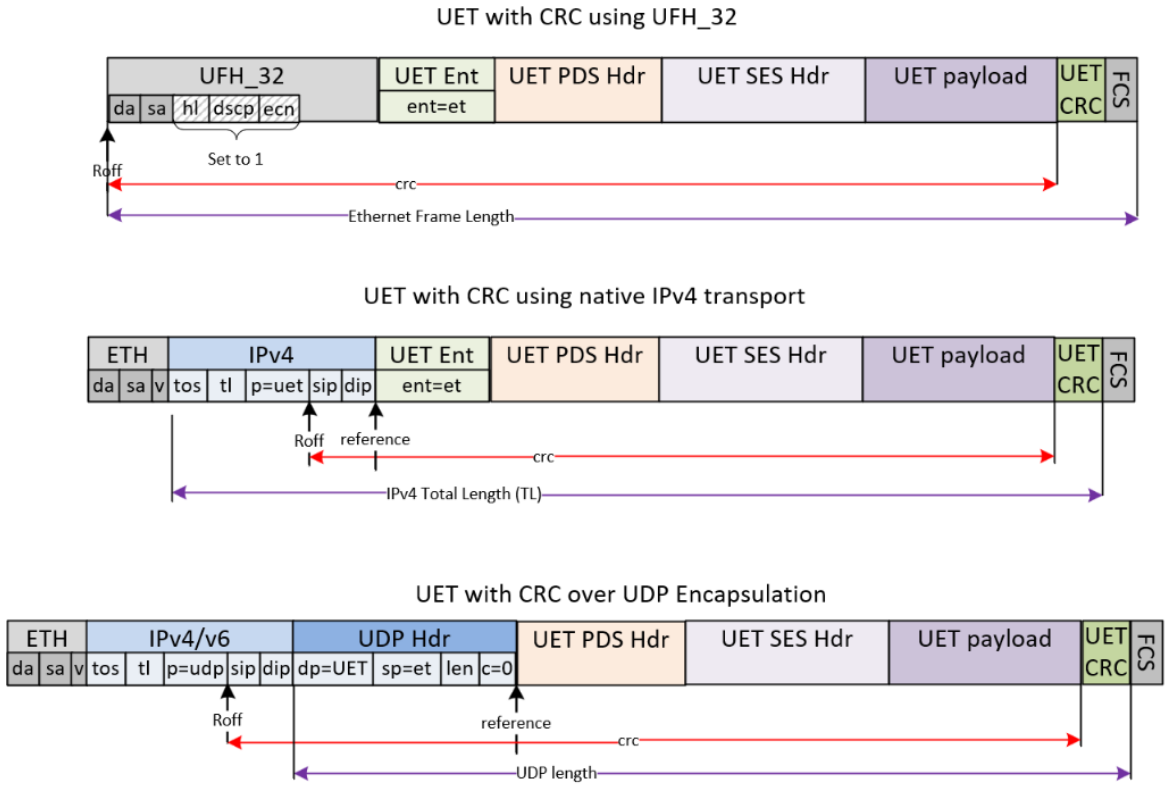


Figure 3-121 - UET CRC Coverage

The Roff offset is calculated automatically based on the packet encapsulation and is not a user-configurable parameter.

When the configuration parameter UET_DATA_PROTECT is set to TSS, the CRC MUST NOT be used. When TSS is enabled, the ICV provides protection equivalent to the CRC.

For a packet sent when the configuration parameter *UET_Data_Protect* is set to CRC, the CRC MUST be calculated for all packets including CPs based on the CRC calculation procedure defined in the steps below and appended as the UET trailer.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

All packets received when the configuration parameter *UET_Data_Protect* is set to CRC MUST be validated based on the CRC procedure, unless the packet was trimmed. The process of trimming can remove the UET CRC, and verification of the CRC cannot be done on trimmed packets.

If the CRC validation fails, the following occurs:

1. The packet is dropped and not passed to SES.
2. The validation failure event is reported.
3. The diagnostic counter UET_CRC_ERR_COUNT is incremented.

The CRC algorithm is CRC32C, also known as Castagnoli. It uses the iSCSI polynomial (0x1EDC6F41, or reversed 0x82F63B78) per IETF RFC 3385 [52]. CRC32C polynomial is:

$$G(x) = x^{32} + x^{28} + x^{27} + x^{26} + x^{25} + x^{23} + x^{22} + x^{19} + x^{18} + x^{14} + x^{13} + x^{11} + x^{10} + x^{9} + x^{8} + x^{6} + 1$$

The CRC calculation procedure uses the following steps:

1. The CRC is calculated over the packet starting at the first byte of the source IP address for both IPv4 and IPv6 packets, as defined by the Roff offset.
2. For UDP packets, the checksum is set to zero for calculation and verification.
3. The initialization value is 0xFFFFFFFF, and the polynomial in normal form is 0x1EDC6F41.
4. Galois Field (GF2) polynomial division of the message, M(x), by the generator polynomial, G(x), is used to produce the remainder, R(x).
5. The calculation is performed in big-endian byte order with the least significant bit first.
6. The CRC value is the complemented value of R(x) and appended to the end of the UET payload in network byte order. The 32 bits of the CRC value are placed after the UET payload as the UET trailer, so that the x^{31} term is the left-most bit of the first octet, and the x^{0} term is the right-most bit of the last octet.

3.5.26.1 UET CRC over UFH_16

The UET CRC calculation over UFH_16 uses the same CRC32C algorithm; however, the CRC calculation is performed differently:

$$\mathrm{G(x)} = \mathrm{x^{32} + x^{28} + x^{27} + x^{26} + x^{25} + x^{23} + x^{22} + x^{19} + x^{18} + x^{14} + x^{13} + x^{11} + x^{10} + x^{9} + x^{8} + x^{6} + 1}$$

The CRC calculation uses the following steps:

1. The CRC is calculated over the packet starting as defined by Roff.
    a. Roff starts at the first byte of the packet with the UFH_16 header.
2. The 32-bit expected PSN is appended after the SES headers and payload in the calculation of the CRC when **pds.type** = RODL over the UFH_16 header.
3. Any masked fields are set to 0x0.
    a. The **ufh.dscp and ufh.ecn** fields are masked when using the UFH_16 header**.**
4. The initialization value is 0xFFFFFFFF and the polynomial in normal form is 0x1EDC6F41.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

5. Galois Field (GF2) polynomial division of the message, M(x), by the generator polynomial, G(x), is used to produce the remainder, R(x).
6. The calculation is performed in big-endian byte order with the least significant bit first.
7. The CRC value is the complemented value of R(x) and appended to the end of the UET payload in network byte order. The 32 bits of the CRC value are placed after the UET payload as the UET trailer, so that the x^{31} term is the left-most bit of the first octet, and the x^{0} term is the right-most bit of the last octet.

If validation fails, the same processing is performed for UET over non-UFH addressing except when RODL is used without PDS retransmission. This configuration is described in Section 3.5.10.5.

The UET CRC MUST be included when using the UFH_16 header. The CRC is required as it carries the PSN state. The coverage when using the UFH_16 header is illustrated in Figure 3-122.

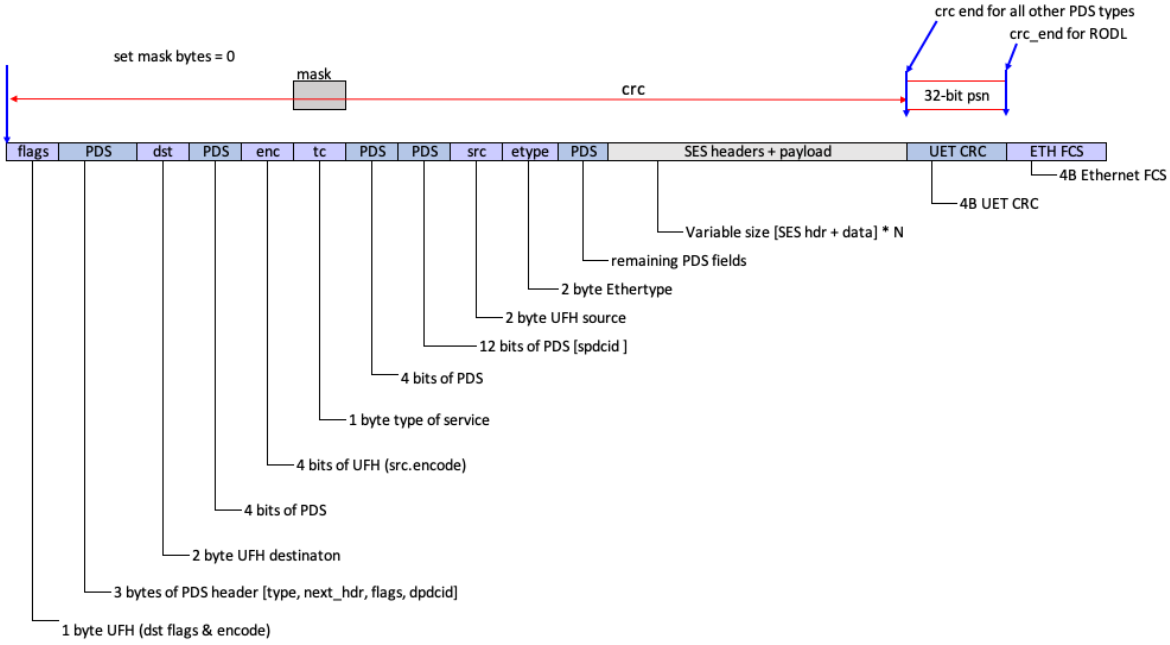


Figure 3-122 - UET CRC coverage when using the UFH_16 header

The UET CRC is always located directly before the Ethernet FCS. When the sum of the UFH header, PDS header, SES header(s) and SES payload is less than 56 bytes, sufficient padding is added after the last SES payload to reach the 56-byte size. This assures the packet meets the minimum Ethernet frame size of 64 bytes when the 4-byte UET CRC and 4-byte FCS are added.

3.5.27 Padding of Ethernet Packets

Figure 3-123 shows example packets where pad bytes are used. The first packet is a RODL request sending a read request over the UFH_32 header. The second is an ACK packet over the UFH_16 header.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

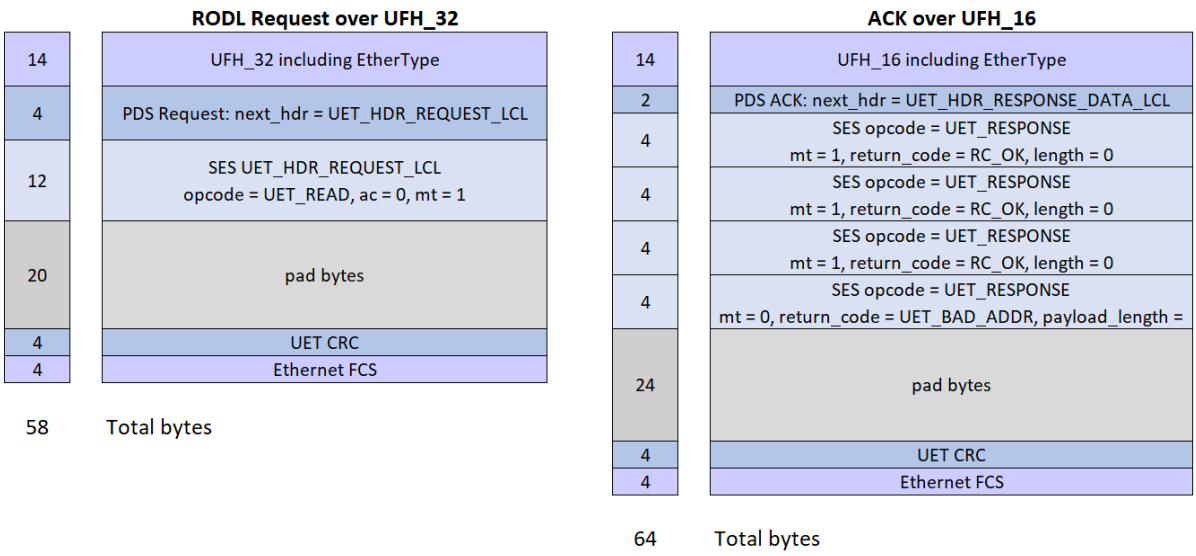


Figure 3-123 - Example packets using padding

Informative Text:
The size and padding calculation exclude any other headers such at VLAN or CSIG. These headers may be removed by network elements. It is required that the total size of the base packet be 64B or larger.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

3.6 Congestion Management Sublayer (CMS)

UET defines an end-to-end congestion management solution, UET-CC (UET Congestion Control), for packet buffer congestion in a best-effort (sometimes called ”lossy”) Ethernet network. The goal is to achieve high network efficiency, reduce packet loss, and minimize latency, including tail latency, and to ensure reasonable fairness between competing flows.

The target network (referred to as a backend network) is primarily carrying HPC and AI training and/or inference workloads. To avoid unfairness, it is recommended that all traffic in the same traffic class as UET-CC traffic runs UET congestion control. UET-CC is not intended for wide-area network traffic, as it assumes a low-latency control loop: UET-CC aims at expected base RTTs that range from 1 µs to 20 µs. It is recommended that network fabric control traffic such as BGP running over TCP, or other traffic not using UET-CC, is run in a separate traffic class. UET-CC performs adaptive load balancing. Therefore, it should be able to load balance around traffic in higher priority traffic classes, including non-UET traffic classes, as long as that traffic does not itself congest the network.

The Unified Local profile and the RODL delivery mode do not require support for congestion management.

Congestion management can be decomposed into the following components:

- **Telemetry:** Determining the congestion state of network paths, locally on the endpoint and on the network; this information may be collected at and/or used by the initiator, switches along the path, or target.
- **Sender-based window:** Controlling the maximum outstanding, unacknowledged data, as measured in bytes.
- **Receiver-credit congestion control:** Controlling the rate at which data may be transmitted to a specific destination to more directly control incast.
- **Multipath path selection:** Modifying the path packets take to minimize congestion using adaptive packet spraying, with a granularity of spraying at the packet level.

3.6.1 UET CC Guidelines [Informational]

The design of UET-CC is based upon a number of core design decisions, listed below for background. Although words typically associated with normative requirements are used, this section does not specify requirements for compliance, which are provided when each feature is specified separately.

- UET-CC is designed to operate with existing network switches. This necessitates that UET-CC rely only on basic features including:
    - The ability of switches to classify traffic and provide independent queuing based on traffic class (TC), where TC is assigned based on IP DSCP or VLAN priority code point (PCP).
    - The ability of switches to provide explicit congestion notification (ECN) marking as specified by IETF RFC 3168 [14].

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

- UET-CC may provide improved performance if packet trimming is supported and enabled in switches.
- UET-CC supports multipathing through the use of adaptive packet spraying.
- Some unsprayed UET-CC traffic may be present, but it is intended that the majority of traffic is sprayed. UET-CC is designed to support the co-existence of sprayed and a minority of unsprayed traffic.
    - High-bandwidth AI workloads have been shown to benefit significantly from multipathing.
- UET-CC is designed to provide at least a moderate level of fairness. Strict fairness between competing UET-CC sessions for network capacity is not a requirement.
    - In particular, short-term imbalance (transient unfairness) in bandwidth or latency between connections is less important than reducing tail latency.
    - Complete blocking (starvation) of any connection must be avoided.
- UET-CC aims to provide fairness between competing congestion control contexts (CCCs). Each CCC may provide congestion control for multiple PDCs. Fairness between multiple PDCs within a CCC is left to implementations. When two competing CCCs contain different numbers of PDCs, it is not a goal of UET-CC to provide fairness at the PDC level.

3.6.2 Congestion Control Algorithms

UET-CC introduces three congestion control algorithms that can be implemented by devices targeting best-effort networks. Network signal congestion control (NSCC) MUST be implemented. Receiver-credit congestion control (RCCC) is an optional algorithm that may be used in conjunction with NSCC, or in a stand-alone capacity. Transport flow control (TFC) is designed to provide a point-to-point flow-controlled channel. TFC is an optional CC mechanism.

3.6.3 Congestion Control Algorithm Design Targets

UET-CC is designed to operate on best-effort networks and address:

- Network congestion
- Incast at the target
- Bulk-data messaging using requests (e.g., UET_WRITE, UET_SEND, etc. of messages greater than one BDP in size or a stream of such operations between a pair of FEPs aggregating to more than one BDP in size)
- Moderate-bandwidth interfering flows in the network
- Flow control (using TFC) for point-to-point protocols

Additional use cases (e.g., bulk data UET_READ transactions) are believed to be addressed, but were not the primary design point; however, some use cases (e.g., lossless networks) are specifically not covered in the initial design.

3.6.4 Telemetry and Network Switch Services

This section describes the supported telemetry mechanisms for monitoring the congestion state in the network, including the endpoint. Due to the guiding decision that the protocol must work on existing

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

switches, the only REQUIRED in-network telemetry is ECN marking. Endpoint telemetry including end-to-end latency and endpoint state (e.g., dropped packets) is included. Support for packet trimming is REQUIRED by UET-CC endpoints but is OPTIONAL to support or enable in UE switches.

3.6.4.1 Explicit Congestion Notification (ECN)

The use of ECN is a required mode for the UE congestion management and MUST be supported by UE-compliant endpoints and switches. As defined in IETF RFC 3168 [14], ECN is a single bit of telemetry per packet that is provided by network switches.

Each switch monitors the queue depth for the traffic class used by the protocol. As IP packets traverse the network, the ECN flag is set if a configured threshold is crossed (called deterministic marking). Alternatively, the probability of marking the packet may vary based on the queue depth (called probabilistic marking). UET switches MUST support using instantaneous queue depth when determining whether to ECN-mark a packet.

If NSCC is enabled or if path-aware ECN-based load balancing is enabled, all UET data packets MUST be marked as ECN-capable by setting the IP ECN-ECT bit. In other cases, setting ECN-ECT is OPTIONAL.

The NSCC algorithm defined in this specification assumes that the ECN congestion experienced (ECN-CE) flag is set when a packet is dequeued, whereas IETF RFC 3168 [14] implies ECN-CE is set when a packet is enqueued but would otherwise have been dropped. Setting ECN-CE on dequeue is common practice in current network switches, as it provides a more up-to-date indication of network congestion. The UET-CC sender algorithm will not perform as desired if ECN-CE is set on enqueue. Switches SHOULD perform ECN marking when the packet is dequeued.

3.6.4.2 Round-Trip Time (RTT)

The NSCC algorithm uses latency as an indication of the level of congestion in the path from the sender to the receiver. To measure latency, UET-CC measures the round-trip time (RTT). To avoid return-path congestion being misinterpreted as forward path congestion, it is RECOMMENDED that UET ACKs, NACKs, and control packets are sent using a higher priority traffic class than UET data packets (Section 3.6.4.7). There may be some delay between a UET data packet arriving at the destination and an ACK being transmitted. To avoid this receiver delay being misinterpreted as a sign of network congestion, the destination is REQUIRED to measure this delay and report it in ACK packets as **pds.ack_cc_state.service_time** (Section 3.6.9.2.4). The source then subtracts **pds.ack_cc_state.service_time** from the measured round-trip time to obtain an estimate of the actual network round-trip time. The algorithm to measure RTT is therefore:

1. The source saves a timestamp for some or all transmitted PDS request packets.
2. The destination PDS calculates the service time at the target:
    a. The destination PDS captures the arrival time of the received packet and the transmit time of the corresponding PDS ACK.
    b. Service time = ACK.transmit time – PDS_request.arrival time.
    c. Service time is carried in the associated ACK packet.
3. The source calculates the RTT using the timestamp from the arrival of the ACK message.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

a. RTT = ACK.arrival_time – PDS_Request.transmit_time – ACK.service time.

At the destination, the service time should be calculated as close to the transmission time of the ACK packet as practically possible. Any delay between calculating the service time and actually sending the ACK will be interpreted as network latency by the NSCC algorithm.

SERVICE_TIME is specified in units of 128 ns, with a goal of overall accuracy of ~500 ns. Implementations SHOULD maintain saved timestamps with sufficient accuracy and precision to achieve this goal.

3.6.4.3 Endpoint Congestion State

Endpoints can monitor congestion using locally available information. This is endpoint telemetry, which can be used without any support from network switches. Example metrics that could be measured include indications of local congestion at endpoints, such as:

- The current offered workload to multiple destinations at the source, which indicates potential source congestion (outcast).
- The depth of the packet buffer at the destination, which indicates potential destination congestion (e.g., the destination cannot process packets fast enough or cannot sink data into memory fast enough).
- Transmit and receive rates for non-UET traffic.

This information may be included in the congestion state used to adjust the rate limiting.

3.6.4.4 Packet Trimming

Packet trimming is a method where, rather than dropping packets when a switch buffer is congested, packets are trimmed to a small size and forwarded with higher priority to the destination. This capability was introduced in NDP [31] and Cut Payload [30]. This is a form of in-network telemetry provided by network switches.

Packet trimming is intended to enable the use of relatively shallow packet buffer queues in the network switches. In turn, shallow packet buffer queues help minimize the RTT, allowing the overall control loop to work more efficiently. When a trimmed packet arrives at the destination:

- The destination knows the source is attempting to send to it and can allocate resources such as credit to transmit, if required.
- The destination knows the network is congested on the path the packet took.
    - The EV used to determine the path of the packet can be reported back to the source, allowing the source to temporarily avoid using the congested path.
- The destination knows a specific packet was dropped and can quickly request retransmission, reducing the recovery time:
    - Trimmed packets are forwarded with high priority, so they arrive quickly and provide an early indication of packet loss. Recovering lost packets quickly reduces the state needed to track out-of-sequence arrivals and helps avoid increases in flow completion time due to tail loss.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

- UET sprays packets across many paths, resulting in reordering. This complicates loss detection, as the simple arrival of a higher sequence number cannot be used to infer loss.

Packet trimming helps mitigate the problem of loss detection by informing the destination which packets were lost, providing congestion state and explicit drop notifications. This allows congestion management algorithms to activate and allows reliability mechanisms to efficiently recover lost packets.

In its simplest form, the procedure is:

- Packets that are eligible to be trimmed are sent with a specific diffserv codepoint, DSCP_TRIMMABLE, chosen for that particular network and class of UET traffic.
- Switches that are configured to perform trimming will only trim packets that they know to be trimmable, as indicated by DSCP_TRIMMABLE.
- When a packet with DSCP_TRIMMABLE arrives at such a switch, if a queue would overflow and the packet would be dropped, the packet is truncated to no smaller than MIN_TRIM_SIZE, which is a network-specific configuration parameter.
- The IP length field is modified and the IP header checksum (for IPv4) is updated so that the trimmed packet is a valid IP datagram.
- Any payload fields left in the truncated packet (including the UDP length field, if UDP is used) are unmodified.
- The DCSP is changed to DSCP_TRIMMED (also configured for that particular network and class of UET traffic) to indicate to the network that the packet has been trimmed and cannot be trimmed again, and to indicate to the receiving FEP that the packet was trimmed.
- After trimming, the trimming switch should treat the packet in the same way it would treat a newly arrived packet with DSCP_TRIMMED.
- The network should typically be configured so that DCSP_TRIMMED indicates to switches that a different traffic class should be used, so that trimmed packets do not sit behind untrimmed packets in switch queues.

This description of packet trimming is provided for guidance only; more complete details of trimming and compliance requirements are in Section 4.1.

Packet trimming can assist in the efficient fast startup of multiple PDCs converging on a common destination by providing congestion signals that explicitly identify lost packets. Fast startup is used to minimize latency. Time used to slowly build up to faster rates is added latency and reduced network utilization. However, simultaneous fast startup can lead to excessive congestion and packet drops, which may lead to higher latency and longer tail latency. Packet trimming helps recover from such congestion quickly.

UE FEPs MUST be capable of receiving trimmed data packets, as identified by the **ip.dscp** field, and generating NACKs for them (see Section 3.5.15.1). Implementation and enabling of trimming in switches in UET deployments is OPTIONAL.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

When trimming is used with UET-CC, MIN_TRIM_SIZE MUST configured large enough for trimmed packets to include the UET PDS Request header and any optional *CC_State* included in the header — taking into account any encapsulations that may occur.

When trimming occurs on the last hop in the network, a different code point, DSCP_TRIMMED_LASTHOP, MAY be set to indicate to the destination where the congestion occurred, as last hop congestion due to incast may be handled differently from other congestion. In the switch, DSCP_TRIMMED_LASTHOP should be treated the same as DSCP_TRIMMED. In the receiving FEP, DSCP_TRIMMED_LASTHOP indicates that the packet should be processed in the same way as a DSCP_TRIMMED packet, except:

- It is not used as an input to the load-balancing algorithm (at either the sending or receiving FEP) because no alternate path is available.
- It is not used as a congestion signal to NSCC when RCCC is also enabled, because RCCC can handle last-hop congestion by itself.

The UET-CC algorithms in this specification assume that either all data packets sent on a particular PDC are marked as eligible for trimming (using DSCP_TRIMMABLE) or none are. Mixing trimmable and non-trimmable data packets in a PDC is NOT RECOMMENDED. Similarly, it is recommended that all sprayed PDCs in a traffic class are marked as eligible for trimming, or none are. Although it is not required that all PDCs sent in a specific traffic class are marked as eligible for trimming, care should be taken when mixing trimmable and non-trimmable PDCs in the same network to avoid the potential for unfair sharing of the network.

3.6.4.5 Priority Flow Control

Priority flow control (PFC) is defined in clause 36 of IEEE Std 802.1Q-2022 [15]. PFC uses a queue buffer depth threshold. When the threshold is crossed, a PFC is sent to the attached device. All traffic on the specified traffic class(es) is paused until another PFC message clears the pause or a timer expires. These messages are generated and accepted by Ethernet switches and NICs.

UET-CC is designed for operation in a best-effort network and assumes that PFC is not enabled for end-to-end UET traffic on switch-to-switch links in the network. Enabling PFC on such links is likely to reduce performance, as it will violate the latency assumptions of the CC algorithms. PFC SHOULD NOT be used anywhere in a best-effort network.

3.6.4.6 Credit-based Flow Control

The UE Link Layer specification defines credit-based flow control (CBFC) that operates hop-by-hop in the network to provide more fine-grained flow control than can be provided by PFC. CBFC allows a receiving NIC or downstream Ethernet switch to regulate traffic of the specified traffic classes sent to it by an upstream NIC or Ethernet switch. CBFC is specified in Section 5.2.

As with PFC, UET-CC assumes that CBFC is not enabled for end-to-end UET traffic on switch-to-switch links in the network. Enabling CBFC on such links is likely to violate the latency assumptions of the CC

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

algorithms, which may reduce performance. CBFC SHOULD NOT be used anywhere in a best-effort network.

Implementation Note:
SES and PDS in conjunction with the TC mappings presented in Section 3.6.4.7 are designed for either best-effort or lossless operation. Historical deployments have utilized flow control on the egress link (egress switch to NIC) in networks that were otherwise best-effort networks; however, this is not a recommended deployment model. Implementations that wish to support this model and use the best-effort TC mappings should take great care.

Informative Text:
CBFC, which operates at the link level, should not be confused with receiver-credit congestion control, which operates end-to-end and is one of the UET-CC algorithms.

3.6.4.7 Mapping UET to Traffic Classes and DSCP

UET-CC is designed to operate using either two traffic classes (TCs) or three traffic classes, where each TC is provided independent packet buffering, ECN marking, trimming, and scheduling services relative to other TCs. UET packets are mapped to a traffic class based on DSCP. UET defines mappings onto between two and six DSCP values that utilize abstract names (e.g., DSCP_TRIMMABLE). Each of the DSCP names is mapped to an underlying DSCP value based on the network (and class of UET traffic).

The allocation of packet types to traffic classes depends on whether the network is best-effort or lossless. This specification focuses on best-effort operation, but traffic classes for lossless are also described in Section 3.6.4.7.3.

3.6.4.7.1 DSCP Mappings

In the simplest deployments, UET can use as few as two DSCP values, as illustrated in Table 3-85. This operational model does not use trimming and segregates packets into either a bulk data DSCP or a control DSCP. The bulk data DSCP is named DSCP_NO_TRIM to retain some nomenclature overlap with the trimming use case. A third DSCP, DSCP_TRIMMABLE_RTX, can also be used without trimming to separate retransmitted packets.

Table 3-85 - UET DSCP Mappings without Trimming

PDS Packet Types	2 DSCP Mapping	3 DSCP Mapping
PDS Requests (RUD, ROD, RUDI)	DSCP_NO_TRIM	DSCP_NO_TRIM
Retransmitted PDS Requests	DSCP_NO_TRIM	DSCP_TRIMMABLE_RTX
Control Packets	DSCP_CONTROL or DSCP_NO_TRIM	DSCP_CONTROL or DSCP_NO_TRIM
PDS ACKs	DSCP_CONTROL	DSCP_CONTROL

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

PDS Packet Types	2 DSCP Mapping	3 DSCP Mapping
PDS NACKs	DSCP_CONTROL	DSCP_CONTROL
RUDI Send/Write Response Packets	DSCP_CONTROL	DSCP_CONTROL
RUDI Read Response Packets	DSCP_NO_TRIM	DSCP_NO_TRIM
Note: • Control packets default to using DSCP_CONTROL. Special use cases—such as those needing sequencing behind other packets—MAY place control packets on DSCP_NO_TRIM.		

If trimming is enabled, additional information is needed to separate trimmable from trimmed packets. This is accomplished by using a third DSCP, as shown in Table 3-86. Additional DSCP values can be used to provide additional information to UET-CC, as shown in the 4+ DSCP Mapping column Table 3-86. Additional DSCP values beyond four may be needed to indicate a packet should not be trimmed (DSCP_NO_TRIM) if an ACK, NACK, or control packet would be larger than the trim size, for example. Similarly, it may be advantageous to indicate that a packet is a retransmitted packet.

UET does not mandate that specific numeric values be associated with these DSCP names; the mapping of DSCP names to specific values is the responsibility of the network operator and is dependent on the specific network configuration. Similarly, it is possible for a network to have more than one class of UET traffic, in which case these DSCP names and traffic classes may be replicated as appropriate.

Table 3-86 - UET DSCP Mappings with Trimming

PDS Packet Types	3 DSCP Mapping	4+ DSCP Mapping
PDS Requests (RUD, ROD, RODL, RUDI)	DSCP_TRIMMABLE	DSCP_TRIMMABLE
Retransmitted PDS Requests	DCSP_TRIMMABLE	DSCP_TRIMMABLE or DSCP_TRIMMABLE_RTX[1]
Control Packets[2]	DSCP_CONTROL or DSCP_TRIMMABLE	DSCP_CONTROL or DSCP_TRIMMABLE or DSCP_NO_TRIM[3]
PDS ACKs >= *Trimmable_ACK_Size*	DSCP_TRIMMABLE	DSCP_TRIMMABLE
PDS ACKs < *Trimmable_ACK_Size*	DSCP_CONTROL	DSCP_CONTROL
PDS NACKs	DSCP_CONTROL	DSCP_CONTROL
RUDI Send/Write Response Packets	DSCP_CONTROL	DSCP_CONTROL
RUDI Read Response Packets	DSCP_TRIMMABLE	DSCP_TRIMMABLE or DSCP_NO_TRIM
Trimmed Packet	DSCP_TRIMMED	DSCP_TRIMMED
Packet Trimmed on Last Hop	DSCP_TRIMMED	DSCP_TRIMMED_LASTHOP
Note:		

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

PDS Packet Types	3 DSCP Mapping	4+ DSCP Mapping
1. It is advantageous if retransmitted packets are not dropped. To enable this, deployments and implementations MAY use a DSCP_TRIMMABLE_RTX value for retransmitted packets. This would be the fifth DSCP. 2. Control packets default to using DSCP_CONTROL. Special use cases—such as those needing sequencing behind other packets—MAY place control packets on DSCP_TRIMMABLE. 3. Control packets that are trimmed (due to using DSCP_TRIMMABLE) are dropped at the target. In deployments where it is preferable to drop a control packet that requires sequencing behind data packets, a sixth DSCP may be used: DSCP_NO_TRIM.		

The codepoint DSCP_TRIMMABLE_RTX is used for retransmitted data packets. Using this codepoint is OPTIONAL. If it is not used, the packets SHOULD use the DSCP_TRIMMABLE codepoint (DSCP_NO_TRIM when trimming is not enabled).

UET benefits when retransmitted data packets are not dropped or trimmed, as this may cause the packet tracker to run out of space, hitting the PDC MPR limit and causing a performance loss. However, it is NOT RECOMMENDED to place retransmitted data packets in DSCP_CONTROL, because this can delay control packets and confuse the congestion control loop. To reduce their probability of trimming or loss, switches SHOULD be configured with a higher trim threshold for DSCP_TRIMMABLE_RTX than for DSCP_TRIMMABLE (or DSCP_NO_TRIM) packets.

3.6.4.7.2 Traffic Classes for Best Effort Networks

When using two traffic classes, the traffic classes are referred to as TC_low and TC_high. When using three traffic classes, they are referred to as TC_low, TC_med, and TC_high. Traffic in TC_high SHOULD receive priority treatment over TC_med, and TC_med SHOULD receive priority treatment over TC_low. How such priority treatment is provided in UE switches is implementation specific. For example, this can be provided by appropriate use of variants of weighted round-robin algorithms. UET expects that low-rate traffic sent on the TC_high traffic class will experience very small queuing delays irrespective of the load in TC_med and TC_low traffic classes. Similarly, low-rate traffic sent on the TC_med traffic class SHOULD NOT be delayed significantly by TC_low traffic.

Implementation Note:
Care should be taken in the implementation of priorities. In some traffic patterns, DSCP_CONTROL traffic could represent as much as 40% of the total bandwidth. For example, a basic implementation of the HPC Challenge (HPCC) RandomAccess benchmark [32] sends a single tiny request to each of many destinations. These patterns may not leverage UET-CC as effectively as bulk data flows; however, consideration of these workloads should be given when implementing the handling of DSCP_CONTROL traffic and when setting QoS policies for UET networks. Giving DSCP_CONTROL traffic strict priority or allocating a small fraction of overall bandwidth to DSCP_CONTROL traffic could lead to unforeseen network behaviors.

The recommended mapping of DSCP codepoints to traffic classes is shown in Table 3-87. Separate mappings are shown for using two TCs or three TCs (depending on network configuration).

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Table 3-87 - UET DSCP to TC Mappings for Best Effort Networks

DSCP	2 TC Mapping	3 TC Mapping
DSCP_CONTROL	TC_high	TC_high
DSCP_TRIMMED	TC_high	TC_med
DSCP_TRIMMED_LASTHOP	TC_high	TC_med
DSCP_TRIMMABLE	TC_low	TC_low
DSCP_TRIMMABLE_RTX	TC_low	TC_low
DSCP_NO_TRIM	TC_low	TC_low
Note: • DSCP_TRIMMABLE_RTX should be used to reduce the probability of loss of a retransmitted packet—even though it shares the TC_low traffic class.		

RUDI packets are not controlled by UET-CC. RUDI packets that consume substantial bandwidth to a destination SHOULD be mapped to a separate traffic class. How this traffic class is configured and the DSCP used are network-specific parameters that depend on the use case.

A single PDC mapped over two best-effort traffic classes is illustrated in Figure 3-124. The PDC is between two PDS instances, as depicted within the dashed box in Figure 3-124.

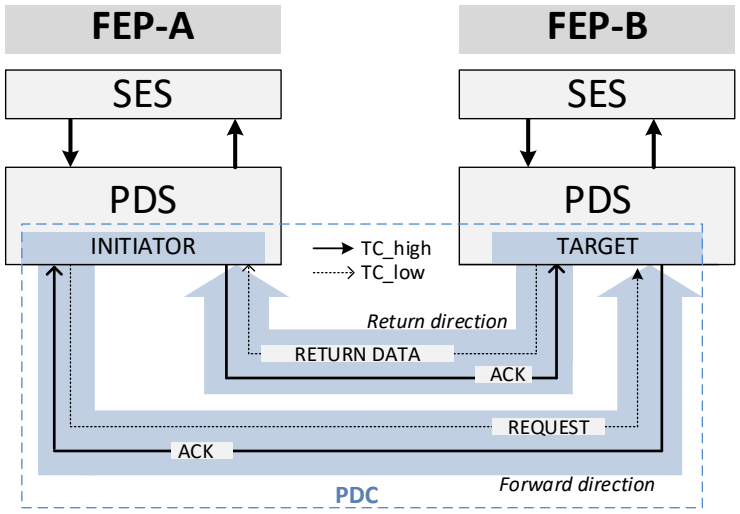


Figure 3-124 - PDC mapped over Best-Effort Ethernet: Traffic Class Mapping (ROD and RUD)

3.6.4.7.3 Traffic Classes for Lossless Networks

When UET is used in lossless networks, a different mapping of packet types to traffic classes is required to avoid potential deadlocks. In a lossless network, two TCs, TC_request and TC_response, SHOULD be

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

used. All UET requests SHOULD be mapped to TC_request. Packets flowing in the return direction (i.e., ACK/NACK packets and RUDI response packets) SHOULD be mapped to TC_response. This avoids deadlock when pausing TC_request.

The goal is to allow responses through when request packets are paused. If this were not the case and requests caused responses to be paused, then a deadlock is possible because a resource-limited FEP may have to pause incoming requests pending a response that frees resources. If pausing requests cannot cause responses to be paused, such a deadlock is prevented.

PDS requests from target to initiator are read responses carrying data that is larger than *Max_ACK_Data_Size*. To prevent deadlock, implementations MUST guarantee that return data carried in PDS requests from target to initiator can be accepted from the network unconditionally.

Implementation Note:
The requirement to unconditionally sink read data on lossless networks requires some pre-allocation of resources when the read request is issued. The acceptance and processing of the read data cannot be dependent on the ability to send an acknowledgement.

A single PDC mapped over lossless Ethernet traffic classes is illustrated in Figure 3-125. The PDC functions the same for both best-effort and lossless networks, with the only difference being the TC mapping for the PDS return data packets generated by the target.

Control packets are mapped onto TC_request and TC_response based on their type and whether they have a response.

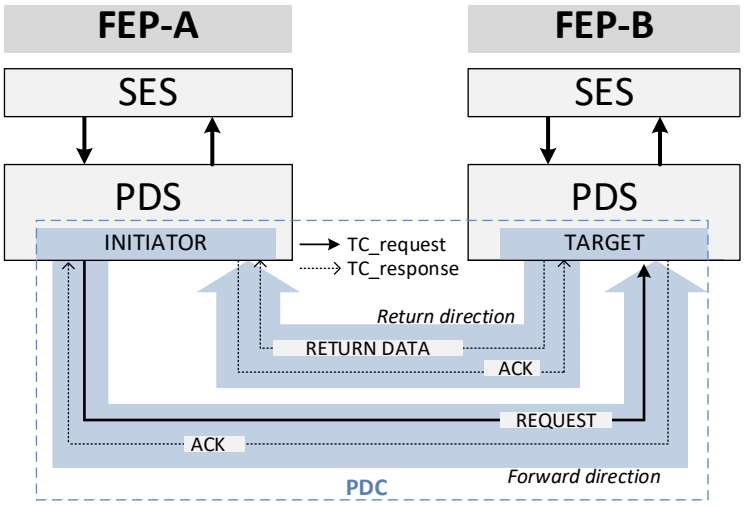


Figure 3-125 - PDC mapped over Lossless Ethernet: Traffic Class Mapping (ROD and RUD)

Table 3-88 - UET Control Packet to TC Mappings for Lossless Networks

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Control Packet	2 TC Mapping
NOOP	TC_request
ACK Request	TC_request
Clear Command	TC_request
Clear Request	TC_response
Close Command	TC_request
Close Request	TC_response
Probe	TC_request
Credit	TC_response
Credit Request	TC_request
Negotiation	TC_request

3.6.4.8 Congestion Signaling (CSIG)

Congestion Signaling (CSIG) is an in-band telemetry method that efficiently aggregates and reports per-switch congestion signals along a packet's path. CSIG carries an aggregated congestion signal as a tag inserted into each packet, updated by switches along its path to expose per-switch congestion state. When a CSIG-tagged packet arrives at its destination FEP, the signal value(s) are extracted and passed to the transport layer. If the transport congestion control algorithm requires feedback at the source, the destination encodes the extracted signal in a CSIG Reflection Header (3.6.9.4). This header is carried back to the source within an extended acknowledgement (ACK_CCX or NACK_CCX) generated by the transport layer. In this way, congestion information observed along the forward path is reliably returned to the sender for use in congestion control, load balancing, or other functions.

If a sender-based CC algorithm (such as NSCC) is in use at the transport layer, or if other sender-based functions (e.g., multi-path load balancing) need the information collected by the CSIG tag, the *CSIG-Reflect()* function at the transport layer MUST be enabled. The *CSIG-Reflect()* function populates the CSIG Reflection Header with information extracted by the *CSIG-Term()* function. The CSIG Reflection Header is returned to the source via an ACK or NACK packet generated by the transport. When *CSIG-Reflect()* is enabled, the PDS MUST generate extended acknowledgements containing up to 128 bits of congestion state (the ACK_CCX and NACK_CCX fields in ACKs and NACKs, respectively). The extended congestion state field (**pds.ack_ccx_state** or **pds.nack_ccx_state**) carries the CSIG Reflection Header back to the source FEP. Section 3.6.9.4 describes the format of the CSIG Reflection Header as well as the **pds.ack_ccx_state** and **pds.nack_ccx_state** formats used for CSIG Reflection.

Implementation of the *CSIG-Reflect()* function is REQUIRED in a CSIG-capable FEP and MUST be enabled when NSCC is enabled. *CSIG-Reflect()* MAY be enabled if NSCC is not enabled.

If a UET source FEP requires CSIG information to be reflected by the destination FEP, it SHOULD set the **pds.flags.ar** bit when inserting a CSIG tag in a packet.

When a CSIG-tagged packet arrives at a destination FEP whose *CSIG-Reflect()* function is enabled, if PDS processing at the destination FEP results in the sending of a positive acknowledgement back to the source, the PDS MUST generate an ACK_CCX packet. If the processing results in a negative

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

acknowledgement, the PDS MUST transmit a NACK_CCX packet. PDS and CMS populate the ACK_CCX or NACK_CCX packet with all the fields for a given CC algorithm. For ACKs, these are all the fields up to and including the **pds.sack_bitmap** field. If NSCC is enabled, the **pds.ccx_type** field is set to NSCC_CSIG and PDS/CMS populate the **pds.ack_ccx_state.rcvd_bytes, pds.ack_ccx_state.ooo_count, pds.ack_ccx_state.rc** and **pds.ack_ccx_state.rcvr_cwnd_pend** fields. When only RCCC is enabled, **pds.ccx_type** is set to RCCC_CSIG and PDS/CMS optionally populate the **pds.ack_ccx_state.credit** and **pds.ack_ccx_state.ooo_count** fields. CMS then takes the CSIG Reflection Header from the *CSIG-Reflect()* function and writes it into the lower 64 bits of the **pds.ack_ccx_state** field as shown in Figure 3-132 and Figure 3-133. For NACKs, PDS/CMS populate all the fields up to and including the NACK payload. They then take the CSIG Reflection Header from the *CSIG-Reflect()* function to construct the **pds.nack_ccx_state** field as shown in Figure 3-134.

When a packet without a CSIG tag arrives at a destination FEP whose *CSIG-Reflect()* function is enabled, processing and generation of ACKs and NACKs follows the procedures specified in 3.6.9.4. The types of ACKs and NACKs generated are as determined by PDS configuration and processing and can be ACK, ACK_CC, or NACK without extended CC state.

If **pds.flags.ar** is not set on the CSIG-tagged packet, the destination FEP may employ other implementation-specific policies to determine when to reflect CSIG information. For example, the FEP's policy may reflect only when the received signal value is over or under some configured threshold. Determining when to reflect CSIG information when CSIG-tagged packets arrive without **pds.flags.ar** set, or when ACKs are coalesced, is outside the scope of this specification.

3.6.4.8.1 Reflection Header in UET

The 64-bit CSIG Reflection Header is carried in the **pds.ack_ccx_state** and **pds.nack_ccx_state** fields of ACK_CCX and NACK_CCX packets, respectively. FEPs copy the tag data portion of the CSIG tag (2 B for the compact and 6 B for the wide CSIG tag, excluding the TPID) into the least-significant bytes of the CSIG Reflection Header, and they set the most significant 4 bits of the CSIG Reflection Header to indicate the CSIG tag type. Refer to CMS Section 3.6.9.4 for a description of the CSIG Reflection Header fields.

3.6.5 UET CC Protocol Operation Overview

The UET Congestion Control protocol, UET-CC, as defined in this specification, provides congestion management on a best-effort Ethernet network.

3.6.5.1 Types of Congestion [Informational]

Congestion control is a key component required to enable best-effort network operation in large datacenter networks. While PFC is useful for certain use cases, large-scale PFC deployments suffer from various pathologies that are well documented in the literature [15], [1]. UET can operate with both lossless and best-effort networks; this specification refers to best-effort networks, so avoids these issues by design.

As congestion has different causes, congestion control requires different mechanisms that work in conjunction to ensure predictable high network performance. Compared to the internet, back-end datacenter networks have much lower and more predictable latency, higher bandwidth for end hosts,

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

and many more available paths between a source and a destination. UET is optimized for back-end datacenter networks – an area targeted traditionally by HPC networks. UET is aimed at AI and HPC workloads, which commonly consist of synchronized collectives that aim to run on a network with many redundant paths at close to 100% load, while caring mostly about the final completion time within a group collective. Congestion control algorithms designed for other contexts often perform poorly in such environments.

UET-CC is intended to be used in networks with topologies such as fat trees where, in the absence of failures, there are many routed paths from a source to a destination, and they are all of equal length and capacity. Future UET-CC specifications may extend CC to other network topologies.

There are five main causes of congestion in datacenter networks:

1. Network congestion localized to a subset of switch-to-switch links in the topology, mainly due to improper load balancing (e.g., ECMP hash collisions leading to multiple flows being placed on the same paths) or failures or interference from ordered traffic.
2. Overall network congestion when most or all available paths between two end hosts are congested; for instance, in the case where the network core is under-provisioned for the traffic it carries.
3. Outcast when the source apps are trying to send to many destinations simultaneously and thus generating more traffic than the source's connection to the network can manage.
4. Incast when many sources are sending to the same destination at the same time.
5. Local congestion at the destination when accessing memory.

HPC networks have traditionally relied on link-level flow control to provide a lossless L2 implementation together with simple end-to-end flow-control algorithms as adopted in RDMA over Converged Enhanced Ethernet[10] (also known as RoCE). Single-path transports (e.g., RoCE/TCP) running traditional congestion control algorithms such as DCTCP [17], DCQCN [18], or Swift [19], coupled with ECMP flow hashing, is the standard for Ethernet-based datacenter congestion control today. However, many research works have shown this approach deals poorly with flow collisions ((1) in the list above). Instead, when the transport actively sends packets from each flow across multiple paths, collisions are reduced (see MPTCP [29], NDP [31], Homa [20], pHost [21], MP-RDMA [22]), network utilization increases, and flow completion times become low and predictable.

Instead of relying on end-to-end multipathing, HPC networks have traditionally used adaptive packet routing (see [24], [25], and [26]) with extensions such as fine-grained L2 flow-control for traffic isolation [27]. UET specifies ECMP flow hashing schemes for load balancing but, at the same time, allows vendors to enhance networks with compatible but proprietary features such as adaptive packet spraying or fine-grained end-to-end flow control. Those are outside the scope of this specification.

[10] Converged Enhanced Ethernet (CEE) is specified in IEEE Std 802.1Q

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

3.6.5.2 Adoption of Multipath

The PDS adopts multipath packet transmission at its core. The approach the PDS adopts is to use packet spraying: Each packet is placed onto a pseudo-randomly chosen path by selecting an appropriate entropy value and placing it in the packet. This unifies the best of HPC's packet-adaptive routing with Ethernet's flow hashing. The PDC/CCC uses a configurable and typically large number of entropies (e.g., 64-256) and essentially round robins between them; however, when feedback packets indicate that the path associated with a given entropy is congested (e.g., via an ECN mark or a trimmed packet), the source may decide to reduce the number of packets placed on that path. This is similar to packet-spraying solutions in datacenters [28] and source-adaptive routing [26]. Details are provided in Section 3.6.16.

Informative Text:
An alternate approach, not taken by the PDS, is typified by MPTCP[29]. This approach uses a relatively small number of paths (8-16) and keeps state for each path, including congestion state and per-path sequence numbers. This approach permits better loss detection per sub-flow but requires more state to implement. It also creates an upper bound on the number of usable paths due to the size of the window, which results in poorer load balancing in certain cases, and so reduced throughput.

3.6.5.3 Window Based Operation

Multipathing helps avoid flow collisions, but congestion can still appear for at least the following reasons:

1. Oversubscribed networks.
2. Outcast traffic patterns at the source.
3. Incast traffic patterns at the destination.

To deal with this, the PDS limits the amount of data in flight between any source and destination.

Three congestion control mechanisms are specified:

1. **Network-signal Congestion Control (NSCC):** NSCC maintains an explicit congestion window at the source. The source PDS estimates packets in flight. When the congestion window is larger than the number of inflight bytes in packets sent but not yet acknowledged, NACKed, or inferred to be lost (timed out or other mechanism), then the source can send. The source PDS considers packets to have left the network when they are (a) acknowledged by the destination; (b) negatively acknowledged by the destination; or (c) inferred by the source to be lost. The source adjusts the window size based on congestion feedback from the destination.
2. **Receiver-credit Congestion Control (RCCC):** An RCCC sender maintains a pool of credit. It can send whenever there is credit in its credit pool. As it uses credit, the source conveys requests for more credit to the destination via the credit target field in data packets or in Credit Request CPs and receives credit back from the destination in Credit CPs or ACK packets. In the steady state, the amount of credit flowing between the destination and the source constitutes an implicit window. The destination paces out the sending of credit to all its sources so that in aggregate their rate does not cause congestion at the destination's incoming link(s). Thus, the size of the

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

credit window between a single source and its destination varies depending on how many sources are sending to that destination.

3. **Transport Flow Control (TFC):** TFC maintains a pool of credit. It can send whenever there is credit in its credit pool. As it uses credit, the source may convey requests for more credit to the destination via the credit target field in data packets. The source receives credit back from the destination in either Credit CPs or ACKs. The destination sends credit based on available receive buffer resources to avoid packet drops. This mechanism differs from RCCC in that it is intended for point-to-point services with limited buffering and low tolerance for packet loss. Multipath may or may not be used with TFC.

Note that the window-based approaches are in contrast to a rate-based approach, such as RoCE/DCQCN. The window-based approach is preferable because of its inherent stability due to packet clocking: In severe congestion events, a window-based source will stop transmission as it waits for explicit confirmation that packets have left the network. In contrast, in a rate-based approach such as DCQCN, lack of feedback is implied to be a sign of appropriate network operation, and the rate is increased. This approach works in lossless networks, where PFC kicks in as a backstop, but in best-effort networks where packets may be silently discarded due to congestion, increasing throughput in response to a lack of congestion feedback results in poor performance.

3.6.5.4 UET CC Startup Behavior

In the case of NSCC, the congestion window is dynamically adjusted by the congestion control algorithm; it is increased when there is no congestion to discover and use spare network capacity and it is decreased to cope with congestion. Traditionally, protocols such as TCP would start with a conservative window (e.g., 10 packets) and then increase until congestion is detected. This ensures that flows can reach a steady state after a number of round-trip times.

In the context of datacenters, there are at least three reasons that the standard congestion control approach is suboptimal:

1. Many flows are around one bandwidth-delay product (BDP) or less, meaning that they can never reach steady state.
2. The network is well provisioned meaning that, in many cases, flows can operate at line rate.
3. The BDP of the idle network is well known.

(3) implies that the maximum size of the window (*MaxWnd*) can be restricted to around 1.5 BDPs. This allows enough for a source to fill the path to a destination (requiring 1 BDP with no queues) and for just enough queueing to ensure high multipath utilization. There is no reason to have a window larger than this bound.

(1) and (2) taken together imply that new flows should, by default, start at line rate, using a window around *MaxWnd*. The alternative – starting with a smaller window and dynamically increasing it in the absence of congestion – would inflate the flow completion time of small flows.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Starting at line rate will work well most of the time, when the destination is idle and when there is no overall network congestion. The biggest risk when starting at line rate is for the destination to be already receiving other flows, such that its link is already saturated. In such cases, many packets will be lost. The difficulty in such cases is knowing quickly which packets need retransmission and adapting the window to enable safe and fast retransmission. When available, packet trimming enables accurate and fast loss detection.

Finally, outcast is a case where a source has many active PDCs to different destinations, and the PDCs are bottlenecked by the source's link(s). UE FEPs include a scheduler that determines which PDC sends when, so that data is generated for sending at no more than the source's *linkspeed*. The main consideration here is that during an outcast, the scheduler rather than the CC window may be what limits transmission. Under such circumstances, the default initial window may be much larger than can be used, and so care must be taken when initializing the control loop.

After the initial burst of packets, two different approaches for adjusting the window of in-flight data are permitted: one where the destination dictates how the window evolves based on explicit demand, and one where the source controls the window based on feedback from the destination and the network. The sender- and receiver-driven approaches behave similarly in the first RTT, starting at line rate by default and spraying packets across paths to avoid collisions. They differ, however, in their handling of incast and overall network congestion.

RCCC addresses incast at the receiver by tracking the unsatisfied demand (the amount of unsent data, or backlog) from each source. The receiver gives credit in a round robin (or any other sharing approach) to each source to avoid congesting its incoming network links. The source can measure unsatisfied demand by counting the number of outstanding bytes yet to be sent for the messages it is currently processing for any given target. In oversubscribed networks, RCCC needs an additional mechanism to deal with core congestion.

NSCC addresses incast and overall congestion with a single mechanism. Its incast handling does not directly take into account what the destination knows about the incast traffic, but it does not need an additional mechanism to cope with the other types of congestion.

Both sender- and receiver-driven approaches need to take care in outcast scenarios, where neither the network nor the destination is the bottleneck. In this scenario, network or destination feedback might suggest the absence of congestion. However, the FEP cannot transmit at line rate to any specific destination due to the source scheduler dividing the link's capacity among multiple PDCs destined for various endpoints. CC algorithms need to take care not to use this lack of negative feedback as an indication that they should increase the sending window, as this can lead to unnecessary congestion when the outcast ends and the CC algorithm then sends at an unreasonable rate.

In this document we present algorithms for both receiver- and sender-driven approaches. TFC is described in Section 3.6.15.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

3.6.6 Congestion Control Context (CCC)

Between two FEPs, one or more PDCs may be established using the same traffic class for transmitted data. These PDCs are called a "PDC group." A congestion control context (CC context or CCC) controls the admission of data packets into the network for a PDC group.

A CCC controls the admission of data packets into the network in a single direction; thus, for bidirectional data transmission two CCCs are used, one controlling admission in each direction. The CCC utilizes control packets in the reverse direction to determine the fate of those data packets, and so adapt its admission decisions for future data packets. Reverse-path PDS ACK packets and control packets are not congestion controlled and SHOULD use a separate traffic class and queue from data traffic (Section 3.6.4.7). The reverse CCC may need to factor the bandwidth used by ACK packets into its admission decision.

As more than one PDC can be active simultaneously within the same CCC, a congestion control algorithm cannot assume that all data packets handled by a CCC share one sequence number space. Rather, a CCC needs to keep track of the total number of bytes in packets it admits to the network and updates that byte count when packets are acknowledged, NACKed, or when it is determined that the packet has, with high likelihood, been lost (referred to as an inferred loss). Conceptually, the CCC keeps track of state by keeping records of sequence numbers of data packets and their responses, timeouts, etc., but in practice the PDC and CCC are unlikely to be implemented in a manner that replicates this state in both PDC and CCC. For simplicity of explanation, it is assumed that the CCC sees reverse path control packets and updates its own state accordingly.

When multiple PDCs transfer data in the same direction from the same sending FEP to the same receiving FEP using the same traffic class, those PDCs SHOULD be mapped to a common PDC group and SHOULD share a CCC. This ensures the CCCs do not unnecessarily compete with one another, and that load balancing of the aggregate traffic flow is performed effectively. This applies to PDCs transferring data in the forward direction as well as those transferring data in the reverse direction. That is, read response data mapped onto a reverse direction PDC from FEP-B to FEP-A SHOULD be controlled by the same CCC as write request data mapped onto a forward direction PDC from FEP-B to FEP-A. Two RUD PDCs using different CCCs would still be expected to converge, but would be less optimal.

UET permits multiple PDC groups with each group using a different (set of) traffic class(es). When two PDC groups transfer data packets from the same source to the same destination (using different traffic class mappings), a separate CCC MUST be used for each PDC group. This is because congestion measured on one traffic class cannot safely be used to infer congestion on another traffic class.

3.6.7 CCC for ROD PDCs

The ROD delivery mode provides reliable ordered delivery. UET achieves this by sending all the packets of a ROD PDC via the same network path. ROD flows SHOULD use one CCC per PDC, not shared with other ROD flows or with RUD flows. This ensures that when a single-path ROD PDC encounters a flow collision within the network, other ROD PDCs and RUD PDCs sending to the same destination FEP are not adversely affected by congestion state of a flow that may have taken a different network path.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Multiple CCCs between the same FEPs using the same traffic class MAY share some state, such as a baseline estimate for RTT.

3.6.7.1 CCC for RUDI & UUD

RUDI and UUD do not support UET CC. Neither uses an ACK packet to carry the required congestion control state. The rationale is that these will typically be used when sending small messages to large numbers of destinations. Closed-loop congestion control for a single-packet PDC is generally not meaningful, as there is no flow to control.

When RUDI and/or UUD is used to carry significant bandwidth, these SHOULD NOT be mapped to the same traffic class as UET CC traffic.

3.6.8 Source Context

Each CCC gates admission of packets to the network for the corresponding PDCs. Each CCC does this using feedback from the network and the destination, and it uses this feedback to control the aggregate amount of unacknowledged data permitted to enter the network using that CCC. Multiple CCCs can co-exist at one source. This can happen when each CCC controls traffic to a different destination, when CCCs to the same destination use different traffic classes, when PDCs to the same destination mix single-path ROD and RUD traffic, or when there are multiple single-path ROD PDCs to the same destination. When multiple CCCs simultaneously allow data to be sent, it is the role of a scheduler in the sending FEP to determine the order that PDCs may send, and how the transmission capacity is allotted among them. Scheduler behavior is implementation specific.

With protocols such as TCP, if multiple connections have sufficient congestion window to send, it is common for them all to do so until their windows fill. If their aggregate rate exceeds the link speed, they can then build a substantial output queue downstream of the congestion control gate. With UET, it is NOT RECOMMENDED to allow PDCs to build a significant output queue, though some small amount of queuing downstream of the CCC gate may be needed by implementations.

For example, when a CCC is limited by a window and an ACK arrives that causes that CCC to extend the window, the CCC would then normally cause a packet to be transmitted (this is known as ACK-clocking and is a desirable property for congestion control). However, other CCCs at the same source may be in the same situation, and so it is possible for multiple CCCs to allow transmission simultaneously. The source's scheduler must keep track of which CCCs currently permit transmission and schedule transmission from PDCs under the control of these CCCs. When multiple CCCs or PDCs within a CCC have data available to send, the source's scheduler selects an eligible PDC in an implementation-specific manner.

Conceptually then, the following steps occur:

1. The CCC determines whether any of its PDCs may be scheduled.
2. The scheduler then decides which PDC to send next from those that are schedulable.
3. On the selected PDC, a packet is generated to be sent. This may be new data or a retransmission.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

4. The CCC is contacted again to finalize fields such as the EV, which require late binding because CCC state may have changed since the CCC became ready to send.

The interactions between the reliability module, its scheduler, and the CCC are described in more detail in Section 3.6.8.1.

3.6.8.1 Abstract interface between PDS and Congestion Control

UET PDS and congestion control is typically implemented in the same device in an integrated manner, but for the purposes of specification, they are defined as distinct modules separated by an abstract API. An optimized implementation is not required to implement this API, provided that its externally observable behavior is indistinguishable from the behavior specified by this abstract API.

A CCC generally does not need to know of the existence of a particular PDC, or of the transmission or acknowledgment of a packet with a specific sequence number. Rather, the PDC tracks the transmission of packets and their acknowledgment. We define a set of abstract events that communicate state changes between the PDS and the CMS. The API is assumed to be stateful. The PDS keeps track of which CCC is associated with each PDC, whereas the CCC does not know about PDCs.

3.6.8.1.1 Source API

The PDS MUST allocate a CCC whenever the first PDC to a destination is created, and MAY deallocate it when the last PDC to a destination is destroyed. A CCC MAY be maintained after the last PDC is destroyed to cache congestion state (but see Section 3.6.14.1 regarding closing RCCC CCCs and Section 3.6.15.3 regarding closing TFC PDCs).

- **`AllocateCCC`**`(bool:sprayed)`

  The CMS will create a new CCC.

  The parameter *sprayed* is TRUE if the CCC will handle sprayed traffic, and FALSE if it will handle single-path traffic. The PDS SHOULD map multiple sprayed PDCs in the same traffic class to the same destination to the same CCC. This is usually transparent to congestion control.

- **`DeallocateCCC`**`()`

  The CMS will deallocate resources associated with the CCC.

When multiple PDCs have data to be sent, the source uses a scheduler to determine access to the outgoing link(s). Notionally, a PDC can be scheduled when it has new data to send or data to retransmit, and its associated CCC is in the READY state.

A CCC can be in one of four states:

- A CCC is READY if there is data (new or retransmissions) ready to be sent by any of its PDCs and the CCC will permit sending at this time.
- A CCC is ACTIVE if there is data not yet sent by any of its PDCs, or data queued for retransmission by any of its PDCs, but the CCC will not permit sending.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

- A CCC is PENDING if it is not active, but there is data that has been sent by any of its PDCs that is not yet acknowledged.
- A CCC is IDLE if it is not ready, active, or pending.

The state machine in Figure 3-126 indicates the transitions between the four states.

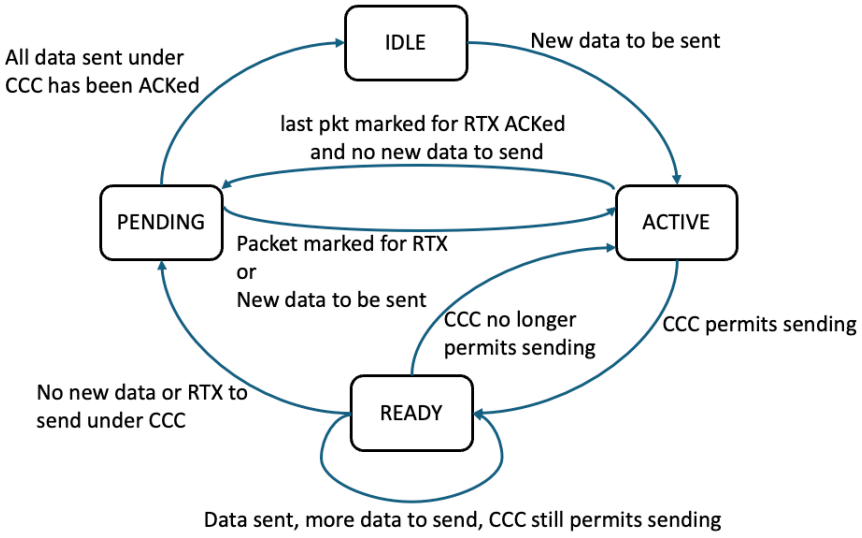


Figure 3-126 - CCC State Machine

To summarize the state transitions:

- A CCC transitions from IDLE state to ACTIVE when new data arrives to be sent on any PDC associated with the CCC.
- A CCC transitions from ACTIVE to READY state if the CCC's congestion control algorithm permits sending. For window-based control, this is when there is space in the source's *cwnd*. For destination credit control, this is when the source has sufficient credit. If both are used, both conditions must be satisfied.
- A CCC transitions from READY back to ACTIVE if the CCC's congestion control algorithm no longer permits sending. This typically occurs when data has been sent, filling the *cwnd* or consuming credit. It can also occur when sending does not occur, if an ACK or NACK arrives or a timeout occurs that causes the *cwnd* to be reduced.
- When in READY state, if data is sent but the CCC still allows sending, the CCC stays in READY state.
- A CCC transitions from READY to PENDING state when no PDC associated with the CCC has any more new data to send or packets marked for retransmission.
- A CCC transitions from PENDING to IDLE state when all data has been ACKed on all PDCs associated with the CCC. A CCC MUST NOT be removed except when in IDLE state, unless all PDCs associated with the PDC have been declared failed.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

- A CCC transitions from PENDING to ACTIVE state if a packet sent on a PDC associated with the CCC is marked for retransmission. This can be caused by receiving a NACK, inferring loss from a SACK, or due to a retransmission timeout.
- A CCC transmissions from ACTIVE to PENDING state if there are no new data packets to send, and the last data packet marked for retransmission is acknowledged.

Note that multiple transitions may happen consecutively triggered by a single state change. For example, a CCC may immediately transition from IDLE to ACTIVE to READY if new data arrives to be sent and the CCC is already in a state that permits sending.

The scheduler will rotate between PDCs in READY state, sending packets from them in turn. How the scheduler chooses which PDC to service next is implementation specific.

Packet arrivals or timer events trigger actions in the source CCC behavior. These actions are:

- **OnACK():**An ACK packet arrived for a PDC under this CCC.
- **OnNACK():**A NACK packet arrived for a PDC under this CCC. The PDS indicates if a PDC is fully established (i.e., a PDCID was returned from the remote device).
- **OnInferredLoss():**A PDC retransmit timer expired for a PDC under this CCC, or loss was otherwise inferred from received SACKs. The PDS indicates if a PDC is established.
- **OnNewData():**New data was queued for a PDC under this CCC.
- **OnCreditUpdate():**New credit arrived for the CCC.
- **OnSend():**The scheduler sent a packet from a PDC under this CCC.

Each of these actions can cause the state machine for the CCC to move between the READY/ACTIVE/PENDING/IDLE states. These actions do not directly cause data packets to be sent; rather, the CCC moves to the READY state and then data may be sent on any corresponding PDC when the scheduler determines it is time for that PDC to send. These actions may, however, cause control packets to be sent.

In addition, when the scheduler is ready to send using a PDC, it calls *GetSendParams()* on the CCC, which returns the CC-related fields for the packet. It is not desirable, or even possible, to compute all packet fields at the time that a CCC becomes ready. One of these fields is the *Entropy* value (EV) (3.6.10.1) that determines the packet's path. For optimal path selection, it is preferable to determine this when the packet is finally going to be sent, because incoming ACKs and NACKs arriving between the CCC being ready and the packet being sent may change the choice of path/EV or may change which PDC under that CCC is selected for transmission.

When supporting rendezvous, PDS can query CMS at the behest of SES. When PDS calls *GetEagerEstimate(),* CMS returns the current value of *cwnd*. This is passed through PDS to SES to be used to set the size of the eager portion of the message.

The CCC actions discussed in this section are further described in pseudocode in Section 3.6.12. Before the actions are called, some data needs to be prepared. This preparation is detailed in Section 3.6.10.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

3.6.9 UET-CC Header Formats and Fields

The congestion control state is carried in PDS headers. The following fields are defined in this section:

- **pds.req_cc_state**
- **pds.ack_cc_state**
- **pds.ack_ccx_state**
- **pds.nack_ccx_state**

The **pds.req_cc_state** field is used for RCCC and TFC. The **pds.ack_cc_state** field is used by NSCC, RCCC, and TFC.

The **pds.ack_cc_state** field is 64 bits when using ACK_CC and is 128 bits when using ACK_CCX (extended length). The currently defined formats for NSCC, RCCC, and TFC use **pds.type** = ACK_CC and **pds.cc_type** = CC_NSCC or CC_CREDIT. The field setting **pds.cc_type** = CC_NSCC is used whenever NSCC is enabled.

3.6.9.1 pds.req_cc_state—RCCC and TFC

This 32-bit field is carried in the PDS request packets. The bits are globally configured; there is no type field in the header to enable parsing. One type is defined for RCCC and TFC, as illustrated in Figure 3-127, to carry the **pds.req_cc_state.credit_target** field as well as the source's CCC context identifier (**pds.req_cc_state.ccc_id**).

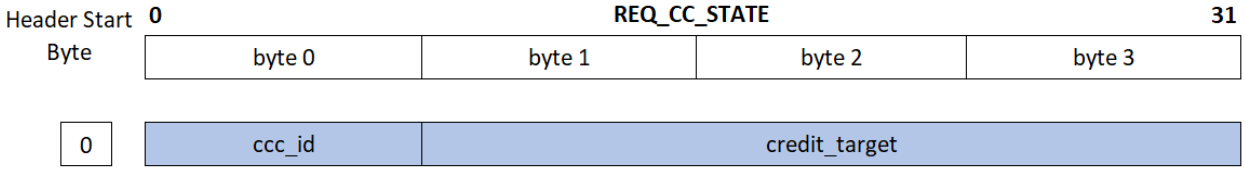


Figure 3-127 - pds.req_cc_state Header Format for RCCC / TFC

Table 3-89 - Header Fields for pds.req_cc_state for RCCC/TFC

Field Name	Field Size (in bits)	Field Description
ccc_id	8	Identifier assigned by source
credit_target	24	Estimate of the amount of data the source CCC has to send • Units: 256 bytes • Wraps to 0x0 at 0xFFFFFF

pds.req_cc_state.ccc_id is defined in Section 3.6.14.1 and is used to identify the source's CCC context to the destination.

The **pds.req_cc_state.credit_target** field represents the cumulative number of bytes at the source that have been made ready to transmit in 256 B units. The amount of work is the sum of all work ever posted to the PDCs sharing the CCC. When the source increases **pds.req_cc_state.credit_target**, this serves as a request to the destination for more credit. A CC algorithm MAY restrict how quickly the

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

pds.req_cc_state.credit_target field increases (as opposed to setting **pds.req_cc_state.credit_target** directly to the total amount of data to be transmitted) to avoid excessive credit being sent to a source that cannot immediately use it.

The **pds.req_cc_state.credit_target** field is defined in Section 3.6.14

3.6.9.2 pds.ack_cc_state for NSCC

NSCC uses the **pds.ack_cc_state** field when **pds.type** = ACK_CC and **pds.cc_type** = CC_NSCC. The format is shown in Figure 3-128.

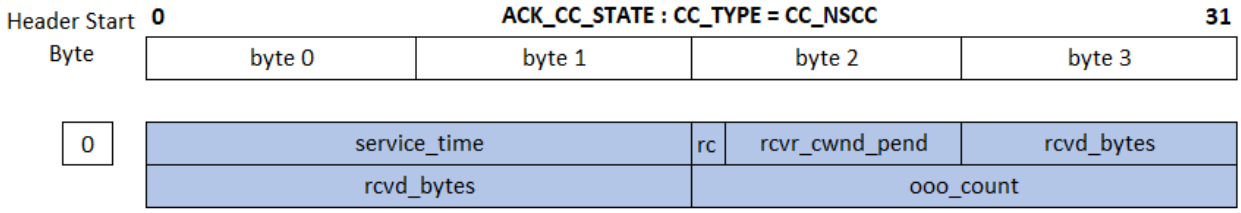


Figure 3-128 - pds.ack_cc_state for UET TYPE = ACK_CC and CC_TYPE = CC_NSCC

Table 3-90 - pds.ack_cc_state for ACK_CC with CC_TYPE = CC_NSCC

Field Name	Size (in bits)	Field Description
service_time	16	Service time at destination, measured from the time the PDS request packet arrives at the Ethernet MAC to the time a PDS ACK packet is transmitted by the Ethernet MAC • Units: 128 nsec • Accuracy target: ~500 nsec 0xFFFF indicates service time exceeded 8.38848 ms
restore cwnd (rc)	1	Indicates *cwnd* should be restored after destination congestion ends
rcv_cwnd_pend	7	This field indicates the congestion level at the destination
rcvd_bytes	24	Per PDC received byte count, maintained at the destination and incremented when a network packet is accepted • Units: 256 bytes • Wraps to 0x0 at 0xFFFFFF
ooo_count	16	Count of out of order packets

3.6.9.2.1 pds.ack_cc_state.rcvd_bytes for NSCC

This field represents the number of bytes the destination has accepted on the PDC in 256 B units. It is a 24-bit field that wraps.

The algorithm to calculate **pds.ack_cc_state.rcvd_bytes** at the destination is in Section 3.6.13.9.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

For use of **pds.ack_cc_state.rcvd_bytes** at the source, refer to Section 3.6.10.3.

3.6.9.2.2 pds.ack_cc_state.rcv_cwnd_pend and pds.ack_cc_state.rc for CC_NSCC

The **pds.ack_cc_state.rcv_cwnd_pend** field is used by the destination to request that the source reduces its congestion window because the destination is becoming backlogged. The **pds.ack_cc_state.rc** field indicates whether the source should restore the original value of the congestion window when flow control ends. Use is described in Section 3.6.13.2.

3.6.9.2.3 pds.ack_cc_state.ooo_count for NSCC

The **pds.ack_cc_state.ooo_count** field MAY be used to trigger an early loss detection event at the source as described in the PDS Section 3.5.15.4.1. The **pds.ack_cc_state.ooo_count** field is an instantaneous estimate of the number of out-of-order packets, and its value goes up and down. It is not cumulative, and the field does not wrap. While **pds.ack_cc_state.ooo_count** is carried in the **pds.ack_cc_state** field, its value is not used by the congestion management algorithms.

The **pds.ack_cc_state.ooo_count** field is a count of all packets received on each PDC that are out of order, meaning at least one packet with a lower PSN has not yet been received. If out-of-order count is implemented on a PDC, a 16-bit **pds.ack_cc_state.ooo_count** for that PDC MUST be supported. This covers the maximum range of active PSNs on a PDC, which is –32640 as defined in 3.5.11.13

Support for setting **pds.ack_cc_state.ooo_count** is optional; if a destination does not calculate the out-of-order count, then **pds.ack_cc_state.ooo_count** MUST be set to 0xFFFF to indicate the field is invalid.

3.6.9.2.4 pds.ack_cc_state.service_time for NSCC

This field is carried in the ACK_CC when **pds.cc_type** = CC_NSCC. The **pds.ack_cc_state.service_time** field is used to adjust the RTT calculation in NSCC, ensuring it accounts for processing delays at the destination FEP. Refer to Section 3.6.4.2 for usage of this field.

Service time is defined as the time when the first byte of the associated ACK is transmitted by the destination's MAC minus the time the first byte of a packet triggering the ACK arrived at the destination's MAC. This time may be estimated, e.g., the packet may be buffered after the transmit time is captured, with a goal of providing ~500 nsec accuracy. The **pds.ack_cc_state.service_time** field is represented using 128 nsec granularity such that the least significant bit represents 128 nsec.

When set to 0x0, **pds.ack_cc_state.service_time** is not valid and MUST be ignored. The encoding of 0x0 simplifies the process of ignoring service time for many of the calculations.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

3.6.9.3 pds.ack_cc_state for TFC and RCCC

TFC and optionally RCCC, when NSCC is not enabled, use **pds.ack_cc_state** when **pds.type** = ACK_CC and **pds.cc_type** = CC_CREDIT. The format is shown in Figure 3-129.

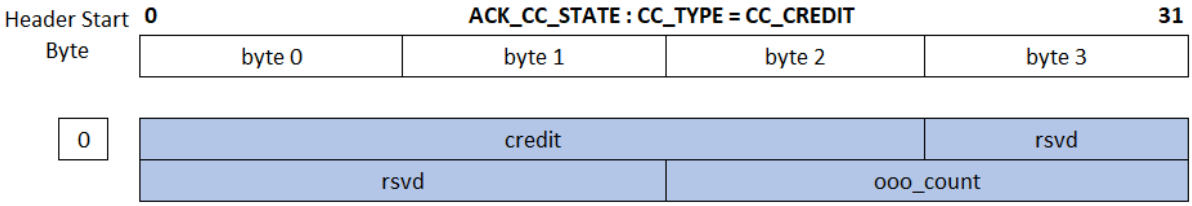


Figure 3-129 - pds.ack_cc_state for UET TYPE = ACK_CC and CC_TYPE = CC_CREDIT

Table 3-91 - pds.ack_cc_state for ACK_CC with CC_TYPE = CC_CREDIT

Field Name	Size (in bits)	Field Description
credit	24	Credit, allocated by the destination (units: 256 bytes)
ooo_count	16	Count of out of order packets (units: packets)

3.6.9.3.1 pds.ack_cc_state.credit for TFC and RCCC

This field is carried in the ACK_CC when **pds.cc_type** = CC_CREDIT. It is used to provide a credit-based flow control between two FEPs.

The **pds.ack_cc_state.credit** field allows the destination to indicate to the source how much additional data can be sent on this CCC. The source MUST stop transmitting when credit is exhausted. The source MUST NOT transmit more packets than are allowed by the credit allocation (e.g., do not go below zero when tracking credit).

The **pds.ack_cc_state.credit** field is in units of 256 bytes and is a cumulative field that wraps. It indicates the total amount of credit allocated by the destination to the CCC since it started (this excludes any speculative credit the source started with when the CCC was established, but that was not allocated by the destination). The RCCC and TFC pseudocode describe how the credit is used at the source. When NSCC is enabled, credit is returned in Credit CPs (see Section 3.5.16.6).

3.6.9.3.2 pds.ack_cc_state.ooo_count for TFC and RCCC

The **pds.ack_cc_state.ooo_count** field when TFC or RCCC is enabled serves the same purpose and is processed the same as when NCSS is enabled. See Section 3.6.9.2.3.

3.6.9.4 CSIG Reflection Headers in pds.ack_ccx_state and pds.nack_ccx_state

The 64-bit CSIG Reflection Headers are carried in the **pds.ack_ccx_state** and **pds.nack_ccx_state** fields of ACK_CCX and NACK_CCX packets, respectively. The format of the CSIG Reflection Header depends on the size of the CSIG tag data being reflected. The format of the CSIG Reflection Header for a compact CSIG tag is shown in Figure 3-130 and the header fields are defined in Table 3-92. The format of the CSIG Reflection Header for the wide CSIG tag is shown in Figure 3-131 and the header fields are defined in

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Table 3-93. FEPs copy the tag data portion of the CSIG tag (2 B for the Compact and 6 B for the wide CSIG tag, excluding the TPID) into the least significant bytes of the CSIG Reflection Header, and set the most significant 4 bits of the CSIG Reflection Header to indicate the type. Type CSIG_COMPACT indicates signal data from a compact CSIG tag and type CSIG_WIDE indicates data from a wide CSIG tag.

Table 3-92 - CSIG Reflection Header Fields for Compact CSIG Tag

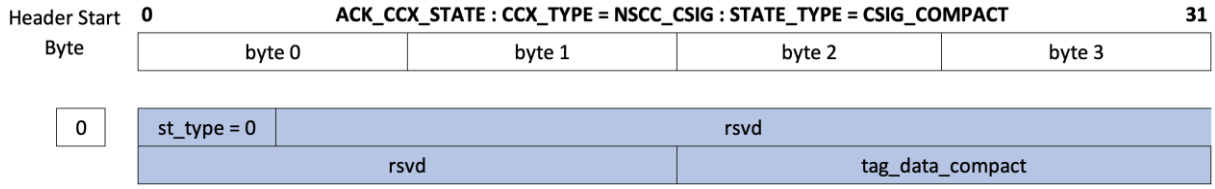


Figure 3-130 - CSIG Reflection Header for Compact CSIG Tag

Field Name	Field Size (in bits)	Field Description
st_type	4	Defines the contents of the CSIG reflection header. Set **csig.st_type** = 0 for the CSIG_COMPACT format.
rsvd	44	Reserved
tag_data_compact	16	Tag data from Compact CSIG tag

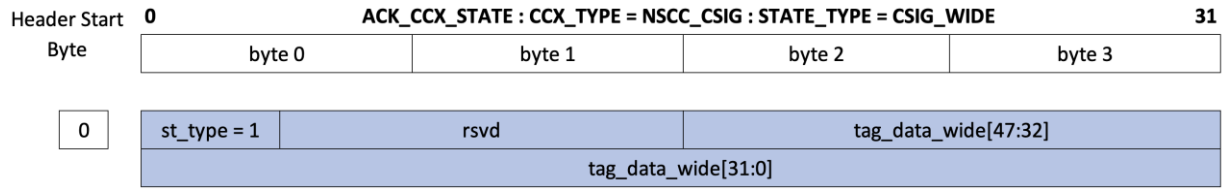


Figure 3-131 - CSIG Reflection Header for Wide Compact CSIG Tag

Table 3-93 - CSIG Reflection Header Fields for Wide CSIG Tag

Field Name	Field Size (in bits)	Field Description
st_type	4	Defines the contents of the CSIG reflection header. Set **csig.st_type** = 1 for the CSIG_WIDE format.
rsvd	12	Reserved.
tag_data_compact	48	Tag data from wide CSIG tag

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

3.6.9.5 pds.ack_ccx_state - NSCC_CSIG

UET TYPE = ACK_CCX and CCX_TYPE = NSCC_CSIG are defined for NSCC with CSIG enabled. The format of this variant of ACK_CCX_STATE is shown in Figure 3-132 and the field definitions are shown in Table 3-94.

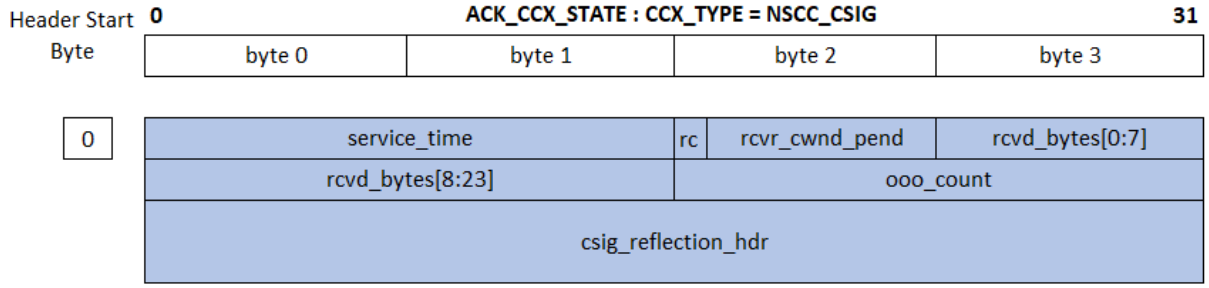


Figure 3-132 - pds.ack_ccx_state for UET_TYPE = ACK_CCX and CCX_TYPE = NSCC_CSIG

Table 3-94 - pds.ack_ccx_state for ACK_CCX and CCX_TYPE = NCSS_CSIG

Field Name	Size (in bits)	Field Description
service_time	16	same as ACK_CC_STATE for CC_TYPE = CC_NCSS
restore cwnd (rc)	1	same as ACK_CC_STATE for CC_TYPE = CC_NCSS
rcv_cwnd_pend	7	same as ACK_CC_STATE for CC_TYPE = CC_NCSS
rcvd_bytes	24	same as ACK_CC_STATE for CC_TYPE = CC_NCSS
ooo_count	16	same as ACK_CC_STATE for CC_TYPE = CC_NCSS
csig_reflection_hdr	64	Compact or wide-format CSIG Reflection Header

3.6.9.6 pds.ack_ccx_state - RCCC_CSIG

UET TYPE = ACK_CCX and CCX_TYPE = RCCC_CSIG are defined for RCCC with CSIG enabled. The format of this variant of ACK_CCX_STATE is shown in Figure 3-133 and the field definitions are shown in Table 3-95.

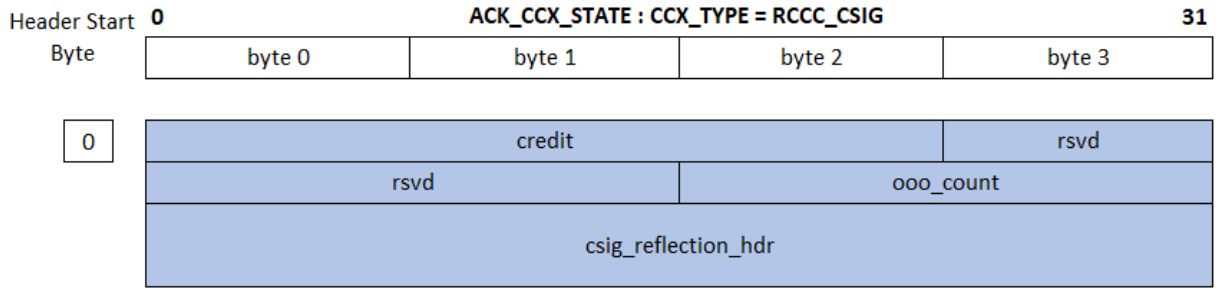


Figure 3-133 - pds.ack_ccx_state for ACK_CCX and CCX_TYPE = RCCC_CSIG

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Table 3-95 - pds.ack_ccx_state for ACK_CCX with CCX_TYPE = RCCC_CSIG

Field Name	Size (in bits)	Field Description
credit	24	same as ACK_CC_STATE for CC_TYPE = CC_CREDIT
rsvd	24	Reserved
ooo_count	16	same as ACK_CC_STATE for CC_TYPE = CC_CREDIT
csig_reflection_hdr	64	Compact or wide-format CSIG Reflection Header

3.6.9.6.1 pds.nack_ccx_state - CSIG

UET TYPE = NACK_CCX and NCCX_TYPE = CSIG are defined for CSIG. The format of this **pds.nack_ccx_state** variant is shown in Figure 3-134 and the field definitions are shown in Table 3-96.

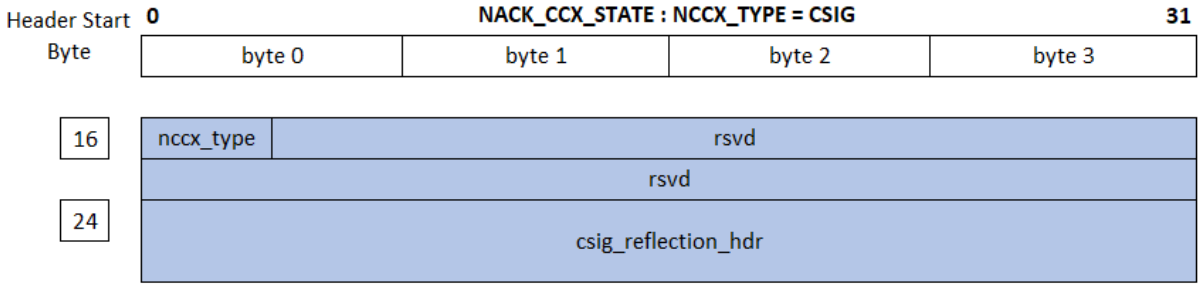


Figure 3-134 - pds.nack_ccx_state for NCCX_TYPE = CSIG

Table 3-96 - pds.nack_ccx_state for NCXX_TYPE = CSIG

Field Name	Size (in bits)	Field Description
nccx_type	4	Defines the contents of the NACK_CCX_STATE. Set pds.nack_ccx_state.nccx_type = 0 for CSIG.
rsvd	60	Reserved
csig_reflection_hdr	64	Compact or wide-format CSIG Reflection Header

3.6.10 Common Congestion Control Event Processing

The pseudocode and associated descriptive text in the congestion management sections below use the following conventions: raw fields extracted from packets are indicated with their header name (e.g., pds.cc_type) and when referenced in text they are highlighted in bold font. Variable names prefixed `ccc` are state that is local to the CCC, and variable names prefixed `pdc` are state that is specific to that PDC. Working variables are highlighted in descriptive text using italic font (e.g., *Entropy*). Comments and inline explanations are shown as "#comment" to distinguish them from English language pseudocode (in fixed-width font), which is a part of the algorithm itself.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

3.6.10.1 Extracting *Entropy* from packets

The working variable *Entropy* is extracted from received packets depending upon how they are encapsulated. When native IPv4 and IPv4 encapsulations are used the packet includes the PDS Entropy Header (see Section 3.5.10.1) and the *Entropy* value is extracted from the **pds.entropy** field. When UDP encapsulation is used, the *Entropy* value is extracted from the **udp.source_port** field. In the following sections, when *Entropy* is referenced, it is assumed to have been extracted from the packets or updated by the CC algorithms.

3.6.10.2 ACK Preprocessing for Congestion Control

UET ACK packets may contain ACK *CC_STATE* in the **pds.ack_cc_state** field. The following two types of ACK *CC_STATE* are defined:

- When **pds.cc_type**=CC_NSCC. The **pds.ack_cc_state** field contains data used for the NSCC congestion control mechanism.
- When **pds.cc_type** =CC_CREDIT. The **pds.ack_cc_state** field contains data used for TFC flow control and (optionally) for RCCC congestion control.

Preprocessing of ACK *CC_STATE* is described in sections 3.6.10.3 and 3.6.10.4.

The *Entropy* value and the **pds.flags.m** field (indicating congestion) from the ACK are used for active load balancing between paths. Before processing the ACK **pds.ack_cc_state** field (if present), the source should update the path state it holds.

```
if pds.flags.m==1:
      process_ev(Entropy, ECN)
else
      process_ev(Entropy, NO_ECN)
```

If the CCC is in TFC mode or is for a ROD CCC, this step does not apply.

3.6.10.3 ACK *CC_STATE* Preprocessing for pds.cc_type=CC_NSCC

An ACK arriving on a PDC can indicate the cumulative total of bytes received by the peer on that PDC in the **pds.ack_cc_state.rcvd_bytes** field. The CCC does not specifically know about PDCs, so the source performs the following calculations to determine the number of newly received bytes at the destination on this PDC:

```
newly_rcvd_bytes = 0
if pds.ack_cc_state.rcvd_bytes > pdc.prev_rcvd_bytes:
      newly_rcvd_bytes = (pds.ack_cc_state.rcvd_bytes-
                              pdc.prev_rcvd_bytes) << 8
      pdc.prev_rcvd_bytes = pds.ack_cc_state.rcvd_bytes
```


Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

The left shift is required, as **pds.ack_cc_state.rcvd_bytes**is in units of 256 bytes, whereas *newly_rcvd_bytes* needs to be in bytes. As the **pds.ack_cc_state.rcvd_bytes** field can wrap, modular arithmetic must be used for the comparisons and subtractions. Note that unlike most CCC states, *pdc.prev_rcvd_bytes* is per-PDC state.

If NSCC is enabled, then the **pds.ack_cc_state.rcvr_cwnd_pend** and the **pds.ack_cc_state.rc** fields are also extracted into the *Rcv_Cwnd_Pend* and *Restore_Cwnd* variables respectively. Otherwise, these fields SHOULD be ignored.

If NSCC is enabled, the CCC needs to know the RTT sample measured from the ACK arrival time. This is performed using a timestamp stored when a data packet is transmitted. This packet state is therefore also passed to *OnACK()*. Whether a valid RTT can be calculated depends on whether the ACK acknowledges a retransmitted data packet. Thus, the **pds.flags.retx** bit from the ACK packet is also passed to *OnACK()*.

On ACK arrival, the source must look up the state associated with the *ACK_PSN* from the ACK:

```
pkt_tx_state = pdc.find_packet(ACK_PSN)
```

The *Service_Time* is extracted from the **pds.ack_cc_state.service_time** field. If the ACKed data packet had the **ip.ecn.ce** bit set at the destination the ACK has the **pds.flags.m** bit set.

The *OnACK()* action is then performed:

```
OnACK(newly_rcvd_bytes, Entropy, pds.flags.m, pkt_tx_state,
      ack_arrival_time, Service_Time, pds.flags.retx, Rcv_Cwnd_Pend,
      Restore_Cwnd)
```

3.6.10.4 ACK *CC_STATE* Preprocessing for pds.cc_type=CC_CREDIT

An ACK arriving may carry credit provided by the destination in the ACK **pds.ack_cc_state** field. Credit is allocated to the CCC rather than to a specific PDC. The *Credit* value is extracted from the **pds.ack_cc_state.credit** field, and `OnCreditUpdate(Credit)` is performed. Note that *Credit* is in units of 256 bytes.

3.6.10.5 NACK Preprocessing for CC

A NACK indicates one packet was not processed at the destination. Typically, it is sent because the packet was trimmed in the network, but a NACK can also be sent for other reasons as indicated in the **pds.nack_code** field of the NACK packet. Irrespective of the **pds.nack_code**, the NACK is processed by the CCC to update the CCC's inflight state, even if the **pds.nack_code** indicates to close the PDC. However, only NACKs with the **pds.nack_code** field set to UET_TRIMMED or UET_TRIMMED_LASTHOP will elicit a congestion response.

The source must look up the state associated with the NACKed packet using the **pds.nack_psn** field from the NACK:

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

```
pkt_tx_state = pdc.find_packet(pds.nack_psn)
```

If the packet was previously SACKed or cumulatively acknowledged, this state lookup will fail. CMS takes no further action in processing this NACK[11].

The source then performs the following calculations in an implementation-defined way:

```
nominal_pktsize = calculate_nominalpktsize(pkt_tx_state)
```

The *OnNACK()* action is then performed using these values:

```
OnNACK(nominal_pktsize, pds.nack_psn, Entropy, pds.flags.m,
      pkt_tx_state, nack_arrival_time, pds.nack_code, pds.flags.retx)
```

3.6.10.6 Retransmission Timeouts and Inferred Loss

The PDS can detect loss using a retransmission timeout or may infer loss from information returned in the SACK bitmap of an ACK or in other implementation-defined ways. In either case, the source MUST look up the state associated with the missing packet and compute the nominal packet size in an implementation-defined way:

```
pkt_tx_state = pdc.find_packet(lost_psn)
nominal_pktsize = calculate_nominalpktsize(pkt_tx_state)
```

The *OnInferredLoss()* action is then performed:

```
OnInferredLoss(nominal_pktsize)
```

3.6.10.7 CSIG Reflection Header Preprocessing for CC

If CSIG is enabled, the returning ACK state contains reflected CSIG information in the *csig_reflection_hdr*. If the ACK state is present, the implementation extracts the *csig_reflection_hdr* from **pds.ack_ccx_state.** All fields except the signal value are copied unmodified into the *csig_refl_data* structure.

```
csig_refl_data.t = csig_reflection_hdr.t
csig_refl_data.lm = csig_reflection_hdr.lm
csig_refl_data.d = csig_reflection_hdr.d
```

The signal value is also de-quantized while being copied:

```
csig_refl_data.s = quant_reverse_lookup(csig_reflection_hdr.s)
```

The de-quantization is done by looking up the metric value and the bucket ranges provided by the control plane using the inverse of the process described in Section 4.2.4.1.5 for the compact tag format

[11] While this is expected to be rare, a severely delayed NACK could cause this behavior.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

and in Section 4.2.4.1.6 for the wide tag format. The min value of each bucket is used to represent the resulting value of the de-quantization process. For the *max(Delay)* signal, the value conversion must follow the precision requirements discussed in Section 3.6.13.3.

If no reflection header is present, the NULL value indicator is passed to the functions using this input.

3.6.11 Congestion Control Modes

In this specification, three CC modes are defined:

1. **Network-signal Congestion Control** (NSCC): NSCC uses a combination of ECN and network latency to detect and respond to congestion by adapting a congestion window at the source and is clocked by state in the ACK **pds.ack_cc_state** field when **pds.cc_type** = CC_NSCC.
2. **Receiver-credit Congestion Control** (RCCC): RCCC uses explicit Credit CPs from the destination to control the sending rate of all sources to that destination.
3. **Transport Flow Control** (TFC): TFC uses credit sent from the destination to control sources with the aim of avoiding buffer overflow at the destination. TFC is not, by itself, a congestion control mechanism, so the expectation is for it be used across only a single link where congestion is managed internally to the ingress FEP. If it is used across multiple network hops, it should be used in conjunction with additional mechanisms to avoid or manage congestion at intermediate switches. TFC credit MAY be sent in ACKs using the **pds.ack_cc_state** field when **pds.cc_type** = CC_CREDIT or in the **pds.payload.credit** field of Credit CPs.

For ROD or RUD CCCs, either NSCC or RCCC may be used by themselves, or both can be used together. Generally, it is RECOMMENDED that all CCCs in a cluster use the same combination for NSCC and RCCC.

3.6.12 Overall CCC Pseudocode

The CCC algorithm is divided into a common part that is independent of which CC algorithms are selected, and specific parts for NSCC and RCCC. This section specifies the common part that handles generic state and load balancingI.

3.6.12.1 Sender Algorithm

The pseudocode for the common parts of the NSCC and RCCC corresponding to the abstract API at the source is shown in Section 3.6.12.3.

General Configuration State

nscc： TRUE if NSCC is enabled

rccc： TRUE if RCCC is enabled

disable_quick_adapt： TRUE if quick adapt has been disabled

It is RECOMMENDED to enable quick adapt unless RCCC is enabled.

NSCC and RCCC general configuration states are listed in sections 3.6.13.3, 3.6.14.2, and 3.6.14.5.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

General CCC State:

ccc.backlog = 0

ccc.backlog holds the total number of bytes (including packet headers, as calculated by *nominal_pktsize*) that PDS told the CCC to send on this CCC across all the PDCs associated with the CCC.

ccc.waiting_rtx = 0

ccc.waiting_rtx holds a count of the number of packets currently marked as needing retransmission.

ccc.rtx_backlog = 0

ccc.rtx_backlog holds the total number of bytes (including packet headers, as calculated by *nominal_pktsize*) that the PDS has pending retransmit on this CCC across all of the PDCs associated with the CCC.

ccc.inflight_pkts = 0

ccc.inflight_pkts holds the total number of packets in flight on this CCC across all of the PDCs associated with the CCC.

3.6.12.2 Nominal Packet Size

The pseudocode in Section 3.6.12.3 refers to the size of the packet for many values. Where stated, these use *nominal_pktsize*. It is important that the source and destination come up with the same value, even if the length of the packet changes in flight due to, for example, network telemetry. *nominal_pktsize* is an approximation of the actual packet size that is sufficiently accurate for the purposes of congestion control, but which cannot change in flight.

```
nominal_pktsize = transport_pktsize + 40
```

transport_pktsize is the UDP length of the packet if UDP encapsulation is used, or the size of the packet from the start of the UET entropy header to the end of the UET trailer if UDP encapsulation is not used.

3.6.12.3 Overall CCC Pseudocode

```
OnACK(newly_rcvd_bytes, Entropy, M_Flag, pkt_tx_state, ack_arrival_time,
      Service_Time, Retx_Flag, Rcv_Cwnd_Pend, Restore_Cwnd,
      csig_refl_data):
      #Each packet that was ACKed for the first time (e.g., by a cumulative
      #ACK or a SACK bit being set) should not be retransmitted.  Remove it
      #from the retransmit list if it is there.
      foreach newly acked packet:
            unmark_packet_for_rtx(acked_pkt)

      if M_Flag == 1:  #pkt was ECN marked
```


Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

```
            process_ev(Entropy, ECN)
      else:
            process_ev(Entropy, NO_ECN)

      if nscc:
            NSCC.OnACK(newly_rcvd_bytes, M_Flag, pkt_tx_state,
                        ack_arrival_time, Service_Time, Retx_Flag,
                        Rcv_Cwnd_Pend, Restore_Cwnd, csig_refl_data)

      if rccc:
            RCCC.OnACK()
      #Each ACK may acknowledge multiple packets (cumulative ACK, SACK, etc)
      ccc.inflight_pkts -= number_of_newly_acked_packets
update_state()

OnNACK(nominal_pktsize, NACK_PSN, Entropy, M_Flag, pkt_tx_state,
       nack_arrival_time, reason, Retx_Flag, pdc_present):
      if reason == TRIM_NACK: #Non-last hop trim
            process_ev(Entropy, NACK)
      else if M_Flag == 1:   #pkt was ECN marked before it was trimmed
            process_ev(Entropy, ECN)
      else:  #last hop trim, or other cause—no need to load balance
            process_ev(Entropy, NO_ECN)

      if nscc:
            NSCC.OnNACK(nominal_pktsize, pkt_tx_state, nack_arrival_time,
                        Retx_Flag, reason, rccc)

      mark_packet_for_rtx(NACK_PSN, nominal_pktsize)
      if rccc:
            RCCC.OnNACK(pdc_present)
      ccc.inflight_pkts--
      update_state()

OnCreditUpdate(Credit):
      if rccc:
            RCCC.OnCreditUpdate(Credit)
      update_state()
```


Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

```
OnInferredLoss(pkt_state, nominal_pktsize, pdc_present):
      if nscc:
            NSCC.OnInferredLoss(nominal_pktsize)
      mark_packet_for_rtx(pkt_state.psn, nominal_pktsize)
      if rccc:
            RCCC.OnInferredLoss(pdc_present)
      process_ev(pkt_state.entropy, TIMEOUT)
      ccc.inflight_pkts--
      update_state()

OnSend(nominal_pktsize, is_rtx):
      if is_rtx:
            ccc.waiting_rtx -= 1
            ccc.rtx_backlog -= nominal_pktsize
      else:
            ccc.backlog -= nominal_pktsize
      if nscc:
            NSCC.OnSend(nominal_pktsize)
      if rccc:
            RCCC.OnSend(nominal_pktsize)
      ccc.inflight_pkts++
      update_state()

OnNewData(delta_backlog):
      ccc.backlog += delta_backlog
      if rccc:
            RCCC.OnNewData()
      update_state()

GetSendParams(free_port_list) -> port, entropy, credit_target, ack_req:
      determine entropy and NIC port
      if rccc:

            credit_target = ccc.credit_target
      ack_req = FALSE
      if nscc:
            #ack_req is TRUE if the CCC would like AR to be set.
```


Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

```
            #PDS may also set AR for other reasons.
            ack_req = NSCC.AckRequest()
      return port, entropy, credit_target, ack_req

mark_packet_for_rtx(psn, nominal_pktsize):
      #marking the packet for retransmission is a PDC function, but there
      #are CC side effects below. Mark the psn for retransmission as soon
      #as allowed by cwnd
      ccc.waiting_rtx += 1
      ccc.rtx_backlog += nominal_pktsize
      update_state()

unmark_packet_for_rtx(acked_pkt):
      #check if we were going to retransmit a packet that was just acked
      if acked_pkt is marked for retransmission:
            unmark the packet for retransmission
            ccc.waiting_rtx -= 1
            ccc.rtx_backlog -= acked_pkt.size

update_state():
      if ccc.backlog == 0 and ccc.waiting_rtx == 0: #no more data to send
            if ccc.inflight_pkts == 0:
                  ccc.state = IDLE
            else:
                  ccc.state = PENDING         #still waiting for Acks
      else if ccc.state == IDLE or ccc.state == PENDING:
            ccc.state = ACTIVE

      can_send = TRUE           #does each active CC algorithm allow sending?
      if ccc.state == ACTIVE or ccc.state == READY:
            if nscc:
                  can_send &= NSCC.CanSend()
            if rccc:
                  can_send &= RCCC.CanSend()

      if ccc.state == ACTIVE and can_send:
            ccc.state = READY   #add CCC to scheduler ready list
      else if ccc.state == READY and not can_send:
```


Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

```
            ccc.state = ACTIVE  #remove CC from scheduler ready list
```

process_ev(entropy, reason):

UET-CC does not prescribe how the feedback information received from ACKs and NACKs is used to load balance. Oblivious load balancing and adaptive load balancing are both permitted, as described in Section 3.6.16. Some load balancing schemes require little (or no) action in *process_ev()*, and others may vary in what they do based on the *reason*. Load balancing schemes that do not require processing of an *entropy* for a given *reason* are not required to process the *entropy* for that *reason*. If the CCC is used for a ROD or TFC PDC, *process_ev()* does nothing.

3.6.12.4 Receiver Algorithm

The following pseudocode defines the common CCC behavior at the CCC destination. The conditions for sending an ACK are specified in the PDS Section 3.5.12. RCCC also maintains a credit timer that clocks out credit. This is detailed in Section 3.6.14.5.

```
OnRX(pkt):
      if rccc: RCCC.OnRX(pkt)
      if nscc: NSCC.OnRX(pkt)
      if pkt.IsTrimmed:
            lastHopTrim = (pkt.ip.dscp == DSCP_TRIMMED_LASTHOP)
            sendNACK(pkt.pds.psn, Entropy, lastHopTrim, pkt.ip.ecn.ce)
      else if conditions are met to send an ACK:
            sendACK(pkt.pds.psn, Entropy, pkt.ip.ecn.ce)
            #other ACK CC_STATE fields will be filled in by the
            #relevant CC algorithm
```

3.6.13 Network Signal-based Congestion Control

This section specifies the NSCC algorithm, which is based on the SMaRTT [9] and Strack [10] algorithms.

NSCC uses a congestion window, *ccc.cwnd*, that limits the amount of outstanding in-flight data. NSCC permits sending only when *ccc.cwnd* > (*ccc.inflight* + MTU[12]). As with other window-based congestion control protocols, NSCC relies on ACK clocking to regulate the amount of data permitted into the network. The arrival of ACKs indicates that previously transmitted data has left the network and allows the transmission of additional data. In NSCC, the ACK clock that decreases *ccc.inflight* is driven from the **pds.ack_cc_state.rcvd_bytes** field returned by the destination in ACK packets, which indicates the amount of data that has arrived at the destination. When the source is told by the destination that an additional amount of data has left the network, *ccc.inflight* is reduced, allowing more data to be sent. Similarly, *ccc.inflight* is reduced if a packet is known or inferred to have been lost. These mechanisms are resilient to reordering caused by multipath spraying. Note that *ccc.inflight* can go slightly negative

[12] In some implementations, it may be possible to use the actual packet size instead of the MTU.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

temporarily, due to **pds.ack_cc_state.rcvd_bytes** being rounded up to units of 256 bytes and due to ACK reordering; this does not affect correctness but does require that *ccc.inflight* is maintained as a signed value.

A single *ccc.cwnd* is maintained for packets sprayed across multiple paths; in effect, it tracks the available capacity in the network across the aggregate of paths. The core NSCC algorithm adapts *ccc.cwnd* based on observed network conditions. The primary control loop is driven by a combination of ECN and the bottleneck queueing delay. The bottleneck queueing delay can be either explicitly provided by CSIG network telemetry using the *max(Delay)* signal or implicitly derived from on the packets' end-to-end RTT.

The packets' total queueing delay can be derived by calculating RTT – *base_RTT*, where *base_RTT* is the expected latency when there is no queue. In most cases, a single bottleneck dominates the total queueing delay of a packet, which makes the total queueing delay a good approximation for the bottleneck queueing delay when the CSIG *max(Delay)* signal is not available. RTT is measured by storing the transmit time of each packet at the source and subtracting both the ACK arrival time and the **pds.ack_cc_state.service_time** as reported in the ACK packet.

The goal is to keep the bottleneck queueing delay (hereafter referred to simply as *Delay*) within bounds. Note that a high delay can be measured only after a packet has spent time to traverse the queue. In contrast, modern Ethernet switches set ECN on a packet when it is dequeued, based on the queue size at the time of dequeue. This means that the single-bit ECN signal is a leading indicator of congestion, whereas *Delay* is a lagging multibit indicator. The destination echoes the received ECN value back to the source. This leads to four combinations of ECN and *Delay* that are used to determine appropriate responses:

- ECN is not set, *Delay < target_qdelay*:
    - Network is uncongested
    - Perform *proportional_increase()*
- ECN is not set, *Delay >= target_qdelay*:
    - Network was congested, but congestion has reduced
    - Perform *fair_increase()*
- ECN is set, *Delay < target_qdelay*:
    - Network is becoming congested for packets after this one, but is not yet regarded as congested
    - No increase or decrease needed
- ECN is set, *Delay >= target_qdelay*:
    - Network is congested
    - Perform *multiplicative_decrease()*

For each of these cases, a different congestion response is appropriate.

The *proportional_increase()* action, performed when uncongested, by default increases *ccc.cwnd* by a

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

quantity proportional to the difference between the *Delay* and the *target_delay*. Thus, when the network is lightly loaded, *ccc.cwnd* increases faster than when the load approaches the congestion threshold. If the network has been underloaded for some time, as might happen when competing flows terminate, *proportional_increase()* can trigger *fast_increase()* to converge more rapidly on a new operating point. The *fast_increase()* action performs an exponential increase and is terminated by any sign of incipient congestion.

The *fair_increase()* action is performed after a packet experiences congestion, but ECN indicates the queue behind it has drained below the ECN setting threshold. The *fair_increase()* action prevents transmission rate undershooting. It performs an additive increase, which helps competing CCCs converge toward fairness: all CCCs receiving the same signal increase *ccc.cwnd* by the same amount, so CCCs with a smaller *ccc.cwnd* see a larger proportional increase than CCCs with a larger *ccc.cwnd*.

The *multiplicative_decrease()* action is performed when the *Delay* is over threshold and ECN does not indicate the queue is decreasing. In this case, the average delay provides a direct indication of the extent to which the queue holds more data than is desired. The *multiplicative_decrease()* action then directly reduces *ccc.cwnd* proportional to this queuing excess. If all flows take the same action, the queue is expected to return to the target level in just over one RTT.

The goal of the algorithm is to keep queue sizes small without sacrificing throughput, thereby avoiding steady-state packet loss and minimizing the duration of loss when the offered load changes too rapidly to control. In AI networks, this condition typically occurs when collectives cause synchronized flow startup.

Finally, The *quick_adapt()* action is performed when loss is detected or when delay across many paths is excessively high. The *quick_adapt()* function uses the measured achieved goodput to directly set *ccc.cwnd*. This means that if incast congestion occurs, for example, in the second RTT, the *ccc.cwnds* of the PDCs involved in the incast will combine to avoid overloading the destination's link. The *quick_adapt()* action does not guarantee instantaneous fairness, but in subsequent RTTs *fair_increase()* and *multiplicative_decrease()*cause convergence. Because *quick_adapt()* relies on a measure of achieved bandwidth, it can be triggered only once per RTT. When the *quick_adapt()* action is triggered, many packets may already be queued, and most of the corresponding acknowledgements will indicate that ECN was set. Until these packets are ACKed, *multiplicative_decrease()* is suppressed.

NSCC is designed to work with active ECN-based load balancing, where the ECN signal is also used to balance congestion across paths, in addition to being used as an NSCC congestion signal. As a result, NSCC does not react to low levels of congestion, indicated only by ECN and not by *Delay*, as this is often a symptom of imperfect load balancing. When ECN-based load balancing is used, it is desirable to allow the load balancing algorithm to react before reducing the aggregate send rate, except under high congestion when the ECN signal saturates and is used only to reduce the aggregate load.

NSCC leaves it to the implementor to handle outcast (e.g., when multiple CCCs are active and *ccc.inflight* never approaches the *ccc.cwnd* limit) if needed. During a period of outcast, it is possible that the congestion windows (*ccc.cwnd*) of multiple sending CCCs increase to their maximum values beyond the sending capacity of the outgoing FEP. This occurs because each CCC is operating below its nominally

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

allowed rate while the network delay appears to be low to NSCC.

3.6.13.1 Calculating RTT

NSCC needs to know the RTT sample measured from the ACK arrival. The PDC SHOULD store the transmit timestamp for each packet sent and update this timestamp each time the packet is retransmitted. Generally, the source will subtract the stored transmit timestamp from the time of the ACK arrival and subtract off the destination's *Service_Time* to calculate the RTT. However, care must be taken with retransmitted packets, where there is a potential ambiguity as to whether the ACK is for the original or the retransmitted packet. To resolve this ambiguity, a source SHOULD store whether a packet has been retransmitted. Data packets carry the **pds.flags.retx** bit indicating that a packet is a retransmission. This **pds.flags.retx** flag is echoed in the ACK message. Thus, if the **pds.flags.retx** bit in the ACK is not set, and the packet has been retransmitted, the ACK MUST NOT be used to update the RTT measures.

An implementation MAY choose to not update RTT measures when the **pds.flags.retx** bit in an ACK is set. It is preferable, however, to update RTT based on ACKs of retransmissions, but again care must be taken to avoid any potential ambiguity when a packet is retransmitted a second time. It is RECOMMENDED that a source records if a packet is retransmitted a second (or subsequent) time. If a packet has been retransmitted only once, and the **pds.flags.retx** bit is set in the corresponding ACK, then it is safe to update RTT. Otherwise, it is unsafe, and the RTT should not be updated. One way to implement this is to maintain a two-bit *rtx_count*, which records whether a packet has been retransmitted a) never, b) once, or c) more than once. For clarity, the pseudocode in Section 3.6.13.6 assumes such an implementation, but other implementations are permitted.

3.6.13.2 Destination Flow Control

The destination can modulate the source's transmit rate using a receiver window penalty, *Rcv_Cwnd_Pend*. This flow control is intended to be used if the destination is itself congested and is not keeping up with the arrival rate of traffic. The **pds.ack_cc_state.rcvr_cwnd_pend** field is returned in the ACK and MUST be set to a value between 0 and 127. For example:

- Setting **pds.ack_cc_state.rcvr_cwnd_pend** to 0 does not flow control sources, i.e., this field has no effect.
- Setting **pds.ack_cc_state.rcvr_cwnd_pend** to 127 slows all the sources by the maximum amount, down to a window of one packet per RTT.
- Setting **pds.ack_cc_state.rcvr_cwnd_pend** to 64 in all SACKs sent for an RTT halves all source windows (*cwnd*), thus reducing the bandwidth to the receiver.

The **pds.ack_cc_state.rc** flag in the ACK indicates whether the source should save the current value of *cwnd* when *Rcv_Cwnd_Pend* indicates a period of receiver flow control has started, and restore the original *cwnd* value when receiver flow control ends.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Receivers MAY dynamically determine *Rcv_Cwnd_Pend* depending on the degree of destination congestion. The specifics of how the *Rcv_Cwnd_Pend* is calculated is implementation dependent. The field may be set to 0 if it is not supported by the receiver.

The source responds to non-zero values of *Rcv_Cwnd_Pend* by modifying the congestion control response. This is detailed in *apply_cwnd_penalty()* in the NSCC pseudocode.

Table 3-97 - Congestion Control Configuration Parameters

Name	Type	Default	Description
config_base_rtt	unsigned integer	See Section 3.6.13.3	The round-trip time in seconds of the longest path across the fabric of an MTU-sized packet when no other traffic is present. (Units: seconds, but see the precision discussion in 3.6.13.3)
sender.linkspeed	unsigned integer	NIC *linkspeed*	The maximum achievable bitrate over the source's link, measured in bytes/second.
Receiver.linkspeed	unsigned integer	NIC *linkspeed*	The maximum achievable bitrate over the receiver's link, measured in bytes/second. Usually this is the same as the source's *linkspeed*. If unknown, set to source's *linkspeed*.
MTU	unsigned integer	Payload MTU + Transport Overhead + 40	Nominal MTU. An approximation of the MTU using the Payload MTU from the semantic sublayer plus the transport overhead plus 40 bytes. Matches the *nominal_pktsize* of a full Payload MTU packet.
BDP	unsigned integer	See Section 3.6.13.3	The nominal bandwidth-delay product needed to achieve high throughput across the longest path in an unloaded network. (units: bytes)
MaxWnd	unsigned integer	1.5 * BDP	Upper bound on *cwnd*. (units: bytes)
target_qdelay	unsigned integer	See Section 3.6.13.3	Target queueing delay. (Units: seconds, but see the precision discussion in 3.6.13.3)
Gamma	floating point	0.8	
alpha	floating point	See Section 3.6.13.3	
fi	floating point	5 * MTU * scaling_a	Fair increase constant.
fi_scale	floating point	0.25 * scaling_a	
max_md_jump	floating point	0.5	Limit on how much multiplicative decrease can decrease *cwnd* (unitless).
base_BDP	unsigned integer	150000	The hypothetical BDP of a network with 100 Gb/s source and receiver *linkspeed*s and a 12 µs base RTT. (units: bytes)
scaling_a	floating point	BDP/base_BDP	Used to scale to networks with differing *linkspeed*s or RTT.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Name	Type	Default	Description
scaling_b	floating point	target_qdelay / 0.000012	Used to scale parameters as *target_qdelay* is adjusted relative to 12 µs baseline.
eta	floating point	0.15 * MTU * scaling_a	Additive increase constant to improve fairness.
qa_threshold	unsigned integer	See Section 3.6.13.3	Average queuing delay before *quick_adapt()* kicks in without seeing losses.
adjust_bytes_threshold	unsigned integer	8 * MTU	Apply decrease at least every 8 ACKs.
adjust_period_threshold	unsigned integer	config_base_rtt	Apply decrease at least once per RTT.
qa_gate	unsigned integer	3	quick_adapt is enabled when achieved_bytes < (ccc.max_wnd/2^{qa_gate}).
csig_maxdelay_enable	boolean	FALSE	Enable using *max(Delay)* signal provided by CSIG.
csig_missing_threshold	unsigned integer	0	If set to value > 0, receiving the specified number of ACKs without CSIG data will transition the CC into fallback state using the calculated congestion delay.

3.6.13.3 NSCC Configuration Parameters

The following configuration parameters and constants are used to tune NSCC behavior. An implementation SHOULD allow the configuration of all parameters shown in Table 3-97.

Implementation Note:
Table 3-97 defines the type of several parameters as floating-point numbers. The precision of floating-point arithmetic is not defined and is an implementation-specific decision. Any valid floating-point rounding mode may be selected for floating-point calculations.

The *config_base_rtt* parameter is defined as the round-trip time in seconds of the longest path across the fabric of an MTU-sized packet when no other traffic is present, set by configuration for all CCCs. While for the purposes of consistency in the formulas below, this is in units of seconds. Implementations should maintain *base_rtt* to the nearest 128 ns or better, to match the *tx_timestamp* precision.

BDP defaults to:

```
BDP = min(sender.linkspeed, receiver.linkspeed) * config_base_rtt
```

BDP is in units of bytes, so *linkspeed*s above should be measured in units of bytes/second. The destination *linkspeed* will typically be the same as the source *linkspeed*. If it is unknown, use the source's *linkspeed*.

alpha defaults to:

```
alpha = 4.0 * scaling_a * scaling_b * MTU / target_qdelay
```


Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

target_qdelay, qa_threshold, and *adjust_period_threshold* all measure time. They are expressed here in units of seconds for the purposes of consistency in these formulae. Implementations should maintain *target_qdelay* to the nearest 128 ns or better to match the *tx_timestamp* precision.

target_qdelay defaults to:

`config_base_rtt * 0.75` when trimming is used

`config_base_rtt * 1.0` when trimming is not used

The parameter *qa_threshold* is used when trimming is disabled; it should be set to a high value to disable it when trimming is used. The *quick_adapt()* action needs to be triggered before tail dropping occurs in the network; thus, *qa_threshold* is a derivative of the tail drop threshold. The *qa_threshold* parameter should be set to (*drop threshold/Plane_BDP -1)*configure_base_RTT*. Using the recommended tail drop threshold of *5*Plane_BDP*, this yields a *qa_threshold* of *4*target_qdelay*. See Section 3.6.17 for details on the recommended switch settings.

3.6.13.4 NSCC Source State

When an NSCC CCC is initialized, the following per-CCC state is initialized:

Table 3-98 - NSCC Source State

Name	Type	Default	Description
ccc.cwnd	unsigned integer	*MaxWnd*	Congestion window that constrains the in-flight data. (units: bytes)
ccc.inflight	signed integer	0	The amount of unacknowledged data still thought to be in transit. (units: bytes)
ccc.saved_cwnd	unsigned integer	0	Original value of *cwnd* saved when destination flow control starts.
ccc.base_rtt	unsigned integer	*config_base_rtt*	Base RTT for the CCC. Starts at *config_base_rtt*, but is reduced if an RTT sample is lower. (units: 128 ns)
ccc.achieved_bytes	unsigned integer	0	Achieved received bytes from past RTT for quick adapt. (units: bytes)
ccc.received_bytes	unsigned integer	0	Achieved received bytes to trigger *fulfill_adjustment()*. (units: bytes)
ccc.fi_count	unsigned integer	0	Count of packet bytes that see no ECN-CE and also see an RTT that is around base_rtt. (units: bytes)
ccc.trigger_qa	boolean	FALSE	Set to TRUE to trigger *quick_adapt()* when the deadtime since the last call to *quick_adapt()* has expired.
ccc.qa_endtime	unsigned integer	0	The time value of the end of Quick_Adapt time window. (units: local time units)
ccc.bytes_to_ignore	unsigned integer	0	Used after *quick_adapt()* to keep track of how much data to ignore. (units: bytes)

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Name	Type	Default	Description
ccc.bytes_ignored	unsigned integer	0	Used after *quick_adapt()* to count how many bytes have been ignored so far. (units: bytes)
ccc.inc_bytes	unsigned integer	0	Used to accumulate window increase for later application.
ccc.last_adjust_time	unsigned integer	now	The last time that accumulated *cwnd* changes were applied. (units: local time units)
ccc.increase_mode	boolean	FALSE	Latches TRUE when entering *fast_increase* mode. Cleared to FALSE to exit *fast_increase* mode.
ccc.last_dec_time	unsigned integer	now	The last time that *multiplicative_decrease* was performed. (units: local time units)
ccc.max_wnd	unsigned integer	*MaxWnd*	The scaled version of *MaxWnd* based on measured RTT.
ccc.csig_missing_cnt	unsigned integer	0	Counts missing CSIG data in *OnACK()* calls

ccc.inflight maintains the total number of bytes (including packet headers, as calculated by nominal packetsize) that the CCC has recorded as being sent and not yet been notified that have been ACKed, NACKed, timed out, or otherwise inferred to be lost on all PDCs associated with this CCC. Under some circumstances *inflight* temporarily can go slightly negative, so *inflight* SHOULD be stored as a signed integer.

3.6.13.5 NSCC Source Algorithm

```
NSCC.OnACK(newly_rcvd_bytes, M_Flag, pkt_tx_state, ack_arrival_time,
            Service_Time, Retx_Flag, Rcv_Cwnd_Pend, Restore_Cwnd,
            csig_refl_data):
      ccc.inflight -= newly_rcvd_bytes
      ccc.bytes_ignored += newly_rcvd_bytes
      ccc.received_bytes += newly_rcvd_bytes
      ccc.achieved_bytes += newly_rcvd_bytes

      rcv_limit_mode =
            apply_cwnd_penalty(Rcv_Cwnd_Pend, Restore_Cwnd, newly_rcvd_bytes)

      if (!csig_maxdelay_enable
          || ccc.csig_missing_cnt > csig_missing_threshold):
            rtt_sample = calculate_rtt(pkt_tx_state, ack_arrival_time,
                                       Service_Time, Retx_Flag)
            if rtt_sample != INVALID_RTT:
                  update_base_rtt(rtt_sample)
```


Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

```
                delay = rtt_sample – ccc.base_rtt
                update_delay(delay)
          else:
                return
    else:
          if csig_refl_data and csig_refl_data.t == “max(Delay)”:
                delay = csig_refl_data.s
                update_delay(delay)
          else:
                ccc.csig_missing_cnt += 1
                return

    if quick_adapt(is_loss=FALSE, M_Flag, delay):
          return;

    #The pds.flags.m bit in the ACK indicates ECN was marked in the
    #request. There are four core cases (3 with actions):
    if M_Flag == 0 and delay >= target_qdelay and not rcv_limit_mode:
          fair_increase(newly_rcvd_bytes)
    else if M_Flag == 0 and delay < target_qdelay and not rcv_limit_mode:
          proportional_increase(newly_rcvd_bytes, delay)
    else if M_Flag == 1 and delay >= target_qdelay:
          multiplicative_decrease()

    #we’ve accumulated window changes for long enough, now apply them
    if (now - ccc.last_adjust_time) >= adjust_period_threshold
        or ccc.received_bytes > adjust_bytes_threshold:
          #This is potentially done per packet or at a lower frequency
          #(RTT/4, RTT/2 and so on).
          fulfill_adjustment()

NSCC.OnInferredLoss(nominal_pktsize):
    cwnd = max(cwnd - nominal_pktsize, MTU)
    ccc.bytes_ignored += nominal_pktsize
    ccc.inflight -= nominal_pktsize


NSCC.OnNACK(nominal_pktsize, pkt_tx_state, nack_arrival_time, Retx_Flag,
```


Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

```
      reason, rccc):
    adjust_cwnd=FALSE
    ccc.inflight -= nominal_pktsize
    rtt_sample = calculate_rtt(pkt_tx_state, nack_arrival_time,
                               service_time=0, Retx_Flag)
    if rtt_sample != INVALID_RTT:
          update_base_rtt(rtt_sample)
    #If the NACK was generated by trimming,
    #update the delay with the estimate for the trimmed queue delay
    if reason == UET_TRIMMED or reason == UET_TRIMMED_LASTHOP:
          update_delay(config_base_rtt)
          ccc.bytes_ignored += nominal_pktsize
    if reason == UET_TRIMMED
       or (reason == UET_TRIMMED_LASTHOP and rccc == FALSE):
          #Only adjust the cwnd on a last hop trim if RCCC is not enabled
          adjust_cwnd = TRUE
          ccc.trigger_qa = TRUE

          #is_loss causes delay to be ignored, so set delay to 0
          #A trimmed packet counts as ECN marked for quick adapt, so set
          #M_Flag
          if quick_adapt(is_loss=TRUE, M_Flag=1, delay=0):
                #if quick_adapt ran, don't adjust cwnd again
                adjust_cwnd=FALSE

    if adjust_cwnd == TRUE :
          cwnd = max(cwnd - nominal_pktsize, MTU)


NSCC.OnSend(nominal_pktsize):
      ccc.inflight += nominal_pktsize

NSCC.CanSend():
      #An implementation MAY be precise and consider inflight+pktsize if
      #desired. Note: using pktsize could cause a PDC with small packets to
      #starve a PDC with large packets when CWND is one MTU. This is an
      #issue to be handled in the implementation of the scheduler.
      return ccc.inflight + MTU <= ccc.cwnd
```


Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

NSCC.AckRequest():

```
#NSCC performs better if the destination generates an ACK whenever
#the source's cwnd is full. If the cwnd is not large enough to trigger
#an ACK, request an ACK.
return (ccc.cwnd - ccc.inflight) < MTU
      or ccc.cwnd < pds.ACK_Gen_Trigger
```

Implementation Note:
The use of the *ACK_Gen_Trigger* parameter in the test in *NSCC.AckRequest()* would require FEPs to have symmetric configuration of this parameter. Other implementation methods are permitted, such as device-specific configurations to provide the lowest value of *ACK_Gen_Trigger* in the system to the transmit logic.

Also, the *NSCC.AckRequest()* test assumes the packet size of this packet has been incorporated into *ccc.inflight* before calling *NSCC.AckRequest().*

3.6.13.6 NSCC Internal Functions

calculate_rtt(pkt_tx_state, ack_arrival_time, Service_Time, Retx_Flag):

```
rtt_sample = INVALID_RTT
#other algorithms are permitted to determine if the RTT is valid
#when a packet has been retransmitted.
if (pkt_tx_state.rtx_count == 0 and Retx_Flag == FALSE)
    or (pkt_tx_state.rtx_count == 1 and Retx_Flag == TRUE):
      rtt_sample = ack_arrival_time - (pkt_tx_state.tx_timestamp
                  + Service_Time)
return rtt_sample
```

fulfill_adjustment():

```
ccc.cwnd += ccc.inc_bytes/ccc.cwnd
if (now - ccc.last_adjust_time) >= adjust_period_threshold:
      ccc.last_adjust_time = now
      ccc.cwnd += eta

if ccc.cwnd > ccc.max_wnd:
      ccc.cwnd = ccc.max_wnd
ccc.inc_bytes = 0
ccc.received_bytes = 0
```


Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

```
fair_increase(newly_rcvd_bytes):
      #inc_bytes will be divided by cwnd before being added to cwnd
      ccc.inc_bytes += fi * newly_rcvd_bytes

proportional_increase(newly_rcvd_bytes, delay):
      fast_increase(newly_rcvd_bytes, delay)
      if ccc.increase_mode:
            return
      ccc.inc_bytes += alpha * newly_rcvd_bytes*(target_qdelay - delay)

multiplicative_decrease():
      ccc.increase_mode = FALSE     #turn off fast increase
      ccc.fi_count = 0
      avg_delay = get_avg_delay()
      if avg_delay > target_qdelay:
            if (now - ccc.last_dec_time) > ccc.base_rtt:
                  ccc.cwnd *= max(1-gamma*(avg_delay-
                              target_qdelay)/avg_delay,
                              max_md_jump)
                  ccc.cwnd = max(ccc.cwnd, MTU)
                  ccc.last_dec_time = now

quick_adapt(bool is_loss, bool M_Flag, qdelay):
      if disable_quick_adapt == TRUE:
            return FALSE
      qa_done_or_ignore = FALSE

      if ccc.bytes_ignored < ccc.bytes_to_ignore and M_Flag == 1:
            #We are still in the "bytes to ignore" phase,
            #don't run quick adapt, but reset the fulfill-adjustment counters
            qa_done_or_ignore = TRUE
      else if now >= ccc.qa_endtime:
            if ccc.qa_endtime != 0
               and (ccc.trigger_qa or is_loss or qdelay > qa_threshold)
               and (ccc.achieved_bytes < (ccc.max_wnd >> qa_gate)):
                  #we have a trim packet or very large RTT
                  ccc.cwnd = max(ccc.achieved_bytes, MTU)
```


Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

```
                ccc.bytes_to_ignore = ccc.inflight
                ccc.bytes_ignored = 0
                ccc.trigger_qa = FALSE
                qa_done_or_ignore = TRUE
          ccc.achieved_bytes = 0
          ccc.qa_endtime = now + ccc.base_rtt + target_qdelay

    #If we are either in the bytes to ignore phase or ran quick adapt,
    #reset fulfill-adjustment counters
    If qa_done_or_ignore == TRUE:
          ccc.inc_bytes=0
          ccc.received_bytes=0

    return qa_done_or_ignore

fast_increase(newly_rcvd_bytes, delay):
    if delay ~= 0:
          ccc.fi_count += newly_rcvd_bytes
          if ccc.fi_count > ccc.cwnd or ccc.increase_mode:
                ccc.cwnd += newly_rcvd_bytes  * fi_scale
                if ccc.cwnd > ccc.max_wnd:
                      ccc.cwnd = ccc.max_wnd
                ccc.increase_mode = TRUE
                return
    else:
          ccc.fi_count = 0
    ccc.increase_mode = FALSE

update_base_rtt(raw_rtt):
    if ccc.base_rtt > raw_rtt:
          ccc.base_rtt = raw_rtt
          ccc.max_wnd = 1.5 * sender.linkspeed * (ccc.base_rtt in seconds)

apply_cwnd_penalty(Rcv_Cwnd_Pend, Restore_Cwnd, newly_rcvd_bytes):
    if Rcv_Cwnd_Pend > 0:
          if ccc.saved_cwnd == 0 then
                ccc.saved_cwnd = ccc.cwnd
          window_decrease = (Rcv_Cwnd_Pend * newly_rcvd_bytes) >> 7
```


Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

```
            ccc.cwnd = min(ccc.cwnd, ccc.inflight)
            ccc.cwnd = max(mtu, ccc.cwnd – window_decrease)
      else if Restore_Cwnd and ccc.saved_cwnd > 0:
            ccc.cwnd = ccc.saved_cwnd
            ccc.saved_cwnd = 0
      return Rcv_Cwnd_Pend > 0
```

The average delay is used in multiplicative decrease and *quick_adapt()*. How this should be calculated is an implementation-specific decision. For example, an implementation that updates a stored running average when *update_delay()* is called, and returns that value when *get_avg_delay()* is called, would suffice. Other methods to reduce the per-ACK computation costs are also permitted.

It is a requirement that *get_avg_delay()* be callable multiple times per RTT and still produce an appropriate average.

```
update_delay(delay):
      add this sample to the delay averaging state


get_avg_delay():
      return the average delay over the last base_rtt.
```

3.6.13.7 Initializing *base_rtt*

The fairness of NSCC depends on getting a good low RTT sample to seed *base_rtt*. This is especially the case when a local flow starts up and competes with a pre-existing longer-distance flow. Under these circumstances, the local flow may not get a good measurement of *base_rtt*, leading to unfairness against the longer distance flow.

When a new NSCC CCC to a destination is established, a source SHOULD send a Probe CP on the control traffic class using the DSCP_CONTROL diffserv codepoint to obtain a good sample of *base_rtt*. As a Probe CP cannot bring up a PDC, if this Probe CP is sent immediately using the control traffic class, it is likely to arrive before the first data packet of the PDC and would then be discarded. Thus the source SHOULD wait until the first ACK or NACK is received before sending the Probe CP. If an implementation knows no more data will be sent after the second RTT, this Probe CP MAY be safely omitted.

Sending a Probe CP is unnecessary if a source has a cached *base_rtt* from a previous CCC, can reliably estimate *base_rtt*, or is handling a PDC that is known to transmit little data (typically around one BDP).

3.6.13.8 NSCC Destination State

NSCC is primarily a sender-based algorithm. It depends on the *Rcvd_Bytes* value calculated at the destination to drive its ACK-clock. *Rcvd_Bytes* is maintained per PDC.

An NSCC destination maintains the following CC state.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Table 3-99 - NSCC Destination CC State

Name	Type	Default	Description
pdc.rcvd_bytes	unsigned integer	0	The total number of accepted new bytes on this PDC. (units: bytes)

3.6.13.9 NSCC Destination Algorithm

```
OnRX(pkt):
      if pkt contains data and pkt is not trimmed and pkt is not a duplicate:
            pdc.rcvd_bytes += pkt.nominal_pktsize
```

The **pds.ack_cc_state.rcvd_bytes** field in the ACK sent from the destination is in units of 256 bytes. It is derived from the *pdc.rcvd_bytes* as follows:

```
Rcvd_Bytes = ceil(pdc.rcvd_bytes / 256.0)
```

The value needs to be rounded up to avoid never acknowledging small packets when the remaining *inflight* is smaller than 256 bytes. The *Rcvd_Bytes* value can be equivalently calculated using integer arithmetic as:

```
Rcvd_Bytes = (pdc.rcvd_bytes + 255) >> 8
```

3.6.13.10 Using CSIG max(Delay)

NSCC can use the CSIG *max(Delay)* signal for determining the bottleneck queueing delay instead of the default method of deriving it from the packets' RTT. This requires the CSIG to be enabled in the NSCC CCC as well as in the corresponding FEP. In NSCC, the *csig_maxdelay_enable* configuration parameter (see Table 3-97) must be set to TRUE. The FEP's CSIG policy MUST enforce the insertion of CSIG tags with the *max(Delay)* signal in all packets (see Section 4.2.5.2).

All ACK packets processed by CSIG-enabled NSCC are expected to include the CSIG *max(Delay)* header. If for some reason the CSIG *max(Delay)* signal is not available in an ACK received by the CCC, *ccc.csig_missing_cnt* is increased, and the acknowledgement is ignored. If the counter crosses a threshold, NSCC falls back to calculating the queueing delay based on RTT measurements. It is up to the administrator to respond to this event and choose further actions. NSCC will stay in the fallback state.

Informative Text:
The inclusion of CSIG tags for signal types other than *max(Delay)* on distinct packets of an NSCC PDC is outside the scope of this specification. In these situations, care is required to prevent spurious fallback to RTT-based congestion signals and to adjust NSCC parameters when *max(Delay)* feedback is less frequent than per-ACK. This can be achieved by resetting the counter for ACKs without a *max(Delay)* signal, requiring the configuration of the fallback threshold to suppress such transitions, or by other suitable means.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

3.6.14 UET Receiver-Credit Congestion Control

The UET receiver-credit congestion control (RCCC) service is derived from EQDS[11]. RCCC uses end-to-end credit control messages ("pull messages") sent by the destination at a defined "rate" to all concurrent sources to that destination in order to control their transmissions and so avoid persistent congestion. Under normal conditions, credit will be sent at a rate that matches the destination's link speed.

From the source's point of view, RCCC is similar to a window-based protocol: Each source starts with an initial amount of credit. Sending a packet consumes that packet's size in bytes from credit. A source cannot send if credit is less than or equal to zero[13]. The destination will send credit messages to the source to increase credit and allow the source to send more data. If there is no outcast, an RCCC source can start at line rate by sending up to a BDP of "speculative packets". Where multiple sources send to one destination at the same time (incast), this will create congestion at the last-hop switch.

To deal with incast, RCCC leverages information available at the destination to make optimal, instantaneous changes to the transmission rate of each of the active sources, as demand to that destination changes. In detail, after the first RTT, the destination leverages backlog information from sources to schedule incoming traffic in the next RTT. It does so by sending credit control packets to the appropriate sources. The credit packets are control packets that are paced such that the data packets they elicit arrive at the destination's line rate. Typically, a credit packet will carry a few MTUs of credit, so credit packets themselves impose only a small overhead.

If there is an outcast, an RCCC source still starts with a BDP of speculative credit, but less than a BDP will be sent before the first ACK packet is returned. At this point the remaining speculative credit is removed so that the source does not burst beyond what the destination can cope with when other PDCs from the outcast stop sending.

When RCCC is enabled, ECN should be disabled on the last hop in the network, on the link from the last hop switch to the destination FEP. This permits more accurate load balancing based on the ECN signal to be performed, as last-hop congestion (which cannot be load balanced) no longer is conflated with the ECN load-balancing signal. ECN from the last hop is not required for congestion control, as RCCC directly manages congestion on the last hop.

When combined with UET's packet spraying and load balancing, the mechanisms described above are sufficient to obtain near-optimal performance in a fully provisioned network. Network core congestion can, however, appear when the network core is oversubscribed, either by construction or due to failures.

An additional mechanism is required to handle oversubscribed networks.

[13] It would be slightly more accurate if there had to be at least *nominal_pktsize* bytes of credit available, but it is simpler to implement if credit only has to be greater that zero to send. The one packet offset has negligible effect on performance.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

It is RECOMMENDED to enable NSCC to complement RCCC. NSCC depends on ECN to trigger a congestion window reduction, so disabling ECN on the last hop means that NSCC will be relatively insensitive to incast, leaving RCCC to more effectively manage the incast, while NSCC can still respond to congestion due to oversubscription.

When enabling NSCC is not possible or desirable, the RCCC rate control algorithm described in Section 3.6.14.7.1 MAY be used to handle oversubscribed networks.

With RCCC, the source notifies the destination of its backlog so the latter has an accurate view of the demand of the sources. On receipt of any trimmed or data packet, the destination knows the source's demand and can schedule credit.

RCCC benefits from packet trimming in switches to ensure that even under incast, the destination has an accurate view of the demand of the sources. On receipt of a trimmed packet, the destination knows the source's demand and can schedule credit.

To ensure that the destination knows how to divide its credit flow between sources so that credit is not wasted unnecessarily, the source sets the **pds.req_cc_state.credit_target** field in the PDS Request header. When work is posted to a PDC, this causes the CCC's backlog of work to be increased, and this in turn increases the **pds.req_cc_state.credit_target**. The **pds.req_cc_state.credit_target** field is cumulative; when **pds.req_cc_state.credit_target** is increased, this indicates to the destination that the source requires the amount of extra credit indicated by how much **pds.req_cc_state.credit_target** was increased by. This ensures that **pds.req_cc_state.credit_target** is resilient to the reordering of data packets.

3.6.14.1 CCC Identifiers

The CCC Identifier, *CCC_ID*, identifies a specific instance of RCCC or TFC. The *CCC_ID* is assigned by the source and carried in PDS Request packets in the **pds.req_cc_state.ccc_id** field. When RCCC or TFC is enabled, the **pds.type** field MUST be RUD_CC_REQ or ROD_CC_REQ. When these types are used, the **pds.req_cc_state** field is present and it carries **pds.req_cc_state.ccc_id** and **pds.req_cc_state.credit_target** fields, as shown in Section 3.6.9.1.

When there is a PDS Request to send by an initiator of a PDC, the PDS determines if there is an existing PDC and/or CCC:

- A lookup is done on PDC mapping tuple as defined in the PDS.
    - If there is an existing, available PDC, then the associated CCCs (one for forward direction and one for the reverse direction) are used.
        - AI Base implementations do not require the reverse direction CCC.
    - If there is an existing PDC but the implementation chooses to create another PDC to the same destination FEP, the CCCs from the existing PDC are used.
- If there is not an existing PDC then:
    - If the PDC is RUD, a lookup is done on {destination FEP address, traffic class}.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

        - If there is a match to existing CCCs (forward direction and reverse direction), then those CCCs are used.
        - If there is no existing CCC, then forward- and reverse-direction CCCs are allocated.
    - If the PDC is ROD, then allocate forward and reverse CCCs.

In summary, if there is an existing CCC, a new RUD PDC SHOULD use the existing CCC, and a new ROD PDC MUST NOT use the existing CCC. If there is no CCC, then a new CCC and *CCC_ID* are allocated. The *CCC_ID* is scoped such that a *CCC_ID* is unique at the source among CCCs sending to the same destination FEP. The same *CCC_ID* value MAY be used for PDCs to different FEPs. Similar to PDCIDs, the assigned *CCC_IDs* are locally scoped.

The *CCC_ID* SHOULD be used for both new requests and responses with payload larger than *Max_ACK_Data_Size*. That is, there SHOULD be one common pool of shared CCCs for both semantic requests and semantic responses with payload.

When a PDS Request arrives at the destination, which is the target of the PDC, the PDS determines if there is an existing CCC:

- If the **pds.flags.syn** field is set, a PDC lookup is attempted using the **pds.spdcid** and **ip.src_addr** FEP address:
    - If the PDC is already established, the associated CCC is used.
    - If the PDC has not been established, then a new PDC will be established.
        - A lookup on {**ip.src_addr**, **pds.ccc_id**} is done to assign the CCC. This lookup MUST be performed against both source and destination CCCs at the target of the PDC. Lookups for source and destination CCCs are performed independently, and allocations can occur independently.
        - If the lookup finds a match, the PDC is associated with the existing CCC instance.
        - If the lookup does not find a match, a new CCC is allocated.
- If **pds.flags.syn** is cleared, then a lookup on **pds.dpdcid** is done:
    - On the forward direction (packets arriving at target), the **pds.dpdcid** field is used to find the associated CCCs that were associated with the PDC when it was established.

Note that the lookup used to associate a CCC with a PDC at the destination does not include the **pds.spdcid** field, meaning that multiple PDCs may share a CCC. Also, the PDC type and traffic class are not included, as the source MUST NOT use the same *CCC_ID* for different CCC instances to the same destination FEP. The traffic on a forward path of one PDC MAY share a CCC with the return path of another PDC. This is accomplished by the lookup of both source and destination CCC state when the CCC is associated with the PDC.

When a new PDC is established that would be associated with an existing CCC, if all the PDCs associated with the CCC are already in the process of closing, then the source MUST NOT use the existing CCC and instead MUST allocate a new CCC. This prevents a potential deadlock where a source has no remaining credit to start a new PDC and no active PDC on which to request such credit.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

When a new PDC is established that is associated with an existing CCC, and if that CCC has no credit and no additional credit has been requested for that CCC, a source SHOULD send a Credit Request CP requesting more credit. This Credit Request CP MUST be sent using any of the previously existing PDCs associated with that CCC.

Informative Text:
The new PDC cannot be used to request credit until it is established, and Credit Request CPs are not allowed to establish a PDC at the target. PDS requests, which could establish the PDC and request credit, cannot be sent when the CCC does not have credit.

At the source, one or more PDCs from a source FEP to a destination FEP are associated with a CCC. Credit is allocated to a CCC so that any PDC in that CCC can use that credit to send. The destination (receiver) sends credit either in ACK packets or in Credit CPs. These ACKs or Credit CPs are sent associated with a PDC, but the credit can be used by any PDC associated with the same CCC.

The **pds.req_cc_state.credit_target** field indicates the desired credit associated with the aggregate of PDCs managed by the CCC. For example, when packets are posted to any PDC in the CCC, the **pds.req_cc_state.credit_target** field is incremented by the size of those packets.

In order for the destination to maintain state on a per CCC basis, the source includes a *CCC_ID* in the **pds.req_cc_state.ccc_id** field. The *CCC_ID* identifies the CCC at the destination to which this *Credit_Target* should be applied. This keeps the scheduler using the credit on the source aligned with the CCC instance allocating credit at the destination.

An RCCC CCC MUST be shut down when the last PDC using it is closed.

3.6.14.2 RCCC Source State

This section contains pseudocode describing the RCCC algorithm. Implementations are not required to precisely follow this pseudocode but SHOULD aim to achieve the same behavior.

RCCC defines the following global constants, with the associated default values:

As with NSCC, `BDP` defaults to:

```
BDP = min(sender.linkspeed, receiver.linkspeed) * config_base_rtt
```

`BDP` is in units of bytes, so *linkspeed*s above should be measured in units of bytes/second. The destination *linkspeed* will typically be the same as the source *linkspeed*. If it is unknown, use the source's *linkspeed*.

```
MaxWnd = 1.5 * BDP
```

MaxWnd is in units of bytes.

On CCC creation, the following CCC local state is instantiated:

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Table 3-100 - RCCC Source State

Name	Type	Default	Description
ccc.credit	unsigned integer	*MaxWnd*	Amount of credit currently at source for this CCC. (units: bytes)
ccc.credit_target	unsigned integer	0	Cumulative credit target source has requested (or is about to request). Represents cumulative credit requested since the CCC was established. (units: bytes)
ccc.cumulative_credit	unsigned integer	0	Cumulative credit value the destination has sent to the source since the CCC was established.
ccc.credit_target_sent	boolean	FALSE	TRUE if source has told the destination its current credit target.
ccc.speculating	boolean	TRUE	TRUE if the source is in the startup phase using speculative credit.

3.6.14.3 RCCC Credit Timer

RCCC requires a credit timer at the source to cover a corner case where credit packets can be lost. When a sender enters an active state — that is, it has no credit available but has data to send — it starts a timer with a timeout equal to the retransmit timeout. When the timer expires, the RCCC.OnSourceCreditTimer() pseudocode is executed.

3.6.14.4 RCCC Source Algorithm

The RCCC algorithm processes the CCC API events as follows:

```
RCCC.OnACK():
      #A PDC is always present if we received an ACK
      pdc_present=TRUE
      RCCC.stopSpeculating(pdc_present)

RCCC.OnNACK(pdc_present):
      RCCC.stopSpeculating(pdc_present)
      #waiting_rtx will have changed, so the pull target will have changed
      ccc.credit_target = computeCreditTarget()
      ccc.credit_target_sent = FALSE

RCCC.OnInferredLoss(pdc_present):
      RCCC.stopSpeculating(pdc_present)
      ccc.credit_target = computeCreditTarget()
      ccc.credit_target_sent = FALSE
```


Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

```
RCCC.OnSend(nominal_pktsize):
      ccc.credit -= nominal_pktsize
      ccc.credit_target_sent = TRUE

RCCC.OnCreditUpdate(Credit)
      #A PDC is always present when receiving a credit update
      RCCC.stopSpeculating(pdc_present=TRUE)
      if Credit > ccc.cumulative_credit:
            ccc.credit += (Credit - ccc.cumulative_credit) << 8
            ccc.cumulative_credit = Credit
      update_state()

RCCC.OnNewData():

      credit_target = computeCreditTarget()
      if credit_target != ccc.credit_target
            ccc.credit_target_sent = FALSE
            ccc.credit_target = credit_target

RCCC.CanSend():
      if ccc.credit > 0:
            return TRUE
      if ccc.credit_target_sent = FALSE
         or (ccc.credit_target_sent = TRUE
             and computeCreditTarget()> ccc.credit_target):
            #insufficient credit available and destination does not
            #know about our backlog
            ccc.credit_target = computeCreditTarget()
            #send Credit Request CP requesting credit.
            send_credit_request(ccc.credit_target)
            ccc.credit_target_sent = TRUE
      return FALSE

RCCC.stopSpeculating(pdc_present):
      if ccc.speculating:
            ccc.speculating = FALSE
```


Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

```
            if ccc.credit > 0:
                  ccc.credit = 0
      #If no PDC is present, preserve an MTU of credit because
      #we have to do something to establish the PDC
      if pdc_present == FALSE
            ccc.credit = MTU
```

RCCC.computeCreditTarget():

```
      #if nscc is not used, both ccc.max_wnd and ccc.cwnd will be MaxWnd
      #in this calculation
      delta = min(ccc.backlog + ccc.rtx_backlog, ccc.max_wnd, ccc.cwnd + mtu)
            – ccc.credit
            if ((ccc.speculating or delta > 0) and delta < mtu))delta = mtu;
      return (ccc.cumulative_credit + (delta >> 8))
```

RCCC.OnSourceCreditTimer():

```
      #either we don't have any work to do, we have pending packets which
      #will trigger timeouts or we already have credit.
      if ccc.state != READY:
            return
      RCCC.stopSpeculating()
      send_credit_request(ccc.credit_target)
      ccc.credit_target_sent = TRUE
```

3.6.14.5 RCCC Destination State

A "sender" as referenced in sections 3.6.14.5.1 and 3.6.14.5.2 is associated with a sending FEP and an associated *CCC_ID* (Section 3.6.14.1). All PDCs from the same source associated with the same CCC are mapped to the same sender in the pseudocode of 3.6.14.6. If more than one CCC is established from the same sending FEP (as indicated by CCD_ID), then these CCCs correspond to separate senders.

3.6.14.5.1 RCCC Global Destination State

On FEP initialization, the following state is established:

Table 3-101 - RCCC Global Destination State

Name	Type	Default	Description
active_senders	List of sources	empty	This is a list of sources that have currently requested additional credit that has not yet been satisfied.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Name	Type	Default	Description
idle_senders	List of sources	empty	This is a list of sources that have not currently requested additional credit.
credit_timer	Timer	unset	Timer used to trigger the sending of Credit CPs.
credit_rate	floating point	1	Used to dynamically modulate the total credit rate sent by the destination.

3.6.14.5.2 Per-Source Destination State

On CCC start (as indicated by the establishment of a PDC with a new CCC ID) the state shown in Table 3-102 is established and associated with the source.

Table 3-102 - RCCC Per-Source Destination State

Name	Type	Default	Description
sender.cumulative_credit	unsigned integer	0	The highest credit given to this source. (units: 256 bytes)
sender.credit_target	unsigned integer	0	Largest value of credit_target (corresponding to cumulative demand) from this source. (units: 256 bytes)

3.6.14.6 RCCC Destination Algorithm

On receipt of a data packet or trimmed data packet, *OnRX()* is performed.

```
OnRX(pkt):
      if pkt.credit_target > sender.credit_target:
            sender.credit_target = pkt.credit_target
            if sender not in active_senders
               and sender.cumulative_credit < sender.credit_target:
                  #previously inactive source has unsatisfied demand.
                  #insert the source at the end of the source list
                  active_senders.insert_at_back(sender)
                  if sender in idle_senders:
                        idle_senders.remove(sender)
      else if pkt.type == credit_request
            # if a credit_request was received with a credit_target <=
            # sender.credit then send a credit packet with current value
            # of sender.cumulative_credit
            send credit(cumulative_credit = sender.cumulative_credit)
```

OnCreditTimer() is designed to issue credit at a rate where the amount sent within a given time period matches the amount of data the destination can receive in that time period. Typically, *k* is a small constant such as four, so that each credit message allows the transmission of four MTUs of data. Smaller

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

values of *k* result in less bursty sharing of incoming capacity but require a higher volume of control messages.

Only a single credit timer is needed per FEP[14] for a traffic class to handle any number of incoming PDCs in that traffic class at the FEP.

```
OnCreditTimer (once every k * MTU/(linkspeed*credit_rate) seconds):
      if active_senders.size() > 0:
            #round robin across active sources (other sharing strategies
            #possible, not shown)
            sender = active_senders.pop_front()
            sender.cumulative_credit =
            min(sender.credit_target, sender.cumulative_credit + k * mtu)
            sendCreditControlMessage(sender, sender.cumulative_credit)

            if sender.cumulative_credit < sender.credit_target:
                  active_senders.push_back(sender)
            else:
                  #source has no more demand, add to idle_senders list
                  idle_senders.push_back(sender)
      else if idle_senders.size() > 0:
            #We have no active sources, send credit to idle sources to
            #allow them to speculative start if needed.
            sender = idle_senders.pop_front()
            sender.cumulative_credit += k * mtu
            sendCreditControlMessage(sender, sender.cumulative_credit)

            if sender.cumulative_credit < sender.credit_target + MaxWnd:
                  #only send up to MaxWnd credits
                  idle_senders.push_back(sender)
```

When sending a Credit CP, the **pds.payload.credit** field is set to the value of *sender.cumulative_credit*.

3.6.14.7 Dynamic Credit Rate

The destination credit rate is typically constant in normal operations, being set to a value marginally smaller than *linkspeed* (e.g., 99%). In certain cases, however, it is desirable for the destination to dynamically reduce its total credit rate. To enable this, RCCC can modify the *credit_rate* global

[14] Where multiple FEPs are co-located in such a way as to share CCCs, only one timer is needed per TC per group of FEPs sharing CCCs.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

destination variable. The range of values of *credit_rate* is (0,1), where 1 is the default and means that credit is sent at (near) *linkspeed*. The credit rate must be lower, bounded to a value strictly larger than 0.

Two cases where reducing the credit rate is useful is described in sections 3.6.14.7.1 and 3.6.14.7.2, but there may be other cases where an implementation might choose to dynamically adjust the credit rate. An RCCC destination MAY dynamically change the credit rate for any implementation-specific purpose.

3.6.14.7.1 RCCC Destination Flow Control

If a receiving FEP is unable to keep up with the arrival rate of data, a backlog of data will build at the destination. NSCC destination flow control uses *Rcv_Cwnd_Pend* to modify NSCC's *cwnd*, but RCCC does not use this mechanism.

To control the size of the backlog, an RCCC destination MAY adapt the *credit_rate* to reduce the arrival rate of data from all sources. UET-CC does not mandate a specific algorithm to set *credit_rate*, as destination congestion is implementation specific, but the following proportional control algorithm MAY be used.

The *max_backlog* variable is set to the maximum buffer size available. The *crtBacklog()* function returns the size of the current backlog of received data in bytes.

```
OnRx(pkt) [or periodically]:
      if crtBacklog() < MaxWnd:
            credit_rate = 1
      else if crtBacklog() > max_backlog * 95 / 100:
            credit_rate = 0.001
      else
            credit_rate =
                (max_backlog - crtBacklog())/(max_backlog - MaxWnd)
```

3.6.14.7.2 RCCC Aggregate Rate Control

In multi-tier oversubscribed networks, there can be cases where the load on the network core exceeds capacity. In such cases RCCC might provide suboptimal performance due to receivers providing credit at line rate, which in turn allows senders to transmit at line rate in aggregate to that receiver. One option to avoid this issue is to enable NSCC in conjunction with RCCC, as described in Section 3.6.14.

When enabling NSCC together with RCCC is not possible or not desirable, the receiver MAY reduce its total credit dynamically, based on observed congestion for its incoming traffic. In this case, the algorithm described in this specification SHOULD be used.

3.6.14.7.2.1 Goals and Approach

The goal of this algorithm is to avoid persistently trimming packets in oversubscribed networks and to share fairly the oversubscribed network core under persistent congestion.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

The RCCC oversubscribed congestion control algorithm is derived from HPCC [57]. HPCC is a sender-based algorithm that computes a congestion window based on network information. It relies on in-band network telemetry to compute the maximum queue depth across all hops and maximum relative bandwidth utilization across all hops.

Informative Text:
HPCC's key measure of network congestion is *U*, computed as an EWMA of *max(queue_length) / BDP + utilization* where *queue_length* and *utilization* are instantaneous values measured using in-band network telemetry by data packets and then reflected in ACK packets. The algorithm adjusts the *CWND* to control *U* to reach a configured setpoint (*U_Target*).

RCCC adapts HPCC to work in the context of UET and at the receiver. The key modifications are:

- Instead of updating the sender congestion window, it modifies the credit rate of the receiver.
- Instead of requiring in-band network telemetry, RCCC uses congestion signaling when available.
- When CSIG is not available, ECN and packet trimming are used to estimate queue length and network utilization. The presence of the CSIG tag in incoming packets is used to select between CSIG- and ECN-based operation on a per-packet basis.
- HPCC needs to carry all the in-band telemetry information to the sender to enable it to make congestion control decisions. RCCC uses the information at the receiver, and thus does not need to relay the information to the sender; hence no protocol changes are needed.

CSIG provides only bottleneck hop information instead of per-hop information, but this is sufficient for congestion control. RCCC can use the optional maximum normalized queue depth CSIG signal (*max(nQD))* or the mandatory maximum hop delay (*maxDelay)*, which provides the maximum latency a packet experienced in any switch along the path (this value is quantized).

The latter is directly correlated with queue length but is a lagging signal. These signals have been found to work similarly; they do, however, require different configuration parameters values to work well.

HPCC uses network utilization estimation to be able to operate without standing queue (thus slightly underutilizing the network) and to enable fast increases when utilization is low. In the context of RCCC, the aim is to fully load the network core, so the target utilization will be set to 1.5-1.75 of base RTT, resulting in an operating point similar to NSCC (0.75 base RTT setpoint). An approximation of utilization = 1 is therefore used, at the cost of slower increases in underutilized networks. Preliminary experiments using this approach demonstrate strong performance.

To improve the accuracy of utilization information and to enable faster increases under low network load, the *min(ABW/C)* CSIG signal may be used. Because each packet can only carry one

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

type of CSIG tag, a sender would send a fraction of its packets with the *max(Delay)* signal and another fraction with the *min(ABW/C)* signal. However, this approach increases controller complexity. The controller would require an additional configuration parameter (i.e., the fraction of *min(ABW/C)* packets), reducing the frequency of obtaining the latency signal, and potentially creating issues with multiple traffic class scenarios, without clear corresponding benefits. Therefore, this approach is not considered further.

The last difference to HPCC is using ECN to estimate queue depth. The key observation is that the ECN marking rate is directly derived from queue length by routers. The receiver can observe the (average) ECN marking rate, and armed with knowledge of the ECN marking parameters (see Section 3.6.17), can infer what the queue length is. In particular, if the receiver knows ECN configuration parameters *queue_low.min_thresh* and *queue_low.max_thresh*, it can measure the average ECN marking rate and reverse engineer the queue length as follows:

- If the average marking rate is ~0, queue length must be <= *queue_low.min_thresh*
- If the average marking rate is 1, queue length must be >= *queue_low.max_thresh*
- If 0 < *ecn_rate* < 1, queue length = *queue_low.min_thresh* + *ecn_rate* * (*queue_low.max_thresh* – *queue_low.min_thresh*)

As long as the setpoint for HPCC is set in between *queue_low.min_thresh* and *queue_low.max_thresh* , the receiver has a direct way to determine the queue length. Experiments show that by using an EWMA on the ECE bit at the receiver, the ECN marking rate can be tracked with very good accuracy (median error less than 0.3%), which means that the resulting error in queue depth estimation is also low.

The RCCC oversubscribed congestion control algorithm has two parts: a part that is run on arrival of data or trimmed packets at the receiver, and another one that runs the congestion control algorithm that may run less frequently. The full pseudocode is provided in Section 3.6.14.7.2.2.

On each packet arrival, the algorithm updates the *average_delay* value with an exponential moving average, using information from CSIG tags, ECN marks, and packet trim status. If the CSIG tag is present and has type *max(Delay)*, the quantized signal values are de-quantized by looking up the metric value and the bucket ranges provided by the control plane using the inverse of the process described in Section 4.2.4.1.5 for the compact tag format and in Section 4.2.4.1.6 for the wide tag format. The min value of each bucket is used to represent the resulting value of the de-quantization process.

Once every *CC_INVOCATION_DELTA* microseconds, where *CC_INVOCATION_DELTA* is a configuration parameter, the congestion control algorithm is invoked, which updates the receiver credit rate (See Table 3-101 in Section 3.6.14.5.1). The *CC_INVOCATION_DELTA* parameter offers a tradeoff between responding quickly and reducing the cost of congestion control invocations. By setting the *CC_INVOCATION_DELTA* parameter to small values, the congestion control algorithm can be invoked on a per-packet basis, but this requires more processing and might become a performance bottleneck at

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

high packet rates. To reduce computation costs, *CC_INVOCATION_DELTA* can be set to cover multiple packet arrivals; however, to ensure stability it SHOULD NOT be set to more than the base RTT. The suggested value for *CC_INVOCATION_DELTA* is 1 µs, which offers a good tradeoff between computation and responsiveness.

On FEP initialization, the following state is initialized for oversubscribed congestion control. These values are maintained once for all CCCs from all sources. They are not per CCC.

Table 3-103 - RCCC Global Destination State for Aggregate Rate Control

Name	Type	Default	Description
reference_rate	float	1	*reference_rate* is only updated periodically (at most once per 1 RTT).
increment_stage	unsigned int	0	A counter of the number of additive increase phases performed since the last multiplicative increase.
average_delay	unsigned int	0	Estimated queueing delay.
next_rate_change	int	0	Time when the *reference_rate* will be updated.
last_cc_invocation	int	0	Time when the CC was last invoked.

The oversubscribed CC algorithm uses the following configuration parameters. The default parameter values listed in Table 3-104 are chosen assuming the network uses the recommended switch configuration values in Section 3.6.17.

Table 3-104 - RCCC Aggregate Rate Control Parameters

Name	Type	Default	Description
min_rate	floating point	0.05	The minimum credit rate.
min_thresh_delay	int	0.2 * base_rtt	Queueing delay experienced by a packet seeing a *queue_low.min_thresh* queue.
max_thresh_delay	int	0.8 * base_rtt	Queueing delay experienced by a packet seeing a *queue_low.max_thresh* queue.
trim_delay	int	2 * base_rtt	Maximum single-hop queueing delay for a data packet in the network.
rai	floating point	0.025	Additive increase constant.
target_u	floating point	1.5	Target network utilization.
MAX_INCREMENT_STAGE	int	0	Number of additive increase stages before multiplicative increase.
CC_INVOCATION_DELTA	int	1 us	Time between two consecutive invocations of the CC algorithm.
UPDATE_INTERVAL	float	2.5	The length of time (as a multiplier of the base RTT) to wait before updating the reference rate.
g	float	1/8	EWMA gain used to track average queue delay.
f	float	1/8	EWMA gain to track ECN marking rate. 1/8 used for linkspeed < 400Gbps, 1/16 for BDPs >= 400Gbps (for 4k MTU).

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

3.6.14.7.2.2 RCCC Aggregate Rate Control Algorithm

When RCCC Aggregate Rate Control is enabled, the following *OnRX(pkt*) code must be run on each packet arrival in addition to the *OnRX(pkt)* packet processing described in Section 3.6.14.6. The code updates the *credit_rate* (a FEP global variable, see Table 3-102), which is used to control the frequency of credit creation at the receiver as specified in the *OnCreditTimer()* pseudocode in Section 3.6.14.6).

```
OnRX(pkt):
    if (pkt.csig.t== "max(Delay)"):
        //UPDATE average delay based on CSIG signal maxDelay
        average_delay = g * quant_reverse_lookup_for_maxDelay(pkt.csig.s) +
                        (1-g) * average_delay
    else:
        est_delay = 0
        if (pkt is trimmed):
            est_delay = trim_delay
        elif (pkt.ECN marked):
            est_delay = max_thresh_delay
        else :
            est_delay = min_thresh_delay

        average_delay = f * est_delay + (1-f) * average_delay

    delta = now() - last_update

    if (delta > CC_INVOCATION_DELTA):
        DoOversubscribedCC()
        last_update = now()


DoOversubscribedCC():
    u = average_delay / base_rtt + 1

    update_interval = UPDATE_INTERVAL * base_rtt +
                      max(average_delay, base_rtt)
    updateWc = now() >= next_rate_change

    if (u >= target_u || increment_stage >= MAX_INCREMENT_STAGE):
        credit_rate = reference_rate / (u / target_u) + rai

        if (credit_rate > 1):
            credit_rate = 1
        if (credit_rate < min_rate):
            credit_rate = min_rate

        if (updateWc):
            increment_stage = 0
            reference_rate = rate
            next_rate_change = now() + update_interval
    else:
        credit_rate = reference_rate + rai

        if (credit_rate > 1):
            credit_rate = 1
        if (credit_rate < min_rate):
            credit_rate = min_rate
```


Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

```
    if (updateWc):
        increment_stage++
        reference_rate = rate
        next_rate_change = now() + update_interval
```

3.6.15 Transport Flow Control (TFC)

Transport Flow Control is provided to allow a destination to control the amount of data transmitted from one or more sources. TFC uses a point-to-point credit mechanism, with the destination allocating credit and the source consuming credit. A source MUST NOT transmit a packet if it does not have credit.

For the purposes of transmission of credit from the destination to the source, TFC is aligned with the RCCC credit mechanism and uses the same header fields. It uses a 24-bit **pds.credit** field that is carried from the destination to the source in either PDS Credit CPs or using the **pds.cc_type**=CC_CREDIT header in PDS acknowledgements. When using **pds.cc_type**=CC_CREDIT with TFC, **pds.ooo_count** MUST be set to 0xFFFF, indicating the field is invalid and **pds.ooo_count** MUST be ignored on receipt.

As with RCCC, the **pds.credit** field is a monotonically increasing number that wraps at its maximum value. An increase in the value of the **pds.credit** field provides additional credit to the source. This is used to provide tolerance to packet loss. A single bit in the **pds.credit** field represents 256 bytes of credit. When a Credit CP or an ACK conveying credit arrives at the source, the source's state is updated in a similar way to RCCC:

```
TFC.OnCreditUpdate(Credit):
if Credit > ccc.cumulative_credit:
ccc.credit += (Credit - ccc.cumulative_credit) >> 8
ccc.cumulative_credit = Credit
```

As with RCCC, the source is permitted to send if *ccc.credit* > 0. Note that being able to send is not dependent on packet size, so if *ccc.credit* is greater than zero, the destination MUST be able to accept an MTU-sized packet.

TFC differs from RCCC credit in that a TFC source can retransmit without using credit. The rationale for this difference is that the role of TFC credit is to prevent the source from sending data that the destination cannot buffer, whereas the role of RCCC is to manage network bandwidth. With TFC, a packet can be safely retransmitted without needing additional credit because the corresponding buffer at the destination must have already been allocated for the original packet.

Note that retransmission at the PDC level is conceptually separate from any retransmission that may occur at the link level using link-layer retry (LLR).

Because TFC controls access to receive buffers, different destinations may have different requirements when it comes to buffer management. Some destinations may allocate buffers for entire MTU-sized packets, so the transmission of a packet must then consume an MTU of credit. Other destinations may

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

allocate buffers in cells, whereby the arrival of a smaller packet will consume fewer cells than the arrival of a larger packet.

To accommodate different buffer allocation strategies, the approach taken by TFC is to allow the destination to manage a receive buffer as a number of "cells," where each cell is of size *tfc_dest_cell_size*. As credit is allocated in units of 256 bytes, a range of cell sizes from 256 bytes to 9 KB, in multiples of 256 bytes, MUST be supported. The *tfc_dest_cell_size* variable defaults to 256 bytes, the same as a unit of credit. When the source transmits a packet, the number of cells required to store the packet is calculated, and hence the number of bytes of credit consumed, *credit_used*, is calculated.

A TFC source can send if *TFC.CanSend()* returns TRUE:

```
TFC.CanSend(rtx):
      if rtx or ccc.credit > 0:
            return TRUE
      else
            return FALSE
```

Sending a packet reduces *ccc.credit* as follows:

```
TFC.OnSend(credit_used, rtx):
      if rtx == FALSE:
            ccc.credit -= credit_used
```

In *CanSend()* and *OnSend()*, *rtx* indicates whether the packet is a retransmission. Retransmissions do not require credit to send and do not consume credit when sending.

Because it may be possible for packet sizes to change in transit, and as destinations may wish to store additional data along with an arriving packet, *tfc_pkt_size* is defined as the UET payload size + *tfc_pkt_overhead* bytes. The source and destination must use the same value of the constant *tfc_pkt_overhead*. Constant *tfc_pkt_overhead* defaults to 256 bytes, the same as one unit of credit. It is RECOMMENDED that *tfc_pkt_overhead* is configurable between 0 and 256 bytes, but an implementation MAY use a constant size of 256 bytes — in which case a destination may need to allocate additional buffering if larger encapsulations are used.

The *tfc_pkt_overhead* parameter is intended to cover any additional memory required at the destination to store the arriving packet. This may include space to store the Ethernet, IP, UDP, and SES headers; FCS; and any additional metadata the destination may store along with the packet. The destination MUST be able to store any valid arriving packet for which it has allocated credit.

TFC manages a unidirectional flow of data from a sending FEP to a receiving FEP. As the requirements for *tfc_dest_cell_size* and *tfc_pkt_overhead* are primarily determined by the destination FEP, when PDCs are established in both directions between a pair of FEPs, the values of *tfc_dest_cell_size* and *tfc_pkt_overhead* may be different in one direction from the other.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

The values of *tfc_dest_cell_size* and *tfc_pkt_overhead* cannot be modified on an active PDC. Either their default values MUST be used, or any non-default values MUST be configured or communicated out of band prior to data being sent on the PDC.

3.6.15.1 Examples of credit_used calculation

The values of UET payload size, *tfc_dest_cell_size* and *tfc_pkt_overhead* determine the amount of credit required to send a packet.

For example, to send an UET payload size of 4096 bytes with a *tfc_dest_cell_size* set to 512 bytes and default *tfc_hdr_size*, then:

```
tfc_dest_cell_size = 512B (configured value)
tfc_pkt_overhead = 256B (default value)
tfc_pkt_size = UET payload + tfc_pkt_overhead
credit_used = ROUNDUP(tfc_pkt_size/tfc_dest_cell_size)*tfc_dest_cell_size
```

Thus with a 4096 byte UET payload, *tfc_pkt_size* is 4096 + 256 = 4352 bytes.

Sending a *tfc_pkt_size* of 4352 bytes requires 4352/512 = 8.5 cells; rounded up this requires 9 cells, and so *credit_used* = 4608 bytes.

Alternately, a destination may wish to maintain data structures for packet buffers on a per-packet basis. In such a case, *tfc_dest_cell_size* would be set to the maximum value of 9216 bytes. Sending the same 4 KB packet would use the same *tfc_pkt_size* of 4352 bytes, but this would require only one 9126-byte cell. The *credit_used* would then be 9216 bytes. Such a destination would then allocate *n* * 9216 bytes of credit for *n* packet buffers.

3.6.15.2 TFC Credit Initialization

When a TFC PDC is created, the *ccc.credit* value MUST be set to *tfc_pkt_overhead* or a larger configured value, but it cannot be less than *tfc_pkt_overhead*. The intent of setting the initial credit to *tfc_pkt_overhead* is to permit the source to send an SES NO_OP packet that contains a credit request in the **pds.req_cc_state.credit_target** field, allowing the source to request enough additional credit to send data. If an implementation reports a *tfc_pkt_overhead* of 0, a larger configured value should be used. The rationale for using an SES NO_OP instead of a Credit Request CP to request initial credit is that an SES NO_OP can bring up a PDC, whereas a Credit Request CP cannot. The ACK for the SES NO_OP may carry the available credit, or a separate Credit CP may be used.

Once a PDC is established, if more credit is required than initially requested, the source can convey this to the destination using the **pds.req_cc_state** field in PDS Request packets or the **pds.payload** field in Credit Request CPs. The **pds.payload** field and Credit Request CP is used in the same way as RCCC. It carries a **pds.payload.ccc_id** to identify the CCC context at the source and a **pds.payload.credit_target** field, which carries the cumulative amount of credit requested since the CCC was created.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

As with RCCC, multiple TFC PDCs may be controlled by the same CCC, or each PDC may be given its own CCC. It is RECOMMENDED that the decision to share a TFC CCC between PDCs be a configurable option. When a CCC controls multiple PDCs, credit allocated to one of the PDCs controlled by the CCC may be used by any of the PDCs controlled by the CCC.

The source controls whether multiple PDCs share a CCC. The PDS Request header carries the **pds.req_cc_state.ccc_id** field. If multiple PDCs between a pair of FEPs are assigned the same *CCC_ID*, then the destination MUST associate these PDCs with the same CCC. If different *CCC_IDs* are used, then the PDCs MUST be associated with independent CCC instances.

3.6.15.3 TFC CCC Shutdown

When the last PDC sharing a TFC CCC is shut down, the CCC is also shut down. Any credit allocated for the CCC is no longer usable by the source to send new data, and any buffers at the destination reserved for the use of that credit may be freed for other uses.

At the source, a CCC MUST NOT be reused for a new PDC after the last PDC using that CCC has entered the QUIESCE state in the Single PDC Close State Machine of Figure 3-57. This procedure prevents a potential race where the destination thinks the last PDC using a CCC has been shut down and frees buffers, but the source attempts to start a new PDC using that CCC.

3.6.15.4 TFC CCC Credit Timer

Like RCCC, TFC CCC requires a credit timer at the source to cover a corner case where packets containing credit can be lost. When a sender enters an active state — that is, it has no credit available, but has data to send — it starts a timer with a timeout equal to the retransmit timeout. When the timer expires, the *TFC.OnSourceCreditTimer()* pseudocode is executed.

```
TFC.OnSourceCreditTimer():
      #either we don't have any work to do, we have pending packets which
      #will trigger timeouts or we already have credit.
      if ccc.state != READY:
            return
      send_credit_request(ccc.credit_target)
```

3.6.16 Multipath Path Selection

This section describes the path selection algorithms used to load-balance the network. It is RECOMMENDED that a UET endpoint performs multipath load balancing for RUD and RUDI traffic, but the choice of precise implementation is left to implementations.

The protocol relies on multipath forwarding to distribute packets from flows across multiple network paths. With the exception of the single-path ROD service, packet spraying is typically done using per-packet ECMP, which uses hashing of network header fields to spray packets across many paths. This approach can achieve higher throughput and lower flow completion times than what can be achieved by a single path.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Some network switches can perform packet spraying themselves, rather than hashing the header fields to determine the path. It is RECOMMENDED that such behavior not be enabled for UET traffic if path-aware multipath spraying (Section 3.6.16.4) is used, as it prevents UET getting a clean signal as to which paths are congested.

Despite attempts to evenly distribute the traffic, some path-specific congestion is likely to occur. Hashing is not guaranteed to spread the packets evenly. The presence of other traffic that is not sprayed may impact available bandwidth on some paths. In addition, failures may render a path unavailable. Improved throughput and reduced flow completion times can be achieved if the sources actively balance traffic.

3.6.16.1 Path Entropy

When packet spraying is employed, packets are sprayed across multiple paths by varying the value of an entropy field that carries the *Entropy* value (EV) in packet headers (see Section 3.6.10.1). The entropy field is used by switches to determine which ECMP path each packet takes to the destination.

Three modes of operation are permitted:

- Single path (used only by ROD). All packets in a CCC use the same EV for some period of time, before the EV is changed.
- Oblivious spraying. The packets of a CCC are sprayed across many paths by changing the EV through a large space (typically at least 256 values). EVs are chosen pseudo-randomly with the goal of spreading traffic evenly across all possible paths.
- Path-aware spraying. The packets of a CCC are sprayed across many paths by adaptively changing the EV to avoid congested paths, **based on** feedback provided by the destination as to which EVs saw congestion**.**

3.6.16.2 Single-Path Entropy

At PDC startup, the reliability module generates an *AllocateCCC()* event with *sprayed*=FALSE. The CCC chooses an entropy value to use for the packets of the flow. To ensure that, in the absence of route changes or packet loss, packets arrive in order, only a single entropy value is used at one time for this CCC.

If network congestion is observed and it is determined that the congestion is not occurring at the final link to the destination (which cannot be avoided by re-routing), then the CCC MAY change the entropy value it uses. To minimize the impact of reordering and any potential retransmission, it is RECOMMENDED that the CCC either change the entropy value when the flow has become idle, or if there is no idle period, that it does not change the entropy value more often than every *t_reroute* RTTs. How long a flow should be idle before changing entropy value is a configuration choice. It is safe to switch whenever there are no packets in transit, but such a conservative choice will not necessarily lead to improved performance. It is RECOMMENDED that the default value of *t_reroute* be set to at least 10 RTTs and that this value be configurable.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

When more than one single-path PDC exists between a source and destination, PDS should request the allocation of different CCCs for them. The different CCCs between the same pair of FEPs SHOULD coordinate to allocate different entropy values to reduce the frequency that CCCs use the same single path.

Informative Text:
The motivation for using different but coordinated CCCs is to reduce the impact of flow collisions. A flow collision on one PDC is unlikely to affect another PDC if they use different EVs, and congestion state from one single-path EV is not a good predictor of available capacity using a different single-path EV.

3.6.16.3 Oblivious Multipath Spraying

The source maintains a space of entropy values to use for the CCC, but it does not keep information about which entropy values are less congested than others. A typical size for the entropy space is 256 values, but other sized spaces MAY be used. The source SHOULD use entropy values from the entropy space in a pseudo-random order, with the goal being to use all the entropy values once before repeating. It is beneficial for each source to independently re-randomize the pseudo-random sequence periodically to reduce the probability of multiple sources synchronizing on sequences that hash to similar path sequences.

How the source chooses a pseudo-random sequence is implementation defined. A simple modular counter, while not pseudo-random, suffices for protocol interoperability and so is permitted, but it relies on switch ECMP hash functions to provide pseudo-random load balancing. This may be sufficient in many deployments. If so, the counter SHOULD be initialized randomly to avoid synchronization.

Oblivious spraying is most effective when combined with packet trimming. With packet trimming, on congestive loss, the source will receive a NACK quickly from the destination and will retransmit the packet as soon as congestion control permits. The retransmitted packet will, with high probability, take a different path. The use of a relatively low trimming threshold of around one BDP prevents queues growing unnecessarily large, and as trimmed packets use little capacity at the congestion bottleneck and elicit fast retransmissions on a different path, oblivious spraying with trimming can do a good job of actively load balancing the network.

Oblivious spraying without packet trimming is discouraged, as it will result in load balancing that is less even than path-aware spraying. However, in a network without packet trimming, a source MAY choose to employ oblivious spraying if it cannot hold enough per-path state.

3.6.16.4 Path-aware Multipath Spraying

As with oblivious spraying, the source maintains a space of entropy values to use for the CCC and uses those entropy values in a pseudo-random order. The size of this space SHOULD be chosen adaptively so that each value used can be considered for reuse within less than two RTTs. This approach is to ensure that load balancing is not responding to out-of-date path congestion information.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

In reverse-direction ACK and NACK packets, the destination feeds back the *Entropy* from the received packet together with the **pds.flags.m** ("marked") flag that indicates that this path experienced congestion. The **pds.flags.m** is set when the arriving packet that triggers an ACK or NACK has the **ip.ecn.ce** bit set.

The destination MAY use other forms of telemetry to determine whether a path is congested.

On receipt of an ACK or NACK packet with the **pds.flags.m** flag set, the source SHOULD ensure that the indicated *Entropy* is not reused for at least one round-trip time. An exception is when ACKs or NACKs indicate that the majority of paths are congested. Under such circumstances, the load balancing signal has become saturated, and skipping paths will no longer be effective. The fraction of paths that must be congested before the destination decides the load balancing signal is saturated SHOULD be a configuration option. A reasonable default value for this configuration option is 50% of paths congested. This value is a tradeoff: It is undesirable to reduce the entropy pool unnecessarily during network oversubscription or incast; however, it is desirable to maintain load balancing for as long as reasonably possible when congestion occurs.

How the source chooses to avoid reusing the path is implementation specific. For example, two possible approaches are:

1. *Entropy* from ACKs received that do not indicate congestion via the **pds.flags.m** flag can be re-used for outgoing packets, as in REPS [12]. This self-clocking mechanism essentially stores the *Entropy* values "on the wire." If a packet is lost, the packet is not ACKed, so the *Entropy* is not reused; similarly, if the packet is ACKed but the **pds.flags.m** flag indicates congestion, then the EV is not reused. REPS can be extended with a small local circular buffer (significantly smaller than the supported EV space, e.g., eight values) to cache good entropies. These entropy values are then consumed when sending out data packets. If an outgoing packet requires an EV but the buffer is empty, a new (random) EV can be used.
2. Maintain a bitmap where each bit in the bitmap corresponds to an *Entropy* value used [10]. When feedback indicates a path selected by an *Entropy* value is congested, the corresponding bit in the bitmap is set. The source rotates through the entropy space using values in a pseudo-random order. When the source is choosing an EV for a packet to send, if the corresponding bit is set, it skips that EV, clears the bit, and tries the next EV in sequence. To ensure feedback is recent, the size of the active *Entropy* value space may be dynamically changed, so that uncongested entropy values are reused within approximately two RTTs.

Combinations of these methods or other methods are also permitted. Path-aware spraying permits more active load balancing across the paths selected by the entropy values.

3.6.16.5 Detecting and Handling Path Failure

CMS is primarily aimed at networks with switches that perform ECMP load balancing based on a hash of the *Entropy* in the packet header. In such networks there is a mapping of a single path to a destination with an *Entropy* value. Path-aware multipath spraying can use this mapping, together with ECN feedback and NACKs generated due to packet trimming, to improve load balancing beyond what the switches

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

achieve by themselves. Most such networks will use a dynamic routing protocol to determine the ECMP set for a destination address. When a failure occurs in such a network, it is expected that dynamic routing will eventually remove failed paths from consideration, and so EVs that previously mapped to a failed path will then be remapped to a working path. Such dynamic rerouting may take many round-trip times, and all UET connections that spray across the failed path will suffer packet loss until rerouting has taken place.

How UET-CC load balancing handles such transient failures is an implementation decision. For example, if path-aware load balancing is being performed by UET-CC and packet trimming is enabled, then actual packet loss should be relatively rare. In such circumstances, a UET-CC source might choose to avoid reusing an EV that resulted in a timeout or SACK-based loss inference decision for some significant period of time, long enough for routing to have reconverged. However, if trimming is not used, then such inferred packet loss may be more common. An implementation would therefore need to take care removing too many EVs from the entropy set it is using.

Some networks may have a static mapping of EVs to paths with no dynamic routing protocol used to route around failures. UET-CC's path-aware multipath spraying can be used in such networks to actively route around failed paths by not reusing EVs that result in timeouts or SACK-based loss inference. In such networks, such active path choice takes on greater importance, because routing will not remove the paths from use. How to do this is again implementation dependent.

When a UET-CC endpoint does not perform path-aware multipath spraying, it is not a requirement for the network to maintain a stable mapping between EVs and paths. In such networks, switches may perform local load-balancing decisions on a per-packet basis. When this is being performed, UET-CC load balancing SHOULD NOT attempt to remove from use any EVs corresponding to packet loss, as there will not be a stable correspondence between an EV and a failed path.

3.6.16.6 Multi-port and Multi-plane Operation

UET supports multi-port NICs, and the CCC can spray traffic across multiple ports that are simultaneously used for a single PDC. There are many ways multiple ports can be used, and not all of them are supported by this specification. Among the possibilities are options numbered as follows:

1. Each port connects to a separate plane and uses the same FEP address. No inter-plane routing is provided.
2. Each port connects to a separate plane and uses a different FEP address.
3. Each port connects to a single plane and uses the same FEP address. Anycast routing is used to route packets to one of the destination ports with the same FEP address.
4. Each port connects to a single plane and uses a different FEP address.

Options 1 and 3 are supported by this specification.

In this specification, the CCC does not maintain addressing information, so multiple ports with different addresses are not directly supported.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Option 2 is not directly supported by the CCC, but once the CCC has chosen an EV and port, an implementation could map the destination FEP/port to an appropriate IP address using an additional mapping table for destination IP addresses. This specification does not cover this use case.

Load balancing for option 4 requires the CCC to choose an EV/port/FEP address combination, and load-balance between the different possible source ports and destination ports between the same pair of FEPs. This specification does not cover this use case.

3.6.16.6.1 Multiple Separate Planes, One Shared FEP Address for All Ports.

This use case is option 1 listed above. UET RUD PDCs can be created between a pair of multi-port FEPs configured in a multi-plane topology where each plane is logically or physically separate, such that a packet sent from FEP A using its port on plane P will be received by the port at FEP B that is also on plane P. In such cases, a single IP address is used by all the ports of each FEP. In such configurations, a CCC can load-balance traffic between planes and within planes by spraying traffic across a range of entropy values, where each EV is mapped to a specific outgoing port, and hence to a plane. Each EV for a PDC is used on only one plane. UET ACK and NACK packets specify the EV from the respective data packet, so it is unambiguous from the EV which path the feedback refers to.

Multi-port operation is supported via the *GetSendParams()* interface between the PDS scheduler and the CCC.

```
GetSendParams(free_port_list) -> port, Entropy, Credit_Target
```

When *GetSendParams()* is called, the scheduler supplies a list of the ports that currently have spare capacity to send a packet. There MUST be at least one free port in this list.

The CCC will choose an EV that is associated with one of the free ports in the *free_port_list*, performing load balancing using ECN feedback to avoid congested paths, as in single-port operation.

3.6.16.6.2 Multi-port, Single Plane, One IP address for All Ports

This use case is option 3 above. In this case multi-port NICs are connected to a single plane, but all ports of the same FEP share an IP address. Anycast routing can then ensure that all the ports can be part of the ECMP set used to route packets. In such cases the CCC will choose the outgoing port (if more than one free port is available), but the incoming port is determined by the switches hashing the EV from the packets. From the point of view of the CCC, option 2 behaves in the same way as option 1, in that the CCC MUST consistently map a chosen EV to a single outgoing port so that the EV in ACKs and NACKs refer to an unambiguous path.

3.6.16.6.3 Single-port Spraying on a Multi-port NIC

An implementation MAY map a FEP to a single port on a multi-port NIC. For this specification, options 2 and 4 SHOULD be handled in this manner. In such cases, each port is visible to the application as a separate FEP, and it is then the application's responsibility to load-balance traffic between FEPs and hence between ports. In this case the PDS and hence CCC are unaware that other ports exist; the CCC

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

SHOULD map its entire entropy space to the port managed by its PDS. The *free_port_list* in *GetSendParams()* can then indicate only a single port, so the choice of port by the CCC is a no-op.

3.6.17 Switch Configuration for UET CC

UET-CC uses ECN both for load balancing and as the congestion signal driving NSCC. UET-CC also optionally uses trimming for oblivious load balancing and to provide early loss indication. Thus, the correct configuration of network switches is important for UET-CC performance.

Plane_BDP is defined as the bandwidth-delay product of the lowest bandwidth path between a source and destination, measured without queuing, where the network path traverses the core of the network. In most networks this will be:

```
Plane_BDP = min(sender.linkspeed, receiver.linkspeed) * config_base_rtt
```

For a multi-ported NIC on a multi-rail topology, the *linkspeed* is typically the *linkspeed* of a single port, not the aggregate bandwidth of all ports.

Informative Comment

Plane_BDP may differ from the BDP value used in CCC algorithms over a multi-ported NIC in a multi-rail topology. In this scenario, the CCC BDP value is based on the sprayed capacity across all ports in the spraying set, but *Plane_BDP* considers only a single port or rail. For a single-port NIC, *Plane_BDP* and BDP are the same.

UET can be used with two or three traffic classes (3.6.4.7) for each UET PDC. In the three-traffic class case, these are:

- Low priority (TC_low)
  Queue setting for this TC is referred to as queue_low.
- Medium priority (TC_med)
  Queue setting for this TC is referred to as queue_med.
- High priority (TC_high)
  Queue setting for this TC is referred to as queue_high.
- When two traffic classes are used, the medium priority TC (TC_med) will not be used.

The RECOMMENDED default settings for ECN when using probabilistic marking are:

```
queue_low.min_thresh = 0.2 * Plane_BDP
queue_low.max_thresh = 0.8 * Plane_BDP
```

ECN can also be set using a deterministic marking threshold. The recommended deterministic ECN marking threshold is 0.5 * *Plane_BDP*. In general, probabilistic marking is preferred.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

ECN SHOULD be disabled for *queue_med* and *queue_high*.

When trimming is enabled, the RECOMMENDED default settings are:

```
queue_low.trim_threshold = Plane_BDP
queue_med.drop_threshold = Plane_BDP
queue_high.drop_threshold = Plane_BDP
```

If the DSCP_TRIMMABLE_RTX codepoint is used to protect retransmitted packets, the *queue_low.trim_threshold* for these packets is RECOMMENDED to be 1.5 * *Plane_BDP*.

Informative Text:
These queue threshold settings are based on simulated results to date. Lower drop thresholds may lead to additional loss of high-priority traffic. Additional tuning of deployed implementations may be required.

In the absence of trimming, the UET congestion management algorithm supports widely deployed shared switch buffer architectures. Although the algorithm is capable of handling highly variable latencies caused by the shared buffer, it is recommended to bound the tail drop threshold. The RECOMMENDED drop threshold for a given queue should be at least 2 * *Plane_BDP* and at most 5 * *Plane_BDP*.

Rather than using strict priority, a deployment MAY choose to use the same priority for *queue_med* and *queue_low*, but instead use weighted round robin or a similar scheduling scheme to ensure that when the trim rate is high, *queue_med* gets a large share of the bandwidth. In such cases a reasonable default is for *queue_med* to be weighted to receive 75% of the capacity when competing with *queue_low*.

3.6.17.1 CSIG Configuration

The overall bucket boundaries for the *max(Delay)* signal quantization SHOULD be configured based on the trim or drop threshold recommendations, regardless of whether the compact or wide CSIG header format is used.

The recommended *queue_low.trim_threshold, queue_med.drop_threshold*, and *queue_high.drop_threshold* are specified as factors of *Plane_BDP*, which is derived from the link speed and *config_base_rtt*. This information is used to derive expected maximum packet queueing times per trim or drop threshold, which equates *Plane_BDP* to *config_base_rtt* as follows:

```
Plane_BDP = min(sender.linkspeed, receiver.linkspeed) * config_base_rtt
```

However, this equation assumes that only a single queue is active per port. To account for scenarios where other, higher-priority queues are active and may increase maximum queueing times, it is RECOMMENDED to apply a 2-times multiplier to the threshold-derived maximum packet queueing time.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Based on the above methodology, the recommended trim and drop thresholds with their derived, RECOMMENDED bottom-of-the-bottom-bucket to top-of-the-top-bucket boundaries are:

- With trimming:
    - *queue_low.trim_threshold*: 1.5 * *Plane_BDP*
    - Bucketization range: [0, 3 × *config_base_rtt*]
- Without trimming:
    - *queue_med.drop_threshold, queue_high.drop_threshold*: At least 2 * *Plane_BDP* and at most 5 * *Plane_BDP*
    - Bucketization range: at least [0, 4 × *config_base_rtt]* at most [0, 10 *× config_base_rtt*]

Intermediate bucket boundaries SHOULD be distributed linearly between the minimum and maximum values.

When CSIG is enabled for multiple traffic classes, it applies the settings uniformly across all of them. The maximum trim or drop thresholds among all affected traffic classes within the port SHOULD be used as the reference values for bucket boundary calculation.

3.6.18 Programmable Congestion Management

Ultra Ethernet programmable congestion management (PCM) provides support for alternative congestion control (CC) and load balancing (LB) algorithms within the transport architecture. PCM is designed to support algorithms that reduce congestion and efficiently utilize network resources across multiple paths.

CC support in UE PCM is categorized into two primary mechanisms. The first mechanism governs the sending of data within the network by restricting the number of bytes in flight, through either a send window or credit scheme. The second optional mechanism limits the send rate. While hybrid approaches are possible, the baseline congestion control specified herein limits the number of bytes in the network only by using windows for NSCC and credits for RCCC.

LB support is provided through two approaches: per-packet load balancing, where algorithms are invoked for every transmitted and acknowledged request packet, and EV table-based load balancing, which relies on a hardware-maintained table of entropy values (EV table). PCM algorithms manipulate this table to optimize traffic distribution dynamically.

A robust congestion management framework must be capable of operating across endpoints handling a diverse range of traffic types. Large AI systems frequently encounter mixed workloads consisting of both AI and storage traffic. These mixed workloads necessitate an approach that provides fundamental quality-of-service (QoS) mechanisms not only within the network at each switch buffer, but also at the endpoints to ensure appropriate traffic differentiation and prioritization.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Furthermore, distance-aware congestion management is a critical requirement in Ultra Ethernet (UE) to accommodate the growing diversity of traffic patterns across modern network deployments. Storage traffic is frequently mirrored across multiple datacenters, often spanning distances of thousands of kilometers. The replication process requires congestion management techniques that account for latency and bandwidth constraints associated with mixed short- and long-distance transmission.

Similarly, AI and high-performance computing (HPC) systems increasingly operate across distributed datacenter infrastructures [55]. These architectures generate network flows that traverse both short intra-datacenter paths and long inter-datacenter links [56]. To optimize performance and minimize congestion, endpoints are required to differentiate between short-distance and long-distance flows. This distinction enables adaptive congestion control mechanisms that adjust transmission behavior according to the network distance, ensuring efficient data transfer across varying deployment scenarios. By integrating distance-aware congestion management, UE can address the specific needs of storage replication and AI/HPC workloads while maintaining network stability and performance.

While baseline algorithms such as NSCC and RCCC provide strong defaults, they are fixed. Evolving site conditions create a need for forward-compatible mechanisms. Operators may need a way to adopt new methods, swap in alternative algorithms, or apply targeted modifications for the new conditions.

PCM addresses these gaps by providing a programmable way to extend and adapt congestion policies in the network. PCM provides configurability to network administrators. This approach enables fine-grained control over congestion management policies, allowing adjustments to optimize network performance in response to specific deployment requirements and evolving traffic conditions.

PCM offers functionality similar to the congestion control mechanisms implemented in the QUIC protocol [54]. However, unlike QUIC, where congestion control operations are managed in software at the transport layer, PCM enables designers to offload these functions entirely to the network hardware at the endpoint. This hardware-based approach ensures low-latency congestion management, reduces computational overhead on hosts, and enables highly efficient and scalable control mechanisms that operate seamlessly within high-performance networking environments.

3.6.18.1 PCM Architecture

The UE PCM architecture is designed as a logically structured framework composed of four distinct components:

1. Incoming signals: Signals that gather information about the current state of the network as well as relevant operational state from the endpoint, form the first component. These signals provide data necessary for congestion analysis and management.
2. Congestion control (CC) algorithm: The algorithm, including its state, that processes the collected signals to compute appropriate control actions. By interpreting network conditions and endpoint states, the algorithm determines precise adjustments required to optimize data flow and mitigate congestion. The CC algorithm defines congestion control.
3. Load balancing (LB) algorithm: The algorithm that assigns entropy values to transmitted request packets and reacts to network signals when responses for transmitted requests are received.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

4. Control table: A structured table that stores computed adjustments by control mechanisms associated with each congestion control context (CCC). This table dictates transmission scheduling behavior based on dynamically updated control variables, ensuring efficient traffic regulation.

Both CC and LB algorithms use local and global state to store and exchange variables between invocations.

This logical architecture is designed to be independent of any specific physical implementation, allowing for compatibility with NSCC and RCCC congestion control schemes, as well as optional load balancing mechanisms. The following sections provide a detailed definition of each architectural component and its role in congestion management. Figure 3-3 shows an overview of the architecture.

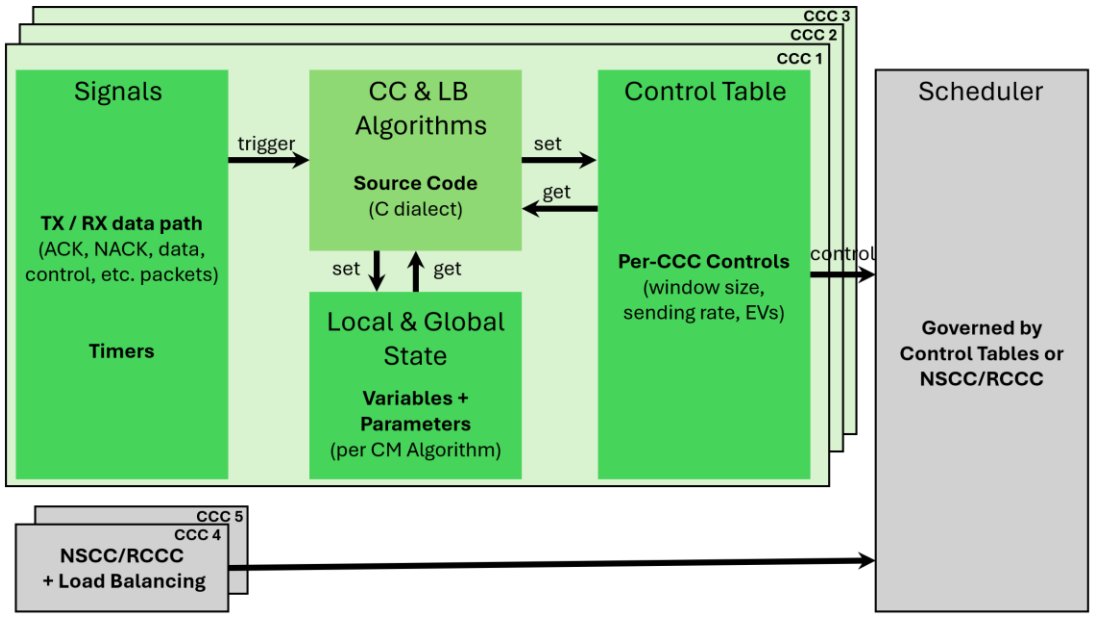


Figure 3-135 - PCM Logical Architecture Overview

FEPs that implement PCM support one or more active congestion control algorithms bound to different CCCs. Algorithms operate independently of other active algorithms and can coordinate through global variables. The PCM runtime is responsible for delivering signals to, and applying algorithm-initiated operations to, hardware.

A possible algorithm lifecycle can be as follows:

- Load: Algorithms are loaded by an implementation-specific PCM tool to a specific FEP.
- Instantiate: The PCM tool creates an instance of the algorithm.
- Configure: Parameters are setup for an algorithm instance.
- Assign: The algorithm instance is assigned to a subset of network traffic.
- Activate: The algorithm instance is activated and controls network traffic.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

- Stop/Deactivate: The instance is quiesced and no longer controls data traffic.
- Destroy: The instance is torn down and all per-instance resources and state are released.
- Unload: When all instances of the algorithm have been stopped, the algorithm may be unloaded.

These steps illustrate a typical algorithm lifecycle. It is not mandated that these steps are explicitly followed by any particular implementation. Tools may merge or skip steps (e.g., stop and unload).

This specification defines how to use floating-point datatypes. Implementations MUST offer integer datatypes and MAY offer floating-point types. If floating-point types are not supported, the compilation of a program using floating point MUST fail. Integer datatypes MAY be used to approximate floating point.

3.6.18.2 Congestion Management Algorithm Specification and Invocation

Congestion management algorithms are a fundamental component of the UE PCM framework, responsible for dynamically regulating congestion through interactions with system signals and CC controls. They can be triggered by defined signals or timers, potentially allowing invocation for each incoming acknowledgment (ACK) packet to ensure prompt responsiveness to network conditions. These algorithms can both read and modify signals and controls, enabling adaptive congestion management strategies.

At the time of algorithm invocation, the system captures a snapshot of all relevant signal values. These values remain fixed for the duration of the algorithm's execution. Signals arriving after the invocation timestamp do not affect the current execution and are queued for processing during the next invocation cycle.

Informative Text:

PCM programs may assume that the algorithm is invoked immediately after the timed or control trigger occurs. Delays could lead to inconsistent signal observations and reduced algorithm fidelity. The exact details depend on the implementation.

3.6.18.3 Load Balancing Algorithm Specification and Invocation

Like congestion management algorithms, load balancing algorithms are invoked by data path transmit, ACK/NACK, and retransmit timeout events. Implementations may use triggers to modulate the frequency of these events on a per-CCC basis. LB algorithms are logically distinct from CC algorithms and are expressed as independent programs with their own entry point, accumulators, and triggers.

PCM supports two distinct LB implementation models: per-packet and EV table.

3.6.18.3.1 Per-Packet

In the per-packet model, LB algorithms are invoked for every transmitted and received packet, and the data path does not maintain any EVs or EV state. Algorithms can associate per-packet private state with

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

transmitted packets; this state is available to the algorithm when the corresponding ACK or NACK for the packet is received.

3.6.18.3.2 EV Table

In the EV table model, the HW maintains a logical table of EVs for every active CCC. For each transmit, the hardware LB algorithm selects an EV, and for every received ACK or NACK packet the HW may invoke the PCM algorithm, depending on runtime configuration.

In this model, each HW LB algorithm is described by a YAML schema under an *lb_algorithm* stanza. This scheme is used to:

- Specify and validate the per-CCC and per-EV configuration.
- Generate LB API bindings to enable the algorithm to manipulate per-EV and per-CCC state.

Using this mechanism, implementations can achieve hardware performance while still maintaining software control.

The detailed YAML schema and fields for EV table LB algorithms are described in Section 3.6.18.8.5. The key concepts and primitives are described in Table 3-105:

Table 3-105 - YAML Schema Concepts and Primitives

Primitive	Description
hw_algo	Names the hardware LB implementation for the instance.
hw_config	Algorithm-specific, static configuration consumed by the HW algorithm.
ev_fields	Per-EV fields; defines EV values and state accessible or modifiable by SW. Multiple fields may be defined for every EV; each field is defined by a unique name, type, and whether it can be modified or accessed by SW.
ccc_fields	Per-CCC fields; used for CCC wide parameters. Like ev_fields, multiple fields may be defined for every CCC.

In the EV table model:

- Each CCC has an EV table of fixed depth, *E*, addressed by an index into a per-CCC logical table of range [0, *E*].
- For each transmit:
  - The hardware LB algorithm selects an EV in the EV table.
    - The selection process is algorithm-specific.
  - The hardware LB algorithm derives the on-wire entropy from the EV (as stored in the EV table), the values of exposed per-CCC and per-EV fields, and private state and logic.
- The LB algorithm:
  - The algorithm uses the PCM LB API to read and update (where allowed) *ev_fields* and *ccc_fields*.
  - The algorithm interacts with the hardware LB algorithm via the same trigger and signal mechanisms as for congestion management.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Informative Text:
The path-aware multipath spraying algorithm (3.6.16.4) is an example of a sophisticated scheme that can be supported in the EV table model. It maintains CCC-level path configuration and per-EV penalties and exposes the hardware-computed on-wire EV (including both path index bits and higher-order randomness) to PCM.

The YAML syntax for the algorithm is outlined below:

```
lb_algorithm:
  op_model: ev-table
  lb_entry: uet_lb_uet_bmp
  hw_algo: uet-bmp

  hw_config:
    path_random_reinit:
      type: uint
      comment: "interval in microseconds between recomputation of
path_random (upper EV bits)"

  ccc_fields:
    - name: no_of_paths
      type: uint
      sw_access: ro
      init: hw_init

    - name: ev_max_penalty
      type: uint
      sw_access: ro
      range: [0, 15]
      init: cfg_provided

    - name: ev_skip_count
      type: uint
      sw_access: ro
      range: [0, 0xFFFF]
      init: zero

  ev_fields:
    - name: entropy
      type: uint
      role: ev_value
      sw_access: ro
      range: [0, 0xFFFF]
      init: hw_init

    - name: ev_skip_penalty
      type: uint
      sw_access: rw
      range: [0, 0xFF]
      init: zero
```


Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

3.6.18.4 Congestion Signals

Congestion signals are key metrics gathered either from the network or directly at the endpoint to inform congestion management decisions. These signals provide critical information regarding current network conditions and performance. For instance, incoming acknowledgment (ACK) packets contain service time information and an indication that the associated request was explicit congestion notification (ECN) marked. Additionally, endpoint time measurements can be utilized to measure round-trip time (RTT) for each packet, offering insight into latency variations and network responsiveness.

Updates to such congestion signals often occur at a higher frequency than the invocation of the CC or LB algorithm itself. As a result, signals may undergo multiple updates before being processed, with accumulated values maintained according to predefined rules such as summation or maximum aggregation. This ensures that the CC and LB algorithms operate on an appropriately synthesized set of data, allowing for accurate congestion control adjustments while maintaining computational efficiency.

Ultra Ethernet categorizes congestion signals into three distinct types: mandatory signals, optional signals, and vendor-specific signals. Mandatory signals form the core set that all PCM implementations MUST support to ensure baseline compatibility and functionality across deployments. Optional signals provide additional capabilities that MAY be implemented selectively based on specific requirements, allowing flexibility in adapting congestion management behavior to different use cases. Vendor-specific signals MAY be used to enable further extensibility, permitting individual vendors to introduce customized metrics tailored to proprietary networking solutions. While the mandatory signals ensure interoperability across all UE implementations, optional and vendor-specific signals offer adaptability to specialized workloads and infrastructure configurations. Any UE PCM implementation MUST support the default signal set, MAY include any subset of the optional signals, and vendors MAY extend the available signal set with their own enhancements.

Signals are configured through a YAML definition specified at algorithm load time as outlined in Section 3.6.18.8.5.

3.6.18.4.1 Signal Accumulation Operators

PCM defines a set of accumulation operations for processing congestion signals. These operations determine how multiple signal updates are combined over time. The specific accumulation functions used in UE are formally specified in Table 3-106, providing a structured approach to handling signal updates within the congestion management framework. Any implementation MUST support all defined accumulation operations. Table 3-110 lists mandatory and default accumulators for all signals.

Table 3-106 - Mandatory Signal and Global Variable Accumulation Operators

Accumulation Operation	Description
uet_op_sum	Running sum
uet_op_max	Running maximum
uet_op_min	Running minimum
uet_op_last	Simply store last update (overwrite)

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

In addition, the optional accumulation operators are defined in Table 3-107.

Table 3-107 - Optional Signal Accumulation Operators

Accumulation Operation	Description
uet_op_ewma	Exponentially weighted moving average for newly samples x defined as `ewma = x * s + ewma * (1-s)` with s being a configurable smoothing factor This accumulation operator conceptually requires floating point support, and all variable types are float. Implementations MAY use integer or fixed-point to approximate the floating-point computations.

When a new update to a congestion signal is received, the update is integrated into the current signal value using the predefined accumulation operation. The update occurs atomically, meaning that each signal update is applied without interruption, preserving data integrity and preventing partial modifications. Additionally, the update order is undefined, as PCM assumes associative operators, allowing updates to be applied irrespective of sequencing constraints. This may affect the accuracy of some operators that are not actually associative such as *uet_op_ewma().*

Furthermore, all signal updates originating from a specific source, such as an acknowledgment (ACK) packet, are processed atomically. This guarantees that multiple updates contained within a single packet are applied simultaneously before any subsequent packet influences the corresponding signals. For instance, if a packet carries updates to both an explicit congestion notification (ECN) signal and a round-trip time (RTT) signal, both updates are applied at the same instant, ensuring consistency before any other packet modifies either of those signals. This atomic handling mechanism maintains consistency in congestion signal processing within UE's congestion management framework.

Multiple accumulation operations (e.g. *uet_op_last()* and *uet_op_sum()*) may be defined for the same signal.

3.6.18.4.2 Accessing and Managing Signals

Signals are accessed and configured through the predefined functions listed in Table 3-108. Note that the *signal_index*, which selects a specific signal, is configured in the YAML definition specified in Section 3.6.18.8.5 and used as a macro in the program.

The function *uet_signal_get()* MUST be supported while the functions *uet_signal_set()* and *uet_signal_update()* MAY be supported by an implementation.

Table 3-108 - Signal Access and Management Functions

Function Name	Description
uet_int_t uet_signal_get(unsigned int signal_index)	Read a signal value

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Function Name	Description
int uet_signal_set(unsigned int signal_index, uet_int_t value);	Set a signal to an absolute value, optional
int uet_signal_update(unsigned intsignal_index, uet_int_t update);	Apply the signal's configured accumulation operator, optional

Informative Text:
Signal Mnemonics: Even though the *signal_index* fields are all integers for simple compiler optimization based on constants, it is recommended to use mnemonic representations such as macros to refer to signals in the source code. The function interface allows compiler developers to easily replace any function invocation, e.g., with a simple load or store to a special address. Those functions may be defined in the header file as inline functions to permit implementation-specific optimization. Replacing such function invocations is significantly simpler than replacing accesses to structures, for example.

In this and the following functions, *signal_index* can be accessed by a mnemonic macro defined in the YAML configuration (see Section 3.6.18.8.5).

Signal Wraparound: Algorithms SHOULD be robust to signal value wraparound. In situations where wraparound is not acceptable, algorithms may be supported by using counters wide enough to guarantee wraparound will not occur for the lifetime of any given connection in the use case.

3.6.18.4.3 Mandatory Signals

Table 3-109 defines the set of mandatory signals that MUST be supported. Signals are triggered by transmitted (tx) or received (rx) packets or timer events (time). Packets fall in the following three categories: request (req) packets send data (from initiator to target for write and send, or from target to initiator for read), acknowledgment (ack/nack) packets carry information about previous request packets, and control (ctl) or credit packets are used for transport-specific controls (e.g., check the status of a packet at the receiver or the state of a path).

The naming convention follows the format *uet_<tx/rx/time>_<packet-type>_<name>* for all packet-triggered signals and *uet_<name>* for all others. Note that the bit-width of all signals can be selected in the YAML configuration.

Table 3-109 - Mandatory Signals

Signal Name	Function	Description	Datatype	Unit
uet_rx_ack	CC, LB	ACK packet received	uint	1
uet_rx_ev	LB	Entropy value (EV) of received packet	uint	1
uet_rx_ack_bytes	CC	Number of bytes acknowledged in last ACK (at sender)	uint	256 B

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Signal Name	Function	Description	Datatype	Unit
uet_rx_ack_count	CC	Counts the number of requests acknowledged (this considers cumulative acknowledgments)	uint	1
uet_rx_ack_ecn	CC, LB	ECN flags set in incoming ACK packets	uint	1
uet_rx_ack_probe	CC, LB	Probe packet ACK received	uint	1
uet_rx_ack_rtt	CC, LB	Round-trip time of incoming ACK packet wrt with respect to the request or control packet triggering them	uint	128 ns
uet_rx_ack_svc	CC	Remote service time (at target)	uint	128 ns
uet_rx_credit	CC	Credit packet received (query value with uet_signal_get_credit_value())	uint	1
uet_rx_credit_req	CC	Credit request packet received (could be either in control packet or extension header)	uint	1
uet_rx_credit_req_value	CC	Value received in last credit request packet	uint	1
uet_rx_ctl	CC	Control packet received	uint	1
uet_rx_nack	CC	NACK packet received (may be caused by trimming, loss detection, or resource exhaustion at target)	uint	1
uet_rx_nack_ecn	CC, LB	ECN flags set in incoming NACK packets	uint	1
uet_rx_nack_last_hop_trim	CC, LB	Trim NACK packets indicating trim at the last hop to the destination received	uint	1
uet_rx_nack_rtt	CC, LB	Round-trip time of incoming NACK packet wrt with respect to the request or control packet triggering them	uint	128 ns
uet_rx_nack_trim	CC, LB	Trim NACK packet received	uint	1
uet_rx_probe	CC, LB	Probe packet received	uint	1
uet_rx_probe_id	CC, LB	ID of last probe packet received	uint	1
uet_rx_req	CC	Request packet received	uint	1
uet_rx_req_ecn	CC	Request packed marked with ECN received	uint	1
uet_rx_req_trim	CC, LB	Trimmed request packet received	uint	1

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Signal Name	Function	Description	Datatype	Unit
uet_rx_req_last_hop_trim	CC, LB	Trimmed request packet indicating trim at the last hop to the destination received	uint	1
uet_time_current	CC, LB	Current timestamp	uint	128 ns
uet_time_last_update	CC, LB	Timestamp of last signal update of any configured signal	uint	128 ns
uet_tx_backlog_bytes	CC	Backlog size at sender (how much data to be sent)	uint	1 B
uet_tx_req	CC	Request packet sent	uint	1
uet_rx_req_bytes	CC	Number of bytes in a received request packet	uint	1 B
uet_tx_req_bytes	CC	Number of bytes in a sent request packet	uint	1 B
uet_tx_rto	CC, LB	Re-transmit timeout	uint	1
uet_tx_cookie	LB	Used exclusively by per-packet LB architectures to associate up to 32 bits of algorithm-private state with this transmitted.	uint	1
uet_rx_cookie	LB	Used exclusively by per-packet LB architectures to retrieve the algorithm-private cookie associated with a corresponding transmitted packet when its ACK or NACK is received.	uint	1
uet_tx_ev	LB	Entropy value (EV) of transmitted packet	uint	1
uet_tx_inflight_bytes	CC	Inflight byte count	uint	1
uet_tx_cwnd	CC	Current congestion window	uint	1
uet_tx_send_rate	CC	Sending rate in bytes/s	uint	128 MiB/s
uet_tx_ack_per_packet	CC	If true, ACK every packet (local at RX engine)	bool	bool
uet_tx_remote_ack_per_packet	CC	If true, request an ACK per packet from target	bool	bool
uet_rx_ack_gen_bytes	CC	Byte count to trigger ACK packet after (local at RX)	uint	1 B
uet_rx_ack_gen_timer	CC	Timer after which ACKs trigger (local at RX)	uint	128 ns
uet_rx_cwnd_penalty	CC	Receiver congestion window penalty of UE's default congestion control	uint	1
uet_tx_fep_credit_rate	CC	Set the FEP's credit rate	uint	0.01 (1% increments)

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Signal Name	Function	Description	Datatype	Unit
uet_rx_telegram_local	LB, CC	Local telegram received	uint	1
uet_rx_telegram_peer	LB, CC	Remote telegram received	unit	1
uet_rx_telegram_type	LB, CC	Telegram type	uint	1
uet_tx_ev_tbl_depth	LB	Depth of EV table	uint	1
uet_tx_ev_tbl_idx	LB	EV table Index	uint	1

3.6.18.4.4 Signal-Accumulation Function Combinations

Not all combinations of signals and accumulation functions are useful. This specification defines a mandatory subset to ensure implementations can specialize for a minimal, interoperable baseline. Other combinations may be provided. An implementation MAY implement accumulation functions for any signal that it finds useful.

3.6.18.4.4.1 CC Signal-Accumulation Functions

Table 3-110 - CC Signal Accumulation Functions

Signal/Accumulation	uet_op_sum	uet_op_max	uet_op_min	uet_op_last
uet_rx_ack	MUST	N/A	N/A	MAY
uet_rx_ack_bytes	MUST	N/A	N/A	MUST
uet_rx_ack_count	MUST	N/A	N/A	MAY
uet_rx_ack_ecn	MAY	N/A	N/A	MUST
uet_rx_ack_gen_bytes	N/A	N/A	N/A	MAY
uet_rx_ack_gen_timer	N/A	N/A	N/A	MAY
uet_rx_ack_probe	MAY	N/A	N/A	MAY
uet_rx_ack_rtt	MUST	MUST	MUST	MUST
uet_rx_ack_svc	MUST	MAY	MAY	MUST
uet_rx_cookie	N/A	N/A	N/A	MUST
uet_rx_credit	MUST	N/A	N/A	MAY
uet_rx_credit_req	MAY	N/A	N/A	MAY
uet_rx_credit_req_value	MAY	MAY	MAY	MUST
uet_rx_ctl	MAY	N/A	N/A	MAY
uet_rx_cwnd_penalty	N/A	N/A	N/A	MAY

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Signal/Accumulation	uet_op_sum	uet_op_max	uet_op_min	uet_op_last
uet_rx_ev	N/A	N/A	N/A	MAY
uet_rx_nack	MUST	N/A	N/A	MAY
uet_rx_nack_ecn	MAY	N/A	N/A	MUST
uet_rx_nack_last_hop_trim	MAY	N/A	N/A	MUST
uet_rx_nack_rtt	MUST	MUST	MUST	MUST
uet_rx_nack_trim	MUST	N/A	N/A	MAY
uet_rx_probe	MAY	N/A	N/A	MAY
uet_rx_probe_id	N/A	N/A	N/A	MUST
uet_rx_req	MUST	N/A	N/A	MAY
uet_rx_req_bytes	MUST	N/A	N/A	MUST
uet_rx_req_ecn	MAY	N/A	N/A	MAY
uet_rx_req_last_hop_trim	MAY	N/A	N/A	MAY
uet_rx_req_trim	MAY	N/A	N/A	MAY
uet_rx_telegram_local	N/A	N/A	N/A	MUST
uet_rx_telegram_peer	N/A	N/A	N/A	MAY
uet_rx_telegram_type	N/A	N/A	N/A	MUST
uet_time_current	N/A	N/A	N/A	MUST
uet_time_last_update	N/A	N/A	N/A	MUST
uet_tx_ack_per_packet	N/A	N/A	N/A	MAY
uet_tx_backlog_bytes	N/A	N/A	N/A	MUST
uet_tx_cookie	N/A	N/A	N/A	MUST
uet_tx_cwnd	N/A	N/A	N/A	MUST
uet_tx_fep_credit_rate	N/A	N/A	N/A	MAY
uet_tx_inflight_bytes	N/A	N/A	N/A	MUST
uet_tx_remote_ack_per_packet	N/A	N/A	N/A	MAY
uet_tx_req	MAY	N/A	N/A	MAY
uet_tx_req_bytes	MUST	N/A	N/A	MUST
uet_tx_rto	MAY	N/A	N/A	MUST

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Signal/Accumulation	uet_op_sum	uet_op_max	uet_op_min	uet_op_last
uet_tx_send_rate	N/A	N/A	N/A	MAY

3.6.18.4.4.2 LB Signal-Accumulation Functions

Table 3-111 - LB Signal-Accumulation Functions (ET = Event Table, PP = Per-Packet)

Signal/Accumulation	Operation Model	uet_op_sum	uet_op_max	uet_op_min	uet_op_last
uet_rx_ack	ET, PP	MAY	N/A	N/A	MUST
uet_rx_ack_ecn	ET, PP	MAY	N/A	N/A	MUST
uet_rx_ack_probe	ET, PP	MAY	N/A	N/A	MAY
uet_rx_ack_rtt	ET, PP	MUST	MUST	MUST	MUST
uet_rx_cookie	PP	N/A	N/A	N/A	MUST
uet_rx_ev	ET, PP	N/A	N/A	N/A	MUST
uet_rx_nack_ecn	ET, PP	MAY	N/A	N/A	MUST
uet_rx_nack_last_hop_trim	ET, PP	MAY	N/A	N/A	MUST
uet_rx_req_trim	ET, PP	MAY	N/A	N/A	MAY
uet_rx_req_last_hop_trim	ET, PP	MAY	N/A	N/A	MAY
uet_rx_nack_rtt	ET, PP	MUST	MUST	MUST	MUST
uet_rx_nack_trim	ET, PP	MAY	N/A	N/A	MUST
uet_rx_probe	ET, PP	MAY	N/A	N/A	MAY
uet_tx_probe_id	ET, PP	N/A	N/A	N/A	MUST
uet_rx_telegram_local	ET, PP	N/A	N/A	N/A	MUST
uet_rx_telegram_peer	ET, PP	N/A	N/A	N/A	MAY
uet_rx_telegram_type	ET, PP	N/A	N/A	N/A	MUST
uet_time_current	ET, PP	N/A	N/A	N/A	MUST
uet_time_last_update	ET, PP	N/A	N/A	N/A	MUST
uet_tx_cookie	PP	N/A	N/A	N/A	MUST
uet_tx_ev	PP	N/A	N/A	N/A	MUST
uet_tx_ev_tbl_depth	ET	N/A	N/A	N/A	MUST

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Signal/Accumulation	Operation Model	uet_op_sum	uet_op_max	uet_op_min	uet_op_last
uet_tx_ev_tbl_idx	ET	N/A	N/A	N/A	MUST
uet_tx_rto	ET, PP	MAY	N/A	N/A	MUST

3.6.18.4.5 Optional Signals

Table 3-112 defines the set of optional signals that MAY be supported. If a YAML definition uses a signal that is not supported by the specific implementation, the compiler MUST fail.

3.6.18.4.5.1 CSIG Signals

Ultra Ethernet defines congestion signaling (CSIG) in Section 4.2. CSIG is an in-band telemetry method to collect information on the fabric's congestion state as the packets traverse the network. Each switch along the path applies a simple aggregation function to a locally measured congestion value and the value carried in the CSIG tag of the received packet. The result of the aggregation function at each switch is that the packet egresses with a single, path-summary result. In many useful cases, the relevant metric is the minimum or maximum of a signal across the path—i.e., the value where the signal is most constraining (e.g., the minimum available bandwidth or the maximum queueing delay).

Within PCM, CSIG signals are accumulated to provide a standardized source of congestion signals that algorithms can consume for their decisions. The CSIG specification defines multiple signal types (e.g., available bandwidth and delay), but each packet carries exactly one CSIG signal at a time, encoded as a quantized value. CSIG information can be consumed at the receiver (termination) or reflected back to the sender, enabling sender- or receiver-based PCM algorithms to react by adjusting windows, rates, or path selection.

If a switch performs packet trimming, then a CSIG-tagged packet's **csig.d** bit (do-not-modify flag) is set. This leaves the tag unchanged thereafter and preserves the congestion signal as before the trimming. This information is to be made available to the PCM algorithms when CSIG support is enabled in PCM.

The PCM interface exposes these readings as optional CSIG signals delivered to the algorithms.

CSIG always does tag termination at the receiver if it receives a CSIG packet. If NSCC is enabled, the *CSIG-Reflect()* function is mandatory, which populates the CSIG Reflection Header within ACK/NACK packets. However, an algorithm can have only one source of the data at any given time–The *CSIG-Term()* function is the natural input for receiver-based CC, whereas *CSIG-Reflect()* would provide the input to the sender-based CC.

A per-group set of signals could be defined, but invocations can be easily controlled by checking if the CSIG signal type received is terminal or reflected.

PCM exposes a boolean *uet_rx_csig_type* where 0=TERM and 1=REFLECT. Algorithms SHOULD validate the origin on each sample and then operate on the unified fields {*uet_rx_csig_t, uet_rx_csig_s, uet_rx_csig_m, uet_rx_csig_d*}.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

CSIG signal values may be 5 bits wide when extracted from the 4-byte compact CSIG tag or20 bits wide when extracted from the 8-byte wide CSIG tag. The CSIG signal returned to PCM is of a fixed width that can support both 5-bit and 20-bit values.

CSIG values exposed in PCM for algorithms SHALL carry the quantized bucket index *s* as an opaque unsigned integer together with {*t, m, d*}. PCM implementations SHALL supply de-quantization parameters via a YAML file, and SHOULD provide a utility function *csig_dequantize(t, s)* that returns the representative raw value.

Table 3-112 - Optional Signals

Signal Name	Description	Datatype	Unit
uet_rx_csig_d	Do-not-modify bit	uint	1
uet_rx_csig_m	Locator metadata	uint	1
uet_rx_csig_s	Quantized value	uint	1
uet_rx_csig_t	Received signal type	uint	1
uet_rx_csig_type	Source is CSIG-TERM = 0, or CSIG-Reflect = 1	bool	1
uet_rx_restore_cwnd	Value of ACK_CC_STATE the restore cwnd (rc) bit **pds.ack_cc_state.rc**	uint	1
uet_rx_staging_buffer_occ	Occupancy of the local receiver staging buffer (value between 0 and 100 in percent)	uint	1

3.6.18.4.6 Vendor-specific Signals

Any implementation MAY support vendor-specific signals. Their names MUST be prefixed with <*vend*>, a vendor-specific alias encoded in the variable name, e.g., *uet_tx_ack_vend_<signal_name>* or *uet_rx_req_vend_<signal_name>*. The <*signal_name*> MAY encode any additional information such as vendor name.

3.6.18.5 Triggers

Algorithms request invocation when an accumulator matches their criteria, as determined by triggers. Every trigger is associated with a single accumulator instance and is identified by a trigger identifier. The set of triggers available to an algorithm, and their association with accumulation functions and signals, is defined by the algorithm's YAML configuration profile. Triggers cannot be created or deleted at runtime, but an algorithm can modify them to change their configuration and match criteria.

Triggers are defined in the YAML specification and given a macro name to reference them from the algorithm code.

3.6.18.5.1 Trigger Definition

Triggers are state machines that determine when to take an action based on a unified count-or-time condition.

Each trigger is defined by the following configuration fields:

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

- *trigger_id*: Unique identifier for the trigger within the CCC
- *accumulator_value*: Value of the accumulator being evaluated
- *op*: Comparison operator used to evaluate a predicate against an accumulator value. Allowed operators are: ==,!=, >, >=, <, <=, *

Informative Text:
The '*' operator is a "wildcard match," used when the algorithm should always be triggered; it is the logical equivalent of the compound operator: (==||!=)

- *mode*: Trigger evaluation mode. Only one mode is supported: COUNT_OR_TIME. In COUNT_OR_TIME mode, the trigger may fire due to accumulated TRUE evaluations or due to elapsed time since the trigger last fired.
- *mode_ref_val*: Tuple specifying count (*mode_ref_count_val*) and time (*mode_ref_time_val*) thresholds for determining whether a trigger has fired. Predicates tests against this tuple of reference values are defined as:

```
time_predicate = accumulator_value (op) mode_ref_time_val
count_preficate = accumulator_value (op) mode_ref_count_val
```

  A true evaluation is any evaluation for which at least one predicate evaluates to true. Semantics of *mode_ref_val* tuple are defined as follows:

mode_ref_count_val	mode_ref_time_val	Semantic Behavior
0	0	No evaluation: No evaluation is performed; trigger fires immediately if armed
0	≠0	Time-only mode: Trigger fires when elapsed time since the trigger last fired is greater than or equal to *mode_ref_time_val*
≠0	0	Count-only mode: Trigger fires after *mode_ref_count_val* true evaluations of the predicate
≠0	≠0	Combined mode: Trigger fires after *mode_ref_count_val* true evaluations of the predicate or when elapsed time since the trigger last fired is greater than or equal to *mode_ref_time_val*, whichever occurs first

- *arm_mode*: Controls trigger evaluation and behavior after firing. Allowed values:
  - DISARMED: Trigger does not evaluate predicate and cannot fire
  - ONCE: Trigger evaluates predicate; when it fires it automatically transitions to DISARMED and MUST be explicitly re-armed to fire again
  - CONTINUOUS: Trigger remains armed after each fire; after firing, the count of true evaluations and elapsed time since last fire are reset and evaluation continues
- *action*: Action taken when trigger fires. Allowed values:
  - INVOKE: Invoke the PCM algorithm
  - RECORD: Record the trigger fired in the trigger bitmask; do not invoke algorithm
- *invoke_deadline*: Upper bound on how long an invoke action may be deferred after trigger fires. Applies only if *action* is INVOKE

3.6.18.5.2 Trigger Field Resolution

Trigger field resolution is show in Table 3-113:

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Table 3-113 - Trigger Field Resolution

Field Name	Datatype	Unit
mode_ref_count_val	uint	1
mode_ref_time_val	uint	128 ns
invoke_deadline	uint	128ns

3.6.18.5.3 Trigger Evaluation

Triggers are logically evaluated as follows:

- When a packet is received, all armed triggers associated with accumulators tracking uet_rx signals are evaluated
- When a packet is transmitted, all armed triggers associated with accumulators tracking uet_tx signals are evaluated
- When a time event occurs, all armed triggers associated with accumulators tracking uet_time signals are evaluated

The algorithm MUST be invoked exactly once to deliver all fired (non-deferred) triggers; piecemeal delivery is not allowed. Algorithms determine the set of fired triggers by examining the trigger bitmask delivered with every invocation.

Multiple deferred invocation triggers are reconciled as follows:

- A per-CCC deadline variable maintains the minimum deadline to the next invocation.
- For every evaluated trigger, if the trigger has a non-zero invocation deadline, the per-CCC deadline is updated as follows:

```
CCC.invoke_deadline = min(CCC.invoke_deadline,now() +
trigger.invoke_deadline)
```

3.6.18.5.4 Trigger Updates

On successful application of a trigger update, the PCM subsystem MUST reset all trigger working states. After resetting, the trigger MUST be immediately eligible for evaluation and firing under the new configuration. All trigger reference value fidelities match those of the associated signal.

3.6.18.5.5 Trigger Modification

The PCM API provides support for algorithms to manipulate their triggers as seen in Table 3-114:

Table 3-114 - Trigger Modification API

Function Name	Description
int uet_modify_trigger(unsigned int signal_index, const struct uet_trigger_attr *attr, uint32_t attr_mask);	Modify a trigger's configuration
int uet_query_trigger(unsigned int signal_index, struct uet_trigger_attr *attr, uint32_t attr_mask);	Query a trigger's configuration

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

The function *uet_trigger_mask()* returns an unsigned integer that can be used to identify the trigger that invoked the algorithm. It can be compared to macros defined for each defined signal in the YAML configuration corresponding to the signal.

Informative Text (example for *uet_trigger_mask()*):
Imagine an algorithm is set up with two triggers with the following YAML file:

```
- signal:
  name: "uet_time_current"
  macro: "t_time"
  accumulator: "uet_last"
     trigger:
       op:wc
       op_val: 0
      ...

- signal:
  name: "uet_rx_ack"
  macro: "c_acks"
  accumulator: "uet_sum"
     trigger:
      op: gte
      op_val: <integer>
     ...

void algorithm() {
  uet_trigmask_t fired_triggers;
  uet_trigger_mask(&fired_triggers);
  if(fired_triggers & t_time) {
    // algorithm was triggered by timer
  }
  if(fired_triggers & c_acks) {
    // algorithm was triggered by ack count received
  }
}
```

Informative Text:
Implementations SHOULD ensure that the function *uet_trigger_mask()* is implemented in a fast and lightweight manner appropriate for the platform. The specific mechanism used to achieve this efficiency is implementation-defined.

3.6.18.5.6 Detecting and Recovering from Dropped Events

As outlined in Section 3.6.18.9, PCM can drop a subset of event types. Algorithms can detect and recover from dropped events by using the function in Table 3-115.

Table 3-115 - Dropped Event Detection Functions

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Function Name	Description
int uet_trigger_drop_mask(uint32_t *trigger_mask);	Deliver all triggers for which events have been dropped

3.6.18.5.7 Load Balancing Support

3.6.18.5.7.1 Per-EV Triggers

Per-EV triggers enable LB algorithms to react to events associated with specific EVs in the CCC EV table. By evaluating conditions and updating state locally in hardware, these triggers eliminate the overhead of frequent software callbacks.

Per-EV triggers are defined using the YAML signal construct configuration schema; the *scope* field MUST be set to EV for every per-EV signal. Multiple EV triggers are supported within a single algorithm configuration. Triggers defined in this manner are automatically attached to every EV index in the CCC EV table. It is not possible to bind triggers to individual EV indices; the association is uniform across all slots in the EV table for the lifetime of the CCC.

Per-EV signals are evaluated whenever a packet associated with a given EV is transmitted or received.

On invocation, an algorithm can determine whether any per-EV trigger has fired by checking the macro name against the output of *uet_trigger_mask()*.

Algorithms can further iterate through the set of fired per-EV triggers using the functions below:

Table 3-116 - Per-EV Access and Manipulation APIs

Function Name	Description
int uet_ev_trigger_next(unsigned int signal_id)	Advances the iterator to the next EV index whose per-EV trigger fired for the specified *signal_id*.
int uet_ev_trigger_iter_reset(unsigned int signal_id)	Reset the per-EV iterator for *signal_id.*
int uet_ev_signal_id(unsigned int ev_signal_id, unsigned int ev_idx)	Returns the per-EV signal identifier for the given signal and EV index. The output of this function is used in *uet_signal_*()* and *uet_trigger_*()* calls.

3.6.18.5.7.2 Per-Packet Operation Mode

Trigger support for per-packet LB mode MUST support triggers configured as follows:

Table 3-117 - Per-Packet Operation Mode

Property	Value
op	*
opval	N/A
mode	EVENT-COUNT
mode_val	1
action	INVOKE-ALGORITHM
enable	TRUE

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Runtime trigger modification (3.6.18.5.5) is not supported.

3.6.18.6 Control Operations

The following control operations are available to algorithms:

3.6.18.6.1 CCC Controls

CCC controls include setting the window size, sending rate, or the EV to use for the CCC. Table 3-118 defines mandatory control functions. The control outputs are defined as macros in uet_pcm.h.

Table 3-118 - Mandatory CCC Control Functions

Function Name	Description
int uet_set_cwnd(unsigned int value)	Set the CCC congestion window

Table 3-119 shows optional CCC control functions that architectures MAY support. If the rate control is supported, the rate limit SHOULD apply in tandem with the *cwnd* limit.

Table 3-119 - Optional CCC Control Functions

Function Name	Description
int uet_set_send_rate(unsigned int value)	Set CCC send rate
int uet_set_ack_per_packet(unsigned int value)	Set ack-per-packet property; non-zero signifies every packet should be ACKed
int uet_set_ack_gen_bytes(unsigned int value)	Set byte threshold for ACK generation
int uet_set_ack_gen_timer(unsigned int value)	Set timer value for ACK generation
int uet_set_rx_cwnd_penalty(unsigned int value)	Set the local RX congestion window penalty value

3.6.18.6.2 State Controls

The functions shown in Table 3-120 are used to access algorithm variables. Variables can be local or global and are defined as such in the YAML configuration. Local and global variables form the algorithm's state. Local variables are guaranteed to be unchanged during algorithm invocations, while global variables can be accessed and updated by algorithms in other CCCs. Global variables are presented as a snapshot at invocation time (similar to signals) and the variables are not updated during the algorithm execution. Global variables cannot be set but can be updated using an accumulation operator defined in the YAML configuration.

Table 3-120 - Local and Global State Controls

Function Name	Description
int uet_get_variable_int(unsigned int variable_index, uet_int_t *value)	Get the integer variable at the specified index.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Function Name	Description
int uet_get_variable_uint(unsigned int variable_index, uet_uint_t *value)	Get the unsigned integer variable at the specified index.
int uet_get_variable_float(unsigned int variable_index, uet_float_t *value)	Get the floating-point variable at the specified index.
int uet_set_variable_int(unsigned int variable_index, uet_int_t value)	Set the integer local variable at the specified index.
int uet_set_variable_uint(unsigned int variable_index, uet_uint_t value)	Set the unsigned integer local variable at the specified index.
int uet_set_variable_float(unsigned int variable_index, uet_float_t value)	Set the floating-point local variable at the specified index.
int uet_update_variable_int(unsigned int variable_index, uet_int_t value)	Update the integer global variable at the specified index using the operation defined in the YAML.
int uet_update_variable_uint(unsigned int variable_index, uet_uint_t value)	Update the unsigned integer global variable at the specified index using the operation defined in the YAML.
int uet_update_variable_float(unsigned int variable_index, uet_float_t value)	Update the floating-point global variable at the specified index using the operation defined in the YAML.

Informative Text:

Indexes into the local state do not necessarily map directly to memory offsets due to the varying size of the elements. The details are implementation-dependent and not defined here.

3.6.18.6.3 PCM Telegrams

PCM algorithms can exchange out-of-band messages with software running on the local node as well as the peer CCC using optional telegrams. The structure and use of telegrams is algorithm-specific; they may be used in any situation in which coordination or information exchange between the algorithm and the host or remote peer is required.

Telegrams are fixed-size structures of up to 30 bytes in length, composed of one or more fields. An algorithm can define multiple types of telegrams. There is no definition-level distinction between telegrams exchanged locally or with peers, although support is provided for an algorithm to distinguish between the source of received telegrams. Telegrams are defined using the YAML schema provided in Section 3.6.18.8.5. Telegrams of different types are uniquely identified both locally and remotely by the value of the structure *type_id* field.

Informative Text:

The 30-byte length constraint for telegrams results from specific design decisions: telegrams are transmitted as control packets, the total payload is limited to 32 bytes, ensuring the complete

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

packet—including all protocol headers—remains small. Of this total, 2 bytes are reserved for the telegram type header and other control fields, leaving 30 bytes available for actual payload data.

The PCM runtime delivers received telegrams to algorithms through the algorithm's registered callback function. A single telegram is delivered per invocation. Algorithms determine the telegram type using the *uet_get_tgm_type()* function and the invocation context using the *uet_invoke_ctx()* function. Telegram field values are accessed using the telegram field access functions. Fields MUST be delivered to the algorithm in the PCM implementation's native endian order.

Table 3-121 - Telegram Type Controls

Function Name	Description
int uet_set_tgm_type(int tgm_type)	Set the telegram message type. This function sets the telegram message type as defined in the YAML. The input *tgm_type* is an appropriate tgm macro value.

Table 3-122 - Telegram Field Access Controls

Function Name (Argument(s))	Description
int uet_get_field_int(int field_index, int32_t *value)	Get the integer field at *field_index*. The *field_index* argument is an appropriate field macro identifier as defined in the YAML.
int uet_get_field_uint(int field_index, uint32_t *value)	Get the unsigned integer field at *field_index*. The *field_index* argument is an appropriate field macro identifier as defined in the YAML.
int uet_get_field_float(int field_index, float *value)	Get the float field at *field_index*. The *field_index* argument is an appropriate field macro identifier as defined in the YAML.
int uet_set_field_int(int field_index, int32_t value)	Set the integer field at *field_index*. The *field_index* argument is an appropriate field macro identifier as defined in the YAML.
int uet_set_field_uint(int field_index, uint32_t value)	Set the unsigned integer field at *field_index*. The *field_index* is an appropriate field macro identifier as defined in the YAML.
int uet_set_field_float(int field_index, float value)	Set the float field at *field_index*. The *field_index* is an appropriate field macro identifier as defined in the YAML.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Algorithms send telegrams locally or to remote peers as follows: The algorithm sets the telegram type using the *uet_set_tgm_type()* function, populates the field values using the appropriate *uet_set_field_*()* functions, and transmits the message locally or to the remote peer using the appropriate function. Algorithms can transmit telegrams regardless of the callback invocation context. Implementations MUST support sending local telegrams and they MAY support sending telegrams to remote peers or collector nodes.

Telegram structures are invalidated when the current invocation exits; algorithms MUST reconstruct the structure on subsequent invocations as required.

Table 3-123 - Telegram Transmit Controls

Function Name (Argument(s))	Description
int uet_send_tgm_lcl(void)	Send a telegram locally.
int uet_send_tgm_peer(void)	Send a telegram to the remote peer PDC. If the remote peer is not connected to the remote PDC at the time the operation is issued, the function returns -ENOTCONN; otherwise, it returns 0 to indicate successful enqueue for transmission. Sending a telegram to a remote peer that is not PCM controlled is treated as a local fatal error (Sec. 3.6.18.7.1).

3.6.18.6.3.1 Processing Telegrams

The PCM runtime is responsible for forwarding sent telegrams to the local host or remote peer as follows:

Local telegrams are forwarded to the host in an implementation-specific manner. The mechanism used to receive logs and alert host software is configuration-specific. Example mechanisms include:

- Kernel logs (e.g., dmesg).
- Device-specific control APIs (e.g., netlink/sysfs/debugfs).
- Vendor-specific user-space daemons.

Peer telegrams are forwarded to the remote PDC via Telegram CPs (see Section 3.5.16.8). Telegram structures are encoded as payload in these control packets with the following format:

- Fixed header (2 bytes):
  - *type_id* (8 bits): Identifies the telegram type and determines the interpretation of the payload. This field MUST be populated with the telegram type identifier as defined by the corresponding telegram YAML specification.
  - reserved (8 bits): Reserved for future use. This field MUST be set to 0x0 by the sender and MUST be ignored by the receiver.
- Fields (up to 30 bytes):

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

- Contains the telegram-specific field values associated with the indicated *type_id*.
- The encoding, ordering, and semantics of these fields are defined by the telegram type.
- If the telegram-specific payload is shorter than 30 bytes, the remaining bytes MUST be set to 0x0 by the sender and MUST be ignored by the receiver.

Regardless of whether the telegram is forwarded locally or to a peer, field values MUST be transmitted in big-endian format and packed in accordance with the YAML specification for the structure

3.6.18.6.4 Control Packets

PCM algorithms can trigger control packets such as probe packets or credit packets to the peer. Those packets are scheduled through the current CCC and thus use the associated window, rate, and EV.

Table 3-124 - Control Packet Sending

Function Name (Argument(s))	Description
int uet_send_ctrl_probe(unsigned int id, unsigned int ev)	Send probe control packet with the specified opaque ID.
int uet_send_ctrl_credit(unsigned int credit)	Send credit control packet with credit amount.

3.6.18.6.5 Load Balancing

PCM provides the LB controls for EV table implementations in Table 3-125:

Table 3-125 - Load Balancing Controls

Function Name (Argument(s))	Description
int uet_get_ev_field_int(uint32_t ev_idx, int field_index, uet_int_t *value);	Generic EV field accessor function for integer EV fields. *field_index* is YAML-defined per-EV field macro.
int uet_get_ev_field_uint(uint32_t ev_idx, int field_index, uet_uint_t *value);	Generic EV field accessor function for unsigned integer EV fields. *field_index* is YAML-defined per-EV field macro.
int uet_get_ev_field_float(uint32_t ev_idx, int field_index, uet_float_t *value);	Generic EV field accessor function for float EV fields. *field_index* is YAML-defined per-EV field macro.
int uet_set_ev_field_int(uint32_t ev_idx, int field_index, uet_int_t value);	Generic EV field setter function for integer EV fields. *field_index* is YAML-defined per-EV field macro.
int uet_set_ev_field_uint(uint32_t ev_idx, int field_index, uet_uint_t value);	Generic EV field setter function for unsigned integer EV fields. *field_index* is YAML-defined per-EV field macro
int uet_set_ev_field_float(uint32_t ev_idx, int field_index, uet_float_t value)	Generic EV field setter function for float EV fields. *field_index* is YAML-defined per-EV field macro
int uet_get_ccc_field_int(int field_index, uet_int_t *value);	Generic CCC field accessor function for integer CCC fields. *field_index* is YAML-defined per-CCC field macro
int uet_get_ccc_field_uint(int field_index, uet_uint_t *value);	Generic CCC field accessor function for unsigned integer CCC Fields. *field_index* is YAML-defined per-CCC field macro
int uet_get_ccc_field_float(int field_index, uet_float_t *value);	Generic CCC field accessor function for float CCC fields. *field_index* is YAML-defined per-CCC field macro
int uet_set_ccc_field_int(int field_index, uet_int_t value);	Generic CCC field setter function for integer CCC fields. *field_index* is YAML-defined per-CCC field macro
int uet_set_ccc_field_uint(int field_index, uet_uint_t value);	Generic CCC field setter function for unsigned integer CCC fields. *field_index* is YAML-defined per-CCC field macro

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Function Name (Argument(s))	Description
int uet_set_ccc_field_float(int field_index, uet_float_t value);	Generic CCC field setter function for float CCC fields. *field_index* is YAML-defined per-CCC field macro

3.6.18.7 Error Handling and Recovery

This section defines mandatory behavior for signaling, containing, and recovering from fatal errors originating from the data path or PCM subsystem.

The error handling and recovery model is defined to enforce the following invariants:

1. A CCC MUST NOT operate in an uncontrolled, undefined, or partially corrupt state.
2. Any condition that compromises correctness, safety, or determinism of CCC operation MUST result in immediate and permanent disablement of the CCC by transitioning it to the ERROR state.

3.6.18.7.1 Fatal error conditions and state transition

A CCC in the ACTIVE state MUST atomically transition to the ERROR state if any of the following fatal conditions occur:

- The hardware detects an internal fault while generating PCM signal updates, evaluating triggers, invoking an algorithm, or applying algorithm-initiated operations
- A PCM algorithm performs an invalid operation as defined by this specification, including (but not limited to) illegal PCM API usage or out-of-range index access
- A PCM algorithm explicitly requests fatal termination of the CCC by invoking *uet_ccc_fatal()*

Upon detection of a fatal condition, the implementation MUST make the ACTIVE→ERROR transition atomically visible to all components that can observe the CCC or PDC state.

When a CCC transitions to the ERROR state:

- All PDCs serviced by the CCC MUST simultaneously transition to an ERROR condition consistent with the CCC ERROR state.
- There MUST be no transition out of ERROR except via explicit reinitialization or reset performed outside the PCM subsystem.

3.6.18.7.1.1 Algorithm API

Algorithms choose to cope with unexpected events or conditions by performing algorithm-mediated control actions (e.g., setting the congestion window to a single packet). When an algorithm encounters an (algorithm-defined) unrecoverable error, it can transition a CCC into ERROR state by using the function defined in Table 3-126.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Table 3-126 - ERROR State Transition Function

Function Name (Argument(s))	Description
void uet_ccc_fatal(int reason)	Transition the CCC into ERROR state. The value of r*eason* is algorithm-defined and is used to signal the cause of the error for debug and telemetry purposes; it is not interpreted or used by the PCM subsystem.

3.6.18.7.2 ERROR state semantics

3.6.18.7.2.1 Data path

Once a CCC has entered the ERROR state, the implementation MUST ensure the following:

- The CCC is logically disabled.
- No new packets associated with the CCC are admitted to the data path.
- All in-flight, acknowledged, and queued packets associated with the CCC are logically dropped.
- No further PCM events or telegrams are generated for the CCC.
- No API operations invoked by the algorithm are applied.

3.6.18.7.2.2 PCM subsystem behavior

After the CCC enters the ERROR state:

- Any PCM algorithm invocation already in progress MAY be aborted or MAY complete execution; however, regardless of completion or abortion, it MUST NOT produce any externally visible side effects of any kind.
- No mutation of the CCC or PDC state is permitted after the CCC enters the ERROR state, including mutations attempted by an in-flight algorithm invocation.
- Any PCM API invocation targeting the CCC (including those issued by an in-flight algorithm invocation) MUST fail and MUST return the error code -ENOTRECOVERABLE.
- Further PCM algorithm invocations for the CCC MUST NOT occur.

3.6.18.7.3 Error reporting and observability

When a CCC transitions to the ERROR state, the implementation MUST record a single CCC-scoped fatal error indication with the following information:

- Error source. One of:
  - IMPLEMENTATION: Fatal error detected by the implementation.
  - PCM_ALGORITHM: CCC transitioned to ERROR due to a PCM algorithm action.
- Error value:
  - If IMPLEMENTATION: *Value* is implementation-defined.
  - If PCM_ALGORITHM: *Value* MUST equal the reason supplied to *uet_ccc_fatal().*

Once a fatal error indication is recorded for a CCC, subsequent fatal causes for that CCC MUST NOT overwrite, append, or accumulate additional fatal indications. The fatal error indication MUST remain available for software inspection until CCC reset or reinitialization.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

The implementation SHOULD emit a diagnostic message identifying the CCC instance and the recorded fatal error indication.

The implementation MUST provide a software-visible mechanism to determine, for each CCC, whether it is in ACTIVE or ERROR state and, if in the ERROR state, to retrieve the recorded fatal error indication. This mechanism MUST satisfy all the following:

- The ERROR state MUST be observable by software no later than the point at which the ACTIVE→ERROR transition becomes atomically visible within the implementation.
- Retrieval of CCC state and fatal error indication MUST be non-destructive (i.e., it MUST NOT clear the indication).

3.6.18.7.4 Reset and reinitialization

Recovery from the ERROR state is outside the scope of PCM. An implementation MAY provide a platform-specific mechanism (e.g., driver, firmware, or device reset) to reinitialize a CCC. If a CCC is reinitialized, the implementation MUST ensure the reinitialized CCC appears as a fresh instance with no retained PCM state.

3.6.18.8 Algorithm Lifecycle Management

Logically, PCM algorithms are loaded, instantiated, configured, assigned to a subset of data traffic, and activated to provide congestion control capabilities. The remainder of this section outlines this flow.

3.6.18.8.1 Loading Algorithms

PCM algorithms are dynamically loaded using implementation-specific tools. Algorithm implementors specify required PCM features for a particular algorithm by providing a load-time specification along with the algorithm that is consumed by the load tool to determine if the implementation can support the algorithm and to configure the PCM subsystem for the algorithm. Load-time specifications provide the following information on an algorithm basis:

1. The set of configuration parameters and associated default values.
2. All signals and their mapping to indices, and initial values for new CCCs.
3. All outputs and their mapping to indices.
4. All local storage variables and their initial values.
5. The invocation callback function.
6. The set of initial triggers for new CCCs.
7. The default *cwnd* and *send_rate* values for new CCCs.

Load-time specifications are expressed in a hierarchical YAML format as described in Section 3.6.18.8.5.

3.6.18.8.2 Algorithm Instances

Algorithm instances are specific, independent realizations of a loaded PCM algorithm, complete with configuration and state, and are used to manage congestion control for an administrator-defined subset of bulk data generated by the FEP.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

3.6.18.8.3 Configuring Algorithm Instances

Algorithm instances are configured using implementation-specific tools. The set of configurable parameters are algorithm-specific and provided to the PCM subsystem as YAML definitions as described in Section 3.6.18.8.5.

3.6.18.8.4 Assigning Algorithm Instances to Network Traffic

PCM supports assigning algorithm instances to network traffic on a UET-instance basis. In this model it is assumed the FEP is configured to operate over one or more UET instances. Every instance has two or three traffic classes as outlined in the Section 3.6.4.7. All the traffic in a PCM-managed UET instance MUST be controlled exclusively by a single CC and a single LB algorithm instance.

3.6.18.8.5 Load-time Algorithm Configuration YAML Format

PCM algorithms, their signals, accumulations, local state, and control outputs are configured via a specified YAML format. The YAML file contains the following top-level keys:

- Algorithm: Details of the algorithm specification
- Signal: Configuration of the used signals, accumulations, and triggers
- Control: Output control definitions
- Variable: Definition of local-state variable and its initialization
- Parameter: Constant algorithm configuration parameters

The YAML template below defines the format of the file and its valid fields. Optional fields are in square brackets. The type of *<c-identifier>* is a string that adheres to the C programming language definition of an identifier. The type *<enum>* refers to names in the referenced table. The types *uint, int, float* refer to the corresponding C types expressed in usual C string syntax (float values in scientific notation, e.g., “1.3e5”). A *<scalar>* is any of those.

```
# ——— CC Algorithm metadata (mapping node) ———
cc_algorithm:
  name: <c-identifier>     # Algorithm name (entry point of the function)
  description: <string>    # Algorithm description (free text)
  max_ack_coalescing: <uint>   # max number of ACKs to be coalesced
  [uint_min_bitwidth:  <uint>] # minimum required bitwidth for unsigned integers
  [int_min_bitwidth:   <uint>] # minimum required bitwidth for signed integers
  [float_min_bitwidth: <uint>] # minimum required bitwidth for float numbers

# ——— Input signals (sequence node) ———
signal:
  name: <enum>           # Signal name (see Table XXX)
  macro: <c-identifier>  # Signal name in algorithm code (defined as macro)
  scope:<EV|CCC>         # Scope of signal (per-EV or per-CCC)
  [accumulator: <enum>]  # Signal accumulation (see Table YYY, also for default)
  [accumulator_const: <scalar>] # Acc. Constant, only used for s in uet_op_emwa
    [trigger:
      op: <eq,neq,gt,gte,lt,lte,wc>  # Trigger comparison operator
      mode: count_or_time            # Trigger fire mode
      mode_refval: <uint> <uint>     # Fire mode reference value tuple
      arm_mode: DISARMED|ONCE|CONTINUOUS   # Trigger armed mode
      action: INVOKE|RECORD          # Action performed on fire
      invoke_deadline: <uint>        # Invocation deadline
```


Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

```
    ]
# ——— Control outputs (sequence node) ———
control:
  name: <enum>            # Control name (see Table ZZZ)
  macro: <c-identifier>   # Control name in algorithm code (defined as macro)
  initial_value: <uint>   # Control value at (re)start of algorithm

# ——— Algorithm-specific internal state variables (sequence node) ———
[variable]:
  macro: <c-identifier>   # Variable name in algorithm code (defined as macro)
  dtype: <uint|int|float> # datatype from C code
  initial_value: <scalar> # Variable value at (re)start of algorithm

# ——— Algorithm-specific global state variables (sequence node) ———
[global_variable]:
  macro: <c-identifier>   # Variable name in algorithm code (defined as macro)
  dtype: <uint|int|float> # datatype from C code
  initial_value: <scalar> # Variable value at creation time
  accumulator: <enum>     # Variable accumulation (see Table YYY, also for default)

# ——— Algorithm-specific parameters (sequence node) ———
[parameter]:
  macro: <c-identifier>   # Parameter name in algorithm code (defined as macro)
  dtype: <uint|int|float> # datatype from C code
  value: <scalar>         # Value
  description: <string>   # Parameter description

# ——— Telegrams (sequence node) ———
[tgm]:
   macro: <c-identifier>     # Message name as referenced in algorithm code (macro)
   [collector: <IP-address>] # Address for telegram collector destination
 type_id: <uint8>            # Message type id; unique for every message type
   field:                    # This structure is repeated for every field
     macro: <c-identifier>   # Field name
     dtype: <uint|int|float> # Field type
     description: <string>   # Field description

# ——— LB Algorithm metadata (mapping node) ———
lb_algorithm:
  op_model: ev-table | per-packet
  lb_entry: <string>              # PCM LB algorithm entry point (C symbol)
  hw_algo: <string>               # hardware LB algorithm name, e.g. "round_robin"
                                  # REQUIRED for op_model: ev-table
                                  # MUST NOT be present for op_model: per-packet
  hw_config:                      # hardware-specific configuration
    # algorithm-specific keys and structure
    # REQUIRED for op_model: ev-table
    # MUST NOT be present for op_model: per-packet

  ev_fields:                      # per-EV fields (EV-table model only)
    - name: <string>
      type: uint | int | float
      role: ev_value | null       # only known role is "ev_value"; can omit/set null
      sw_access: ro | rw | wo | none
      range: [min, max] |
             { enum: [v0, v1, ...] }
      init: cfg_provided | zero | hw_init |
            { explicit_value: <scalar> }

  ccc_fields:                     # per-CCC fields (EV-table model only)
    - name: <string>
      type: uint | int | float
```


Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

```
    sw_access: ro | rw | wo | none
    range: [min, max] |
           { enum: [v0, v1, ...] }
    init: cfg_provided | zero | hw_init |
          { explicit_value: <scalar> }
```

For example, the signal interface allows the definition of timed trigger that fires every 2 ms as:

```
signal:
  name: "uet_time_current"
  macro: "t_time"
  accumulator: "uet_last"
    trigger:
      op: wc                # Wildcard operator, don't compare on value
      mode: count_or_time   # Trigger fire mode
      mode_refval: 0 15625  # Fire every 2ms
      arm_mode: continuous  # Fire continually
      action: invoke        # Invoke PCM
      invoke_dealine: 0     # Don't delay invocation
```

Parameters can be used to differentiate multiple instances of the same algorithm. For example, there could be two instances of NSCC—one for AI and one for storage traffic—with different parameters. The configuration to invoke is defined in Section 3.6.18.8.5. Each of the two instances would use a separate YAML configuration, and the difference between those instances (their personality) can be configured through parameters in the YAML file.

3.6.18.9 Algorithm Runtime Model

To guarantee correct operation regardless of implementation, PCM algorithms MUST be written to assume the following runtime PCM runtime properties:

- The PCM subsystem invokes algorithm callback functions when one or more triggers fire due to TX, RX, or TIME events, when a transmit EV is required by the data path or when telegrams are delivered to the algorithm.
- The algorithm is invoked once for every set of fired RX, TX, or TIME triggers, or EV transmit request or received telegrams.
- Each invocation delivers only the set of fired triggers, an EV Transmit Request, or a telegram.
- The PCM subsystem guarantees that for a given CCC there is a maximum of one invocation active at any time.
- For TX or RX events, the set of fired triggers associated with these events is delivered to the algorithm in packet order, but events may be skipped.
    - TX events are delivered in packet order.
    - RX events are delivered in packet order.
    - There is no delivery order guarantee between TX and RX events.
    - When events are skipped, no triggers for the associated event are delivered.
- For TIME events, fired triggers are reliably delivered but may be delayed.
    - There is no delivery order guarantee between different TIME events.
    - There is no delivery order guarantee between TIME, RX, and TX events.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

- For transmit EV requests, requests are reliably delivered in data path generation order.
- Telegrams are delivered in received order.
    - Local and remote telegrams are delivered in received order.
    - There is no delivery order guarantee between local and remote telegrams.
- Control operations are executed in issued order.

Informative Text:
Parallelization and concurrency depend on the implementation; as long as the specified runtime model is supported, implementations may innovate freely.

3.6.18.10 eBPF Compliance

Implementations MAY advertise PCM eBPF compliance. This allows implementations to formally verify loaded algorithms for safety and security. Algorithms written to comply with this capability MUST adhere to the following constraints:

- Only bounded loops
- No recursion
- All pointer accesses must be expressed so they can be verified for range and access validation
- Limit stack size to a maximum of 512 B
- Limit call depth to a maximum depth of 8 calls
- Limit tail-call depth to a maximum of 32 tail-calls

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

3.7 Transport Security Sublayer (TSS)

This section defines an optional method of end-to-end confidentiality, integrity, and basic service protection for UET. The extent of this security solution is FEP to FEP. Mapping from the FEP to other security infrastructure (e.g., OCP SIOV, PCIe SRIOV, or CXL/PCIe TDISP) within the node is out of scope of this document. The specification has been inspired by sRDMA [33] and ReDMArk [3] and adopts some of their approaches in order to allow efficient prevention of all documented attacks. In addition, this specification builds heavily on from PSP, IPSec, and MACSec specifications.

3.7.1 Introduction

Many good choices exist for transport security. However, having a large number of dynamic endpoints makes traditional point-to-point encryption protocols (e.g., TLS, IPsec) difficult to deploy due to the large session state (keys) required. In addition, the ephemeral nature of the PDS layer creates issues in using a connection-oriented protocol. Google's PSP [34] is a new protocol that makes a number of simplifications with respect to IPsec. One of PSP's key simplifications is utilizing a key derivation function (KDF) to derive per-security association keys from a single, receiver-based, primary key. This reduces the amount of key storage required at the receiver. However, it still requires a key exchange and storage for each connection at the sender. The Transport Security Sublayer (TSS) described here addresses this problem for large-scale HPC or AI/ML deployments with a large number of endpoints. One of the key reasons for this secure protocol definition is to solve the transmitter state scaling issue and provide an efficient mapping to the UET.

3.7.2 Security Model

3.7.2.1 Trust Model

A ”zero trust” security architecture is used. This implies an end-to-end solution where elements of the transport network (e.g., switches) are not trusted. For purposes of this standard, a portion of the FEP hardware responsible for the transport security processing is trusted and MUST form the trust anchor for the solution.

3.7.2.2 Threat Model

The threat model describes attacker tools, threats, and mitigations of in-scope threats.

Table 3-127 - Threat Model Definitions

Term	Definition
Attacker	Entity that wants to extract information from a communication or modify communicated data.
Ciphertext	The packet data containing the encrypted plaintext that is sent on the wire between sender and receiver.
Information	Data or properties of the data exchanged between two participants that would allow the attacker to take or cause an adverse action. Examples include cryptographic keys, decisions of the FEP processing, etc.
Intermediary/switch	An entity that routes or forwards packets to a receiver.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Term	Definition
Plaintext	The original data that needs to be encrypted by the sender before transmission and the resulting data after it is decrypted by the receiver.
Protocol secrets	UET secrets that are protected from users of the protocol and/or attackers to maintain the trusted connection.
Side channel	A method for an attacker to extract information without the knowledge of the sender or receiver.
Threat	Damage or danger that could expose protocol secrets, allow the leaking of packet data, or degrade the integrity of the network.
Threat mitigation	How TSS specifically addresses the possible threat.
In-scope threat	Threat that is explicitly addressed by TSS and has defined mitigations.
Out-of-scope threat	A threat that is not considered or addressed in this specification.
Trusted entity	Portion of the FEP entrusted to handle key material and perform cryptographic functions.
Privileged entity	A portion of the FEP and kernel driver that is responsible for assigning transport-critical information such as JobID and security context.
User entity	User application that uses a UET transport service

Table 3-128 describes the various mechanisms available to the attacker. On-path attacks are where the attacker is inserted in line between the sender and receiver. Off-path attacks describe the case where the attacker within the network uses networking infrastructure (mirror, etc.) to observe and inject traffic.

Table 3-128 - Tools Available to Attacker

Attack Tool	Type of Attack	Purpose
Arbitrary network packet insertion	• Chosen ciphertext	Purpose is to cause protocol faults or misplacement of data. In-line and off-path insertion is in scope.
Arbitrary network packet deletion	• Chosen ciphertext	Deletion of packets on path.
Arbitrary network packet replication	• Replay attack	Replication of packets on and off path.
Arbitrary network packet modification	• Chosen ciphertext at application layer (user)	Modification of packets on and off path.
Arbitrary privilege injection packet modification	• Chosen ciphertext at driver/privilege layer (user)	Compromise of privileged portions of driver/FEP is out of scope.
Arbitrary application insertion	• Chosen plaintext • Performed at application layer	Purpose is to cause protocol faults or misplacement of data.
Observe (snoop) traffic	• Performed at any location within the network or endpoint outside of the privilege and/or trusted portion of the FEP	Observe application patterns between sender and receiver to obtain information about higher layer protocols. Collection of data for off-line data analysis.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Attack Tool	Type of Attack	Purpose
Arbitrary timing attacks	• Chosen cipher or plaintext • Packet can be inserted at a specific time or sent by brute force to achieve desired timing	Observe application patterns between sender and receiver to obtain information about higher layer protocols.

3.7.2.2.1 Security Assumptions

The threat model is based on the following assumptions:

- There are no errors in the implementation of the UET protocol, regardless of whether it is implemented in hardware, software, or firmware. This implies that the attacker cannot be inside the protocol.
- The protocol MUST authenticate all portions of the packet including authenticated data and encrypted payload data that is used in the security solution.
- The protocol MUST encrypt all payload data portions of the packet.
- The protocol MUST maintain data separation between secure domains.
- The protocol MUST maintain data separation between clients in a client-server model.
- An implementation of the UET protocol MUST NOT disclose protocol secrets to users of the protocol and/or an attacker. FEPs MUST have a secure location in which to store and/or retain this information.
- The protocol cannot defend against an attacker from within the trusted portion of the FEP. An attacker in the FEP can use the protocol but is constrained by the requirements of this threat model.
- Other than the trusted entities within a FEP, the UE network and network elements such as switches are untrusted.
- Attestation, authentication of each FEP, and key generation/distribution/refresh are implementation specific and are considered out of scope.
- Critical transport parameters (JobID, security context, etc.) are assigned by an uncompromised privileged entity.
- Congestion information (ECN, trimmed packets, etc.) from the switches is not authenticated and is therefore assumed to be untrusted.

Table 3-129 - Threats and Mitigations

Threat from Attacker	Threat Mitigation
Extracting protocol secrets (keys, etc.)	Minimize the amount of packet information that is unencrypted to minimize the attack surface. • Ensure protocol secrets are not contained in unencrypted information regions. • Ensure that IV is not reused (nonce) in AEAD cipher. • Ensure the key invocation limit for AEAD cipher is not violated.
Arbitrarily insert traffic into the network outside of the endpoint as a manipulator-in the-middle	• Reliable ordered delivery (ROD) and reliable unordered delivery (RUD) utilize the packet sequence number (PSN) to detect packet insertion, removal, out-of-order packets, and packet replay. Connection replay is protected using a secure *Start_PSN* as described in the UET PDS specification. • Reliable unordered delivery for idempotent operations (RUDI) packet insertion is inherently vulnerable to replay. Packet removal and replay detection is implementation-defined.
Arbitrarily replay legitimate traffic over the network outside of the endpoint as a manipulator-in the-middle	
Arbitrarily remove traffic from the network outside of the endpoint as a manipulator-in the-middle	

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Threat from Attacker	Threat Mitigation
	• Unreliable unordered delivery (UUD) requires a higher-level application to implement packet insertion, removal, and replay detection and is out of scope for UET.
Arbitrarily modify existing traffic in the network outside of the endpoint as a manipulator-in the-middle	• The packet data including encrypted payload is authenticated so any modified data can be detected, and the packet is dropped and reported when it fails authentication. Once a packet is authenticated, the receiver can trust the routing information it contains and verify it is the intended recipient of the received packet. • Trimmed packets utilized for congestion control do not contain an ICV since the original payload is removed when it is trimmed. This makes them vulnerable to this attack. A trimmed packet is used only for congestion control state and cannot open a connection or cause misplacement or corruption of original data.
Perform timing attacks by brute force or inserting traffic at a particular time as a manipulator-in the-middle	Replay of memory transactions that open a previously closed PDC is considered a timing attack. The protocol will require the use of a secure *Start_PSN* that is saved by the sender and receiver or assigned by the receiver and is never reused for the same encryption/decryption key. Attempting to replay the request to open a connection after the connection was closed will result in the request being rejected and reported by the receiver since the PSN will be invalid. PDS is required to close after 2^{31}-32K packets. This prevents replaying across counter wraps.
Arbitrarily insert traffic into the network at the application layer and observe ciphertext	The protocol utilizes AES-GCM-256 encryption to make brute force discovery of the plaintext cryptographically infeasible.
Snoop traffic at any location in the network to derive actionable data or information as a side channel	• The protocol authenticates the entire packet including packet headers used in the security protocol and encrypt all payload data. • Side-channel attacks based on the encrypted data is considered out of scope of the UET threat model. • Side-channel attacks against the clear text authenticated header are also considered out-of-scope.
Manipulation of IP DSCP	• Considered out of scope since it requires trusting/authenticating the switch function.
Connection Setup replay	• Epoch-based rejection (see Section 3.7.10) of old requests by TSS. • When the PDC is created, either a random or *Expected_PSN* is used to set *Start_PSN* (see Section 3.5.8.2.1). • Trimmed packets received by TSS cannot create a PDC.
Nonce reuse	• The *TSC* field is constructed from a packet counter and epoch. Reuse properties of the nonce are achieved if epoch is distributed by SDME, and a FEP does not reuse epoch if it cannot be stored across reboot.
Nonce hiding	• Additional random *IVMask* or *IVMask* from KDF is XORed with IV before being presented to the cipher engine.
Unreliable transport considerations	• A PDC is torn down if the number of authentication failures is above a threshold. • IV is obfuscated using *IVMask*.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Threat from Attacker	Threat Mitigation
PDS PSN re-use (wrapping)	• Timing attack using an old packet sequence number to replay the packet data. • Mitigation: Tear down the PDC after reaching *Start_PSN+Limit_PSN_Range*.
PDC/SES/CMS resource exhaustion	• Upper layers have limited resources that could be consumed for a denial of service. • Mitigation: A FEP SHOULD limit resources per source FEP.

3.7.2.3 Endpoint Considerations

A portion of the FEP is assumed to be trusted. TSS terminates within the trusted portion of the FEP. The mechanism for establishing trust to this trusted component is implementation dependent.

Implementation Note:
The security of the FEP is complex and highly dependent upon the customer requirements and vendor implementations. Vendors are encouraged to implement DPA counter measures or other techniques to reduce the attack surface to the cryptographic implementations. In addition, cryptographic hardware SHOULD be implemented in accordance with industry best practices and certified by third-party testing programs (e.g., NIST's CAVP program).

3.7.2.4 Switch Considerations

The switch is not a trusted entity from an end-point security perspective. The following sections describe a few security considerations for the switch.

3.7.2.4.1 Quality of Service/Congestion Marking

The IP differentiated services code point (DSCP) and explicit congestion notification (ECN) are used by UET congestion control to improve performance. Both fields are carried in the IPv4 TOS and IPv6 Traffic Class fields. Tampering or misconfiguration of these parameters can impede packet delivery and might cause a DoS condition. Without authentication of the ECN/DSCP header fields these changes are undetectable at the FEP, and both are considered out of scope of the trust model.

3.7.2.4.2 Packet Trimming Congestion Signal

Packet trimming is an optional UE feature. During congestion the header portion of a packet is trimmed and forwarded to the receiver to be used by UET as a congestion signal and to trigger a retransmission. A trimmed packet allows the receiver to gain additional information and affords rapid handling of retransmission. The process of trimming packets prevents authentication since portions of the packet are removed including the authentication tag. However, it is possible to decrypt trimmed packets under some circumstances and cryptographic algorithms. An implementation MAY use this decrypted information (e.g., PDS headers) for congestion control or loss detection.

An attacker could modify portions of the ciphertext, including the UET header, to perform an oracle attack. The results of the attack can affect only congestion control/packet replay state and MUST NOT affect data placement or integrity/authentication. This is the same threat model as the ECN (explicit congestion notification) described in Section 3.7.2.4.1.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Details of the trimming feature are covered in the UE trimming specification 4.1.

3.7.2.5 TSS Logical Interfaces [informative]

This section contains non-normative text that describes example PDS-TSS and TSS-Link interfaces from a logical perspective to illustrate the information that crosses the interface. The interfaces between these sublayers are implementation specific.

The TSS is responsible for authentication of the packet within the secure domain. It relies on information provided by the link layer and PDS as well as local state (SDKDB).

Before packets are sent to the TSS from either the PDS or the link layer, the following steps MUST be complete:

- SDKDB is configured for active secure domains.
- Global parameters configured (e.g., *Rekey_Mask*).
- Required initialization and configuration of cipher hardware (KDF, AES, etc.).

Figure 3-136 summarizes an example PDS-TSS interface using C function signatures. This example is presented to provide a framework for better understanding the overall architecture but does use some terminology that is not introduced until subsequent sections.

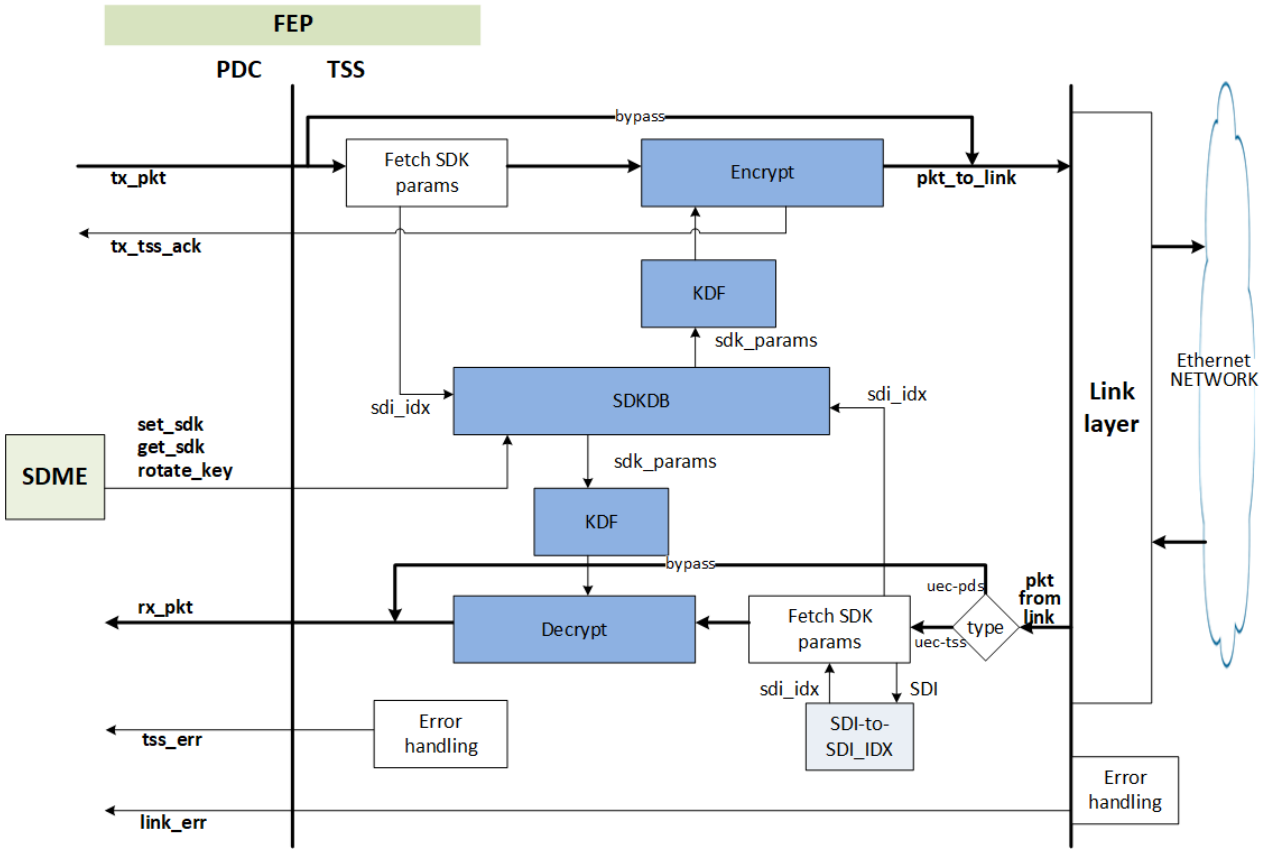


Figure 3-136 - Example PDS-TSS-Link Logical Interface

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

3.7.2.5.1 PDS—TSS Interface

The following text defines logical structures passed between the PDS and TSS. The associated example function calls are listed in Table 3-34.

PDS - TSS

```
uint32_t sdi_idx            # index into SDKDB database
uint32_t src_ssi            # secure source identifier, optionally valid
                            #   depending on ssi-mode
uint32_t dest_ssi           # SSI for destination, needed for client-server
                            #   KDF mode when source SSI is not used
uint16_t pkt_len            # packet length in bytes including headers,
                            #   excluding TSS headers
uint8_t port_num            # port number identifying port to transmit packet
uint16_t tx_pkt_handle      # locally assigned packet handle, used to
                            #   associate TSS tx result with tx_pkt
void *pkt                   # ptr to packet that includes SES & PDS headers
                            #   as well as all UDP/IP/ETH headers
uint16_t tx_tss_resp        # response back to PDS indicating success/fail
                            #   on transmit path
bool    rx_auth             # if set, the TSS authenticated the received
                            # packet and it was mapped to a secure domain
uint8_t tss_err             # if non-zero, indicates TSS error type
uint8_t link_err            # if non-zero, indicates link error event type
bool    tx_bypass           # if set, TSS does not encap or process packet
```

SDME - TSS

```
struct sdi_param *sdi_info  # security parameters, e.g., see Table 3-133
struct sdki                 # SDKi key
uint8_t select_an           # selects either SDKiAN0 or SDKiAN1
```

Table 3-130 - Example Functions between PDS/SDME and TSS

Function Name	Direction	Description and Parameters
tx_pkt()	PDS to TSS	Packet generated by SES, passed from PDS to TSS to be processed, encrypted, and transmitted. {*sdi_idx, src_ssi, dest_ssi, pkt_len, port_num, tx_pkt_handle, *pkt, tx_bypass*}
tx_tss_ack()	TSS to PDS	TSS letting PDS know if the processing was successful {*tx_pkt_handle, tx_tss_resp*}
rx_pkt()	TSS to PDS	Packet arrived from network, passed to TSS to be processed, decrypted, and passed to PDS {*port_num, *pkt, auth, pkt_len*} In this example, the SDI from the packet is mapped to a local index for efficiency in fetching SDI params
set_sdkdb()	SDME to TSS	SDME programming an SDKDB entry {*sdi_idx, *sdi_info*}
get_ sdkdb()	SDME to TSS	SDME fetching a current SDKDB entry {*sdi_idx, *sdi_info*}

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Function Name	Direction	Description and Parameters
rotate_key()	SDME to TSS	SDME updating the SDKDB {*sdi_idx, select_an, sdki*}
tss_err()	TSS to sys	Generic error event in TSS, report to sys/mgmt

3.7.2.5.2 TSS—Link Interface

The following defines logical structures passed between the link layer and TSS. The associated example function calls are listed in Table 3-131.

Table 3-131 - Example Functions between link layer and TSS

Function Name	Direction	Description and Parameters
pkt_to_link()	TSS to Link	Packet from TSS to link layer for transmission {*port_num, *pkt, pkt_len*}
pkt_from_link()	Link to TSS	Packet from link layer to TSS from network {*port_num, *pkt, link_err, pkt_len*}
link_err()	Link to sys	Generic error event in TSS, report to sys/mgmt

Implementation Note:
The TSS layer MAY drop frames based on misconfiguration or improper security parameters (e.g., wrong *SDK*, which causes an authentication failure). These packets would be never delivered from the PDS perspective and assumed dropped. This will cause the PDS to retransmit the packet, and if configuration is not corrected it may eventually cause the PDC to time out and be torn down. If multiple SDIs share the same PDC resources, this timeout may affect the other SDIs that are not misconfigured.

3.7.3 Architecture

A central concept in TSS is a secure domain, which is a collection of FEPs that can communicate with confidentiality and authenticity guarantees with a shared secret. The secure domain is explicitly identified independently from the source FEP using separate fields in the packets. This allows for independent scaling of sources and domains.

The secure domain spans from the TSS layer to all FEPs within the domain. Each FEP's TSS is deemed trusted, meaning it is sufficiently segregated from the user parts of the system and attested according to the system's security requirements. The process of establishing a trust relationship with the TSS varies by implementation. Cryptographic operations and the storage of key materials occur exclusively within the TSS. From the standpoint of an application, a “service” is initiated via Libfabric APIs and linked to addressing details {JobID, PIDonFEP, Resource Index} along with security parameters (SDI, AN, etc.). These resources are set up and authenticated within the FEP's privileged segment and the kernel driver framework. A binding verification in the SES layer confirms that a service adheres to its designated secure domain. Lastly, conventional kernel isolation methods (e.g., process ID) are employed to route packets to the user space.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

TSS does not have the concept of a controlled/uncontrolled port that supports transition between protected (encrypted/authenticated) and normal traffic. All TSS security parameters are exchanged as part of the initialization of the service. Mixing of services with and without TSS MUST be supported simultaneously. Other non-UE packets (e.g., TCP/IP) that do not use TSS services MUST be ignored and passed along to upper layers.

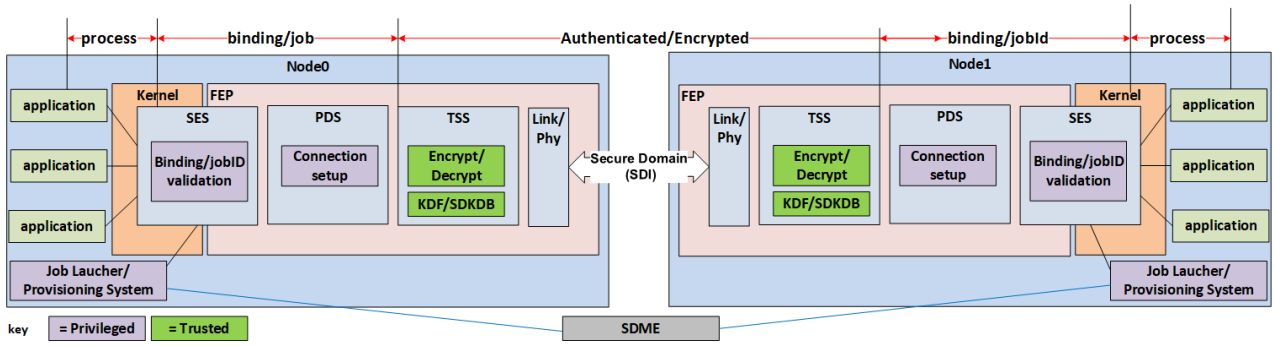


Figure 3-137 - TSS Architecture

Implementation Note:
The security model requires cooperation with the SES layer and libfabric mapping. A service is mapped to an endpoint in libfabric. The SES layer maps the endpoint to a JobID and Resource Index, which is mapped to a security context that contains the keys and secure domain (SDI). In addition, replay protection relies on PDS services. See Section 3.4.2.3.

3.7.3.1 Use Cases

The following section describes the common use cases considered in the development of this specification. Other uses cases are possible but not considered and are out of scope. In these use cases, a FEP MAY contain multiple jobs and groups of processes related to a single OS instance. Finally, a node MAY contain one or more FEPs.

KDF modes are defined in Section 3.7.7. For clarity, the possible KDF modes for each use case are specified. The SSI MAY be carried explicitly in the frame using the SSI field or source IP address depending upon configuration.

3.7.3.1.1 Single Job Within a Secure Domain

A single job may contain several processes across multiple nodes. An SD is created for the job, and an SDI is assigned that is used by all FEPs that are part of that job. Note that there is no security-level isolation between processes within the job or secure domain. All KDF modes MAY be used with this use case.

Once all processes of the job are complete on all nodes, the SD MAY be removed.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Informative Text:
An administrator can maintain state across job runs to reduce initialization time. Figure 3-138 shows two jobs with separate domains.

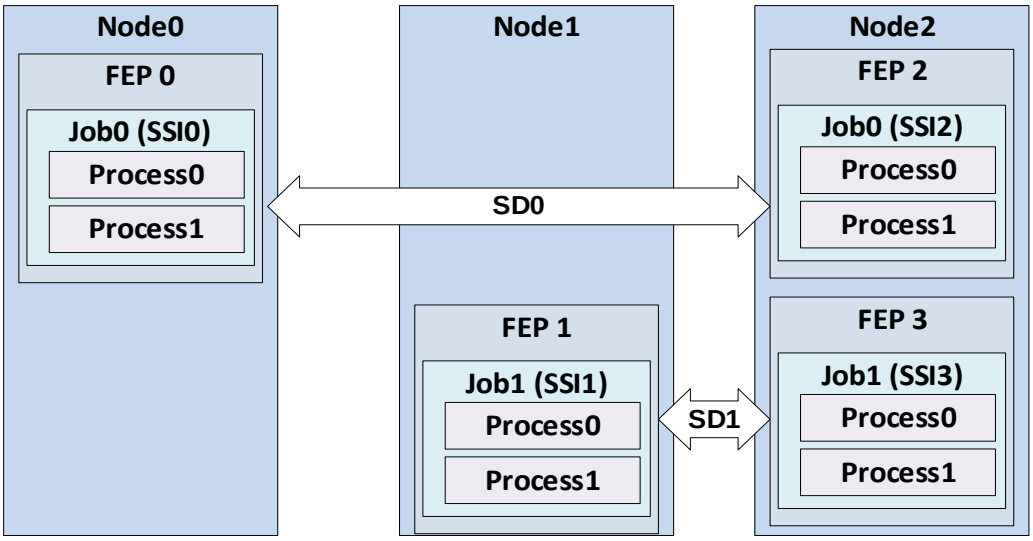


Figure 3-138 - Single Job in a Secure Domain

3.7.3.1.2 Multiple Jobs Within Secure Domain

This use case has multiple jobs with several processes, each across multiple nodes. The jobs in this example do not require isolation between FEPs.

A secure domain is created and assigned an SDI and used by all associated jobs within the domain. All KDF modes MAY be used with this use case. Relative addressing SES mode is typically used in this use case.

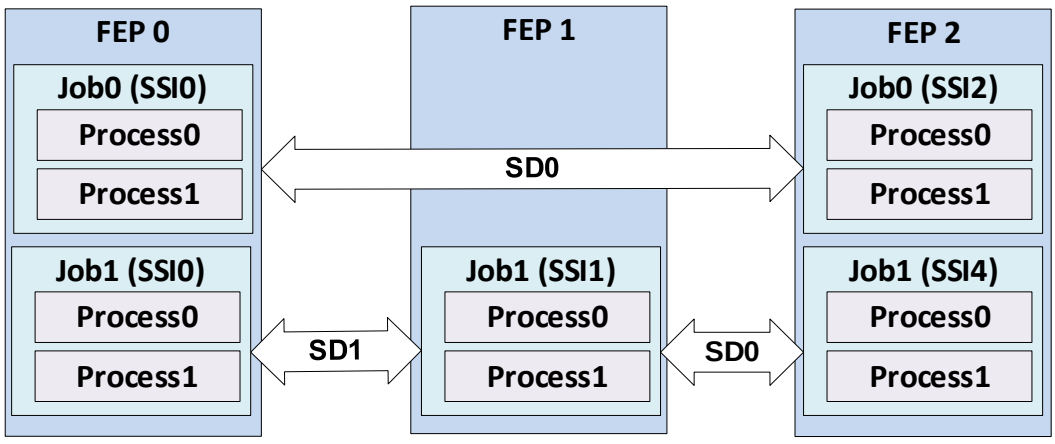


Figure 3-139 - Multiple Jobs in a Single SD

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

3.7.3.1.3 Client-Server Model

The client-server model is used when several clients communicate to the same server. In this model, within an SD a client can decrypt packets only from the server and is prevented from decrypting other clients' traffic. The server uses a common key (*SDK*s) to derive keys for each client. This is similar to how PSP derives keys, except the SDI selects a group instead of it being global to the NIC. To reduce per-client overhead at the server, per-client keys are not stored but derived using the server key. This is accomplished using a combination of KDF modes at clients and server.

Like the other cases, the process starts with creating an SDI for the secure domain and unique secure source identifiers within the domain. Unlike the other cases, each client is assigned a different key by the server, derived from the single server group key (*SDK*). This implies the server is distributing keys to the clients or that the SDME is aware of the client-server relationship. The clients use direct KDF mode to encrypt traffic to the server. The server uses cluster KDF mode for traffic received from clients and client-server KDF mode for traffic sent to clients. Client-server communication is asymmetric in that it uses client-server KDF mode for server-transmitted packets and cluster KDF mode for packets received by the server.

Implementation Note:
Job identifiers (JobIDs) are added to Table 3-132 for clarity but are not strictly used by TSS. In typical deployments, clients are identified by the server using the JobID. For clients A and B, the JobIDs at the server are JobID-A and JobID-B, respectively. The server can use either a unique server JobID (JobID-srv) or the client's JobID (JobID-A or JobID-B). This use case typically uses absolute addressing in the SES layer. In addition, buffers may be exposed to specific jobs only.

The following example shows how two clients A and B communicate with server S in secure domain *SDI0*.

Table 3-132 - Client-Server Key Generation

Case	Source Key Gen	Destination Key Gen	Comment
A->S	A: SDKDB(*SDI0*) -> *SDKa, SSIa* *JobID-A*	S: SDKDB(*SDI0*)->*SDKs*, KDF(*SDKs, SSIa*)->*SDKa*	• Direct KDF mode at A • Cluster KDF mode at S
B->S	B: SDKDB(*SDI0*) -> *SDKb, SSIb* *JobID-B*	S: SDKDB(*SDI0*)->*SDKs*, KDF(*SDKs, SSIb*)->*SDKb*	• Direct KDF mode at B • Cluster KDF mode at S
S->A	S: SDKDB(*SDI0*) -> *SDKs, SSIs* KDF(*SDKs, SSIa*) -> *SDKa* *JobID-Srv* or *JobID-A*	A: SDKDB(*SDI0*)->*SDKa*	• Server KDF mode at S • Direct KDF mode at A
S->B	S: SDKDB(*SDI0*) -> *SDKs, SSIs* KDF(*SDKs, SSIb*) -> *SDKb* *JobID-Srv* or *JobID-B*	B: SDKDB(*SDI0*)->*SDKb*	• Server KDF mode at S • Direct KDF mode at B
Note: • *SDI0* is the secure domain used. • A->S is Client A sending to Server S (similar for B) • S->A is Server S sending to Client A (similar for B)			

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

3.7.3.1.4 FaaS (Function-as-a-Service)

FaaS refers to Function-as-a-Service workloads such as an AI inference system where the same cluster of accelerators might be serving different models from different tenants. This MAY be implemented using client-server KDF mode.

3.7.4 Secure Domains

A secure domain is a collection of FEPs that can communicate with confidentiality and authenticity guarantees with a shared secret. A secure domain uses a common symmetric key and optionally a KDF to secure communication. FEPs within a secure domain MUST be assigned a unique secure source identifier (SSI) or an IP address provided it is unique within the domain. Each secure domain MUST be assigned a secure domain indicator (SDI) that MUST be unique within the reachable network.

Figure 3-140 shows a secure domain key (*SDKx*) is used for all members of the secure domain (i.e., the figure shows direct *kdf-mode*). The SSIs are unique within the secure domain and assigned to FEPs 0 through 3.

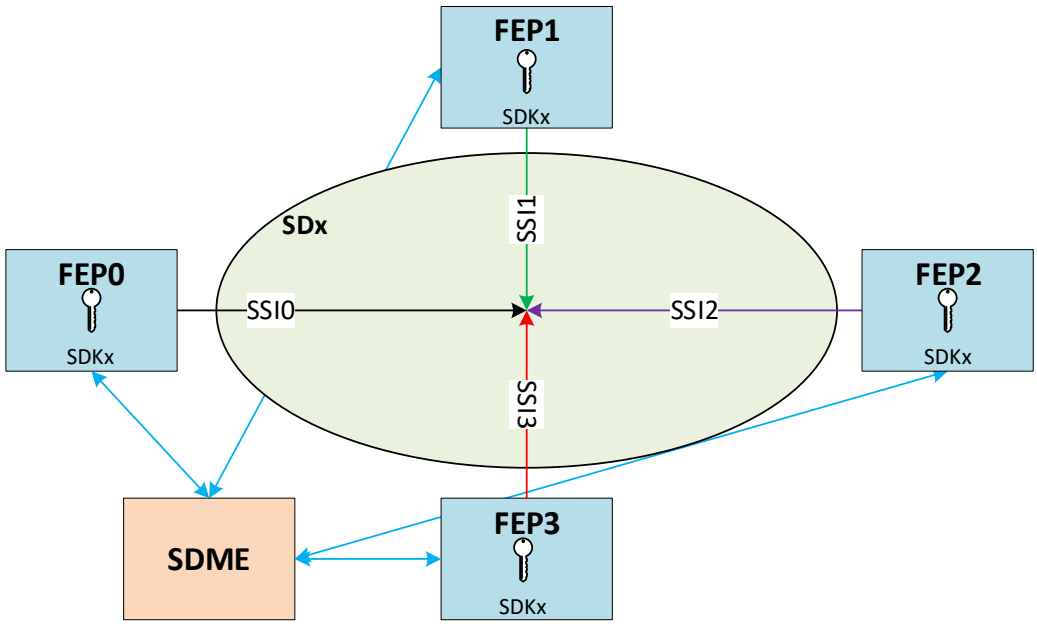


Figure 3-140 - Secure Domain with Four FEPs

Secure domains are administered by a secure domain management entity (SDME). The SDME MAY perform its responsibilities in conjunction with or be contained with other parts of the job launch services. The communication mechanism between the SDME and the FEP is out of scope of this specification.

A security association defines the initial secure domain key (*SDKi*), which is found using the secure domain identifier (SDI) and association number (AN) along with the SDKDB.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

A key epoch is used to help coordinate changing keys and ensure nonce (number used only once) properties of the IV. This concept is different than changing the security association, which specifically installs a new key. Changing the epoch performs two functions:

1. It changes a portion of the *TSC*, affecting the IV for the symmetric cipher to ensure uniqueness.
2. It MAY change the symmetric key if a KDF is used (e.g., cluster KDF mode).

The epoch MAY be global to the SDI or based on a source within the SDI. If a FEP leaves and rejoins a domain for any reason where the *TSC* cannot be preserved, the SDME MUST coordinate either an epoch update or a key rotation operation. The epoch MUST be a monotonically increasing number for the same association (e.g., without a key rotation). An epoch is scoped within a security association and MUST reset when a new key is installed.

The SDME is responsible for:

- Creating secure domains and assigning SDI values.
- Managing and updating the domain or source epoch.
- Admitting new FEPs to a secure domain (allowing them to join the secure domain).
- Managing SSI assignment to FEPs (this may directly use the source IP address).
- Distributing security parameters (*SDK*, AN, etc.) to FEPs within the secure domain. These parameters are used to populate global or SDKDB configuration. This may include coordinating the client-server KDF mode key distribution.
- Removing FEPS from secure domain.
- Coordinating key rotation.

3.7.4.1 Joining a Secure Domain

When a FEP joins a secure domain, the SDME assigns an SSI and provides security parameters. Once the FEP receives the information from the SDME, the FEP is free to send or receive packets securely. The SDME MUST provide a mechanism to ensure every FEP in the secure domain has current security parameters before FEPs are allowed to send.

3.7.4.2 Removing a FEP from a Secure Domain

The SDME may administratively remove a FEP from a security domain at any time. This can be accomplished in two ways. The first method is using the key rotation procedure defined in Section 3.7.5.1. The second method involves creating a new security domain without the FEP and transition the remaining FEPS to the new domain.

3.7.4.3 Rejoining a Secure Domain

To ensure that a nonce is not reused, a FEP MUST either re-establish communication with SDME such that the epoch can be updated or ensure that after reboot/power-failure the nonce is preserved such that it is not reused. See Section 3.7.5.5 for more details.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

3.7.5 Key Lifetime and Security Considerations

The symmetric keys (*SDK*s) used in the block cipher can be utilized only for a limited number of packets (invocations) before rekeying is required. Guidance on the invocation limits is described in [36] and further analyzed in [35].

In addition, it is assumed that the input *SDKi* is a “cryptographic key” in accordance with NIST specifications. This implies that the key is secret, random, and uniformly distributed through the bit space of key.

Several papers analyze attacks using a fixed nonce, authenticated data, and encrypted data as outlined in IETF RFC 9001 Appendix B [45] based on the analysis in Hoang et al. "The Multi-user Security of GCM, Revisited: Tight Bounds for Nonce Randomization" [46] .

During the development of this standard, discussions established an agreed-upon limit between 2^{27} and $2^{34.5}$ packets, contingent on the assumptions of the security model (single vs multi-user).

IETF RFC 8645 [47] provides a framework for discussing various strategies to manage the lifetime of symmetric keys. The direct KDF mode utilizes an “explicit approach” per Section 5.1 of the RFC. This explicitly updates the keys using the key rotation approach described the key rotation Section 3.7.5.1. All KDF modes MUST support key rotation. Cluster KDF mode MAY utilize an “implicit approach” specified in the same RFC. This mode utilizes a KDF with a parallel construction. Using this approach, keys are generated from the master key (*SDKi*) using a KDF based on a counter in the packet. The KDF rekeying Section 3.7.5.2 provides a detailed explanation of how this counter is constructed.

Key invocations are tracked at the transmitter. The transmitter MUST not use the key beyond the invocation limit. The number of invocations can be tracked using the *TSC* counter logic, which is reset to zero when a key rotation is initiated at the transmitter.

A FEP MUST drop packets if the authentication limit has been exceeded (**tss.tsc.counter** >=*invokeFatalTheshold*) and MUST provide an indication to the SDME if an SDI is beyond this limit.

In addition, a FEP SHOULD indicate to the SDME that an SDI is approaching this limit (**tss.tsc.counter** > *invokeWarnThreshold*) to allow the SDME time to coordinate a transition to a new key.

3.7.5.1 Key Rotation

Key rotation updates the security association, which in turn updates the SDK and optionally the *IVMask* for all FEPS within an SD. In direct KDF mode, key rotation MUST update the *IVMask*.

To facilitate key rotation, a receiver MUST maintain multiple keys per SD identified by the association number (AN). Changing the association resets both the counter and epoch to zero. The epoch is used to ensure that when FEPs join and leave an association, the uniqueness of the IV is preserved. The counter portion of the *TSC* is used to manage key-lifetime and ensure the IV uniqueness.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

The SDME controls when the key rotation occurs, and it MAY occur at any time. An example of when the SDME MAY update the association is when one of the FEPs in the domain indicates it is approaching the invocation limit (*invokeWarnThreshold*). When rotating keys, the following steps MUST be performed:

1. The SDME informs all FEPs in the SD to transition to a new AN.
    a. The SDME MAY provide a new *SDK* or instructions on how to obtain/derive a new *SDK* for the association.
    b. The key epoch is set to 0.
2. All FEPs MUST update the SDKDB for the SDI with the new *SDKi* in the correct key slot (*SDKiAN0* or *SDKiAN1*), update *TxActiveAN*, and set *RxActiveAN* to accept both keys.
3. All FEPs MUST start using the new security association (AN) for all traffic sent within the SDI.
    a. The counter portion of the *TSC* is reset to zero when the first packet is sent using the new security association.
    b. Packets received during this interval MAY have the new or old AN.
4. After a transition period, the old AN MUST be deactivated by updating *RxActiveAN* to only accept the new association (key). This transition period MUST be less than the key lifetime.
    a. Packets received with the non-active security association (AN) after this period MUST be silently dropped and counted (*sdiXXInInvalidSa*).
    b. Clearing the *sdixxInAuthFailPkts* counter.
5. All FEPs are now using the new AN.

3.7.5.2 KDF Rekeying

Implicit rekeying SHOULD be implemented by using the counter portion of the *TSC* in the key derivation function when using cluster KDF mode. Rekeying reduces the number of cryptographic operations of the derived key but does not improve replay protection as described in Section 3.7.5.

The rekey interval is defined by how many bits are used from the counter field of the *TSC*. The *Rekey* field is created using the *Rekey_Mask* and *Rekey_Shift* from the counter portion of the *TSC*. This allows extraction up to 32 bits from the **tss.tsc.counter** field:

```
Rekey = (pkt.tss.tsc.counter & Rekey_Mask) >> Rekey_Shift
```

Rekey MAY be disabled by setting the *Rekey_Mask* to zero. Once the selected portion (*Rekey*) of the counter changes, the output *SDK* will change since it is an input to the context field of the KDF.

For example, if the *Rekey_Mask* = 0xFFFF_0000_0000 and the *Rekey_Shift*=32, every (2^{32} -1) packets a new key will be derived. Figure 3-141 depicts the sequence of events.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

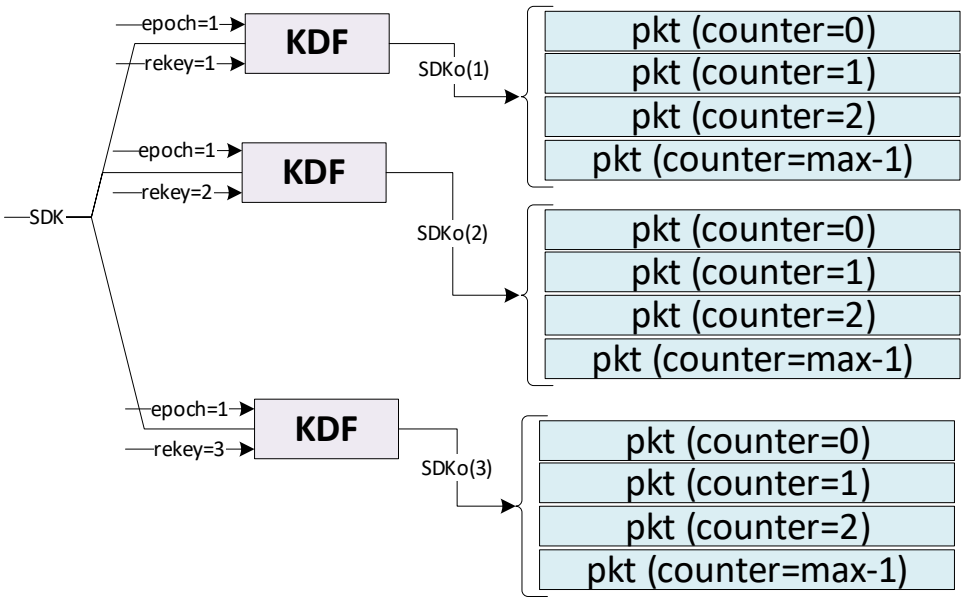


Figure 3-141 - KDF Rekeying Example

3.7.5.3 KDF Algorithms

The KDF construction is described in Section 3.7.8. The implementation of the KDF relies on a PRF (pseudo random function), which can be implemented using several algorithms (e.g., HMAC, AES-CMAC, etc.). There are several trade-offs with respect to implementation complexity and security strength. The default algorithm that MUST be supported is AES-CMAC-256 in counter mode. Implementations are free to use other algorithms within the framework of the existing construction. The assumption in this specification is that the security strength of the input key is preserved via the KDF.

Implementation Note:
This specification defines only the AES-CMAC-256 algorithm for the *kdf-algo* field. Identifying other algorithms and constructions is not currently specified and if used by implementations requires a consistent global configuration.

3.7.5.4 Symmetric Algorithms

TSS MUST use an authenticated encryption with associated data (AEAD) cryptographic algorithm, which combines both encryption and authentication. A portion of the plaintext, referred to as additional authentication data (AAD), is authenticated but not encrypted. An integrity check value (ICV) is carried with the data and used as part of the authentication.

Implementation Note:
The AES-GCM with 256-bit key size consistent with CNSA 2.0 [37] post-quantum resistant parameters. Other standards may use different assumptions for post-quantum resistance (e.g., NIST SP 800-131A [38])

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

The default cipher algorithm MUST be AES-GCM [AES-GCM] with a 256-bit key and a 16 byte ICV selected by the algorithm from the SDKDB.

There are several academic papers that provide an informative overview of the desired cryptographic properties of AEAD ciphers. Specifically references [44], [49], [50], and [51] provide the terminology used in the following sections. Additionally, the IETF Crypto (CFRG) working group is developing on an RFC draft that also summarizes these requirements and provides additional information [39].

Because out-of-order reception is used, the integrity of the AEAD cipher also depends on limiting the number of attempts to forge (authentication failures) packets. An FEP MUST drop ALL packets on the domain when the *sdiXXInAuthFailPkts* counter exceeds the *authFailThreshold*, then generate an event to the SDME.

The input key into the symmetric cipher MUST be a cryptographic key per NIST SP800-108 [40].

3.7.5.5 IV Considerations

AEAD ciphers treat the IV as a nonce and MUST NOT be reused for the same key (*SDK*). This property is called “nonce misuse” and will be prevented by constructing the nonce from an epoch and counter per the *TSC* Section 3.7.5.6.

Nonce hiding provides confidentiality for the nonce value used to encrypt plaintext. This is accomplished by creating an *IVMask* which MUST be XORed with the packet IV before it is sent into the cipher engine.

TSS MUST use a deterministic IV construction for AES-GCM ciphers per Section 8.2 of [36]. Section 9.1 of the AES-GCM specification [36] discusses loss of power and reboot with respect to a repeating IV. If a FEP reboots or reinitializes, or loses power, the FEP MUST reestablish communication with the SDME to update the epoch before sending UET traffic. Updating the epoch ensures the nonce properties of the nonce. An FEP that can store the current *TSC* MAY continue using the stored IV provided the nonce properties are preserved.

In direct KDF mode, the IVMASK MUST be provisioned by the SDME as part of the key-rotation process. The IVMASK MUST be a cryptographically random number. This is achieved by generating the IVMASK from a NIST 800-90C TRNG random source.

3.7.5.6 Time-stamp Counter (*TSC*)

The time-stamp counter is used as part of the IV for the symmetric cipher. The FEP MUST guarantee that each packet has a strictly increasing *TSC* and that packets MUST never be sent with the same *TSC* value.

The *TSC* counter is a 64-bit field that MUST be constructed by concatenation of two fields:

1. A 16-bit epoch identifier that MAY be source-specific or global to the secure domain.
2. A 48-bit counter MUST be incremented for each packet sent.

The counter portion has these additional requirements:

1. The counter MUST be initialized to 0 when the epoch is started, changed, or updated.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

2. If the counter wraps or is greater than *invokeFatalTheshold*, then all packets MUST be dropped within the secure domain and the *sdiXXOutInvokeFail* counter MUST be incremented for each failure (packet).

3.7.6 Secure Domain Key Database (SDKDB)

The secure domain is identified by the SDI carried within the packet. The *maxSDI* is a property of the FEP. The *baseSDI* and *maxSDI* are configured by the SDME after collecting the *maxSDI* from all FEPs and determining a compatible range of SDIs. The SDI MUST be allocated between the range of *baseSDI* and (*baseSDI*+*maxSDI*-1). The FEP MUST allow placement of *baseSDI* between 1 and 2^{24}-*maxSDI*-1, ensuring that SDIs are allocated in a contiguous range. If the SDKDB lookup fails, then the packet MUST be discarded and counted (*ifPPInErroredPkts*). SDI zero MUST NOT be allocated to be consistent with IPSec and PSP implementations.

The SDKDB lookup returns the fields shown in Table 3-133.

Table 3-133 - SDKDB Fields

Field	Size/type	Description
kdf-mode	enum	KDF mode (direct, cluster, client-server)
ssi-mode	boolean	Determines if SSI is obtained from the TSS header or the source IP address.
TxActiveAN	1 bit	Active AN used for transmit packets
AES-GCM-256	AES-GCM-256	AES-GCM-256
algo	enum	Cipher mode used: Only AES-GCM-256 is defined
encap-type	3 bits	Packet encapsulation used for security association 0b000: Native IPv4 0b001: Native IPv6 0b010: UDP over IPv4 0b011: UDP over IPv6 0b100: Native UFH_32 0b101 – 0b111: Reserved
Aoff	unsigned integer (2-byte units)	An offset from the start of the UET header towards the start of the packet that indicates the start of the AAD (may be passed from PDS with each packet). This is valid only for a single encapsulation type. See Figure 3-145.
Coff	unsigned integer (2-byte units)	An offset from the start of the UET header towards the end of the packet that indicates the the boundary between the AAD and the cipher text (may be passed from PDS with each packet). This is valid only for a single encapsulation type. See Figure 3-145.
SDI	24 bits	SDI used in packet—included in SDKDB if the database is indexed by a pointer (e.g., *sdi_idx* in logical interface)
SDKiAN0	256 bits	*SDKi* for AN0
SDKiAN1	256 bits	*SDKi* for AN1
Rekey_Mask	48 bits	Optional MASK for *Rekey*
Rekey_Shift	Integer (0-31)	Optional SHIFT for *Rekey*
IVMask0	96 bits	*IVMask* for AN0. Only for direct KDF mode (generated per packet, otherwise via KDF)

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Field	Size/type	Description
IVMask1	96 bits	*IVMask* for AN1. Only for direct KDF mode (generated per packet, otherwise via KDF)
current_epoch	16 bits	Current epoch to use for transmitted packets and epoch-based rejection
kdf-algo	enum	KDF algorithm; the default is AES-CMAC-256

The SSI MAY be explicitly carried within the packet (i.e., The SSI present bit, **tss.sp**=1) or the source IP address (**ip.src_addr**) MAY be used. The *ssi-mode* field MAY be a global or per SDI configuration.

Rekey is created using the *Rekey_Mask* and *Rekey_Shift* fields and MAY be used to automatically rekey via the KDF as described in Section 3.7.5.2. *Rekey* is disabled by setting the *Rekey_Mask* to zero.

Coff and *Aoff* are used to configure the AAD plaintext data for the packet. These offsets are 0 or greater and are relative to the layer3/layer4 payload depending on encapsulation. AAD refers to the authenticated but not encrypted portion of the packet. Refer to Section 3.7.11 for an illustration of the authenticated and encrypted portions of the packet.

Each SDKDB has two associated keys (*SDKiAN0, SDKiAN1*) that correspond to each AN0 and AN1, respectively. The *RxActiveAN* is used to enable each AN key. If a packet arrives with a disabled AN, an invalid AN error is generated to PDS and the packet is dropped. For direct KDF mode, *IVMask* is determined by selecting *IVMask0* or *IVMask1* from the SDKDB using the same mechanism as the SDKBD keys. Other KDF modes use the output of the KDF for the *IVMask*.

A secure domain MUST use a consistent packet encapsulation as defined by the *encap-type* field in the SDKDB.

3.7.7 KDF Modes

The three defined KDF modes are direct, cluster, and client-server. Direct KDF mode MUST use the SSI (*ssi-mode*=TRUE) if the packet is IPv6 to guarantee IV uniqueness. Other modes MAY use SSI or IP addresses to convey source information, provided they meet uniqueness criteria.

If a packet arrives with an SSI and the SDKDB is not configured to use SSIs, then the packet is dropped and the *sdiXXInInvalid* is incremented. Implementations MUST support direct KDF mode and SHOULD support both cluster and client-server KDF modes. An implementation MUST indicate which modes are supported via *kdfModes* parameter.

The following outline summarizes the *kdfModes*:

1) Direct KDF mode: Uses the key directly from the SDKDB.
    a. *SDK* = SDKDB[SDI||AN]
2) Cluster KDF mode: Sender and receiver derive keys based on source from group key (*SDKi*). The same derivation is used for both sending and receiving packets.
    a. *SDKi* = SDKDB[SDI||AN]

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

b. *SDK* = KDF(*SDKi*, label="U1", context="*current_epoch*|| *Rekey* ||SSI") where SSI may be **ip.src_addr**

3) Client-server KDF mode:
   a. Packets sent by server are derived based on destination within a group key. This is used by a server to generate client-specific keys by using the destination SSI (*destSSI*) in the KDF. Note that the SSI in the frame is still the source (server) SSI. Finally, automatic rekeying is not supported in this mode.
      i. *SDKi* = SDKDB[SDI||AN]
      ii. *SDK* = KDF(*SDKi*, label="U2", context=" *current_epoch* ||*DestSSI*/**ipv4.dest_addr/ipv6.dest_addr**")
   b. Packets received by the server are handled as cluster KDF mode packets.

The label and context fields in item 2.b above are a text strings that are required by the NIST specifications.

Figure 3-142 describes how the *SDK* lookup and interface to the crypto engines are implemented by the FEP.

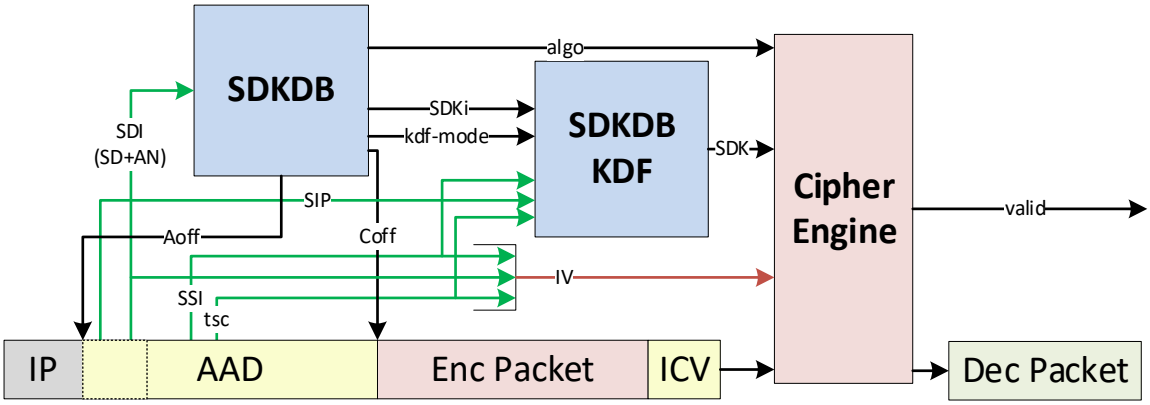


Figure 3-142 - SDKDB database and crypto interfaces

3.7.8 KDF Construction

The KDF MAY be used to derive keys in the cluster and client-server KDF modes. Note that for client-server both client and server KDFs are used. The KDF is implemented with single or two input blocks, depending on how the source FEP is represented. The single-input block KDF MUST be used for IPv4 packets or when the SSI is explicitly carried in the packet. For IPv6 packets without an explicit SSI, two input blocks MUST be used.

The KDF algorithm is specified per security domain using the *kdf-algo* field. The default KDF construction MUST be implemented per NIST specification SP 800-108 [40] using AES-CMAC-256 and parameters per Table 3-134. Other algorithms and constructions are implementation specific.

The default KDF MUST be based on the NIST specification SP 800-108 Section 4.1 [40]. CMAC is used as the associated pseudorandom function (PRF) in counter mode. The CMAC MUST be implemented using

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

AES-256 [41] per NIST specification SP 800-38B [42]. The KDF input parameters MUST be supported and are summarized in Table 3-134.

Implementation Note:
The KDF implementation is a vital part of the security solution. Vendors are encouraged to implement DPA or other techniques to reduce the attack surface of the cryptographic implementations. In addition, cryptographic hardware SHOULD be implemented in accordance with industry best practices and certified by third-party testing programs (e.g., NIST Cryptographic Algorithm Validation Program (CAVP) for the KDF [43]) .

Table 3-134 - KDF Parameter Summary for AES-CMAC-256

Parameter	Value	Description
Mode	Counter	Family of KDF
MAC	CMAC	PRF used in KDF
Block Cipher	AES-256	128-bit block size
Length L	352 bits	Length of derived key (*Ko*) key and *IVMask* in bits
Size of L	16 bits	Size of L in bits
Size of i	8 bits	Size of iteration counter in bits
LocationCounter	BeforeFixedData	Location of counter in the data
Counter Encoding	Big-endian	Both counters MUST be encoded as big-endian (MSB first) binary strings when hashed
Label Length	16 bits	Supported length of Label field (fixed) in bits
Context Length	80 bits/208 bits	Supported lengths of Context fields (fixed) in bits
iterations (n)	3	Number of PRF calls (generates 384-bit fixed output)
Ki size	256 bits	KDF input key: *SDKi* in this document
Ko size	256 bits	Output or derived key: The *SDK* in this document
IVMask	96 (128 bits)	Output of KDF used to hide nonce/IV before it is used in the symmetric cipher.

The KDF is a function that derives a key, *Ko*, or an *SDK* from an input key *Ki* or *SDKi*, label and context. The KDF is constructed from a fixed portion, a constant for all iterations, and a counter used to create outputs larger than a single iteration. The fixed portion (*KDF_fixed*) consists of a *Label, Delimiter* (0x00), *Context*, and *Length* (L) fields. The *Label* and *Context* fields MUST be set based on the *kdf-mode* and *ssi-mode*. The *Length* is the bit length of the derived key and MUST be 352 bits for AES-256+ *IVMask*. In summary, the fixed portion MUST be:

```
KDF_fixed = Label||0x00||Context||L
```

Each iteration creates a 128 bit output with this construction. Therefore, three iterations are required. The iteration counter, *i*, MUST be initialized to 1 and MUST be incremented for each iteration. Only the first 12 bytes of the KDF output are used for the *IVMask* on the third iteration. The KDF output is used as follows:

```
Ko(i) = PRF(Ki, 1||KDF_fixed) || PRF(Ki, 2||KDF_fixed)
IVMask = PRF(Ki, 3||KDF_fixed)[0-11]
```


Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

The PRF MUST be implemented using CMAC with AES-256 as the block cipher. The inputs to the CMAC function are always aligned to the block size of the cipher to simplify implementation. The *Label* field MUST be 16 bit and contain an ASCII-encoded text based on the type of KDF used.

The *Rekey* field is part of the *Context* of the *KDF_fixed* portion and MAY be used in the KDF *Context* field to facilitate automatic rekeying. *Rekey* MAY be only in cluster *kdf-mode*. From 0 to 32 bits are extracted from the *TSC* based on a mask and shift operation:

- *Rekey* = (**tss.tsc** & *Rekey_Mask*) >> *Rekey_Shift*)
    - During the logical right shift operation, zeros are injected.
    - *Rekey* MUST be initialized to zero such that if 0 bits are extracted the *Rekey* is zero.
- The *Rekey_Mask* and *Rekey_Shift* shift MAY be global or part of the SDKDB.
    - All members of the SD MUST use the same *Rekey_Mask* and *Rekey_Shift* values.

3.7.8.1 Cluster KDF mode for IPv4 and Packets with Explicit SSI

For IPv4 and SSI packets, the *Context* field MUST be 80 bits and constructed from the following fields based on *ssi-mode*:

- *LabelCluster* = 0x5531 or ‘U1’ in ASCII
- *Contextv4* = *current_epoch* || *Rekey* || **tss.ssi**: if *ssi-mode* is TRUE.
- *Contextv4* = *current_epoch* || *Rekey* || **ipv4.src_addr**: if *ssi-mode* is FALSE.

The KDF is invoked in the following manner:*IVMask* ||*SDKo* = KDF(*SDKi*, context=*Contextv4*, label=*Label*)

Figure 3-143 depicts how each iteration is accomplished for an IPv4/SSI cluster KDF mode.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

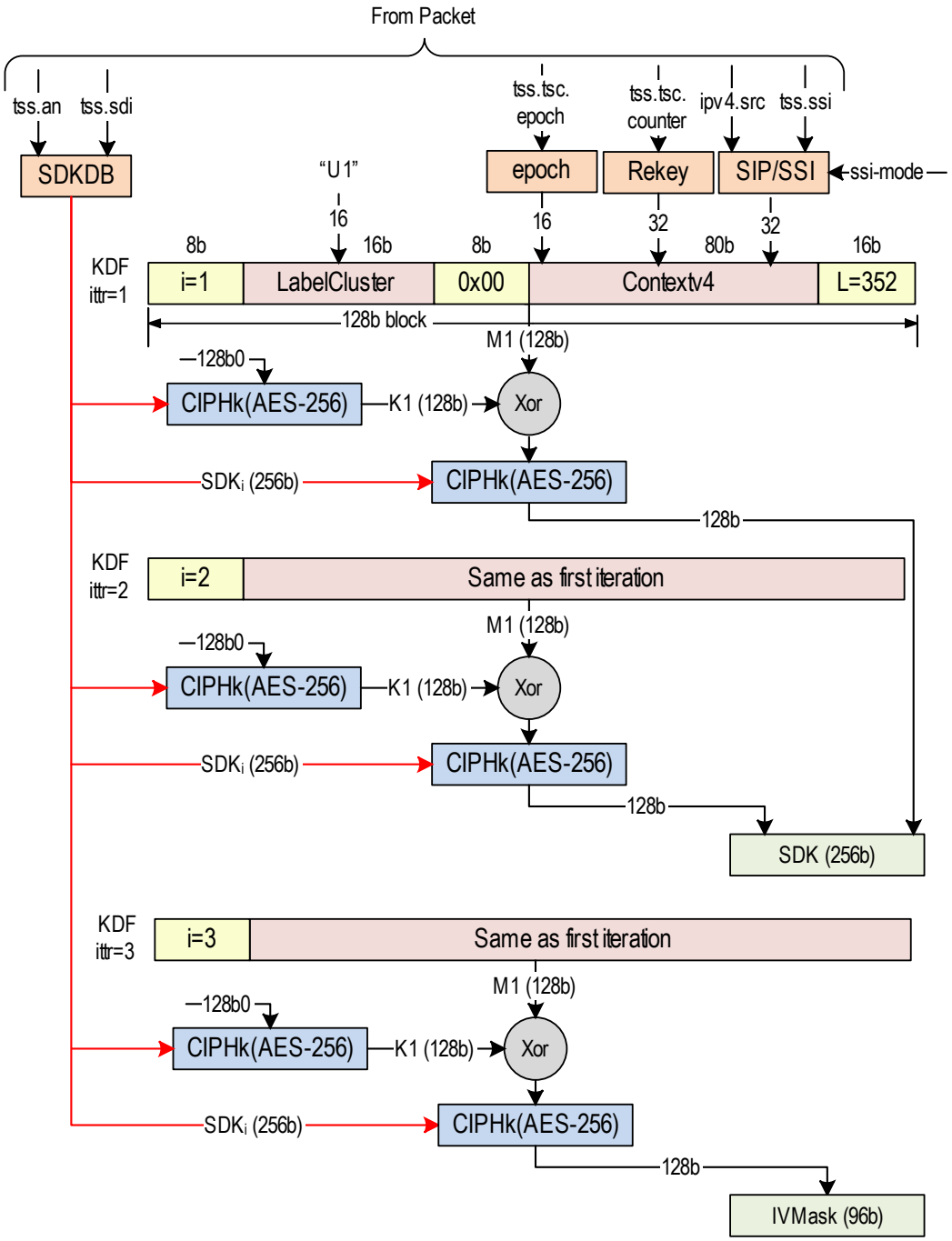


Figure 3-143 - Cluster KDF mode for IPv4 and Packets with Explicit SSI

3.7.8.2 Cluster KDF mode for IPv6 without explicit SSI

For IPv6 traffic MUST use two-block mode to accommodate the larger IPv6 source address. The *Context* in the IPv6 case is 208 bits and MUST be the following field:

- *LabelCluster* = 0x5531 or 'U1' in ASCII
- *Contextv6* = 0x00000000|| *current_epoch* || *Rekey* || **ipv6.src_addr**

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

The KDF is invoked in the following manner: *IVMask* ||*SDKo* = KDF(*SDKi*, context=*Contextv6*, label=*LabelCluster*)

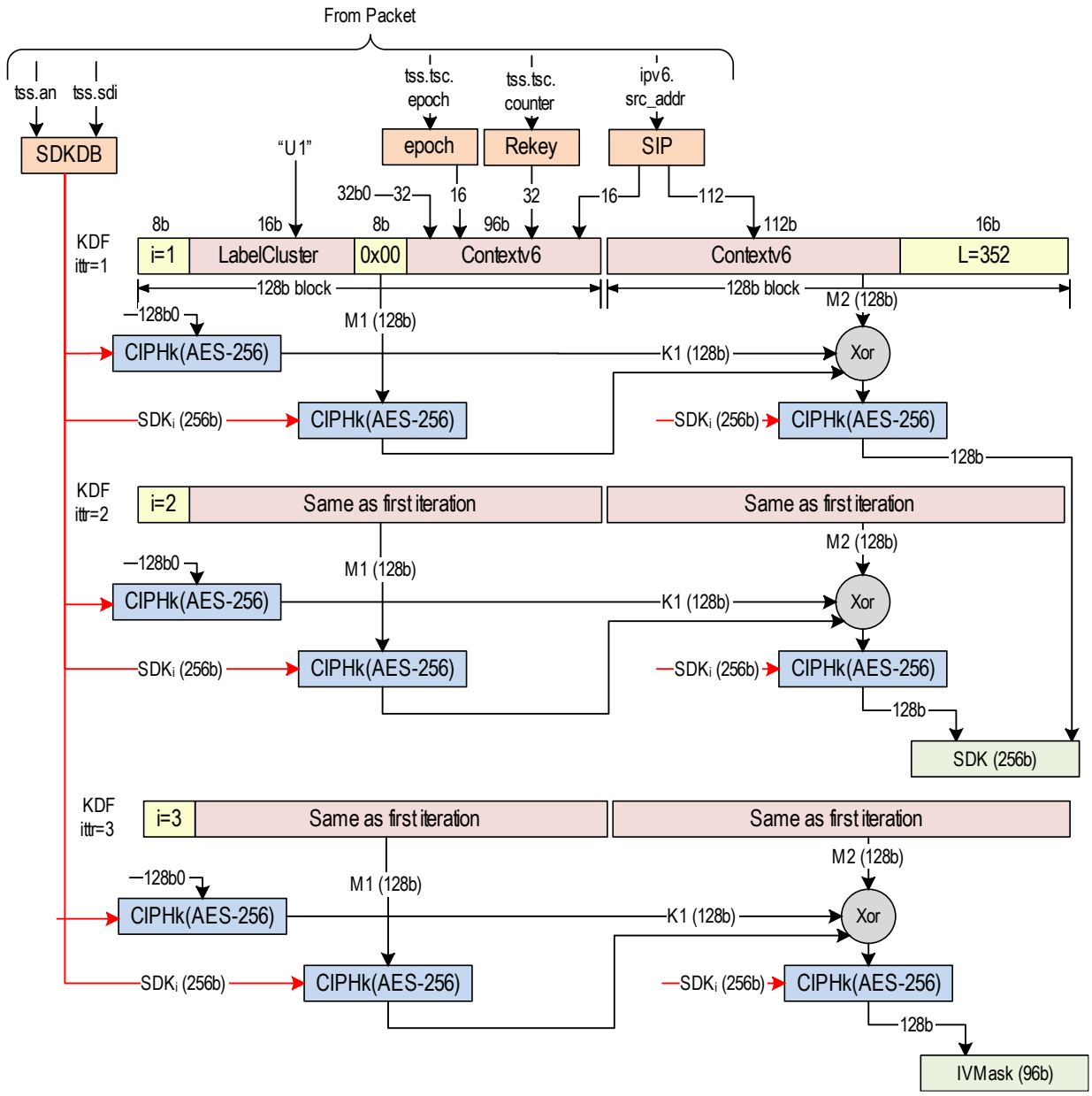


Figure 3-144 - Cluster KDF mode for IPv6 without explicit SSI

3.7.8.3 Server KDF Mode

The server KDF mode is used to assist server key scaling and MUST use the cluster KDF mode construction, except destination information is used in the *Context*. The *Label* is 0x5532 or 'U2' in ASCII and the *Context* is defined as follows:

- *LabelSrv* = 0x5532 or 'U2' in ASCII
- *Contextv4* = *current_epoch*||0x00000000||*DestSSI*:if ssi-mode==TRUE

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

- *Contextv4 = current_epoch*||0x00000000||**ipv4.dest_adddr:**if pkt==IPv4
- *Contextv6 = current_epoch*||0x00000000||**ipv6.dest_adddr:**if pkt==IPv6

The KDF is invoked in the following manner: *IVMask*||*SDKo* = KDF(*SDKi*, context=*Contextv4/6*, label=*LabelSrv*)

3.7.8.4 KDF Examples

The following are examples of the KDF's calculations. The source code to generate these examples is provided in a UEC git repository in the test_uec_kdf.c file <https://github.com/ultraethernet/uet-ref-prov/blob/main/crypto/test/test_uec_kdf.c>. These were verified using vectors from the NIST CAVP program [43] for "SP 800-108 Key Derivation Using Pseudorandom Functions (KBKDF)". The specific vectors are: [PRF=CMAC_AES256], [CTRLOCATION=BEFORE_FIXED], [RLEN=8_BITS] with L=352.

In all cases, the input key is:

- *SDK* = 0x34448a064292601b11a0978f56a2d34cf3fc35ede1a6bc04f8db3e5243a2b0ca
- *Rekey_Mask* = 0x0000FFFF00000000
- *Rekey_Shift* = 32

Create the *Rekey* field by extracting the bits from the counter field from the *TSC*.

- *TSC* from packet = 0x1DC074E500000023
- *Rekey* = 0x000074E5
- *Epoch*= 0x1DC0

For the IPv4 cluster KDF mode cases with *ssi-mode*=FALSE:

- *Label* = 0x5531 or 'U1' in ASCII
- *IPv4 SIP* = 0xC0A82A01 or 192.168.42.1
- *Context4* = 0x1dc0000074e5c0a82a01
- *Kdf_fixed4* = 0x5531001dc0000074e5c0a82a010160
- *SDK* = 0x151b4ddb30112971ddeff3213000ee74d8f18aac2135601f1e5215e505fed449
- *IVMask* = 0xa7c6992c26b0bd5dd5c20e0c

For the IPv6 cluster KDF mode cases with *ssi-mode*=FALSE:

- *Label* = 0x5531 or 'U1' in ASCII
- *IPv6 SIP* = 0x20010cb0000000000fc0000000000abc or
  2001:0cb0:0000:0000:0fc0:0000:0000:0abc
- *Context6* = 0x000000001dc0000074e520010cb0000000000fc0000000000abc
- *Kdf_fixed6* = 0x553100000000001dc0000074e520010cb0000000000fc0000000000abc0160
- *SDK* = 0x55d1ee8647bd53fad0e5325795af18e7559b7d42a895edf70f9c170341e8f767
- *IVMask* = 0x9c96a5752fb5669ccd5ded3f

For IPv4 Client-server KDF mode with *ssi-mode*=FALSE

- *Label* = 0x5532 or 'U2' in ASCII
- *IPv4 SIP* = 0xC0A82A01 or 192.168.42.1
- *Context4* = 0x1dc000000000c0a82a01
- *Kdf_fixed4* = 0x5532001dc000000000c0a82a010160

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

- *SDK* = 0x245e67ab286218530edd53c26ea9ec33c96b35192d0a0eb54d08be281c5d304b
- *IVMask* = 0x02d5ee33684201990e1629e0

For IPv6 Client-server KDF mode with *ssi-mode*=FALSE

- *Label* = 0x5532 or 'U2' in ASCII
- *IPv6 SIP* = 0x20010cb0000000000fc0000000000abc or
  2001:0cb0:0000:0000:0fc0:0000:0000:0abc
- *Context6* = 0x000000001dc00000000020010cb0000000000fc0000000000abc
- *Kdf_fixed6* = 0x553200000000001dc00000000020010cb0000000000fc0000000000abc0160
- *SDK* = 0xb59002ad3e6a9ae5864878730070f8916e43e5011acaa4be3504256185f24d97
- *IVMask* = 0x209a3d29581028fb0da2749e

3.7.9 Replay Protection

It is necessary to detect when a packet has been replayed on the network, through either normal retry mechanisms or a malicious actor observing and playing back a packet. Replay protection will be handled at the layers above this protocol.

Informative Text:
Both ROD and RUD use a dynamic connection protocol. This is similar to 0-RTT in that it is susceptible to replay attack. The UET PDS specification uses a mechanism to keep track of close information to ensure connection-open is not replayed. For existing PDCs the following applies:

- Reliable ordered delivery (ROD): Replay protection is inherent in the in-order protocol (reject old PSNs), and no additional processing is required. Please see PDS specification for connection creation replay protection—specifically, the connection setup replay in Section 3.5.8.2.1.
- Reliable unordered delivery for non-idempotent operations (RUD): Apply-once semantic can be used for replay protection. Please see PDS specification for connection creation replay protection—specifically, the connection setup replay in Section 3.5.8.2.1
- Reliable unordered delivery for idempotent operations (RUDI): Due to the idempotent nature of this transport, replay protection at the transport layer is not possible and NOT provided by TSS. Replay protection MAY be achieved by adding a nonce at the application layer. Another solution is to rely on rekeying to minimize attack window. Finally, RUDI MAY be disabled.
- Unreliable and unorder delivery (UUD): Application solution or epoch-based rejection MAY be used.

3.7.10 Epoch-based packet rejection

A key epoch can be used to reject old frames in the network. The epoch is carried in the *TSC* field in the packet and the SDME controls when the epoch is incremented.

If epoch-based rejection is enabled (*epochBasedRejection*) then packets with older epochs MUST be dropped and the *sdiXXInLatePkts* counter MUST be incremented. The number of older epochs accepted

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

is defined by *rxMaxEpochLifetime*. Parameters *epochBasedRejection* and *rxMaxEpochLifetime* MAY be global parameters OR stored in SDKDB per SDI.

Mathematical operations on wrapping unsigned sequence numbers (integers) can be done by a variety of methods, some of which are described in IETF RFC 1982 [48], which considers the wrap conditions. The intent is to reject epochs lower (older) then the current epoch.

The following pseudo code summarizes the check using a wrap subtract function that implements the wraparound sequence math from IETF RFC 1982 [48]:

```
if( epochBasedRejection == TRUE) &&
( wrapSubtract(SDI[pkt.tss.SDI].current_epoch – pkt.tss.tsc.epoch) >
rxMaxEpochLifetime){
     drop packet and sdiXXInLatePkts[pkt.tss.SDI]++
}
```

3.7.11 TSS over the UFH_32 Header

The primary goal for the Unified Local profile is to reduce the wire overhead and maintain 8-byte alignment to facilitate efficient mapping at the Ethernet layer.

The regular TSS mode is supported when using the UFH_32 header. The regular TSS mode MUST be supported if TSS is supported in the Unified Local profile.TSS Packet Processing

UET defines three types of encapsulations:

- Directly over IP using the IP header to identify the UET protocol.
- UDP encapsulation over IP, where the UDP port number of 4793 is used to identify the UET protocol.
- Directly over UFH_32 where the EtherType 0xE790 is used to identify the UET.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

An overview of the encapsulations over IPv4, IPv6, IP/UDP, and UFH_32 is depicted in Figure 3-145.

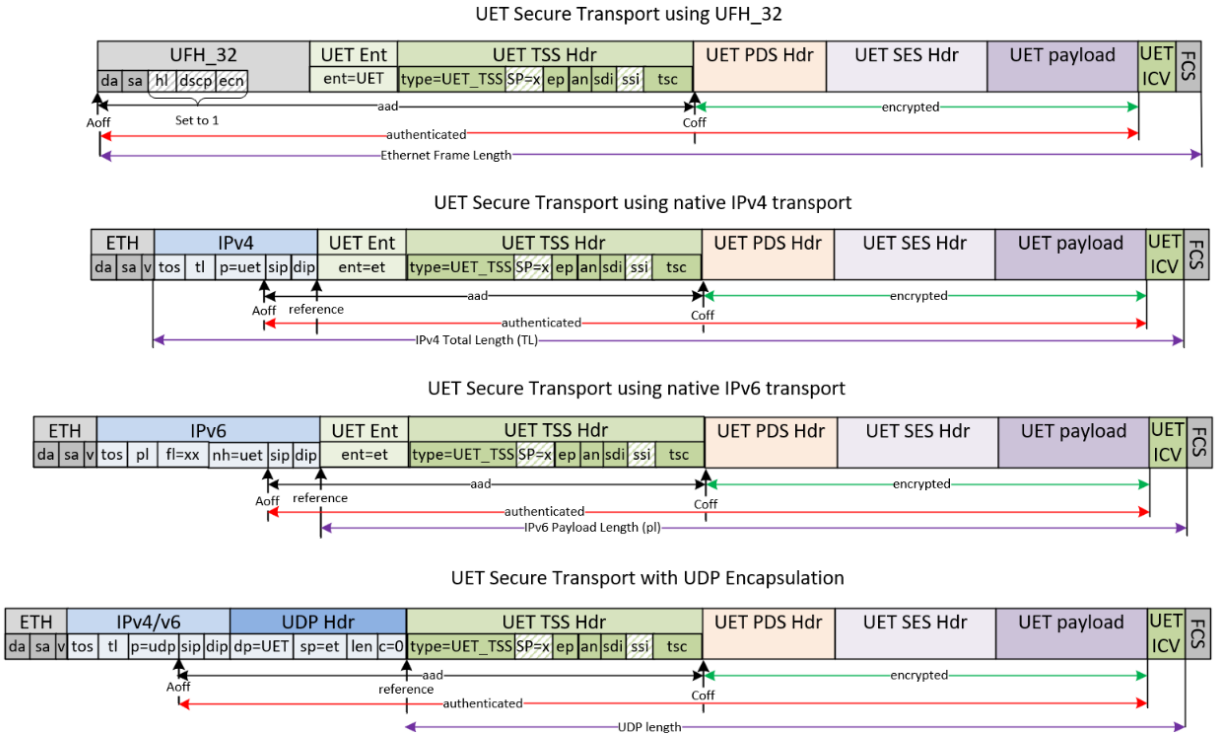


Figure 3-145 - UET Secure Transport Packets

When using native IP encapsulation, the first header of the IP payload is the PDS entropy header, and the next header is the TSS header. When using UDP/IP encapsulation, the first header of the UDP payload is the TSS header. The PDS header implicitly always follows the TSS header (i.e., no next header field is present in the TSS header). The following section describes how these headers are constructed.

The length of the encapsulated payload, L, includes the UET header, UET payload, and UET trailer. The length is included within the **ipv4.total_length**, **ipv6.payload_length** or **udp.length** field depending on how the length field is encoded in the packet encapsulations shown in Figure 3-146, Figure 3-147, and Figure 3-148.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

The header layout for UET over native IPv4 transport is shown in Figure 3-146.

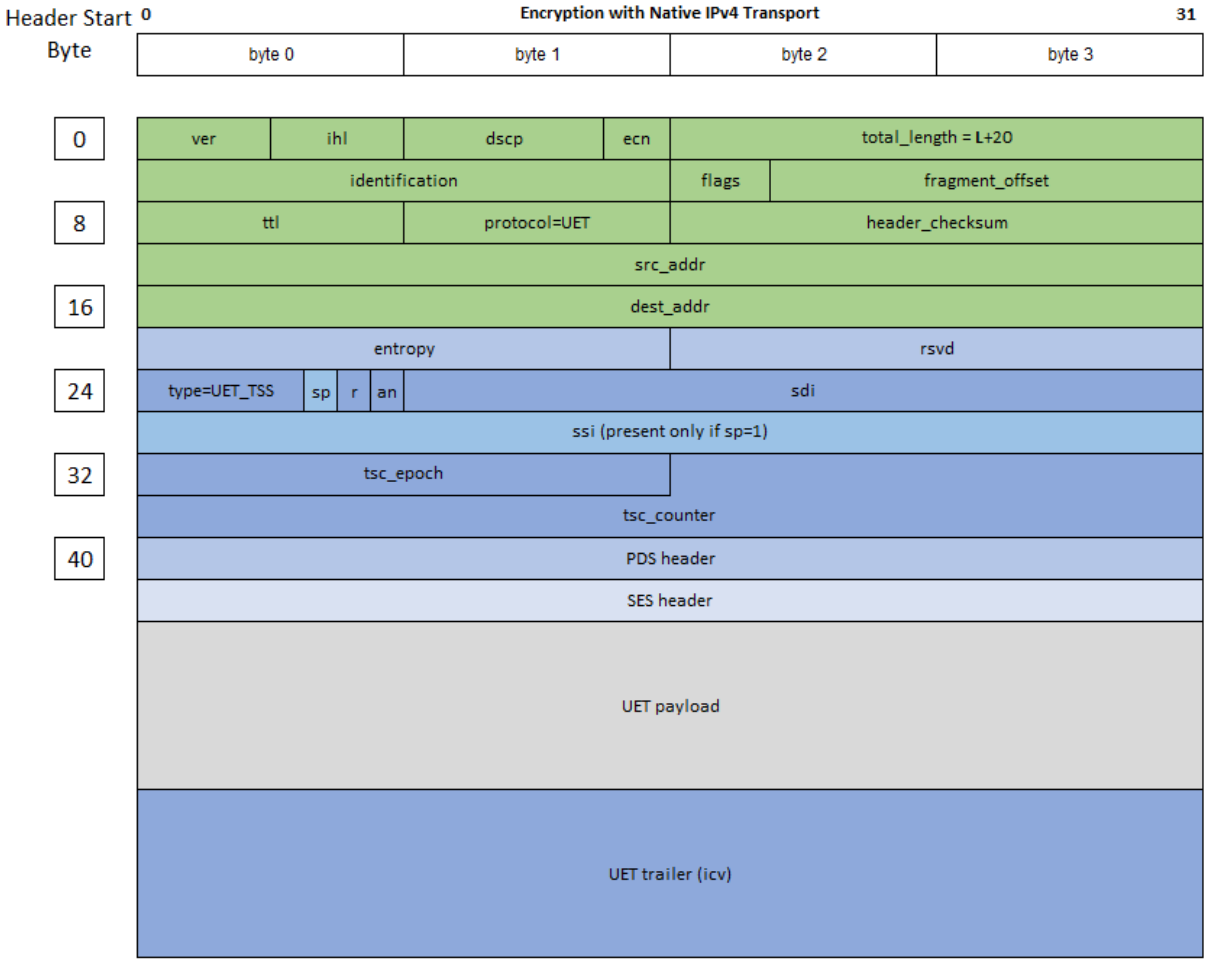


Figure 3-146 - TSS with Native IPv4 Transport

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

The header layout for UET over IPv6 native transport is shown in Figure 3-147.

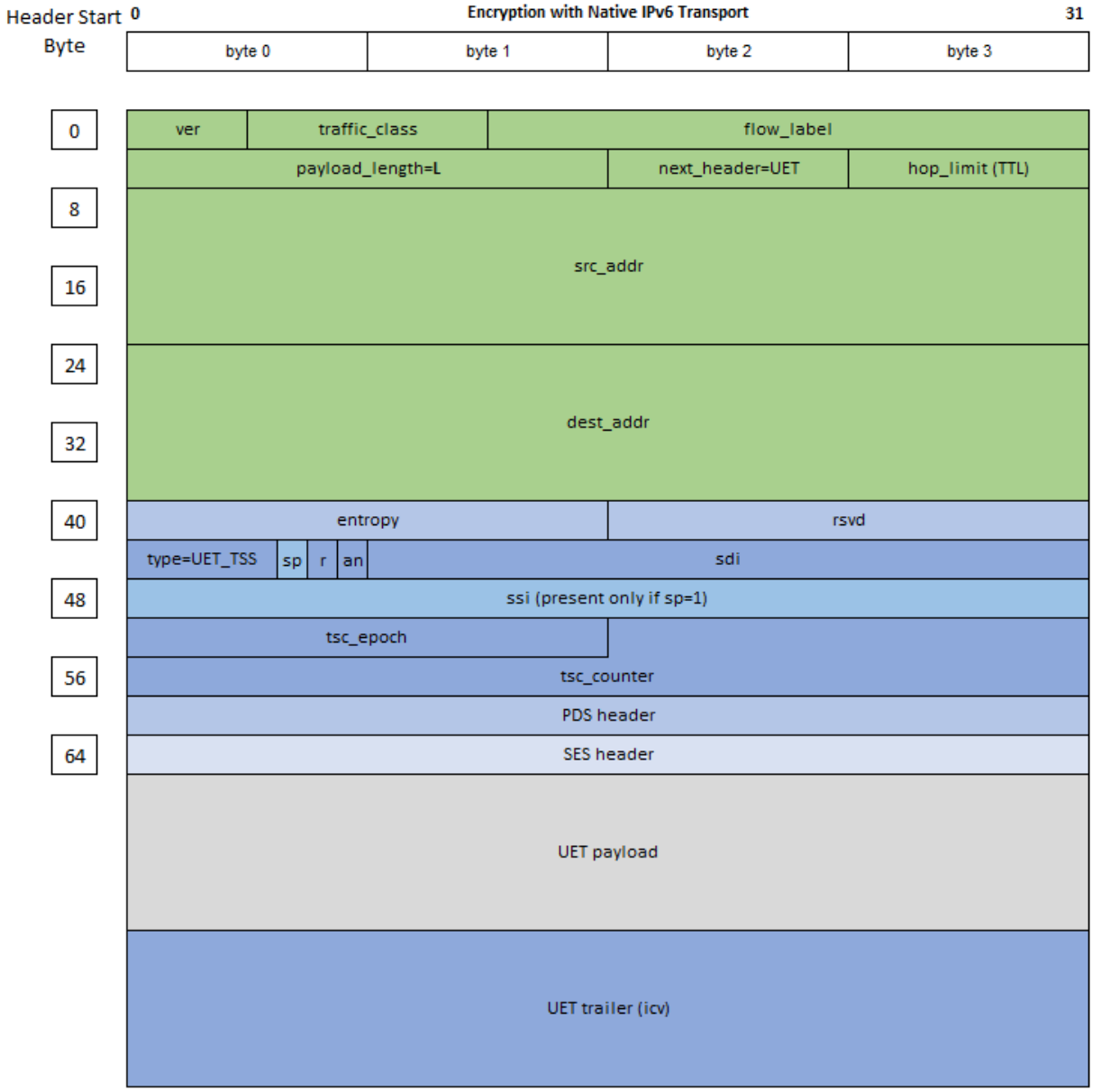


Figure 3-147 - TSS with Native IPv6 Transport

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

For networks that retain UDP encapsulation, destination UDP port 4793 indicates the packet is a UET packet. Figure 3-148 depicts UET over UDP encapsulation.

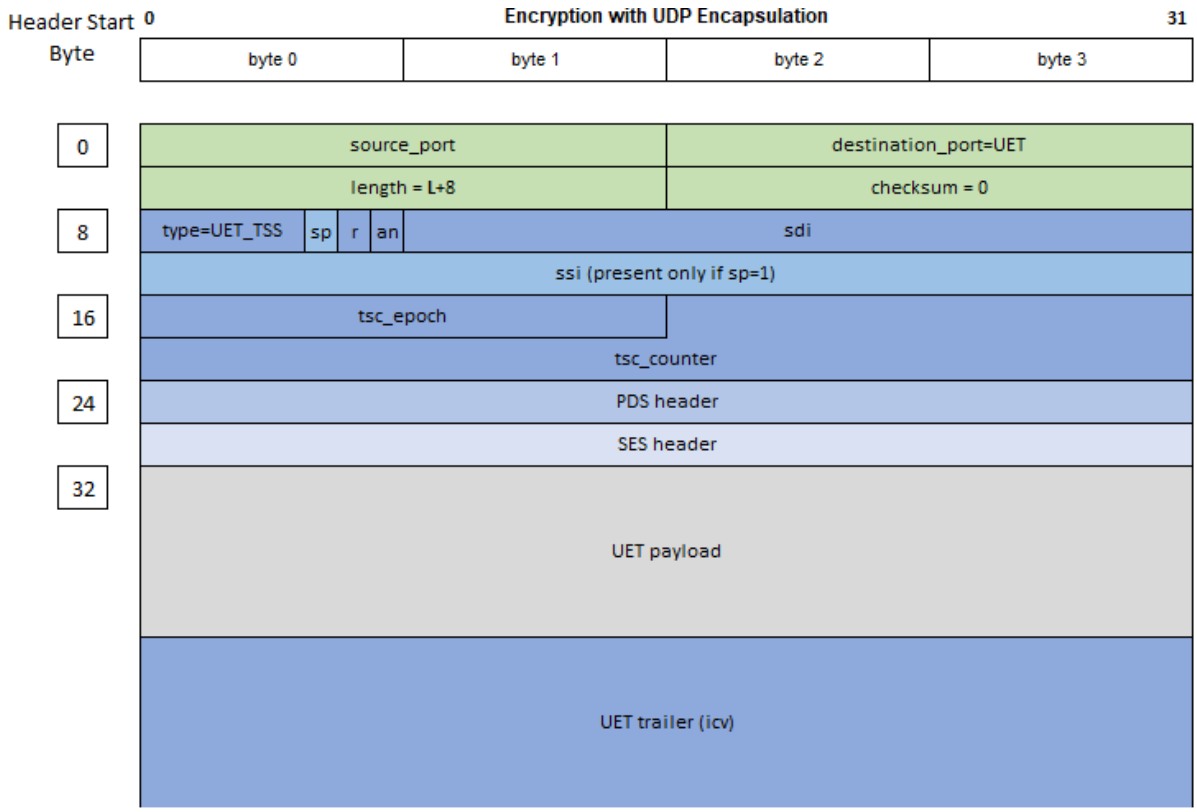


Figure 3-148 - TSS with UDP Encapsulation

For scale-up networks using UFH_32 encapsulation, EtherType 0xE790 indicates the packet is a UET packet. Figure 3-149 depicts encrypted UET over UFH_32 encapsulation.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

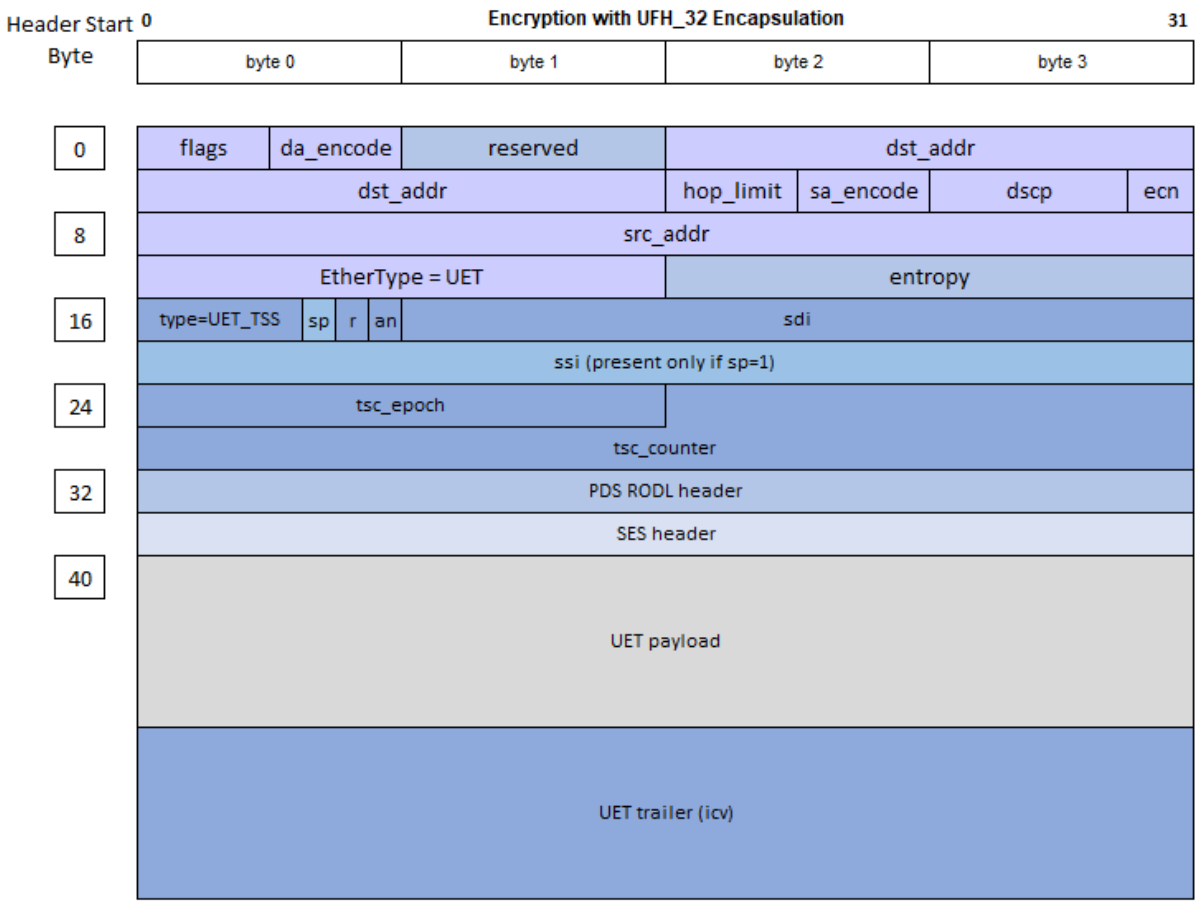


Figure 3-149 - TSS with UFH_32 Encapsulation

Table 3-135 defines the UET TSS header fields and the default field values.

Table 3-135 - TSS Headers Fields

Field	Size (bits)	Note
type	5	Header type, this MUST be set to UET_TSS
sp	1	Set if SSI header present in packet
r	1	reserved
an	1	Association number
sdi	24	Secure Domain Identifier
ssi	32	Source identifier within an SC (optional) based on SP=1
tsc	64	Time-stamp counter (16-bit epoch, 48-bit counter)

UET entropy header, TSS header, PDS header, SES header, UET payload, and ICV MUST fit completely within a maximum payload defined by the transport (Payload MTU). This implies all the listed components MUST be completely contained within a single packet.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

3.7.11.1 Packet Encryption

This section describes the encryption and decryption of packets. A reference for the packet encryption is in the UE reference provider git repo in the crypto directory <https://github.com/ultraethernet/uet-ref-prov/tree/main/crypto>.

3.7.11.1.1 Authentication and Confidentiality Offsets

The extent of the authentication and confidentiality depends on the use case and threat model. The authentication offset (*Aoff*) defines the start of the AAD. The cipher offset (*Coff*) is used to delineate the boundary between authenticated data (AAD) and the ciphertext. *Aoff* and *Coff* are returned from an SDKDB lookup. The AAD is the contiguous set of bytes in the packet referenced by the *Aoff* and *Coff* offsets. *Aoff* and *Coff* are both greater than or equal to 0 and therefore cannot overlap. *Aoff* is an offset from the start of the UET header towards the start of the packet, and *Coff* is an offset from the start of the UET header towards the end of the packet. See Figure 3-145 for more details on the relative offsets. Note the two lengths reflected in the GMAC portion of AES-GCM is the length of the AAD (A) and ciphertext (C). The AAD length is the byte distance between locations in the packet determined by *Aoff* and *Coff*. The ciphertext length is the byte distance between the location in the packet determined by *Coff* and the remainder of the UET payload.

The AAD MUST include all fields used in the security solution. For example, if the **ip.src_addr** field is used in the KDF, then **ip.src_addr** MUST be part of the AAD. The most secure solution SHOULD configure the *Coff* such that only the TSS header is exposed and the rest of the PDS, SES, UET payload is encrypted.

3.7.11.1.2 IV Generation

There are several ways of creating the IV from the *TSC* and fields in the packet. Table 3-136 describes how the IV is generated.

Table 3-136 - IV Construction

IV	Packet Type	ssi-mode	tss.sp	Kdf modes
tss.ssi||**tss.tsc**	IPv4	TRUE	TRUE (0b1)	All
ip.src_addr||**tss.tsc**	IPv4	FALSE	FALSE (0b0)	All
ERROR/Not Valid	IPv4	TRUE	FALSE (0b0)	-
ERROR/Not Valid	IPv4	FALSE	TRUE (0b1)	-
tss.ssi||**tss.tsc**	IPv6	TRUE	TRUE (0b1)	All
ERROR/Not Valid	IPv6	TRUE	FALSE (0b0)	-
ERROR/Not Valid	IPv6	FALSE	TRUE (0b1)	-
tss.sdi||**tss.tsc**	IPv6	FALSE	FALSE (0b0)	Direct KDF mode not supported

Note:

- The **tss.tsc** field referenced in this table is the concatenation of the **tss.tsc.epoch** and **tss.tsc.counter** fields shown in the TSS header diagrams

TSC generation is specified in Section 3.7.5.6. Once the IV is created, it is then XORed with the *IVMask* from the SDKDB or KDF before it is used by cipher engine. The error cases in Table 3-136 are used to catch misconfiguration between the *ssi-mode* in the SDKDB and the packet. The TSS implementation MUST detect the IV error cases in Table 3-136 and drop the packet and increment the *sdiXXInInvalid*.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

In direct KDF mode the *IVMask* is chosen from *IVMask0* or *IVMask1* of the SDKDB based upon the enabled AN.

3.7.11.1.3 ICV Calculation

The integrity check value (ICV) MUST be calculated over the AAD and ciphertext. The ICV MUST be appended to the end of the packet. For AES-GCM the ICV length MUST be 16 bytes. Packets that fail authentication MUST be dropped and the *sdiXXInAuthFailPkts* counter MUST be incremented. Trimmed packets will fail authentication, but MAY be used as a congestion signal per the limitations described in Section 3.7.2.4.2.

When calculating the ICV for UDP packets the **udp.checksum** field is set to zero before the ICV calculation is made.

3.7.11.2 Packet Transmission

- Packets are received from the PDS with security and link information.
- Packets that bypass the TSS are assumed to be UE packets that are not using TSS services and MUST be passed to the link unmodified. The *outPPRxEncryptionBypassPkts* packet counter MUST be incremented.
- SDKDB information is retrieved using the SDI.
    - If the SDKDB is not valid the packet is dropped and the *ifPPOutErroredPkts* counter is incremented.
- Direct KDF mode.
    - The *SDK* from SDKDB is used directly to encrypt packet.
- Client-server KDF mode.
    - *DestSSI* or destination IP addresses is used in the KDF to construct the *SDK* using client-server KDF mode.
    - The epoch from *SDKDB.current_epoch* is used in KDF.
- Cluster KDF mode.
    - *SrcSSI* or **ip.src_addr** is used in KDF to construct the *SDK* based on SDKDB *ssi-mode* field.
    - *Rekey* is calculated and used in KDF (see Section 3.7.5.2).
    - The epoch from *SDKDB.current_epoch* is used in KDF.
- UET TSS header is added.
    - The **tss.type** field is set to UET_TSS.
    - **tss.sdi** and **tss.an** are set to *SDKDB.SDI* and *SDKDB.TxActiveAN*.
    - The **tss.ssi** field is populated based on the *SDKDB.ssi-mode*. If the *ssi-mode* is TRUE then the **tss.ssi** field is populated in the packet with *SrcSSI* and the **tss.sp** field is set to 0b1. Otherwise, the **tss.ssi** field is not populated, and the **tss.sp** field is set to 0.
    - *TSC* is generated.
        - The **tss.tsc.epoch** portion is copied from *SDKDB.current_epoch*.
        - If the **tss.tsc.counter** wrapped or is greater than *invokeFatalTheshold* the packet MUST be dropped and the *sdiXXOutInvokeFail* counter is incremented.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

        - If the **tss.tsc.counter** is greater than *invokeWarnThreshold*, then the SDME SHOULD be informed.
- The packet is encrypted and ICV appended to frame according to the process in Section 3.7.11.1.
    - The IV is determined according to construction rules in Table 3-136.
    - The *Aoff*, *Coff*, and *SDK* fields are used in encryption.
- The SDKDB statistics are updated as follows:
    - *sdiXXOutAuthPkts* is incremented when the packet is successfully encrypted and sent.
- For any condition where the packet cannot be sent (e.g., invalid SDI, invalid SDKDB parameters), the secure transport MUST ensure that unencrypted plaintext is not sent on the wire, that the packet is dropped, and the *ifPPOutErroredPkts* counter is incremented.

3.7.11.3 Packet Reception

- The packet is received from link layer with a valid FCS and with any additional link information.
- The UET TSS header is processed as follows:
    - If the **tss.type** field is not UET_TSS, then the packet bypassed encryption and MUST be sent to PDS without modification. The *rx_auth* flag MUST NOT be set and the *inPPRxEncryptionBypassPkts* counter MUST be incremented.
    - If the **tss.type** field is UET_TSS then packet is processed by TSS before being sent to PDS.
    - The **tss.sp** field indicates if SSI is present.
    - The **tss.an** and **tss.sdi** fields are extracted from packet to form the security association.
- An SDKDB look-up is performed using **tss.an** and **tss.sdi** from TSS header.
    - If an SDKDB entry is not found, then the packet MUST be dropped and the *ifPPInErroredPkts* counter MUST be incremented.
    - *RxActiveAN* is retrieved from the SDKDB.
    - Validation of the **tss.an** field against *RxActiveAN* is performed as follows:
        - If *RxActiveAN* = 0b00 then the packet MUST be dropped.
        - If *RxActiveAN* = 0b01 and **tss.an** = 0 then the packet MUST be dropped.
        - If *RxActiveAN* = 0b10 and **tss.an** = 1 then the packet MUST be dropped.
        - If *RxActiveAN* = 0b11 the then the packet is allowed to continue.

      If the packet is dropped, then the *sdiXXInInvalidSa* counter MUST be incremented and a lastAN error indication MUST be sent to PDS.

    - Once **tss.an** has been processed, the *SDKi* is selected from the SDKDB (*SDKiAN0* or *SDKiAN1*).
    - If *SDKDB.encap-type* does not match the encapsulation of the receive packet or if *Aoff* or *Coff* are invalid, the packet MUST be dropped and the *sdiXXInInvalid* counter MUST be incremented.
    - Any other SDKDB errors (invalid key, etc.) detected MUST cause the packet to be dropped and the *sdiXXInInvalid* counter MUST be incremented.
    - If the SSI mode in the packet (**tss.sp**) does not match the SDKDB *ssi-mode* field, then the packet is dropped and *sdiXXInInvalid* counter incremented.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

- If *kdf-mode* is Direct-KDF.
    - The selected *SDKi* is used directly in the cipher operation for the packet.
- If *kdf-mode* is KDF cluster/client-server.
    - The selected *SDKi* is used in the KDF operation for the packet.
    - **tss.ssi** or **ip.src_addr** is used in KDF to construct *SDK* based on *SDKDB.ssi-mode* field.
    - *Rekey* is calculated based on **tss.tsc.counter** and used in the KDF (see Section 3.7.5.2).
    - The epoch from **tss.tsc.epoch** is used in the KDF.
- Epoch-based rejection.
    - *SDKDB.current_epoch* is compared to **tss.tsc.epoch** according to rules in Section 3.7.10. If the check fails, the packet MUST be dropped and the *sdiXXInLatePkts* counter MUST be incremented.
- Packet Decryption.
    - The IV is created per Table 3-136 and XORed with the *IVMask* from the SDKDB or from the output of the KDF before it is used in the cipher.
    - The AAD and ciphertext are constructed based on *Aoff* and *Coff* from the SDKDB per Section 3.7.11.1.1.
    - The packet is decrypted according to the process in Section 3.7.11.1.
    - The ICV from the UET trailer is used to authenticate packet according to the process in Section 3.7.11.1.3.
        - Packets that fail authentication MUST be dropped and the *sdiXXInAuthFailPkts* counter MUST be incremented.
        - If the *sdiXXInAuthFailPkts* > *authFailThreshold* then the authentication failure threshold had been exceeded, and the packet MUST be dropped and the SDME MUST be notified. Note that the *sdiXXInAuthFailPkts* counters increments for all enabled AN (*RxActiveAN*). An implementation MUST reset this counter after a key rotation is complete.
        - Packets that pass authentication MUST be passed to PDS with *rx_auth* set to TRUE and the *sdiXXInAuthPkts* counter MUST be incremented.
        - If an encrypted trimmed packet is received it cannot be authenticated, but the beginning of the packet can be decrypted. An implementation MAY chose packet drop the packet or pass it to the PDS with the *rx_auth* set to FALSE. The reception of a trimmed packet MUST NOT increment the *sdiXXInAuthFailPkts* counter.
- The SDKDB statistics are updated as follows:
    - The *sdiXXInAuthPkts* counter is updated for frames that are properly authenticated (described in the Packet Decryption list item above).
    - Packets bypassing the receive crypto process (e.g., **pds.type**=UET_PDS) MUST cause the *inPPRxEncryptionBypassPkts* counter to be incremented, and when passed to PDS the *rx_auth* flag MUST be set to FALSE.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

3.7.12 Statistics, Parameters, and Events

Table 3-137 describes the counters and statistics that MUST be supported. Counters MUST initialize to zero and wrap when their bit width is exceeded. These counters MUST be provided to the management entity within the FEP. These are modeled after the Ethernet MIB counters [44]. All counters MUST be 64 bits when presented to the SDME.

Table 3-137 - TSS Security Counters

Counter	Description	Units	Scope	Imp
sdiXXInAuthPkts	Number of successfully received, decrypted, and authenticated (pass authentication) packets.	packets	SDI	M
sdiXXInAuthFailPkts	Number of received packets that fail authentication and are not trimmed.	packets	SDI	M
sdiXXInInvalid	Number of received packets with errors.	packets	SDI	M
sdiXXInInvalidSa	Number of received packets dropped due to an inactive or invalid security association.	packets	SDI	M
sdiXXInLatePkts	Received packets dropped/rejected based on epoch-based rejection.	packets	SDI	O
sdiXXOutInvokeFail	The number of transmit packets dropped because invoke limit is reached.	packets	SDI	M
sdiXXOutAuthPkts	Number of authentication and encrypted packets transmitted. This is also the number of invocations of the *SDK* for the *TxActiveAN*.	packets	SDI	M
ifPPOutErroredPkts	Packets with invalid information from PDS and cannot be mapped to a secure domain. These packets MUST be dropped and not sent on wire.	packets	PORT	M
ifPPInErroredPkts	Packets received with invalid security information and cannot be mapped to secure domain. These packets have a UET TSS header.	packets	PORT	M
inPPRxEncryptionBypassPkts	Packets that are received with a UET PDS header where the type is not UET_TSS. These packets MUST bypass encryption and authentication and be presented to PDS.	packets	PORT	M
outPPRxEncryptionBypassPkts	UET PDS packets sent that bypass encryption and authentication. These packets are passed to the link layer unmodified.	packets	PORT	M
sdiXXInBindingFailurePkts	Received packets that fail the SES binding check.	packets	SDI	M

Note:

- XX is the SDI number and PP is the port number.
- M=mandatory, O=Optional if feature is not supported but mandatory if it is implemented.

Table 3-138 describes various parameters used in this specification.

Table 3-138 - TSS Security Parameters

Parameter	Description	Scope	Imp
invokeWarnThreshold	Invoke warning limit	SDI/Global	M
invokeFatalTheshold	Invoke fatal limit	SDI/Global	M

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Parameter	Description	Scope	Imp
authFailThreshold	When *sdiXXInAuthFailPkts* is greater than this value, packets for the SDI are dropped.	SDI/Global	M
ssi-mode	A boolean value that indicates how to identify the SSI. A value of TRUE means use the SSI. A value of FALSE means use the source IP address.	SDI/Global	O
rxMaxEpochLifetime	Max lifetime for epoch-based rejection	SDI/Global	O
epochBasedRejection	Enable for epoch-based rejection (dropping)	SDI/Global	O
Rekey_Mask	Rekey mask used to generate *Rekey*	SDI/Global	O
Rekey_Shift	Rekey shift used to generate *Rekey*	SDI/Global	O
maxSDI	Maximum number of supported SDIs	Global	M
baseSDI	Base SDI number, provisioned by SDME	Global	M
kdfModes	Supported KDF modes • 0bxx1: Direct • 0bx1x: Cluster • 0b1xx: Client-server	Global/Mask	M
maxCoff	Maximum *Coff* configuration	SDI/Global	M
maxAoff	Maximum *Aoff* configuration	SDI/Global	M
Note: • M=mandatory, O=Optional			

Table 3-139 describes the events or errors the TSS layer can generate.

Table 3-139 - TSS Security Events/Errors

Event	Description	Scope	Imp
invokeWarn	Invoke warning limit	SDI	M
invokeFatal	Invoke fatal limit	SDI	M
authFail	When *sdiXXInAuthFailPkts* is greater than this value, packets for the SDI are dropped	SDI	M
Note: • M=mandatory, O=Optional			

3.8 References

[1] T. Hoefler, D. Roweth, K. Underwood, B. Alverson, M. Griswold, V. Tabatabaee, M. Kalkunte, S. Anubolu, S. Shen and A. Kabbani, "Datacenter ethernet and rdma: Issues at hyperscale," arXiv preprint arXiv:2302.03337, 2023.

[2] IETF RFC 6040, "Tunnelling of Explicit Congestion Notification," 2010. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc6040.

[3] B. Rothenberger, K. Taranov, A. Perrig and T. Hoefler, "{ReDMArk}: Bypassing {RDMA} security mechanisms," in 30th USENIX Security Symposium (USENIX Security 21), 2021.

[4] IETF RFC 3692, "Assigning Experimental and Testing Numbers Considered Useful," 2004. [Online]. Available: https://www.rfc-editor.org/rfc/rfc3692.html.

[5] R. Brightwell, W. W. Schonbein, K. Pedretti, K. S. Hemmert, A. B. Maccabe, R. E. Grant, B. W. Barrett, K. Underwood, R. Riesen and T. Hoefler, "The Portals 4.3 Network Programming Interface," 2022.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

[6] P. Grun, S. Hefty, S. Sur, D. Goodell, R. D. Russell, H. Pritchard and J. M. Squyres, "A Brief Introduction to the OpenFabrics Interfaces - A New Network API for Maximizing High Performance Application Efficiency," in 2015 IEEE 23rd Annual Symposium on High-Performance Interconnects, 2015.

[7] Z. Liang, J. Lombardi, M. Chaarawi and M. Hennecke, "DAOS: A Scale-Out High Performance Storage Stack for Storage Class Memory," in Supercomputing Frontiers, 2020.

[8] R. Brightwell and K. Underwood, "Evaluation of an eager protocol optimization for MPI," in European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting, 2003.

[9] T. Bonato, A. Kabbani, D. De Sensi, R. Pan, Y. Le, C. Raiciu, M. Handley, T. Schneider, N. Blach, A. Ghalayini, D. Alves, M. Papamichael, A. Caulfield and T. Hoefler, "SMaRTT-REPS: Sender-based Marked Rapidly-adapting Trimmed & Timed Transport with Recycled Entropies," 2 April 2024. [Online]. Available: https://arxiv.org/abs/2404.01630v1.

[10] Y. Le, R. Pan, P. Newman, J. Blendin, A. Kabbani, V. Jain, R. Sivaramu and F. Matus, "STrack: A Reliable Multipath Transport for AI/ML Clusters," 21 July 2024. [Online]. Available: https://arxiv.org/abs/2407.15266.

[11] V. Olteanu, H. Eran, D. Dumitrescu, A. Popa, C. Baciu, M. Silberstein, G. Nikolaidis, M. Handley and C. Raiciu, "An edge-queued datagram service for all datacenter traffic," in 19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22), Renton, WA, 2022.

[12] T. Bonato, A. Kabbani, A. Ghalayini, M. Papamichael, M. Dohadwala, L. Gianinazzi, M. Khalilov, E. Achermann, D. De Sensi and T. Hoefler, "REPS: Recycled Entropy Packet Spraying for Adaptive Load Balancing and Failure Mitigation," 31 July 2024. [Online]. Available: https://arxiv.org/abs/2407.21625.

[13] Zats, A. Iyer, G. Ananthanarayanan, R. Katz, I. Stoica and A. Vahdat, "FastLane: Agile Drop Notification for Datacenter Networks," UC Berkeley Tech Report, UCB/EECS-2013-173, 2013.

[14] IETF RFC 3168, "The Addition of Explicit Congestion Notification (ECN) to IP," 2001. [Online]. Available: https://www.rfc-editor.org/rfc/rfc3168.html.

[15] IEEE Std 802.1Q-2022, "IEEE Standard for Local and Metropolitan Area Networks – Bridges and Bridged Networks," 2022. [Online]. Available: https://ieeexplore.ieee.org/document/10004498.

[16] C. Guo, H. Wu, Z. Deng, G. Soni, J. Ye, J. Padhye and M. Lipshteyn, "RDMA over commodity ethernet at scale," in Proceedings of the 2016 ACM SIGCOMM Conference, 2016.

[17] IETF RFC 8257, "Data Center TCP (DCTCP): TCP Congestion Control for Data Centers," 2017. [Online]. Available: https://www.rfc-editor.org/rfc/rfc8257.

[18] Y. Zhu, H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. H. Yahia and M. Zhang, "Congestion control for large-scale RDMA deployments," ACM SIGCOMM Computer Communication Review, pp. 523-536, 2015.

[19] G. Kumar, N. Dukkipati, K. Jang, H. M. Wassel, X. Wu, B. Montazeri, Y. Wang, K. Springborn, C. Alfeld, M. Ryan, D. Wetherall and A. Vahdat, "Swift: Delay is simple and effective for congestion control in the datacenter," in SIGCOMM '20: Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication, 2020.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

[20] B. Montazeri, Y. Li, M. Alizadeh and J. Ousterhout, "Homa: A receiver-driven low-latency transport protocol using network priorities," in Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, 2018.

[21] P. X. Gao, A. Narayan, G. Kumar, R. Agarwal, S. Ratnasamy and S. Shenker, "pHost: Distributed near-optimal datacenter transport over commodity network fabric," in Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies, 2015.

[22] G. Chen, Y. Lu, B. Li, K. Tan, Y. Xiong, P. Cheng, J. Zhang and T. Moscibroda, "MP-RDMA: Enabling RDMA With Multi-Path Transport in Datacenters," IEEE/ACM Transactions on Networking, vol. 27, no. 6, pp. 2308-2323, 2019.

[23] IETF RFC 8684, "TCP Extensions for Multipath Operation with Multiple Addresses," 2020. [Online]. Available: https://www.rfc-editor.org/rfc/rfc8684.html.

[24] A. Singh, Load-balanced routing in interconnection networks, Stanford University, 2005

[25] J. Kim, W. J. Dally, S. Scott and D. Abts, "Technology-Driven, Highly-Scalable Dragonfly Topology," ACM SIGARCH Computer Architecture News, vol. 36, no. 3, pp. 77-88, 2008.

[26] P. Geoffray and T. Hoefler, "Adaptive Routing Strategies for Modern High Performance Networks," in 2008 16th IEEE Symposium on High Performance Interconnects, 2008.

[27] De Sensi, S. Di Girolamo, K. H. McMahon, D. Roweth and T. Hoefler, "An In-Depth Analysis of the Slingshot Interconnect," in SC20: International Conference for High Performance Computing, Networking, Storage and Analysis, 2020.

[28] A. Dixit, P. Prakash, Y. C. Hu and R. R. Kompella, "On the impact of packet spraying in data center networks," in 2013 Proceedings IEEE INFOCOM, 2013.

[29] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik and M. Handley, "Improving datacenter performance and robustness with multipath TCP," in Proceedings of the ACM SIGCOMM 2011 Conference, Toronto, Ontario, Canada, 2011.

[30] P. Cheng, F. Ren, R. Shu and C. Lin, "Catch the whole lot in an action: Rapid precise packet loss notification in data center," in 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14), 2014.

[31] M. Handley, C. Raiciu, A. Agache, A. Voinescu, A. W. Moore, G. Antichi and M. Wójcik, "Re-architecting datacenter networks and stacks for low latency and high performance," in Proceedings of the Conference of the ACM Special Interest Group on Data Communication, 2017.

[32] HPCS, "HPC Challenge," [Online]. Available: https://hpcchallenge.org/hpcc/. [Accessed 08 05 2025].

[33] K. Taranov, B. Rothenberger, A. Perrig and T. Hoefler, "{sRDMA}--Efficient {NIC-based} Authentication and Encryption for Remote Direct Memory Access," in 2020 USENIX Annual Technical Conference (USENIX ATC 20), 2020.

[34] GitHub, "google/psp psp-open-source project," 2021. [Online]. Available: https://github.com/google/psp.

[35] A. Luykx and K. G. Paterson, "Limits on authenticated encryption use in TLS," Cryptology ePrint Archive, 2024.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

[36] NIST SP 800-38D, "Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC," 2007. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38d.pdf.

[37] NSA CNSA Suite 2.0, "CNSA Suite 2.0 and Quantum Computing FAQ," 2024. [Online]. Available: https://media.defense.gov/2022/Sep/07/2003071836/-1/-1/0/CSI_CNSA_2.0_FAQ_.PDF.

[38] NIST SP 800-131A Rev. 2, "Transitioning the Use of Cryptographic Algorithms and Key Lengths," March 2019. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-131Ar2.pdf.

[39] A. Bozhko, "draft-irtf-cfrg-aead-properties-09 - Properties of AEAD Algorithms," 11 10 2024. [Online]. Available: https://www.ietf.org/archive/id/draft-irtf-cfrg-aead-properties-09.html.

[40] NIST SP 800-108r1 (Revision 1, Update 1), "Recommendation for Key Derivation Using Pseudorandom Functions," 2024. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-108r1-upd1.pdf.

[41] NIST FIPS-197, "Advanced Encryption Standard (AES)," 2001. [Online]. Available: https://csrc.nist.gov/files/pubs/fips/197/final/docs/fips-197.pdf.

[42] NIST SP 800-38B, "Recommendation for Block Cipher Modes of Operation: the CMAC Mode for Authentication," 2016. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-38B.pdf.

[43] NIST CAVP, "Cryptographic Algorithm Validation Program," NIST Computer Security Resource Center, 2025. [Online]. Available: https://csrc.nist.gov/Projects/Cryptographic-Algorithm-Validation-Program/Key-Derivation.

[44] IETF RFC 3635, "Definitions of Managed Objects for the Ethernet-like Interface Types," 2003. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc3635.

[45] IETF RFC 9001, "Using TLS to Secure QUIC," May 2021. [Online]. Available: https://www.rfc-editor.org/rfc/rfc9001.html.

[46] V. T. Hoang, S. Tessaro and A. Thiruvengadam, "The multi-user security of GCM, revisited: Tight bounds for nonce randomization," in Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, 2018.

[47] IETF RFC 8645, "Re-keying Mechanisms for Symmetric Keys," August 2019. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc8645.

[48] IETF RFC 1982, "Serial Number Arithmetic," August 1996. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc1982.

[49] M. Bellare, R. Ng and B. Tackmann, "Nonces Are Noticed: AEAD Revisited," in Advances in Cryptology – CRYPTO 2019, Springer, 2019, pp. 235-265.

[50] P. Rogaway and T. Shrimpton, "A Provable-Security Treatment of the Key-Wrap Problem," in Advances in Cryptology - EUROCRYPT 200, Berlin, Heidelberg, 2006.

[51] T. Shrimpton, "A Characterization of Authenticated-Encryption as a Form of Chosen-Ciphertext Security," 2004. [Online]. Available: https://eprint.iacr.org/2004/272.

[52] IETF RFC 3385, “Internet Protocol Small Computer System Interface (iSCSI) Cyclic Redundancy Check (CRC)/Checksum Considerations,” September 2002. [Online]. Available: https://datatracker.ietf.org/doc/rfc3385/.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

[53] Github, "uet-pcm / uet_pcm.h," 2026. [Online]. Available: https://github.com/ultraethernet/uet-pcm/blob/main/uet_pcm.h. [Accessed 2026].
[54] IETF RFC 9000, "QUIC: A UDP-Based Multiplexed and Secure Transport," 2021. [Online]. Available: https://www.ietf.org/rfc/rfc9000.html.
[55] T. Chen, A. Kubicek, L. Huang and T. Hoefler, "{CrossPipe}: Towards Optimal Pipeline Schedules for {Cross-Datacenter} Training.," in 2025 USENIX Annual Technical Conference (USENIX ATC 25), 2025.
[56] T. Bonato, S. Abdous, A. Kabbani, A. Ghalayini, N. Gebara, T. Lam, A. Agarwal, T. Chen, Z. Yu and K. Taranov, "Uno: A One-Stop Solution for Inter-and Intra-Data Center Congestion Control and Reliable Connectivity," in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2025.
[57] Y. Li, R. Miao, H. H. Liu, Y. Zhuang, F. Feng, L. Tang, Z. Cao, M. Zhang, F. Kelly and M. Alizadeh, "HPCC: High precision congestion control," in Proceedings of the ACM special interest group on data communication, 2019.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

4 UE Network Layer

4.1 Packet Trimming

Network switches store packets in buffers prior to forwarding them on busy egress ports. Available buffer memory in modern switches is scarce as it is typically implemented as SRAM and is limited by chip area. If the buffer is unable to accommodate an arriving packet, a switch either drops the packet or signals the upstream port to pause the flow of traffic. Both solutions are known to have performance issues.

This specification defines a mechanism that responds to buffer shortage in switches: In short, switches trim the packet payload and use a different traffic class to forward the resulting header to the destination.

This specification focuses only on sending the trimmed packet to the destination. Sending a trimmed packet back to the source, as proposed in [2] , [4] is not part of this specification.

Informative text: Motivation

In modern data centers, switch buffers are under pressure for a few reasons. Chip bandwidth is doubling every 18 months, but chip cost/area and power consumption are not scaling equivalently, constraining the total amount of buffer available. Certain datacenter traffic patterns, on the other hand, are synchronized and often involve line-rate bursts from multiple senders to the same destination (a pathology called incast), placing pressure on limited switch buffers.

Today, switches commonly operate in two modes: best effort and lossless.

In best-effort mode, when a packet arrives and it does not fit in the available buffer on the switch, the packet is dropped. This approach simplifies the handling of switch buffer overload scenarios but poses challenges for transport protocols, which must detect lost packets and handle their retransmission.

In single-path transport protocols, this typically involves setting a threshold for out-of-order received packets—such as TCP's three duplicate ACKs—to determine if a packet has been lost. This works if the dropped packet does not happen at the tail of a transmission; in such cases probes may be needed to avoid expensive retransmit timeouts.

For a packet-spraying transport such as UET, detecting packet loss using an out-of-order degree threshold is more challenging. Packets may be buffered on congested paths and arrive significantly later, necessitating a large out-of-order degree threshold to prevent spurious retransmissions. For packet-spraying transports, packet loss detection can still use probes, sent on each possible path, but this comes at a higher cost.

In lossless mode, when a packet arrives and the receive buffer utilization exceeds a threshold, the switch sends a PFC message on the ingress port where the packet was received, causing the upstream switch to pause further transmission until the receive buffer has sufficient space to store packets.

Lossless mode has the advantage of reducing packet loss to a minimum (loss will generally occur only due to bit errors uncorrected by Ethernet FEC or when the PFC headroom fills up), but it brings its own problems: congestion spreading, tree saturation, and a risk of deadlocks (e.g. see [1] for a description of some of the issues encountered in a production network).

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Packet trimming is useful functionality that can improve packet loss detection. Packet trimming MUST only be performed on valid IP packets that fail buffer admission checks. Conceptually, packet trimming normally occurs at the point of buffer admission, after switch ingress processing has validated the L2 frame and the L3 header, and the switch next hop lookup has found a valid egress port for the packet. However, this depends on switch-architecture.

Packet trimming helps transports that accept trimmed packets and use the information in these packets to enable fast retransmission of the original packets.

Packet trimming uses at least two categories of differentiated service codepoints (DSCPs) that are configured by the network operator: The first category is for packets that are eligible for trimming (called TRIMMABLE). and the second is for packets that have been trimmed (TRIMMED). Only packets belonging to the TRIMMABLE category SHALL be trimmed. Each category of DSCPs MUST include at least one value. Example DSCP values from the trimmable category are {1,2,3} and from the trimmed category are {4,5,6}; however, this spec does not place any constraints on the values of the DSCPs used.

For each network device (generically called switch) that supports trimming, the network operator MUST associate (i.e., configure a mapping) a DSCP_TRIMMED codepoint with each DSCP_TRIMMABLE codepoint. For instance, in the example above, we could have the following mappings: {1->4; 2->5; 3->6}. Multiple trimmable codepoints can be mapped to the same trimmed codepoint, if necessary.

The first step of trimming involves reducing the IP payload length to a length larger than or equal to the MIN_TRIM_SIZE parameter. MIN_TRIM_SIZE is a lower bound, and it depends on both the transport protocols used in the network as well as the tunneling protocols, if any. The actual trimmed IP payload size MAY be influenced by architecture-specific constraints (e.g., switch cell size), and it MUST be larger than or equal to MIN_TRIM_SIZE to allow the transport to detect the identity of the trimmed packet. For instance, in the context of UE, MIN_TRIM_SIZE should be large enough to ensure the trimmed packet contains the parts of the PDS header that include the PSN, PDC identifiers and any tunnel encapsulation identification fields (e.g., VXLAN header—if present). MIN_TRIM_SIZE is a network-wide configuration parameter that depends on the transport and tunneling/virtualization protocols (if any) used in the network. Guidelines on how to select MIN_TRIM_SIZE are provided in Table 4-1.

When an incoming packet with a DSCP that is DSCP_TRIMMABLE fails buffer admission, the switch MUST trim the incoming packet. The trimmed packet size MUST NOT be larger than the original packet size. When the incoming packet's IP payload length is smaller than MIN_TRIM_SIZE and buffer admission fails, the switch MAY treat the packet as trimmed (without updating its length but updating its DSCP, as shown in the pseudocode), or it MAY drop the packet.

Trimming aims to reduce the amount of information carried by the network in overload scenarios: It reduces the size of the data packets by dropping their payload, turning them into trimmed packets. Trimming is most effective when it is applied to large data packets (e.g., 4 KB), because it reduces the data rate significantly (e.g., on the order of 4 KB/MIN_TRIM_SIZE) while enabling single RTT loss detection. Conversely, trimming packets just fractionally larger than MIN_TRIM_SIZE provides little data reduction and can even be detrimental if a large fraction of bandwidth is used by trimmed packets. To

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

disable trimming for smaller packets, transport protocols SHOULD mark such packets as non-trimmable by using an appropriate DSCP.

After trimming, the switch MUST set the DSCP header field in the IP header to the corresponding DSCP_TRIMMED codepoint associated with the DSCP_TRIMMABLE codepoint in the original packet. This allows the endpoint to detect that the packet was trimmed and will signal to downstream processing that the packet was trimmed.

After trimming, the trimmed packet MUST be a valid IP packet; thus, the switch updates the IP total length field to reflect the new size and updates the IP header checksum to ensure subsequent switches do not inadvertently drop the resulting packet. The Ethernet frame check sequence (FCS) field MUST also be updated such that it is a valid FCS.

The trimmed packet MUST be treated as a new incoming packet for DSCP_TRIMMED for any subsequent processing within the switch performing trimming, which means that it obeys buffer admission rules for the queue associated with DSCP_TRIMMED codepoint on the appropriate egress port. If buffer admission fails for the queue associated with DSCP_TRIMMED, the normal procedure for queue overflow should be followed (e.g., typically the packet will be dropped).

The following pseudocode describes switch behavior when trimming IPv4 / IPv6 packets:

```
// A switch has two queues per port: queue_trimmable and queue_trimmed.
// queue_trimmable and queue_trimmed SHOULD be mapped to different physical
// queues. While not recommended, queue_trimmable and queue_trimmed MAY be
// mapped to the same queue, but with smaller drop thresholds for TRIMMABLE
// packets.

OnRX(pkt): //switch ingress pipeline processing

if pkt.IP.DSCP == DSCP_TRIMMED:  //already trimmed packets
    if (queue_trimmed.size < queue_trimmed.drop_threshold):
        enqueue(pkt,queue_trimmed)
    else:
        drop(pkt)
else if pkt.IP.DSCP == DSCP_TRIMMABLE:
    if queue_trimmable.size < queue_trimmable.drop_threshold:
        //queue_trimmable is the relevant queue associated with
DSCP_TRIMMABLE
        enqueue(pkt,queue_trimmable)
    else: //trim the packet
        trim(pkt)
        if (queue_trimmed.size < queue_trimmed.drop_threshold):
            enqueue(pkt,queue_trimmed)
        else:
            drop(pkt)
else:
    //all other packets—normal processing applies:
    //e.g. enqueue or drop on failed admission checks, etc.

trim(pkt):
    pkt.IP.DSCP = DSCP_TRIMMED
```


Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

```
    PKT_TRIM_SIZE = IP payload size after trimming; must be at least
MIN_TRIM_SIZE.
    if (pkt.IP.version == 4)://IPv4 processing
        pkt.IP.payload = TRIM(pkt.IP.payload,PKT_TRIM_SIZE)
        pkt.IP.TotalLength = PKT_TRIM_SIZE + IP.HeaderLength * 4
        update(pkt.IP.HeaderChecksum)
    else if (pkt.IP.version == 6): //IPv6 processing
        pkt.IP.payload = TRIM(pkt.IP.payload,PKT_TRIM_SIZE)]
        pkt.IP.PayloadLength = PKT_TRIM_SIZE

    //the frame FCS must be recomputed to transmit after trimming.
    pkt.Ethernet.FCS = ...
```

The TTL processing of the trimmed packet is the exact same behavior of the non-trimmed packet.

The buffer admission thresholds for both trimmable and trimmed packets in the pseudocode above are given as static thresholds for simplicity. However, dynamic buffer admission thresholds SHOULD also be supported for both trimmable and trimmed packets.

While a packet with a DSCP_TRIMMED codepoint SHOULD follow the same path through the network as the associated DSCP_TRIMMABLE packet, switches MAY select different paths for trimmed packets to the same destination.

While packet trimming is very useful to ensure single RTT loss detection and thus to trigger fast retransmission, there is no guarantee that for each packet failing buffer admission checks a trimmed packet will be delivered to the destination. This is because trimmed packets are still subject to packet loss at the trimming switch, at subsequent switches along the path, or at the destination. Thus, transport protocols are required to implement loss detection mechanisms (e.g., timeouts) to use as a backstop when a packet is lost and an associated drop notification is not delivered to the receiver.

Informative text: Rationale for Trimming Design

Packet trimming is a form of drop notification; packet switches have long had monitoring capabilities to at least record packet drops, typically by sending the first 64B of the dropped packet to a configured monitor. When a drop happens, the drop notification can be sent to the packet receiver or sender instead, to speed up loss recovery.

An initial proposal that integrated drop notification as a part of the dataplane was the FastLane work from Berkeley [4] . Here, when a packet fails buffer admission, a notification is sent to the packet's source. This approach, also called back-to-sender (BTS), delivers faster notification to the sender, but it is more complex to implement especially when encapsulation (e.g., VXLAN) is used. Another relevant work in this space is the IEEE draft on Source Flow Control (SFC, P802.1Qdw) that targets a broader scope and includes back-to-sender and source flow control [7] . Standardizing the BTS mode of operation is not addressed in this specification.

The Cut Payload paper is the first proposal that suggested sending the drop notification to the receiver instead [3] . Sending the notification to the receiver is simpler to support in switches, as it involves only editing the packet length and the DSCP field. It also has the advantage that it enables near-optimal incast behavior via receiver credit scheduling as proposed by pHost [5] , NDP [2] , and Homa [6] .

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

With Cut Payload, when a certain buffer admission threshold has been passed, the incoming packet's payload is trimmed and the packet enqueued in the same queue. The packet is otherwise left unchanged. Cut Payload has the advantage of requiring a single traffic class, but it has the drawback that it can lead to congestion collapse under some extreme scenarios: When packets are arriving faster in the congested queue than the drain rate of their (trimmed) headers, in steady state the network will carry just headers, and no useful data.

An evolution of Cut Payload called packet trimming was proposed in the NDP paper [2] . The key difference is that trimmed packets are placed into a separate traffic class. Because the associated queue is typically empty, a first benefit is that trimmed packets bypass data packets and arrive quickly at the destination, enabling them to request fast retransmission. Fast retransmission enables the safe configuration of aggressively small buffer thresholds for data packets, typically at around 1 bandwidth-delay product.

A second benefit of utilizing two traffic classes is to restrict the bandwidth available to trimmed packets, preventing congestion collapse. Fair-queueing allows trimmed packets to consume no more than 50% of the bandwidth, while aggressively prioritizing the headers, which are significantly smaller in size. For example, approximately 60 headers can be forwarded for every 4 KB MTU data packet. WDRR with 25% of the bandwidth allocated to trimmed packets is also a good configuration option, at the expense of moderate trimmed packet loss in large incasts.

4.1.1 Interactions with explicit congestion notification

Admitted packets with a DSCP_TRIMMABLE codepoint may be subject to ECN marking if configured by the network administrator. ECN configuration is therefore orthogonal to trimming for trimmable packets.

A switch SHOULD NOT perform ECN marking on trimmed packets. This applies to the packets trimmed at the switch or arriving with a DSCP_TRIMMED codepoint. Because trimming does not change the ECN bits (see paragraph above), and trimmed packets are not subject to ECN marking, the trimmed packets carry the ECN bits of the original data packet. The original ECN bits are useful for congestion control.

CMS Section 3.6.17 provides guidance on appropriate ECN configuration for UET.

4.1.2 Where can trimming be enabled?

Trimming requires that the destination can identify a trimmed packet and process it accordingly. Networks SHOULD prevent destinations that do not support trimming from receiving trimmed packets. Such packet reception may result in undesirable behavior (e.g., a trimmed packet passing header validation checks and then being processed by the upper layers as a regular packet). This implies that trimming is appropriate for backend, datacenter, or enterprise networks where the network operator has knowledge of trimming support and can control DSCP_TRIMMABLE codepoint assignment. If a destination is within the public Internet, it is not possible to know whether the receiver supports trimming; therefore, trimmed packets SHOULD NOT be forwarded outside the backend, datacenter, or enterprise network.

4.1.3 Interactions with upper protocols

Packet trimming SHOULD NOT change any upper layer header fields beyond the outer IP header.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

When a trimmed packet is given to the destination FEP or a switch, it MUST have a DSCP_TRIMMED codepoint regardless of any intermediate representations or encapsulations used in the network. When the trimmed packet uses IPv4, the header checksum field MUST be valid.

Trimmed packets are typically consumed by upper layer protocols to ensure fast retransmission of lost packets. As such, these protocols MUST be trimming-aware when trimming is enabled and MUST be able to identify the original packet based on the trimmed packet they received. To enable this identification, MIN_TRIM_SIZE MUST be set to a value large enough to provide enough information for the upper layer protocol to identify the original packet that was trimmed.

For example, to support trimming with UET, MIN_TRIM_SIZE MUST be configured to a value that ensures that, after trimming, all the headers up to and including the PDS request/response header are kept, to preserve path entropy (for load balancing), the tunnel encapsulation information (if present—for host transport demultiplexing), as well as the PSN and PDC identifiers (to identify the packet).

When multiple encapsulation types are simultaneously used within the same network, the MIN_TRIM_SIZE value MUST be set to the maximum size needed to preserve all relevant transport headers. Switches MAY dynamically determine the appropriate MIN_TRIM_SIZE based on the encapsulation type of each packet.

Table 4-1 summarizes the requirements for MIN_TRIM_SIZE.

Table 4-1 - Trim Size Requirements for Various Transport Protocols[15]

Transport protocol	Transport layer fields to be kept after trimming	MIN_TRIM_SIZE
Ultra Ethernet Transport/IP	Entropy header (4 B), PDS request header (16 B)	20 B
Ultra Ethernet Transport/UDP/IP	UDP (8 B), PDS request header (16 B)	24 B
UET/IPv4 over VXLAN	UDP (8 B), VXLAN (8 B), Ethernet (14 B), IP (20 B), entropy header (4 B), PDS request header (16 B)	70 B
UET/UDP/IPv4 over VXLAN	UDP (8 B), VXLAN (8 B), Ethernet (14 B), IPv4 (20 B), UDP (8 B), PDS request header (16 B)	74 B
UET/UDP/IPv6 over VXLAN	UDP (8 B), VXLAN (8 B), Ethernet (14 B), IPv6 (40 B), UDP (8 B), PDS request header (16 B)	94 B

The numbers in Table 4-1 make several assumptions about the header stack of received packets. For example, the table assumes security is not enabled for UET. If security is enabled, the UET security header sits before the PDS header; thus the MIN_TRIM_SIZE is increased with the size of the UET security header (16 B). The table assumes the inner packet carried within a VXLAN tunnel does not include VLAN tags. Additionally, the table assumes no IP options exist, and other headers have no optional extensions.

4.1.4 Mapping DSCPs to traffic classes for Ultra Ethernet transport

Tables in Section 3.6.4.7 provide a mapping of DSCP values to traffic classes for UET. The traffic classes are generically named TC_low, TC_med, and, TC_high, and their use depends upon the type of UET

[15] When IP version is not specified, the value applies to both IPv4 and IPv6.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

packet and whether the network is best-effort or lossless. Implementations are free to map DSCPs to additional traffic classes; however, the mapping is intended to maintain the independent queuing and processing of the different packet types as specified in Section 3.6.4.7 of the congestion management sublayer chapter.

4.1.4.1 Single instance of UET per network

As specified in the congestion management Section 3.6.4.7, UET RECOMMENDS the use of three traffic classes for each UET PDC: low priority for UET data traffic (TC_low), medium priority for trimmed data packets (TC_med), and high priority for UET control traffic (TC_high). Optionally, trimmed data packets can use the TC_high traffic class if only two traffic classes are available. The priorities of the traffic classes described here are relative to one another and not necessarily relative to other traffic classes used by non-UET traffic.

UET requires at least two DSCPs, one for data packets called DSCP_ TRIMMABLE and one for control packets called DSCP_CONTROL. Switches MUST be configured to map packets with DSCP_TRIMMABLE to the TC_low traffic class and DSCP_ CONTROL to the TC_high traffic class (see 3.6.4.7of CMS).

If trimming is enabled, UET data packets SHOULD be marked as trimmable by setting the DSCP field to a value from the trimmable category (e.g., see Section 4.1 with respect to DSCP_TRIMMABLE).

Switches MUST place trimmed packets into a different traffic class than untrimmed data packets to ensure rapid delivery to the destination. Trimmed and control packets SHOULD be placed in separate traffic classes, TC_med and TC_high respectively.

The codepoint configured for DSCP_TRIMMED MUST be distinct from DSCP_TRIMMABLE.

The codepoint configured for DSCP_TRIMMED MUST be different from DSCP _CONTROL. This enables the receiver to easily detect trimmed packets and allows the implementation of different drop thresholds in the switches for the TC_med traffic class, thus protecting the control traffic from excessive trimmed traffic.

For DSCPs from the trimmed category (see Section 4.1), it is also useful if the receiver can distinguish between incast (last hop) trims and non-last hop trims.

A unique DSCP called DSCP_TRIMMED_LAST_HOP SHOULD be set on packets that are trimmed at the last hop switch (e.g. top-of-rack (TOR)) and are destined to directly connected hosts (e.g., TOR downlinks). In such cases, packets with DSCP_TRIMMED_LAST_HOP are mapped to the TC_med or the TC_high traffic class if only two TCs are available.

In summary, for a single UET instance, three traffic classes are RECOMMENDED and between three and four DSCPs are used, depending on DSCP availability. The recommended number of DSCPs is four: DSCP_TRIMMABLE for data and trimmable packets, DSCP_CONTROL for control packets, DSCP_TRIMMED for trimmed packets, and DSCP_TRIM_LASTHOP for last hop trimmed packets.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

4.1.4.2 Multiple instances of UET per network

In some circumstances, two isolated instances of UET may be running in the same network but using different traffic classes. In this case, the single-instance UET model described above can be extended. To extend the traffic class mapping for the multiple instance case, additional DSCP values for DSCP_TRIMMABLE can be used, called DSCP_TRIMMABLE1 and DSCP_TRIMMABLE2, which are mapped to two different lower-priority traffic classes.

A shared higher-priority traffic class SHOULD be used for the control traffic of both instances. Alternatively, deployments MAY choose to use separate traffic classes, if available, to isolate control traffic for each UET instance.

Using a single traffic class for control traffic avoids overusing scarce traffic classes in the network. This, however, reduces isolation between UET instances, because one instance with excessive control traffic may reduce performance for another instance.

In the case where a common DSCP_CONTROL codepoint is used for the control traffic of multiple instances, the DSCP_TRIMMED codepoint SHOULD be mapped to a different medium-priority or high-priority traffic class. When traffic classes are scarce, trimmed packets MAY be also mapped to the higher-priority traffic class, and packets other than DSCP_CONTROL packets SHOULD have a lower drop threshold to reduce the chance that bursts of trimmed packets lead to control packet loss.

When using a single DSCP for all control packets and another DSCP for trimmed packets from multiple UET instances, the DSCPs can be used to correctly identify trimmed packets. However, the target FEP cannot rely on the DSCP field to demultiplex between trimmed packets belonging to different UET transport instances.

With this approach, when running N UET instances, the network will be configured to have N + 2 traffic classes and N+3 DSCPs: one codepoint for each of the N data packet codepoints mapping to the N data traffic classes (lower priority); one codepoint for control packets mapping to the control traffic class (higher priority); and one codepoint for trimmed and one for trimmed last-hop packets, mapping to the trimmed traffic (medium priority).

The alternative is to use a separate set of traffic classes for each UET instance. This provides better isolation between the control and trimming traffic for the instances of UET transport at the expense of using additional traffic classes. For example, when a sufficient number of traffic classes are available, a straightforward approach is for two UET instances to utilize four distinct codepoints for trimmed packets: DSCP_TRIMMED1, DSCP_TRIMMED_LAST_HOP1, DSCP_TRIMMED2, and, DSCP_TRIMMED_LAST_HOP2. One pair of codepoints are configured for each UET instance, and the codepoints are mapped to the respective medium priority traffic classes.

In total, if the network has N UET instances, it will use N*3 traffic classes and a minimum of N*4 DSCPs. This approach is expensive, both in the number of traffic classes used and the number of codepoints used, but offers the best isolation.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

4.1.5 Mapping DSCPs to traffic classes for other transports

The DSCPs used in packet trimming can be mapped to traffic classes in multiple ways, depending on the availability of traffic classes, the requirements of the host transport and applications that utilize the trimming information, and the operator's requirements.

To implement cut payload [3] -style behavior, both DSCP_TRIMMABLE and DSCP_TRIMMED codepoints SHOULD be mapped to the same traffic class in the network. The drop threshold for packets with DSCP_TRIMMABLE SHOULD be set to a lower value than the drop threshold for packets with DSCP_TRIMMED, as shown in Figure 4-1; otherwise, all trimmed packets may be dropped.

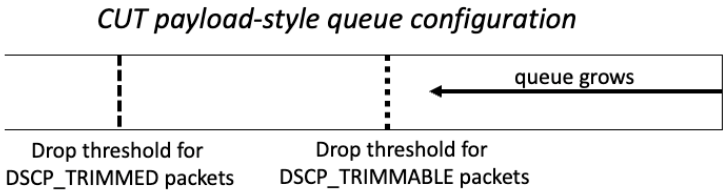


Figure 4-1 - Drop Threshold Settings

To implement NDP [2] -style behavior, DSCP_TRIMMABLE and DSCP_TRIMMED codepoints SHOULD be mapped to two traffic classes (DSCP_TRIMMABLE to the TC_low traffic class and DSCP_TRIMMED to the TC_high traffic class). The two traffic classes SHOULD be configured to share capacity in a weighted deficit round-robin manner (e.g., fair queueing or 2:1 WDRR for class B). The result of this mapping is that data packets from transports that support packet trimming are carried on the TC_low traffic class, while the resulting headers (trimmed packets) are carried on the TC_high traffic class from the point they are trimmed to their destination. This approach enables, at the cost of using two traffic classes instead of one, faster delivery of trimmed packets and, with appropriate queue scheduling, can bound the total bandwidth used by trimmed packets to a fraction of the link capacity. This removes the possibility of congestion collapse.

4.1.6 Security considerations

In a zero-trust security model, networking infrastructure is not considered a trusted entity from the endpoint's perspective. A trimmed packet MAY allow the receiver to gain additional information from the trimmed packet and affords rapid handling of the congestion information.

In the context of UET, trimming a packet prevents authentication, because portions of the packet are removed including the authentication tag (the ICV). However, it MAY be possible to decrypt the trimmed packet under some circumstances (for instance when the cipher used is AES in counter mode).

In the context of the receiver, processing of unauthenticated encrypted trims MUST NOT: (a) cause a change of state of the connection, beyond what an attacker can achieve via ECN marking or dropping packets, or (b) cause data from the trimmed packet to be accepted or subsequent data to be rejected. In particular, the arrival of a trimmed packet MUST NOT cause a congestion response or retransmission to take place beyond what would have happened if the packet was dropped. In response to receiving a

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

trimmed packet, a responder SHOULD send a signal (e.g., UET NACK) to the requestor indicating that the request requires retransmission.

4.1.7 References

[1] C. Guo, H. Wu, Z. Deng, G. Soni, J. Ye, J. Padhye and M. Lipshteyn, "RDMA over commodity ethernet at scale," in *Proceedings of the 2016 ACM SIGCOMM Conference*, 2016.

[2] M. Handley, C. Raiciu, A. Agache, A. Voinescu, A. W. Moore, G. Antichi and M. Wójcik, "Re-architecting datacenter networks and stacks for low latency and high performance," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, 2017.

[3] P. Cheng, F. Ren, R. Shu and C. Lin, "Catch the whole lot in an action: Rapid precise packet loss notification in data center," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, 2014.

[4] D. Zats, A. P. Iyer, G. Ananthanarayanan, R. H. Katz, I. Stoica and A. Vahdat, "FastLane: Agile drop notification for datacenter networks," *Dept. EECS, Univ. California at Berkeley, Berkeley, CA, USA, Rep. UCB/EECS-2013-173,* 2013.

[5] P. X. Gao, A. Narayan, G. Kumar, R. Agarwal, S. Ratnasamy and S. Shenker, "pHost: Distributed near-optimal datacenter transport over commodity network fabric," in *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, 2015.

[6] B. Montazeri, Y. Li, M. Alizadeh and J. Ousterhout, "Homa: A receiver-driven low-latency transport protocol using network priorities," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, 2018.

[7] IEEE P802.1Qdw, "Source Flow Control," 2022. [Online]. Available: https://1.ieee802.org/tsn/802-1qdw/.

[8] IETF RFC 2119, "Key words for use in RFCs to Indicate Requirement Levels," 1997. [Online]. Available: https://www.rfc-editor.org/rfc/rfc2119.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

4.2 Congestion Signaling (CSIG)

Contemporary machine learning (ML) training and high-performance computing (HPC) workloads compute across a switched network fabric in closely coupled, synchronous fashion. As a result, they demand short flow completion times, even at the tail of the distribution. For these workloads, overall application progress can stall when even a single flow's completion is delayed.

When a flow completes, it abruptly releases the capacity it had previously consumed at links along its path. Other flows that share those links must be able to increase their rates rapidly to put that newly idle capacity to use promptly. Moreover, achieving low tail flow completion times requires effective multi-path spreading of flows across the fabric's many paths.

In oversubscribed workloads, stretching of tail completion times for "losing" flows can become a bottleneck for overall application performance. These performance constraints demand unprecedented responsiveness from congestion control protocols, and in turn call for new inputs to congestion control that offer fine-grained and timely information about the congestion state of the network fabric.

End-host congestion control has long employed simple end-to-end signals, such as round-trip time (RTT) and packet loss events, to summarize the extent of congestion on a full path. Using explicit congestion notification (ECN) marks, switches directly provide a simple, Boolean indication to end hosts of whether, short of a drop, a packet encounters more than some threshold queue length at one or more hops along its path. These established signals are coarse grained*:* as end-to-end, path-summary metrics, RTT, and packet-loss event counts reflect the total congestion along a full path and do not reveal conditions at any individual switch.

An ECN mark offers only a single-bit quantification of congestion, and as a path summary, reports only the presence of congestion along any extent of the full path, or its absence along all of it. These signals do not meet the finer-grained congestion control needs of contemporary ML and HPC workloads. For example, to increase a flow's sending rate rapidly to match the abruptly released capacity on a link precisely, a sender would benefit from discovering the available bandwidth at the single most constrained link along a path: the link with the minimum available bandwidth.

While per-link available bandwidth is readily measurable by individual switches, it is not directly observable by end hosts. Absent finer-grained signals, status-quo congestion control approaches are limited to searching conservatively for a path's available bandwidth by increasing their sending rate and detecting when they congest the path—a slow process.

As another example, end-to-end delay measurements mix together queueing delays from all congested hops along a path, such that in delay-based congestion control, the *number* of congested hops determines a sender's perceived extent of congestion, which can yield bandwidth unfairness between flows. Only the queueing delay at a flow's single most congested hop (its true bottleneck) should determine a sender's sending rate. This quantity, again, is not directly observable by end hosts using today's coarse-grained, path-summary signals.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

These ML training and HPC workloads present further challenges in network fabric monitoring and performance analysis. Traditional forms of network telemetry, such as time series of link utilizations or packet drop rates across all links in the fabric, often fall short in explanatory power because they are *decoupled from application context*. They are labor-intensive to correlate with specific application flows at specific time points, and often are recorded at too coarse a resolution in time to reveal the pathology experienced by a flow (e.g., during an extremely brief traffic burst). Effective performance diagnosis and monitoring at scale require fine-grained network telemetry, both in time and at a per-switch granularity, that closely correlates in-network measurements with a workload's context—ideally, with the *experience of an application's individual packet*.

This specification describes congestion signaling (CSIG), a simple and efficient in-band network telemetry method that collects fine-grained signals that describe the network fabric's congestion state. The central insight underlying CSIG's design is that for many telemetry signals useful in congestion control and performance analysis, the result of a simple aggregation function computed over the congestion state at switches along a packet's path is of interest. For example, for several signal types, the congestion state only at a single switch along a path is of interest, where the switch of interest is the one where a signal's value is an extremum (maximum or minimum) among the signal's values at all switches along the path. By leveraging this insight, CSIG offers the following desirable properties:

- **Targeted, per-switch signal granularity**: CSIG can return a signal from the single bottleneck switch along a path and thus exposes congestion state not generally available from end-to-end, whole-path signals.
- **Space-efficient telemetry collection**: While CSIG measures signal data at each switch along a path, a measurement adds only a short, 4-byte (or for increased measurement precision, 8-byte) tag to a packet's header, regardless of path length in hops. This space efficiency is possible because CSIG tags carry only a single aggregated value (e.g., maximum or minimum value observed thus far along a path, depending on the signal type being measured).
- **Compute-efficient telemetry collection**: The operations a switch must perform to process a CSIG tag are simple and latency-efficient.
- **In-band collection that captures application/transport context**: Because CSIG tags are low in space overhead and in processing overhead at switches, an application or transport may attach them to every packet it transmits. The congestion signals collected by these packets will reflect how switches process that application's and transport's actual packets (and not how distinct packets sent by a measurement system, which arrive at different points in time than "native" application traffic, are processed).
- **Transport- and encapsulation-agnostic**: CSIG tags are carried in layer 2 headers and thus switch processing of CSIG tags does not involve processing of transport headers, nor does it incur any complication when used with encrypted transports.
- **Extensibility**: CSIG provides for future specification of additional signal types within the protocol's framework.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

4.2.1 Terminology

Term	Description
CSIG	Congestion signaling
EOP	End of packet
Locator metadata	Contextual information stored in a CSIG tag to identify where in the network that tag was updated
Monitoring entity	An entity responsible for collecting telemetry captured via CSIG and/or other methods, and provisioned to scale appropriately to report arrival rates
Probe	CSIG-tagged packet created for the express purpose of in-network telemetry collection (in contrast with CSIG-tagged packets sent by non-telemetry applications)
Signal type and value	In-network telemetry information of interest
SOP	Start of packet
VLAN tag	An IEEE Std 802.1Q-2022 clause 9-compliant VLAN tag [2]

4.2.2 Overview

This specification defines CSIG, a simple in-band network telemetry method in which packets collect fine-grained measurements at switches as they traverse a network path. CSIG incurs low telemetry collection overhead by limiting in-packet state to a single, fixed-size CSIG tag that stores a signal value measured at a single switch (independent of the number of hops the packet traverses).

Each switch along a path inspects and (where appropriate) updates this single tag in accordance with a signal type encoded in the tag and policies configured by the network administrator (e.g., choices of parameters that govern quantization and time granularity of measured values). Network endpoints can consume arriving CSIG signals for use by in-band functions such as congestion control and load balancing. Monitoring entities at network endpoints can also process arriving CSIG signals for use in such out-of-band functions as performance analysis, debugging, and SLA assurance. When included in packets that comprise applications' usual traffic flows, CSIG tags gather in-context telemetry that indicates the state of the network path as experienced by an application's packet. In addition, monitoring frameworks can include CSIG tags in dedicated probe packets to measure the network's state.

The CSIG tag is a link-level Ethernet tag that carries telemetry information end to end. Its definition is independent of any specific layer 3 network encapsulation or layer 4 transport. Although a CSIG tag does not itself specify the higher-layer transport and/or application for which it carries a signal, that signal must nevertheless be interpreted in context. This specification discusses means to associate the signal in a CSIG tag with the appropriate higher-level transport context (in Ultra Ethernet Transport [UET], the appropriate congestion control context [CCC]), and means for reflection of the signal in a tag on a UET packet arriving at a destination endpoint back to the source endpoint in the header of a UET layer 4 ACK or NACK.

Informative Text:
Note that because CSIG is an optional UE feature, for CSIG tags to be carried end to end in a UE fabric all switches in the UE fabric must support CSIG, and the fabric must be configured to enable the use of CSIG. Please refer to further discussion of these deployment considerations in Section 4.2.10.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Each switch along the path traversed by a CSIG-tagged packet identifies the signal type encoded in the tag and updates the associated value within the tag using a simple aggregation function. The switch can also write locator metadata within the tag to indicate properties of the network hop that updated the signal.

CSIG supports three signal types in its base profile:

- The max(Delay) signal finds the delay bottleneck along a path. When each switch along a CSIG-tagged packet's path receives the packet, it conditionally updates the queueing delay signal value stored in the CSIG tag using the maximum function, where the switch overwrites the delay signal value in the CSIG tag with its own local delay value if it exceeds that in the CSIG tag.
- The min(ABW) signal finds the available bandwidth bottleneck along a path. When each switch along a CSIG-tagged packet's path receives the packet, it conditionally updates the available bandwidth signal value stored in the CSIG tag using the minimum function, where the switch overwrites the available bandwidth value in the CSIG tag with its own local available bandwidth value from the egress port if it is less than that in the CSIG tag.
- The min(ABW/C) signal does the same, but for each switch's relative available bandwidth, normalized by output port capacity.

(Full definitions of these CSIG signal types may be found in Section 4.2.4.1.2.)

Figure 4-2 illustrates through a simple example how CSIG telemetry functions on endpoints and in the network fabric. It depicts a high-level schematic of a CSIG-tagged packet, showing both the CSIG tag carried on a packet's forward path and the reflection of the signal from the destination endpoint back to the source endpoint in an layer 4 transport header. (Examples of the placement of the CSIG tag in other layer 2 frame types may be found in Figure 4-8 and Figure 4-9.) It further shows the local signal value at each hop for three signal types—max(Delay), min(ABW), and min(ABW/C)—and the distinct bottleneck locations for each of these signal types along the path. Upon receiving a CSIG-tagged packet, each switch along the path conditionally updates the signal value in the CSIG tag according to the signal type specified in the CSIG tag, the min() or max() function specified in the signal type's definition (and name), and the local value for that signal type at that switch.

This specification defines the CSIG telemetry method in an application-agnostic fashion. It further discusses CSIG's relevance to multiple use cases relevant in the UE setting.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

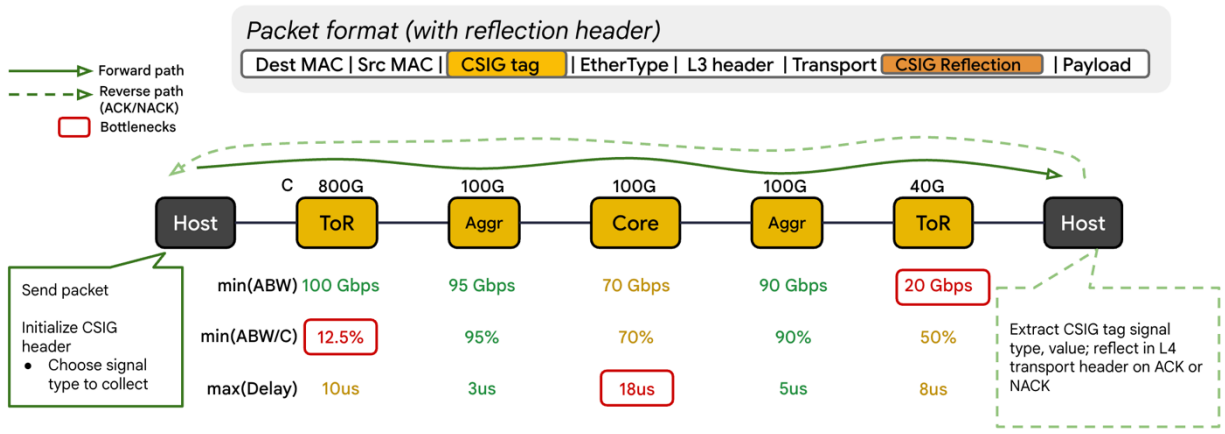


Figure 4-2 - Example of CSIG Endpoint and Fabric Processing

4.2.3 Use Cases

The collection of network telemetry with CSIG benefits multiple UE-relevant use cases.

4.2.3.1 End-to-End Congestion Control

When a sender's transport protocol includes CSIG tags on its packets, the returning information about the congestion state at switches along each packet's path can be used to improve congestion control's performance. This information can include the least available bandwidth at any link along a packet's path (useful for promptly and precisely claiming newly released link capacity in NSCC and RCCC; see sections 4.2.9.1.3 and 4.2.9.2.3) and the greatest queueing delay at any queue along a packet's path (useful for improving the bandwidth fairness of delay-based congestion control schemes, such as NSCC, and for responding to oversubscription in RCCC; see Section 4.2.9.3.4).

4.2.3.2 Multi-Path Load Balancing

CSIG signals describing the least available bandwidth at any link along a packet's path are also useful for choosing higher-spare-capacity paths in multi-path load balancing, as applied in NSCC and RCCC (see sections 4.2.9.1.3 and 4.2.9.2.3).

4.2.3.3 Telemetry

The signal collected in a CSIG tag provides useful telemetry information describing the congestion state of a path through the fabric.

These signals can help developers understand the extent to which their applications' packets encounter congestion (e.g., service delay, normalized queue depth) or available link capacity. This insight can be used to inform application design changes or debugging efforts.

CSIG tags can assist in localizing congestion points within the network fabric and evaluating how effectively the fabric's provisioning meets workload demands.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

For telemetry purposes, signals from CSIG tags can be collected by software running on FEPs, either by forwarding the CSIG signal directly to the receiving application or by routing it to a dedicated telemetry collection application.

Methods for collection of telemetry from received CSIG signals are outside the scope of this specification.

4.2.4 CSIG Tag Formats

This specification defines two CSIG tag formats: a 4-byte *compact* CSIG tag and an 8-byte *wide* CSIG tag.

The compact CSIG tag (Figure 4-3), is a 4-byte header field identical in frame offset and field width to an IEEE Std 802.1Q-2022 clause 9-compliant VLAN tag (hereafter referred to as "VLAN tag" for brevity) [2] . This choice of compact CSIG tag format simplifies adding CSIG functionality to the many switches that can leverage VLAN tag match-action support to recognize and update CSIG tags.

All UE-compliant switches and hosts that support CSIG MUST support the compact CSIG tag format.

In comparison, the wide CSIG tag format (Figure 4-4), 8 bytes in length, targets deployments in which switches have native CSIG support. The wide tag contains the same fields with the same functions as the compact tag, but by virtue of the greater number of bits available for data fields in the wide tag, the wide tag supports more signal types (**csig.t**: 16, vs. 8 for compact tags), finer-grained signal value measurements (**csig.s**: 1 million quantized values vs. 32 buckets for compact tags), and more detailed locator metadata (**csig.lm**: 15 bits vs. 6 bits for compact tags). The **csig.t** constants defined for signal types are identical for the compact and wide tag formats, but the wide tag format supports additional (as yet unused) signal type definitions that have no corresponding definitions in the compact tag format. The compact and wide tag formats apply different transformations in their **csig.s** signal value representations. The definitions of the **csig.d** (Do Not Modify) bit and **csig.lm** (Locator Metadata) fields in the wide CSIG tag format are the same as those for the compact CSIG tag format, apart from the greater width of the LM field in the wide CSIG tag format.

UE-compliant switches and hosts that support CSIG MAY additionally support the wide CSIG tag format.

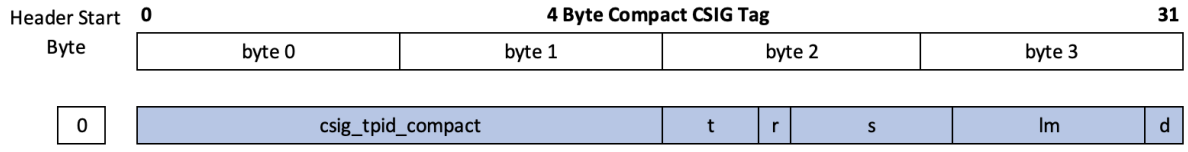


Figure 4-3 - 4-Byte Compact CSIG Tag

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

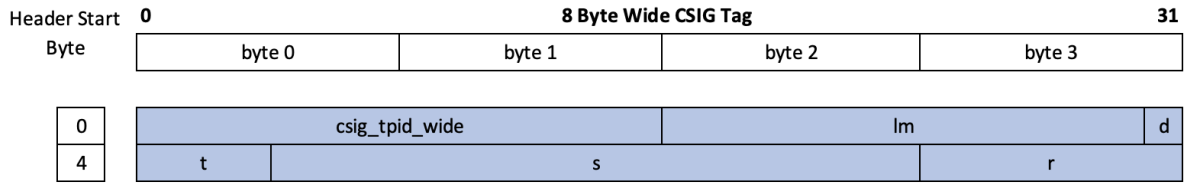


Figure 4-4 - 8-Byte Wide CSIG Tag

The field definitions for the Compact CSIG tag format are shown in Table 4-2.

Table 4-2 - 4-Byte Compact CSIG Tag Field Definitions

Bit Offset	Width (in bits)	Field Name	Comments
0-15	16	csig_tpid_compact	Configured EtherType
16-18	3	t	Type of signal
19	1	r	Reserved
20-24	5	s	Quantized signal value
25-30	6	lm	Locator metadata
31	1	d	Do not modify the CSIG tag

The 8-byte wide CSIG tag fields are wider and arranged in a different order, as shown in Table 4-3.

Table 4-3 - 8-Byte Wide CSIG Tag Field Definitions

Bit Offset	Width (in bits)	Field Name	Comments
0-15	16	csig_tpid_wide	Configured EtherType
16-30	15	lm	Locator metadata
31	1	d	Do not modify the CSIG tag
32-35	4	t	Signal type
36-55	20	s	Quantized signal value
56-63	8	r	Reserved

4.2.4.1.1 CSIG Tag Protocol Identifier

The CSIG tag protocol identifier (**csig.tpid**) is a fixed, constant, two-octet EtherType that identifies the CSIG tag. The compact and wide CSIG tags are identified by distinct EtherTypes to distinguish between the compact and wide CSIG tag formats, to simplify hardware, and to make best use of the tag formats' limited bits.

This specification refers to the EtherType value identifying compact CSIG tags as **csig.csig_tpid_compact,** and the EtherType value identifying wide CSIG tags as **csig.csig_tpid_wide**.

Until permanent assignments of CSIG EtherTypes are obtained, CSIG implementations MUST support configurable EtherType values for the compact and wide tags. EtherType values used for the compact tag and wide tag (if supported) MUST be distinct and MUST NOT conflict with any other EtherTypes in use in the CSIG deployment domain.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

CSIG implementations SHOULD support configurable EtherType values for the compact tag and wide tag (if supported) even after a permanent assignment has been obtained, subject to the same constraints as above on configured values.

4.2.4.1.2 Signal Type (csig.t)

The signal type field identifies the type of signal information stored in the signal value **csig.s** field of the CSIG tag. Each signal type's definition in this specification includes an associated aggregation function. When updating the signal value in a CSIG tag, the switch pipeline compares its local value for that signal type to the signal value in the CSIG tag using this aggregation function. Table 4-4 and Table 4-5 shows the supported signal types and associated aggregation functions.

Table 4-4 - Signal Types for the Compact Tag

t	Signal	Profile	Aggregation Function	Comments
0	ABW	base	min	Available bandwidth per port
1	ABW/C	base	min	Relative available bandwidth per port
2	Delay	base	max	Per-hop delay
3	nQD	extended	max	Queue depth normalized by port speed
7	Experimental	extended	[configured]	Reserved for experimental use

Table 4-5 - Signal Types for the Wide Tag

t	Signal	Profile	Aggregation Function	Comments
0	ABW	base	min	Available bandwidth per port
1	ABW/C	base	min	Relative available bandwidth per port
2	Delay	base	max	Per-hop delay
3	nQD	extended	max	Queue depth normalized by port speed
15	Experimental	extended	[configured]	Reserved for experimental use

A single signal type (**csig.t)** is reserved for experimental use for each of the compact and wide CSIG tag formats: 7 for the compact tag, and 15 for the wide tag. A UE fabric whose switches and endpoints support CSIG MAY configure UE switches and endpoints to use the experimental signal type for the compact and/or wide tags. The means of configuring the experimental signal types in UE switches and endpoints is implementation-specific and outside the scope of this specification.

UE switches MUST support the set of signals defined in the CSIG base profile (see Section 4.2.7.1).

4.2.4.1.3 Signal Value (csig.s)

The signal value field carries a switch-measured value for the signal type encoded in the CSIG tag. It is quantized into buckets with explicit, configured [min, max] ranges in the compact tag format, and encoded in a base-relative, uniform bucket-size representation in the wide tag format.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

The switch computes the signal's local value and quantizes it. The entity within the switch that computes a signal's local value may vary with the signal type; the definitions of individual signals in this specification discuss the entity(ies) that compute a signal's local value. The pre-quantization locally computed value and quantized value MUST both be integers; this constraint simplifies implementation by avoiding the complexity of floating-point arithmetic, both in the computation of local signal values and in their quantization. Individual signal type definitions can straightforwardly enforce this integer-value constraint by specifying units accordingly. The switch pipeline compares the quantized local signal value with the incoming CSIG tag's quantized signal value according to the aggregation function associated with the CSIG tag's signal type.

4.2.4.1.4 CSIG Tag Quantization Algorithm

Figure 4-5 illustrates how a switch's hardware might maintain signals' local values and quantize them. In accordance with the signal type in an arriving packet's CSIG tag, the switch pipeline compares the quantized signal value in the packet's CSIG tag with the quantized local value for the signal of the same type and, as appropriate, updates the signal value in the packet. The use of a background thread to compute a signal's local value in Figure 4-5 is a non-normative example; other implementations, such as ones based on per-packet updates, are possible. The details of local signal computation, within the confines of the per-signal definitions in this specification, are implementation-specific. An implementation will compute a local signal value, quantize it, and perform *CSIG-Compare-and-Update()* of the quantized value per the compact or wide tag.

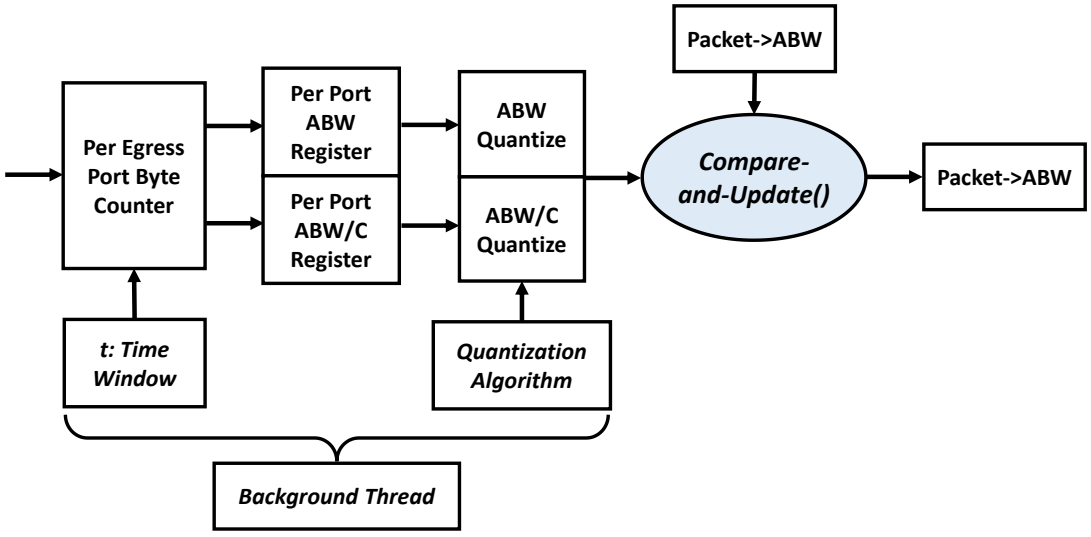


Figure 4-5 - Example Local Signal Quantization and Packet Signal Compare-and-Update

4.2.4.1.5 Compact CSIG Tag Quantization Algorithm

The quantization applied to values stored in compact CSIG tags is a simple mapping of the raw, unquantized value into one of several buckets. Each bucket is parameterized by minimum and maximum

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

signal values set by the switch control plane during configuration. The buckets MUST not overlap in their ranges and their ranges MUST be contiguous. Bucket ranges are inclusive.

Because different signal types often carry values of differing magnitudes, quantization bucket ranges for compact CSIG tags are configured separately for each signal type. The switch control plane sets buckets' minimum and maximum values during configuration.

The compact CSIG tag includes 5 bits of signal value. There are $2^5 = 32$ quantized values, each corresponding to one bucket. Before quantizing a raw signal's value, the switch hardware MUST ensure that the raw signal's value is of a precision (length in bits) that matches the precision of that signal's quantization bucket ranges. If the precision of the raw signal's value exceeds that of the signal's quantization bucket ranges by *e* bits, the hardware right bit-shifts the raw signal's value by *e* bits. No bit-shift is performed if the precision of the raw signal value is less than or equal to that of the signal's quantization bucket ranges. The hardware then performs a simple range check on the resulting signal value to map the raw value to the bucket in which it falls. The index of that bucket is the quantized value.

If a raw signal value is less than the minimum bound of bucket index 0 (the least raw signal value represented in the quantization's configuration), it is out-of-range low, and the switch hardware MUST quantize that raw signal value as falling within bucket index 0. If a raw signal value is greater than the maximum bound of bucket index 31 (the greatest raw signal value represented in the quantization's configuration), it is out-of-range high, and the switch hardware MUST quantize that raw signal value as falling within bucket 31. Switch implementations SHOULD keep event counters for out-of-range low and out-of-range high occurrences during quantization. The reporting and use of these counters is implementation-specific and outside the scope of this specification.

Table 4-6 - Quantization Algorithm—Compact Tag

Bucket Index	Range
0	[min, max]
1	[min, max]
…	
31	[min, max]

Note:
- 32 buckets
- Fully configurable
- 32 range checkers per signal type (any sharing across signal types is implementation-specific)

4.2.4.1.6 Wide CSIG Tag Quantization Algorithm

The wide CSIG tag includes 20 bits of signal value. There are $2^{20} = 1048576$ quantized values, each corresponding to one bucket. For a given signal type, the quantization is parameterized by a bucket range exponent $e >= 1$ and base value $b >= 0$, where *b* MUST be either zero or a power of two. 2^e contiguous unquantized signal values map to each bucket. To quantize an unquantized signal value *v*, a device computes $(v - b)$ and maps the result to the index of the bucket in which it falls—a simple shift

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

right by *e* bits, as each bucket spans 2^e unquantized values. The resulting bucket index is the quantized value.

The switch control plane sets *e* and *b* per signal type during configuration.

Each bucket index *i* represents the range of unquantized values:

$$[b + i2^e, b + (i + 1)2^e - 1]$$

And thus, for a given *b* and *e*, the full range of unquantized values representable in quantized form is:

$$[b, b + 2^{e+20} - 1]$$

To quantize an unquantized signal value *v*, yielding bucket index *i*, the corresponding quantized value, a device computes:

$$\mathrm{i} = (v - b) \gg \mathrm{e}$$

The result of the quantization function is undefined for input unquantized signal values *v* that fall outside the full range.

This quantization function is efficient to implement in hardware, given the simplicity of the subtraction and shift-right operations.

As with the handling of out-of-range values in the quantization for the compact tag, if a raw signal value is less than *b* in the wide-tag quantization, the switch hardware MUST quantize it as falling in bucket 0, and if a raw signal value is greater than $(b + 2^{e+20} - 1)$, the switch hardware MUST quantize it as falling in bucket (2^{20}-1). For the wide tag quantization, too, switch implementations SHOULD keep event counters for out-of-range low and out-of-range high occurrences during quantization. The reporting and use of these counters is implementation-specific and outside the scope of this specification.

4.2.4.1.7 Locator Metadata (csig.lm)

Locator metadata is a 6-bit field in the compact tag and a 15-bit field in the wide tag. It identifies the location within the network and/or attributes of the switch where the signal value in a CSIG tag was updated. When a switch updates the signal value in a packet's tag, it MUST also update the locator metadata in the tag.

Locator metadata is updated by the bottleneck hop (i.e., the hop updating the signal value as part of the *CSIG-Compare-and-Update()* operation). Locator metadata may provide deployment-specific information for deployment-specific differentiation or vendor extensions (e.g., vendor-specific enhancements to UE CC algorithms). Locator metadata may provide additional hints to improve UE congestion management, path selection, or just provide visibility of the bottleneck hop as telemetry data.

Informative Text:
The distinction between a last-hop bottleneck and one at any earlier point along a path is useful to tell incast apart from "non-leaf" congestion. This distinction is important because load balancing

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

should not attempt to change paths in an incast but should do so when congestion occurs at any other link. Trimmed packets already use distinct DSCP code points to distinguish the incast and non-incast cases.

Some example information that can be encoded in locator metadata is as follows:

- Outgoing Link Capacity
  - For normalized signal types like ABW/C, knowing the link capacity is helpful.
  - Common link speeds can be encoded in 2 bits.
- Outgoing Link Orientation
  - Uplink, downlink, side link, penultimate hop link.
  - Fabric information.
  - Fat tree stages or group ID in DragonFly++ topology.
  - Rail/plane ID in multi-rail or multi-plane topology.
- Outgoing PortID/DeviceID
  - Possibly only applicable to wide format: 9 bits are needed for 512-radix switches.
  - TTL of packet on egress: 8 bits of information to detect the exact hop in the path.

4.2.4.1.8 Do Not Modify (csig.d)

The **csig.d** bit provides a Do Not Modify indication for a CSIG tag. When a switch receives a CSIG tag with **csig.d** = 0, that switch MUST, according to the signal type (**csig.t**) specified in the tag and the signal type's aggregation function, perform *CSIG-Compare-and-Update()* on the signal value (**csig.s**) in that tag. When a switch receives a CSIG tag with **csig.d** = 1, that switch MUST NOT modify the signal value (**csig.s**) in that tag. When a CSIG tag is first initialized, its **csig.d** bit MUST be set to zero, to indicate that receiving switches are permitted to update the signal value in that tag.

In a UE fabric configured to perform packet trimming and that supports CSIG, all switches MUST support the following base case behavior: When a CSIG-tagged packet arrives at a heavily congested queue, and the receiving switch trims the packet, that switch MUST set the CSIG tag's **csig.d** bit, and SHOULD make no further modifications to the CSIG tag. Some deployments MAY overwrite the CSIG tag's signal value (**csig.s**) with signal information from the switch where trimming occurs; this policy may benefit congestion control at endpoints. Those use cases and their configuration are outside the scope of this specification.

A CSIG-capable switch may support configurations in which it does not set the **csig.d** bit when trimming CSIG-tagged packets. Those use cases and their configuration are outside the scope of this specification. Moreover, all UE-compliant switches that support CSIG MUST support the base case behavior of setting the **csig.d** bit in the CSIG tag on any packet they trim.

The rationale for this base case behavior of setting the **csig.d** bit in CSIG tags on trimmed packets is that once a packet has been trimmed, it traverses high-priority queues, whose measured signal values may not be representative of the congestion state at the lower-priority queues traversed by untrimmed

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

packets. Rather than potentially overwrite the CSIG tag with an unrepresentative signal from the higher-priority queues at subsequent switches along the path, this approach preserves the signal value in the tag prior to the trimming of the packet, which reflects the representative congestion state of lower-priority queues along the path prior to trimming.

Some deployments MAY set the **csig.d** bit under other circumstances than at packet trim time to prevent a CSIG tag from being updated by specific switches in the fabric. Those use cases and their configuration are outside the scope of this specification.

4.2.4.1.9 Reserved (csig.r)

The **csig.r** bits SHALL be set to 0 by the FEP that initializes and inserts a CSIG tag on a packet for transmission (see 4.2.5.2). The **csig.r** bits are ignored upon receipt.

4.2.5 Functionality Within the CSIG Domain

The CSIG domain is defined as the set of UE fabric elements that participate in CSIG tag processing. The following sections describe the functionality implemented at devices within the CSIG domain.

4.2.5.1 Capability Discovery

UE 1.1 Link Negotiation (Section 5.3) provides support for negotiating CSIG capability over each link using the UE Link Negotiations Options Organization Specific LLDP TLV. End-to-end CSIG configuration is done through other local or system-wide management capabilities such as the UE Switch Abstraction Interface (SAI) and the UE RedFish interfaces. The LLDP capabilities provided allow the ports of a link to indicate if they are capable of processing only compact CSIG tags, capable of processing both compact and wide CSIG tags, or not capable of processing CSIG tags.

A host or switch that is CSIG-capable MUST support the compact tag and the base profile. The host or switch MAY additionally support wide tags and MAY additionally support the extended profile. For a switch to be capable of processing CSIG tags at a port it MUST be able to forward the CSIG tag information through the switch along with the data packet to any other switch port and update the CSIG tag at the egress port. In addition, a host or switch that is CSIG-capable MUST be able to strip the CSIG tag formats it supports.

4.2.5.1.1 CSIG LLDP Transmit and Receive States

When a device that supports CSIG determines that a peer device on a link does not support CSIG, that device performs the *CSIG-Strip-and-Forward()* function (see Section 4.2.5.4) when forwarding CSIG-tagged packets on that link. When a device supports compact CSIG tags but the peer does not, it MUST strip the compact CSIG tag from the packet before transmitting it on the link, irrespective of the **csig.d** bit's value. When the device supports wide CSIG tags but the peer does not, it will strip the wide CSIG tag from the packet before transmitting it on the link, irrespective of the **csig.d** bit's value.

The local and remote LLDP database values of the *csig-capable* object are used in conjunction with the presence or absence of the *csig-capable* object and with the local management configuration settings to determine the forwarding of CSIG tags at the port. Table 4-7 describes how transmission and reception of CSIG-tagged packets are processed at a port controlled using LLDP.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Table 4-7 - LLDP Port Configuration for CSIG

Port's LLDP local DB csig-capable object	Port's LLDP remote DB csig-capable object	Port's CSIG Receiver Log/Forward/Discard	Port's CSIG Transmitter Strip/Forward
NULL or CSIG-NO	Any	No CSIG Functions	No CSIG Functions
CSIG-COMPACT	NULL or CSIG-NO	Log CSIG tag error count Forward compact Discard wide tag packets	Strip compact CSIG tag if present and forward the packet
CSIG-COMPACT	CSIG-COMPACT	Log wide tag error count Forward compact Discard wide tag packets	Forward the packet with a compact CSIG tag
CSIG-COMPACT	CSIG-COMPACT-WIDE	Log wide tag error count Forward compact Discard wide tag packets	Forward the packet with a compact CSIG tag
CSIG-COMPACT-WIDE	NULL or CSIG-NO	Log CSIG tag error count Forward all	Strip any CSIG tag if present and forward the packet
CSIG-COMPACT-WIDE	CSIG-COMPACT	Log wide tag error count Forward all	Strip any wide CSIG tag if present and forward the packet Forward the packet with a compact CSIG tag
CSIG-COMPACT-WIDE	CSIG-COMPACT-WIDE	Forward all	Forward the packet with any CSIG tag

If the local port does not support CSIG but does support the UE Link Options LLDP TLV *csig-capable* object, then it MUST set the local UE Link Option LLDP TLV *csig-capable* object to CSIG-NO. If the local port supports only compact CSIG tags and also supports the *csig-capable* object, then it MUST set the local UE Link Option LLDP TLV *csig-capable* object to CSIG-COMPACT. If the local port supports both compact and wide CSIG tags and the *csig-capable* object, then it MUST set the local UE Link Option LLDP TLV *csig-capable* object to CSIG-COMPACT-WIDE.

Provided the system has enabled the use of LLDP for CSIG configuration, a port MUST perform the operations described in Table 4-7. A system MAY provide capabilities to override the LLDP configuration for CSIG on a port or switch; however, if the UE Link Option LLDP TLV *csig-capable* object is supported it MUST still be set consistent with the system configuration (i.e., if CSIG is disabled by a local system configuration then *csig-capable* is set to CSIG-NO even if the switch is capable of processing CSIG tags).

4.2.5.1.2 CSIG LLDP Example

When LLDP is enabled on a peering link, the devices at the link's endpoints MUST perform steps to reconcile their capabilities. Figure 4-6 depicts an example of state transitions for devices D1 and D2, both of which support the CSIG compact tag, connected by a peering link.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

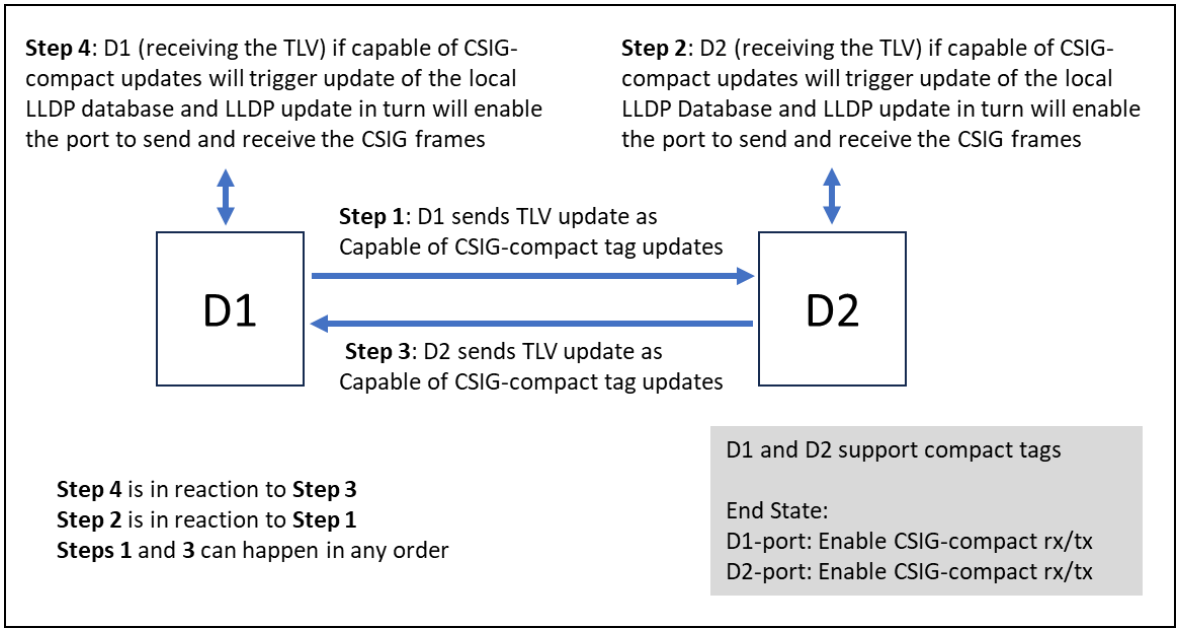


Figure 4-6 - CSIG LLDP Example

4.2.5.2 Tag Initialization

CSIG tag initialization occurs when a UE FEP inserts a CSIG tag (compact or wide) into a packet. The FEP's policy governs generation of CSIG tags in packets.

Informative Text:
UE fabric switches do not insert CSIG tags in packets that do not already carry them; thus, they also do not initialize CSIG tags.

Depending on the design and implementation of the CC algorithm and telemetry software in use at the FEP, the FEP's policy may:

- Insert CSIG tags in all of an application's packets (i.e., CSIG does not generate additional packets and gathers telemetry in a pre-existing application's packet stream).
- Insert CSIG tags in 1 in N of an application's packets (i.e., CSIG does not generate additional packets and gathers telemetry in a pre-existing application's packet stream).
- Insert CSIG tags in probe packets. A telemetry application generates probe packets for the express purpose of signal collection from UE fabric elements.

A FEP that inserts a tag in a packet MUST also initialize that CSIG tag's **csig.t** (type) and **csig.s** (signal) values (as specified in Table 4-8), and it MUST set the **csig.d** and **csig.r** (reserved) bit(s) to zero.

Table 4-8 - Signal Initial Values

T	Signal	Math Function	Initial Value
0	ABW	min	All '1b'
1	ABW/C	min	All '1b'

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

T	Signal	Math Function	Initial Value
2	Delay	max	All ‘0b’
3	nQD	max	All ‘0b’

4.2.5.3 Tag Transit

UE fabric switch elements perform the CSIG transit function.

The transit function entails:

- When the received CSIG tag’s **csig.d** bit is set to zero:
  - Perform *CSIG-Compare-and-Update()* with the received CSIG tag in accordance with the signal type **csig.t**
- When the received CSIG tag’s **csig.d** bit is set to one:
  - Do not update the CSIG tag; leave it unchanged when forwarding the packet. (See Section 4.2.4.1.8 for the full discussion of the **csig.d** bit and trimmed packets.)

Example *CSIG-Compare-and-Update()* sequence:

```
Compare-and-Update(pkt.csig.signal, pkt.csig.lm)

if (pkt.csig->t == “min-ABW[/C]”) {
  if (local_abw[/c] at packet egress port < pkt.csig.signal) {
    pkt.csig->signal = local_abw[/c] at packet egress port;
    pkt.csig->lm = local_lm;
  }
}
if (pkt.csig->t == “max-Delay”) {
  if (local_delay > pkt.csig.signal) {
    pkt.csig->signal = local_delay;
    pkt.csig->lm = local_lm;
  }
}
```

Note that in the pseudocode above, local values are quantized so that they are comparable to the quantized values in the received CSIG tag.

4.2.5.4 Strip and Forward

The *CSIG-Strip-and-Forward()* function at a switch strips the CSIG tag from a packet and forwards the packet without the tag. When a switch learns through LLDP that a peer switch on some link cannot process CSIG tags, that switch MUST invoke *CSIG-Strip-and-Forward()* on CSIG-tagged packets before forwarding them on that link to ensure that these packets are not dropped by the non-CSIG-capable receiving peer. This action is performed regardless of the value of the packet’s CSIG tag’s **csig.d** bit.

4.2.5.5 Tag Termination

The destination FEP consumes received CSIG tags by performing the termination (Term) function, thereby ending the forward path of the CSIG domain.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

The *CSIG-Term()* function does the following:

- Extracts the information carried in the CSIG tag (**csig.t, csig.s , csig.lm, csig.d** bit)
- For a packet arriving on a transport flow using receiver-based congestion control, presents the extracted CSIG tag information to the receiving FEP's transport layer (for onward delivery to the CC algorithm and/or receiver-based application)
- For a packet arriving on a transport flow using sender-based congestion control, presents the extracted CSIG tag information to the *CSIG-Reflect()* function in the receiving FEP's transport layer, to allow return of the CSIG tag information to the flow's sending FEP
- Strips the CSIG tag from the incoming packet

See Section 4.2.5.5.1 for a description of the *CSIG-Reflect()* function.

4.2.5.5.1 Reflection

When a CSIG-tagged packet from a source FEP arrives at a destination FEP that is CSIG-enabled, the *CSIG-Term()* function at the destination FEP extracts the CSIG tag information and passes it to the transport layer. If the transport layer is using a sender-based CC algorithm (such as NSCC), or other sender-based functions (such as multi-path load balancing) that need the information in the CSIG tag, the *CSIG-Reflect()* function at the transport layer MUST be enabled. The *CSIG-Reflect()* function populates the CSIG Reflection Header with information extracted by the *CSIG-Term() f*unction. The CSIG Reflection Header is returned to the source via an ACK or NACK packet generated by the transport layer. This enables the source FEP to receive feedback on path conditions indicated by the CSIG tag received at the destination. Section 3.6.9.4 describes the format of the CSIG Reflection Header as well as the **pds.ack_ccx_state** and **pds.nack_ccx_state** formats used for CSIG reflection.

Implementation of the *CSIG-Reflect()* function is REQUIRED in a CSIG-capable FEP.

Figure 4-7 shows the logical interactions among the *CSIG-Term()*, CMS, PDS, and *CSIG-Reflect()* functions. The detailed implementation of the *CSIG-Reflect()* function depends upon the congestion control algorithm in use. Detailed procedures for constructing the CSIG Reflection Header in the ACK_CCX and NACK_CCX fields, setting other transport header fields, and their use in particular congestion control algorithms (e.g., NSCC and RCCC) are specified in 3.6.9.4.

4.2.6 CSIG Tag Layout

The CSIG tag lies within an Ethernet frame's layer 2 headers. An EtherType unique to the deployment (4.2.4.1.1) identifies a CSIG-tagged frame.

When layer 2 encapsulation (e.g., IEEE Std 802.1ad or stacked tags) is not in use on a frame, the CSIG tag is always placed as the last tag in the layer 2 header. Tunneling of CSIG-tagged packets at layer 2, layer 3, or above is out of scope of this specification.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

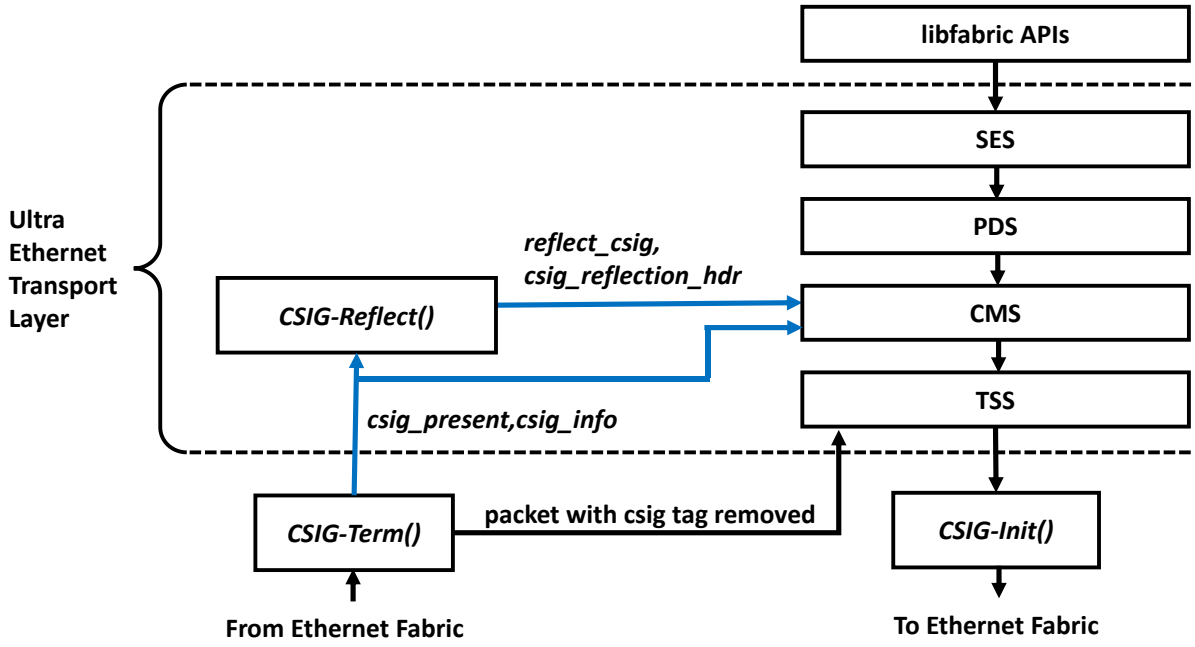


Figure 4-7 - CSIG Reflection Entity Interactions

Figure 4-8 shows the position of an Ethernet frame's CSIG tag among the fields of the frame's Ethernet and IEEE Std 802.1Q headers. In these examples, the CSIG tag is highlighted within the layer 2 Ethernet header.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.



Figure 4-8 - CSIG Tag Position in Ethernet Headers

To further exemplify that the CSIG tag is positioned as the last tag before the original EtherType, Figure 4-9 shows the position of a CSIG tag when using both IEEE Std 802.1AE and IEEE Std 802.1Q. The same relative position applies for both the compact and wide tag formats.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

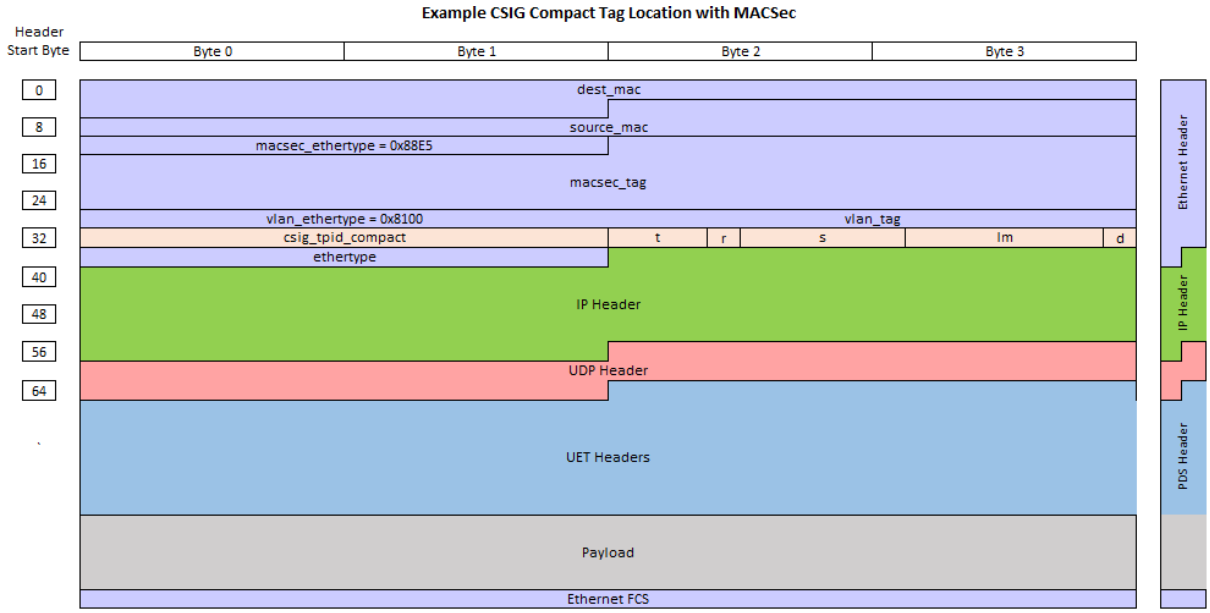


Figure 4-9 - Example CSIG Tag Position with MACSec

4.2.7 CSIG Profiles

We define two CSIG configuration profiles: a base profile, which describes the set of signals that all CSIG-capable devices MUST support, and an extended profile, which describes a further set of signals, any subset of which (including the null subset) CSIG-enabled devices MAY optionally support. We envision that the extended profile may grow over time to include further signals as the need arises. Table 4-9 summarizes the signals for these profiles.

4.2.7.1 CSIG Base Profile

All CSIG-enabled devices MUST support the base profile, which includes the following signals:

- *ABW*
- *ABW/C*
- *Delay*

These signals are useful inputs to the NSCC congestion control algorithm, for which UEC mandates support. They are also broadly useful telemetry signals for network performance analysis.

4.2.7.2 CSIG Extended Profile

The extended profile augments the base profile with further signal types and provides for extending CSIG with additional signal types in future revisions of this specification. CSIG-enabled devices MAY support any subset of these further signal types (including none of them):

- *nQD*
- *Experimental*

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

CSIG-capable devices that support any signals in the CSIG extended profile beyond those in the CSIG base profile MUST support indicating which of those extended-profile signals they support. The CSIG SAI specification describes interrogation of which extended-profile signals a CSIG-capable device supports.

Table 4-9 - CSIG Profiles

Profile	Signals Supported
Base	ABW, ABW/C, Delay
Extended (all of Base, and additionally, any subset)	nQD, Experimental

Informative Text:
UE devices that support CSIG MUST support the base profile.

4.2.8 CSIG Traffic Rate Measurement Interval

The CSIG ABW and ABW/C signal types involve the measurement of a traffic rate. Such a measurement is taken over some time interval. Because congestion control algorithms at end systems typically consume congestion signals measured at path-RTT granularity, CSIG-capable devices SHOULD choose an interval for traffic-rate measurement that is no longer than the unloaded RTT for the shortest-propagation-delay path within the fabric.

Comparisons of traffic rates measured at distinct switches and carried in CSIG tags are valid when switches along a CSIG-tagged packet's path use the same measurement interval. Toward ensuring that all switches along a path support at least one common measurement interval, a CSIG-capable switch SHOULD support a measurement time interval t of 102.4 µs and default to this value at boot. Further, a CSIG-capable switch SHOULD support as many values as an implementation permits of the measurement time interval t drawn from the range of values [128 ns, 1024 ms] with an increment of 128 ns. A switch SHOULD support the same measurement intervals for both the ABW and ABW/C signal types and MUST configure the same measurement intervals across all switches in the fabric.

A switch constrained by its arithmetic capabilities can choose to approximate the time interval. In such cases the time interval **MUST** be within the bounds of loss of precision during quantization.

Implementation Note:
For signals that measure a traffic rate (e.g., ABW and ABW/C), implementors should take note of the interaction between the spacing in time of CSIG-tagged packets within a flow and the time interval over which a device measures a traffic rate. For CSIG-tagged packets within a flow to obtain representative traffic rate measurements that cover traffic dynamics over time, it is essential they arrive at a device at least once per the measurement interval in use by that device. If the gap between successive CSIG-tagged packet arrivals at the device is significantly longer than the device's traffic rate measurement interval, the signal values written by that device into passing CSIG-tagged packets will not cover traffic volumes measured during such gaps. It is not generally possible to choose a traffic rate measurement interval at devices to ensure CSIG-tagged packets will arrive often enough to avoid

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

such gaps; if an application-limited flow sending CSIG-tagged packets from an end host goes idle on a long-lived transport connection for some significant period, such a gap may occur.

End hosts may also perform averaging over multiple CSIG signal values received in multiple packets over the course of an RTT or longer.

4.2.9 CSIG Signals

This section describes each signal, the algorithm for any associated computation, boundary conditions, and the signal's applicability in example use cases.

In the text that follows, we name each signal according to signal type and the comparison function—*min*() or *max*()—used in *CSIG-Compare-and-Update()* operations on tags containing that signal.

4.2.9.1 min(ABW)

When a source FEP sends a packet with a CSIG tag for signal type *min(ABW)*, and that packet traverses a path through the UE fabric, that packet's CSIG tag records the minimum absolute available bandwidth along the path: among all switches along that packet's path, the minimum quantized absolute available bandwidth in bytes per second (Bps) at any output physical port traversed by that packet. Both the unquantized and quantized absolute available bandwidth values are represented as integers.

4.2.9.1.1 Available Bandwidth Computation

The *ABW* signal uses a simple, time window-based computation to measure traffic rate and, thus, available bandwidth. Given the following definitions:

Table 4-10 - Available Bandwidth Computation Terms

Formula Term	Definition
t	input: measurement interval (seconds)
p	input: output port speed (Bps)
m	input: total traffic sent on output port during t (Bytes)
r	traffic rate (Bps)
A	available bandwidth (Bps, integer value)
i	quantized available bandwidth (bucket index, integer value)
$Q(A)$	quantization function for *ABW* signal type and tag width

Define the traffic rate, available bandwidth, and quantized available bandwidth as follows:

$$r = \frac{m}{t}$$

$$A = (p - r)$$

$$\mathrm{i} = \mathrm{Q}(\mathrm{A})$$

Update the *min(ABW)* signal value in the CSIG tag in the packet as follows:

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

$$\text{pkt} \rightarrow \text{minABW} = \min(i, pkt \rightarrow minABW)$$

All network devices in a CSIG domain SHOULD be configured with the same value for t and the same quantization function(s)—for each of the compact and/or wide CSIG tag formats they support—for use with the *min(ABW)* signal type. Undesirable behavior may result if measurement intervals t differ across devices.

4.2.9.1.2 Load Accounting

UE-compliant switches MUST support a mode of operation in which they exclude MAC-generated frames in per-port transmit load accounting. *min(ABW)* measurements are intended to include only data frames subject to congestion control by an originating FEP. Other MAC-generated frames (e.g., PAUSE, LLR, CBFC, and PFC frames) SHOULD NOT be included in per-port transmit load accounting. Switches MAY support other variations of transmit load accounting outside the scope of this specification.

4.2.9.1.3 UET Congestion Control Use Cases

A potential use case for the *min(ABW)* signal in UE congestion control concerns improving the effectiveness of multi-path load balancing. Both the NSCC and RCCC congestion control algorithms incorporate multi-path load balancing, whereby a sending FEP selects and writes an entropy value (EV) in each packet it sends, and switches include this EV in the input to the hash functions they use for ECMP path selection. Distinct flows may obliviously select EVs that direct their traffic to collide at the same output port of a switch. When other uncongested paths are available, such flow collisions cause otherwise avoidable congestion at that port's queue.

Current proposals for multi-path load balancing in UET, such as REPS, use the presence or absence of ECN marks on returning transport ACKs as an indication of whether a packet encountered congestion from a flow collision when sent with a particular EV. By efficiently informing sending FEPs of the minimum available bandwidth along a packet's path, the *min(ABW)* CSIG signal may allow sending FEPs to perform a finer-grained comparison of the merits of the paths selected by distinct EVs, such that a sending FEP may be able not only to avoid EVs that have experienced congestion, but also to select an EV preferentially for a path that offers comparatively more available bandwidth than other EVs' paths.

4.2.9.2 min(ABW/C)

The *min(ABW/C)* signal is similar to the *min(ABW)* signal, but its signal value is normalized by the output port capacity C in bytes per second. Both the unquantized and quantized normalized available bandwidth values are represented as integers.

In fabrics with heterogeneous port speeds, comparison of absolute available bandwidths measured at ports of different speeds does not reliably reveal which port is operating at a greater *fraction* of its capacity. Normalizing by capacity reliably identifies the port operating at the greatest fraction of its capacity, even when comparing across ports of different speeds.

4.2.9.2.1 Available Bandwidth Computation

The *min(ABW/C)* signal uses a simple, time window-based computation of normalized available bandwidth, represented as an integer value between 0 and 100.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Table 4-11 - Available Bandwidth /C Computation Terms

Formula Term	Definition
p	Input: output port speed (Bps)
A	Input: available bandwidth (Bps, integer value, from ABW signal value)
F	available fraction of capacity (integer percentage)
i	quantized available bandwidth (bucket index)
$Q(F)$	quantization function for *ABW/C* signal type and tag width

Define the traffic rate, available fraction of capacity, and quantized normalized available bandwidth as follows:

$$\mathrm{F} = 100\left(\frac{A}{p}\right)$$

$$\mathrm{i} = \mathrm{Q(F)}$$

Update the *min(ABW/C)* signal value in the CSIG tag in the packet as follows:

$$pkt \rightarrow minABWC = min(i, pkt \rightarrow minABWC)$$

All network devices in a CSIG domain SHOULD be configured with the same value for t and the same quantization function(s)—for each of the compact and/or wide CSIG tag formats they support—for use with the *min(ABW/C)* signal type. Undesirable behavior may result if measurement intervals t differ across devices.

4.2.9.2.2 Load Accounting

Outbound traffic measurements performed for the *min(ABW/C)* signal are subject to the same constraints as those applied to measurements for the *min(ABW)* signal, as described in Section 4.2.9.1.2.

4.2.9.2.3 UET Congestion Control Use Cases

A central problem in congestion control is prompt allocation of newly spare capacity on a link to flows that can use it. For example, when a flow releases its share of a bottleneck link, it is desirable for other flows that share the same bottleneck to rapidly claim the newly freed capacity. In this situation, a sender faces a trade-off. It can increase its rate slowly and conservatively by probing, which risks leaving capacity unused for multiple RTTs and lengthening flow completion time. Alternatively, it can increase its rate aggressively, and risk overshooting the spare capacity, which incurs congestive delay and packet losses.

NSCC incorporates a *fast_increase()* mechanism that reclaims spare capacity aggressively when path queueing delay is modest and many ACKs have returned in the recent past without an ECN mark, both of which indicate that a path is lightly loaded. While *fast_increase()* successfully helps flows reclaim spare capacity rapidly, it is heuristic in nature in the sense that it is oblivious to the explicit measure of spare capacity at a bottleneck on a path: it increases a sender's congestion window a constant two MTUs per returning ACK.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

The *min(ABW/C)* signal provides the NSCC sender with a direct, quantitative measure of the spare capacity at a path's bottleneck. This information allows NSCC to better match the increase in a flow's sending rate to spare capacity, and thus avoid undershoots and overshoots, both of which can lengthen flow completion times. Because the *min(ABW/C)* signal is normalized by an output port's line rate, it correctly relatively ranks the fractional spare capacities of output ports along a path, even when they run at heterogeneous port speeds.

4.2.9.3 max(Delay)

When a source FEP sends a packet with a CSIG tag for signal type *max(Delay)*, and that packet traverses a path through the UE fabric, that packet's CSIG tag records, among all switches along the packet's path, the maximum service delay in nanoseconds incurred by the packet at any one switch. Both the unquantized and quantized delay values are represented as integers.

As each switch computes its own intra-switch delay using only switch-local timestamps, there is no need for clock synchronization across switches in support of these measurements.

4.2.9.3.1 Per-Hop Delay Measurement Method

In a CSIG-capable fabric, at each hop along a path, the switch pipeline measures per-packet delay.

The queueing delay *D* experienced by a packet is given by:

$$D = t_2 - t_1$$

where t_2 is the local timestamp in nanoseconds when the start-of-packet (SOP) signal is sent by the output port's MAC, and t_1 is the local timestamp in nanoseconds when the SOP signal is received from the input port's MAC.

4.2.9.3.2 Timestamping Points in Switch Pipeline

Figure 4-10 shows the switch pipeline and the two points at which a switch SHOULD timestamp a packet's arrival and departure. The aim is for this delay measurement to capture the queueing delay experienced by the packet. IP refers to ingress pipeline processing, and EP refers to egress pipeline processing in a switch.

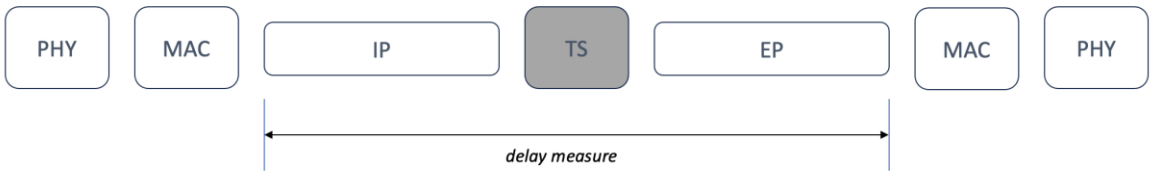


Figure 4-10 - Timestamping Points in Switch Pipeline

Measuring the interval between the start of the ingress pipeline and the end of the egress pipeline offers two principal advantages:

- It's straightforward to perform *CSIG-Compare-and-Update()* on the packet in these stages.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

- Timestamping in these stages includes only queueing and pipeline delays.

4.2.9.3.3 Point in Packet for Timestamping

SOP indicates the start-of-packet event at a switch port. A UE CSIG-capable switch MUST measure the delay experienced by a packet by timestamping the packet at the SOP event at the ingress port and at the SOP at the egress port, then taking the difference between the latter timestamp and the former one. This approach does not include any extraneous delay proportional to a packet's length in the measurement and is compatible with cut-through switch fabrics. A switch MAY subtract a fixed value from this measurement to account for the switch's uncongested forwarding delay.

4.2.9.3.4 UET Congestion Control Use Cases

The *max(Delay)* signal is a trailing indicator of congestion: a packet must traverse a switch's queue before a measurement of the delay incurred at that queue can be calculated. The value of this signal as an input to delay-based congestion control algorithms, such as NSCC, stems from its more targeted measurement of congestion than is possible with total end-to-end path delay. *max(Delay)* reveals the queueing delay experienced by a packet at a path's single most constrained bottleneck, whereas whole-path delay measurements, such as those used as input by NSCC, sum the queueing delays from *all* congested queues along a packet's path.

The more targeted congestion measurements enabled by *max(Delay)* can materially improve the bandwidth fairness of delay-based congestion control protocols. Previously published work [1] demonstrates that when two flows traverse the same congested queue, if one of the two flows encounters no other congested queues along its path and the other flow encounters one or more additional congested queues along its path, delay-based congestion control schemes that use whole-path delay measurements exhibit bandwidth unfairness between the two flows. Such congestion control schemes reduce a flow's sending rate in proportion to the amount that the flow's total path delay exceeds a target path delay.

Because the flow traversing multiple congested queues experiences greater total path delay, even though only a single queue along a path can be the most constrained bottleneck for any flow, its sending rate will be reduced proportionally more than that of the flow traversing only one congested queue. Poseidon, a delay-based congestion control algorithm that uses CSIG *max(Delay)*-like feedback [1] , improves fairness for such flow pairs by enabling the control loops for each of the two flows to respond to congestion only at that flow's single true bottleneck (most congested) queue, such that the flow that traverses multiple congested queues is not disproportionately penalized.

The *max(Delay)* signal should offer similar bandwidth fairness improvements to NSCC congestion control. Exploratory simulations of a version of NSCC modified to replace whole-path delay signals with CSIG *max(Delay)* signals demonstrate material bandwidth fairness improvements over baseline NSCC in scenarios in which multiple flows traverse a shared congested queue, while distinct flows among them traverse varying numbers of congested queues.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

4.2.9.4 max(nQD)

When a source FEP sends a packet with a CSIG tag for signal type *max(nQD)*, and that packet traverses a path through the UE fabric, that packet's CSIG tag will record the maximum normalized queue depth at packet dequeue encountered along the path: among all switches along that packet's path, the maximum queue depth (in bytes) normalized by the output port's speed (in bytes per second) at any one switch. Both the unquantized and quantized normalized queue depth values are represented as integer values in nanoseconds; note that queue depth in bytes normalized by port speed in bytes per second yields a unit of time.

In deployments with heterogeneous output port speeds, absolute queue depths at queues for output ports of different speeds are not directly comparable as indicators of the extent of congestion. For example, a shorter absolute queue depth in front of a slower link might take longer to drain than a longer absolute queue depth in front of a faster link. In such a case, a shorter absolute queue depth might indicate more severe congestion than a longer absolute queue depth. For this reason, to render queue depths across potentially heterogeneous port speeds comparable, the *max(nQD)* signal normalizes queue depth by output port speed.

max(nQD) is a leading indicator of congestion and corresponds to "expected delay" rather than "experienced delay": Queue depth is measured instantaneously at the point when a CSIG-tagged packet is dequeued for onward forwarding. In this sense, this signal is analogous to ECN marks, but with a multi-bit indication of the extent of queueing, as opposed to ECN's single-bit, thresholded queue depth indication.

Informative Text:
While normalizing by output port speed addresses heterogeneous port speeds, other considerations may arise in some deployments that make comparison of *nQD* values across switches problematic. For example, in a deployment that uses multiple traffic classes of different priorities, if CSIG-*nQD*-tagged packets travel in a traffic class at any priority other than the top priority, the drain rate for those packets from that traffic class's queue will vary over time, depending on the time-varying load in any higher-priority traffic class(es). As such, the correctness of the max(*nQD*) signal's value—accumulated along a path as an indicator of the point of greatest congestion along that path—relies on the absence of unpredictable, time-varying load in traffic classes of higher priority than the traffic class in which a CSIG-*nQD*-tagged packet travels.

4.2.9.4.1 Normalized Queue Depth Computation

A switch computes the local *nQD* value by normalizing the instantaneous queue depth at the time of dequeue by a base port speed.

```
// iQD: the instantaneous queue depth (in bytes) at the time of dequeue
// PortSpeed: the current port's speed
// BasePortSpeed: the slowest port speed in the network
PortSpeedRatio = PortSpeed / BasePortSpeed
PortSpeedRatioLog2 = log2(PortSpeedRatio)
```


Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

```
if (PortSpeedRatio_is_power_of_2)
      Adjusted_iQD = iQD >> PortSpeedRatioLog2
else
      Adjusted_iQD = iQD / PortSpeedRatio
Local_CSIG_nQD = Quantization (Adjusted_iQD)
Update CSIG(nQD) Tag when Local_CSIG_nQD is greater than the current "Signal
Value"
```

When all ports in a fabric run at the same speed, *PortSpeedRatio* is effectively 1, and *PortSpeedLog2Ratio* is 0.

In UE deployments with heterogeneous output port speeds, the *BasePortSpeed* can be set to 100 Gbps. Relative to that base value, for port speeds of 100/200/400/800/1600/3200 Gbps, we obtain *PortSpeedRatioLog2* values of 0/1/2/3/4/5 respectively.

This simple shift logic does not suffice for future non-power-of-2 speeds in a network with heterogeneous port speeds. Consider a 300 Gbps port speed in a network with heterogeneous port speeds as an example: division by 3 logic would be required with a *BasePortSpeed* of 100 Gbps.

Refer to sections 4.2.4.1.4 and 4.2.4.1.6 for the *CSIG-Quantization()* algorithms for the compact and wide CSIG tag formats, respectively.

Informative Text:
nQD may be useful as a replacement for ECN, as *nQD* logically is a multi-bit ECN signal. *nQD* may be usable as a load balancing signal, a congestion control signal, and a telemetry signal.

Use of the *nQD* signal type in deployments with application traffic in multiple traffic classes of different priorities is outside the scope of this specification. Future versions of this specification may address the use of the *nQD* signal in such deployments.

4.2.10 CSIG Deployment Considerations

As specified in Section 4.2.4, two CSIG tag formats are defined: a compact (4-byte) format and a wide (8-byte) format. A global configuration setting for the entire UE fabric and set of FEPs indicates whether the preferred tag format for the UE CSIG domain is the compact format or the wide format. All FEPs SHOULD use this global configuration setting to determine whether to insert a compact- or wide-format CSIG tag in sent packets to collect congestion information from the UE fabric, such that there will not be a mix of compact and wide CSIG-capable paths in the same UE CSIG domain. In some cases, operators may deploy a mix of CSIG-capable and non-CSIG-capable switches in the same UE fabric. A global configuration setting for the entire UE fabric and set of FEPs will indicate whether CSIG is enabled for the entire fabric or disabled for the entire fabric. This setting SHOULD be enabled only when all switches in the UE fabric support CSIG, such that the fabric will not mix paths that support CSIG feedback-guided congestion control and paths that do not support CSIG-feedback-guided congestion control.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

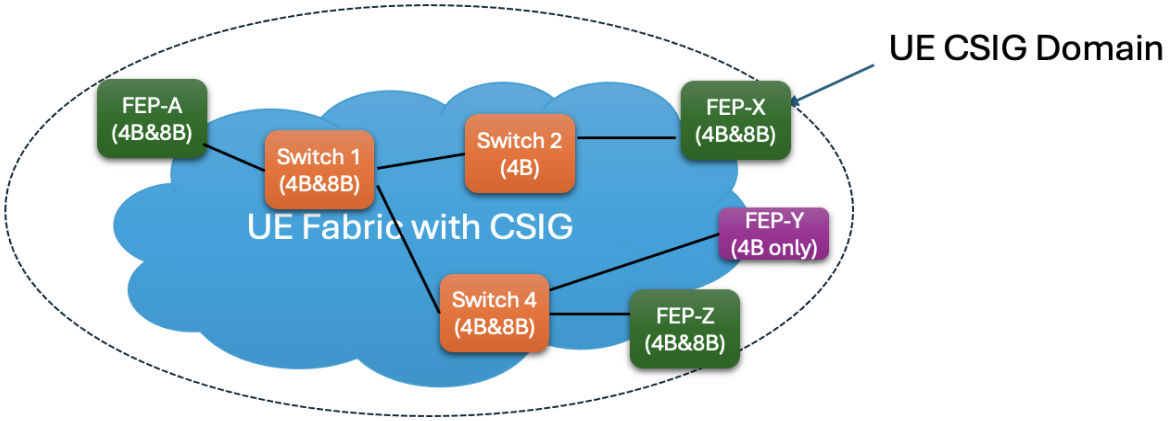


Figure 4-11 - CSIG Domain

Informative Text:
In some cases, such as in multi-tenancy, when an operator deploys a mix of switches in the UE fabric, some of which support only the compact tag format and some of which support both the compact and wide tag formats, some applications MAY use CSIG with wide tags on those paths where switches support them and others MAY use compact tags. In these cases, those flows using wide tags benefit from the greater precision and/or signal type variety of wide tags on paths where they are available. Such behavior is outside the scope of this specification, as are the mechanisms and policies FEPs use to configure the choice of compact or wide tags in sent traffic. Whether a congestion control scheme works correctly when CSIG-compact-tagged flows and CSIG-wide-tagged flows share a bottleneck depends on the specifics of the congestion control scheme.

In some cases, when an operator deploys a mix of switches in the UE fabric, some of which support CSIG and some of which do not, the operator MAY wish to use CSIG as feedback in congestion control on those paths that support CSIG, while CSIG will not be used on those paths that do not support CSIG. Such behavior is outside the scope of this specification. This specification contemplates only the scenarios in which congestion control for all flows uses CSIG feedback and in which congestion control for no flows uses CSIG feedback. Whether a congestion control scheme works correctly when CSIG-feedback-enabled flows and non-CSIG-feedback-enabled flows share a bottleneck depends on the specific details of the congestion control scheme.

4.2.10.1 Deployment of New Signal Types

New CSIG signal types may be introduced in the future. When a new signal type is introduced in a deployment, it is recommended that use of that signal type be enabled only after all switches and FEPs support that signal type, as indicated in global configuration state. When support for a new signal type is heterogeneous across FEPs and switches in a deployment, how FEPs determine when to use the new signal type is outside the scope of this specification.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Whether a switch supports processing of a new signal type is NOT discoverable through LLDP. Such discovery is not necessary, as a switch receiving a packet with a CSIG tag for an unsupported signal type can still parse and forward the packet correctly. Instead, the central management entity is responsible for configuring switches that receive a packet with a CSIG tag for an unsupported signal type to either 1) *CSIG-Strip-and-Forward()* the packet or 2) *CSIG-Forward()* the packet while preserving the CSIG tag unchanged. A deployment will specify which one of a few policies to follow when a switch receives a tag for a signal type it does not implement (e.g., *Forward()* without updating the tag or *CSIG-Strip-and-Forward()*).

4.2.11 References

[1] W. Wang, M. Moshref, Y. Li, G. Kumar, T. E. Ng, N. Cardwell and N. Dukkipati, "Poseidon: Efficient, Robust, and Practical Datacenter {CC} via Deployable {INT}," in 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23), 2023.

[2] IEEE Std 802.1Q-2022, "IEEE Standard for Local and Metropolitan Area Networks – Bridges and Bridged Networks," 2022. [Online]. Available: https://ieeexplore.ieee.org/document/10004498.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

5 UE Link Layer

The Link Layer Working Group is responsible for defining UE specifications that operate at and support the link layer. These optional-to-implement specifications enhance performance and reliability of the link layer while also simplifying configuration. The features defined by the UE link layer specifications can operate with, and in some cases work as an alternative to, existing Ethernet link functionality.

The following sections describe the UE link layer specifications in detail:

1. Section 5.1 Link layer retry (LLR), which reduces the impact on latency when a packet is lost due to imperfect links. LLR requires only a single-link round-trip time to determine loss and restart the flow of packets.
2. Section 5.2 Credit-based flow control (CBFC) is an alternative mechanism to priority-based flow control (PFC) flow control. Rather than sending PFC indications when a receive buffer is approaching full, a CBFC receiver periodically sends buffer space updates (credits) to its link partner. This allows the partner to preferentially send traffic on traffic classes with available buffer space.
3. Section 5.3 Link negotiation allows UE devices supporting optional UE link layer features to automatically determine if the link partner is capable of operating with the same mutually supported UE link layer features. Link negotiation defines procedures and managed objects that operate on top of existing IEEE Std 802.1AB (LLDP) implementations that are widely deployed.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

5.1 Link Layer Retry (LLR)

The impact of delay on efficiency becomes more significant as network link speeds, utilization, and scale increase. Latency-sensitive workloads are negatively impacted by the delays created from the traditional end-to-end approach of retransmitting lost packets. Local error handling at the link layer has proven valuable in scale-out HPC networks, such as those used in exascale systems. The UE specification provides a link layer retry (LLR) capability for Ethernet.

The LLR mechanism is frame-based. Each frame passed to the transmit LLR from the MAC client is assessed for eligibility for LLR. If the MAC client does not desire LLR for the frame, or if it is otherwise classified as LLR-ineligible, then the frame is sent as a standard Ethernet frame. If the frame is LLR-eligible, then the LLR TX assigns a sequence number and stores the frame in a replay buffer for possible retransmission if the link partner does not receive the frame.

The sequence number for each LLR-eligible frame passed from the LLR to the MAC is placed into the preamble prepended by the MAC. The link partner sends acknowledgements (LLR_ACKs) for LLR-eligible frames that are successfully received, and these LLR_ACKs are used to free space in the replay buffer. Negative acknowledgements (LLR_NACKs) are used to signal when a missing LLR-eligible frame is detected (due to receipt of a subsequent LLR-eligible frame). Timeouts are used to guarantee that LLR-eligible frames will be replayed if an LLR_NACK is lost or if a replay is corrupted and there is no LLR_ACK to indicate progress.

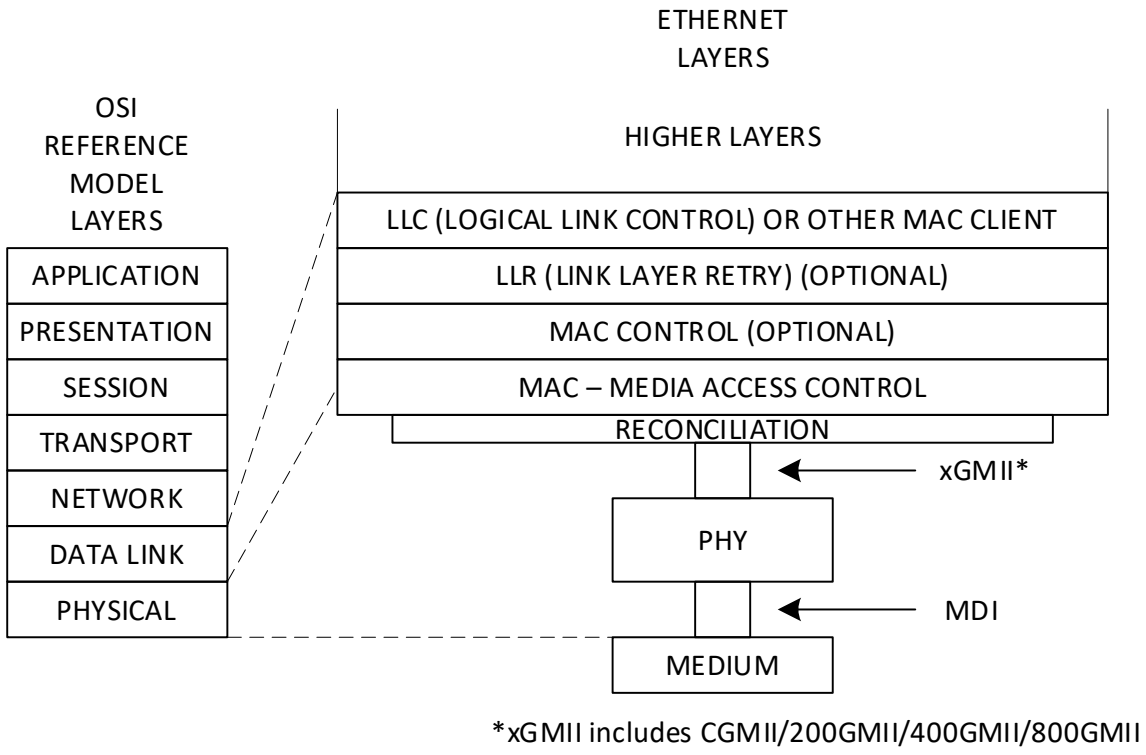


Figure 5-1 - Architectural position of LLR.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

The receive MAC delivers LLR-eligible frames and LLR-ineligible frames to the LLR layer. For each LLR-eligible frame, the MAC relays the sequence number of the LLR-eligible frame extracted from the preamble. The LLR checks that the sequence number of the LLR-eligible frame is the next expected sequence number. The LLR discards LLR-eligible frames that are not expected and returns an LLR_NACK to the link partner to indicate that a frame has been lost.

During normal operation, the LLR receiver periodically sends LLR_ACKs that acknowledge the most recent successfully received frame. When an out-of-sequence frame is received, a single LLR_NACK is sent to indicate the error. No further LLR_ACKs or LLR_NACKs are sent until the frame with the expected sequence number is successfully received.

LLR provides lossless link operation for data from the Ethernet LLC/MAC client layer. In the standard IEEE layering model, LLR is located between the LLC/MAC client layer and MAC control layer. LLR is above the MAC control layer because the sending of pause frames from the MAC control layer is not blocked by LLR while it is performing a replay operation. However, it should be noted that the reception of pause frames does not stop the replay of previously sent data frames by LLR and acts only on new data frames (LLR-ineligible or LLR-eligible) from the LLC/MAC client layer. In this respect, showing the LLR above the MAC control layer is slightly misleading, and care should be taken when implementing the pause mechanism. Priority-based flow control (PFC) frames interact with LLR in the same way as pause frames.

The correct MAC control pause behavior is achieved by bypassing the MAC control request and information interfaces around the LLR. Requests to send pause frames are made to the MAC control

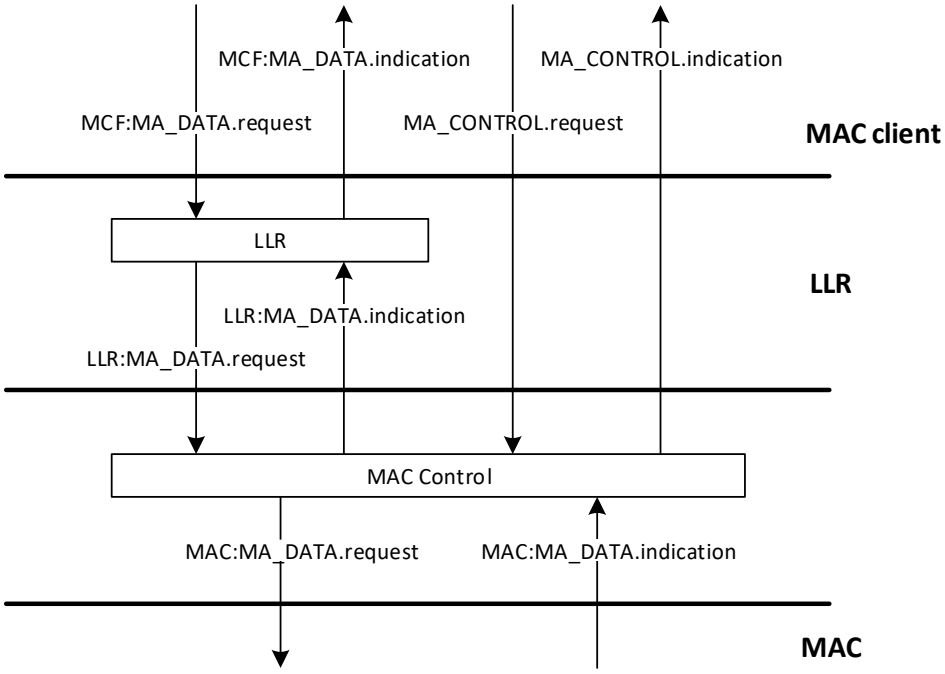


Figure 5-2 - MAC Control interface connectivity.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

layer directly from the LLC/MAC client and are made irrespective of the LLR's current state. Similarly, received pause frame indications are passed around the LLR to the LLC/MAC client and do not affect the LLR operation.

Because received pause frames stop only new MAC client data being passed to the LLR and do not stop LLR replay data, it is important that the maximum amount of data held in the replay buffer not be much greater than the bandwidth-delay product for the link. If the LLR replay buffer were to contain significantly more data than the bandwidth-delay product, then pause would not operate as expected. A configuration is provided in the LLR to limit the maximum amount of data held in the replay buffer.

Informative Text:
The process for buffer sizing, accounting for the link bandwidth-delay product, is outside the scope of this specification and should be performed by the implementer.

5.1.1 Frame structure

The UE LLR specification introduces a new modified frame preamble, usable with both IPv4 and IPv6 addressing to support link layer retry (LLR). When UE features such as LLR are not in use, UE link frames are indistinguishable from standard IEEE 802.3 Ethernet frames.

All frames MUST begin with the 8-byte UE link frame preamble defined in Section 5.1.1.1.

5.1.1.1 UE Link frame preamble

The UE link frame preamble is shown in Table 5-1. LLR-ineligible frames carry the standard preamble. LLR-eligible frames are indicated by the 0x7 in Byte 1. For those frames, the LLR sequence number is placed in bytes 1-3, using a 20-bit sequence number. With minimum 64-byte frames at 800 Gbit/sec, this allows for a round-trip time of over 500 µs. The longest RTT is expected to be much less than 5 µsec, but this field width allows for link speed increases in the future.

The remaining bytes (4-6) match those of LLR-ineligible frames.

Table 5-1 - MII Format for UE Link Frame Preamble

Frame type	Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
LLR-ineligible	0xFB	0x55	0x55	0x55	0x55	0x55	0x55	0xD5
LLR-eligible	0xFB	[3:0] = 0x7 [7:4] = frame_seq[19:16]	frame_seq[15:8]	frame_seq[7:0]	0x55	0x55	0x55	0xD5

The UE link frame preamble (start block) uses the same block type field as specified in IEEE Std 802.3-2022, clause 82.2.3.3, with sync header set to 2'b10 and control block field of 0x78, as shown in Table 5-2. Both the block type field and octets D_1 through D_7 are transmitted from least-significant bit to most-

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

significant bit. For example, an LLR-eligible frame with frame_seq[19:0]=0x4A2D3 would result in the following preamble being transmitted:

10_00011110_11100010_01000101_11001011_10101010_10101010_10101010_10101011

Table 5-2 - 64B/66B Block Format for UE Link Frame Preamble

Input Data	Sync	Block Payload							
Bit Position:	0 1	2							65
Control Block Format		Block Type Field							
S_0 D_1 D_2 D_3 D_4 D_5 D_6 D_7	10	0x78	D_1	D_2	D_3	D_4	D_5	D_6	D_7
LLR-ineligible	10	0x78	0x55	0x55	0x55	0x55	0x55	0x55	0xD5
LLR-eligible	10	0x78	<3:0> = 0x7 <7:4> = frame_seq <19:16>	frame_seq <15:8>	frame_seq <7:0>	0x55	0x55	0x55	0xD5

5.1.2 Interface modifications

The transmit path interface from the MAC client to the LLR (MCF:MA_DATA.request) requires two additional fields to support UE link operation.

Table 5-3 - MAC Client to LLR Transmit Path Additional Fields

Field Name	Description
frame_desire_llr	Indicates the desire for the frame to be sent as LLR-eligible. Values: TRUE (LLR-desired), FALSE (Default transmission)
frame_bad	Indicates that the frame payload data is bad. This is used to ensure that errors are correctly propagated to the link partner but will not trigger an LLR replay.

The transmit path interfaces from the LLR to the MAC control (LLR:MA_DATA.request) and from the MAC control to the MAC (MAC:MA_DATA.request) each require three additional fields to support LLR operation.

Table 5-4 - LLR to MAC Control to MAC Transmit Path Additional Fields

Field Name	Description
frame_type	Indicates the type of frame. Values: LLR_ELIGIBLE, LLR_INELIGIBLE
poison_fcs	Indicates that the MAC should send the frame with a poisoned FCS. A poisoned FCS ensures that errors are correctly propagated to the link partner but will not trigger an LLR replay.
llr_field	The LLR-related control fields to be placed into the preamble of LLR_ELIGIBLE frames by the MAC; consists of a 20-bit sequence number for the frame.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

The LLR_INELIGIBLE frame type is used to send IEEE Std 802.3 format frames without any LLR protection. Pause and PFC frames are always sent as LLR_INELIGIBLE. The LLR_ELIGIBLE frame type is used to send IEEE 802.3 format frames with LLR protection.

The receive path interfaces from the MAC to the MAC Control and from the MAC Control to the LLR each require three additional fields to support LLR operation.

Table 5-5 - MAC to MAC Control to LLR Receive Path Additional Fields

Field Name	Description
frame_type	Indicates the type of frame. Values: LLR_ELIGIBLE, LLR_INELIGIBLE
fcs_status	Indicates the status of the frame's FCS. Values: GOOD, POISONED, BAD An FCS status of BAD allows the LLR to send an LLR_NACK to trigger a replay. An FCS status of POISONED will not trigger the sending of an LLR_NACK but will ensure that the error is propagated.
llr_field	The LLR related control fields that are extracted from the preamble of LLR_ELIGIBLE frames by the MAC.

The receive path interface from the LLR to the MAC client (MCF:MA_DATA.indication) requires one additional field to support UE link operation.

Table 5-6 - LLR to MAC Client Receive Path Additional Field

Field Name	Description
frame_bad	Indicates that the frame payload data is bad. Frames received with either a poisoned or bad FCS will set the frame_bad flag TRUE.

5.1.3 LLR Operation

The LLR has two modes of operation: OFF and ON. The OFF mode of operation disables LLR functionality and removes its effects from the IEEE layering model so that a port behaves as an IEEE 802.3 standard port. The ON mode of operation enables the LLR mechanism and provides lossless link-level behavior for LLR-eligible frames, while providing IEEE 802.3 standard behavior for LLR-ineligible frames.

When the LLR mode is OFF, all frames are passed through the TX LLR from the MAC client to the MAC control layer and are not stored in the replay buffer or assigned a sequence number. When the LLR mode is OFF, all frames are passed through the RX LLR from the MAC control layer to the MAC client layer without any frames being filtered.

When the LLR mode is ON, the LLR mechanism is active. All LLR-ineligible frames are passed through the TX LLR from the MAC client to the MAC control layer and are not stored in the replay buffer or assigned a sequence number, while all LLR-eligible frames are handled as described in the following sections. When the LLR mode is ON, unexpected (sequence number missed or sequence number duplicated) LLR-eligible frames are filtered by the RX LLR, while expected LLR-eligible frames and LLR-ineligible frames (not terminated by the MAC control layer) are passed through the RX LLR to the MAC client layer without being filtered.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

5.1.3.1.1 Control Ordered Set

The LLR uses special Control Ordered Sets (CtlOS) to send control messages to the link partner when it is in the ON mode. The PCS periodically inserts CtlOS into the data stream in a manner that is oblivious to the data frames being sent (i.e., CtlOS can be sent pre-emptively). This limits the maximum latency for sending control messages to the link partner and provides a guaranteed bandwidth for those control messages.

Note that support for these CtlOS must be present in the MAC and in the PCS and RS of the physical layer. This includes supporting the additional Control Ordered Sets described in Table 5-7, with indicated data fields. That data needs to be communicated from/to the LLR layer. Lastly, the PCS must support the existence of CtlOS in the block stream, including during frame data. See the UE PHY specification for additional details.

There are four types of CtlOS that the LLR needs to send and receive:

Table 5-7 - LLR Control Ordered Sets

CtlOS Type	Description
LLR_ACK	LLR frame acknowledgement.
LLR_NACK	LLR frame negative acknowledgement.
LLR_INIT	Sent to initialize the next_rx_seq state in the link partner.
LLR_INIT_ECHO	Sent to indicate that an LLR_INIT CtlOS has been received and processed and that the station is ready to receive LLR frames.

In addition to the standard 802.3-defined Sequence ordered sets, the UE LLR specification defines a Control Ordered Set (CtlOS) for purposes of LLR messaging. The corresponding 64B/66B block formats are shown in Table 5-8—which matches the format of Figure 82-5 - 64B/66B block formats in IEEE Std 802.3-2022.

Table 5-8 - UE LLR Link Control Ordered Set 64B/66B Block Format

Input Data	Sync	Block Payload									
Bit Position:	0 1	2									65
Control Block Format		Block Type Field									
O_0 D_1 D_2 D_3 D_4 D_5 D_6 D_7	10	D_0	D_1	D_2	D_3	O_0	D_4[7:4]	D_5	D_6	D_7	
LLR_ACK	10	0x4B	0x01	ack_nack_seq <19:4>		0x6	ack_nack_seq <3:0>	0	0	0	
LLR_NACK	10	0x4B	0x02	ack_nack_seq <19:4>		0x6	ack_nack_seq <3:0>	0	0	0	
LLR_INIT	10	0x4B	0x03	init_seq <19:4>		0x6	init_seq <3:0>	init_data<15:0>		0	
LLR_INIT_ECHO	10	0x4B	0x04	init_seq <19:4>		0x6	init_seq <3:0>	init_data<15:0>		0	

The UE LLR Link Control Ordered Set feature uses the same block type field as specified in IEEE 802.3-2022, clause 82.2.3.9, with sync header set to 2’b10, and control block field of 0x4B, and O code set to 0x6. Octets D_1 through D_3 and D_5 through D_7; the O code; and the upper four bits of octet D_4 are all

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

transmitted from least-significant bit to most-significant bit. For example, an LLR_ACK with ack_nack_seq[19:0]=0x12345 would result in the following CtlOS being transmitted:

10_11010010_10000000_01001000_00101100_01101010_00000000_00000000_00000000.

For the LLR signaling CtlOS, D_1 is set to the CtlOS message type. Four different message types are supported and are identified by the value of D_1: LLR_ACK, LLR_NACK, LLR_INIT, and LLR_INIT_ECHO (see Section 5.1.9 for more detail on the purpose of these messages). The 20-bit message sequence number is carried in D_2 (sequence number bits <19:12>), D_3 (sequence number bits <11:4>), and D_4[7:4] (sequence number bits <3:0>). In addition, the LLR_INIT and LLR_INIT_ECHO types carry init data bits <15:8> in D_5 and bits <7:0> in D_6. For example, an LLR_INIT with init_seq[19:0]=0x8A72C and init_data[15:0]=0x4DE1 would result in the following CtlOS being transmitted:

10_11010010_11000000_01010001_01001110_01100011_10110010_10000111_00000000.

LLR_ACK/LLR_NACK Control Ordered Set should be transmitted periodically (according to ctlos_target_spacing) to ensure an adequate frequency of communication between stations, with the appropriate sequence number corresponding to the most recently processed received frame. Implementations may deviate (for any given interval between successive Control Ordered Set) from the target spacing, but the LLR_TX MUST transmit the CtlOS LLR_ACK / LLR_NACK at a rate between a maximum rate of once every 400 bytes and a minimum rate of once every 17296 bytes[16] (unless the ACK/NACK transmit state machine is in the NACK_SENT state). To mitigate the impact on per-frame switch latency, a CtlOS SHALL NOT be inserted among the first 256 bytes of a frame. Similarly, two CtlOS within the same frame MUST be spaced apart by at least 2 KB of data so that underrun can be managed on cut-through switches.

5.1.4 LLR configuration

The LLR REQUIRES certain configuration registers to be accessible to management software for the purposes of configuring and controlling its operation. The llr_mode_local and llr_mode_remote are enabled from the link negotiation process. The llr_mode_local is set when negotiation determines that both ends of a link support LLR. The llr_mode_remote is set when negotiation receives indication from the remote end of the link that its llr_mode_local has been set to ON.

Table 5-9 - LLR Configuration Registers

Variable name	Description
llr_mode_local	Configures the mode of operation of the LLR. Once this is set, an LLR_INIT may be received. Values: OFF, ON
llr_mode_remote	Configures the mode of operation of the LLR. Once this is set, LLR_INIT may be sent. Values: OFF, ON

[16] Allowing a minimum rate of once every 17296 bytes provides a tolerance of up to 400+512 bytes beyond the maximum configurable CtlOS_target_spacing value of 16384 bytes. The extra 912 bytes represents the cumulative impact of implementation factors that can delay the specific point at which CtlOS is inserted into the stream, such as insertion of a CtlOS from another function as well as avoiding a CtlOS within the first 256 bytes of a frame.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Variable name	Description
outstanding_seq_max	A configuration for the maximum permitted value of outstanding_seq. Values: 0 to 524288 The absolute maximum permitted value for outstanding_seq_max is 524288, which is equivalent to consuming half of the total sequence number space. An implementation is permitted to support a significantly lower maximum for outstanding_seq_max than the absolute maximum value. Setting this register to a value that is greater than an implementation's own maximum is not allowed.
outstanding_data_max	A configuration for the maximum permitted value of outstanding_data. This SHOULD be set to the bandwidth delay product for the link to ensure correct operation of the pause and PFC mechanisms.
replay_timer_max	A configuration to set the value of replay_timer at which replay_timer_expired is set and a replay is initiated. Values: 0 to 65535 ns with a resolution better than 10 ns.
replay_ct_max	A configuration to set the maximum number of times a replay is performed before the LLR mechanism gives up and enters the FLUSH state. A value of 255 (i.e., all-ones) is used to indicate that there is no maximum. Values: 0 to 255
pcs_lost_status_timer_max	The value at which the pcs_lost_status_timer is considered to have expired. See pcs_lost_status_timer and pcs_lost_status_timer_expired. Values: 0 to 4.29 s with a resolution better than 100 ns.
data_age_timer_max	The value at which the data_age_timer is considered to have expired. See data_age_timer and data_age_timer_expired. Values[1]: 0 to 4.29 s with a resolution better than 100 ns.
llr_init_behavior	Configures the behavior of the TX LLR when the TX state machine is in the INIT state. Values: DISCARD, BLOCK, BEST_EFFORT
llr_flush_behavior	Configures the behavior of the TX LLR when the TX state machine is in the FLUSH state. Values: DISCARD, BLOCK, BEST_EFFORT
re_init_on_discard	Configures the behavior in the event of an LLR replay failure. Values: TRUE, FALSE When the value is TRUE, then the LLR will attempt to re-initialize. When the value is FALSE, then the LLR will wait for management intervention.
ctlos_target_spacing	A configuration indicating the target number of bytes between transmission of successive LLR_ACK / LLR_NACK CtlOS. Values 400 to 16384 with suggested value 2048.
Note: 1. The minimum usable value is implementation dependent and affected by link round trip time.	

5.1.5 LLR transmit path operation

The LLR TX path is controlled by the state machine shown in Figure 5-3.

In addition to the state machine, the LLR TX path maintains the following variables to control its operation, as shown in Table 5-10:

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

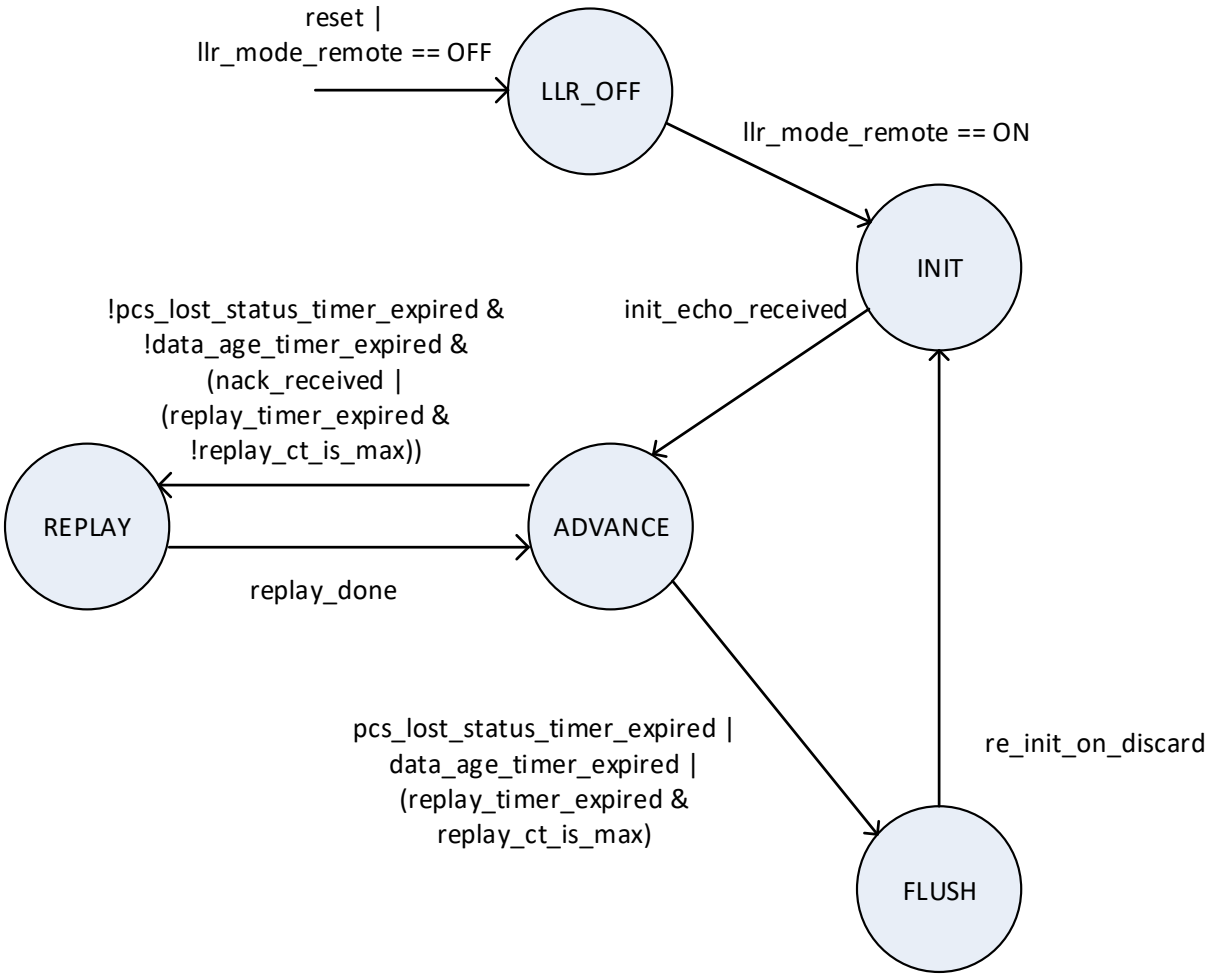


Figure 5-3 - LLR transmit state machine

Table 5-10 - LLR TX Path Variables

Variable name	Description
init_echo_sent	The init_echo_sent variable is used by the ACK/NACK transmit state machine shown in Figure 5-4 and pulses each time a valid LLR_INIT_ECHO Control Ordered Set is sent (see 5.1.6).
tx_seq	The sequence number to be used in the frame_seq field (of the frame preamble) for the next LLR frame to be transmitted. Set to 0 whenever llr_mode_remote is changed. Incremented by 1 at the end of each LLR frame that is transmitted when llr_mode_remote is ON. The value wraps from all ones to all zeros. A 20-bit variable with values: 0 to 1048575
acked_seq	Stores the sequence number acknowledged in the most recent successfully received LLR_ACK or LLR_NACK. Set to all ones whenever llr_mode_remote is changed. Set to init_seq-1 when an LLR_INIT CtlOS is sent. A 20-bit variable with values: 0 to 1048575

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Variable name	Description
outstanding_seq	Contains the number of unacknowledged frames that have been transmitted. The variable is determined as follows: outstanding_seq = ((tx_seq – 1) – acked_seq) mod 1048576 A 20-bit variable with values: 0 to 524287
outstanding_data	Contains the total amount of data (in bytes) for the unacknowledged frames.
nack_received	Set to TRUE when an LLR_NACK is received. Set to FALSE when an LLR_ACK is received. Also set to FALSE when in the LLR_OFF or INIT state or upon entry to the REPLAY state.
replay_timer	A timer used to initiate a replay in the event that an LLR_NACK is lost or a replay is not successful. The rules governing the operation of the replay_timer are as follows: 1) The replay_timer is set to zero and held at zero whenever outstanding_seq is zero. 2) The replay_timer is activated (if not already) when the end of a new LLR frame is transmitted. 3) The replay_timer is activated (if not already) when the end of a replay frame is transmitted. 4) The replay_timer is set to zero when an LLR_ACK is received that is acknowledging some frames held in the replay buffer. If the outstanding_seq is non-zero after the LLR_ACK has been processed, the replay timer is activated; otherwise it is held at zero (see rule 1). 5) The replay_timer is set to zero when an LLR_NACK is received. The replay_timer is then held at zero until a replay commences (it will be reactivated at the end of the first replay frame as described in rule 3). If an LLR_ACK is received before the replay commences, then the replay timer is activated again if the resulting outstanding_seq is non-zero (see rule 4). 6) If the replay_timer reaches replay_timer_max, it has expired. It is then set to zero and held at zero until a replay commences (it will be reactivated at the end of the first replay frame as described in rule 3). If an LLR_ACK is received before the replay commences, then that LLR_ACK should be acknowledging some unacknowledged frames in the replay buffer. The replay timer will then be activated again if the resulting outstanding_seq is non-zero (see rule 4). The replay timer SHOULD have a resolution better than 10 ns. The size of the replay_timer should allow the timer to count 10x the expected LLR round trip time for its intended use case. As an example, for links up to 500 m, a 16-bit nanosecond timer is sufficient.
replay_timer_expired	Set to one when the replay_timer reaches replay_timer_max. It is set to zero when the state machine enters the REPLAY or FLUSH states. It is also set to zero if an LLR_ACK is received that is acknowledging some unacknowledged frames in the replay buffer.
replay_ct	A count of the number of times a replay has been performed without receiving an LLR_ACK for some unacknowledged frames. It is incremented by one each time that the REPLAY state is exited and nack_received is zero. It SHOULD also be set to zero whenever an LLR_NACK is received, since that will also acknowledge additional frames. It is set to zero whenever an LLR_ACK is received for some unacknowledged frames.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Variable name	Description
	It is set to zero in the INIT state.
replay_ct_is_max	Set to TRUE when the replay_ct is greater than or equal to the replay_ct_max configuration value. If the replay_ct_max value is all ones, then replay_ct_is_max is never set TRUE.
replay_done	Set to TRUE when all of the frames in the LLR replay buffer are transmitted. If nack_received is set to TRUE, replay_done SHOULD be set to TRUE. It is not necessary or productive to continue the replay once an LLR_NACK has been received.
PCS_status	A Boolean value that indicates the status of the link. Set to TRUE when the link is up and set to FALSE otherwise.
pcs_lost_status_timer	A timer used to progress the LLR transmit state machine into the FLUSH state when the PCS_status has been FALSE for too long (even if there are no unacknowledged frames). The rules governing the operation of the pcs_lost_status_timer are as follows: 1) The pcs_lost_status_timer is set to zero and stopped whenever the LLR state is LLR_OFF, INIT, or FLUSH, or whenever the PCS_status is TRUE. 2) The pcs_lost_status_timer is activated whenever the PCS_status is FALSE. 3) If the pcs_lost_status_timer reaches pcs_lost_status_timer_max, it has expired. It is then held at pcs_lost_status_timer_max until the LLR transmit state machine enters the FLUSH state, at which point it will be set to zero. The pcs_lost_status_timer SHOULD have a resolution better than 100 ns. The size of the pcs_lost_status_timer should allow the timer to count to 4 seconds. A value of 0 causes immediate expiration of the timer. Support for non-zero values is optional.
pcs_lost_status_timer_expired	Set to TRUE when the pcs_lost_status_timer equals pcs_lost_status_timer_max.
data_age_timer	A timer used to time how long data has been resident in the replay buffer without any forward progress being made on acknowledging the data. The data_age_timer is an approximate measure of the age of the data in the replay buffer without requiring an implementation to timestamp each frame. Expiry of the data_age_timer will result in the data held in the replay buffer being discarded. The purpose of the data_age_timer is to discard data that is too old, regardless of the number of replays that may have been performed. In particular, it can be used if the replay_ct_max is set to all ones. The rules governing the operation of the data_age_timer are as follows: 1) The data_age_timer is set to zero and stopped whenever outstanding_seq is zero. 2) The data_age_timer is activated (if not already) when the end of a new LLR frame is transmitted. 3) The data_age_timer is set to zero when an LLR_ACK or LLR_NACK is received that is acknowledging some frames held in the replay buffer. If outstanding_seq is non-zero after the LLR_ACK/LLR_NACK has been processed, then the data_age_timer is activated; otherwise it is held at zero (see rule 1).

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Variable name	Description
	4) If the data_age_timer reaches data_age_timer_max, it has expired. It is then held at data_age_timer_max until the LLR state machine enters the FLUSH state or is reset by rules 1 or 3 being met. The data_age_timer SHOULD have a resolution better than 100 ns. The size of the data_age_timer SHOULD allow the timer to count to 4 seconds.
data_age_timer_expired	Set TRUE when the data_age_timer is equal to data_age_timer_max.
LLR_status	Indicates the link status pertaining to LLR-eligible frames. When the llr_mode_remote is OFF, LLR_status takes the same value as PCS_status (i.e., PCS_status is passed through the LLR to the management agent unchanged). When the llr_mode_remote is ON, LLR_status takes the value of TRUE when the state machine is in the ADVANCE or REPLAY states, and it takes the value of FALSE when in all other states. The LLR link stays up during LLR replays that are occurring over a PCS loss of status (loss of alignment) event.

The TX LLR handles LLR-desired frames passed to it from the MAC client in the manner described below. Three styles of behavior are permitted:

- DISCARD: The LLR takes LLR-desired frames from the MAC client and discards them without passing them to the MAC control layer. These discards are counted in the tx_discards counter.
- BLOCK: The LLR does not take any LLR-desired frames from the MAC client and effectively blocks data transmission of LLR-desired frames from the higher layers. This mode of operation guarantees lossless data delivery but might have undesirable effects on the higher layers.
- BEST_EFFORT: The LLR takes LLR-desired frames from the MAC client and passes them to the MAC control layer in the same manner as if the llr_mode_remote were OFF. That is, the frames are not placed in the LLR replay buffer or assigned a sequence number, and the frame_type is set to LLR_INELIGIBLE.

When the TX state machine is in the LLR_OFF state, then the TX LLR uses the BEST_EFFORT style for TX LLR-desired frames. When the TX state machine is in the INIT state, then the TX LLR uses the llr_init_behavior configuration to control how it handles TX LLR-desired frames. When the TX state machine is in the FLUSH state, then the TX LLR uses the llr_flush_behavior configuration to control how it handles TX LLR-desired frames. When the TX state machine is in the ADVANCE or REPLAY state, then the TX LLR handles LLR-desired frames as follows:

- The TX LLR will not accept a new LLR-desired frame for transmission from the MAC client if the state machine is not in the ADVANCE state.
- The TX LLR will not accept a new LLR-desired frame for transmission from the MAC client if the outstanding_seq value is greater than or equal to outstanding_seq_max (where outstanding_seq_max is <= 524288). Note that an implementation is not required to support 524288 unacknowledged frames and may have an outstanding_seq_max value that is considerably less than 524288. The maximum number of outstanding_seq that an

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

implementation supports will depend on the implementation's sublayer delays, the intended link length to be supported, and the minimum frame size that the implementation supports.

- The TX LLR will not accept a new LLR-desired frame for transmission from the MAC client if the outstanding_data value is greater than or equal to the outstanding_data_max. This is to ensure the correct operation of the pause and PFC mechanisms by limiting the maximum amount of data that can be replayed to the same amount as the bandwidth-delay product for the link.
- The TX LLR will not accept a new LLR-desired frame for transmission from the MAC client if there is not sufficient space in the replay buffer to store the entire LLR-eligible frame. The LLR is not permitted to truncate LLR-eligible frames when it has run out of replay buffer space. The size of the replay buffer is implementation-dependent and should be sized based on the implementation's sublayer delays and intended link length to be supported.
- When an LLR_NACK is received, the nack_received variable is set to TRUE. The TX LLR completes the transmission of any frame (LLR-eligible or LLR-ineligible) that it is in the middle of transmitting. It then transitions to the REPLAY state, where a replay of the LLR-eligible frames held in the replay buffer is performed. Transmitting new LLR-ineligible frames may be allowed during the REPLAY state. This is recommended particularly for time-sensitive LLR-ineligible frames. Expiry of the replay_timer if the replay_ct is less than replay_ct_max will also transition the state machine to the REPLAY state.

Expiry of the replay_timer when the replay_ct equals replay_ct_max (if replay_ct_max is not 255 (i.e., all-ones)) will transition the state machine to the FLUSH state.

When in the FLUSH state, the LLR discards all frames held in the replay buffer without transmitting them. The state machine will then transition to the INIT state if the re_init_on_discard control is TRUE.

5.1.6 Transmission of LLR_ACKs/LLR_NACKs

LLR_ACKs and LLR_NACKs are sent using Control Ordered Set (see Table 5-8). Transmitted LLR_ACKs and LLR_NACKs contain a sequence number (ack_nack_seq) that is the sequence number of the most recent successfully received LLR-eligible frame. A successfully received LLR-eligible frame is defined as one that has an fcs_status of GOOD or POISONED and for which expected_frame is TRUE.

Once an LLR_INIT_ECHO has been sent by the local LLR, LLR_ACKs are periodically sent to the link partner to indicate the sequence number of the most recent successfully received LLR-eligible frame.

Reception of an LLR-eligible frame with an unexpected sequence number that is not a duplicate (missing_frame per Table 5-11, regardless of the value of fcs_status), or reception of an LLR-eligible frame with the expected sequence number and an fcs_status of BAD, will result in the sending of a single LLR_NACK. No further LLR_ACKs or LLR_NACKs will be sent until an LLR-eligible frame is received with the expected sequence number and an fcs_status of GOOD or POISONED. At that point, the sending of LLR_ACKs is resumed.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Implementation Note:
If a receiver drops an LLR-eligible frame and would also drop its retransmission (e.g., due to exceeding the MTU), the LLR protocol expects the receiver to treat the original frame as correctly received to prevent retransmission.

When the ACK/NACK transmit state machine is in the SEND_ACKS state, the LLR will send an LLR_ACK that contains next_rx_seq – 1 whenever there is an opportunity to send a Control Ordered Set (subject to the CtlOS transmission priority rules).

When the ACK/NACK state machine is in the SEND_NACK state, the LLR will send a single LLR_NACK at the next opportunity (subject to the CtlOS transmission priority rules), then transition to the NACK_SENT state.

No LLR_ACKs or LLR_NACKs are sent in the NACK_SENT state.

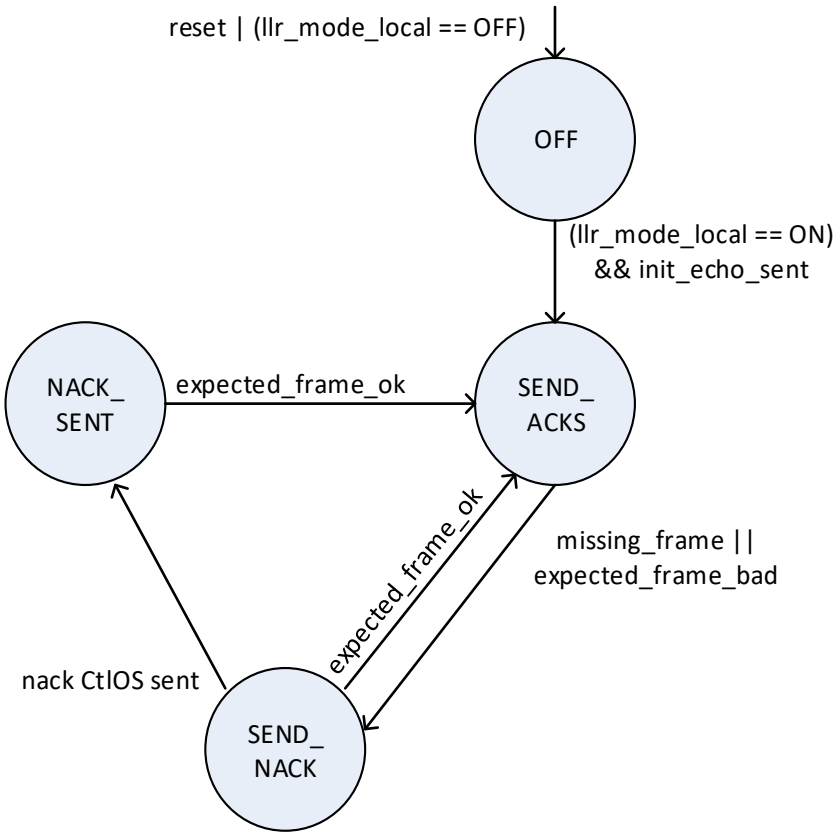


Figure 5-4 - ACK/NACK transmit state machine.

5.1.7 LLR receive path operation

The LLR receive path maintains the variables shown in Table 5-11 to control its operation.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Table 5-11 - LLR RX Path Variables

Variable name	Description
init_echo_received	The init_echo_received variable is used by the LLR transmit state machine shown in Figure 5-3 and pulses each time a valid LLR_INIT_ECHO Control Ordered Set is received (see 5.1.9.1.2).
next_rx_seq	The expected sequence number of the next LLR-eligible frame to be received. It is set to the value contained in the LLR_INIT Control Ordered Set when an LLR_INIT Control Ordered Set is received; it is incremented by one each time a LLR frame is received that has an fcs_status of GOOD or POISONED and for which expected_frame is TRUE.
next_rx_seq_vld	Indicates that the value in next_rx_seq is valid. Set to FALSE whenever the ACK/NACK transmit state machine enters the OFF state. Set to TRUE when llr_mode_local is ON and an LLR_INIT Control Ordered Set has been received.
frame_seq	The sequence number for the LLR-eligible frame that is being processed. A 20-bit variable with values: 0 to 1048575
expected_frame	Is TRUE when the sequence number of a received LLR-eligible frame is the next expected sequence number. The variable is determined as follows: expected_frame = next_rx_seq_vld && (frame_seq == next_rx_seq)
expected_frame_ok	Pulses TRUE when an LLR-eligible frame is received with an fcs_status of GOOD or POISONED and expected_frame is TRUE.
expected_frame_bad	Pulses TRUE when an LLR-eligible frame is received with an fcs_status of BAD and expected_frame is TRUE.
duplicate_frame	Is TRUE when the sequence number of a received LLR-eligible frame indicates that the frame has already been successfully received. (I.e., when the frame_seq is 'behind' the next_rx_seq) duplicate_frame = next_rx_seq_vld && !expected_frame && ((next_rx_seq – frame_seq) <= 524288)
missing_frame	Is TRUE when the sequence number of a received LLR-eligible frame indicates that a frame has been lost. (I.e., when the frame_seq is 'ahead' of next_rx_seq) missing_frame = next_rx_seq_vld && !expected_frame && !duplicate_frame

When llr_mode_local is ON, LLR-eligible frames are filtered by the RX LLR if expected_frame is FALSE.

5.1.8 Received ACK/NACK processing

The ack_nack_seq of received LLR_ACKs (or LLR_NACKs) should either match an unacknowledged frame that is held in the replay buffer or be equal to acked_seq. This can be done by applying the following equation:

```
ack_nack_seq_ok = (((tx_seq -1) – ack_nack_seq) mod 1048576 <= 524288) &&
                  ((ack_nack_seq – acked_seq) mod 1048576 < 524288)
```

If ack_nack_seq_ok is FALSE, then the ack_nack_seq value is an error and the rx_ack_nack_seq_error counter is incremented.

If ack_nack_seq_ok is TRUE and ack_nack_seq is not equal to acked_seq then the LLR_ACK/LLR_NACK is acknowledging some frames in the replay buffer. The replay buffer can remove all the frames up to and

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

including the frame whose sequence number is the ack_nack_seq. The variable acked_seq is then loaded with ack_nack_seq.

If an LLR_NACK is received, the nack_received variable is set to TRUE. When an LLR_ACK is received, the nack_received variable is set to FALSE. The nack_received variable is also set to FALSE when the state machine is in the INIT state or upon entry to the REPLAY state.

5.1.9 Control Ordered Set transmission and reception

5.1.9.1.1 LLR_INIT CtlOS

The LLR_INIT CtlOS is sent periodically when the LLR transmit state machine is in the INIT state. This allows for the loss of the LLR_INIT or LLR_INIT_ECHO CtlOS due to link errors. The CtlOS contains an init_seq value that is equal to the tx_seq variable. When an LLR_INIT CtlOS has been sent, a subsequent LLR_INIT CtlOS is sent after at least four times the number of ctlos_target_spacing bytes have been transmitted. This provides other CtlOS an opportunity to be transmitted while allowing for the loss of the LLR_INIT or LLR_INIT_ECHO CtlOS due to link errors. The init_data field may optionally be used by implementations to compute the link round trip time. The receiver MUST echo that value back, so that it MAY carry a timestamp or identifier for determining the link delay.

The use of init_data by the sender is optional; its function is opaque to the receiver. A sender can use it to determine link round-trip time or can use other means. Alternatively, the outstanding_data_max could be set based on the bandwidth delay product for the link plus the PFC activation times, as described in IEEE Std 802.1Q-2022 Annex N, to ensure correct operation of the PAUSE and PFC mechanisms.

When an LLR_INIT CtlOS is received, the LLR receiver sets its next_rx_seq variable to the value in init_seq and sets the next_rx_seq_vld variable to TRUE. An LLR_INIT_ECHO CtlOS is also scheduled for transmission, echoing back the init_seq and the init_data from the LLR_INIT.

5.1.9.1.2 LLR_INIT_ECHO CtlOS

The LLR_INIT_ECHO CtlOS is sent when llr_mode_local is set to ON and the LLR receives an LLR_INIT CtlOS from the link partner. The LLR_INIT_ECHO CtlOS contains a copy of the init_seq and init_data values that were received in the LLR_INIT CtlOS. When an LLR_INIT_ECHO CtlOS is received, the value contained in init_seq is compared to tx_seq. If the values are equal, the init_echo_received is pulsed, which will transition the LLR transmit state machine out of the INIT state. Loss of LLR_INIT_ECHO CtlOS is tolerated. During initialization, LLR_INIT CtlOS is sent repeatedly, and an LLR_INIT_ECHO CtlOS is sent in response to each LLR_INIT CtlOS.

5.1.9.1.3 LLR_ACK CtlOS

The LLR_ACK CtlOS is sent when the ACK/NACK state machine is in the SEND_ACKS state. The LLR_ACK CtlOS contains the ack_nack_seq value that is equal to next_rx_seq – 1.

When an LLR_ACK CtlOS is received, it is processed as described in Section 5.1.8. If an LLR_ACK is lost, the next LLR_ACK will update the ack_seq, allowing the send buffer to be properly emptied. Thus a lost

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

LLR_ACK might stress the send buffer capacity, but as a rare event it should not affect performance significantly.

5.1.9.1.4 LLR_NACK CtlOS

The LLR_NACK CtlOS is sent when the ACK/NACK state machine is in the SEND_NACK state. The LLR_NACK CtlOS contains the ack_nack_seq value that is equal to next_rx_seq – 1.

When an LLR_NACK CtlOS is received, it is processed as described in Section 5.1.8. If an LLR_NACK is lost, the transmitter continues sending new frames (rather than promptly starting retransmission). That data will be discarded until the replay_timer expires and replay is initiated.

5.1.9.1.5 Control Ordered Set transmission priority

LLR Control Ordered Set is selected for transmission based on the priority shown in Table 5-12:

Table 5-12 - Control Ordered Set Transmission Priority

Transmission Priority	Control Ordered Set
Highest	LLR_INIT_ECHO
	LLR_INIT
Lowest	LLR_ACK or LLR_NACK

There is no priority difference between LLR_ACKs and LLR_NACKs, as the LLR should never be attempting to send an LLR_NACK at the same time as an LLR_ACK. If credit-based flow control is also active, arbitration between CBFC CtlOS transmission and LLR CtlOS transmission should employ an algorithm that avoids starvation of either. To that end, round robin is recommended.

5.1.10 Error propagation

There are certain error scenarios where the LLR must indicate that the data in an LLR-eligible frame is bad without provoking the link partner to send an LLR_NACK and trigger a replay. These scenarios include cut-through switch architectures and the occurrence of multi-bit RAM errors (e.g., in the LLR replay RAM). In these cases, an LLR replay will not result in the delivery of correct data, as the data stored in the replay buffer is irrecoverably corrupted.

To provide the ability to indicate that the data in an LLR-eligible frame is bad without triggering a replay, the MAC poisons the FCS of transmitted LLR-eligible frames. The FCS is poisoned by applying an XOR of 0xAAAAAAAA. Specifically, follow the procedure of IEEE Std 802.3-2022 “3.2.9 Frame Check Sequence (FCS) field” except replace “e)” with:

> e) The odd terms of the bit sequence are complemented, and the result is the CRC.

The LLR sets the poison_fcs flag on the transmit interface to the MAC to indicate that it wishes the FCS to be poisoned. When the MAC receives an LLR-eligible frame with a poisoned FCS, it indicates this by setting the fcs_status field on the receive interface to POISONED.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

An example frame, with the well-known 01-80-C2-00-00-01 (as a point of reference) as its destination address and with a good FCS (0x0d 0x7f 0x0f 0x58) is as follows:

```
0x01 0x80 0xc2 0x00 0x00 0x01 0x00 0x0a 0x35 0x55 0x45 0x43 0x00 0x2e 0xc7
0x11 0x18 0xba 0x0e 0x82 0x93 0x3a 0x75 0xa3 0x30 0xed 0x84 0xc1 0x7b 0x4b
0x2f 0xe7 0xf3 0xc9 0x22 0x9d 0x03 0xf8 0xe2 0x9a 0x02 0xea 0x0b 0x3b 0x8a
0x5b 0x02 0x41 0x6c 0xbb 0x16 0x89 0xc3 0xb6 0x4b 0x10 0x62 0xbe 0xdf 0xbe
0x0d 0x7f 0x0f 0x58
```

The same frame with a poisoned FCS (0xa7 0xd5 0xa5 0xf2) is as follows:

```
0x01 0x80 0xc2 0x00 0x00 0x01 0x00 0x0a 0x35 0x55 0x45 0x43 0x00 0x2e 0xc7
0x11 0x18 0xba 0x0e 0x82 0x93 0x3a 0x75 0xa3 0x30 0xed 0x84 0xc1 0x7b 0x4b
0x2f 0xe7 0xf3 0xc9 0x22 0x9d 0x03 0xf8 0xe2 0x9a 0x02 0xea 0x0b 0x3b 0x8a
0x5b 0x02 0x41 0x6c 0xbb 0x16 0x89 0xc3 0xb6 0x4b 0x10 0x62 0xbe 0xdf 0xbe
0xa7 0xd5 0xa5 0xf2
```

Octets are transmitted in the order written (left to right). The lower (least-significant) bit of each octet is transmitted first. The upper (most-significant) bit of each octet is transmitted last.

LLR-eligible frames received with an fcs_status of POISONED are treated by the RX LLR in the same manner as LLR-eligible frames received with an fcs_status of GOOD: An LLR-eligible frame with an expected sequence number and an fcs_status of POISONED will not trigger a replay and will increment next_rx_seq.

5.1.11 Counters

The counters shown in Table 5-13 are defined for LLR.

Table 5-13 - LLR counters

Counter	Description
tx_init_ctl_os	A count of the number of LLR_INIT Control Ordered Sets transmitted.
tx_init_echo_ctl_os	A count of the number of LLR_INIT_ECHO Control Ordered Sets transmitted.
tx_ack_ctl_os	A count of the number of LLR_ACK Control Ordered Sets transmitted.
tx_nack_ctl_os	A count of the number of LLR_NACK Control Ordered Sets transmitted.
tx_discard	A count of the number of LLR-eligible frames discarded by the LLR TX when the TX state machine is in the INIT state and the llr_init_behavior is set to DISCARD, or when the TX state machine is in the FLUSH state and the llr_flush_behavior is set to DISCARD.
tx_ok	A count of the number of LLR-eligible frames transmitted with a good FCS.
tx_poisoned	A count of the number of LLR-eligible frames transmitted with a poisoned FCS.
tx_replay	A count of the number of times that the transmitter exited the REPLAY state (i.e., completed a replay operation).
rx_init_ctl_os	A count of the number of LLR_INIT Control Ordered Sets received.
rx_init_echo_ctl_os	A count of the number of LLR_INIT_ECHO Control Ordered Sets received.
rx_ack_ctl_os	A count of the number of LLR_ACK Control Ordered Sets received.
rx_nack_ctl_os	A count of the number of LLR_NACK Control Ordered Sets received.
rx_ack_nack_seq_error	A count of LLR_ACK/LLR_NACK sequence number errors. The counter is incremented each time an LLR_ACK or LLR_NACK is received for which ack_nack_seq_ok is FALSE.
rx_ok	A count of the number of LLR-eligible frames received with a good FCS.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Counter	Description
rx_poisoned	A count of the number of LLR-eligible frames received with a poisoned FCS.
rx_bad	A count of the number of LLR-eligible frames received with a bad FCS.
rx_expected_seq_good	A count of the number of LLR-eligible frames received with a good FCS that had the expected sequence number.
rx_expected_seq_poisoned	A count of the number of LLR-eligible frames received with a poisoned FCS that had the expected sequence number.
rx_expected_seq_bad	A count of the number of LLR-eligible frames received with a bad FCS that had the expected sequence number.
rx_missing_seq	A count of the number of LLR-eligible frames received that had a sequence number that indicated a missing LLR-eligible frame in the sequence (irrespective of FCS status).
rx_duplicate_seq	A count of the number of LLR-eligible frames received that had a duplicate sequence number (irrespective of FCS status).
rx_replay	A count of the number of times that the receiver has detected the start of a replay.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

5.2 Credit-based Flow Control

This specification defines the protocols, procedures, and message formats that enable credit-based flow control (CBFC) on a per-priority basis on a UE full duplex link (see Section 6.1.1.2).

CBFC is an alternative to the IEEE 802.1Q-2022 [6] standard priority-based flow control (PFC), which is intended to eliminate frame loss due to buffer congestion at the receiver. When either CFBC or PFC are enabled on a link, the IEEE 802.3 [1] PAUSE mechanism must not be enabled. The choice to use PFC or CBFC should be based on the need for lossless packet delivery and the relative advantages of each mechanism (see 5.2.2).

While CBFC can replace PFC, the protocol is designed to allow for possible simultaneous operation with PFC to manage multiple chip resources.

CBFC is an optional link layer feature in a UE network defined for an individual full-duplex point-to-point connection.

5.2.1 Lossless Packet Delivery Use Cases

Network applications require reliable packet delivery. Ethernet fabrics can be lossless or best effort. In best-effort fabrics, reliable packet delivery is guaranteed by an end-to-end reliable transport protocol, but the hop-by-hop links are best effort where packet drops due to congestion are allowed.

The UE Transport (UET) layer is defined to provide end-to-end reliable packet delivery in a best-effort network by utilizing the following components:

- Packet retransmission from the source.
- Support for out-of-order packet arrival at the receiver.
- Congestion control to limit packet drop probability in switches.

Congestion and the resulting lack of buffer space in switches is the main reason for packet drops in best-effort Ethernet networks.

In a lossless fabric, the hop-by-hop propagation of a packet is lossless via a suitable link-level flow control scheme. At the Link Layer, lossless packet transmission between link partners can be controlled by permitting packet transmission only if there is buffer space available at the receiver by utilizing link-level flow control. PFC and CBFC are two ways of achieving lossless packet delivery on a hop-by-hop basis to achieve a lossless Ethernet fabric. Lossless fabrics remove the need for end-to-end retransmissions and the delays associated with them.

Link-layer lossless packet delivery on a per-priority basis can also be useful in many situations. Small networks and lower loads can benefit from link-level flow control by simplifying network management and endpoint configuration and their buffer requirements. Link-level flow control may be preferable in certain network topologies such as Torus or Dragonfly as well HPC and AI applications with short messages. Link-level flow control may also be preferable for certain traffic classes with short control or command messages at a low rate.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

By providing CBFC on a per-traffic-class basis, congestion control resources and buffer space can be used only for the traffic classes that can benefit from link-level flow control.

5.2.2 CBFC and PFC Relative Advantages

CBFC has several advantages over standard PFC, including:

- For guaranteed delivery, CBFC can enable more lossless VCs with the same amount of burst absorption buffer.
- The sender knows the credit usage by each VC and can use this information for scheduling, load balancing, and adaptive routing.
- CBFC is not as sensitive as PFC to cable length, frame size, or response time of the sender. If cable delay is underestimated, PFC can have buffer overflows and dropped packets, but with CBFC the link would only be underutilized.
- For virtual channels that do not require full throughput, CBFC can simply allocate fewer credits and thus require less buffer space than PFC.

PFC still has certain advantages over CBFC, which include:

- XOFF-XON flow control is simpler to implement than credit-based flow control.
- PFC can enable more sharing and efficient usage of burst absorption buffer across ports.
- PFC is better able to manage multiple resources within a priority class.
- PFC provides full use of the link bandwidth (no messaging overhead) when no flow control events are occurring.

5.2.3 CBFC Feature List

CBFC supports the following features:

- Support for up to 32 VCs per port.
- Flexible assignment of lossless and best effort VCs.
- Lossless packet delivery for each lossless VC.
- Enables receiver port buffer sharing across VCs.
- Forward progress guarantee (deadlock avoidance) for each VC (see Section 5.2.5.2).
- Configurable credit units.
- Resiliency to packet loss and CBFC message loss.
- CBFC and PFC together can manage multiple switch resources.
    - CBFC can be used to manage one resource (e.g., the main input buffer).
    - PFC can be used to manage another resource such as a second buffering point and/or another per-packet resource.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

5.2.4 CBFC Overview

5.2.4.1 General Overview

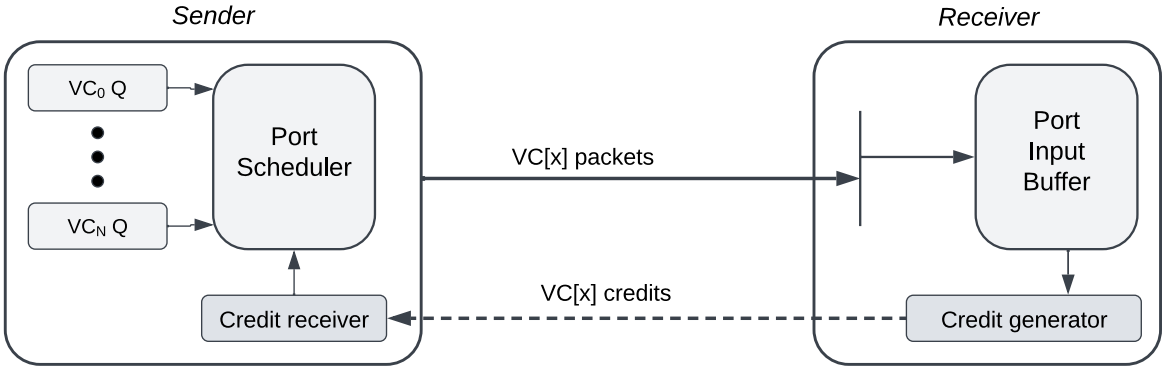


Figure 5-5 - Basic Packet Data and Credit Update Sequence

The sender keeps track of the available buffer space at the receiver in units of credits, and its packet scheduler is allowed to schedule a packet for transmission from a lossless VC queue only when enough buffer space, in credits, is available at the receiver. CBFC messages are used for the return of credits from the receiver to the sender. Credit generation at the receiver is based on port buffer availability in the receiver.

5.2.4.2 Cyclic Counters for Tracking Credits with Resiliency

Cyclic counters that always increment, with "wrap-around," are used for resiliency while tracking credits at both the sender and receiver. These cyclic counters wrap around at the modulus of the width of the counter. For example, if a counter is implemented as 10-bit counter with values 0-1023, when incrementing by N, the new value would be:

$$C_{new} = (C_{old} + N) \bmod 2^{10}$$

To reliably compare two cyclic counter values to determine which is greater, the two counters must have a difference of less than half the counter's maximum value. This is illustrated in Figure 5-6 for an example of a 10-bit counter with the value 768. A second counter with a value in the range 256-767 is less than 768. A counter in the range 769-1023 or 0-255 is greater than 768. Note that the range of 0-255 can represent the values 1024-1279. If one of two counters is always guaranteed to be greater than the other, then the full range can be used.

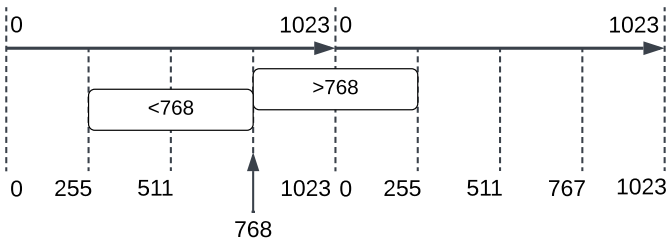


Figure 5-6 - Cyclic Counter Example

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

The basic CBFC credit mechanism has two counters:

1. The number of credits “consumed” (CC) is incremented as packets are transmitted at the sender and placed into the receiver’s data buffer. Both sender and receiver track the number of credits consumed.
2. The number of credits “freed” (CF) by the receiver is incremented as packets are drained out of its input buffer. As credits are freed by the receiver, the receiver must inform the sender of the number of freed credits to allow the sender to transmit additional packets.

In addition to the CC and CF counters, the sender uses per-port and per-VC credit limits (CL) to limit the packets on lossless VCs sent to the receiver to only the amount it can absorb into its buffer. The credit limits are initialized to the number of credits available at the receiver before data transmission starts. The credit limits are used only at the sender.

Both the sender and receiver track both credits consumed and credits freed in cyclic counters that are periodically synchronized between the link partners. By always incrementing the counter values, a lost credit freed message as well as lost packets, which can result in credit leakage, can be automatically accounted for with the next synchronization message.

5.2.5 CBFC Operation

5.2.5.1 Overview

CBFC is a link-layer function defined only for a pair of full-duplex MACs connected by one point-to-point link. shows an example of CBFC within a standard Ethernet Physical Layer model. Although CBFC is defined mainly at the data link layer, it requires some modifications in the standard MAC, RS, and PCS sublayers to correctly forward CBFC CtlOS messages between the link partners. See the UE PHY specification for additional details regarding the changes needed to these sublayers.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

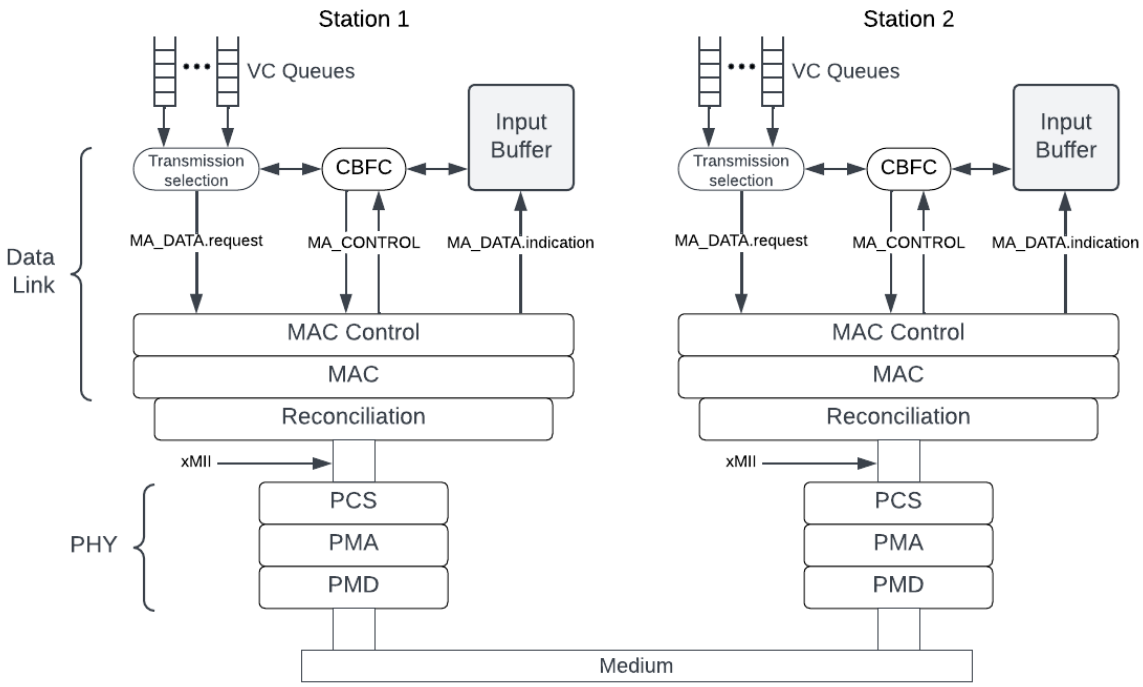


Figure 5-7 - Architectural Position of CBFC in Ethernet Functional Layer Model

5.2.5.2 Configuration Parameters and Initialization

Several configuration parameters must be initialized at both the sender and receiver with compatible values for correct operation. These configuration parameters are shown in Table 5-14.

Table 5-14 - Configuration Parameters and Initialization

Configuration Parameter	Description
CreditSize	The number of bytes represented by each credit. This must be a function of the size of the receiver's buffer allocation unit, or "cell size."
NumVCs	The number of virtual channels supported on the link. 1 ≤ NumVCs ≤ 32.
TotalCredits	The total number of credits available at the receiver. Must be set to a positive value based on the receiver's buffer size.
VC_CreditLimit[x]	The maximum number of credits allowed for VC[x].
PktOvhd	The number of bytes required on a per-packet basis as overhead in the receiver's input buffer. The total number of bytes required at the receiver's buffer for a single packet equals PktOvhd + (frame size in bytes) rounded up to the credit size. This total size is used to determine the number of credits required to transmit a packet. If a part of each packet is not stored in the receiver's input buffer, for example, the CRC may be stripped before storing, then PktOvhd may be a negative value.

The number of credits is a function of the receiver's buffer implementation. CBFC supports two methods of setting credit limits:

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

- The receiver sets a single value for the total credit limit (TotalCredits). It is then up to the sender to distribute these credits among the lossless VCs.
- The receiver sets each VC's credit limit (VC_CreditLimit[x]), and the sender must adhere to these limits on a per-VC basis.

Mixing the two methods is not allowed; for example, if the receiver sets the VC_CreditLimit[x] for some lossless VCs, then it must set the VC_CreditLimit[x] for all lossless VCs and leave the TotalCredits parameter at zero. When the receiver sets the TotalCredits parameter instead of individual VC limits, the sender should distribute the total number of credits across the lossless VCs in an implementation-specific manner. This allows a variety of per-VC credit limit implementations with combinations of dedicated and shared VC credit limits. Note that per-VC credit limits are required to guarantee non-blocking behavior between lossless VCs.

In addition to the configuration parameters listed in Table 5-14, a mapping of packet header fields to VC number must be configured at the sender and receiver so that they assign the same VC number to any given packet. At a minimum, a mapping to VC numbers from the **mac.vlan.pcp_dei** and **ip.dscp** header fields as well as any subset of these fields MUST be supported. The mapping from packet header values to VC number may be configured by upper-layer network management software or through the CBFC lossless VC initialization process (see Section 5.2.8).

Implementation Note:
When both **mac.vlan.pcp_dei** and **ip.dscp** header fields are being used to map to a VC, a consistent configuration of which header fields take precedence is required.

Virtual channels and PFC traffic classes must be mapped consistently with respect to lossless traffic classes at the network level. The exact mapping mechanisms are an implementation-specific feature and beyond the scope of this specification.

5.2.5.3 CBFC Cyclic Counters and State Variables

Sender cyclic counters (required) are shown in Table 5-15.

Table 5-15 - Sender Cyclic Counters

Counter	Description
S_VC_CC[x]	Sender credits consumed by VC[x], where x = 0 to (NumVCs -1). An S_VC_CC counter is incremented by the number of credits required for a packet when the packet is transmitted by the sender.
S_VC_CF[x]	Sender credits freed by VC[x], where x = 0 to (NumVCs -1). An S_VC_CF counter is updated when a CBFC CF_Update message is received from the link partner.

Receiver cyclic counters (required) are shown in Table 5-16.

Table 5-16 - Receiver Cyclic Counters

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Counter	Description
R_VC_CC[x]	Receiver credits consumed by VC[x] where x = 0 to (NumVCs -1). R_VC_CC counters are updated when a packet is received on the link and when a CBFC CC_Update message is received from the sender.
R_VC_CF[x]	Receiver credits freed by VC[x], where x = 0 to (NumVCs -1). An R_VC_CF counter is incremented as buffer space is made available for additional packet reception.

All cyclic counters wrap around at the modulus of the width of the counter. For example, S_VC_CC[x] is implemented as 20-bit counter. When incrementing by N credits, the new value would be:

$$S\_VC\_CC[x] = (S\_VC\_CC[x] + N) \bmod 2^{20}$$

For ease of understanding and readability, the rest of this specification does not show the modulus operation when performing additions or comparisons to these cyclic counters, and they appear as always-increasing values.

State variables are shown in Table 5-17.

Table 5-17 - Sender State Variables

State Variable	Description
S_P_CL	Sender port credit limit. S_P_CL is initialized to the total number of credits available when set by the receiver (TotalCredits).
S_P_CU	Sender port credits in use counter. S_P_CU is incremented by the number of credits used for packets as they are transmitted by the sender, and it is decremented as credits are returned to the sender.
S_VC_CL[x]	Sender credit limit for VC[x]. Set either directly during initialization with a value specified by the receiver or by the sender after distributing TotalCredits among the lossless VCs.
S_VC_CU[x]	Sender credits in use for VC[x]. This variable is equal to S_VC_CC[x] – S_VC_CF[x].

Table 5-18 - Counter and Variable Widths and Initialization

Counter	Position	# of instances	Width (bits)	Initial Value
S_P_CL	Sender	1	20	TotalCredits
S_P_CU	Sender	1	20	0
S_VC_CL[x]	Sender	NumVCs	20	0 ≤ n ≤ TotalCredits
S_VC_CU[x]	Sender	NumVCs	20	0
S_VC_CC[x]	Sender	NumVCs	20	0
S_VC_CF[x]	Sender	NumVCs	15	0
R_VC_CC[x]	Receiver	NumVCs	20	0
R_VC_CF[x]	Receiver	NumVCs	15	0

Note:
- NumVCs is implementation-dependent, but an implementation SHALL support at least one lossless VC for the CBFC mechanism to be enabled.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Counter	Position	# of instances	Width (bits)	Initial Value
• The width of the S_P_CL, S_P_CU, S_VC_CL[x], and S_VC_CU[x] counters MUST support the per-VC CC counter width of 20 bits. These counters are not visible externally.				

CF and CC counter size is determined by the frequency of update messages as well as the round-trip time (RTT) between link partners and the size of the receiver's dedicated input buffer. To support a reach of 500 m, with approximate RTT = 5 µs and credit size of 64 B, then 7.8k credits can be in flight at 800 GE. A counter of 15 bits (up to 16k) for the per-VC credit freed cyclic credit counters is needed. CC_Update messages are meant to be much less frequent than CF_Update messages and use little link bandwidth. CC counters of 20 bits support 2^{19} credits. For a credit size of 64 B, CC counters of 20 bits can support a buffer size of up to 32 MB between updates. For CC_Update messages to use less than 0.01% of link bandwidth they need to be able to represent up to 800000 bytes of data, well within the 32 MB limit, leaving room for future faster port speeds.

5.2.5.4 CBFC Point-to-Point Messages

Two message types are supported for communication between link partners:

- **CF_Update:** Credit freed counter update from receiver to sender. The purpose of this message is to return credits to the sender so they can be reused to transmit additional packets.
- **CC_Update:** Credit consumed counter update from sender to receiver. The purpose of this message is to recapture possible “leaked” credits resulting from packet drops due to link errors such as uncorrectable FEC errors or packet CRC errors.

Table 5-19 - CBFC Message Types

Message	Direction	Format	Frequency	Content
CF_Update	Receiver-to-Sender	CtlOS (8 B)	High	Two R_VC_CF[x] counter values
CC_Update	Sender-to-Receiver	Packet (64 B)	Low	Up to 16 S_VC_CC[x] counter values

The exact format of each CBFC message is described in Section 5.2.6.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

5.2.5.5 Packet and CBFC Message Sequence

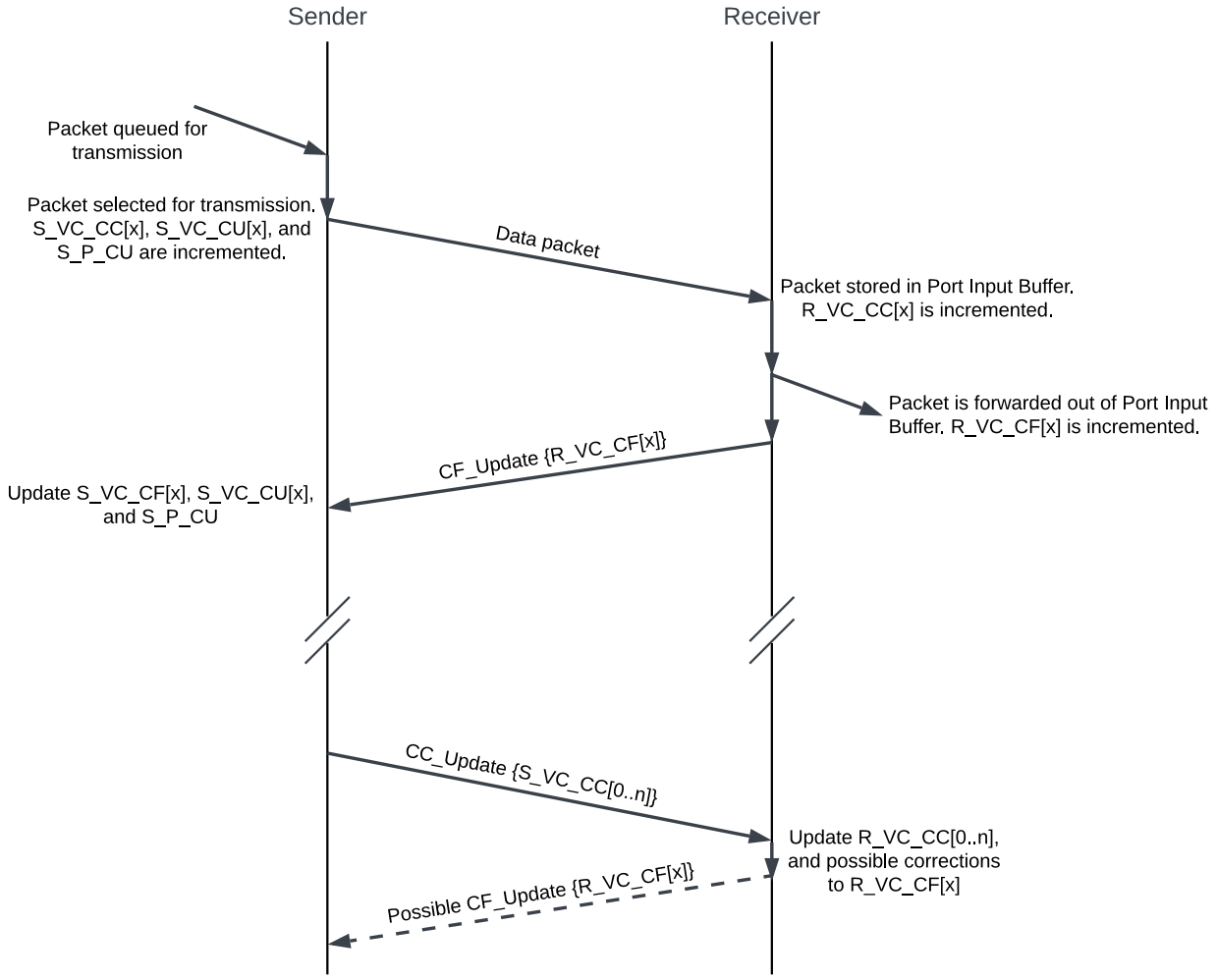


Figure 5-8 - Packet and CBFC Message Sequence with Counter Updates

5.2.5.6 Sender Operations

5.2.5.6.1 Packet Transmission Credit Check

To select a packet from a lossless VC for transmission and to make counter updates, the number of credits required for the packet must first be computed:

```
pktSize = size of the MAC frame in bytes
pktCredits = roundup((pktSize + PktOvhd) ÷ CreditSize)
```

Informative Text:
The size of a MAC frame includes the bytes beginning with the MAC DA and ending with the FCS.

A packet can be selected for transmission in the following cases:

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

- When the receiver sets individual per-VC credit limits:
  `S_VC_CU[x] + pktCredits ≤ S_VC_CL[x]`

OR

- When the receiver sets the TotalCredits limit:
  `S_P_CU + pktCredits ≤ S_P_CL`

Additional per-VC limits are not precluded when checking the S_P_CL, but the only requirement is that the S_P_CL limit shall be observed.

The credit limit check does not preclude cut-through operation. If the number of credits available is sufficient to transmit a packet of the maximum size, then the packet may be selected for transmission as soon as the first bytes of the packet are ready for transmission.

Transmission of packets for best-effort VCs do not require this credit check and do not consume credits.

5.2.5.6.2 Packet Transmission Counter Updates

When a packet is selected for transmission from a lossless VC, these counter updates are made:

```
S_P_CU = S_P_CU + pktCredits
S_VC_CC[x] = S_VC_CC[x] + pktCredits
S_VC_CU[x] = S_VC_CU[x] + pktCredits
```

Where x is the VC number of the scheduled packet.

5.2.5.6.3 CF_Update Message Reception

When the sender receives a CF_Update message with a valid R_VC_CF[x] value, these counter updates are made:

```
credits_returned = R_VC_CF[x] – S_VC_CF[x]
S_P_CU = S_P_CU – credits_returned
S_VC_CU[x] = S_VC_CU[x] – credits_returned
S_VC_CF[x] = R_VC_CF[x]
```

5.2.5.6.4 CC_Update Message Generation

CC_Update packet(s) MUST be generated periodically to cover all lossless VCs (note that VCs 0 to 15 and VCs 16 to 31 are contained in two separate CC_Update packets). The main purpose for the CC_Update is to recover any leaked credits due to packet drops caused by link errors (uncorrectable FEC errors or packet CRC errors).

If a data packet is discarded due to link errors, the sender's S_VC_CC[x] counter was already incremented by the number of credits required. But since the receiver never receives the discarded packet, it would never return the credits to the sender in a CF_Update message. This is known as "credit leakage." The CC_Update message is a means for the sender to inform the receiver of the number of credits it thinks it has consumed, thus allowing the receiver to make any necessary counter corrections

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

and reclaim any leaked credits. The receiver makes corrections to its credit counters, and eventually the sender's S_VC_CF, S_VC_CU, and S_P_CU are also updated.

The CC_Update packet contains up to 16 of the sender's S_VC_CC[x] counter values carried in a 64-byte MAC frame. The S_VC_CC[x] values in the CC_Update packet MUST include all credits consumed for packets transmitted prior to the CC_Update packet and MUST NOT include any credits for packets to be transmitted after the CC_Update packet.

A configurable time interval MUST be provided between generation of CC_Update messages (CC_msg_timer).

5.2.5.7 Receiver Operations

5.2.5.7.1 Packet Reception Counter Updates

When a packet is received, the VC number of the packet is determined (mapped) from packet header fields. The number of credits required by the packet is calculated in the same way that was done at the sender. Once calculated the receiver updates the VC credits consumed.

```
pktSize = size of the MAC frame in bytes
pktCredits = roundup((pktSize + PktOvhd) ÷ CreditSize)
R_VC_CC[x] = R_VC_CC[x] + pktCredits
```

Whether the packet is written into the receiver's input buffer or accorded cut-through operation, possibly bypassing the receiver's data buffer, the R_VC_CC[x] counter must be updated as shown in the pseudocode above. If the packet is cut through without utilizing any buffer resources, the VC's credit freed counter must also be immediately updated as shown in Section 5.2.5.7.2.

5.2.5.7.2 Input Buffer Drain and Counter Updates

When a packet is removed from the receiver's input buffer or is cut through, the packet's VC number is used to update the credit-freed counter by the packet's credit usage:

```
pktSize = size of the MAC frame in bytes
pktCredits = roundup((pktSize + PktOvhd) ÷ CreditSize)
R_VC_CF[x] = R_VC_CF[x] + pktCredits
```

At this point, the R_VC_CF[x] is now eligible to be sent to the sender in the CF_Update message.

5.2.5.7.3 CF_Update Message Generation

CF_Update messages contain R_VC_CF[x] values from the receiver to the sender. They must be generated based on any of the following triggers:

- The R_VC_CF[x] was updated, and the CF_min_timer has expired.
- When a single CF_max_timer exists and has expired.
    - While permissible to generate CF_Update messages for all R_VC_CF[x] values, it is only necessary to generate CF_Update messages for VCs that did not have an R_VC_CF[x] update within the CF_max_timer interval.
- When a per-VC CF_max_timer[x] exists and has expired.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

- It is necessary to generate a CF_Update message for R_VC_CF[x] if it did not have any update within the CF_max_timer[x] interval.

The CF_min_timer is used to guarantee a minimum spacing between CF_Update messages to ensure minimal bandwidth overhead is used for these messages.

Implementations may choose a single CF_max_timer or per-VC CF_max_timer[x]. At least one of these timers MUST be implemented. These are used to periodically update R_VC_CF[x] values in case a previous CF_Update message was lost.

See Section 5.2.6.1 for the format of the CF_Update message.

5.2.5.7.4 CC_Update Message Reception

As described in Section 5.2.5.6.4, CC_Update messages are periodically generated by the sender. When the receiver receives one of these messages, it performs the following counter updates for all valid S_VC_CC[x] values in the message.

```
For x = 0 to (NumVCs-1)
      LostCredits = S_VC_CC[x] – R_VC_CC[x]
      If (LostCredits > 0) then
            R_VC_CF[x] = R_VC_CF[x] + LostCredits
      R_VC_CC[x] = S_VC_CC[x]
end for
```

Implementation Note:
If the number of LostCredits is ever less than zero or greater than 2^{14}-1, a serious error has occurred and SHOULD be reported to the management entity.

5.2.5.8 Handling Non-Ideal Credit Sizes (Informative)

Ideally, the CreditSize parameter should be set to the actual "cell size" of the receiver's input buffer. This makes maximum use of the buffer. However, this may not be possible if the sender and receiver do not both support this ideal CreditSize. In this case, the next CreditSize less than the actual cell size that is supported by both the sender and receiver should be used.

For example, if the input buffer size is 8000 bytes and the native cell size is 80 bytes, the buffer has 100 cells. Ideally, the sender and receiver would set the CreditSize=80 and TotalCredits=100. If the sender does not support the CreditSize of 80, then the CreditSize may be set to 64 bytes. In this case, the TotalCredits must still be set to 100 (not 125), because 100 packets of 64 B would fill the input buffer. This makes an 8000-byte buffer no better that a 6400-byte buffer in terms of CBFC utilization. However, the receiver can optimize the use of its 8000-byte buffer when receiving larger packets. If a packet is received with size between 65 and 80 bytes, the sender will consume two credits; but at the receiver, these packet sizes use only one cell, so one credit can be freed immediately without waiting for this packet to be drained from the input buffer. Care must be taken to free only one additional credit when the packet is eventually removed from the input buffer. The same difference in credits used versus cells used happens if a packet is between 129 and 160 bytes where the sender consumes three credits, but

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

the packet only requires two cells, so one credit can be freed immediately. The difference between the calculated credits and the actual cells used should be relatively easy to calculate at the receiver as packets of any size arrive.

Another alternative is for the receiver to count credits consumed and freed based solely on its native cell size. If the sender uses a conservative CreditSize (smaller than the actual cell size), there can be some “credit leakage” depending on the packet sizes. CC_Update messages can be used to reclaim all leaked cells. In this case, a smaller CC_msg_timer value is recommended.

5.2.6 CBFC Message Formats

5.2.6.1 CF_Update Message

The CF_Update message is one type of CtlOS message. See Section 5.2.11 for the general CtlOS format and requirements.

The CF_Update CtlOS contains two R_VC_CF[x] values, specified as CF1 and CF2, and has the format on the xMII as defined in Table 5-20 and Table 5-21.

Table 5-20 - CF_Update CtlOS Message Format

Message	xMII Data								
	Lane 0	Lane 1	Lane 2	Lane 3	Lane 4		Lane 5	Lane 6	Lane 7
CF_Update	0x5C	Type= 0x10	D2	D3	D4[7:4]	0x6	D5	D6	D7

Note:

- Lane 0 contains the control character value for UE control ordered set.
- Lane 1 contains the CtlOS message type identifier.
- D2—D7 contain the message data as defined in Table 5-21.

Table 5-21 - CF_Update CtlOS Data Field Definitions

xMII Data Field	Msg Field name	#bits	Description
D2[7:3]	CF1_VC_index[4:0]	5	CF1 VC number
{D2[2:0], D3[7:0], D4[7:4]}	CF1_count[14:0]	15	R_VC_CF[CF1_VC_index] counter value
D4[3:0]	reserved (O-code)	4	Set to 0x6
D5[7:3]	CF2_VC_index[4:0]	5	CF2 VC number
{D5[2:0], D6[7:0], D7[7:4]}	CF2_count[14:0]	15	R_VC_CF[CF2_VC_index] counter value
D7[3:0]	Reserved	4	Set to ‘0’ on transmit and ignored upon reception.

The PCS sublayer must encode the CF_Update CtlOS as described in the UE PHY specification into a 64B/66B encoded block, passing all data from xMII lanes D1 to D7 to the 66-bit block.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

5.2.6.2 CC_Update Messages

CC_Update messages are less frequent and take the form of an MAC frame. A single CC_Update packet can contain 16 S_VC_CC[x] values for VC[0..15] or VC[16..31]. The UE message type field is used to distinguish the two flavors of CC_Update packet.

Table 5-22 - CC_Update Message Packet Fields

Packet Field	Name	Size	Description
MACDA	MAC Destination Address	6 B	Set to 01-80-C2-00-00-01 or the 48-bit individual address of the destination station
MACSA	MAC Source Address	6 B	Set to the 48-bit individual address of the source station.
Eth Type	EtherType	2 B	0x8808
Opcode	MAC Control Opcode	2 B	0xFFFE (Extension opcode)
CID	Company ID	3 B	UEC CID (FA-7A-CB)
Msg Type	UE Message Type	1 B	0x01 = CC_Update for VC[0:15] 0x02 = CC_Update for VC[16:31] All other values are reserved.
Data	16 per-VC Credits Consumed values	40 B	16 CC counter values (20 bits each counter): {S_VC_CC[0], S_VC_CC[1], … S_VC_CC[15]} or {S_VC_CC[16], S_VC_CC[17], … S_VC_CC[31]}
FCS	Frame Check Sequence	4 B	CRC32

Note:

- This is a 64 B MAC frame.
- The S_VC_CC[x] value for any unused or best-effort VC SHALL be set to 0.

The values and format for the MACDA, MACSA, EtherType, and MAC control opcode fields ,as well as the UEC CID, are chosen to be compatible with IEEE Std 802.3-2022 Clause 31 “MAC Control” and its Annexes [1]. It is recommended to make the MACDA, MACSA, and CID values configurable at both the sender and receiver. Because the CC_Update packet is exchanged only between link partners, and not forwarded, the MACDA and MACSA are not required to identify it. It can be identified solely by matching the EtherType, Opcode, CID, and Msg Type combined with a good FCS.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

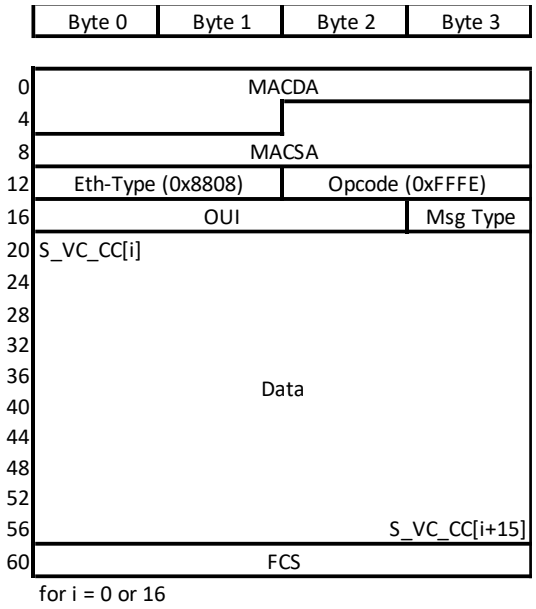


Figure 5-9 - CC_Update Packet Format

5.2.7 MAC and MAC Control Layer Interfaces to CBFC

5.2.7.1 MAC Service Interface Additions

CBFC acts as a MAC client above the MAC control layer as shown in Figure 5-7.

5.2.7.2 MAC Insertion of CBFC Messages

The MAC layer must be modified to allow for CtlOS messages to preempt the transmission of in-progress Ethernet packets. This allows for regular and more frequent CF_Update messages than would otherwise be possible. See Section 5.2.11 for more information on CtlOS message handling.

The counter values contained in a CC_Update packet are related to the packets that are transmitted prior to the CC_Update transmission. When CBFC generates a CC_Update packet, it must be delivered to the MAC (or MAC control) layer using the MA_DATA.request interface to maintain its relative position to other data packets when transmitted.

5.2.7.3 MAC Reception of CBFC Messages

When the MAC (or MAC control) receives a correctly formed CC_Update packet, it must be forwarded to the upper layer MAC client on the MA_DATA.indication primitive for processing in strict packet order relative to all other received data packets, because the payload counters are relative to the CC_Update packet's position to other packets on the physical link.

CF_Update CtlOS messages should be removed from the RX data path and forwarded directly to the CBFC layer via the MA_CONTROL.indication primitive with the CF_Update counter values.

5.2.8 CBFC Initialization

The two end stations of a link, sender and receiver, must have several configurable parameters initialized in a consistent manner for correct operation. Per-link features and parameters may be

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

configured either through management software or through an information exchange using the IEEE Link Layer Discovery Protocol (LLDP) [5].

5.2.8.1 Network-Level Configuration

Higher-level network configuration establishes traffic class (TC) parameters for the entire network. The following CBFC parameters are derived from the network TC parameters for each link based on the capabilities of each link partner:

- NumVCs
  - The number of VCs must be set identically at the sender and receiver.
  - Sender.NumVCs == Receiver.NumVCs
- Lossless VC configuration
  - VCs must be configured consistently as lossless or best effort at the sender and receiver.
- Lossless VC mapping
  - Sender and receiver must be configured to map the same packet properties to the same VC number.

The number of VCs, the mapping of packet fields to VC numbers, and the disposition of a VC as best-effort or lossless must be configured consistently between the link partners: sender and receiver. Note that the link partners may be configured differently in each direction of the link; the sender and receiver of a single end station are independent of each other.

5.2.8.2 Link-Level Configuration

Switch and NIC chip buffer sizes and buffering structures such as the buffer cell size vary between implementations. This makes parameters such as the credit size and number of credits a property of each link and dependent on the receiver implementation.

Link-level negotiation can be used to establish these values between the link partners:

- CreditSize
  - Determined by the receiver's data buffer characteristics.
  - Receiver: Initially set to the cell size of the port input buffer:
    - Receiver.CreditSize = cell_size
    - May adjust to the sender's CreditSize after it is set (see 5.2.5.8).
  - Sender: Ideally set to the same size as at the receiver, but if not possible, then:
    - Sender.CreditSize ≤ Receiver.CreditSize (as close as possible)
- TotalCredits
  - Determined by the receiver's data buffer characteristics.
  - Receiver:
    - Receiver.TotalCredits = Receiver(Input buffer size) ÷ Receiver(cell size)
  - TotalCredits at the sender must be set to the same number:
    - Sender.TotalCredits = Receiver.TotalCredits
  - When the receiver sets individual per-VC credit limits, TotalCredits is not used at the sender.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

- PktOvhd
    - The size of the per-packet overhead is a function of the receiver buffer design and is set to the size of additional space required at input buffer for each packet.
    - Receiver:
        - Receiver.PktOvhd = (additional per-packet buffer space).
    - Sender: Ideally set to the same number as at the receiver but must be conservative if not set to exactly the receiver's value.
        - Sender.PktOvhd ≥ Receiver.PktOvhd
- Per-VC credit limits
    - When the receiver sets the TotalCredits limit:
        - Distribution of the TotalCredits among the lossless VCs at the sender is an implementation-dependent function. The sender's implementation must prevent blocking between VCs.
    - When the receiver sets an individual VC_CreditLimit[x] for each lossless VC:
        - The sender shall set each lossless VC credit limit, S_VC_CL[x], to honor the receiver's requested limits.

While each link partner must be able to be configured by management software with these link-specific parameters, it is also desirable to provide a link-level protocol to allow communication between link partners to set up coordinated values for these parameters. UE link negotiation may be used for this purpose.

5.2.8.3 CBFC Link Level Initialization Using UE Link Negotiation

When a port is reset, all VCs are initially set to best effort delivery. To establish lossless packet delivery, lossless VCs must be initialized using the CBFC lossless VC initialization process through an exchange of LLDP packets between the link partners. As shown in Figure 5-10, a VC's disposition can later be changed back to best effort through the lossless VC removal process without disturbing traffic on other VCs.

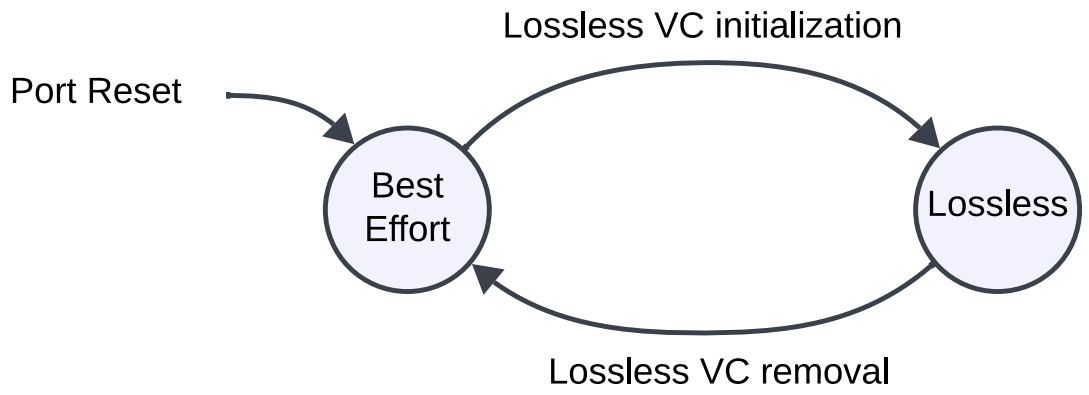


Figure 5-10 - Virtual Channel State

The UE link negotiation mechanism uses the IEEE Link Layer Discovery Protocol (LLDP) [5]to specify UE-enabled features and exchange any necessary information between link partners to support those

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

features. The UE link negotiation specification describes LLDP packet formats for several link-layer features including CBFC.

Each station advertises its capabilities to its link partner using LLDP packets. Parameter values are encoded into “type, length, value” triplets (TLVs) inside the LLDP data unit (LLDPDU) of the LLDP frame. The LLDP frame format and CBFC TLV formats are described in detail in the UE Link Negotiation specification. Sections 5.2.8.3.1 and 5.2.8.3.2 discuss the main CBFC TLV fields needed for the CBFC initialization process as described in Section 5.2.8.4. Not all TLV fields are listed here; the Link Negotiation specification provides the complete list of TLV fields.

LLDP packets transmitted during the lossless VC initialization process MUST NOT use any of the lossless VCs being initialized.

5.2.8.3.1 CBFC Receiver TLV Fields

The CBFC TLV has two sections of receiver values: One section contains port information common across all VCs, and the second section contains information fields specific to each VC.

CBFC TLV receiver port information fields are shown in Table 5-23.

Table 5-23 - CBFC TLV Receiver Port Information Fields

Information Field	Description
R_NumVCs	The total number of VCs supported by the receiver, both best effort and lossless (2-32).
R_TotalCredits	When non-zero, this field contains the total number of credits available at the receiver. When zero, it indicates that separate per-VC credit limits are set by the receiver (see R_VC_CreditLimit[x] in Table 5-24).
R_CellSize	The receiver’s input buffer storage data unit size in bytes (ideal CreditSize).
R_PktOvhd	The receiver’s per-packet buffer overhead. Stored as an eight-bit two’s complement number of bytes, -16 to +127.

CBFC TLV receiver per-VC information fields are shown in Table 5-24.

Table 5-24 - CBFC TLV Receiver Per-VC Information Fields

Information Fields	Description
R_VC_Want[x]	Indicates the receiver wants to make VC[x] a lossless VC.
R_VC_RTR[x]	Indicates the receiver is ready to receive lossless packets on VC[x] using CBFC credits.
R_VC_CreditLimit[x]	Optional credit limit to be applied to VC[x] at the sender. Valid only when R_TotalCredits=0.
R_PKTID_Sel[x] and R_PKTID[x]	Optional fields to specify the VLAN.PCP/DEI or DSCP values to identify VC[x]. See the Link Negotiation specification for the use of these fields.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

5.2.8.3.2 CBFC Sender TLV Fields

The CBFC TLV sender information fields are shown in Table 5-25.

Table 5-25 - CBFC TLV Sender Information Fields

Information Fields	Description
S_VC_RTS	A 32-bit bitmap of all possible VCs supported by this sender. When the sender has completed initialization of lossless VC[x] and is ready to send packets using credits, S_VC_RTS[x] is set to one.
S_CreditSize	The sender's configured CreditSize. This value is set only after receiving the link partner's advertised R_CellSize.
S_PktOvhd	The sender's configured per-packet overhead. This value is set only after seeing the link partner's advertised R_PktOvhd. Stored as an eight-bit two's complement number of bytes, -16 to +127.

5.2.8.4 Lossless VC Initialization Process

Each end station requires two LLDP CBFC TLV databases, one for itself and one for its link partner. They are referred to as the local LLDP database (and local TLV values) and the remote LLDP database (and remote TLV values). A single end station may act as both a sender and receiver, but these functions do not interact and must be configured separately. A CBFC local sender communicates only with the remote receiver, and a local receiver communicates only with the remote sender. The lossless VC initialization process may occur simultaneously and independently for each direction of traffic on a link.

The local sender's TLV information depends on first receiving its remote receiver's information. The local sender SHALL NOT start sending packets using CBFC credits until both stations have completed initialization, as indicated by the sender's ready-to-send (RTS) and the receiver's ready-to-receive (RTR) bits in the respective LLDP databases.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

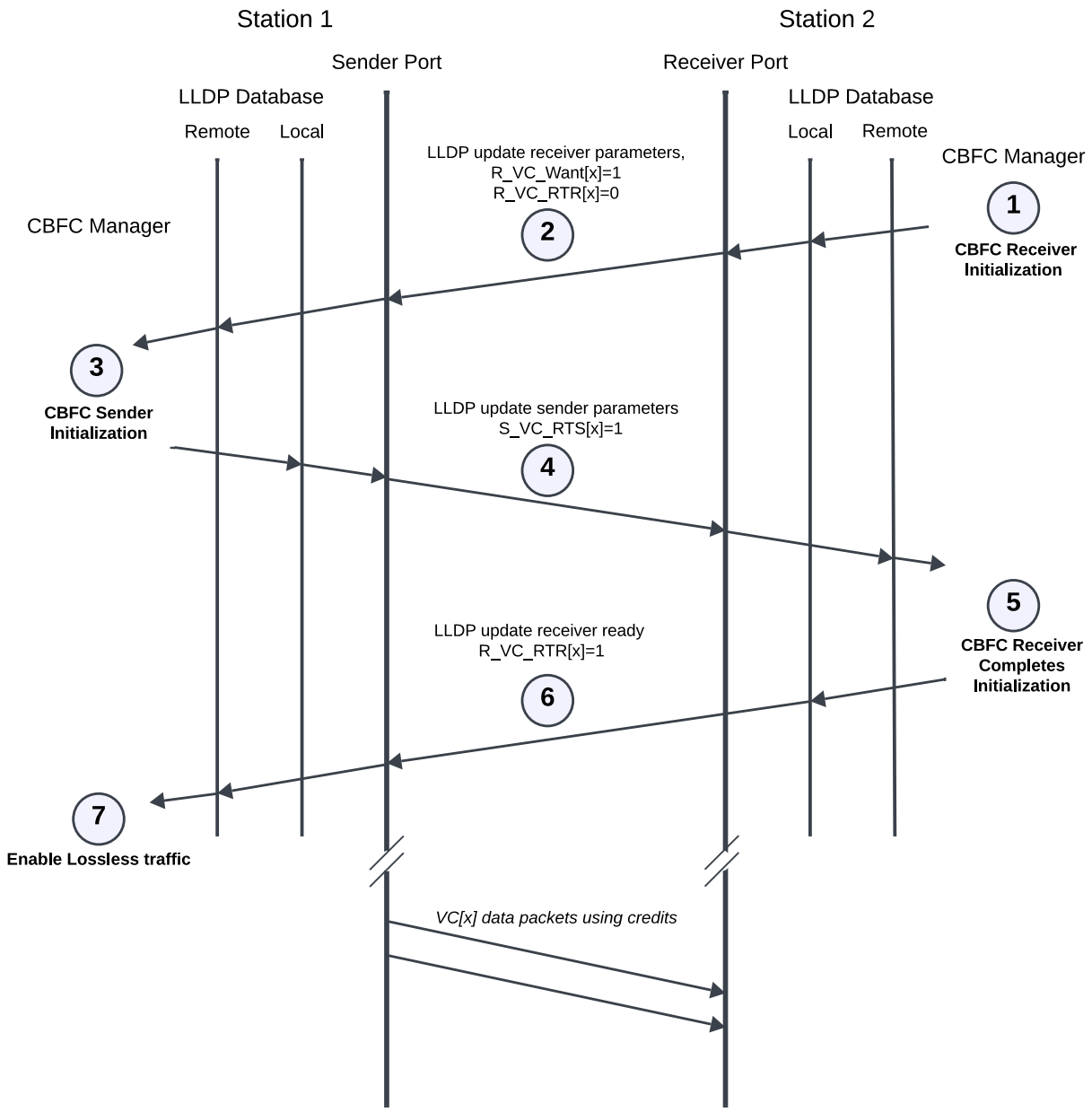


Figure 5-11 - Lossless VC Initialization Process

The lossless VC initialization process follows these steps as shown in Figure 5-11:

1. CBFC receiver Initialization:
   The CBFC management entity at the receiver initializes the receiver port for one or more lossless VCs and initializes the TLV values in the local CBFC LLDP database.
   - Local.R_NumVCs = total number of VCs supported on the link.
   - Local.R_TotalCredits = (buffer size ÷ cell size) or 0 if setting individual VC credit limits.
   - Local.R_CellSize = cell size of the receiver's input buffer (receiver's preferred CreditSize).
   - Local.R_PktOvhd = per-packet buffer overhead (-16 to +127)

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

- For each VC[x] that the receiver wants to initialize as a lossless VC:
  - Local.R_VC_Want[x] = 1
  - Local.R_VC_RTR[x] = 0 (not yet ready to receive packets using credits)
  - Reset the receiver's VC[x] counters
    - R_VC_CC[x] = 0
    - R_VC_CF[x] = 0
  - Optional:
    - Local.R_VC_CreditLimit[x] = per-VC credit limit (or 0 if TotalCredits != 0)
    - Local.R_PKTID_Sel[x] and R_PKTID[x]

2. LLDP packet is sent from receiver to sender with the receiver's new TLV values and is used to update the remote LLDP database at the sender.
3. CBFC sender Initialization:
   CBFC management entity at the sender station, based on the remote LLDP database update, initializes the CBFC sending port and local LLDP database.
   - Set sender port configuration and parameters S_TotalCredits, S_CreditSize, S_PktOvhd based on remote receiver's LLDP values and its own supported feature set.
   - For each VC[x] with Remote.R_VC_Want[x] == 1 and Local.S_VC_RTS[x] == 0:
     - Distribute S_TotalCredits across all lossless VCs (set per-VC credit limits) or use Remote.R_VC_CreditLimit[x] to set the per-VC credit limits.
     - Disable transmission of all packets for VC[x].
     - Reset the sender's VC[x] counters[17]
       - S_VC_CC[x] = 0
       - S_VC_CF[x] = 0
       - S_VC_CU[x] = 0
   - Ensure observance of any new credit limits.
     - If the receiver has set credit limits using port-based TotalCredits:
       - Wait until S_P_CU ≤ S_P_CL
       - Any additional implementation-specific credit limits must also be followed accordingly.
     - If the receiver has set credit limits using per-VC credit limits:
       - Wait until S_VC_CU[i] ≤ S_VC_CL[i], for all lossless VC[i].
   - Update TLV values in Local LLDP database:
     - Local.S_CreditSize = S_CreditSize
     - Local.S_PktOvhd = S_PktOvhd
     - Local.S_VC_RTS[x] = 1 (for all VC[x] with Remote.R_VC_Want[x] == 1)
4. LLDP packet is sent from sender to receiver with the sender's updated TLV values and is used to update the remote LLDP database at the receiver.
5. CBFC receiver completes initialization:
   CBFC management entity completes lossless VC initialization.
   - For each VC[x] with remote.S_VC_RTS[x]==1 and local.R_VC_RTR[x]==0:

[17] The port-based counter, S_P_CU, must be reset to zero only when the port is reset.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

        - Complete any necessary receiver configuration based on remote LLDP sender fields.
        - Set Local.R_VC_RTR[x] = 1
6. LLDP packet is sent from receiver to sender with R_VC_RTR[x] flag(s) set.
7. Enable Lossless Traffic
    - For each VC[x] with remote.R_VC_RTR[x]==1 and local.S_VC_RTS[x]==1:
        - Enable transmission of packets for lossless VC[x] using CBFC credits.

When CBFC is used in a network, a link is likely to be initialized with one or two best-effort VCs and one or two lossless VCs. Over time, additional traffic classes may be added to the network that require additional lossless VCs to be configured. This can be accomplished while the link is active by restarting the lossless VC initialization process and updating the R_VC_WANT, S_VC_RTS, and R_VC_RTR flags for the new lossless VCs. The receiver per-port TLV values R_CellSize and R_PktOvhd must not be changed when initializing additional lossless VCs while some lossless VCs are already active. The method of setting the credit limits — setting TotalCredits or individual per-VC limits — must follow the method chosen for previously established lossless VCs. The R_TotalCredits value may be increased when adding additional lossless VCs.

5.2.8.5 Lossless VC Removal Process

A VC can be changed from lossless to best effort by following the lossless VC removal process. This process is described in Figure 5-12. After a lossless VC is removed, it can remain unused or be used as a best-effort VC.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

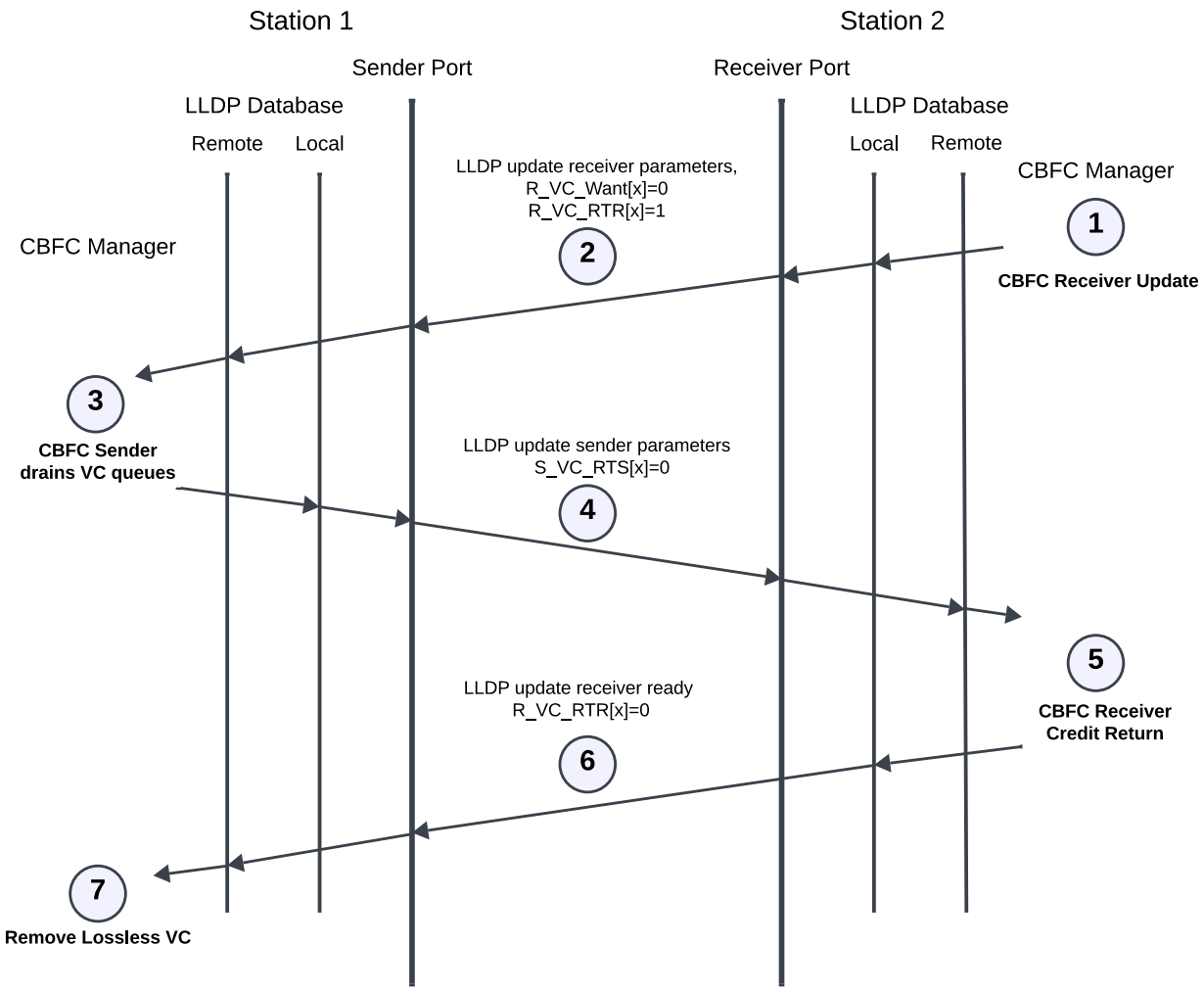


Figure 5-12 - Lossless VC Removal Process

The lossless VC removal process follows these steps as shown in Figure 5-12:

1. CBFC receiver update
   CBFC management entity updates the receiver to remove one or more lossless VCs and sets the TLV values in the local CBFC LLDP database accordingly:
   - Local.R_NumVCs = total number of VCs supported on the link
   - Local.R_TotalCredits = original R_TotalCredits, or optionally a new R_TotalCredits value
   - For each lossless VC[x] that the receiver wants to remove:
     - Local.R_VC_Want[x] = 0
     - Local.R_VC_RTR[x] = 1 (no change)
2. LLDP packet is sent from receiver to sender with the receiver's new TLV values and is used to update the remote LLDP database at the sender.
3. CBFC sender drains VC queues
   CBFC management entity performs the following actions:
   - For each VC[x] with Remote.R_VC_Want[x] == 0 and Local.S_VC_RTS[x] == 1:

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

        - Stop enqueue of packets to VC[x] queues.
        - Wait for VC[x] queues to drain.
    - Set up CBFC sender configuration and parameters.
        - Redistribute new TotalCredits to remaining lossless VCs or update all VC limits based on new Remote.R_VC_CreditLimit[x].
    - Ensure observance of any new credit limits.
        - If the receiver has set credit limits using port-based TotalCredits:
            - Wait until S_P_CU ≤ S_P_CL.
            - Any additional implementation-specific credit limits must also be followed accordingly.
        - If the receiver has set credit limits using per-VC credit limits:
            - Wait until S_VC_CU[i] ≤ S_VC_CL[i], for all lossless VC[i].
    - Update local sender LLDP database:
        - Local.S_VC_RTS[x]=0 (for all VC[x] with Remote.R_VC_Want[x] = 0)
4. LLDP packet is sent from sender to receiver with the sender's new TLV values and is used to update the remote LLDP database at the receiver.
5. CBFC receiver credit return
   CBFC management entity performs the following actions:
    - For each VC[x] with remote.S_VC_RTS[x] == 0 and local.R_VC_RTR[x] == 1:
        - Wait for all VC[x] packets to drain from receiver's buffer and all VC[x] credits to be freed and returned using CF_Update CtlOS to the sender.
        - Update the Local receiver LLDP database:
            - Local.R_VC_RTR[x]=0.
6. LLDP packet is sent from receiver to sender with the receiver's updated TLV values and is used to update the remote LLDP database at the sender.
7. Remove lossless VC
   CBFC management entity performs the following actions:
    - For each VC[x] that has remote.R_VC_RTR[x] changed from 1 to 0, lossless VC[x] is now removed and may be left unused or start operating as a best-effort VC, including enqueue and dequeue of packets on a best-effort basis.

The receiver per-port TLV values R_CellSize and R_PktOvhd must not be changed when performing the lossless VC removal process while other lossless VCs remain active, but it may be desirable to reduce the R_TotalCredits value. A new R_TotalCredits value can be set by the receiver (shown as optional in step #1 above), which propagates to the sender through the LLDP database update. Because the new lossless VC credit limits are not immediately applied when set in step #1, the receiver must maintain the original buffer space (and honor the previous TotalCredits limit) until the receiver sees the sender's LLDP database update as indicated by remote.S_VC_RTS[x]==0 in step #5. After this, the receiver can be sure that the sender is using the new credit limits for any packets received.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

5.2.9 Interactions Between CBFC, PFC, and LLR

When an LLR retransmission is triggered due to a lost packet, it creates several possible hazards that could affect the CFBC credit counters due to:

1. Not all data packets are retransmitted by LLR. LLR considers packets either “retransmit eligible” (LLR-eligible) or “retransmit ineligible” (LLR-ineligible). The LLR protocol requires that all LLR-eligible packets are discarded at the receiver between the time of LLR-NACK generation and the reception of the first retransmitted packet with the correct sequence number (call this the “LLR discard window”). LLR-ineligible packets are accepted by the receiver during the LLR discard window. The LLR protocol also requires the sender to retransmit LLR-eligible packets within an “LLR replay window”. An LLR replay is initiated at the sender upon reception of an LLR-NACK. The LLR replay window includes the time to retransmit all LLR-eligible packets from the first non-ACK’d packet up to the last LLR-eligible packet sent before the reception of the LLR-NACK. LLR-ineligible packets may be interspersed among the retransmitted packets. Normal transmission of both LLR-eligible and LLR-ineligible packets resumes after the LLR replay window.
2. The S_P_CU and S_VC_CC[x] counters are already incremented for all transmitted packets from the time that the first lost packet is transmitted until the LLR-NACK is received at the sender, but these packets are discarded at the receiver if they are LLR-eligible. The R_VC_CC[x] counters must not be incremented for the discarded packets, since they will be updated later when the packets are accepted during retransmission. This also means that the retransmission of LLR-eligible packets during the LLR replay window must not update the S_VC_CC counters (again).
3. A CC_Update packet may have been in-flight during or just prior to the “LLR discard window”. The sender’s S_VC_CC[x] and the receiver’s R_VC_CC[x] counters can become uncorrelated due to the dropping of LLR-eligible packets during the LLR discard window. Therefore, CC_Update packets received during the LLR discard window must always be dropped and must not cause any changes to the R_VC_CC[x] counters. To resolve this issue, CC_Update packets must be marked as LLR-eligible and data packets of type B are not allowed when both features are enabled.

Because LLR and CBFC are independent features, any given packet might have any combination of LLR eligibility and CBFC lossless behavior when both features are enabled. These combinations are summarized for data packets in Table 5-26 along with the actions required at the receiver during the LLR discard window and at the sender during the LLR replay window.

Table 5-26 - LLR and CBFC Combined Handling for Data Packets

Type	Packet properties	Receiver handling in LLR discard window	Sender handling in LLR replay window
A	LLR-ineligible and CBFC Best Effort[1,2]	Packet is accepted. No update to receiver’s CBFC counters.	Packet is not replayed.
B	LLR-ineligible and CBFC Lossless[3]	N/A	N/A
C	LLR-eligible and CBFC Best Effort[1,2]	Packet is discarded.	Packet is replayed.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Type	Packet properties	Receiver handling in LLR discard window	Sender handling in LLR replay window
		No update to receiver's CBFC counters.	No update to sender's CBFC counters.
D	LLR-eligible and CBFC Lossless	Packet is discarded. No update to receiver's CBFC counters.	Packet is replayed. No update to sender's CBFC counters.

Note:

1. CBFC best-effort packets (types A and C) never update any CBFC counters.
2. CBFC lossless counters are updated when the packet is first transmitted before the LLR re-transmission is triggered, and therefore must not be updated again during replay.
3. The combination of LLR-ineligible and CBFC Lossless is not allowed.

CC_Update packets must always be marked as LLR-eligible so that they carry an LLR sequence number and can be discarded if previous CBFC lossless packets (type D) were discarded just prior to the CC_Update packet arrival. In this case, the CC_Update packet itself triggers the start of the LLR discard window. The three allowed types of data packets in Table 5-26, (A, C, and D) are used in Figure 5-13 and Figure 5-14 to illustrate two examples of the combined LLR and CBFC handling during an LLR re-transmission event. In Figure 5-13, the CC_Update packet is received after the LLR-discard window has already been triggered. In Figure 5-14, the CC_Update packet is received as the first "out of sequence" LLR-eligible packet. In both cases, the CC_Update packet is initially discarded by the receiver and it is later replayed.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

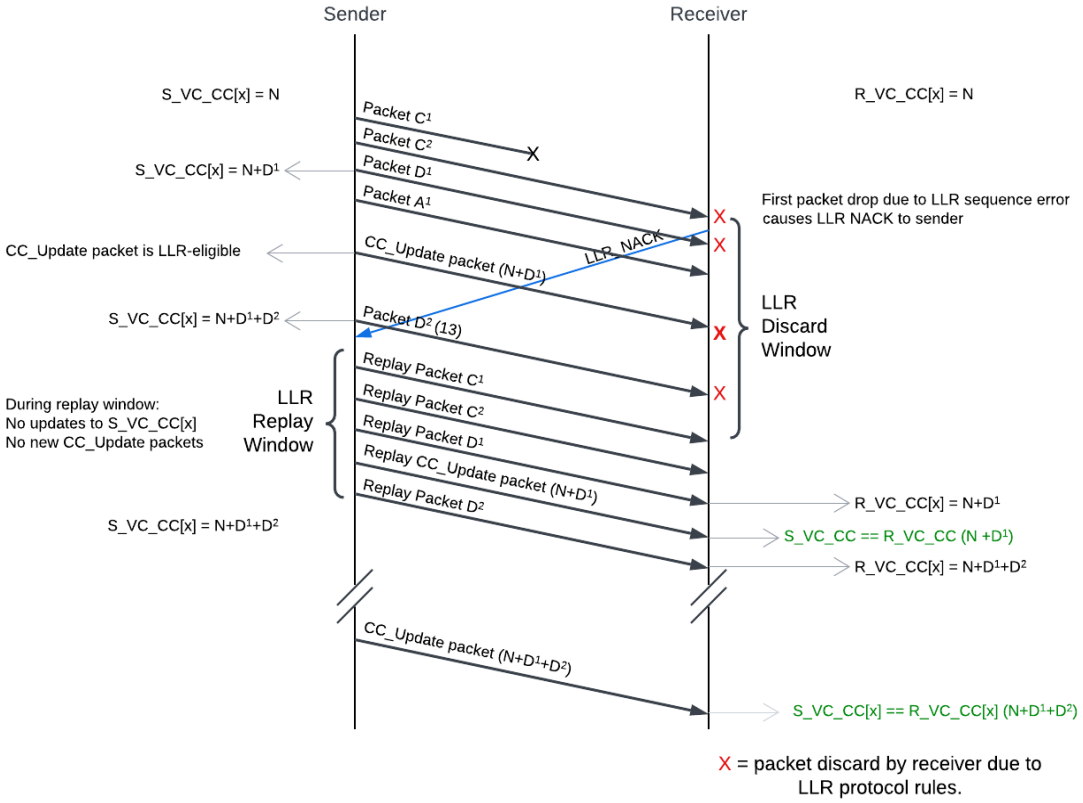


Figure 5-13 - Example CC_Update Packet During LLR Discard Window

When both CBFC and LLR are enabled, all CBFC lossless data packets are required to also be LLR-eligible; but this is the common case and is a reasonable expectation. This has two advantages:

- The CC_Update packet follows the usual LLR rules during the LLR discard window and is simply discarded as an LLR-eligible packet.
- The replay of the CC_Update packet produces a correct CC counter comparison and recovers any leaked credits rather than waiting for the next CC_Update interval (which may also hit an LLR replay event).

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

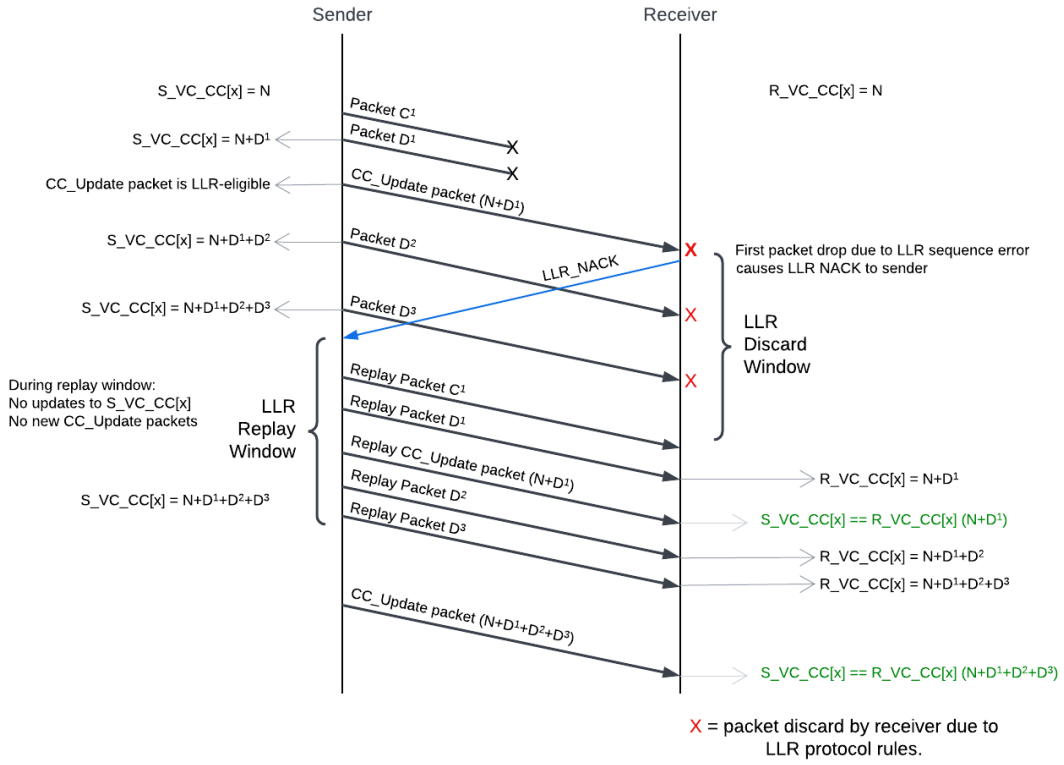


Figure 5-14 - Example CC_Update Packet Triggers the LLR Discard Window

For correct operation when both LLR and CBFC are enabled, the following rules are applied:

1. During the LLR replay window, the S_VC_CC[x] and S_P_CU counters must not be updated when retransmitting packets, because they were already counted when first transmitted.
2. During the LLR discard window, the R_VC_CC[x] counters must not be updated for LLR-eligible packets that are discarded. They will be counted when they are retransmitted and accepted at the receiver.
3. CC_Update packets must be LLR-eligible and carry a valid sequence number. During the LLR discard window, any received CC_Update packet must be discarded along with all other LLR-eligible packets. This prevents incorrect updates to the R_VC_CC[x] counters, because previously discarded LLR-eligible packets make the CC counter checks invalid.
4. During the LLR discard window, any correctly received CF_Update CtlOS must still be used to update the S_VC_CF[x] counters.
5. PFC control packets must be LLR-ineligible.
6. PFC control packets must belong to a best-effort VC.
7. During the LLR discard window, any correctly received PFC packet must still be acted upon. PFC control packets do not affect retransmission of LLR-eligible packets during the LLR replay window (see LLR specification) and affect only subsequent initial packet transmissions.
8. All CC_Update packets SHALL be configured as CBFC best effort. CC_Update packets do not use credits.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

9. During the LLR replay window, no new CC_Update packet shall be transmitted.

5.2.10 Compliance Requirements

5.2.10.1 CreditSize Parameter

The receiver's input buffer is typically divided into cells, and the cell size (in bytes) is the input buffer allocation unit. Packets smaller than the cell size, or larger packet sizes that are not evenly divisible by the cell size, end up wasting some buffer space: the "cell tax." To make optimal use of the receiver's input buffer, the CreditSize parameter should be set to the buffer cell size. However, requiring all implementations of CBFC to support any CreditSize in the sender logic is overly burdensome, and the CBFC mechanism works even if the CreditSize is not set to this "optimal" value. Implementations should support their own native cell size of the port input buffer but are not required to support all possible CreditSize values. See Section 5.2.5.8 for information on handling non-ideal CreditSize values at the sender and receiver.

CBFC requires support for a minimal set of CreditSize values, on powers of 2:

- 32, 64, 128, 256, 512, 1024 and 2048 bytes

In addition, to make better use of the credit mechanism and buffer storage, it is recommended to also support values at 8 B intervals from 32 B to 64 B, 16 B intervals from 64 B to 128 B, 32 B intervals from 128 B to 256 B, 64 B intervals from 256 B to 512 B, 128 B intervals from 512 B to 1024 B, and 256 B intervals from 1024 B to 2048 B.

Additional recommended CreditSize values:

- 40, 48, 56, 80, 96, 112, 160, 192, 224, 320, 384, 448, 640, 768, 896, 1280, 1536, and 1792.

Finer granularity is always better and is encouraged. The programmed CreditSize at the sender must be conservative with respect to the receiver's actual input buffer cell size, such that the programmed CreditSize ≤ (receiver's cell size).

5.2.10.2 Packet Overhead

An implementation shall support the packet overhead parameter, PktOvhd, in the range from -16 to +127 bytes. An implementation may use a PktOvhd value quantized to a granularity greater than one byte if it is conservative with respect to the receiver's actual per-packet overhead buffer requirement, such that the programmed PktOvhd ≥ (actual receiver packet overhead).

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

5.2.10.3 Compliance Requirements Summary

Table 5-27 - Conformance Requirements

Item	Requirement	Required Value(s)
Configuration Parameters		
Minimum number of VCs	MUST be able to support at least this number of VCs.	2 (one lossless VC and one best-effort VC)
Maximum number of VCs	CBFC does not support more than 32 VCs (2 ≤ NumVCs ≤ 32)	The maximum number of VCs supported is implementation-specific up to 32 lossless VCs.
Number of lossless VCs	MUST support a configurable number of lossless VCs up to NumVCs.	1 ≤ n ≤ NumVCs
VC mapping	All VCs must be configurable as either lossless or best effort.	
CreditSize	MUST support a configurable CreditSize, in bytes. More granularity than the required values is suggested (see Section 5.2.10.1).	CreditSize ∈ {32, 64, 128, 256, 512, 1024, 2048}
TotalCredits	MUST support a configurable total number of credits (determined by receiver).	1 ≤ TotalCredits ≤ (2^{19}-1)
VC_CreditLimit[x]	MUST support a configurable credit limit for each lossless VC[x].	1 ≤ VC_CreditLimit[x] ≤ TotalCredits
PktOvhd	MUST be able to support a configurable size of packet overhead in bytes.	-16 ≤ PktOvhd ≤ 127
Link-Level Messages		
CF_Update format	MUST support CF_Update CtlOS format as defined in Section 5.2.6.1.	
CF_min_timer	Receiver MUST support a configurable minimum time (bytes) between CF_Update CtlOS (applied to updated R_VC_CF[x] values).	CF_min_timer ≥ 800 B of wire time
CF_max_timer[1]	Receiver MUST support this timer or the per-VC timer. This timer is a configurable average maximum time (bytes) between CF_Update CtlOS (for all R_VC_CF[x] values).	16 KB to 1 MB with minimum granularity between 1 byte and 16384 bytes
CF_max_timer[x][1]	Receiver MUST support this per-VC timer or a single timer for all VCs. This timer is a configurable average maximum time (bytes) between CF_Update CtlOS for corresponding R_VC_CF[x] value.	16 KB to 1 MB with minimum granularity between 1 byte and 16384 bytes
CC_Update format	MUST support CC_Update packet format as defined in Section 5.2.6.2.	

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Item	Requirement	Required Value(s)
CC_msg_timer	MUST support a configurable period for CC_Update packet generation by the sender.	1 µs ≤ CC_msg_timer ≤ 250 ms
Counters		
S_P_CL	Sender Port Credit Limit	$0 \le S\_P\_CL \le (2^{20} - 1)$
S_VC_CL[x]	Sender VC Credit Limit	$0 \le S\_VC\_CL[x] \le (2^{20} - 1)$
S_VC_CC[x]	Sender VC Credits Consumed	$0 \le S\_VC\_CC[x] \le (2^{20} - 1)$
S_VC_CF[x]	Sender VC Credits Freed	$0 \le S\_VC\_CF[x] \le (2^{15} - 1)$
R_VC_CC[x]	Receiver VC Credits Consumed	$0 \le R\_VC\_CC[x] \le (2^{20} - 1)$
R_VC_CF[x]	Receiver VC Credits Freed	$0 \le R\_VC\_CF[x] \le (2^{15} - 1)$
VC mapping		
VLAN mapping	MUST support a flexible mapping of {VLAN.PCP, VLAN.DEI} to VC number for full range or any subset of PCP/DEI.	
DSCP mapping	MUST support a flexible mapping of **ip.dscp** to VC number for full range or any subset of DSCP.	
Note: 1. Implementations MUST support either a single CF_max_timer or a per-VC CF_max_timer[x] with a minimum granularity between 1 byte and 16384 bytes.		

5.2.11 Control Ordered Sets (CtlOS) in UE Link Layer

The CtlOS is a message mechanism utilized by the credit-based flow control (CBFC) and Link Layer Retry (LLR) features. The general format is extensible to future link-level features as well. It is an 8-byte message encoded as an ordered set in the 64b/66b PCS encoding with an O-code value to distinguish it from standard IEEE Std 802.3 [1]sequence ordered sets.

5.2.11.1 Control Ordered Sets Format

Control ordered sets use a control character of 0x5C on the xMII bus between the MAC/RS and PCS. This distinguishes them from the standard sequence ordered sets, which use a control character of 0x9C.

Table 5-28 - CtlOS Format on xMII

Ordered Set	Lane 0 (D0, control character)	Lane 1 (D1)	Lane 2 (D2)	Lane 3 (D3)	Lane 4 (D4)	Lane 5 (D5)	Lane 6 (D6)	Lane 7 (D7)
CtlOS	0x5C	Type	D2	D3	{D4[7:4], 0x6}	D5	D6	D7

The CtlOS type field is used to encode the CtlOS message type. Table 5-29 defines all UE CtlOS message type values. Each CtlOS message has 5.5 bytes in D2-D7 for message-specific information.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Table 5-29 - UE CtlOS Message Type Values

UE Feature	CtlOS	Msg type value (D1)
LLR	ACK	0x01
	NACK	0x02
	INIT	0x03
	INIT_ECHO	0x04
CBFC	CF_Update	0x10

The PCS 64B/66B encoding must be modified to translate an xMII Lane 0 control character of 0x5C to the block type of 0x4B with an O-code value of 0x6. This definition supports both IEEE sequence ordered sets and the new control ordered sets within the same structure. See the UE PHY specification for additional details.

Table 5-30 - Modified 64B/66B PCS Encoding for Ordered Sets

Input data	Sync	[2:9]	[10:17]	[18:25]	[26:33]	[34:37]	[38:41]	[42:49]	[50:57]	[58:65]
Ordered Set	‘b10	0x4B	D1	D2	D3	O-code	D4[4:7]	D5	D6	D7
Note: • O-code value is 0x0 for sequence ordered sets and 0x6 for UE control ordered sets.										

5.2.11.2 Link Layer and MAC CtlOS Insertion Rules

The following rules govern the insertion of CtlOS messages into the TX data stream.

- CtlOS must always be 8-byte aligned with the xMII control code in lane 0.
- CtlOS are allowed to preempt a packet except:
  - Within the first 256 bytes of a packet.
- A packet that is preempted by a CtlOS must continue transmission of data immediately following the CtlOS.
- CtlOS must be limited to less than 2% of link bandwidth.
  - Enforced by guaranteeing a minimum distance of at least 400 B between two CtlOS.
- CtlOS should use less than 0.5% of link bandwidth under usual circumstances.
  - Spacing of approximately 1600 bytes between two CtlOS.

Guaranteed spacing between two CtlOS has several advantages, including simplifying the RX CtlOS handling logic. The minimum spacing rules must be observed even if there are no packets in transit and the CtlOS would be separated only by IDLE characters.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

5.2.12 CBFC Message Examples (Informative)

5.2.12.1 CBFC CF_Update Message

CBFC CF_Update messages are formatted as an 8-byte CtlOS on the xMII as described in Section 5.2.6.1. In the following example, the CF_Update message contains the R_VC_CF counter values for two lossless VCs. The first VC, number 4, has an R_VC_CF value of 123, and the second VC, number 11, has an R_VC_CF value of 14203. As described in Table 5-21, this corresponds to CF1_VC_index=4, CF1_count=123 (0x7B), CF2_VC_index=11, and CF2_count=14203 (0x377B). The CF_Update CtlOS is formatted on the xMII as shown in Figure 5-15.

MII Fields:	D0	D1	D2	D3	D4	D5	D6	D7
bit position:	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
CF_Update fields:	Control Character	Type	CF1_VC	CF1_count	O-code	CF2_VC	CF2_count	<rsvd>
example (dec):	92	16	4	123	6	11	14203	0
example (hex):	0x5C	0x10	0x4	0x007B	0x6	0xB	0x377B	0x0
example (bin):	0 1 0 1 1 1 0 0	0 0 0 1 0 0 0 0	0 0 1 0 0 0 0 0	0 0 0 0 0 0 0 1	1 1 1 0 1 1 0 1	1 0 1 1 0 1 0 0	0 1 0 1 1 0 1 1	0 1 1 1 0 1 1 1	1 0 1 1 0 0 0 0
MII Fields:	D0	D1	D2	D3	D4	D5	D6	D7
MII bytes:	0x5C	0x10	0x20	0x07	0xB6	0x5B	0x77	0xB0

	TXD<0:63>
TXD<0> to <63>:	0 0 1 1 1 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 1 1 1 0 0 0 0 0 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 0 1 1 1 0 1 1 1 0 0 0 0 0 1 1 0 1

Figure 5-15 - Example CF_Update CtlOS

Each data byte on the xMII interface is transmitted from LSBit to MSBit, and from TXD<0> to TXD<63>.

5.2.12.2 BFC CC_Update Message

CBFC CC_Update messages are formatted as a 64-byte MAC frame as described in Section 5.2.6.2 and contain the S_VC_CC counter values for 16 VCs. For this example, the CC_Update fields have values as described below, which are formatted into eight data transfers of 8 bytes each on the xMII interface, as shown in Figure 5-16. Note that the frame data transfers are preceded by a single xMII transfer with the preamble+SFD and are followed by a single xMII transfer with the terminate (/T/) control character and seven idle (/I/) control characters.

This example uses these field values as defined in Table 5-22:

- MACDA = 01-80-C2-00-00-01
- EtherType = 0x8808 and Opcode = 0xFFFE
- CID = 0xFA7ACB
- CC_Update packet type = 0x01

The MACSA field is normally the MAC address of the source port as specified in Section 5.2.6.2. The value used here is for this example only:

- MACSA = 0A-0B-0C-0D-0E-0F

The 16 S_VC_CC values for this example are shown in decimal and hexadecimal format in Table 5-31.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Table 5-31 - CC_Update Message - Counter values

VC #	S_VC_CC (decimal)	S_VC_CC (hex)	VC #	S_VC_CC (decimal)	S_VC_CC (hex)
0	800008	0xC3508	8	100100	0x18704
1	100001	0x186A1	9	200200	0x30E08
2	200002	0x30D42	10	300300	0x4950C
3	300003	0x493E3	11	400400	0x61C10
4	400004	0x61A84	12	500500	0x7A314
5	500005	0x7A125	13	600600	0x92A18
6	600006	0x927C6	14	700700	0xAB11C
7	700007	0xAAE67	15	800800	0xC3820

Figure 5-16 shows the CC_Update frame format, followed by the example hexadecimal data, followed by the eight data transfers on the xMII bus (TXD<0:63>) for the above example values.

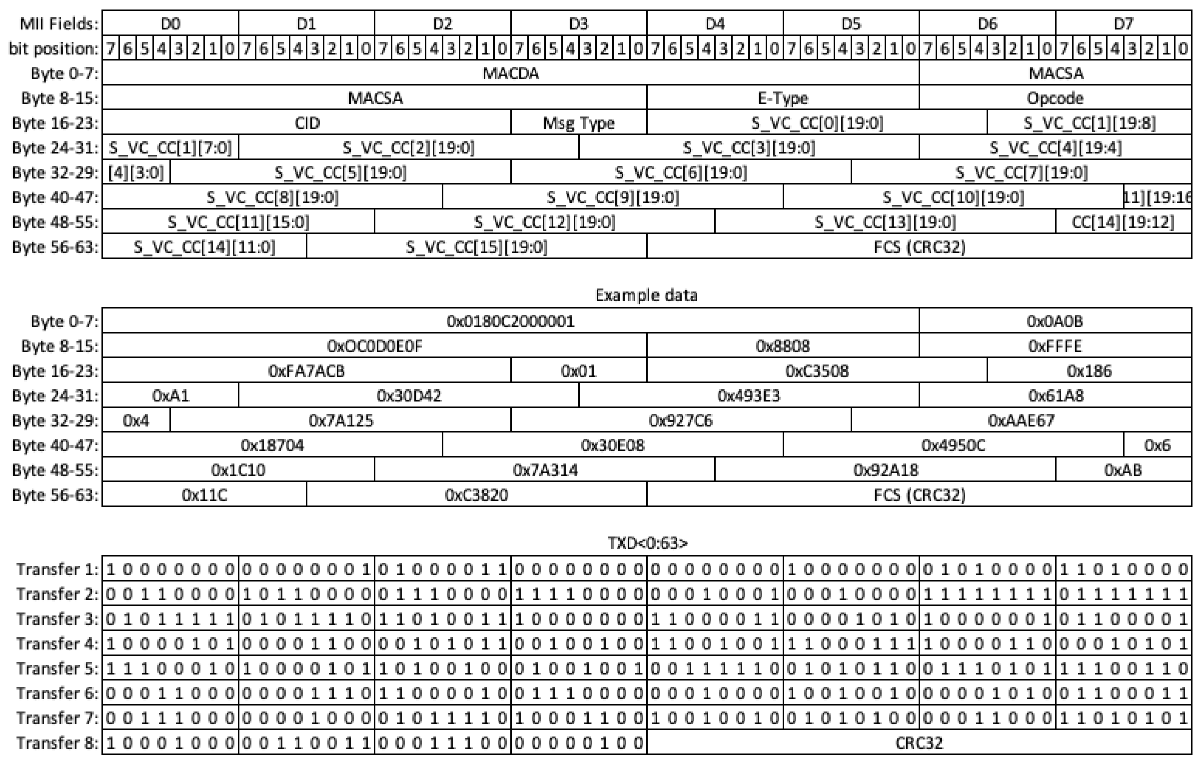


Figure 5-16 - Example CC_Update Frame

Each data byte on the xMII interface, with the exception of the FCS, is transmitted from LSBit to MSBit and from TXD<0> to TXD<63>.

5.2.13 References

[1] IEEE Std 802.3-2022, "IEEE Standard for Ethernet," 2022. [Online]. Available: https://ieeexplore.ieee.org/document/9844436.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

[2] IEEE Std 802.3db-2022, "IEEE Standard for Ethernet - Amendment 3: Physical Layer Specifications and Management Parameters for 100 Gb/s, 200 Gb/s, and 400 Gb/s Operation over Optical Fiber using 100 Gb/s Signaling," 2022. [Online]. Available: https://ieeexplore.ieee.org/document/9988984.

[3] IEEE Std 802.3ck-2022, "IEEE Standard for Ethernet Amendment 4: Physical Layer Specifications and Management Parameters for 100 Gb/s, 200 Gb/s, and 400 Gb/s Electrical Interfaces Based on 100 Gb/s Signaling," 2022. [Online]. Available: https://ieeexplore.ieee.org/document/9999414.

[4] IEEE Std P802.3df-2024, "IEEE Standard for Ethernet - Amendment 9: Media Access Control Parameters for 800 Gb/s and Physical Layers and Management Parameters for 400 Gb/s and 800 Gb/s Operation," 2024. [Online]. Available: https://ieeexplore.ieee.org/document/10472445.

[5] IEEE Std 802.1AB-2016, "IEEE Standard for Local and Metropolitan Area Networks - Station and Media Access Control Connectivity Discovery," 2016. [Online]. Available: https://ieeexplore.ieee.org/document/7433915.

[6] IEEE Std 802.1Q-2022, "IEEE Standard for Local and Metropolitan Area Networks – Bridges and Bridged Networks," 2022. [Online]. Available: https://ieeexplore.ieee.org/document/10004498.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

5.3 UE Link Negotiation

UE link negotiation defines the negotiation of optional link local features for the UE link layer that extend the operation of the UE link beyond those of a standard IEEE Std 802.3 Ethernet. These optional link local features can be used only on a UE link where both link partners support the optional features and desire to enable them.

This specification defines the negotiation of the following optional link local features:

1. Link layer retry (LLR)
2. Credit based flow control (CBFC)
3. Inter-packet gap reduction (IPGR). Also known as idle reduction
4. Congestion signaling (CSIG)
5. Unified forwarding header (UFH)

To support automatic discovery, negotiation, and selection among the optional link local features, a negotiation protocol is built on top of UE link layer configuration information exchanged by the IEEE Link Layer Discovery Protocol [1] , [2] , [9] . The UE link layer configuration information is exchanged by the Link Layer Discovery Protocol (LLDP) within UE Organizationally Specific Type-Length-Values (TLVs), which are formatted as specified in IEEE Std 802.1AB. The UE link configuration information, exchanged by LLDP and stored in the LLDP database, is in turn used to configure and enable optional link local features.

The management processes for each feature use the LLDP database to indicate their port's capabilities and configuration and to determine the capabilities and configuration of their remote link partner. The local and remote LLDP database information is used to enable and disable each feature requiring co-ordination between the link partners, deliver any parameters required to initialize the feature, and provide indication of capabilities available at each port.

TLV tuples describe the capabilities of a UE link, for example LLR and CBFC support, as well as configuration details such as the number of virtual channels supported by CBFC. The capabilities advertised using LLDP can be used by network management to map the network topology along with the capabilities available at each link, allowing the manager to determine paths that support the optional features.

Any device that supports the widely deployed IEEE LLDP can be easily extended (often via software only) to support this negotiation protocol. In the event that a device does not support the UE Link Negotiation Options TLVs, the use of UE link local optional features would be disabled by the negotiation protocol. This provides backward compatibility with standard IEEE 802.3 Link Layer devices that do not provide the UE organizationally specific LLDP TLVs. The use of the negotiation protocol can be overridden by configuration management to allow manual configuration of link features when necessary.

5.3.1 LLDP Overview

LLDP is a link layer protocol that allows an Ethernet device to advertise the capabilities and current status of the system at a port. The port provides the service to a logical link control (LLC) entity that in

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

turn directs LLDP frames to the LLDP entity based on the Ethertype 0x88CC. The information distributed and received in each LLDP protocol data unit (LLDPDU) is stored in one or more databases. Figure 5-17 illustrates LLDP and its relationship to its databases designed by the IETF, IEEE 802, UEC, OpenConfig, and others.

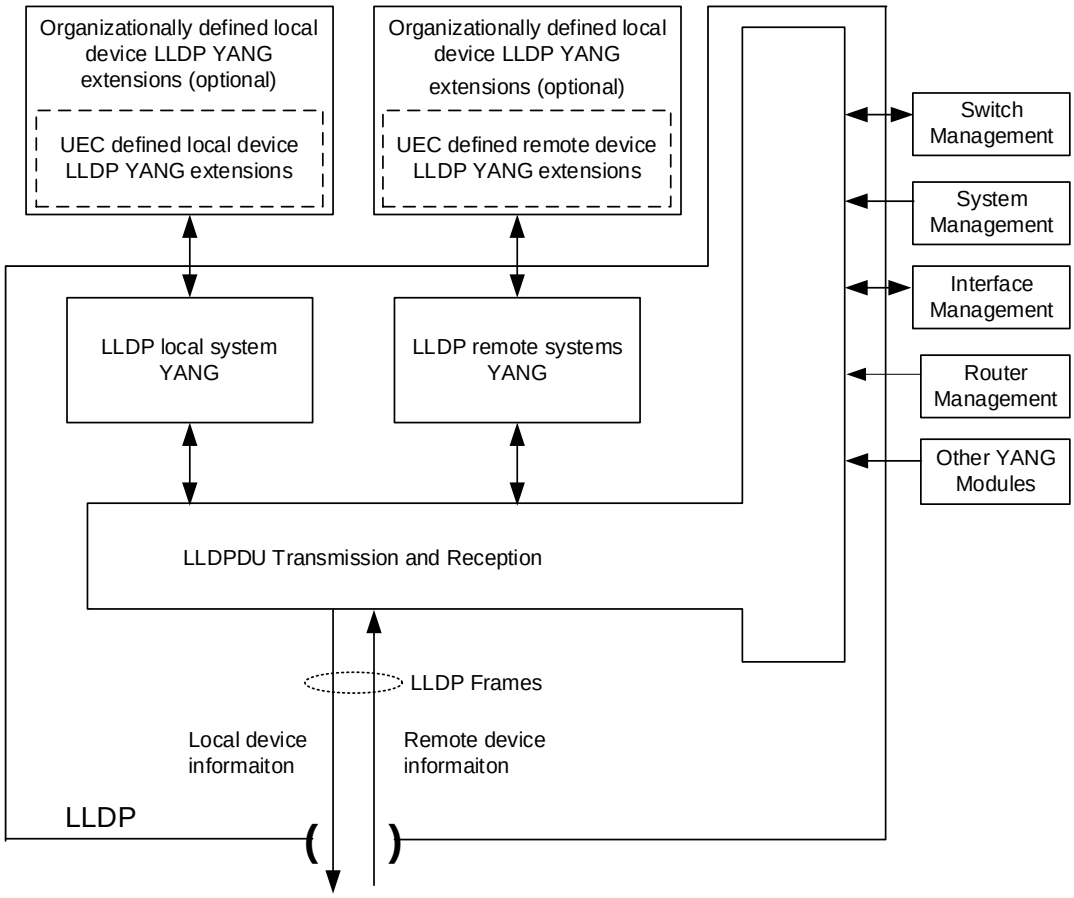


Figure 5-17 - LLDP Agent With the UE LLDP Databases

The UE LLDP database is defined under a UE organizationally defined YANG module. The information within the UE LLDP database is used both by the network management system to discover the capabilities and topology of the network and by the link partners to negotiate the operation of optional UE link features. UE feature negotiation is performed by the local management procedures of each individual feature using the local and remote UE LLDP database information.

The format of an LLDP frame is illustrated in Figure 5-18. The LLDPDU contains a series of TLVs formatted as described in [1] . These include three mandatory TLVs—Chassis ID, Port ID, and Time To Live — followed by a series of optional TLVs. The UE organizationally specific LLDP TLVs are included among the optional TLVs.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

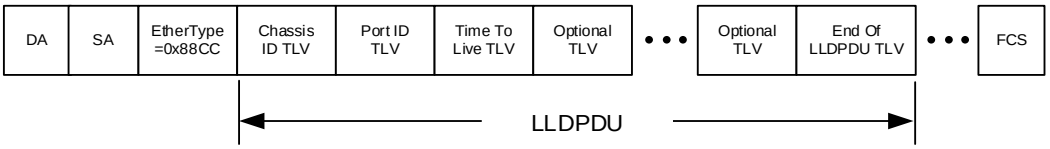


Figure 5-18 - Ethernet Frame Containing an LLDPDU

The destination address of the LLDPDUs carrying the UE organizationally specific LLDP TLVs SHALL use the nearest bridge group address listed in Table 5-32:

Table 5-32 - LLDP Database Group Addresses

IEEE 802.1 Group Address Name	Group Address
Nearest bridge group address	01:80:c2:00:00:0e
Nearest customer bridge group address	01:80:c2:00:00:00
Nearest non-TPMR bridge group address	01:80:c2:00:00:03

5.3.2 UE Organizationally Specific LLDP TLVs

The UE organizationally specific LLDP TLVs are used to:

i) Advertise capabilities available at a UE Link.
ii) Negotiate and activate common link capabilities.

The UEC defines the UE Link Negotiation Options TLV and the UE Link CBFC TLV for configuration of UE link features. The UE Link Negotiation Options TLV is used to discover and configure advanced UE link capabilities. The UE Link Negotiation CBFC TLV is used to discover and configure UE link credit-based flow control (CBFC).

5.3.2.1 UE Link Negotiation Options TLV (Options TLV)

The UE Link Negotiation Options TLV is exchanged via LLDP and conforms to the LLDP TLV specification [1] for organizationally specific TLVs. The UE Link Negotiation Options TLV is also called the Options TLV within this specification.

The Options TLV is used to advertise the following UE link features:

- Link layer retry (LLR): Provides link-level re-transmissions for packets with errors.
- Inter-packet gap reduction (IPGR): Also known as idle reduction. Reduces the number of idle bytes between frames.
- Congestion signaling (CSIG): L2 tag used to providing telemetry.
- Unified forwarding header (UFH): Provides a compressed L2/L3 header.

The Options TLV contains flags for each feature that use a suffix naming convention to designate their definition and use. Flags that do not require negotiation use the suffix “-C” to indicate the feature is available and enabled (i.e., “capable”). For “-C” flagged features, a companion “-S” flag may be provided to indicate the current operating state of the feature. Flags use the suffix “-E” to indicate the feature is

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

“enabled” and the suffix “-W” to indicate the feature is “wanted.” A UE link feature is negotiated using the information contained in the Options TLV. If the Options TLV is not present at either end of the link, the feature is disabled. If both ends of the link provide an Options TLV, each end of the link will see the ”-W”, “-C”, “-E” and ”-S” flags in both their local and remote databases.

Each port seeing both the local and remote ”-W” flag TRUE responds by setting the corresponding ”-E” flag to TRUE in their local database. After setting the “-E” flag in the local database, the port SHALL wait to see the corresponding ”-E” in the remote database before beginning to use the feature. The feature SHALL become operational only when both local and remote enables are TRUE.

Each optional feature defines how this process applies to the specific case and the steps required before full usage of the feature occurs. For example, the LLR logic uses inputs from the LLDP database to control its internal state machine for enabling or disabling LLR.

The current version 1 Options TLV updates the previous version 0 Options TLV to include features for inter-packet gap reduction, congestion signaling, and the unified forwarding header. The version 1 Options TLV also expands the LLR-E field to 2 bits, allowing representation of all the possible LLR operating options (see Table 5-33).

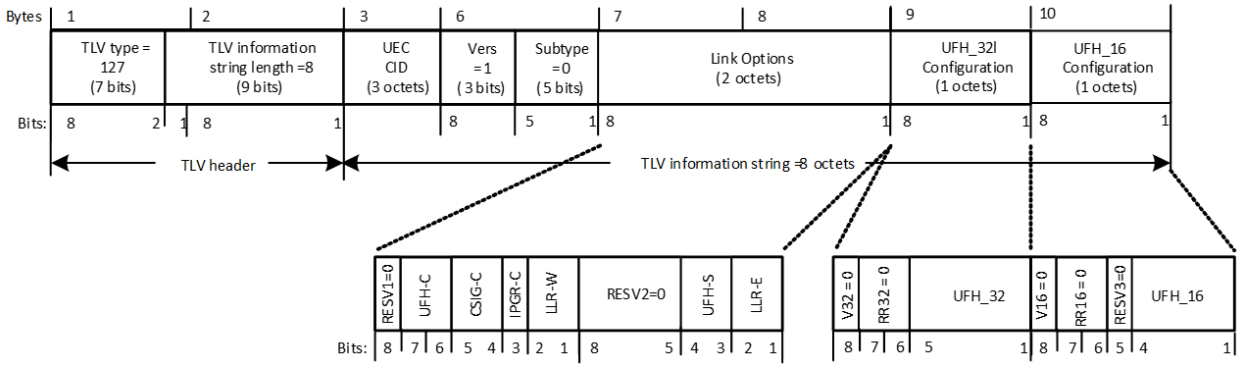


Figure 5-19 - UE Link Negotiation Options TLV (Options TLV)

5.3.2.1.1 TLV Type

The 7-bit IEEE-assigned TLV type 127 that indicates this TLV is encoded using an organizationally specific identifier format and subtype [1] .

5.3.2.1.2 TLV Information String Length

The 9-bit TLV information string length indicates the number of bytes following the information string length field. The version 1 Options TLV length is 8 bytes.

5.3.2.1.3 Company ID (CID)

The 24-bit UEC Company ID (CID) used to identify UE TLVs is FA-7A-CB.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

5.3.2.1.4 Subtype

The 5-bit subtype value 0 identifies the Options TLV. The UE subtype and version fields together occupy the 1-byte field specified by IEEE Std 802.1AB-2016 subclause 8.6.1 for the organizationally defined subtype. The subtype value 0x1F is reserved for extending the UE TLV subtype field.

5.3.2.1.5 Version

The 3-bit version value 1 identifies the amended version of the Options TLV, which supports LLR, IPGR, CSIG, and UFH. Version 1 is backward-compatible with version 0, which ignores all flags except LLR and never advertises IPGR, CSIG, or UFH capabilities. When a port with the version 1 options identifies a remote port advertising a version 0 TLV, the version 1 port SHALL operate with only the version 0 Options TLV feature set and SHALL NOT enable any feature outside the scope of version 0.

Future versions of the Options TLV SHALL be backward-compatible with version 0 and version 1. They can extend the length of the TLV to add additional information and/or provide definitions for currently reserved fields. Older versions ignore any updated reserved fields, unknown code points, or new information provided in the reserved fields of the TLV or in TLV length extensions.

5.3.2.1.6 UE Link Options

The UE link options field describes the UE link optional capabilities that are available at the port sending the Options TLV. In addition to the listed capabilities, some of the options can be enabled or disabled depending on availability at the remote link end. Enable flags are provided in the UE link options field to allow turning features on and off in synchronization with the remote link partner. An enable value of TRUE(1) indicates that the capability is running at the sender of the Options TLV; an enable value of FALSE(0) indicates that the capability is not running at the sender. A port that advertises it wants an optional capability SHALL enable the feature and set the enable flag to TRUE(1) only if the link partner indicates it wants to use the feature. The capabilities are as defined in the following subclauses.

5.3.2.1.6.1 LLR

The 2-bit LLR-W field has four possible values. A value of LLR-W = 0 means the capability is not wanted. A value of LLR-W = 3 means the capability is wanted. A version 0 or 1 UE link port SHALL NOT set LLR-W to the values 1 or 2. A version 0 UE link port receiving LLR-W with the values 0, 1, or 2 from the remote port SHALL disable LLR and set LLR-E to FALSE. A version 1 UE link port receiving LLR-W with the values 0, 1, or 2 from the remote port SHALL disable LLR by setting LLR-E to zero (0). If LLR-W is set to the value 3, the port is LLR-capable and wanting to use bidirectional LLR. For version 0, if the local and remote port's LLR-W = 3, the local port sets the 1-bit LLR-E flag to TRUE(1). For version 1, if the local and remote port's LLR-W = 3, the local port sets the 2-bit LLR-E = 3. Note when a version 0 and 1 are on opposite ends of the link, the version 1 implementation will accept LLR-E = 1 as enabled from the remote version 0 port. When a version 0 and 1 are on opposite ends of the link, the version 0 implementation will see the remote LLR-E = TRUE(1) when the remote end sets LLR-E = 3.

Table 5-33 - UE Link Negotiation Options TLV LLR-W Field Options

LLR-W Value	UE Link optional capability
0	This port does not want to use LLR.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

LLR-W Value	UE Link optional capability
1	Reserved for future specification.
2	Reserved for future specification.
3	This port wants bidirectional LLR (it both sends and receives).

Even if the local port has set the LLR-E flag, indicating the local port is set to use LLR, the local port SHALL NOT begin using LLR until the remote link partner indicates it has enabled LLR by setting its LLR-E flag to TRUE(3). Both link partners SHALL enable LLR before the LLR feature is operational.

5.3.2.1.6.2 IPGR

If the 1-bit IPGR-C is set TRUE(1), the sending port is capable of sending and enabled for receiving reduced idle bytes (see 5.5). A port SHALL NOT transmit reduced idles unless it sees the IPGR-C flag TRUE(1) in the remote LLDP database. If a port sees the remote port is capable of receiving a reduced number of idle bytes indicated by the IPGR-C flag being TRUE(1) in the remote LLDP database, the port seeing the remote IPGR-C flag TRUE(1) SHALL enable idle reduction on transmission if it is capable. If the LLDP Options TLV is NULL or is version 0, the link ends SHALL behave the same as if IPGR-C is set FALSE(0).

5.3.2.1.6.3 CSIG

The 2-bit CSIG-C flag can take one of three values (see Table 5-3). If CSIG-C is set to 0, the port cannot participate in CSIG. If CSIG-C=0 or if the LLDP Options TLV is NULL or if the LLDP Options TLV is version 0, then there isn't any forwarding behavior defined for CSIG. If CSIG-C=1, the port can forward and update only compact CSIG tags. If CSIG-C=2, the port can forward and update both compact and wide CSIG tags. The link negotiation supports only CSIG wide tags in combination with CSIG compact tags, as support for CSIG compact tags is mandatory for all implementations (see Section 4.2).

Table 5-34 - UE Link Negotiation Options TLV CSIG-C Field Options

CSIG-C Value	UE Link CSIG capability
0	The port cannot participate in CSIG (default value).
1	The port can participate in CSIG compact-tag updates only.
2	The port can participate in both CSIG compact and CSIG wide-tag updates.
3	Reserved

If the LLDP advertisement from one or both link partners is CSIG-C=0, then the link end is not capable of processing CSIG tags (though it might forward them); therefore, the link is not capable of forwarding packets with CSIG tags. If one link end is CSIG capable (CSIG-C=1 or CSIG-C=2) while the other is not (CSIG-C=0), then the link end that is CSIG capable SHALL NOT transmit packets with CSIG tags. If the LLDP Options TLV is NULL or is Version 0, the link end SHALL behave the same as if CSIG-C=0.

If the LLDP advertisements from one link partner is CSIG-C=1 and the other link partner advertises CSIG-C=1 or CSIG-C=2, then the CSIG-C=1 link end is capable of processing and forwarding only compact CSIG tags; therefore, the link is capable of forwarding only packets with compact CSIG tags. In this case both

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

ends of the link SHALL NOT send any wide CSIG tags and SHALL log an error if any wide CSIG tags are received.

If the LLDP advertisements from both link partners have CSIG-C=2, then both ends of the link are able to process and forward packets with compact and wide CSIG tags; therefore, the link is capable of forwarding both compact and wide CSIG tags.

5.3.2.1.6.4 UFH

The 2-bit UFH-C field can take one of four values (see Table 5-4). If UFH-C is set to 0, the port cannot participate in forwarding the unified fabric header (UFH). If UFH-C=0 or if the LLDP Options TLV is NULL or if the LLDP Options TLV is version 0, then the port isn't capable of forwarding UFH encoded packets. If UFH-C=1, the port can forward only UFH_32-encoded packets. If UFH-C=2, the port can forward only UFH_16-encoded packets. If UFH-C=3, the port can forward both UFH_32- and UFH_16-encoded packets.

A port SHALL discard all UFH-encoded frames before transmitting them onto the link until it has both the local and remote Options TLV. Once a port sees both the local and remote UFH-C values along with the UFH configuration parameters, it SHALL choose an operating mode that is common to both ports, if one exists; set the local UFH-S to reflect that operational state by setting the UFH-S to the appropriate value from table 5-4; and stop discarding the UFH encoded packets that conform to the chosen operational mode. If there is no common operating mode, UFH-S=0 SHALL be set in the local database and transmitted to the link partner. For an operating mode to be valid, the link partners MUST be using the same number of bits, as indicated by the UFH_32 and UFH_16 configuration fields, and be running the same UFH version, indicated by the V32 and V16 bits.

Table 5-35 - UE Link Negotiation Options TLV UFH-C and UFH-S Field Options

UFH-C or UFH-S Value	UE Link UFH capability or status
0	The port cannot participate in UFH (default value).
1	The port can participate in UFH_32 forwarding.
2	The port can participate in UFH_16 forwarding.
3	The port can participate in both UFH_32 and UFH_16 forwarding.

The UFH_32 parameter indicates how many address bits are used for addressing in the UFH_32 encoding. The range of the UFH_32 parameter is 4 to 32 bits where 32 is represented by UFH_32=0. The UFH_16 parameter indicates how many address bits are used for addressing in the UFH_16 encoding. The range for the UFH_16 parameter is 4 to 16 bits where 16 is represented by UFH_16=0. The V32 and V16 parameters indicate the version of the UFH_32 and UFH_16 encoding respectively. The first version is indicated by a 0 in the V32 or V16 fields, while a future version is indicated by a 1 in the V32 or V16 field.

If the LLDP advertisements from both link partners have the same UFH capabilities, indicated by the value of UFH-C, then both ends of the link are able to forward packets with the indicated UFH encoding. In addition to having matching encodings for the link to exchange packets, the link partners SHALL also

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

have matching versions and matching address bit lengths as specified by the V32, UFH_32, V16, and UFH_16 UFH configuration parameters. For UFH_32 encoding, the V32 and UFH_32 parameters SHALL match. For UFH_16 encoding, the V16 and UFH_16 parameters SHALL match. If both link partners support both UFH_32 and UFH_16 encoding, both encodings SHALL be used only if the V32, UFH_32, V16, and UFH_16 parameters match. If both link partners support UFH_32 and UFH_16 encoding, but if one of the parameter sets—V32/UFH_32 or V16/UFH_16—does not match. then the link SHALL forward only packets that are encoded with the matching parameter set.

If the LLDP advertisements from both link partners have different UFH capabilities, indicated by the value of the UFH-C, the operation SHALL be limited to the packet encoding supported by both ends of the link. The link partners SHALL NOT transmit UFH encoded packets onto the link that are not part of the packet encoding supported by both link partners. If the LLDP advertisement from both link partners have different UFH capabilities without any common encoding, then the ports SHALL NOT transmit packets with UFH encoding on the link.

Each port sets the UFH-S flag in its local database to a value from Table 5-4 that matches its operational state.

The 2-bit RR32 and RR16 fields of the LLDP TLV SHALL be transmitted as zero and ignored on receipt.

5.3.2.1.6.5 RESV1
A 1-bit reserved field that SHALL be transmitted as 0 and ignored on receive.

5.3.2.1.6.6 RESV2
A 4-bit reserved field that SHALL be transmitted as 0 and ignored on receive.

5.3.2.1.6.7 RESV3
A 1-bit reserved field that SHALL be transmitted as 0 and ignored on receive.

5.3.2.2 UE Link Negotiation CBFC TLV (CBFC TLV)
The UE Link Negotiation CBFC TLV is exchanged via LLDP and conforms to the LLDP TLV specification [1] for organizationally specific TLVs. The UE Link Negotiation CBFC TLV is also called the CBFC TLV within this specification.

The CBFC TLV provides the capabilities to:

- Determine if CBFC is supported by the presence or absence of the CBFC TLV in both the local and remote LLDP databases.
- Exchange the receiver buffer parameters including the buffer size (cell size), per-packet overhead, and total credits (each of cell size) assigned to CBFC.
- Negotiate the sender credit size and packet overhead.
- Identify each CBFC virtual channel (VC) the receiver wants to support, along with the VC identifier, state, queue, and packet stream selector, and optionally the per-VC credit limit.
- Allow the sender to indicate to the receiver when the sender is ready to start sending to the wanted VC(s).

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

- Allow the receiver to indicate to the sender when the receiver is ready to begin operation of the wanted VC(s).
- Allow the receiver to modify the credit limit on each VC independently.
- Allow the receiver to stop CBFC on each VC and to re-activate each VC independently.

The CBFC clause provides a complete explanation of how the CBFC TLV exchange is used to configure CBFC service on a link. The CBFC TLV contains all the parameters required to initialize CBFC for each virtual channel. Configuration of VCs is performed independently for each direction on the link. The initialization begins when the receiver sets R_VC_WANT = TRUE(1) for the VC(s) desiring CBFC service. The sender seeing R_VC_WANT = TRUE in its remote database from the remote link partner configures the requested VC(s), then acknowledges the configuration by setting S_VC_RTS = TRUE(1) for the requested VC(s) in its local database. The receiver seeing S_VC_RTS = TRUE for its requested VC(s) in its remote database from its remote link partner configures itself based on the parameters provided by the sender, then indicates it is ready to receive by setting R_VC_RTR = TRUE(1) for the VC(s) in its local database. From this point onward the sender uses CBFC for transmissions on the configured VCs. The sender SHALL NOT set S_VC_RTS = TRUE for a VC unless the remote receiver has set R_VC_WANT = TRUE for the VC. The receiver SHALL NOT set R_VC_RTR = TRUE for a VC unless the remote sender has set S_VC_RTS = TRUE for the VC. The sender SHALL NOT use CBFC for a VC unless the remote receiver has set R_VC_RTR = TRUE for the VC.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

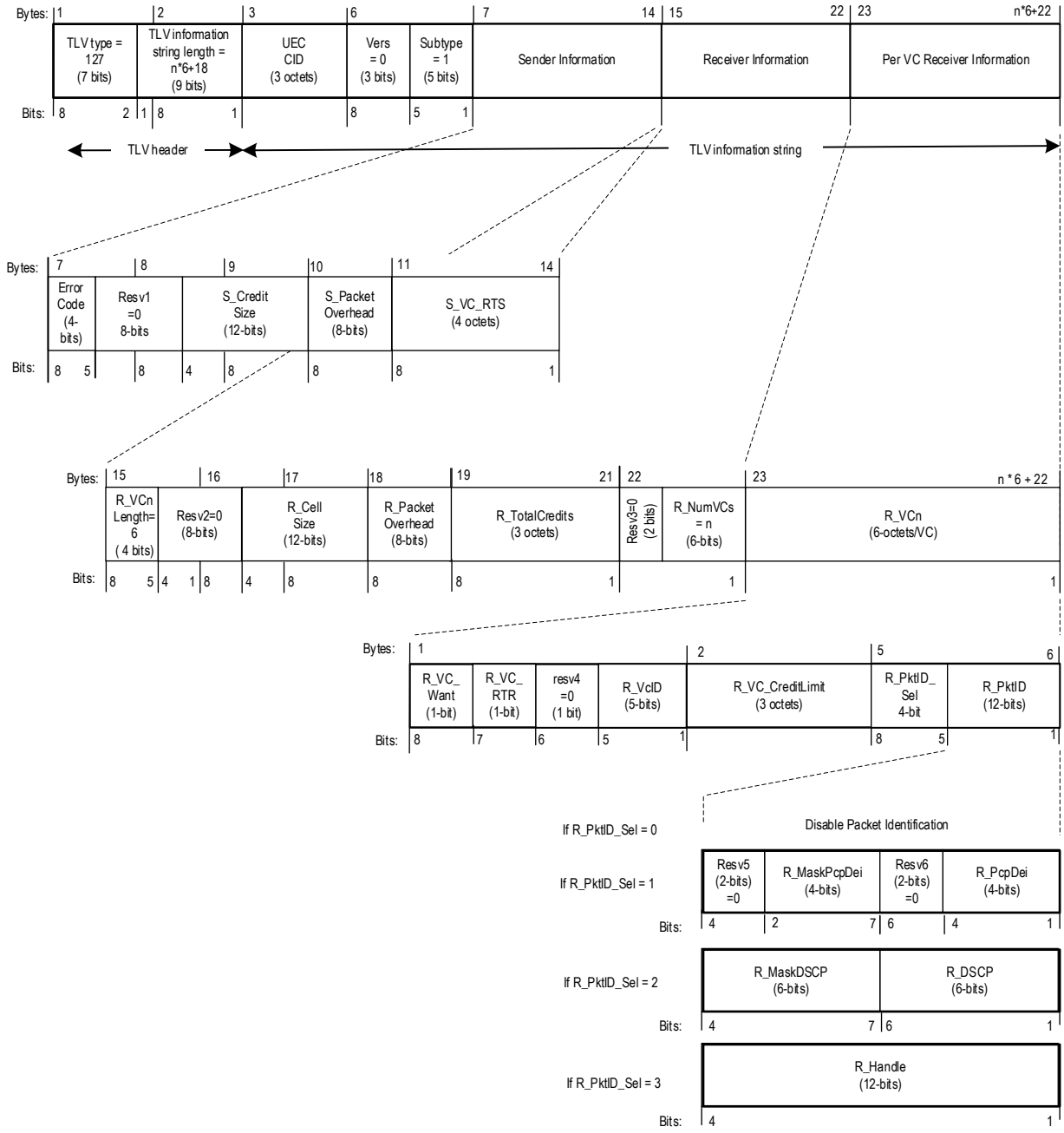


Figure 5-20 - UE Link Negotiation CBFC TLV (CBFC TLV)

5.3.2.2.1 TLV Type

The 7-bit IEEE-assigned TLV type 127 indicating this TLV is encoded using an organizationally specific identifier format and subtype [1] .

5.3.2.2.2 TLV Information String Length

The 9-bit TLV information string length indicates the number of bytes following the information string length field. The CBFC TLV string length depends on the number of receive VCs desired. For 0 <= n <= 32

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

where n is the number of receive VCs listed by this port, the string length is n*6+20. The maximum CBFC TLV information string length is 212 bytes. The maximum total TLV length is 214 bytes.

5.3.2.2.3 Company ID (CID)

The 24-bit UEC Company ID (CID) used to identify UE TLVs is FA-7A-CB.

5.3.2.2.4 Subtype

The 5-bit subtype value 1 identifies the CBFC TLV. The UE subtype and version fields together occupy the 1-byte field specified by IEEE Std 802.1AB-2016 subclause 8.6.1 for the organizationally defined subtype. The subtype value 0x1F is reserved for extending the UE TLV subtype field.

5.3.2.2.5 Version

The 3-bit version used to identify the initial version of the CBFC TLV is 0. Future versions of the TLV SHALL be backward-compatible with version 0. They can extend the length of the TLV to add additional information and definitions for reserved fields. Older versions ignore any updated reserved fields, code points, or TLV length extensions.

5.3.2.2.6 Error_Code

The 4-bit Error_Code is used by the CBFC sender to indicate error conditions to the remote CBFC receiver. If no error is detected the Error_Code = 0. The error codes are:

Table 5-36 - UE Link Negotiation CBFC Error Codes

Value	CBFC Sender Meaning
0x0	No error.
0x1	CBFC sender is unable to support the receiver cell size.
0x2	CBFC sender is unable to support a requested VC.
0x3	CBFC sender is unable to support the credit pool size.
0x4	CBFC sender is unable to support a requested VC credit limit.
0x5	CBFC sender and CBFC receiver packet identification do not match.
0x6	Other error condition.
0x7-0xF	Reserved for future UE specification.

5.3.2.2.7 S_CreditSize

The 12-bit S_CreditSize field identifies the number of bytes in a single credit at the sending port. When S_CreditSize = 0, the CBFC sender has not yet set its credit size. A CBFC sending port waits to see the CBFC receiver cell size, R_CellSize, in the remote LLDP database before selecting its credit size. The S_CreditSize SHALL be valid when any of the S_VC_RTS[x] flags are TRUE. While any S_VC_RTS[x] flag is TRUE, the S_CreditSize SHALL NOT change.

5.3.2.2.8 S_PacketOverhead

The 8-bit S_PacketOverhead field identifies the number of overhead bytes per packet at the sending port. The S_PacketOverhead SHALL be valid when any of the S_VC_RTS[x] flags are TRUE. While any S_VC_RTS[x] flag is TRUE, the S_PacketOverhead SHALL NOT change.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

5.3.2.2.9 S_VC_RTS

The 32-bit S_VC_RTS field contains 32 ready-to-send flags, one for each of the 32 possible VCs. The S_VC_RTS flag in bit position 0 indicates the VC0 CBFC sender state. When the S_VC_RTS flag is TRUE(1), the sender is ready to send but SHALL wait to begin until the receiver sets R_VC_RTR=TRUE.

5.3.2.2.10 R_VCnLength

The 4-bit R_VCnLength field specifies the length of each element of the per-VC fields R_VCn. For version 0 the length is 6 bytes. Future versions can increase the length. Version 0 SHALL skip over any bytes beyond 6 when reading the R_VCn fields.

5.3.2.2.11 R_CellSize

The 12-bit R_CellSize field identifies the number of bytes in a single credit at the receiving port.

5.3.2.2.12 R_PacketOverhead

The 8-bit R_PacketOverhead field identifies the number of overhead bytes at the receiving port.

5.3.2.2.13 R_TotalCredits

The 24-bit R_TotalCredits field identifies the total number of credits, of credit size, available for all VCs at the receiver.

5.3.2.2.14 R_NumVCs

The 6-bit R_NumVCs field indicates the number of receiver VC configuration subfields (R_VCn) following in this TLV.

5.3.2.2.15 R_VCn

R_VCn represents R_NumVCs fields, one for each VC the receiver at this port wants to activate or catalog. Each R_VCn field contains parameters for configuring CBFC for this VC as listed in the following subclauses.

5.3.2.2.15.1 R_VC_Want

The 1-bit R_VC_Want field indicates if the receiver wants CBFC lossless service for this VC. If TRUE(1), this port wants CBFC lossless receiver service for this VC on this port. If FALSE(0) this VC is best effort. If R_VC_Want transitions to FALSE(0), the sender SHALL stop using CBFC credits to qualify transmission of packets on this VC, set the corresponding S_VC_RTS flag to FALSE(0), and return to best-effort forwarding.

5.3.2.2.15.2 R_VC_RTR

If the 1-bit R_VC_RTR field is TRUE, this VC receiver is ready. The R_VC_RTR field SHALL NOT be set TRUE(1) until after a corresponding S_VC_RTS is set TRUE by the remote link partner. If R_VC_RTR transitions to FALSE(0), the sender SHALL stop using CBFC credits to qualify transmission of packets on this VC.

5.3.2.2.15.3 R_VcID

The 5-bit R_VcID field indicates the receive VC number configured by the parameters in R_VCn. VC numbers range from 0 to 31. Implementations are not required to support all 32 possible VCs.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

5.3.2.2.15.4 R_PktID_Sel

The 4-bit R_PktID_Sel field indicates the type of identifier for the VC. Table 5-37 shows the possible R_PktID_Sel options. This is an optional service that provides assurance that both the CBFC sender and receiver are configured to count credits on the same packets. If the link partners do not agree on the packets that belong to each VC, then the credit counts will be incorrect and the link can lose packets even with CBFC.

Table 5-37 - UE Link Negotiation CBFC TLV R_PktID_Sel Field Options

Value	Meaning
0x0	Disable packet to VC identification matching.
0x1	The R_PcpDei contains a 4-bit PCP/DEI value used by this VC. The R_MaskPcpDei parameter identifies which R_PcpDei bits are valid.
0x2	The R_DSCP contains a 6-bit DSCP value used to identify this VC. The R_MaskDSCP parameter identifies which DSCP bits are valid.
0x3	The R_Handle contains a 12-bit stream handle identifying this VC. The handle identifies a stream handle such as specified in IEEE Std 802.1CBdb-2021 clause 6 [10] .
0x4-0xF	Reserved for future specification by UE.

5.3.2.2.15.5 R_PktID

The R_PktID contains one of three sets of fields that are used to identify the packets to VC mapping. The type of field is determined by the R_PktID_Sel parameter. The R_PktID information is provided by the receiver for each receiver VC and can be used by the sender in one of three ways depending on the local configuration of the sender's CBFC management agent. The sender MAY:

1) Ignore the packet to VC mapping.
2) Compare the mapping to its own sending VC configuration to verify agreement.
3) Dynamically install the mapping for its sending VC.

If the receiver elects to disable packet-to-VC mapping identification by setting R_PktID_Sel = 0, then the sender cannot use dynamic packet identification. In the event the receiver disables packet identification while the sender is configured for dynamic mapping, then the sender disables the VC and reports an error.

5.3.2.2.15.6 R_MaskPcpDei

Selected if R_PktID_Sel = 1. The 4-bit R_MaskPcpDei field identifies the bits of the L2 PCP/DEI fields that are valid for identifying this VC. The DEI bit is the least significant bit of the R_MaskPcpDei field.

5.3.2.2.15.7 R_PcpDei

Selected if R_PktID_Sel = 1. The 4-bit R_PcpDei field contains the L2 packet PCP/DEI value after the R_MaskPcpDei mask is applied identifying this VC. The DEI bit is the least significant bit of the R_PcpDei field. The set of packets that are members of the VC are those where (PcpDei & R_MaskPcpDei) == R_PcpDei is TRUE.

5.3.2.2.15.8 R_MaskDSCP

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Selected if R_PktID_Sel = 2. The 6-bit R_MaskDSCP field identifies the bits of the L3 DSCP field that are valid for identifying this VC.

5.3.2.2.15.9 R_DSCP

Selected if R_PktID_Sel = 2. The 6-bit R_DSCP field contains the L3 packet DSCP value after the R_MaskDSCP mask is applied identifying this VC. The set of packets that are members of the VC are those where (DSCP & R_MaskDSCP) == R_DSCP is TRUE.

5.3.2.2.15.10 R_Handle

Selected if R_PktID_Sel = 3. The 12-bit R_Handle is an opaque handle that can be used as an identifier for a packet match method. The R_Handle is intended to cover complex packet match cases that can't be expressed by a simple mask and match of the PCP, DEI, and DSCP bits. The use of R_Handle requires pre-arranged values by network management on each side of the link. These pre-arranged R_Handle lists MUST match on both sides of the link. In some cases, a standard packet match method can be used, such as stream identification as defined in IEEE Std 802.1CB-2017 [12] and IEEE Std 802.1CBdb-2021 clause 6 [10] . In other cases, the methods can be proprietary. There is no requirement to support any particular packet match method or identifier list.

5.3.2.2.15.11 R_VC_CreditLimit

The 24-bit R_VC_CreditLimit field identifies the credits available for this VC receiver. A receiver SHALL NOT set both R_VC_CreditLimit and R_TotalCredits to non-zero values. CBFC operates with either R_VC_CreditLimit or R_TotalCredits, but not both.

5.3.2.2.15.12 Resv

All reserved fields are sent as 0 and ignored on receive.

5.3.3 UE LLDP YANG

This clause specifies the YANG modules that provide control and status monitoring of UE link options. The YANG objects are based on the TLVs detailed in clause 2. This YANG model uses the YANG 1.1 data modeling language as specified in [3] . The YANG framework applies hierarchy by using a uniform resource name (URN) [8] , a private uniform resource identifier (URI), and the YANG objects that form a hierarchy.

The YANG structure that incorporates the UE link YANG modules is represented by Figure 5-21. In this diagram, white boxes are defined by the IETF, gray boxes are defined by the IEEE, and hashed boxes are defined by the UE. As shown in the diagram, the UE link YANG augments IEEE LLDP management [2] .

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

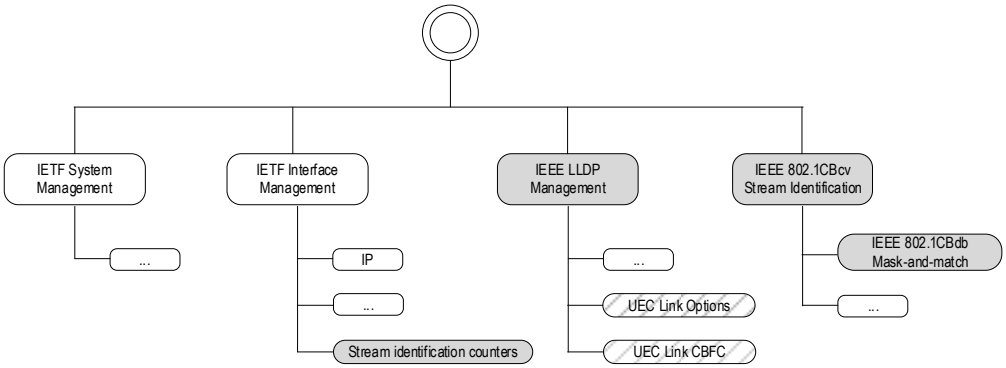


Figure 5-21 - YANG Root Hierarchy with UE Link LLDP Extensions

5.3.3.1 Models for UE LLDP Extension TLV YANG Modules

A UML-like representation of the management model is provided in the following subclauses. The purpose of the UML-like diagram is to show the model design on a single piece of paper. The structure of the UML-like representation shows the name and type of the object followed by the object accessibility. In the UML-like representation, a box with a white background represents information that comes from sources outside UE. A box with a gray background represents objects that are defined by the UEC.

5.3.3.1.1 UE Link Negotiation Options TLV Model

The attributes for the UE Link Negotiation Options TLV object set are obtained from both system-wide and per-port objects. A UML-like representation of the management model for the UE Link Negotiation Options TLV is provided in this subclause. The model augments the IEEE LLDP port model. The UML for the UE extension objects is derived from the UML specified in the IEEE Std 802.1AB and is shown in Figure 5-22.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

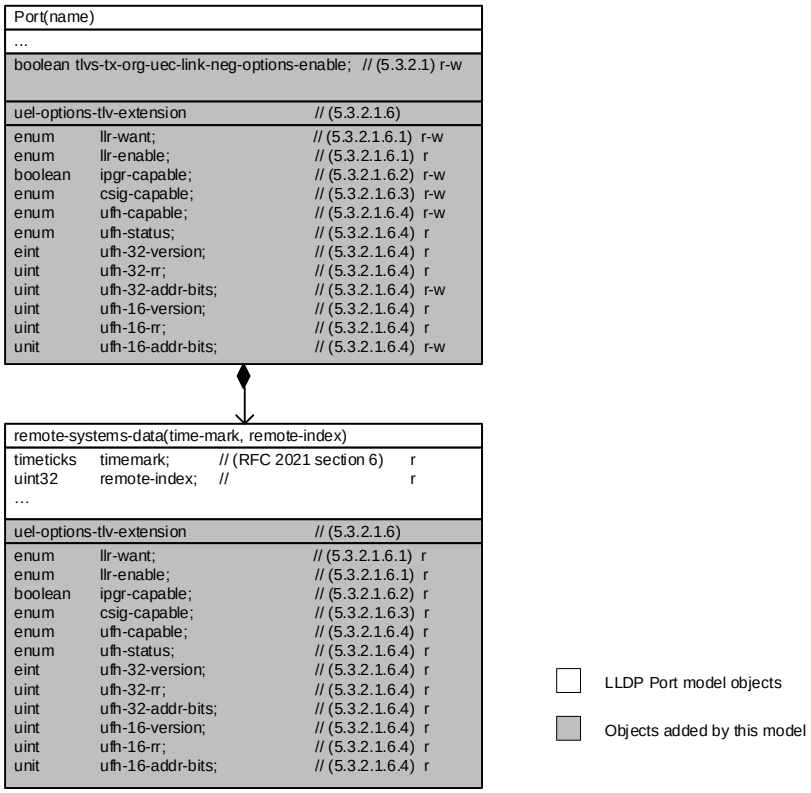


Figure 5-22 - UE Link Negotiation Options TLV Model

5.3.3.1.2 UE Link Negotiation CBFC TLV Model

The attributes for the UE Link Negotiation CBFC TLV are obtained from both system-wide and per-port objects. A UML-like representation of the management model for the UE TLV is provided in this subclause. The model augments the IEEE LLDP port model. The UML for the UE extension objects is derived from the UML specified in the IEEE Std 802.1AB and is shown in Figure 5-23.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

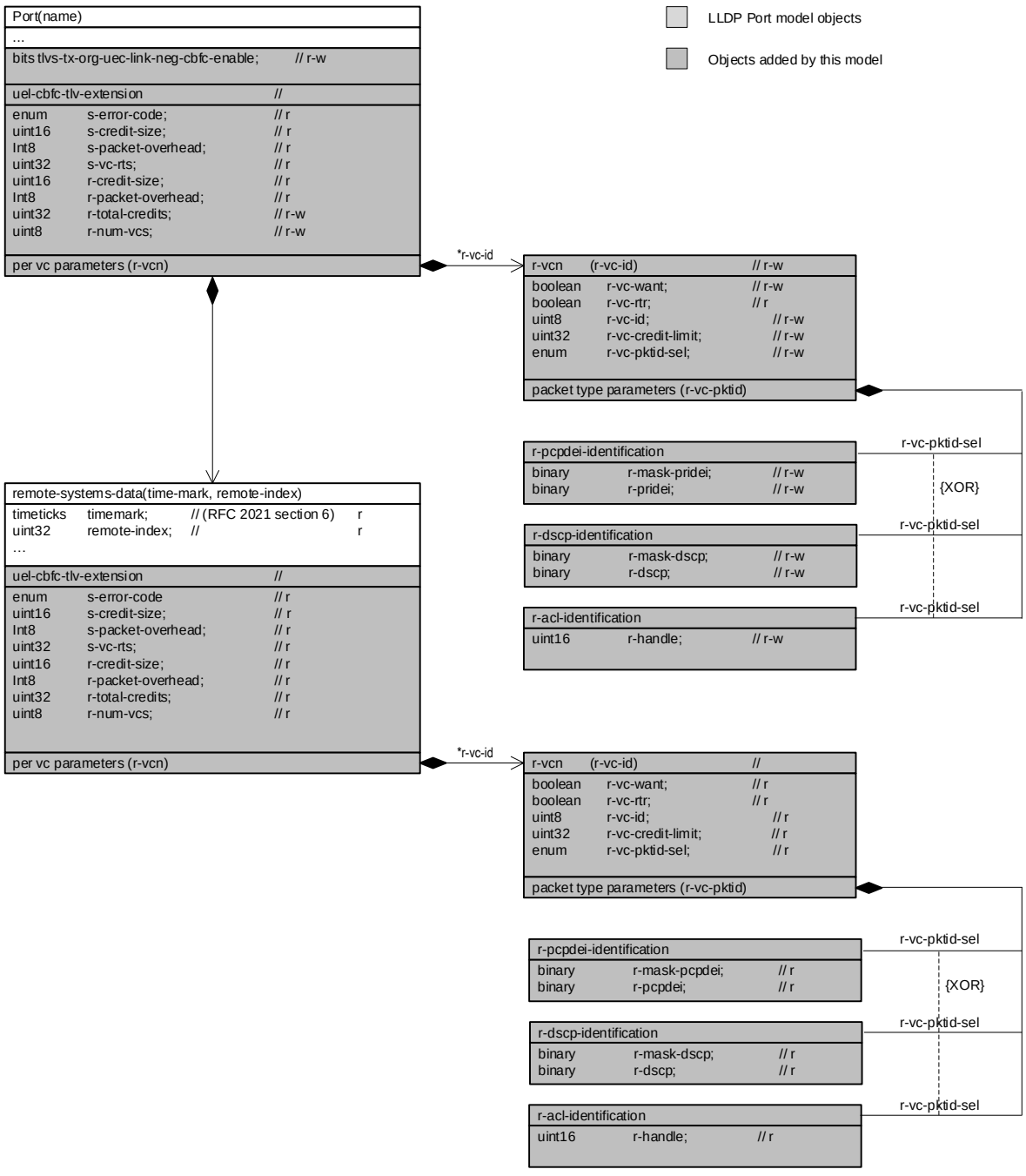


Figure 5-23 - UE Link Negotiation CBFC TLV Model

5.3.3.2 Security Considerations for UE LLDP Extension YANG Modules

The YANG modules defined in this clause are designed to be accessed via a network configuration protocol, e.g., NETCONF protocol [5] . In the case of NETCONF, the lowest NETCONF layer is the secure transport layer, and the mandatory-to-implement secure transport is SSH [6] . The NETCONF access

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

control model [7] provides the means to restrict access for particular NETCONF users to a preconfigured subset of all available NETCONF protocol operations and content.

It is the responsibility of a system's implementor and administrator to ensure that the protocol entities in the system that support NETCONF, and any other remote configuration protocols that make use of these YANG modules, are properly configured to allow access only to those principals (users) that have legitimate rights to read or write data nodes. This standard does not specify how the credentials of those users are to be stored or validated.

Several management objects defined in the `uec-link-neg-options-tlv` and `uec-link-neg-cbfc-tlv` YANG modules are configurable (i.e., read-write) and/or operational (i.e., read-only). Such objects may be considered sensitive or vulnerable in some network environments. A network configuration protocol such as NETCONF can support protocol operations that can edit or delete YANG module configuration data (e.g., edit-config, delete-config, copy-config). If this is done in a non-secure environment without proper protection, it is possible to result in negative effects on the network operation.

5.3.3.3 Schema for UE LLDP Extension YANG Modules

A simplified graphical representation of the data model is used in this document. The meaning of the symbols in these diagrams is as follows:

- Brackets "[" and "]" enclose list keys.
- Abbreviations before data node names: "rw" means configuration (read-write), and "ro" means state data (read-only).
- Symbols after data node names: "?" means an optional node, "!" means a presence container, and "*" denotes a list and leaf-list.
- Parentheses enclose choice and case nodes, and case nodes are also marked with a colon (":").
- Ellipsis ("...") stands for contents of subtrees that are not shown.

5.3.3.3.1 UE Link Negotiation Options TLV Schema

```
module: uec-link-neg-options-tlv

  augment /lldp:lldp/lldp:port:
    +--rw tlvs-tx-org-uec-link-neg-options-enable?   boolean
    +--rw uel-options-tlv-extension
       +--rw llr-want?          identityref
       +--ro llr-enable?        identityref
       +--rw ipgr-capable?      boolean
       +--rw csig-capable?      identityref
       +--rw ufh-capable?       identityref
       +--ro ufh-status?        identityref
       +--ro ufh-32-version?     identityref
       +--ro ufh-32-rr?          uint32
       +--rw ufh-32-addr-bits?   uint32
       +--ro ufh-16-version?     identityref
       +--ro ufh-16-rr?          uint32
       +--rw ufh-16-addr-bits?   uint32
```


Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

```
  augment /lldp:lldp/lldp:port/lldp:remote-systems-data:
    +--ro uel-options-tlv-extension
       +--ro llr-want?          identityref
       +--ro llr-enable?        identityref
       +--ro ipgr-capable?      boolean
       +--ro csig-capable?      identityref
       +--ro ufh-capable?       identityref
       +--ro ufh-status?        identityref
       +--ro ufh-32-version?     identityref
       +--ro ufh-32-rr?          uint32
       +--ro ufh-32-addr-bits?   uint32
       +--ro ufh-16-version?     identityref
       +--ro ufh-16-rr?          uint32
       +--ro ufh-16-addr-bits?   uint32
```

5.3.3.3.2 UE Link Negotiation CBFC TLV Schema

```
module: uec-link-neg-cbfc-tlv

  augment /lldp:lldp/lldp:port:
    +--rw tlvs-tx-org-uec-link-neg-cbfc-enable?   boolean
    +--rw uel-cbfc-tlv-extension
       +--rw s-error-code?                       identityref
       +--ro s-credit-size?                      uint16
       +--ro s-packet-overhead?                  int8
       +--ro s-vc-rts?                           uint32
       +--ro r-cell-size?                        uint16
       +--ro r-packet-overhead?                  int8
       +--rw r-total-credits?                    uint32
       +--rw r-num-vcs?                          uint8
       +--rw r-vcs* [r-vc-id]
          +--rw r-vc-want?                       boolean
          +--ro r-vc-rtr?                        boolean
          +--rw r-vc-id                          uint8
          +--rw r-vc-credit-limit?               uint32
          +--rw r-vc-pktid-sel?                  identityref
          +--rw (paremeters)
             +--:(pcpdei-identification)
             |  +--rw pcpdei-identification
             |     +--rw r-mask-pcpdei?         binary
             |     +--rw r-pcpdei?              binary
             +--:(dscp-identification)
             |  +--rw dscp-identification
             |     +--rw r-mask-dscp?           binary
             |     +--rw r-dscp?                binary
             +--:(handle-identification)
                +--rw handle-identification
                   +--rw r-handle?              uint16
  augment /lldp:lldp/lldp:port/lldp:remote-systems-data:
    +--ro uel-cbfc-tlv-extension
       +--ro s-error-code?                       identityref
       +--ro s-credit-size?                      uint16
       +--ro s-packet-overhead?                  int8
       +--ro s-vc-rts?                           uint32
       +--ro r-per-vc-field-length?              uint8
```


Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

```
      +--ro r-cell-size?                        uint16
      +--ro r-packet-overhead?                  int8
      +--ro r-total-credits?                    uint32
      +--ro r-num-vcs?                          uint8
      +--ro r-vcs* [r-vc-id]
         +--ro r-vc-want?                       boolean
         +--ro r-vc-rtr?                        boolean
         +--ro r-vc-id                          uint8
         +--ro r-vc-credit-limit?               uint32
         +--ro r-vc-pktid-sel?                  identityref
         +--ro (paremeters)
            +--:(pcpdei-identification)
            |  +--ro pcpdei-identification
            |     +--ro r-mask-pcpdei?          binary
            |     +--ro r-pcpdei?               binary
            +--:(dscp-identification)
            |  +--ro dscp-identification
            |     +--ro r-mask-dscp?            binary
            |     +--ro r-dscp?                 binary
            +--:(handle-identification)
               +--ro handle-identification
                  +--ro r-handle?               uint16
```

5.3.3.4 UE Link Negotiation Extension YANG Modules

5.3.3.4.1 YANG Module uec-link-neg-options-tlv

```
module uec-link-neg-options-tlv {
  yang-version "1.1";
  namespace "http://ultraethernet.org/yang/uec-link-neg-options-tlv";
  prefix uel-options-tlv;
  import ieee802-dot1ab-lldp {
    prefix lldp;
  }
  organization
    "Ultra Ethernet Consortium";
  contact
    "UEC-URL: https://ultraethernet.org/
    Contact: UEC Link Layer Working Group Chair";
  description
    "UEC LLDP extension TLV for the LLR, IPGR, CSIG, and UFH
    features

    Copyright @ UEC (2026).

    This version of this YANG module is part of UE Specification v1.1,
        see the specification itself for full legal notices.";
  revision 2026-04-08 {
    description
      "LLDP extension TLV for UEC link negotiation options.
      Included options are Link Layer Retry (LLR), Interpacket Gap
      Reduction (IPGR), Congestion Signaling (CSIG),
      and Unified Forwarding Header (UFH).
      Published as part of UEC 1.1";
    reference
      "UEC 1.1";
```


Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

```
}
revision 2025-03-14 {
  description
    "LLDP extension TLV for UE link negotiation options.
    Included options are Link Layer Retry (LLR).
    Published as part of UE Specification v1.0";
  reference
    "UE Specification v1.0";
}
identity llr-want-selector {
  description
    "Specify if LLR service is wanted by this port.";
}
identity llr-no {
  base llr-want-selector;
  description
    "Indicates LLR service is not wanted by this port. Signaled as the
      value 0x0.";
}
identity llr-want {
  base llr-want-selector;
  description
    "Indicates LLR is wanted at the port. Signaled as the value 0x3.";
}
identity llr-resv {
  base llr-want-selector;
  description
    "Reserved LLR selection values 0x1 and 0x2 for future specification.";
}
identity llr-enable-selector {
  description
    "Specify if LLR service is enabled at this port.";
}
identity llr-disabled {
  base llr-enable-selector;
  description
    "Indicates LLR service is not enabled at this port.
    Signaled as the value 0x0.";
}
identity llr-enabled {
  base llr-enable-selector;
  description
    "Indicates LLR is wanted at the port. Signaled as the value 0x3.";
}
identity llr-enable-resv {
  base llr-enable-selector;
  description
    "Reserved LLR selection values 0x1 and 0x2 for future specification.";
}
identity csig-capable-selector {
  description
    "Specifies the CSIG capability on this port.";
}
identity csig-no {
  base csig-capable-selector;
  description
    "Indicates CSIG service is not available at this port.
```


Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

```
    Signaled as the value 0x0.";
}
identity csig-compact {
  base csig-capable-selector;
  description
    "Indicates only CSIG compact tag service is available at this port.
    Signaled as the value 0x1.";
}
identity csig-compact-wide {
  base csig-capable-selector;
  description
    "Indicates CSIG compact and wide tag service is available at this port.
    Signaled as the value 0x2.";
}
identity csig-resv {
  base csig-capable-selector;
  description
    "Indicates CSIG compact and wide tag service is available at this port.
    Signaled as the value 0x3.";
}
identity ufh-version-selector {
  description
    "Specifies the UFH version on this port.";
}
identity ufh-first-version {
  base ufh-version-selector;
  description
    "Indicates UFH is the first version. Signaled as the value 0x0.";
}
identity ufh-update-version {
  base ufh-version-selector;
  description
    "Indicates UFH is an updated version. Signaled as the value 0x1.";
}
identity ufh-selector {
  description
    "Specifies the UFH encoding on this port.";
}
identity ufh-no {
  base ufh-selector;
  description
    "Indicates no UFH encoding. Signaled as the value 0x0.";
}
identity ufh-32 {
  base ufh-selector;
  description
    "Indicates 32-bit, UFH encoding format.
    Signaled as the value 0x1.";
}
identity ufh-16 {
  base ufh-selector;
  description
    "Indicates 16-bit, UFH encoding format.
    Signaled as the value 0x2.";
}
identity ufh-32-16 {
  base ufh-selector;
```


Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

```
  description
    "Indicates both 32-bit and 16-bit format.
    Signaled as the value 0x3.";
}
grouping uel-options-tlv {
  description
    "UE link negotiation options TLV";
  reference
    "UE Specification v1.1";
  leaf llr-want {
    type identityref {
      base llr-want-selector;
    }
    description
      "Indicates if Link Layer Retry service is wanted at this port.";
  }
  leaf llr-enable {
    type identityref {
      base llr-enable-selector;
    }
    config false;
    description
      "Indicates if Link Layer Retry is enabled on this port.";
  }
  leaf ipgr-capable {
    type boolean;
    description
      "Indicates this port is capable of receiving reduced
       Interpacket Gaps.";
  }
  leaf csig-capable {
    type identityref {
      base csig-capable-selector;
    }
    description
      "Indicates the CSIG capability available at this port.";
  }
  leaf ufh-capable {
    type identityref {
      base ufh-selector;
    }
    description
      "Indicates the capabilities of this port.
       A 0 indicates this port is not capable of processing the Unified
       Forwarding Header (UFH). A 1 indicates this port is UFH_32
       encode capable. A 2 indicates this port is UFH_16 encode
       capable. A 3 indicates this port is both UFH_32 and UFH_16
       capable.";
  }
  leaf ufh-status {
    type identityref {
      base ufh-selector;
    }
    config false;
    description
      "Indicates the current operating configuration of this port.
       A 0 indicates this port is not configured to process the Unified
```


Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

```
      Forwarding Header (UFH). A 1 indicates this port is configured
      to process UFH_32 encoding. A 2 indicates this port is
      configured to process UFH_16 encoding. A 3 indicates
      this port is configured to process both UFH_32 and UFH_16
      encoding.";
  }
  leaf ufh-32-version {
    type identityref {
      base ufh-version-selector;
    }
    config false;
    description
      "Version for UFH_32 encoding. The initial UFH version = 0
       and any future versions = 1.";
  }
  leaf ufh-32-rr {
    type uint32 {
      range "0 .. 3";
    }
    config false;
    description
      "Reserved bits for UFH_32 encoding. Set to zero by version 0.";
  }
  leaf ufh-32-addr-bits {
    type uint32 {
      range "4 .. 32";
    }
    default "32";
    description
      "Address bits for UFH_32 encoding: 32 bits.";
  }
  leaf ufh-16-version {
    type identityref {
      base ufh-version-selector;
    }
    config false;
    description
      "Version for UFH_16 encoding. The initial UFH version = 0
       and any future versions = 1.";
  }
  leaf ufh-16-rr {
    type uint32 {
      range "0 .. 3";
    }
    config false;
    description
      "Reserved bits for UFH_16 encoding. Set to zero by
      version 0.";
  }
  leaf ufh-16-addr-bits {
    type uint32 {
      range "4 .. 16";
    }
    default "16";
    description
      "Address bits for UFH_16 encoding: 16 bits.";
  }
```


Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

```
  }
  augment "/lldp:lldp/lldp:port" {
    description
      "Augments LLDP port with the UE link negotiation options TLV";
    leaf tlvs-tx-org-uec-link-neg-options-enable {
      type boolean;
    description
      "Enable for the UE link negotiation options TLV transmission";
    }
    container uel-options-tlv-extension {
      description
        "The UE link negotiation options TLV";
      uses uel-options-tlv;
    }
  }
  augment "/lldp:lldp/lldp:port/lldp:remote-systems-data" {
    description
      "Augments the port's remote-systems-data with received UE link
       negotiation options TLV";
    container uel-options-tlv-extension {
      description
        "Holds a received UE link negotiation options TLV";
      uses uel-options-tlv;
    }
  }
}
```

5.3.3.4.2 YANG Module uec-link-neg-cbfc-tlv

```
module uec-link-neg-cbfc-tlv {
  yang-version "1.1";
  namespace "http://ultraethernet.org/yang/uec-link-neg-cbfc-tlv";
  prefix uel-cbfc-tlv;
  import ieee802-dot1ab-lldp {
    prefix lldp;
  }
  organization
    "Ultra Ethernet Consortium";
  contact
    "UEC-URL: https://ultraethernet.org/
    Contact: UEC Link Layer Working Group Chair";
  description
    "UE LLDP extension TLVs for the CBFC feature

    Copyright © UEC (2025).

    This version of this YANG module is part of UE Specification v1.0.
    See the specification itself for full legal notices.";
  revision 2025-03-14 {
    description
      "UE LLDP extension TLV for the Credit Based Flow Control
      feature.";
      reference
      "UE Specification v1.0";
  }
```


Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

```
identity r-pktid-selector {
  description
    "Specify the type of VC packet identifier.";
}
identity r-no-packet-identification {
  base r-pktid-selector;
    description
      "Indicates packet identification over LLDP is disabled
      for this VC[x]. Packets are identified by the port
      configuration on each side of the link without any
      LLDP verification.";
}
identity r-pcpdei-identification {
  base r-pktid-selector;
  description
    "Indicates the packet identification for this VC[x]
    is based on mask and match to the PCP/DEI bits.
    Signaled as the value 0x1.";
}
identity r-dscp-identification {
  base r-pktid-selector;
  description
    "Indicates the packet identification for this VC[x]
    is based on mask and match to the DSCP bits.
    Signaled as the value 0x2.";
}
identity r-handle-identification {
  base r-pktid-selector;
  description
    "Indicates the packet identification for this VC[x]
    is based on a matching opaque handle which can
      match a stream handle as specified by IEEE Std 802.1CB
    or other system defined packet identification.
    Signaled as the value 0x3.";
}
identity error-code-selector {
  description
      "Specify the type of error.";
}
identity error-no {
  base error-code-selector;
  description
      "No error. Signaled as value 0.";
}
identity error-cell-size {
  base error-code-selector;
  description
      "The sender is unable to support the receiver
      cell size. Signaled as value 1.";
}
identity error-vc {
  base error-code-selector;
  description
      "The sender cannot support a requested VC.
    Signaled as value 2.";
}
identity error-credit-pool {
```


Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

```
  base error-code-selector;
  description
      "The sender cannot support the credit pool size.
      Signaled as value 3.";
}
identity error-credit-limit {
   base error-code-selector;
  description
      "The sender cannot support the credit
      limit. Signaled as value 4.";
}
identity error-packet-id {
  base error-code-selector;
  description
      "The sender packet identification does not
      match the receiver. Signaled as value 5.";
}
identity error-other {
  base error-code-selector;
  description
      "Other CBFC error detected by sender.
    Signaled as value 6.";
}
grouping uel-cbfc-tlv {
  description
    "UE Link CBFC TLV data";
  reference
    "UE 1.0 Specification";
  leaf s-error-code {
    type identityref {
        base error-code-selector;
    }
    description
      "Indicates either no error or the type of error
          detected at the sender.";
  }
  leaf s-credit-size {
    type uint16 {
      range "0..4094";
    }
    config false;
    description
      "The credit size used by the sender for credit accounting.
      When the value is 0 the sender has not configured a
      credit size. The receiver cannot proceed until the
      sender sets its credit size based on the receiver's
      credit size. The sender credit size is valid when
    s-vc-rts != 0, meaning some VC[x] flag(s) are true.";
  }
  leaf s-packet-overhead {
    type int8 {
      range "-127..127";
    }
    config false;
    description
      "The packet overhead used by the sender for credit
      accounting. The receiver cannot proceed until the
```


Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

```
  sender sets its intended packet overhead based on the
  receiver's packet overhead. The sender packet overhead
  is valid when s-vc-rts != 0, meaning some VC[x] flag(s)
  are true.";
}
leaf s-vc-rts {
  type uint32;
  config false;
  description
    "This port has enabled CBFC transmission for the VCs identified
    by these bit flags. The bit positions n=0-31 indicate VC[n]";
}
leaf r-cell-size {
  type uint16 {
    range "1..4095";
  }
  config false;
  description
    "The cell size used by the receiver. The receiver
    cell size is the receiver's cell/buffer size.";
}
leaf r-packet-overhead {
  type int8 {
    range "-127..127";
  }
  config false;
  description
    "The packet overhead used by the receiver.";
}
leaf r-total-credits {
  type uint32 {
    range "1..16777215";
  }
  description
    "The number of pooled receiver credits for all VCs.";
}
leaf r-num-vcs {
  type uint8  {
    range "1..32";
  }
  description
    "The number of receive VC parameter lists following.";
}
list r-vcs {
  key r-vc-id;
  description
    "A 6-tuple of fields in the UE Link Neg CBFC TLV that
    provides the per VC receiver parameters needed to support
    Credit Based Flow Control on the VC identified by r-vcid.";
  leaf r-vc-want {
    type boolean;
    description
      "When FALSE indicates the receiver want to use best effort
    service on VC[r-vcid]. When TRUE indicates the receiver wants
      to use lossless service on VC[r-vc-id].";
  }
  leaf r-vc-rtr {
```


Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

```
  type boolean;
  config false;
  description
    "When FALSE indicates this VC[r-vcid] has not completed
    configuration. When TRUE indicates this VC[r-vc-id] has
    completed configuration.";
}
leaf r-vc-id {
  type uint8  {
    range "0..31";
  }
  description
    "The index for this receive VC.";
}
  leaf r-vc-credit-limit {
  type uint32 {
    range "0..16777215";
  }
  description
    "The credit limit for this VC[r-vc-id] receiver.
    If set to 0 there is no per VC credit limit used.";
}
leaf r-vc-pktid-sel {
  type identityref {
    base r-pktid-selector;
  }
  description
    "Indicates how packets are identified for this VC[r-vc-id].
        The value 0 disables packet identification.";
}
choice paremeters {
  mandatory true;
  description
    "Three methods to verify or set the packets associated with
    the VC[r-vc-id]. These methods are selected by r-pktid-sel.
  Each of the three methods have a different set of parameters.
  The methods are:
    1)identification by priority / drop eligible, 2)identification
  by DSCP,3)identification by the exchange of an opaque handle
  which can be an IEEE mask-and-match stream handle as defined
  in 802.1CBdb or any other system handle.";
  container pcpdei-identification {
    description
      "Provides mask and match for the pcp/dei bits.";
    leaf r-mask-pcpdei {
      type binary {
        length '4';
      }
      description
        "A 4-bit mask for the pri/dei bits.";
    }
    leaf r-pcpdei {
      type binary {
        length '4';
      }
      description
        "The r-pcpdei value identifying the L2 packet
```


Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

```
            PCP/DEI value, after the r-mask-pcpdei mask is applied,
            which identifies the packets that are members of this VC.";
        }
      }
      container dscp-identification {
        description
          "Provides mask and match for the DSCP bits.";
        leaf r-mask-dscp {
          type binary {
            length '6';
          }
          description
            "A 6 bit mask for the DSCP bits.";
        }
        leaf r-dscp {
          type binary {
            length '6';
          }
          description
            "The r-dscp value identifying the L3 packet
            DSCP value, after the r-mask-dscp mask is applied,
            which identifies the packets that are members of this VC.";
        }
      }
      container handle-identification {
        description
          "Provides an opaque handle used to
          identifies the packets.";
        leaf r-handle {
          type uint16 {
            range "0..4095";
          }
          description
            "A handle used to identify a stream handle or other packet
            identifier which identifies the packets that are members
            of this VC.";
        }
      }
    }
  }
}
augment "/lldp:lldp/lldp:port" {
  description
    "Augments the LLDP port's local data with UE Link CBFC TLV";
  leaf tlvs-tx-org-uec-link-neg-cbfc-enable {
    type boolean;
    description
      "Enable for the UE Link CBFC TLV transmission";
  }
  container uel-cbfc-tlv-extension {
      description
        "Holds the local UE LLDP CBFC TLV data";
        uses uel-cbfc-tlv;
  }
}
augment "/lldp:lldp/lldp:port/lldp:remote-systems-data" {
  description
```


Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

```
      "Augments the LLDP port's remote-systems-data with received
        UE Link CBFC TLV";
    container uel-cbfc-tlv-extension {
      description
        "Holds the received UE LLDP CBFC TLV data";
      uses uel-cbfc-tlv;
    }
  }
}
```

5.3.4 References

5.3.4.1 Normative References

[1] IEEE Std 802.1AB-2016, "IEEE Standard for Local and Metropolitan Area Networks - Station and Media Access Control Connectivity Discovery," 2016. [Online]. Available: https://ieeexplore.ieee.org/document/7433915.

[2] IEEE Std 802.1ABcu-2021, "IEEE Standard for Local and Metropolitan Area Networks - Station and Media Access Control Connectivity Discovery, Amendment 1: YANG Data Model," 2021. [Online]. Available: https://ieeexplore.ieee.org/document/9756407.

[3] IETF RFC 7950, "The YANG 1.1 Data Modeling Language," 2016. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc7950.

[4] IETF RFC 8343, "A YANG Data Model for Interface Management," 2018. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc8343.

[5] IETF RFC 6241, "The YANG 1.1 Data Modeling Language," 2011. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc6241.

5.3.4.2 Informative References

[6] IETF RFC 6242, "Using the NETCONF Protocol over Secure Shell (SSH)," 2011. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc6242.

[7] IETF RFC 6536, "Network Configuration Protocol (NETCONF) Access Control Model," 2012. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc6536.

[8] IETF RFC 4122, "A Universally Unique Identifier (UUID) URN Namespace," 2005. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc4122.

[9] IEEE Std 802.1ABdh-2021, "IEEE Standard for Local and Metropolitan Area Networks – Station and Media Access Control Connectivity Discovery, Amendment 2: Support for Multiframe Protocol Data Units," 2021. [Online]. Available: https://ieeexplore.ieee.org/document/9760302.

[10] IEEE Std 802.1CBdb-2021, "Frame Replication and Elimination for Reliability Amendment 2: Extend Stream Identification Functions," 2021. [Online]. Available: https://ieeexplore.ieee.org/document/9740589.

[11] IEEE Std 802.1CBcv-2021, "Frame Replication and Elimination for Reliability Amendment 1: Information Model, YANG Data Model, and Management Information Base Module," 2021. [Online]. Available: https://ieeexplore.ieee.org/document/9715061.

[12] IEEE Std 802.1CB-2017, "Frame Replication and Elimination for Reliability," 2017. [Online]. Available: https://ieeexplore.ieee.org/document/8091139.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

5.4 Optimized Headers and Packet Rate Improvement

The standard layering of Ethernet and IPv4/v6 headers emphasizes flexibility for heterogenous inter-networking environments. Protocol content is separated across multiple functional layers, and features such as packet fragmentation enable interconnectivity between heterogenous networks. This flexibility comes at the cost of larger packet overheads.

Many HPC and AI applications generate a significant volume of small messages where the payload is small relative to packet headers. Examples include applications accessing partitioned memory over single-hop or multi-hop fabrics. For efficient bandwidth utilization and high goodput, it is desirable that the interconnect overhead also be small.

Special-purpose network fabrics optimize for this requirement by making assumptions about a homogenous environment, reducing flexibility, simplifying the information carried with a packet, and improving encoding over a link by reducing inter-packet gaps. Such techniques can also be applied to an Ethernet-based high-performance interconnect to improve bandwidth efficiency and packet rates.

The current Ethernet and IPv4/v6 header stacks are deeply entrenched, with a rich and diverse ecosystem of tools and silicon built around them. Completely redefining the headers is not practical. Instead, an approach that superimposes information on existing headers is more likely to find acceptance, especially if it enables coexistence with standard Ethernet silicon, tools, and operational practices.

This specification defines an optimized header format called the Unified Forwarding Header (UFH) to supplement the common combination of Ethernet MAC and IP headers. The optimization involves:

- Removing redundancies across the MAC and IP headers.
- Eliminating header fields generally not required in a homogenous, well-controlled fabric.
- Retaining features that allow UFH and standard Ethernet + IP frames to coexist on a link.
- Retaining features that enable extensibility, a key advantage of Ethernet.

UFH is intended for applications in well-contained network domains and is not intended to be a generic replacement for the Ethernet + IP stack. The UFH format defines two variations to match the slightly divergent needs of scale-up and scale-out fabrics. See Section 5.4.4.

This specification does not define any new control protocols. Coexistence of UFH and standard Ethernet frames on a link enables the use of standard Ethernet and IP protocols such as LLDP, ICMP, or DHCP to bootstrap and manage endpoints in a network.

5.4.1 Requirements and Considerations

The UFH definition aligns with the following requirements and considerations:

- UFH frames MUST coexist with standard Ethernet frames on a link. This is essential to leveraging the benefits of the Ethernet ecosystem and in accelerating UFH adoption. This implies no changes below the Ethernet MAC layer.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

- UFH usage is optional, and an initiator SHOULD use it only for data flows that benefit from improved wire efficiency and throughput. Applications that do not benefit from an optimized header can continue to use standard Ethernet and IP formats. (e.g., LLDP, ARP, DHCP)
- Ethernet enables feature extensibility by stacking tags such as VLAN, MACSEC, and CSIG. The UFH definition MUST support header stacking.
- UFH MUST be compatible with LLR (link level retry) and CBFC (credit-based flow control) but not depend on them.
- UFH addressing MUST support scale-out fabrics with at least 256 k endpoints with a stretch goal of 1 M+ endpoints.
- UFH source and destination address fields should not assume a specific address family even if IP addressing is likely to be the de facto standard. Deployments should have the freedom to map different address namespaces to UFH address fields, such as IB LIDs or device IDs, to enable migration of different solutions to Ethernet using UFH.
- The UFH definition SHOULD support stateless translation to standard Ethernet + IP headers, when addresses are interpreted as IPv4 or IPv6. Network elements such as gateway switches MAY optionally enable such a capability.

The following are non-goals:

- Ethernet limits minimum frame size to 64 B. UFH does not modify this limit.
- Ethernet IPG and preamble reduction are outside the scope of the UFH specification.
- Extending existing control protocols such as DHCP or ARP to add UFH awareness is not in scope.
- It is expected that while many existing Ethernet switches will have the flexibility to forward UFH packets, compatibility with standard 802.1 Ethernet bridges is not a goal.

5.4.2 Header Optimization Approach

Large-scale Ethernet fabrics are commonly designed as IPv4/v6 routed topologies. The Ethernet MAC has link-local significance, and the DA and SA fields are swapped at each hop. This creates an addressing redundancy that can be removed by collapsing to one address space. A review of the IPv4 and IPv6 headers suggests other optimization possibilities. Fragmentation can be avoided in homogenous fabrics and length, and header checksum fields can be removed by relying on link capabilities such as Forward Error Correction (FEC), link level retry (LLR), or end-to-end transport checksums such as an ICRC.

summarizes optimizations for single- and multi-hop fabrics.

Table 5-38 - Header Optimizations

IPv4/IPv6 Field Name	IPv4 Size	IPv6 Size	UFH Needs (large fabrics)	UFH Needs (small fabrics)	Comments
Address Version	4	4	1	1	UFH does not assume an address namespace. Mapping to IPv4 or IPv6 is optional and a deployment choice. One bit is reserved to identify address container size.
IHL	4	-	-	-	-

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

IPv4/IPv6 Field Name	IPv4 Size	IPv6 Size	UFH Needs (large fabrics)	UFH Needs (small fabrics)	Comments
DSCP+ECN/Traffic Class	8	8	8	8	Retain same code point name space for IP and UFH.
Total Length	16	-	-	-	Common L4 transports (UDP, UET) have length field.
Identification	16	-	-	-	No support for fragmentation.
Flags	3	-	-	-	
Fragment Offset	13	-	-	-	
TTL/Hop-Limit	8	8	4	-	4 bits is sufficient for proposed network architectures.
Protocol/Next header	8	8	-	-	EtherType after UFH identifies next header.
Header Checksum	16	-	-	-	Rely on FEC + LLR and L4 checksum.
Flow Label	-	20	optional	-	Common L4 transports (UDP, UET) have entropy. Optional for a network header.
SIP	32	128	32	16	Support IPv4 or IPv4-mapped-IPv6 for scale-out. Scale-up deployments can use any suitable namespace.
DIP	32	128	32	16	
Total (bits)	**160**	**320**	**77 +**	**41**	

Any packets or flows requiring unsupported features (such as packets requiring IP fragmentation or tunnels), or that do not benefit from header optimizations, are not candidates for UFH and MUST continue to use standard Ethernet and IP headers.

5.4.3 UFH Definition

The UFH is defined as an overlay on the 12-byte Ethernet MAC header with special encodings in the MAC DA and MAC SA fields. Fields identified as essential to UFH are overlaid on this 12-byte base header, and UFH defines two sub-variants to cover different needs. The Ethernet rule of inserting an EtherType field at offset 12 from the start of frame is followed to maintain backward compatibility and to allow UFH and standard Ethernet + IP frames to coexist on a link. This EtherType identifies the next header in the stack and is not essential for UFH itself.

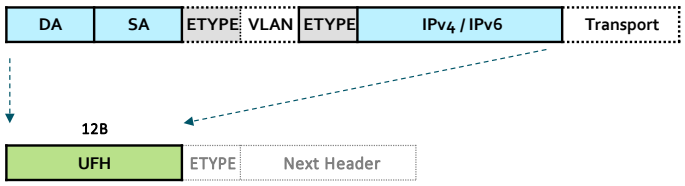


Figure 5-24 - UFH Header Overlay

The IEEE Std 802-2024 specification [2] defines a Structured Local Address Plan (SLAP) to support allocation of local addresses. This specification reserves 4 bits in the MAC address for identification,

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

identified as ZYXM in Figure 5-25. The X and M bits are the well-known local address and multicast bits in the MAC header; the ZY bits are new per SLAP. To identify UFH:

- The Z and Y bits are set to 00 indicating an Administratively Assigned Identifier (AAI)[18].
- The X bit is set to 1 indicating local assignment.
- The M bit is retained to indicate a multicast frame.

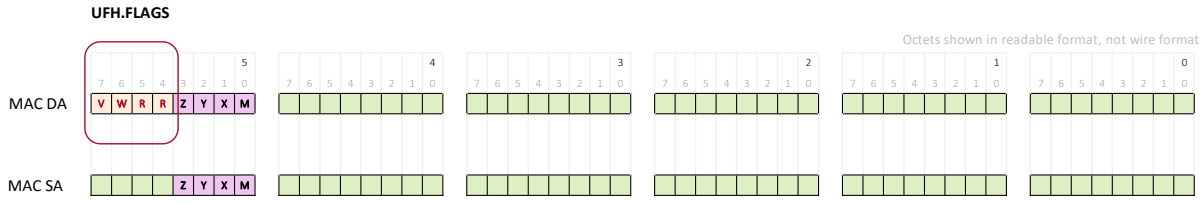


Figure 5-25 - UFH Header Identification

This UFH specification further defines the upper 4 bits of the DA for additional UFH specific flags.

- The V bit identifies UFH version, with V = 0 indicating the current version.
- The W bit is interpreted in the context of V = 0.
  - W = 0 -> Normal format with broad applicability (32-bit address container, hop-limit).
  - W = 1 -> Compressed format for efficiency (16-bit address container).
- The RR bits are reserved and currently set to 00.

Per this spec, a transmitter generating a UFH frame MUST set V to 0, and a receiver must verify the V and W bits to process the frame. V = 1 is available for a future version.

UFH uses the IEEE SLAP AAI encoding as it gives the administrator of a local domain the freedom to define the address structure and reclaim 44 bits from the DA and 44 bits from the SA. The intent behind the Extended Local Identifier (ELI) and Standard Assigned Identifier (SAI) encodings is to share local address space across companies or standards, and the bits used to encode this (CID for ELI, undefined for SAI) will result in fewer available bits and lower efficiency.

The IEEE Std 802-2024 specification does not define a mechanism for different applications to simultaneously use AAI-encoded addresses within a network. This need is not anticipated, but if required, it is possible to use the RR bits for disambiguation.

UFH should be viewed as a distinct header format with its own address space and forwarding semantics as described later in Section 5.4.6. Network devices identify it by matching the flags byte.

[18] Alternative encodings are possible if approved by the IEEE.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Table 5-39 - IEEE Std 802-2024 SLAP Encoding

SLAP quadrant	Y bit	Z bit	SLAP local identifier type	SLAP local identifier
01	0	1	Extended Local	ELI
11	1	1	Standard Assigned	SAI
00	0	0	Administratively Assigned	AAI
10	1	0	Reserved	Reserved

5.4.4 UFH Variants

UFH defines two variants to address the opposing needs of efficiency and broad capability: UFH_32 and UFH_16. As shown in Figure 5-26, UFH_32 reserves 32 bits for addresses, 8 bits for a traffic class, and 4 bits for a hop-limit, making it better suited for multi-hop topologies. UFH_16 reserves 16 bits for addresses and 8 bits for a traffic class, making it better suited for single-hop topologies.

The traffic class field is in the same location in both formats. This covers 6 bits for a DSCP codepoint and 2 bits for the ECN field aligned with IPv4/IPv6 usage and format. The same encodings and namespace SHOULD be retained to enable stateless translation between UFH and IP and to simplify processing in switches.

The UFH_32 format has 8 bits of opaque space while the UFH_16 format has 44 bits. These “opaque” bits can be overlaid with any end-to-end data, and switches MUST ignore them when forwarding packets.

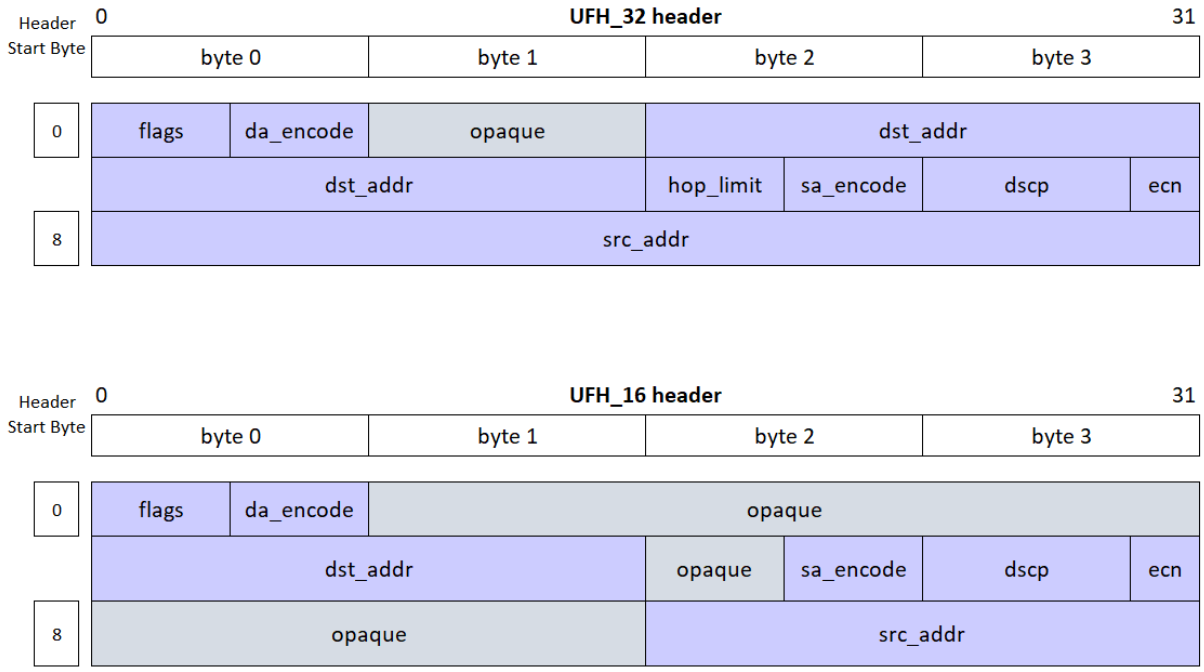


Figure 5-26 - UFH Formats

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Table 5-40 - UFH_32 Field Names

Field Name	Field Size (in bits)	Field Description
flags • v-bit • w-bit • rsvd • rsvd	4	**ufh.flags** = [v, w, rsvd, rsvd] • UFH version, with v=0 indicating the current version • Indicates UFH width for v=0. Setting w=0 for 32-bit format, w=1 for 16-bit format • Reserved • Reserved
da_encode • z-bit • y-bit • x-bit • m-bit	4	**ufh.da_encode** = [z, y, x, m] = Specified by IEEE Std 802 [2] • SLAP quadrant encoding Z bit • SLAP quadrant encoding Y bit • Universal/Local encoding X bit • Individual/Group encoding M bit
opaque	8	For use by higher-layer protocol
dst_addr	32	Destination address
hop_limit	4	Hop-limit processed by the forwarding function analogous to processing of an IP router. See Section 5.4.6.
sa_encode • z-bit • y-bit • x-bit • m-bit	4	**ufh.sa_encode** = [z, y, x, m] = Specified by IEEE Std 802 • SLAP quadrant encoding Z bit • SLAP quadrant encoding Y bit • Universal/Local encoding X bit • Individual/Group encoding M bit
dscp	6	Differentiated services code point aligned with IPv4/IPv6 usage. Together with the **ufh.ecn** field it forms the IPv4 differentiated services field or the IPv6 traffic class field.
ecn	2	Explicit congestion notification aligned with IPv6/IPv6 usage. Together with the **ufh.dscp** field it forms the IPv4 differentiated services field or the IPv6 traffic class field.
src_addr	32	Source address

Table 5-41 - UFH_16 Field Names

Field Name	Field Size (in bits)	Field Description
flags • v-bit • w-bit • rsvd • rsvd	4	**ufh.flags** = [v, w, rsvd, rsvd] • UFH version, with v=0 indicating the current version • Indicates UFH width for v=0. Setting w=0 for 32-bit format, w=1 for 16-bit format. • Reserved • Reserved
da_encode • z-bit • y-bit • x-bit • m-bit	4	**ufh.da_encode** = [z, y, x, m] = Specified by IEEE Std 802 [2] • SLAP quadrant encoding Z bit • SLAP quadrant encoding Y bit • Universal/Local encoding X bit • Individual/Group encoding M bit
opaque	24	For use by higher-layer protocol
dst_addr	16	Destination address

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Field Name	Field Size (in bits)	Field Description
opaque	4	For use by higher-layer protocol
sa_encode • z-bit • y-bit • x-bit • m-bit	4	**ufh.sa_encode** = [z, y, x, m] = Specified by IEEE Std 802 • SLAP quadrant encoding Z bit • SLAP quadrant encoding Y bit • Universal/Local encoding X bit • Individual/Group encoding M bit
dscp	6	Differentiated services code point aligned with IPv4/IPv6 usage. Together with the **ufh.ecn** field it forms the IPv4 differentiated services field or the IPv6 traffic class field.
ecn	2	Explicit congestion notification aligned with IPv6/IPv6 usage. Together with the **ufh.dscp** field it forms the IPv4 differentiated services field or the IPv6 traffic class field.
opaque	16	For use by higher-layer protocol
src_addr	16	Source address

5.4.5 UFH and Header Stacking

UFH retains the use of the EtherType field to identify the next header in the header stack and extend the information carried with a packet. For example, VLAN tags can be used to partition a network domain for multi-tenancy, MACSEC tags can add link security, and CSIG tags can carry telemetry for congestion control or diagnostics. Figure 5-27 shows different tag stacking examples.

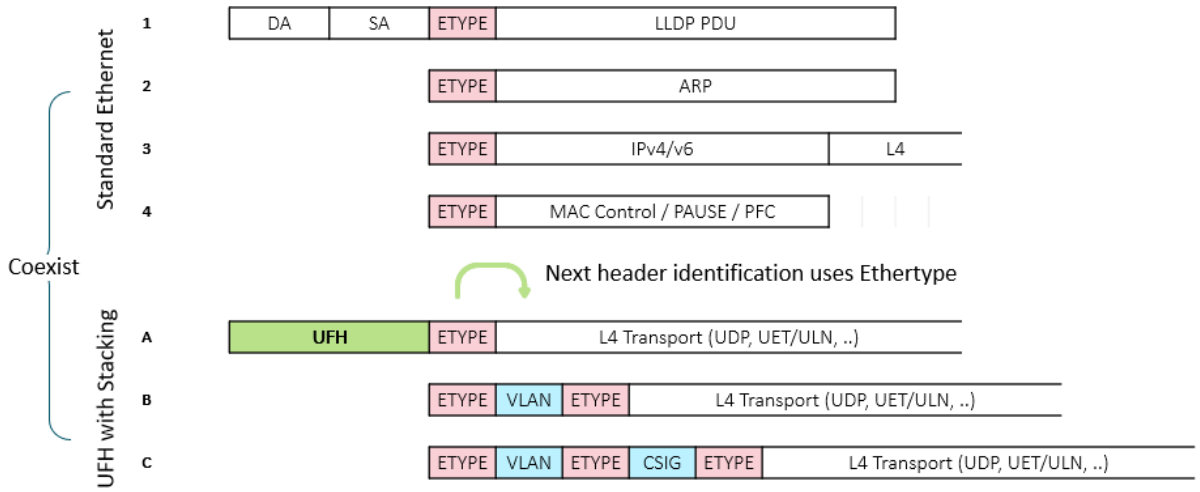


Figure 5-27 - UFH Coexistence and Extensibility

The ordering of tags in the header stack MUST follow Ethernet ordering rules. For example, a VLAN tag if present MUST be the first tag on a frame at offset 12.

Because UFH subsumes the traditional network layer, there is no additional IP header, and tags are followed by an L4 transport header. UFH is designed to be used with UDP, UET, and future transports. New EtherType encodings are required to identify them.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

5.4.6 UFH Termination and Forwarding

UFH packets appear as legal L2 frames on the wire but cannot be forwarded by standard Ethernet bridges as defined in IEEE 802.1. A UFH-capable switch needs to mask out non-address portions of the DA and SA when implementing forwarding behaviors and support additional capabilities defined by the UFH header. A conceptual view of a UFH endpoint and switch is shown in Figure 5-28.

The MAC layer at an endpoint identifies received UFH packets and forwards them to an upper layer through a "MAC Service Access Point." The upper layer validates and terminates UFH packets addressed to the endpoint and passes them to the transport protocol identified in the header stack.

Packets addressed to other endpoints are directed to a UFH switch that implements UFH-specific forwarding functions. The forwarding function looks up the UFH destination address to identify an egress interface, maps the encoded COS to a queue, and applies ECN and hop-limit updates analogous to those applied by an IP router. The forwarding function must ignore any opaque fields and reserved flags in the header.

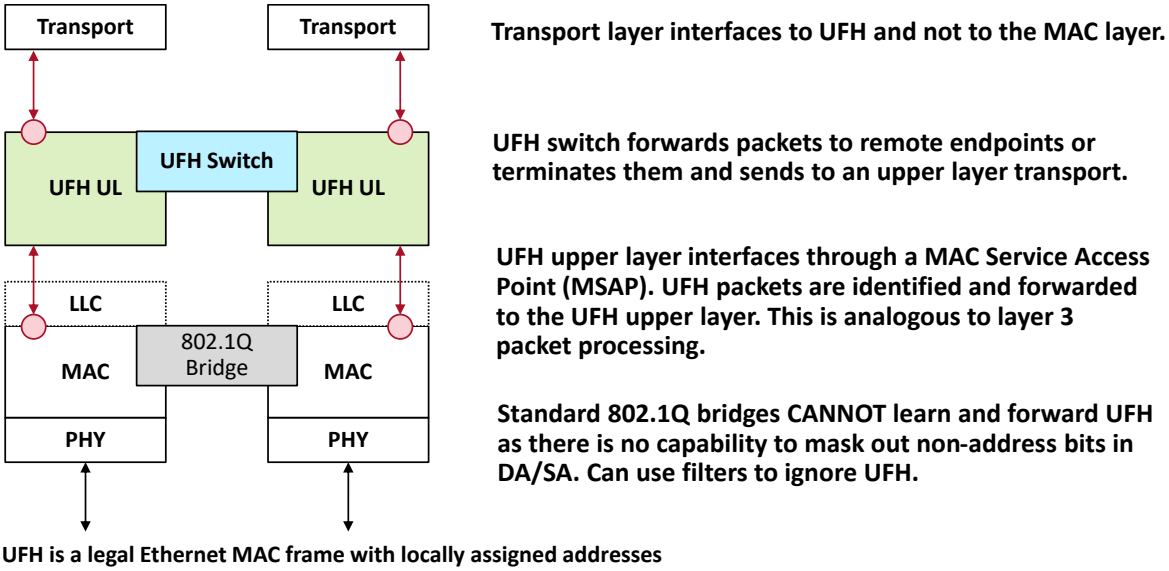


Figure 5-28 - Conceptual Layering of UFH

UFH defines 32-bit and 16-bit address container variants to support different deployment needs. These header variants are known as UFH_32 and UFH_16 respectively. While use of IP addresses is preferred with the 32-bit address variant, this specification intentionally does not mandate the use of a specific address family with either variant. The address fields are viewed as "containers" to which an implementation can map a suitable address namespace and conform to its forwarding rules. This flexibility is to help bespoke HPC and scale-up fabrics migrate to Ethernet.

The address namespace used by an implementation might need fewer than 32 or 16 address bits, and the addresses must be LSB-aligned within an address container. UFH endpoints and switches MUST be configured to ignore unused upper bits of a container when using the address for any function. For

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

example, a scale-up fabric mapping a 10-bit device ID to a 16-bit address container must place it in the lower 10 bits, and endpoints and switches must mask out the upper 6 bits when using the address container.

The UFH forwarding function behaves analogously to an IP router for implementations mapping a 32-bit IPv4 or 128-bit IPv6 address to the address containers. The UFH address containers carry the lower 32 bits or 16 bits of the address, with the upper bits configured consistently with a subnet prefix at every endpoint and switch within a UFH domain (see 5.4.11).

The UFH format retains the ability to mark a frame as multicast via the M bit, and its usage should be in accordance with the needs of the mapped address space. If the mapped address space is IP and the destination is a multicast or broadcast address, the M bit SHOULD be set to 1. A UFH switch can interpret the destination is multicast using either the M bit or the IP address type.

Hop-limit processing for UFH follows the same rules as IPv6 as defined in RFC 8200. Senders initialize the hop-limit to a non-zero value in the range 1–15. Transit switches verify the incoming hop-limit, forwarding packets if it is greater than 1 and dropping them if it is 0 or 1 and the switch is not the destination for the packet. Destinations (endpoints or switches) accept all hop-limits including 0. ICMP generation on hop-limit expiry is possible if a deployment maps an IP address namespace to UFH.

The ECN field is interpreted and marked per the rules for IP packets as described in RFC 3168.

This UFH specification does not define a UFH learning function or a spanning tree analog. The source address within a UFH packet plays no direct part in forwarding; it can be optionally used by an UFH switch for load balancing and by upper layers to track connection partners.

5.4.7 UFH and Queue Assignment

In a routed IP network, the DSCP field serves to assign the traffic class and queue assignments. The DSCP field in UFH is expected to serve the same purpose.

When UFH packets are used with VLANs, there is an option to use the VLAN PRI field for traffic class and queue assignment. In such cases, network configuration will determine the preference given for either. This is no different than usage of IP over Ethernet VLANs.

5.4.8 UFH and Packet Trimming

UFH packets dropped due to congestion may be “trimmed” and sent as drop notifications to a receiver. The UFH DSCP code point MUST be updated in accordance with the UE Packet Trimming specification (see Section 4.1).

5.4.9 UFH and VLANs

VLANs are commonly used to isolate endpoints in an Ethernet network, and they can serve the same function when used in conjunction with UFH. In a multi-tenant UFH environment, tenant identification can be encoded in VLANs and isolation enforced in the switches through either VLAN filtering or VRF assignment.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

5.4.10 UFH and Transport Layer Checksums

Transport protocols such as TCP/IP and UDP/IP include a checksum field whose calculation covers the payload, the transport header, and select fields from the IP header represented by a pseudo-IP header. The pseudo-IP header includes the source and destination IP addresses, the IP protocol used to identify the transport, and the transport length (header + payload).

This method of checksum calculation and verification remains unchanged when such transports are carried over UFH. The initiator and receiver assume the packet is carried over IP and create a suitable pseudo-IP header based on the source and destination UFH address containers, the IP protocol number assigned to the transport, and the length from the transport header.

All endpoints in a UFH domain (see next section) must be configured to consistently use the IPv4 or IPv6 pseudo-header. The UFH address containers are masked and then expanded to the necessary length by adding a suitable address prefix. The prefix can be one that is assigned to the UFH domain and can be set to all zeros if there is no intent to use the full address for any function other than checksum calculation.

Example

- Suppose a UFH-_32 format contains a non-IP 24-bit address “00.aa.bb.cc”.
- The desire is to map it to an IPv6 pseudo-header for checksum calculations.
- Suppose the UFH domain is assigned the IPv6 prefix “XX:YY:ZZ”.
- Then the generated address for checksum calculation is “XX:YY:ZZ::0:aa:bb:cc”.

This approach simplifies protocol migration to UFH and will additionally support stateless translation to Ethernet + IP at gateway switches if required.

A sender MAY set the transport header checksum to zero for the special case of UDP over IPv4 and it MUST be ignored by a receiver.

5.4.11 UFH Deployment Considerations

For simplicity of deployment, it is assumed that all endpoints and switches within a UFH domain can support UFH and can communicate with each other using UFH on ALL network paths. UFH usage within the domain MUST be explicitly enabled using a suitable administrative approach.

If a UFH domain is a standalone cluster with no external connectivity, a sender’s decision to use UFH will be based primarily on the needs of an application. In more complex deployments, a UFH domain may be a local island within a larger network. Devices and paths within the island are UFH-capable, but destinations outside the domain are either not capable or not reachable via a UFH-capable path. A sender’s decision to use UFH will depend on both the application and on the relative locations of the sender and destination.

In theory, it is possible to carry UFH packets from one UFH domain to another by translating them to Ethernet + IP at gateway switches or by encapsulating them in tunnels that support L2 payloads, such as

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

VxLAN and SRv6. These applications are possible as a side effect of retaining the Ethernet MAC structure, and the capability may be useful in special scenarios.

This specification does not describe any methods to ascertain if a sender can use UFH to communicate with a destination. Methods for assigning UFH addresses to endpoints or distributing routes within a domain are also outside the scope of this specification.

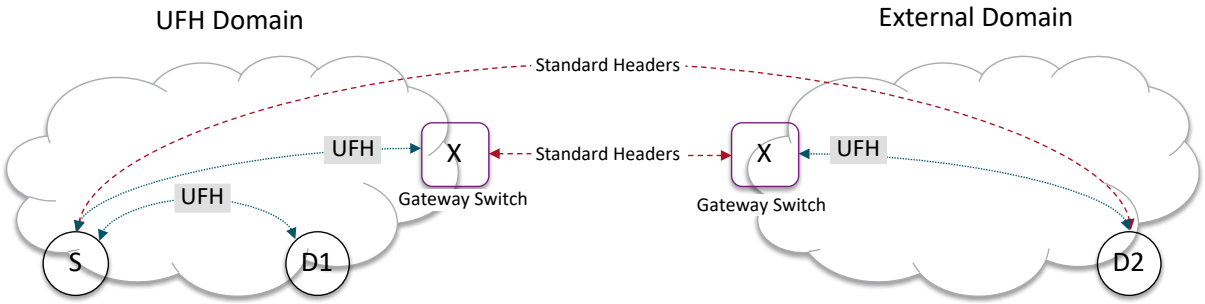


Figure 5-29 - UFH Domains

5.4.12 UFH Efficiency Calculations

Packet efficiency for different header permutations is calculated using the equation below:

$$E = \frac{Payload\ Size}{Payload\ Size + Header\ Overhead}$$

The header overhead adds all L2, L3, and L4 headers, with IPG and preamble ignored for simplicity. Opaque fields are not included in the overhead. A common transport size is assumed for all, and the resulting efficiency numbers in tabular and chart form are as below. For small payloads and compressed headers, the minimum Ethernet frame size is 64 B.

Table 5-42 - Header Overhead

Format	L2	L3	L4	FCS	Total	Notes
Custom (8 B)	8		16	4	28	Address scale 2^{16}
Ethernet + IPv4	14	20	16	4	54	Address scale 2^{32}
UFH_32	13		16	4	33	Address scale 2^{32}
UFH_16	8.5		16	4	28.5	Address scale 2^{16}

Table 5-43 - Efficiency Comparison

Payload Size	32	64	128	256	512	1024	2048	4096
Custom (8 B)	53.33	69.57	82.05	90.14	94.81	97.34	98.65	99.32
Ethernet + IPv4	37.21	54.24	70.33	82.58	90.46	94.99	97.43	98.70
UFH_32	49.23	65.98	79.50	88.58	93.94	96.88	98.41	99.20
UFH_16	50.00	69.19	81.79	89.98	94.73	97.29	98.63	99.31

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

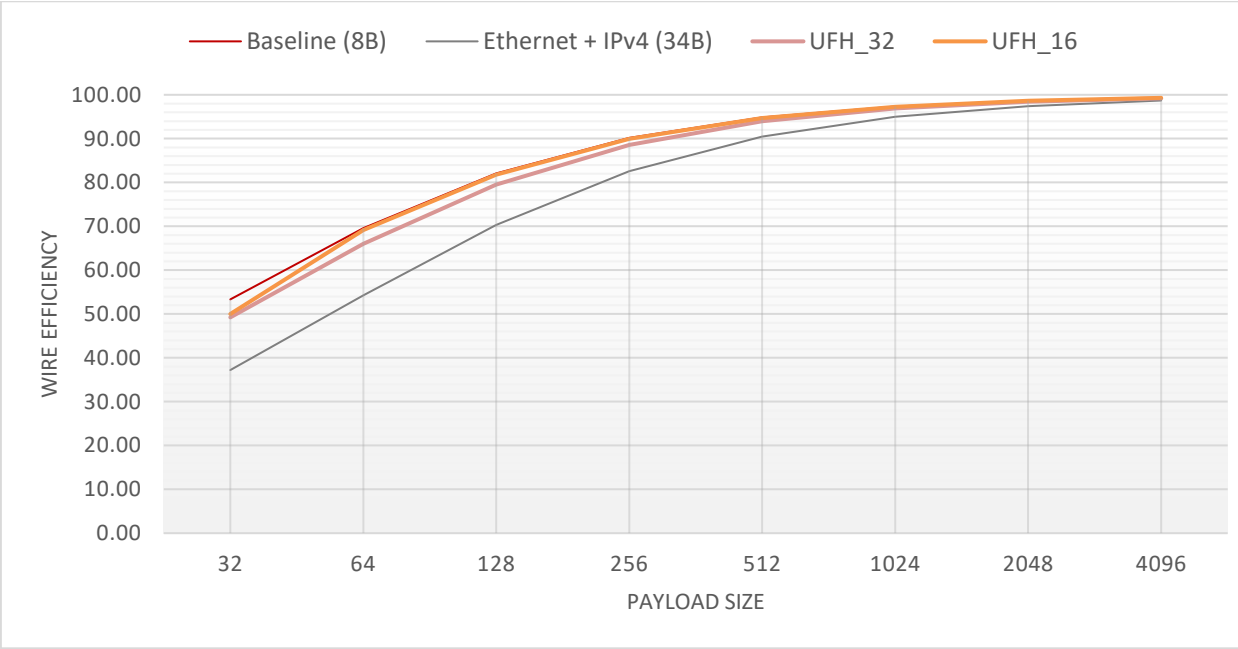


Figure 5-30 - Efficiency Comparison

5.4.13 References

[1] IETF RFC 8948, "Structured Local Address Plan (SLAP) Quadrant Selection Option for DHCPv6," 2020. [Online]. Available: https://www.rfc-editor.org/rfc/rfc8948.

[2] IEEE Std 802-2024, "IEEE Standard for Local and Metropolitan Area Networks: Overview and Architecture," 2024. [Online]. Available: https://ieeexplore.ieee.org/document/10935844.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

5.5 Idle Reduction

This standard specifies an optional feature that reduces the number of idle bytes inserted between frames. The goal of this feature is to improve payload efficiency for small frames and increase the packet rate.

A standard UDP Ethernet frame contains 18 bytes for the Ethernet header and CRC, 20 bytes for the IPv4 header, and 8 bytes of UDP header. With an 8-byte preamble and 12 bytes of inter-packet gap (IPG), there are already 66 bytes of overhead before accounting for the transport header(s). Each of these headers can potentially be reduced in size, increasing transmission efficiency for small payloads.

The idle reduction feature focuses on reducing the average of 12 idle bytes that are required by the standard (as introduced in IEEE Std 802.3-2022 Clause 46.3.1.4). At the latest link speeds specified in IEEE Std 802.3, frames start on an 8-byte boundary, so the number of idles bytes cannot always be exactly 12. Instead, a deficit idle count tracks idle bytes across a stream of frames and ensures the short-term average is at least 12 bytes. Ignoring upper-level protocols, the minimal Ethernet packet of 64 bytes requires 8 bytes of preamble and at least one byte for inter-packet gap (for the terminate block). With the restriction of starting frames on an 8-byte boundary, the minimum number of bytes on the wire is 80. The 12-byte average IPG requirement will cause some minimum frames to be further padded to 88 bytes on the wire, so that the average comes to at least 12 bytes of idle between packets.

To improve this design, this specification removes the 12-byte requirement while retaining the requirement to start frames on an 8-byte boundary. Maintaining this boundary reduces the impact of supporting idle reduction. For minimum-sized frames, this change reduces the wire usage to 80 bytes and eliminates the occasional need to pad to 88 bytes to achieve the target idle count. Furthermore, the frame size can now increase by up to 7 bytes (within the terminate block) while still maintaining the 80-byte wire usage.

5.5.1 Approach

This specification defines a reduction of the per-frame overhead, to increase the efficiency of small size frames. In IEEE Std 802.3 high-speed links, 12 bytes of idle are required. This is an average requirement, since idles on these links are added in blocks of eight bytes. While this is less significant for large frames, it is significant for minimum-sized frames. This optional feature allows most of these idle bytes to be omitted.

When this feature is supported and enabled, an average of 12 bytes of idle is no longer required. A terminate block (with up to 7 data bytes) MUST be included and MUST be sized to ensure 8-byte alignment. With modern network architectures, incoming link data is typically retimed to a core clock that runs faster than the fastest legal line rate. With this structure, issues transitioning from a fast clock to a slow clock do not immediately cause clock-domain-crossing FIFO problems. While the standard inter-frame gap is less significant for large packets, it wastes bandwidth when there is a large fraction of short packets.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

5.5.2 Rate Control

Per the standard, link speeds may deviate from nominal by +/- 50 ppm. When a fast sender sends traffic to a slow receiver, overrun has to be considered. The standard IPG of 12 bytes allows the receiver some slack to retime the incoming data to its slower clock by eliding some idle bytes. Rather than imposing the idle overhead per packet, this feature requires internal flow control in the link logic to provide any necessary compensation.

Modern high-performance Ethernet devices often have a core clock that runs faster than even the fastest legal incoming data rate. Such a clock allows incoming data to transit through a clock domain crossing FIFO without needing to drop any bytes. In this environment, the egress path requires flow control; otherwise, the higher-speed core would overrun the output link.

Besides clock rate issues, forming the proper Ethernet stream on the wire requires inserting control data, such as alignment markers. These markers are infrequent and are properly inserted in the data stream without data loss.

5.5.3 Maximum Packet Rate

With standard idle formatting, the minimum spacing from one frame start to the next frame start averages at least 84 bytes. Because of the 8-byte boundary requirement, the actual spacing might toggle between 80 bytes and 88 bytes. With a 100 GB/s Ethernet link, the frame rate could average as fast as 1.19 G frames/sec. With idle reduction, the 80-byte spacing may be constantly achieved, potentially saving an average of 4 bytes per packet. This raises the peak frame rate possible to 1.25 G frames/sec, realizing the maximum possible benefit from idle reduction.

5.5.4 Requirements

Devices supporting the idle reduction feature SHOULD report this capability to the management software. To ensure seamless interoperability, any link advertising support MUST be capable of receiving and processing frames with the idle reduction defined by this feature, regardless of whether the feature is currently enabled. The enabled capability specifically controls the transmission logic, determining whether the MAC will actively insert reduced-width idle counts during outgoing data streams.

5.5.5 Enabling Idle Reduction

The idle reduction enhancement is optional. Cost-optimized devices might not easily support idle reduction due to their internal architecture. Further, idle reduction on some but not all links in a heterogeneous network could easily be a source of congestion. Consequently, management software may elect not to enable this feature on links that support it, in order to prevent congestion in a network that also contain links without this capability.

To accommodate these limitations, each link SHOULD advertise whether it supports this feature. Using UE link negotiation, capable devices SHOULD advertise IPGR-C (inter-packet gap reduction “capable”). Management software MAY determine, after interrogating each end of all links, to enable this feature. Once enabled, links will no longer be required to ensure that the standard number of idle bytes are transmitted.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

6 UE Physical Layer

Ultra Ethernet (UE) applications, such as AI and HPC systems, require physical layers that support a variety of electrical and optical channels; provide low frame loss ratio, low latency, power efficiency, and reliability at scale; and leverage existing ecosystems and standards that enable seamless interoperability.

Informative Text:
The UE Physical Layer specification strives to maintain consistency with terminology defined by IEEE Std 802.3. However, the 802.3 definition of the term "packet" is inconsistent with other portions of the UE specification. When used in this Physical Layer specification, the 802.3 term "packet" is replaced with "802.3 packet."

6.1 UE PHY Support

UE PHYs include a subset of Ethernet PHYs with 100 Gb/s or 200 Gb/s per lane signaling, defined in IEEE Std 802.3™-2022 [1], IEEE Std 802.3db™-2022 [2], IEEE Std 802.3ck™-2022 [3], IEEE Std 802.3df™-2024 [4], and P802.3dj [5] (henceforth referred to collectively as the Ethernet standard).

A device compliant with the UE PHY specifications SHALL conform to one of the specifications of PHY types listed in either 6.1.1.2or 6.1.2.2.

Informative Text:
Other Ethernet PHYs may be able to support some of the UE specifications but are not considered compliant with the UE specification.

6.1.1 UE PHYs for 100 Gb/s per Lane Signaling

UE PHYs include a subset of Ethernet PHYs with 100 Gb/s per lane signaling, as defined in this section.

6.1.1.1 Media support

The following physical media are supported for 100 Gb/s per lane signaling, with relevant clause numbers in the Ethernet standard:

- Backplane (KR, clause 163)
- Copper cable (CR, clause 162)
- MMF up to 50 m (VR, clause 167)
- MMF up to 100 m (SR, clause 167)
- Parallel SMF up to 500 m (DR, clauses 124, 140)
- WDM SMF up to 2 km (FR, clauses 140, 151)

The following PHY families are considered beyond the scope of UE:

- Parallel SMF up to 2 km (DR-2, clause 124)
- WDM SMF up to 10 km (LR, clauses 124, 140, 151)
- WDM SMF, coherent (ZR, clause 154)

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

6.1.1.2 PHY rates and types supported

The following Ethernet PHY rates and types are in scope for UE:

- 100GBASE-R family
  - 100GBASE-KR1, 100GBASE-CR1, 100GBASE-VR1, 100GBASE-SR1, 100GBASE-DR, 100GBASE-FR1
- 200GBASE-R family
  - 200GBASE-KR2, 200GBASE-CR2, 200GBASE-VR2, 200GBASE-SR2
- 400GBASE-R family
  - 400GBASE-KR4, 400GBASE-CR4, 400GBASE-VR4, 400GBASE-SR4, 400GBASE-DR4, 400GBASE-FR4
- 800GBASE-R family
  - 800GBASE-KR8, 800GBASE-CR8, 800GBASE-VR8, 800GBASE-SR8, 800GBASE-DR8

The clauses within the Ethernet standard that specify these PHY types are listed by media type in Section 6.1.1.1.

In addition, the following 100 Gb/s per lane interfaces are in scope:

- Chip-to-chip attachment unit interfaces (Annex 120F)
  - 100GAUI-1 C2C, 200GAUI-2 C2C, 400GAUI-4 C2C, 800GAUI-8 C2C, 1.6TAUI-16 C2C
- Chip-to-module attachment unit interfaces (Annex 120G)
  - 100GAUI-1 C2M, 200GAUI-2 C2M, 400GAUI-4 C2M, 800GAUI-8 C2M, 1.6TAUI-16 C2M

6.1.2 UE PHYS for 200 Gb/s per Lane Signaling

UE PHYs include a subset of Ethernet PHYs with 200 Gb/s per lane signaling, as defined in this section.

6.1.2.1 Media Support

The following physical media are supported for 200 Gb/s per lane signaling, with relevant clause numbers in the Ethernet standard:

- Backplane (KR, clause 178)
- Copper cable (CR, clause 179)
- Parallel SMF up to 500 m (DR, clause 180)
- WDM SMF up to 500 m (FR4-500, clause 181)

Note that the PHYs for the media above all use the same RS-FEC (544,514) with interleaving of 10-bit symbols from four RS-FEC codewords. These PHYs do not use inner FEC or a separate FEC sublayer.

6.1.2.2 PHY Rates and Types Supported

The following Ethernet PHY rates and types are in scope for UE:

- 200GBASE-R family
  - 200GBASE-KR1, 200GBASE-CR1, 200GBASE-DR1

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

- 400GBASE-R family
    - 400GBASE-KR2, 400GBASE-CR2
- 800GBASE-R family
    - 800GBASE-KR4, 800GBASE-CR4, 800GBASE-DR4, 800GBASE-FR4-500
- 1.6TBASE-R family
    - 1.6TBASE-KR8, 1.6TBASE-CR8, 1.6TBASE-DR8

In addition, the following 200 Gb/s per lane interfaces are in scope:

- Chip-to-chip Attachment Unit Interfaces (Annex 176C)
    - 200GAUI-1 C2C, 400GAUI-2 C2C, 800GAUI-4 C2C, 1.6TAUI-8 C2C
- Chip-to-module Attachment Unit Interfaces (Annex 176D)
    - 200GAUI-1 C2M, 400GAUI-2 C2M, 800GAUI-4 C2M, 1.6TAUI-8 C2M

6.2 Control ordered sets

UE control ordered sets are an OPTIONAL UE feature. A PHY that supports this feature SHALL comply with the requirements in this section.

The control ordered set (CtlOS) is a message mechanism utilized by the UE link layer features credit-based flow control (CBFC) and link layer retry (LLR). The general format is extensible to future link-level features as well. It is an 8-byte message encoded as an ordered set in the 64B/66B PCS encoding with an O code value to distinguish it from standard Ethernet sequence ordered sets.

6.2.1 Sequence ordered sets and control ordered set background

Sequence ordered sets, as defined in IEEE Std 802.3-2022 [1] clause 81, are encoded 8-byte messages that interrupt 802.3 packet transmission with critical link fault status. Three sequence ordered sets are defined: Local Fault, Remote Fault, and Link Interruption. Each sequence ordered set has a defined MII format and a generic 64B/66B PCS translation, as summarized in Table 6-2. See [1], subclause 81.3.2.2, Table 81-4 “Permissible lane encodings of RXD and RXC” and subclause 81.3.4, Table 81-5 “Sequence ordered sets” as well as subclause 82.2.3.3, Figure 82-5 “64B/66B block format”.

Table 6-1 - Ethernet standard sequence ordered sets on MII

Ordered set	Lane 0 (D0, control character)	Lane 1 (D1)	Lane 2 (D2)	Lane 3 (D3)	Lane 4 (D4)	Lane 5 (D5)	Lane 6 (D6)	Lane 7 (D7)
Local Fault	0x9C	0	0	1	0	0	0	0
Remote Fault	0x9C	0	0	2	0	0	0	0
Link Interruption	0x9C	0	0	3	0	0	0	0
Note: • All numbers in this table represent 8-bit values (data octets).								

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

The PCS 64B/66B encoding translates the MII Lane 0 sequence ordered set control code, 0x9C, to a block type of 0x4B plus the 4-bit O code value of 0x0, as shown in Table 6-2.

Table 6-2 - Ethernet standard 64B/66B sequence ordered set encoding

Field name	Sync	Block type	D1	D2	D3	O code	D4[7:4]	D5	D6	D7
Bit positions	**0:1**	**2:9**	**10:17**	**18:25**	**26:33**	**34:37**	**38:41**	**42:49**	**50:57**	**58:65**
Local Fault	‘b10	0x4B	0	0	1	0	0	0	0	0
Remote Fault	‘b10	0x4B	0	0	2	0	0	0	0	0
Link Interrupt	‘b10	0x4B	0	0	3	0	0	0	0	0
Note: • O code in bits [34:37] = 0x0 for the defined fault codes.										

6.2.2 Control ordered sets format

On the MII bus between the RS and PCS, control ordered sets have a Lane 0 (TXD<7:0>) control character with value 0x5C. This distinguishes them from Sequence ordered sets.

Table 6-3 - Ordered set format on MII

Ordered set	Lane 0 (D0, control character)	Lane 1 (D1)	Lane 2 (D2)	Lane 3 (D3)	Lane 4 (D4)	Lane 5 (D5)	Lane 6 (D6)	Lane 7 (D7)
Sequence	0x9C	0	0	D3	0	0	0	0
Control	0x5C	Type	D2	D3	{D4[7:4], 0x6}	D5	D6	D7

For a CtlOS, D4[3:0] SHALL be set to 0x6.

CtlOS uses the same Lane 0 control character as the “signal ordered set" defined in [1], subclause 49.2.4.6, Table 49-1; however, the value 0x6 is different from the one used by the signal ordered set.

The CtlOS Type field is used to identify each CtlOS message type. See the UE Link Layer specification Section 5.2.11.1 for all defined CtlOS type values and the specific data format of D2-D7 for each message type.

6.2.3 PCS required modifications

The PCS 64B/66B encoding is updated to translate an MII lane 0 control character of 0x5C to a block type of 0x4B, with bits 10 to 65 taken from D1 through D7 (where the 4 least significant bits of D4 are 0x6), as shown in Table 6-4.

Table 6-4 - Modified 64B/66B PCS encoding for ordered sets

Bit positions	0:1	2:9	10:17	18:25	26:33	34:37	38:41	42:49	50:57	58:65
Ordered set Content	‘b10	0x4B	D1	D2	D3	O code	D4[4:7]	D5	D6	D7

The O code value is 0x0 for sequence ordered sets and 0x6 for UE control ordered sets.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

The sync field (bits 0:1) and block type field (bits 2:9) are transmitted as 10_11010010, where the leftmost bit is transmitted first[19]. Starting with bits 10:17, each remaining byte is transmitted, starting with its least significant bit.

In the PCS decoder, MII D1 through D7 are taken from bits 10 to 65, while the control character (MII D0) is decoded from the O code in bits 34:37. For an O code value 0x0, MII D0 is set to 0x9C. For an O code value 0x6, MII D0 is set to 0x5C.

The Ethernet standard's state-based PCS encoder and decoder discard any non-data blocks that occur between the Start and Terminate blocks of an 802.3 packet. However, CtlOS are allowed to preempt regular 802.3 packet transmission and can occur at any point in the sequence of MII blocks.

In order to support 802.3 packet preemption, the following requirements apply to the PCS 64B/66B encoder and decoder:

6.2.3.1 PCS 64B/66B Decoder

For 200GBASE-R or 400GBASE-R PHYs using the PCS defined in Clause 119 of [1], 800GBASE-R PHYs using the PCS defined in clause 172 of [4], and 1.6TBASE-R PHYs using the PCS defined in clause 175 of [5], the PCS SHALL use the stateless 64B/66B decoder specified in subclause 119.2.5.8.2 of [5], with the corresponding requirements from the Reed-Solomon decoder in subclause 119.2.5.3 of [5].

For 100GBASE-R PHYs, which use the PCS defined in Clause 82 of [1], the following requirements apply.

The sublayer below the 100GBASE-R PCS is either RS-FEC (as defined in Clause 91 of [1]) or RS-FEC-Int (as defined in Clause 161 of [3]). The term "FEC" is used to refer to either of these sublayers. The specified behavior of the FEC includes marking of received codewords that are uncorrectable by altering the sync headers of certain 66-bit blocks within the codewords (as specified in 91.5.3.3 of [1] and in 161.5.3.3 of [3]). This is different from the specification of the Reed-Solomon decoder within the PCS in 119.2.5.3 of [1], where the whole content of a received uncorrectable codeword is replaced with error blocks (EBLOCK_R).

In a PCS implementation that is integrated with the FEC, the combined PCS+FEC can perform error marking by setting every 66-bit block within an uncorrected codeword (when Clause 91 RS-FEC is used) or two associated uncorrected codewords (when Clause 161 RS-FEC-Int is used) to an error block, as done in the Clause 119 PCS. This behavior is referred to within this specification as "FEC error overwrite marking."

Informative Text:
The FEC error overwrite marking behavior may seem to deviate from the RS-FEC and RS-FEC-Int specifications. However, if the interface between FEC and the PCS is not observable, then this behavior is indistinguishable from the specified one. This is considered a valid implementation choice.

[19] Per the convention in IEEE Std 802.3, the LSB of the hexadecimal value 0x4B in bits 2:9 is transmitted first.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

If a 100GBASE-R PHY performs FEC error overwrite marking, the decoder SHALL use the stateless 64B/66B decoder specified in 119.2.5.8.2 of [5] instead of the state-diagram decoder defined by Figure 82–17 of [1]. In addition, the decoder SHALL set to EBLOCK_R the first four 66-bit blocks (accounting for any alignment marker removal) following an uncorrected codeword (when Clause 91 RS-FEC is used) or two associated uncorrected codewords (when Clause 161 RS-FEC-Int is used).

If a 100GBASE-R PHY does not perform FEC error overwrite marking, the decoder SHALL use the rules in Table 6-5 instead of the state-diagram decoder defined by Figure 82–17 of [1].

Table 6-5 - PCS stateless decoder rules for 100GBASE-R without FEC error overwrite marking

Reset + !align_status	R_TYPE (rx_coded_{i-7} through rx_coded_{i-1})	R_TYPE (rx_coded_i)	Resulting rx_raw_i
1	any block type	any block type	LBLOCK_R
0	any block type	E	EBLOCK_R
0	E	any block type	EBLOCK_R
0	any combination not listed above		DECODE(rx_coded_i)

6.2.3.2 PCS 64B/66B Encoder

The following requirements apply in all supported data rates.The PCS encoder SHALL use the stateless encoder as specified in subclause 119.2.4.1.2 of [5], rather than the state-diagram PCS encoder (defined by Figure 82–16 or Figure 119–14 of [1]).

Informative Text:
The stateless encoder assumes that the block type sequence that the MAC generates is valid, and thus the PCS encodes the blocks as they arrive with no further checking.

6.2.4 RS required modifications

Support of CtlOS requires several enhancements to the reconciliation sublayer (RS) specified in the Ethernet standard.

When the MAC sublayer requests transmission of a CtlOS, the RS indicates a CtlOS on the MII, corresponding to the RS transmission of TXD<63:0> and TXC<7:0>. Otherwise (when not a CtlOS), the RS requests transmission of 64 data bits by the PHY as described in [1], subclause 81.1.7.1.4.

When the RS detects a valid CtlOS on the MII, corresponding to the RS reception of RXD<63:0> and RXC<7:0>, the RS sends an indication to the MAC sublayer. Otherwise (when not a CtlOS), the RS maps the MII signals to PLS_DATA.indication as described in [1], subclause 81.1.7.2.

6.2.4.1 MII transmit functional specifications

The transmit RS is permitted to send a CtlOS (composed of the CtlOS control character and subsequent seven data octets) anywhere in the MII data stream, subject to the restriction that the CtlOS control character is aligned to lane 0 on the transmit MII. See [1], subclause 81.2 for details on the MII data stream.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Insertion of a CtlOS anywhere in the MII data stream SHALL NOT corrupt any 802.3 packet transfers, regardless of whether the CtlOS interrupts 802.3 packet transmission or falls outside 802.3 packet boundaries.

The presence of a CtlOS SHALL NOT impact the Deficit Idle Count (DIC) of the transmit RS. See [1], subclause 81.3.1.4 for details on DIC.

6.2.4.2 MII receive functional specifications

The receive RS SHALL consider a CtlOS (composed of the CtlOS control character and subsequent seven data octets) arriving in lane 0 on the receive MII as a permissible lane encoding (see [1], subclause 81.3.2.2, Table 81-4).

A valid receive CtlOS SHALL have no impact on the RS DATA_VALID_STATUS parameter.

For setting DATA_VALID_STATUS to DATA_VALID (see [1], subclause 81.1.7.5.3), the RS SHALL skip over (not consider) a CtlOS immediately preceding the reception of a Start control character on lane 0. The RXC<7:0> and RXD<63:0> preceding the CtlOS are used to determine whether DATA_VALID_STATUS is set to DATA_VALID upon reception of a Start control character on lane 0.

Table 6-6 - RS Layer CtlOS spacing constraints

Parameter	Value (bits)
RS_CTLOS_MIN_RX_SPACING	2048

For setting DATA_VALID_STATUS to DATA_NOT_VALID (see [1], subclause 81.1.7.5.3), the RS SHALL skip over (not consider) a single occurrence of a CtlOS (not cause DATA_VALID_STATUS to assume the value of DATA_NOT_VALID). Occurrence of multiple CtlOS within a window of RS_CTLOS_MIN_RX_SPACING MII data bits (see Table 6-6) is considered an error, and the RS may set DATA_VALID_STATUS to DATA_NOT_VALID upon reception of a second (and any subsequent) CtlOS received within that window of bits. It is also permissible for the RS to ignore (not cause DATA_VALID_STATUS to assume the value of DATA_NOT_VALID) multiple CtlOS received within that window.

6.2.4.3 Link fault signaling

When the RS variable link_fault status indicates Local Fault or Remote Fault, the RS does not send CtlOS on the MII. During that time (as described in [1], subclause 81.3.4), the RS sends either Remote Fault (in response to link_fault status of Local Fault) or Idle (in response to link_fault status of Remote Fault). When the link_fault status is set to OK, the transmit RS may send CtlOS on the MII.

6.3 FEC statistics for prediction of link quality

FEC statistics for prediction of link quality is an OPTIONAL UE feature. A PHY that supports this feature SHALL comply with the requirements in this section.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Ethernet specifies the link quality in terms of frame loss ratio (FLR). FLR is defined at the MAC/PLS service interface and is measurable in the MAC. An example of FLR requirements can be found in subclause 167.1.1 of IEEE Std 802.3db-2022 [2] and its amendment in IEEE Std 802.3df-2024 [4].

For a 100 Gb/s Ethernet Physical Layer, the required FLR is less than 6.2e-10 (equivalent to BER of 1e-12 for a link that has no error correction and randomly occurring errors). For 200, 400, and 800 Gb/s, and for 1.6 Tb/s Ethernet Physical Layers, the required FLR for 64-octet frames with minimum interpacket gap is less than 6.2e-11 (equivalent to BER below 1e-13 for a link that has no error correction and randomly occurring errors).

Informative Text:
Although not written explicitly in the IEEE Std 802.3 PHY specifications, it is generally expected that the actual FLR in most network links is several orders of magnitude lower than the values above. This is especially true for large networks typical of UE applications. The maximum FLR that is acceptable is application dependent. This specification is intended to help in defining and verifying FLR requirements. Some examples are provided in 6.3.3.

For Ethernet types supported by UE (see Section 6.1), the link is always protected by Reed-Solomon forward error correction (RS-FEC) code, denoted RS(544,514). Three variations of RS-FEC using the RS(544,514) code are defined in clauses 91, 119, and 161 of the Ethernet standard.

Measuring FLR is impossible in the PHY and is not always practical at the MAC. Additionally, the required measurement times depend on link utilization level, which may be unknown. However, the FLR requirements of Ethernet types supported by UE can be translated to an uncorrectable codeword ratio (UCR), which is independent of traffic and measurable at the PHY level. The relationship between UCR and FLR is described in 6.3.1.1.

The effect of UCR on application performance in a large network is easier to comprehend by converting it to a metric in units of time, the mean time between PHY errors (MTBPE). In this context, a PHY error is an event of an uncorrectable RS-FEC codeword, which can cause one or more consecutive MAC frames to be discarded.

Informative Text:
MTBPE is unrelated to the mean time to false 802.3 packet acceptance (MTTFPA). In a compliant Ethernet link, the probability of having an uncorrectable RS-FEC codeword that is not detected as such is extremely small, and the probability of false 802.3 packet acceptance is even smaller. As a result, the MTTFPA for a single link that meets the FLR requirements is many billions of years.

The worst-case MTBPE for a single link that meets the FLR requirements is different across Ethernet link rates due to the different RS-FEC interleaving schemes and FLR specifications. The resulting MTBPE values are summarized in Table 6-7. The calculation of MTBPE is based on the relationship between performance metrics described in 6.3.1.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Table 6-7 - Ethernet-specified FLR and resulting MTBPE

Ethernet link data rate	Specified maximum FLR (for 64-octet frames with minimum interpacket gap)	Worst-case MTBPE (approx.)
100 Gb/s	6.2e-10	1.5 minutes
200 Gb/s	6.2e-11	14.6 minutes
400 Gb/s	6.2e-11	7.3 minutes
800 Gb/s	6.2e-11	7.1 minutes
1.6 Tb/s	6.2e-11	3.6 minutes

Informative Text:
The worst-case MTBPE would occur for borderline-compliant links. In typical networks, most components are expected to have margins, and the MTBPE on most links can be expected to be higher by several orders of magnitude.

6.3.1 Relationship between performance metrics

6.3.1.1 Relationship between UCR and FLR

UCR can be considered a fundamental quality metric of an Ethernet link. It is a function of lower-level metrics that are not observable and thus are beyond the scope of this specification.

Ethernet uses RS-FEC with a technique called codeword interleaving, in which traffic can be spread across multiple codewords that are interleaved in time. The number of codewords interleaved together is referred to as the codeword interleaving ratio (CIR) and is either 4, 2, or 1, where 1 is equivalent to no interleaving. A higher CIR generally improves the RS-FEC's ability to handle correlated errors and reduces the link UCR while increasing the FLR for a given value of UCR.

Informative Text:
The reduction of UCR due to the increased CIR can be orders of magnitude. In comparison, the increase of FLR for a given UCR is up to a factor of 4, so the net effect is beneficial.

Codeword interleaving varies between data rates and PHY types:

- For 100 Gb/s Ethernet, 2-way interleaving (CIR=2) is defined as OPTIONAL for backplane and copper cable PHYs (100GBASE-KR1 and 100GBASE-CR1). If this option is not selected, then CIR=1. For all other PHY types (specifically optical PHYs), interleaving is not available (CIR=1).
- For 200 Gb/s and 400 Gb/s Ethernet, 2-way interleaving (CIR=2) is used for PHY types using up to 100 Gb/s signaling, and 4-way interleaving (CIR=4) is used for PHY types using 200 Gb/s per lane signaling.
- For 800 Gb/s and 1.6 Tb/s Ethernet, 4-way interleaving (CIR=4) is used for all PHY types.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

A single RS-FEC codeword has a payload of 5140 bits, corresponding to 80 PCS blocks or 640 octets (as observed at the MII interface). A block of codewords that are interleaved together is referred to as a codeword group (CG).

The default behavior of Ethernet RS-FEC is that an uncorrectable codeword causes all data in the CG that this codeword is in to be marked as error by the PCS and discarded. As a result, any MAC frame fully or partially included in a CG that includes an uncorrectable codeword will not be delivered to the MAC and will be lost.

The Ethernet FLR requirements are stated for 64-octet frames with minimum interpacket gap (IPG), which creates 84 octets in total per frame (including IPG and preamble). This condition (maximum utilization with smallest frames) results in up to 8 frames per codeword (FPC), which is the maximum possible number. Due to possible misalignment of frame and codeword boundaries, the maximum number of frames that can be lost due to a single codeword error is $CIR \cdot FPC + 1$.

Informative Text:
The maximum number of frames that can be lost due to a single codeword error is not necessarily dependent on link utilization, since frames with minimum IPG can occur even with low utilization.

Table 6-8 shows the size of a CG in octets and the maximum number of frames that can be lost due to a single uncorrectable codeword as a function of the CIR.

Table 6-8 - CIR and CG sizes

	100 GbE with RS-FEC (Clause 91)	200 GbE and 400 GbE (Clause 119), 100 GbE with RS-FEC-Int (Clause 161)	800 GbE (Clause 172) and 1.6 TbE (Clause 175)
CIR	1	2	4
CG size	640 octets	1280 octets	2560 octets
Max. frames lost due to uncorrectable codeword	9	17	33

The conversion factor from UCR to FLR is:

$$\frac{\boldsymbol{FLR}}{\boldsymbol{UCR}} = \frac{\boldsymbol{CIR \cdot FPC + 1}}{\boldsymbol{FPC}} = \boldsymbol{CIR} + \frac{\boldsymbol{1}}{\boldsymbol{FPC}} \qquad \text{(Equation 1)}$$

As the average frame size increases, FPC decreases, and the conversion factor from UCR to FLR changes accordingly, as shown in Table 6-9.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Table 6-9 - UCR to FLR conversion factors

Average frame size (octets)	FPC	Conversion factor, FLR/UCR		
		100 Gb/s with RS-FEC (Clause 91)	200 Gb/s and 400 Gb/s (Clause 119), 100 Gb/s with RS-FEC-Int (Clause 161)	800 Gb/s (Clause 172) and 1.6 Tb/s (Clause 175)
64	8	1.125	2.125	4.125
1500	0.42	3.36875	4.36875	6.36875
9000	0.07	15.0875	16.0875	18.0875

Due to the dependence of FLR on traffic patterns (such as frame size), it is difficult to predict the link performance using FLR measurement. It is thus preferable to use UCR as the performance metric and, if necessary, convert it to FLR based on traffic characteristics.

Informative Text:
Error marking in the PCS may optionally be bypassed, and in that case, error detection is only performed by the MAC using CRC. When this option is enabled, it is possible that some frames in an uncorrectable codeword will reach the MAC without error, and the frame loss ratio will be somewhat lower. The FLR analysis above is valid for the case where error marking is not bypassed.

6.3.1.2 Relationship between UCR and MTBPE

Unlike the relationship between UCR and FLR described in 6.3.1.1, which is dependent on many factors, the relationship between UCR and MTBPE is simple. For each Ethernet data rate, MTBPE on a single link is a deterministic function of the UCR on that link, which can be estimated from FEC statistics as described in 6.3.2.

The number of codewords in a given period of time can be calculated from the period of a single codeword (5440 bits), denoted T_{CW}. Table 6-10 lists the value of T_{CW} for different Ethernet rates.

Table 6-10 - Codeword times

Ethernet rate	T_{CW} [ns]
100 Gb/s	51.2
200 Gb/s	25.6
400 Gb/s	12.8
800 Gb/s	6.4
1.6 Tb/s	3.2

For a specific link, MTBPE can be calculated using Equation 2:

$$\boldsymbol{MTBPE = \frac{T_{CW}}{UCR}} \qquad \text{(Equation 2)}$$

where T_{CW} is provided in Table 6-10 and UCR is the uncorrectable codeword ratio on the link.

The UCR values corresponding to several values of MTBPE are provided in Table 6-11.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Table 6-11 - MTBPE to UCR conversion

MTBPE	Uncorrectable codeword ratio				
	100 Gb/s	200 Gb/s	400 Gb/s	800 Gb/s	1.6 Tb/s
One minute	8.5e-10	4.3e-10	2.1e-10	1.1e-10	5.5e-11
One hour	1.4e-11	7.1e-12	3.6e-12	1.8e-12	9.0e-13
One day	5.9e-13	3.0e-13	1.5e-13	7.4e-14	3.7e-14
One week	8.5e-14	4.2e-14	2.1e-14	1.1e-14	5.5e-15
One month	2.0e-14	9.9e-15	4.9e-15	2.5e-15	1.2e-15
One year	1.6e-15	8.1e-16	4.1e-16	2.0e-16	1.0e-16
10 years	1.6e-16	8.1e-17	4.1e-17	2.0e-17	1.0e-17

Informative Text:
It is sometimes suggested that in a network comprising N Ethernet links of the same data rate, the UCR of a single link should be multiplied by N or, equivalently, MTBPE should be divided by N. However, this scaling assumes that all links have the same UCR. In large networks, it is often the case that the UCR of most links is very low, but some links have much higher UCRs than the average. Consequently, it can be expected that the MTBPE on the network depends on the UCR of good links multiplied by N and on the UCR of a small number of the weakest links. The examples in 6.3.3 demonstrate this dependence. This provides motivation for identifying the weakest links in the network (which can be done through the methods described in 6.3.2) and taking appropriate measures suitable for the application, e.g., routing traffic away from such links or scheduling replacement of parts.

6.3.2 Estimation of UCR from FEC statistics

Ethernet specifications include several RS-FEC counters that can be used to estimate the probability of an RS-FEC codeword to be uncorrectable and thus provide an estimate of the UCR. From the UCR, the MTBPE can be calculated by multiplication with the period of a single codeword.

The following counters are defined for Ethernet BASE-R PHY types that include RS-FEC functionality and are mapped to MDIO registers. All registers are defined as clear-on-read and non-rollover. Implementation of these counters and registers, and the method of accessing them, may depend on PCS implementation.

- **FEC_uncorrected_cw_counter** (see [1], subclause 45.2.3.63) is a 32-bit counter that counts once for each codeword that contains errors that were not corrected (i.e., contains more than *t* symbol errors, where *t*=15 for the RS(544,514) FEC).
- **FEC_corrected_cw_counter** (see [1], subclause 45.2.3.62) is a 32-bit counter that counts once for each codeword that contains errors that were corrected (i.e., contains between 1 and *t* symbol errors).
- **FEC_cw_counter** (see [3], subclause 45.2.1.120a) is a 48-bit counter once for each FEC codeword received. Note that this counter is defined as optional in the Ethernet standard.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

- **FEC_codeword_error_bin_*i*** (see [3], subclause 45.2.1.131a), where *i* takes a value from 1 to 15. Each of these counters is a 32-bit counter that counts once for each codeword that contains exactly *i* symbol errors. Note that these counters are defined as optional in the Ethernet standard.

A UE PHY that supports the OPTIONAL MTBPE estimation SHALL include all of these counters and have them accessible to software.

Reading these counters periodically can provide the probability of specific events by dividing the read value by the total number of codewords since the previous readout, denoted N_{total}. N_{total} may be obtained by reading **FEC_cw_counter** if that counter is available; alternatively, it can be estimated quite accurately by the time since the previous reading. In this specification, it is assumed that the counters are read regularly once per minute, and the approximate values of N_{total} for this period are provided in Table 6-12. Other observation periods may be used instead, with calculations adjusted as appropriate.

Table 6-12 - Number of codewords received in one minute, N_{total} (approximate)

Ethernet link data rate	N_{total} in one minute (approximate)
100 Gb/s	1.2e9
200 Gb/s	2.3e9
400 Gb/s	4.7e9
800 Gb/s	9.4e9
1.6 Tb/s	1.9e10

Methods of estimating the UCR of an active link using these registers are described in 6.3.2.1, 6.3.2.2, and 6.3.2.3.

Informative Text:
The counters above are defined at the PCS decoder, and thus UCR estimates can be calculated only at the receive side of the link. However, a high UCR could indicate issues unrelated to the receiver, such as in the link partner's transmitter or in the cabling.

6.3.2.1 Estimation of UCR using the uncorrected codewords counter

If only **FEC_uncorrected_cw_counter** is available, an estimate of UCR is simply:

$$UCR_{est1} = \frac{FEC_{uncorrected_{cw_{counter}}}}{N_{total}} \quad \text{(Equation 3)}$$

For an observation period of one minute, N_{total} is provided in Table 6-12.

For links in a large network, the UCR is expected to be lower than 1e-12 (and preferably much lower). Therefore, when reading once per minute, the value of **FEC_uncorrected_cw_counter** should almost always be 0. If the UCR needs to be estimated using only this counter, the calculated values of UCR_{est1} should be averaged over longer periods of time. Also, this estimate cannot be used proactively to predict the MTBPE, since the estimated UCR can be calculated only after some PHY error events occur.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

6.3.2.2 Estimation of UCR using the corrected codewords counter

If **FEC_corrected_cw_counter** is available, the UCR can be estimated based on the corrected codeword ratio (CCR), defined by:

$$\boldsymbol{CCR} = \frac{\boldsymbol{FEC}_{\boldsymbol{corrected}_{\boldsymbol{cw}_{\boldsymbol{counter}}}}}{\boldsymbol{N}_{\boldsymbol{total}}} \quad \text{(Equation 4)}$$

For an observation period of one minute, N_{total} is provided in Table 6-12.

Unlike **FEC_uncorrected_cw_counter**, when **FEC_corrected_cw_counter** is read once per minute, most values are expected to be nonzero. Thus, estimates can be made after shorter periods and before actual PHY error events.

Assuming that uncorrectable codewords are rare compared with correctable codewords, the CCR is approximately the probability that a codeword has a nonzero number of symbol errors (neglecting the case that it is uncorrectable). Denoting the probability of an RS-FEC symbol error by symbol error ratio (SER), we have:

$$\boldsymbol{CCR} \approx \mathbf{1} - (\mathbf{1} - \boldsymbol{SER})^{\mathbf{544}} \quad \text{(Equation 5)}$$

And thus,

$$\boldsymbol{SER} \approx \mathbf{1} - (\mathbf{1} - \boldsymbol{CCR})^{\frac{\mathbf{1}}{\mathbf{544}}} \quad \text{(Equation 6)}$$

Assuming symbol errors occur as a stationary process (uncorrelated error model), UCR can be estimated by the expression:

$$\boldsymbol{UCR}_{\boldsymbol{est2}} = \mathbf{1} - \sum_{\boldsymbol{i}=\mathbf{0}}^{\mathbf{15}} \binom{\mathbf{544}}{\boldsymbol{i}} \boldsymbol{SER}^{\boldsymbol{i}} (\mathbf{1} - \boldsymbol{SER})^{\mathbf{544}-\boldsymbol{i}} \quad \text{(Equation 7)}$$

For links that use 1/(1+D) precoding (available in CR and KR PHY types), errors occur in pairs. Consequently, the estimate of Equation 7 is inadequate in the following specific case: 100 Gb/s Ethernet with the IEEE Std 802.3-2022 clause 91 RS-FEC (where interleaving is not used) and precoding enabled. This case has a much higher UCR for the same SER. For links requiring high MTBPE, it is RECOMMENDED to use IEEE Std 802.3-2022 clause 161 RS-FEC_int instead.

Several values of UCR_{est2} based on CCR (assuming uncorrelated errors, with no guard band) are provided in Table 6-13. A graphical representation of the relationship between CCR and UCR_{est2} is shown in Figure 6-1.

Table 6-13 - Calculation of UCR_{est2} from CCR

CCR (measured)	UCR_{est2} (calculated) Assuming uncorrelated errors
0.132	1e-25
0.152	1e-24
0.175	1e-23
0.202	1e-22
0.231	1e-21
0.265	1e-20

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

CCR (measured)	UCR_{est2} (calculated) Assuming uncorrelated errors
0.344	1e-18
0.439	1e-16
0.549	1e-14
0.668	1e-12
0.728	1e-11
0.820	4.3e-10
0.836	8.5e-10

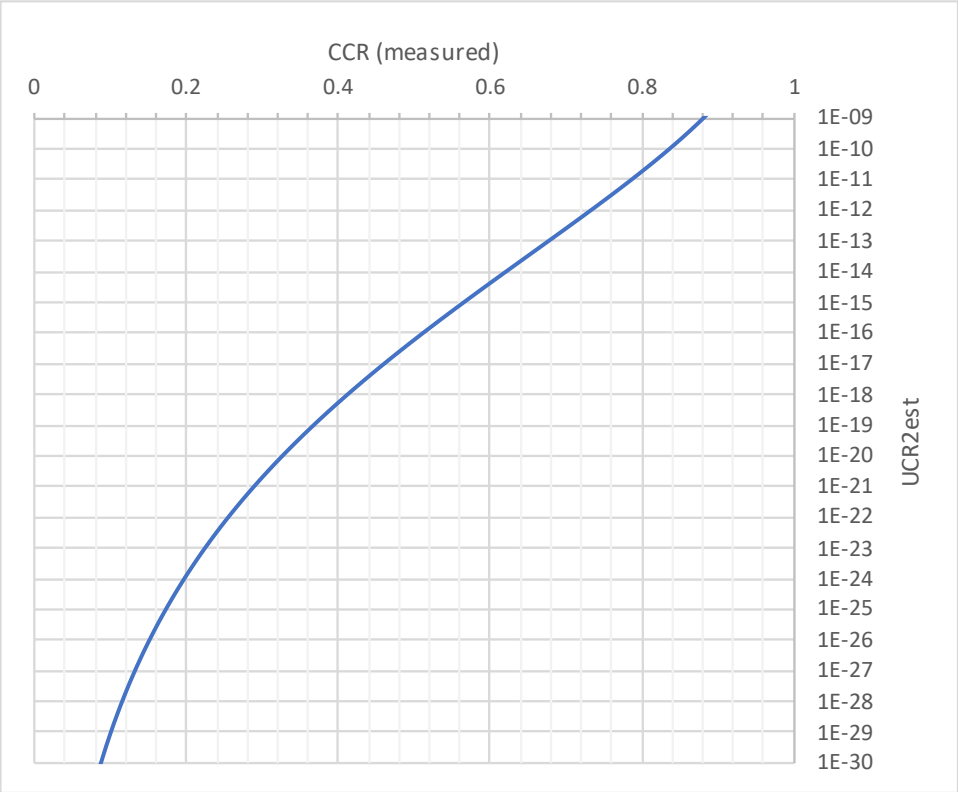


Figure 6-1 - UCR_{est2} as a function of CCR, assuming uncorrelated errors

The uncorrelated error model is not accurate in many practical cases. Correlated errors increase the UCR, so this estimate is biased and can predict a higher MTBPE than the true value. The effect of correlated errors can increase the actual UCR by an unknown factor, and therefore UCR_{est2} should be considered inaccurate and not be used to predict the MTBPE of a link. Despite this possible inaccuracy, this estimate can be useful for identifying outliers in the statistics of links in a large network.

6.3.2.3 Estimation of UCR using the codeword error bin counters

If **FEC_codeword_error_bin_i** are available, the UCR may be estimated based on these counters. This estimate is more reliable than the one based on **FEC_corrected_cw_counter** (see 6.3.2.2) because it accounts for correlated errors, too.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

The UCR can be estimated based on the probability of having k or more symbol errors in a codeword, defined by:

$$\boldsymbol{P_k = Prob(codeword\ has\ \geq k\ symbol\ errors) = \frac{N_k}{N_{total}}} \quad \text{(Equation 8)}$$

where N_k is the sum of the values read from counters k and up:

$$\boldsymbol{N_k = \sum_{i=k}^{15} FEC_{codeword_{error_{bin_i}}}} \quad \text{(Equation 9)}$$

For an observation period of one minute, N_{total} is provided in Table 6-12.

If the errors are uncorrelated, then P_k depends on the RS-FEC SER as follows:

$$\boldsymbol{P_k = \sum_{i=k}^{544} \binom{544}{i} SER^i (1-SER)^{544-i} \approx \sum_{i=k}^{15} \binom{544}{i} SER^i (1-SER)^{544-i}} \quad \text{(Equation 10)}$$

And the UCR for uncorrelated errors is:

$$\boldsymbol{UCR = \sum_{i=16}^{544} \binom{544}{i} SER^i (1-SER)^{544-i}} \quad \text{(Equation 11)}$$

It is thus possible to create a mapping between P_k and UCR for each value of k, using SER as a parameter.

Informative Text:
Calculation of P_k and UCR as in equations 10 and 11 is susceptible to numerical errors. These calculations can be performed by various mathematical tools that provide binomial distribution functions. As an example, the Excel function BINOM.DIST can be used to calculate P_k using the formula "=BINOM.DIST(544-k, 544, 1-SER, TRUE)".

Correlated errors can create different statistics than uncorrelated errors. In this method, the effect of correlated errors is accounted for by calculating the UCR using P_k for several values of k and taking the worst of the results as the estimate. If the estimates obtained with different values of k do not significantly deviate from each other, it is an indication that the errors are uncorrelated.

Informative Text:
A specific example of correlated errors results from using 1/(1+D) precoding in the PHY. Precoding typically results in errors occurring in pairs, and thus the statistics of errors with precoding is different than it is without precoding, because even-numbered bins represent single-error events. Inspecting several even values of k enables detection of poor links whether precoding is used or not.

Assuming an observation period of one minute, the N_{total} values provided in Table 6-12 are between 1e9 and 1e10, and probabilities of 1e-8 and higher can be verified with a reasonable level of confidence. For a probability of 1e-8, the values of P_6, P_8, and P_{10} can be calculated by periodically reading the set of counters **FEC_codeword_error_bin_i**, calculating the cumulative counts N_k, and dividing them by N_{total},

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Informative Text:
Usage of P6, P8, and P10 is a specific choice that is considered suitable for estimating UCR down to about 1e-20 when registers are read once per minute. Other values of k can be used instead, with calculations adjusted as appropriate, to provide faster measurement or better resolution. Specifically, if there are indications of correlated errors, higher values of k may provide better estimates.

per Equation **8**. These values can be used to find the SER values with the uncorrelated errors model, from which corresponding UCR values can be found, denoted UCR_6, UCR_8, and UCR_{10}, respectively.

The relationship between P_6, P_8, and P_{10} and UCR (assuming uncorrelated error model) is listed in Table 6-14. Figure 6-2 shows this relationship on a log-log scale, on which it is close to a straight line for each of the values. The linear trend equations shown in the figure can be used for numeric conversion.

Table 6-14 - Mapping from UCR to P6, P8, and P10 assuming uncorrelated errors

UCR	P_6	P_8	P_{10}
1e-22	7.0e-7	1.1e-9	1.0e-12
1e-21	1.6e-6	3.3e-9	4.1e-12
1e-20	3.8e-6	1.0e-8	1.7e-11
1e-19	8.6e-6	3.1e-8	7.1e-11
1e-18	2.0e-5	9.6e-8	2.9e-10
1e-17	4.5e-5	2.9e-7	1.2e-9
1e-16	1.0e-4	8.9e-7	4.9e-9
1e-15	2.3e-4	2.7e-6	2.0e-8
1e-14	5.0e-4	8.0e-6	8.0e-8
1e-13	1.1e-3	2.4e-5	3.2e-7
1e-12	2.4e-3	7.0e-5	1.3e-6

UCR_{est3} is then determined by the highest UCR of the three:

$$\boldsymbol{UCR_{est3} = \max(UCR_6, UCR_8, UCR_{10})} \qquad \text{(Equation 12)}$$

When the counters are read once per minute, the resolution of P_k will not be better than 1e-9, and for some values of k, it may be estimated as 0; in that case, the corresponding UCR can be taken as the minimum value in the table. Values of P_{10} corresponding to N_{10}=1 may occur occasionally and may be ignored. For better resolution, the value of N_{10} can be averaged over time.

With correlated errors, the calculations of P_k in Equation 10 and of UCR in Equation 11 would not be valid. However, the relationship between P_k and UCR as shown in Table 6-14 is expected to hold for mildly correlated errors (and the accuracy improves for larger values of k).

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

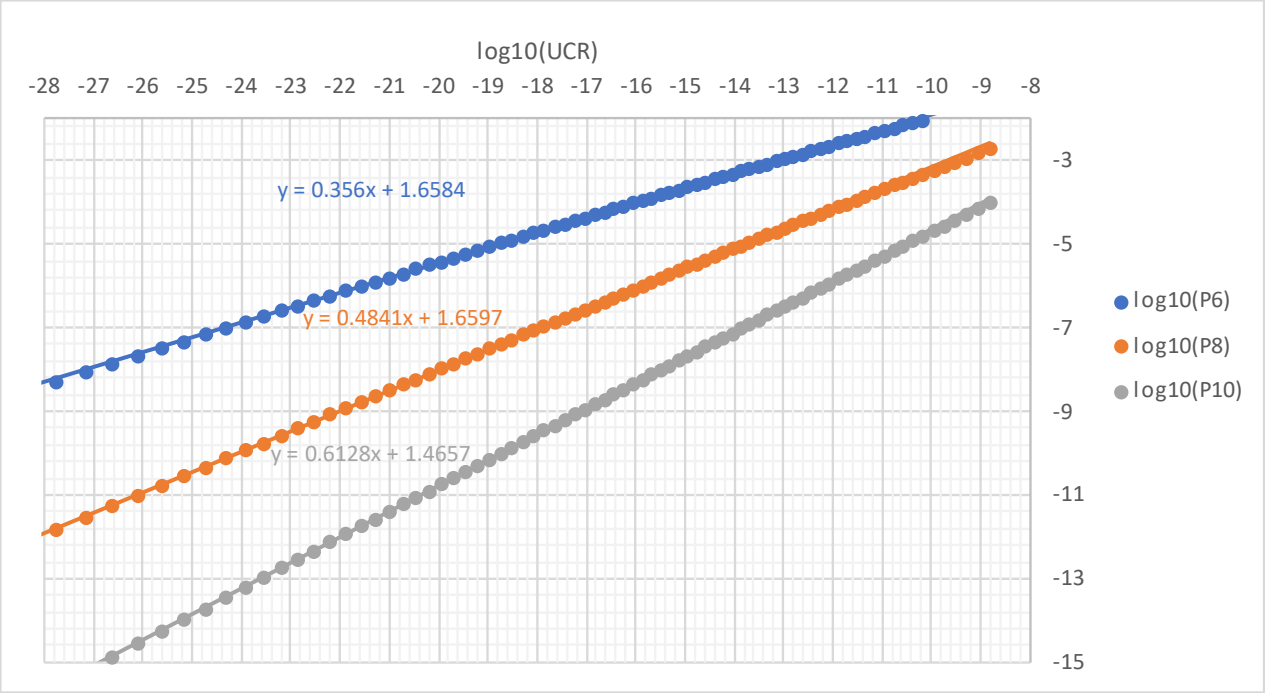


Figure 6-2 - Expected values of P6, P8, and P10 as a function of UCR

6.3.3 Examples

6.3.3.1 MTBPE of a large network

Consider a network with N=100000 links with 400 Gb/s each, where most links have a UCR of ~1e-20 but K=10 links are barely compliant and have a UCR of 1e-10 (where the UCR is estimated through either of the methods described in 6.3.2).

From Table 6-11, the MTBPE of a single bad link (M_1) is about 1 minute, and the MTBPE of a single good link (M_2) is about 20 thousand years. The frequency of errors in the full network is $\frac{K}{M_1} + \frac{N-K}{M_2}$, and the MTBPE is the reciprocal of this value, about 6 seconds.

It can be observed that, for the MTBPE of the network, the worst-case UCR is multiplied by K rather than by N.

If the K=10 worst links have a UCR of 1e-16 instead (corresponding to an MTBPE of about 2 years), the MTBPE of the full network becomes about 7 days. In this case, the MTBPE is mostly governed by the total number of links N, and further improvement of the worst links has negligible effect.

6.3.3.2 Monitoring a network

Consider a network with N=1000 links with 800 Gb/s each. A software agent runs on each node (endpoint or switch) and monitors the PCS FEC counters of each active link.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

One node in the network, denoted endpoint A, has a PCS with only the **FEC_corrected_cw_counter** available. The agent on node A reads this counter once per minute, and the average value is approximately 1e9.

- From Table 6-12, for 800 Gb/s, the value of N_{total} is 9.4e9.
- Using the method in 6.3.2.2, the CCR is approximately 1e9/9.4e9≈0.1.
- Looking at Figure 6-1, this value of CCR corresponds to a UCR of about 1e-29 with uncorrelated errors. Taking a guard band of four orders of magnitude, the estimated UCR is 1e-25.
- From 6.3.1.2, the MTBPE can be estimated as 6.4e-9/1e-25≈6.4e16 seconds, or more than 2 billion years.
- The software agent on node A can report this estimate or just that the link is not expected to experience PHY errors.
- Note that there is no way to tell from the **FEC_corrected_cw_counter** alone whether errors are correlated and thus whether this estimate is reliable. If PHY errors occur despite this estimate, it is likely due to correlated errors.

Another node in the network, denoted node B, also has a PCS with only the **FEC_corrected_cw_counter** available. If the agent on node B reads this counter once per minute, the counter is saturated at 2^32 ≈ 4.3e9. To enable estimation of the MTBPE, the agent has to read the counter more frequently to prevent saturation. When reading the counter once per second, the average count turns out to be about 1.1e8.

- The value of N_{total} for an observation time of one second is 9.4e9/60≈1.57e8.
- Using the method in 6.3.2.2, the CCR is 1.1e8/1.57e8≈0.7.
- Looking at Figure 6-1, this value of CCR corresponds to a UCR higher than 1e-13.
- Even without applying any guard band, from Table 6-11, a UCR of 1e-13 corresponds to an MTBPE of less than one day. Thus, node B is expected to experience frequent PHY errors.
- The software agent can report the estimated MTBPE to a network monitoring system so that the specific link can be handled proactively.

Another node in the network, denoted node C, has a PCS with **FEC_codeword_error_bin_i** counters available. The agent on node C reads these counters once per minute to obtain N_6, N_8, and N_{10}. The average values are approximately 90000, 440, and 1, respectively, where N_{10} is calculated by averaging multiple readouts (the values are mostly between 0 and 2).

- From Table 6-12, for 800 Gb/s and 1-minute observation time, the value of N_{total} is 9.4e9, or approximately 1e10.
- Dividing the average counter values by 1e10, we get that P_6, P_8, and P_{10} are about 9e-6, 4.4e-8, and 1e-10, respectively.
- Looking at Figure 6-2, these values of P_6, P_8, and P_{10} correspond to UCR values of approximately 1.1e-19, 2.0e-19, and 1.7e-19, respectively. Taking the maximum of the three values, the UCR is estimated as 2e-19. Since the values are close, the estimate can be considered accurate.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

- From 6.3.1.2, the MTBPE can be estimated as 6.4e-9/2e-19=3.2e10 seconds, or about 1000 years. Thus, node C is not expected to experience frequent PHY errors.
- The software agent can report the estimated MTBPE to a network monitoring system that aggregates information from all ports on the network. If all 1000 nodes in the network report similar MTBPE estimates, the full network is expected to have a PHY error once a year.

6.3.3.3 Planning a large-scale network with reliability requirements

Consider a plan to build a network with 1 million links of 100 Gb/s each. Two options for reliability requirements are considered:

- Option A targets minimization of retransmissions due to PHY errors, with a requirement that the network-wide MTBPE is less than once per day.
- Option B assumes that occasional retransmissions are acceptable and allows a network-wide MTBPE up to once per minute.

Assuming that all links have the same performance, the minimum MTBPE on each link has to be at least 1 million days (8.6e10 seconds) for option A or 1 million minutes (6e7 seconds) for option B.

At 100 Gb/s, the codeword period is 51.2 ns (see 6.3.1.2), so these requirements can be translated to a maximum UCR of approximately 6e-19 for option A or approximately 8.5e-16 for option B.

Assuming that all nodes have observable **FEC_codeword_error_bin_i** counters, the estimation method in 6.3.2.3 can be used. Using the approximate log-linear relationships shown in Figure 6-2, the maximum allowed values of P_6, P_8, and P_{10} are approximately 1.6e-5, 7.5e-8, and 2.1e-10, respectively, for option A or 2.1e-4, 2.5e-6, and 1.8e-8, respectively, for option B.

From Table 6-12, at 100 Gb/s, N_{total} is approximately 1.2e9 per minute. To verify the UCR requirements, network agents on each node can read the **FEC_codeword_error_bin_i** counters once per minute. With this observation period, the values of N_6, N_8, and N_{10} corresponding to the maximum UCR are approximately 19 800, 90, and 0.26, respectively, for option A or approximately 257000, 2990, and 22, respectively, for option B.

With either option A or option B, if the counters exceed the maximum values on any of the nodes, that node might fail the per-node UCR requirement, and the link associated with it can be considered a weak link. However, the overall network performance can still be acceptable if other links are better than the requirement and compensate for the weak link. It is therefore preferable that a software agent on each node reports the MTBPE to a network monitoring system that aggregates information from all ports on the network to assess the network-wide MTBPE. If the network-wide MTBPE is shorter than the reliability requirement, the weak links should be identified and handled.

6.4 Recommendations

To meet the requirements of AI and HPC environments and enable efficient and scalable communication, special attention is needed in the following areas of PHY and channel design.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

6.4.1 Low error rate

To improve performance and efficiency of communication in UE networks and applications using them, it is RECOMMENDED to minimize the error rate through optimized PHY and channel design and close monitoring of the network.

Ethernet links with higher-than-minimum MTBPE (i.e., on the order of hundreds of years) could reduce 802.3 packet retry and improve the link bandwidth efficiency and overall network performance. It is RECOMMENDED that network management uses MTBPE estimates collected from multiple links, if available, to configure the network for improved application performance.

Informative Text:
The choice of network configuration methods depends on the application. Network configuration can include settings above the physical layer, such as changing routing tables to avoid usage of poor links or enablement of reliability features (such as LLR) for specific links.

6.4.2 Low power

Power consumption is usually one of the limiting factors for scaling AI and HPC systems. It is RECOMMENDED to have a power-optimized PHY that supports a wide range of channel conditions, with configurable power tailored to each usage case.

Emerging technologies such as linear optics and co-packaged optics can also potentially reduce the power of the link.

6.4.3 Low latency

Latency plays a critical role in the performance of AI and HPC systems, and the PHY latency can be an important factor in the overall latency. It is RECOMMENDED to optimize both average and tail latencies for robust and high-performance networks.

Average latency: Examples of PHY technologies under development that would potentially be beneficial to average latency include: low-latency FEC error correction solutions; reducing lane-to-lane skew, as the link latency is dictated by the slowest lane; and new electro-optical integration (e.g., optical direct drive).

Tail latency: Reducing PHY error rate, as discussed in Section 6.4.1, and the resulting 802.3 packet retransmit through link or end-to-end mechanisms, can significantly reduce tail latency. This is especially important for AI and HPC with scalable applications relying on global synchronization.

6.5 References

[1] IEEE Std 802.3-2022, "IEEE Standard for Ethernet," 2022. [Online]. Available: https://ieeexplore.ieee.org/document/9844436.

[2] IEEE Std 802.3db-2022, "IEEE Standard for Ethernet - Amendment 3: Physical Layer Specifications and Management Parameters for 100 Gb/s, 200 Gb/s, and 400 Gb/s Operation over Optical Fiber

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

using 100 Gb/s Signaling," 2022. [Online]. Available: https://ieeexplore.ieee.org/document/9988984.

[3] IEEE Std 802.3ck-2022, "IEEE Standard for Ethernet Amendment 4: Physical Layer Specifications and Management Parameters for 100 Gb/s, 200 Gb/s, and 400 Gb/s Electrical Interfaces Based on 100 Gb/s Signaling," 2022. [Online]. Available: https://ieeexplore.ieee.org/document/9999414.

[4] IEEE Std 802.3df-2024, "IEEE Standard for Ethernet - Amendment 9: Media Access Control Parameters for 800 Gb/s and Physical Layers and Management Parameters for 400 Gb/s and 800 Gb/s Operation," 2024. [Online]. Available: https://ieeexplore.ieee.org/document/10472445.

[5] IEEE P802.3dj/D3.0, February 2026, “IEEE Draft Standard for Ethernet Amendment 13: Media Access Control Parameters for 1.6 Tb/s and Physical Layers and Management Parameters for 200 Gb/s, 400 Gb/s, 800 Gb/s, and 1.6 Tb/s Operation,” 2026. [Online]. Available: https://ieeexplore.ieee.org/document/11408395.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

7 UE Management

“Ultra Ethernet, Ultra Ethernet Consortium, and the Ultra Ethernet Consortium logo are the unregistered trademarks of Ultra Ethernet Consortium in the United States and other countries. Other brands, names, or logos contained herein are the property of their respective owners.

7.1 UE Management Scope

The UE specification covers the layers between the UE PHY and UE software. The initial focus of UE management is to define management specifications for the UE link layer and the UE PHY layer. The following management aspects are covered in the management specifications:

1. PHY configuration for speeds and auto-negotiation.
2. PHY statistics including FEC statistics.
3. Link level retry (LLR) configuration and statistics.
4. Credit-based flow control (CBFC) management including VC mapping, counters, VC/buffer/credit/scheduler configuration, and PFC co-existence.
5. CSIG management.
6. LLDP management including support for options TLV, CBFC TLV, etc.

7.2 High-Level UE Management Architecture

UE management architecture is focused on Ultra Ethernet fabric, switch, and endpoint management. The UE management architecture includes management protocols, transport, and data models. UE management protocol specifications cover baseline operations for management data, topology discovery, transport(s) for management data, security of transports, and streaming support for telemetry data. UE management data models define Ultra Ethernet configuration, state, and events. The UE management schema describes the structure of UE data models.

Figure 7-1 provides a high-level architecture of the UE management stack. The UE Management Controller (UEMC) is a logical entity that manages UE components including UE fabric endpoints, ports, NICs, and switches. The UEMC communicates with one or more UE Management Server (UEMS) instances to access management data and perform management operations. An instance of UEMS can reside on a server or a switch.

Each UEMS exposes a management data model via a protocol over a transport. The UEMS exposes the Redfish data model via a RESTful protocol over HTTPS for a server platform. The UEMS exposes the YANG (Yet Another Next Generation) data model via an OpenConfig protocol over gNMI/gRPC for a switch platform.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

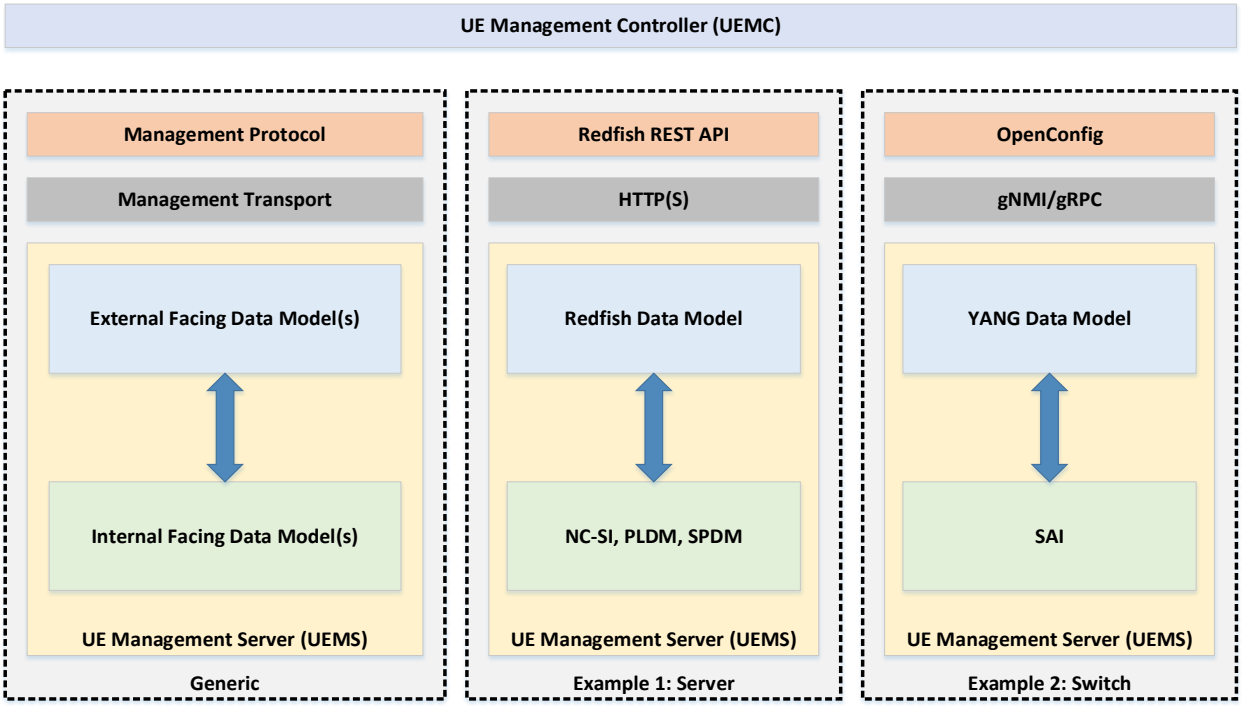


Figure 7-1 - UE Management Architecture

The UE management stack is supported by a variety of industry management standards. To drive interoperability and common management functionality, UE management specifications leverage industry standards, Redfish (internationally recognized by the IEC), YANG, and SAI that is prescribed by the OCP, to cover server and fabric components of Ultra Ethernet.

The Redfish standard specifies an HTTP RESTful interface for exposing system manageability. The Redfish standard consists of an interface specification and a data model specification. The model primarily covers managing the physical and logical aspects of multiple systems. The Redfish data model is described by a schema that is available in OpenAPI, JSON schema, and CSDL formats. The Redfish data model has been extended by other standards to use Redfish in their management domains. Redfish has data models for network endpoints and network fabric. The Redfish data models are extended to manage UE endpoints at the server.

Redfish maps well onto internal facing data models specified by standards Network Controller Sideband Interface (NC-SI), Platform Level Data Model (PLDM), and Security Protocol Data Model (SPDM). These internal-facing data models may be extended for the UE.

YANG defines vendor-neutral common data models for switch configuration and management. YANG supports vendor-neutral automation that can be leveraged for testing and compliance. Any applicable configuration of a UE fabric switch including LLDP, LLR, UET, or INC can be specified using YANG data models. Reference YANG models are specified for the UE. UE reference YANG models can be leveraged by other organizations to create domain-specific YANG models for managing UE fabric switches (e.g., OpenConfig).

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Switch Abstraction Interface (SAI) is widely supported by switch vendors as the abstraction layer to program switch hardware. The following are the benefits of using SAI as an interface specification for UE-capable network switches:

- SAI is an abstract API schema consisting of “C” language header files. It has well-defined conventions, and it provides, at its core, a CRUD abstraction based on tables and attributes.
- Nearly every switch vendor offers SAI implementation. Using SAI, UE extensions can be implemented in existing switches.
- SAI does not impose any restrictions on the kind of network operating system (NOS) using SAI API for any HW programming.
- SAI can be programmatically transformed into other schemas such as gNMI, gRPC, etc.
- The SAI ecosystem contains numerous tools to parse SAI headers and perform various checks and transformations, such as breaking change detection from previous versions.
- The SAI ecosystem has at least two popular test frameworks: SAI-PTF and SAI-Challenger. Thousands of test cases already exist to verify compliance and performance of traditional networking features. Adding test cases for UE enhancements can be a straightforward, incremental effort.

To support Ultra Ethernet features requiring UE switch programming, the SAI specification requires augmentation and updates. UEC works on the SAI enhancements for UE specifications and collaborates with the OCP SAI community for adoption.

7.3 UE Fabric Management

UE fabric management can be viewed as a logical function that sits on top of management interfaces to UE NICs and switches. The UE fabric manager provides a coherent view of a UE fabric and provides a single management entity for creating connections between server NIC endpoints and switch endpoints. Figure 7-2 shows a conceptual view of the UE fabric manager. The specification of UE fabric manager is outside the scope of UE management specifications.

In Figure 7-2, the fabric manager manages a UE NIC using an out-of-band interface and/or an in-band interface. The Redfish data model is used to manage UE endpoints when an out-of-band interface is used. The fabric manager uses YANG data models to manage a UE switch. Within the UE switch, the control plane uses SAI to communicate with the data plane. A vendor-specific SDK is used by the layers below SAI to directly communicate with the switch ASIC pipeline.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

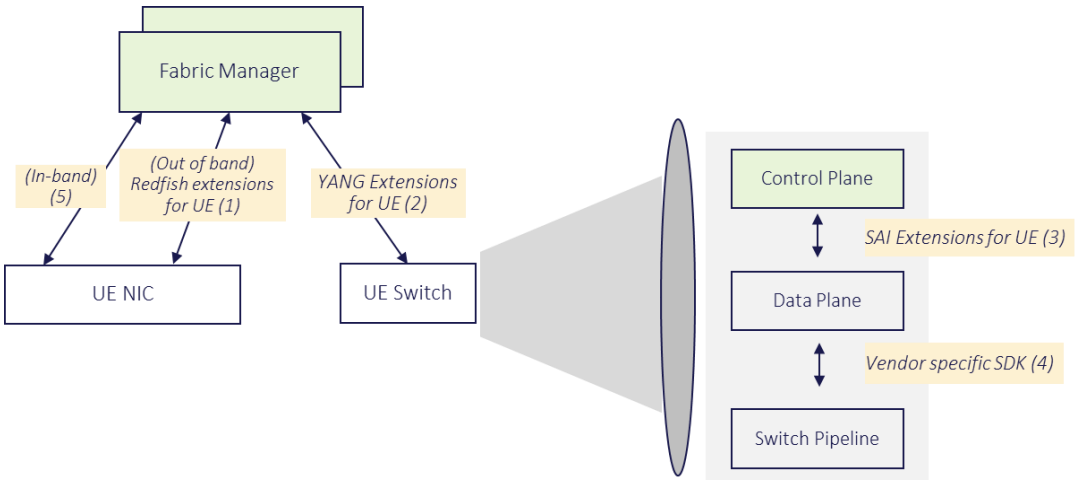


Figure 7-2 - UE Fabric Manager

7.4 Redfish for UE Endpoint Manageability

Remote management of the server can be done via two manageability paths, as shown in Figure 7-3: in-band or out-of-band management. The in-band path uses the same resources that hosts the operating system. The out-of-band path uses dedicated resources that are operationally independent of the state of the in-band resources. This path enables the system to be managed before the OS is operational or when the OS is non-operational (hung or shutdown). The BMC (baseboard management controller) is the component that exposes the Redfish interface to two management entities. The out-of-band interface is accessed by the remote client. The in-band interface (host interface) is accessed by the in-band entities (BIOS, OS, software agent).

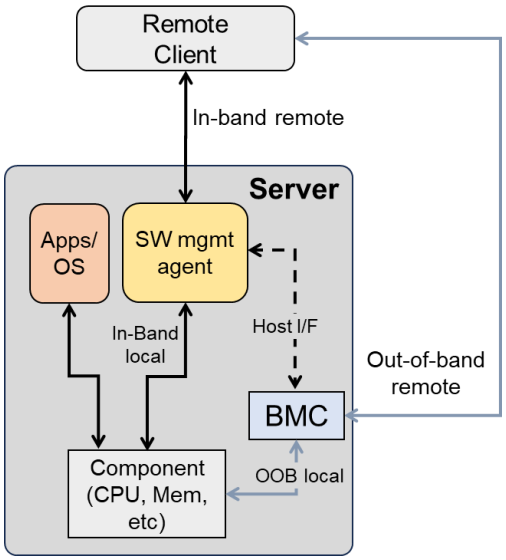


Figure 7-3 - Server Management Architecture

OCP prescribes common manageability for the OCP platform using Redfish profiles. A profile uses the Redfish Interop Validator (open source) to verify the conformance of an implementation.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

UE NIC endpoint manageability shall be prescribed by extending the OCP Ethernet NIC profile. The OCP Ethernet NIC profile prescribes basic manageability that includes:

- Usage guide: Use-case table that shows request-response interaction.
- Profile (JSON): Profile reflects the usage guide and normatively prescribes the Redfish model elements required in implementation. The UE NIC profile extends the OCP Ethernet NIC profile and prescribes UE specific manageability. Table 7-1 lists the use cases covered by the OCP Ethernet NIC profile.

Table 7-1 - OCP Ethernet NIC profile use cases

Use Case	Management Task	Requirement
NIC Configuration	Get NIC information	Mandatory
	Get device function information	Mandatory
	Get port information	Mandatory
	Get PCIe information	Mandatory
	Get PCIe function information	Mandatory
	Set NIC information	Mandatory
	Set device function information	Mandatory
	Set port information	Mandatory
	Set NIC settings to default	Mandatory
NIC Hardware	Get FRU information	Mandatory
System NIC	Get system NIC information	Mandatory
Telemetry	Get NIC metrics	Mandatory
	Get device function metrics	Mandatory
	Get port metrics	Mandatory
Firmware Update	Get firmware version	Mandatory
	Update firmware	Mandatory

7.5 YANG Model for UE Network Manageability

7.5.1 Network Switch Planes

As shown in Figure 7-4, network switches operate on different planes: management, control, and data planes.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

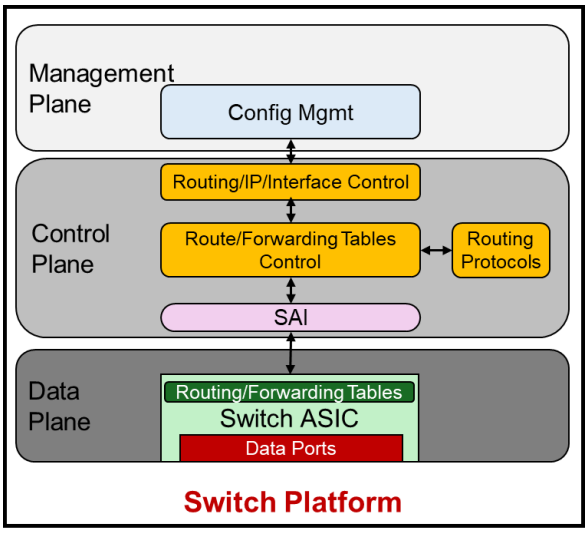


Figure 7-4 - Network Switch Planes

- **Management plane** runs management protocols (e.g., OpenConfig) and manages configuration of routing protocols. The management plane is also responsible for IP and interface management.
- **Control plane** runs routing protocols (e.g., OSPF) and manages routing and forwarding tables. The control plane uses SAI to interface with switch ASIC.
- **Data plane** handles forwarding or routing of data traffic and maintains routing and forwarding tables. The data plane is responsible for performing per-packet functions (e.g., NAT, stat, ACLs).

7.5.2 UE Switch Manageability Architecture

The UE switch manageability architecture builds on top of a switch management plane.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

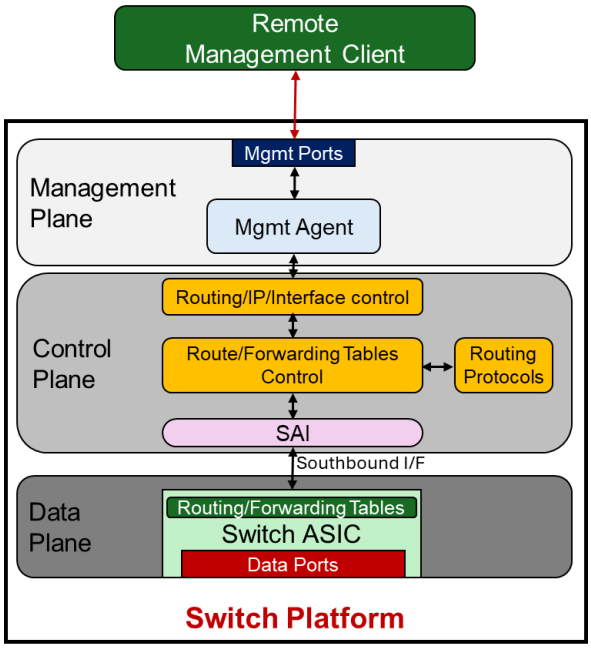


Figure 7-5 - UE Switch Manageability Architecture

The components of the UE switch manageability architecture can be described as follows.

- **Network switch** is a component of the network fabric. Data ports are used for inter-server or inter-node fabric communications. Management port(s) of the switch platform connect to a separate and dedicated physical network infrastructure used for managing switches.
- **A remote management client** manages the switch and directly communicates over management ports with the management agent on the switch. The management interface referenced here is the northbound interface from the NOS on the switch.
- **Management agent** resides inside the NOS and exposes the northbound interface for management. Typically, the management agent is accessed over management ports. Remote management communication may happen over data ports.
- **SAI** is the southbound interface on the switch. SAI is not visible to the remote management client. SAI is used for configuring and monitoring the switch ASIC.

7.5.3 YANG Model Extensions for UE

YANG (Yet Another Next Generation) is a data modeling language used to define the configuration and operational state data of network devices for management and automation. YANG provides a hierarchical, tree-based structure to model data, including configuration data, operational state, remote procedure calls (RPCs), and event notifications. YANG is protocol-independent and can be accessed by various protocols including OpenConfig and Netconf. YANG models can be serialized into various formats such as XML and JSON, facilitating automation and communication between network devices

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

and management systems. UE YANG model specifications will provide a reference model that can be used for UE switch management.

7.6 Switch Abstraction Interface (SAI) for UE

SAI is a subproject under the OCP Networking Project. Figure 7-6 shows the architecture of SAI.

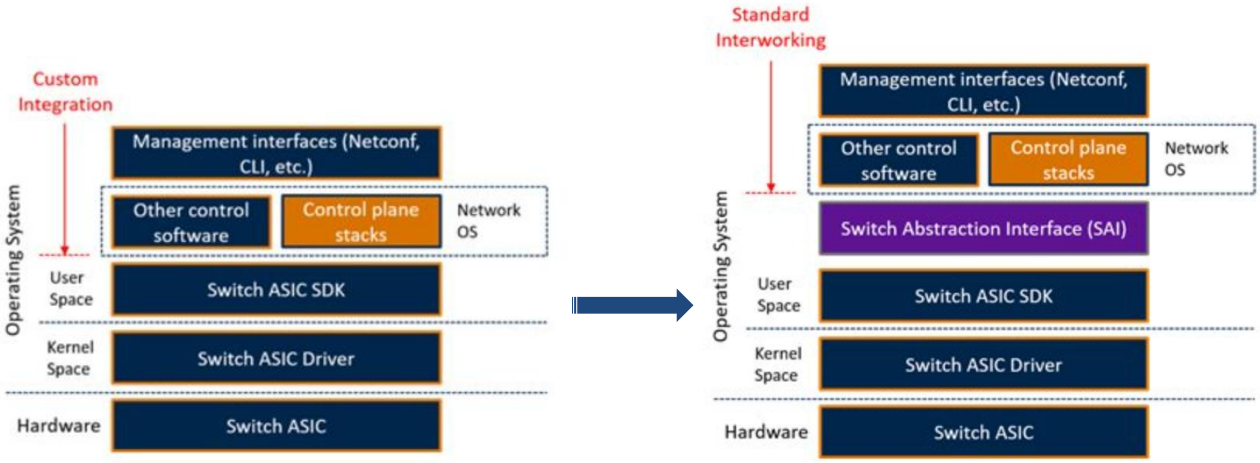


Figure 7-6 - SAI Architecture

7.6.1 SAI Enhancements for UE

SAI enhancements for UE leverage the SAI pipeline and SAI objects. SAI enhancements for LLR, CBFC, and CSIG are specified.

7.6.1.1 SAI Enhancements for LLR

Link layer retry (LLR) is a hop-by-hop mechanism that provides the capability for local retransmission of frames lost due to link errors, thereby avoiding the impact on application performance. LLR uses sequence numbers and periodic acknowledgements to track Ethernet frames. LLR is a new feature in SAI. To support LLR, new attributes and statistics are added to the SAI port object. Additionally, a new object called a *Port* LLR profile is introduced to model the LLR parameters of a port.

7.6.1.2 SAI Enhancements for Credit-Based Flow Control (CBFC)

In CBFC, the sender ensures that the receiver has enough buffer before transmitting a frame. The receiver advertises its buffer availability to the sender using credits that represent the amount of buffer space available. The sender can transmit frames only if it has sufficient credits—ensuring that the receiver has space to accept the incoming data. Periodically, the receiver sends back to the sender the credits for the frames that have left the buffer. CBFC supports up to 32 virtual channels (VCs) per port. Each VC can be configured as either lossless or best-effort, and this configuration can be changed dynamically.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Credit-based flow control (CBFC) is a newly introduced feature in the Switch Abstraction Interface (SAI) that enhances link-level flow control without impacting the existing SAI logical pipeline. The following SAI enhancements are done for CBFC:

- Three new objects:
    - **Virtual channel:** Contains per-VC attributes associated with CBFC sender and CBFC receiver functions on a port.
    - **CBFC credit profile:** Attributes for management of receiver credits (i.e., receiver's buffer) on a per-VC basis.
    - **CBFC credit pool:** Used to provide isolation of credits (i.e., receiver's buffer) between groups of VCs.
- New attributes for CBFC to the port object.
- New QoS map types for VC derivation from TC, queue, or packet's dot1p+DEI.
- New attributes to enable/disable enqueue and dequeue, which are needed for lossless VC initialization and deinitialization.

7.6.1.3 SAI Enhancements for CSIG

Congestion signaling (CSIG) is a simple and efficient network telemetry method that collects fine-grained signals that describe the network fabric's congestion state. The central insight underlying CSIG's design is that for many telemetry signals useful in congestion control and performance analysis, the result of a simple aggregation function computed over the congestion state at switches along a packet's path is of interest. For example, for several signal types, the congestion state is of interest only at a single switch along a path, where the switch of interest is the one where a signal's value is an extremum (maximum or minimum) among the signal's values at all switches along the path.

CSIG does not alter the forwarding behavior of a packet in the SAI logical pipeline. This means that the SAI logical pipeline does not require any changes for CSIG packets. SAI already provides an existing infrastructure that uses the telemetry and management (TAM) object. This TAM infrastructure currently supports in-band network telemetry, streaming telemetry, events, transport for reporting, and various reporting formats. The TAM infrastructure has been enhanced to support CSIG.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

8 UE Compliance

To assist implementors in efforts to comply to this UE specification, the UEC provides test collateral including profile matrices, compliance checklists, and a best practices document describing suggestions for establishing a testbed. The profile matrix and compliance checklist should be used by implementors to self-attest for compliance.

These documents are located at: https://ultraethernet.org/compliance/

The profile matrix states which features are mandatory, conditional, or optional for each Ultra Ethernet profile. UE defines three profiles: AI Base, AI Extended, and HPC. The profile matrix also specifies which items in the compliance checklist need to be tested if the feature or technology is implemented. Figure 8-1 shows an excerpt from the profile matrix.

	A	B	C	D	E	F	G	H	I	J	K
1	**MatrixID**	**Technology/Feature/Setting**	**Protocol/API**	**Value**	**Conditions**	**AI BASE**	**AI EXTENDED**	**HPC**	**ChecklistIDs**	**Notes**	
2	PHY	PHY Type		Bool		Mandatory	Mandatory	Mandatory	PHY-1.1		
3	PHY-1	Control Ordered Sets		Bool	[LL-LLR, LL-CBFC]	Conditional	Conditional	Conditional	[PHY-2.1, PHY-2.2, PHY-2.3, PHY-2.4, PHY-3		
4	PHY-2	FEC statistics for prediction of link quality		Bool		Optional	Optional	Optional	[PHY-5.1, PHY-5.2]		
5	LL-LLR	LLR		Bool		Optional	Optional	Optional	[LL-LLR-1.1, LL-LLR-2.1, LL-LLR-3.1, LL-LLR-		
6	LL-CBFC	CBFC		Bool		Optional	Optional	Optional	[LL-CBFC-1.1, LL-CBFC-1.2, LL-CBFC-1.3, LL		
7	LL-CBFC-1	CBFC Receiver with a non-power-of-2 Cell Size		Bool	Receiver Cell Size is not a power of 2	Conditional	Conditional	Conditional	LL-CBFC-6.1		
8	LL-CBFC-2	CBFC-LLR Interactions		Bool	LL-LLR	Conditional	Conditional	Conditional	[LL-CBFC-7.1, LL-CBFC-7.2]		
9	LL-CBFC-3	CBFC-PFC Interactions		Bool	PFC and CBFC are enabled simultaneously	Conditional	Conditional	Conditional	[LL-CBFC-8.1, LL-CBFC-8.2]		
10	LL-NEG	Link Layer Negotiation		Bool	[LL-LLR, LL-CBFC]	Conditional	Conditional	Conditional	[LL-NEG-1.1, LL-NEG-2.1, LL-NEG-2.2, LL-NE		
11	LL-NEG-LLR	Link Layer Negotiation Link Layer Retry		Bool	LL-LLR	Conditional	Conditional	Conditional	[LL-NEG-LLR-1.1, LL-NEG-LLR-1.2]		
12	LL-NEG-CBFC	Link Layer Negotiation Credit Based Flow Control		Bool	LL-CBFC	Conditional	Conditional	Conditional	[LL-NEG-CBFC-1.1, LL-NEG-CBFC-1.2, LL-NI		

Figure 8-1 - Example of Profile Matrix

The compliance checklist contains a list of the items to test for each specification. The checklist includes the TestID, TestName, Criteria, and the reference to the specification. Figure 8-2 shows an excerpt from the compliance checklist, each row identifying a testable item.

In order to attest compliance to the specification, implementers should run tests to confirm the behavior that is targeted by the checklist items, either by themselves or with the help of a third party.

	A	B	C	D	E	
1	**Priority**	**ChecklistID**	**TestName**	**Criteria**	**SpecReference**	**MatrixIDs**
2		PHY-1.1	PHY Type	The PHY type must be listed in section 6.1.2	UEC 1.0 section 6.1.2	PHY
3		PHY-2.1	Control Ordered Set Format (MII)	The CtlOS have a Lane0 control character of 0x5C and D4[7:4] set to 0x6.	UEC 1.0 section 6.2.2	PHY-1
4		PHY-2.2	Control Ordered Set Format (PCS)	The CtlOS have a block type of 0x4B and bits 10 to 65 taken from D1 through D7.	UEC 1.0 section 6.2.3	PHY-1
5		PHY-4.1	PCS Decoder	The PCS Decoder is described in Table 6-5	UEC 1.0 section 6.2.3	PHY-1
6		PHY-3.1	PCS Encoder	The PCS Encoder is described in Table 6-6	UEC 1.0 section 6.2.3	PHY-1
7		PHY-2.3	CtlOS Indication to MAC	RS sends an indication to the MAC upon reception of a CtlOS	UEC 1.0 section 6.2.4	PHY-1
8		PHY-3.2	Transmit CtlOS during MAC frames	Transmission of CtlOS does not corrupt 802.3 packet transfers	UEC 1.0 section 6.2.4.1	PHY-1

Figure 8-2 - Example of Compliance Checklist

Implementers making a compliance statement should provide the information listed in the Compliance Statement as follows, to indicate that they have tested and are compliant to the profile and features they are claiming support for.

8.1 Compliance Statement

A vendor can claim UE support by performing a self-attestation and declaring the supported features and profiles in their product literature.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

8.1.1 UE Support Requirements

Implementers SHALL call out supported features and associated profiles [e.g. LL-LLR, AI-BASE].
Implementers SHALL execute the checklist tests, corresponding to the profile line items [e.g. LL-LLR-1.1].

8.1.2 Declaration Format

The declaration format is:
<Complies to UE Specifications: [spec ver # - section], [profile + optional/conditional], tested against [checklist doc ver #]>

It is RECOMMENDED that the declaration statement lists the optional/conditional profile features that were tested.

Declaration example: “Complies to UE Specifications: UE Specification v1.0 – Section 5.1, AI-BASE, CT LL-PHY Specifications 1.0”.

Non-mandatory features can be enabled during compliance tests, but it must be possible to disable non-mandatory features if they can cause interoperability issues with other implementations using the same UE profile without the optional features.

8.1.3 Compliance verses Support Terminology

Support is synonymous to Comply. A statement of compliance with UE Specifications should be accompanied by a declaration stating what was tested.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Release Notes

Document: Ultra Ethernet™ Specification
Version: 1.1
Release Date: 10/12/2026
Previous Version: 1.0.3 (Released 07/16/2026)

Overview

This release includes significant new functionality and normative requirements. It also includes a set of editorial corrections and clarifications to improve the readability and accuracy of the Ultra Ethernet™ Specification v1.0.3, published on July 16, 2026.

Summary of Changes

Section	Type	Description of Change
2.3	Enhancement	UE verbs interface mapping.
2.4	Enhancement	In-Network Collectives (INC).
3.3 thru 3.5, 3.7.11	Enhancement	The Unified Local Profile enhancements for scale-up networking to SES, PDS, and TSS, including the new reliable ordered delivery mode for local networks (RODL).
3.4.2.3 Figure 3-8	Enhancement	Adds an optional hardware feature allowing a FEP to allocate Memory Regions (MRs) scoped to a specific PIDonFEP instance.
3.4, 3.5.8.2 Table 3-1, Table 3-3, Table 3-20 thru Table 3-23, Table 3-68,	Enhancement	Adds a Sync Group mechanism to group write transactions and trigger a completion or atomic operation once all writes in the group have been posted.
3.6.18	Enhancement	Programmable congestion management (PCM)
3.6, 4.2, 5.3	Enhancement	Congestion Signaling (CSIG), enhancements to CMS to utilize CSIG and enhancements to LLDP negotiations for enabling CSIG.
5.4	Enhancement	Optimized headers and Packet Rate Improvement using the Unified Forwarding Header (UFH).
5.5	Enhancement	Ethernet idle reduction.
7	Enhancement	Overview of UE management and reference to standalone documentation.
Table 5-27	Correction	Correct the range and granularity of the CF_max_timers to be 16 KB with granularity between 1 byte and 16384 bytes.
2.1	Clarification	Clarify new software interfaces, module relationships and open source projects in the UE Software Overview.
Table 2-13	Clarification	Clarify bit location range of VENDOR_SPECIFIC field in the memory region key format.
3.5.8.2.1	Clarification	Clarify the process of PDC establishment with encryption.
3.5.5, 3.5.8, 3.5.10, 3.5.11	Clarification	PDS conformance checklist statement reference alignment.
Table 5-9	Clarification	Clarify minimum values of data_age_timer_max.
Various	Editorial	Correct minor punctuation, spelling, figure labeling and formatting errors.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.

Impact Statement

This revision defines new optional enhancement updates to protocol functionality and associated requirements while preserving full backward compatibility with previous versions. It introduces a new Unified Local profile with mandatory requirements. Clarifications and editorial updates to existing functionality are non-substantive and do not alter protocol behavior, compliance, or interoperability. Correction updates are normative and required for conforming implementations.

Document Version History

Version	Date	Description
1.0	06/11/2025	Initial version.
1.0.1	09/05/2025	Editorial clarifications and correction to the RCCC source algorithm.
1.0.2	01/21/2026	Correction to CMS congestion control algorithms and other editorial clarifications.
1.0.3	07/16/2026	A number of editorial clarifications, addition of 200 Gb/s per lane signaling, a new Negotiation CP type, corrections to UE PHY CtlOS corruption protection, UE Transport handling of retransmission when pds.flags.syn=1 on PDC close, specifying MP_Range less than 128 and corrections to UE Link Layer to resolve CBFC/LLR race conditions.
1.1	10/12/2026	Major release with new functionality.

Copyright © 2026 Ultra Ethernet Consortium™. All rights reserved.