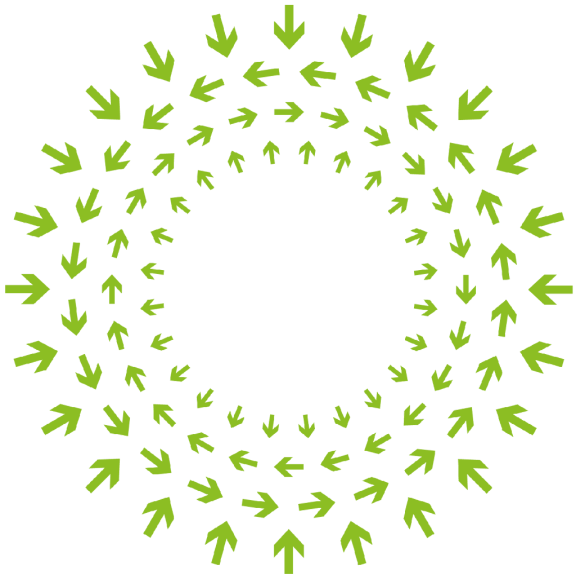


OPEN

Compute Project

Falcon Networking Transport Specification

Revision 1.2.0

Effective: 11/8/2026

Authors (alphabetical): Prashant Chandra, Nandita Dukkipati, Praveen Kumar, Naveen Kumar Sharma, Yuliang Li, Abhiram Ravi, Arjun Singhvi, Hassan Wassel, Google

Table of Contents

1. License **7**
2. Compliance with OCP Tenets **7**
2.1 Openness 7
2.2 Efficiency 7
2.3 Impact 8
2.4 Scale 8
2.5 Sustainability 8
3. Change Log **9**
4. Scope **10**
5. Overview **10**
6. Protocol Architecture **11**
6.1 Protocol Layers 11
6.1.1 Transaction Sublayer 12
6.1.1.1 Push Transaction 12
6.1.1.2 Pull Transaction 13
6.1.2 Packet Delivery Sublayer 14
6.1.3 Falcon Peer Entities 15
6.2 Connections 15
6.3 Congestion Control 16
6.4 Multipath Load-Balancing 17
6.5 Congestion Signaling (CSIG) 18
6.5.1 Protocol summary 18
6.5.2 Realization in Falcon 18
6.6 Sender-Driven Acknowledgement Packet Generation Controls 19
6.7 Error Handling 20
6.8 Security Model 20
6.9 Example Packet Flows 21
6.9.1 RDMA Read Flow 21
6.9.2 RDMA Write Flow 23
6.9.3 Push Transaction Receiver Not Ready (RNR) NACK Flow 24
6.9.4 Pull Transaction RNR NACK Flow 26
6.9.5 Push Transaction Complete In Error (CIE) NACK Flow 27
6.9.6 Early Retransmission using Extended ACK (EACK) 29
7. Packet Formats **30**
7.1 Falcon Packet 30

7.1.1 Transport Mode 31
7.1.2 Tunnel Mode 31
7.2 Falcon Base Header 32
7.3 Pull Request Packet 34
7.4 Pull Data Packet 35
7.5 Push Data Packet 36
7.6 Resync Packet 37
7.7 Acknowledgement (ACK) Packet 38
7.7.1 Base ACK (BACK) Packet 39
7.7.2 Extended ACK (EACK) Packet 41
7.8 Negative Acknowledgement (NACK) Packet 43
8. Transaction Sublayer 47
8.1 Transaction Types 47
8.2 Resource Management 48
8.2.1 Avoiding Deadlocks 48
8.2.1.1 Unconstrained Resource (UR) Implementations 48
8.2.1.2 Constrained Resource (CR) Implementations 49
8.2.2 Resource Pools 50
8.2.3 ULP Resource Management 53
8.2.4 Network Resource Management 54
8.2.5 Connection-level Resource Carving 56
8.3 Connection Scheduler 57
8.4 Initiator Implementation 58
8.4.1 Initiator State 58
8.4.2 Transaction Ordering 59
8.4.3 Transaction Processing 60
8.4.3.1 Pull Request 60
8.4.3.2 Pull Data 60
8.4.3.3 Push Data 61
8.4.3.4 RNR NACK 61
8.4.3.5 CIE NACK 62
8.5 Target Implementation 62
8.5.1 Target State 62
8.5.2 Transaction Ordering 63
8.5.3 Transaction Processing 63
8.5.3.1 Pull Request 63
8.5.3.2 Pull Data 64
8.5.3.3 Push Data 64

8.5.3.4 RNR NACK 65
8.5.3.5 CIE NACK 66
9. Packet Delivery Sublayer 66
9.1 Transmitter Implementation 68
9.1.1 Transmitter State 68
9.1.2 Congestion Control Tx Gating Function 72
9.1.3 Packet Transmission 73
9.1.3.1 Ack Req (AR) generation 73
9.1.3.2 Path Association 73
9.1.4 Early Retransmission 75
9.1.4.1 EACK Algorithm 75
9.1.4.2 Recent Acknowledgement 76
9.1.4.3 Tail Loss Probe 77
9.1.5 Timeout Based Retransmission 78
9.1.6 ACK Generation 78
9.1.7 NACK Generation 79
9.1.8 Resync Generation 80
9.2 Receiver Implementation 80
9.2.1 Receiver State 80
9.2.2 Packet Acceptance Checks 83
9.2.2.1 Packet Integrity Check 83
9.2.2.2 Replay Protection Check 83
9.2.2.3 SPI Check 84
9.2.2.4 Sliding Window Check 85
9.2.3 ACK Processing 86
9.2.4 NACK Processing 87
9.2.5 Resync Processing 88
10. Congestion Control 89
10.1 Delay Measurement 90
10.2 NIC Buffer Occupancy Measurement 92
10.3 Swift Congestion Control Algorithm 92
10.3.1 Process Ack and Nack that is not NIC-resource-exhaustion 92
10.3.2 On a Retransmit 94
10.3.3 On NACK that indicates NIC resource exhaustion 94
10.3.4 Common Methods 95
10.3.5 Congestion Windows 96
10.3.6 Window Guarding 96
10.4 PLB: Protective Load Balancing 98

10.5 Parameters and Variables 99
10.5.1 Fabric congestion window settings 99
10.5.2 NIC congestion window settings 100
10.5.3 PLB setting 100
10.5.4 Common parameters 101
10.5.5 Measurements 101
10.5.6 State 102
10.5.7 Outputs 102
10.6 RUE Architecture 103
10.6.1 Triggering RUE Event Policy 104
10.6.2 RUE Interface 106
10.6.2.1 CC Event Queue 106
10.6.2.2 CC Result Queue 109
11. Error Handling and Resource Reclaim 112
12. Interoperability 115
13. Acknowledgments 117

1. License

THE UPDATED DEFAULT CONTRIBUTOR LICENSE AGREEMENT (CLA) IS OWFa 0.9.
PLEASE VERIFY THE CORRECT CLA/FSA IS USED AND EXECUTED FOR THIS CONTRIBUTION.

1.1. Open Web Foundation (OWF) CLA
Contributions to this Specification are made under the terms and conditions set forth in **Modified Open Web Foundation Agreement 0.9 (OWFa 0.9)**. (As of October 16, 2024) ("Contribution License") by:

Google

Usage of this Specification is governed by the terms and conditions set forth in **Modified OWFa 0.9 Final Specification Agreement (FSA)** (As of October 16, 2024) (**"Specification License").**

You can review the applicable Specification License(s) referenced above by the contributors to this Specification on the OCP website at
https://www.opencompute.org/contributions/templatesagreements

2. Compliance with OCP Tenets

2.1 Openness

The specification complies with the tenet of Openness by empowering the Community with Google's production learnings to help modernize Ethernet. This includes leveraging production-proven technologies at scale including Carousel, Snap, Swift, Protective Load Balancing, and Congestion Signaling (CSIG) that have been openly published previously.

2.2 Efficiency

The specification complies with the tenet of Efficiency. Falcon achieves high performance by combining three key insights that achieve low latency in high-bandwidth, yet lossy, standard Ethernet data center networks. Fine-grained hardware-assisted round-trip time (RTT) measurements with flexible, per-flow hardware-enforced traffic shaping, and fast and accurate packet retransmissions, are combined with multipath-capable and PSP-encrypted Falcon connections. On top of this foundation, Falcon has been designed from the ground up as a multi-protocol transport capable of supporting Upper Layer Protocols (ULPs) with widely

varying performance requirements and application semantics. The ULP mapping layer not only provides out-of-the-box compatibility with Infiniband™ Verbs RDMA and NVMe® ULPs, but also includes additional innovations critical for warehouse-scale applications such as flexible ordering semantics and graceful error handling. Last but not least, the hardware and software are co-designed to work together to help achieve the desired attributes of high message rate, low latency, and high bandwidth, while maintaining flexibility for programmability and continued innovation.

2.3 Impact

The specification complies with the tenet of Impact by introducing a new technology that helps the industry modernize Ethernet. Falcon provides a helpful solution to address demanding workloads that have high burst bandwidth, high Operations per second, and low latency in massive scale AI/ML training, High Performance Computing, and real-time analytics.

2.4 Scale

The specification complies with the tenet of Scale by being designed from the ground up to deliver high bandwidth and low latency in high-bandwidth, yet lossy, Ethernet data center networks. Additionally, Falcon is composed of production-proven technologies delivered at scale including Carousel, Snap, Swift, Protective Load Balancing, and Congestion Signaling (CSIG).

2.5 Sustainability

The specification complies with the tenet of Sustainability by delivering an efficient Ethernet transport technology that minimizes retransmissions and other wasted effort and energy within an Ethernet network. Additionally, the technology allows a wide range of high performance workloads to be run on standard Ethernet networks.

3. Change Log

Date	Version #	Author	Description
14 FEB 2024	0.9	Prashant Chandra Nandita Dukkipati	Defines the Falcon transport protocol
4 APRIL 2024	1.0	Arjun Singhvi	Changes for clarity / accuracy
22 APRIL 2026	1.1	Nandita Dukkipati, Praveen Kumar, Abhiram Ravi, Naveen Kumar Sharma, Arjun Singhvi, Hassan Wassel	Multipathing, Connection-level resource carving, Congestion Signaling (CSIG), Acknowledgment Request (AR), RUE Queue Format enhancements.
15 OCT 2026	1.2.0	Nandita Dukkipati, Naveen Kumar Sharma, Arjun Singhvi, Hassan Wassel	Recent Acknowledgement & Tail Loss Probe (RACK-TLP), EACK-Request bit, Rx Window Expansion, Max-Open-Window Scheduler for Multipathing, On Demand Pull Response Resource Management.

4. Scope

This specification describes the Falcon transport protocol that provides reliable transport for RDMA Operations and NVMe commands.

Not in scope are:

- Details of changes in RDMA and NVMe layers in support of Falcon.
- Details of applications' use of Falcon.
- Details of implementing Falcon in software stacks or hardware NICs.

5. Overview

This specification describes the Falcon protocol that provides reliable transport for RDMA Operations and NVMe commands. Falcon is a connection-oriented, request-response transport protocol that provides end-to-end reliable transfer and per-connection security to upper layer protocols (ULPs) such as RDMA and NVMe. A Falcon connection can either be ordered or unordered. Additionally, a Falcon connection can either be single-pathed or multipathed. Multipathing refers to using multiple network paths simultaneously. Falcon includes a programmable congestion control engine that can be used to implement state-of-the-art congestion control algorithms. Falcon authenticates and encrypts all ULP data on a per-connection basis using the PSP Security Protocol (PSP) or the IPSEC ESP protocol.

When establishing a connection, Falcon software in the control plane negotiates what each endpoint can do. To ensure backward compatibility, endpoints using Specification 1.2.0 must be able to operate with endpoints using only Specification 1.0 or Specification 1.1 features.

The following sections describe key architectural concepts of Falcon in further detail.

6. Protocol Architecture

6.1 Protocol Layers

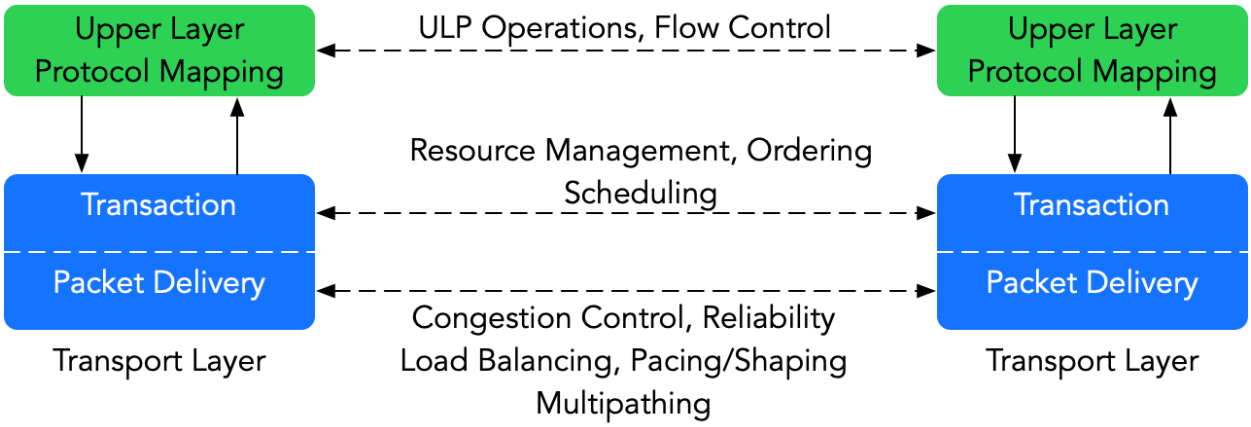


Figure 1: Protocol Layering of Falcon

The figure above shows the protocol layering of Falcon. Falcon itself is composed of two sublayers: transaction and packet delivery. The transaction sublayer primarily deals with ULP transactions and is responsible for resource management and transaction ordering. The packet delivery sublayer primarily deals with network packets and is responsible for reliable delivery, congestion control and multipath load balancing. Falcon does not expose a HW/SW interface of its own. ULPs are responsible for implementing the HW/SW interface, processing of Operations, completion notification and end-to-end flow control.

A ULP Operation (Op) can map to one or more Falcon transactions. A Falcon Transaction is defined by a request and a response. Falcon reliably completes transactions by sending and receiving one or more packets over the network. Falcon notifies the ULP upon finishing the transaction via a response or a completion. Falcon transactions can either be push or pull requests. ULP Ops are mapped to push or pull requests. Falcon's push and pull requests provide basic primitives that enable flexible mapping of different ULP Ops to Falcon transactions. For e.g, RDMA Read and Atomic Operations are mapped to pull requests and RDMA Send and Write operations are mapped to push requests. A ULP Op may also map to a set of Falcon primitives. An example of this is the mapping of NVMe Read and Write Ops to Falcon transactions. The figure below shows the relationship amongst ULP Ops, Falcon transactions and packets.

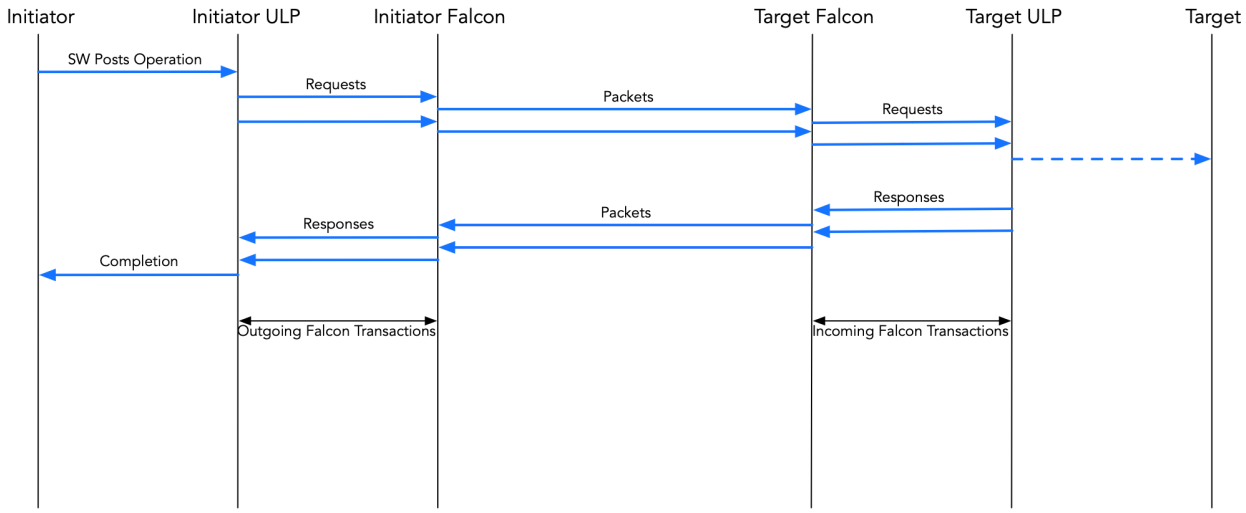


Figure 2: Relationship amongst ULP Ops, Falcon Transactions and Packets

6.1.1 Transaction Sublayer

The transaction sublayer presents a transaction interface to the ULP. A Falcon transaction consists of a request originating from the ULP to Falcon and a response delivered back to ULP from Falcon, signifying the end of the transaction. A ULP can generate two types of transactions: pull and push. All ULP operations must be mapped to push or pull transactions. For example, RDMA read and atomic opcodes are mapped to pull transactions and RDMA send and write opcodes are mapped to push transactions. Each transaction to Falcon has a Request Sequence Number (RSN) that is used for tracking the transaction state and ordering.

6.1.1.1 Push Transaction

A push transaction, e.g, a Write operation, consists of the initiator ULP sending a Push Request to Falcon, which then sends a Push Data packet to the target Falcon. The target Falcon then forwards the Push Data packet to target ULP which acknowledges the data. Finally, the target Falcon sends this acknowledgement (ACK) back to the initiator Falcon via an ack packet or a piggybacked ack, and delivers the ack to the initiator ULP via a completion. Reliability is provided end-to-end at the ULP layer by the virtue that the Falcon packet delivery layer at the target acknowledges Push Data packets only when Target ULP acknowledges the Push Request.

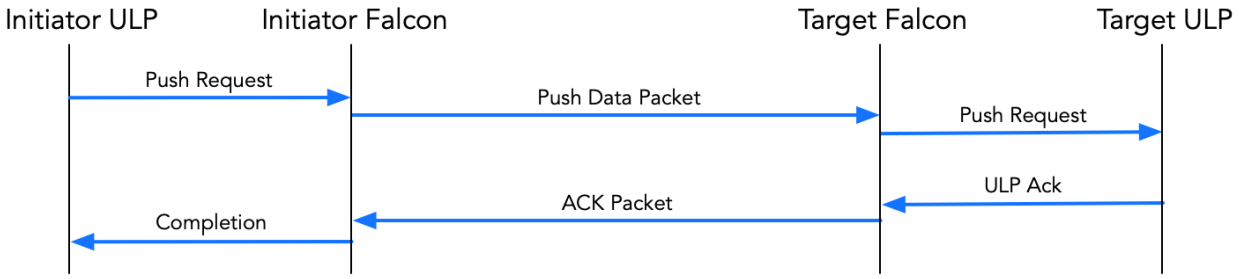


Figure 3: Push Transaction Flow

6.1.1.2 Pull Transaction

A Pull transaction, e.g, Read operation, consists of the initiator ULP sending a Pull Request to Falcon, which transmits a Pull Request packet to the target Falcon. The target Falcon forwards the request to the target ULP, which responds with the Pull Response data. The target Falcon transmits this response via a Pull Data packet back to the initiator Falcon, which finishes the transaction by sending the Pull Response back to the initiator ULP.

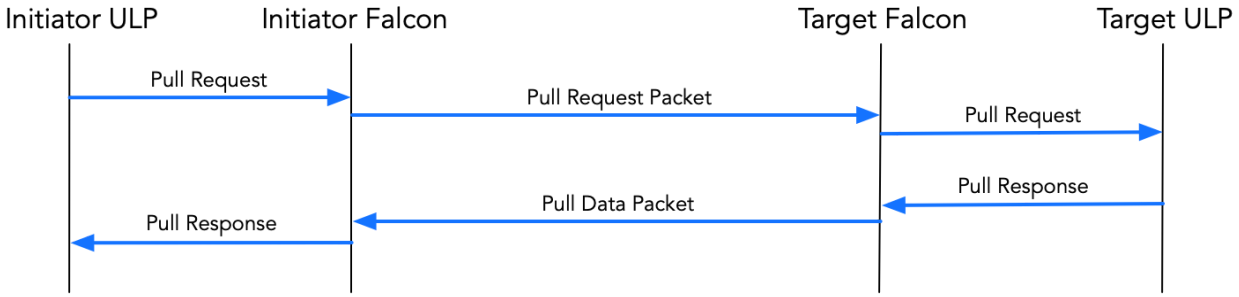


Figure 4: Pull Transaction Flow

6.1.2 Packet Delivery Sublayer

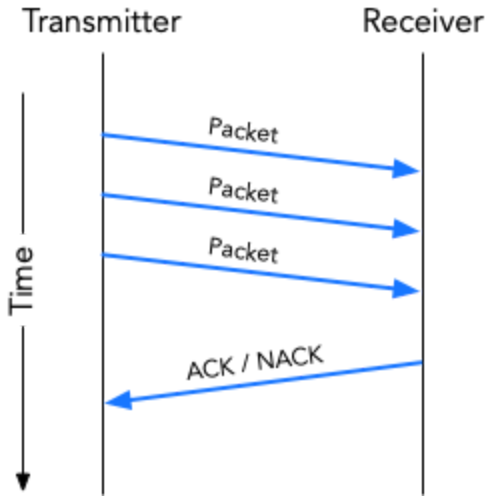


Figure 5: Packet and ACK flow between Falcon Transmitter and Receiver

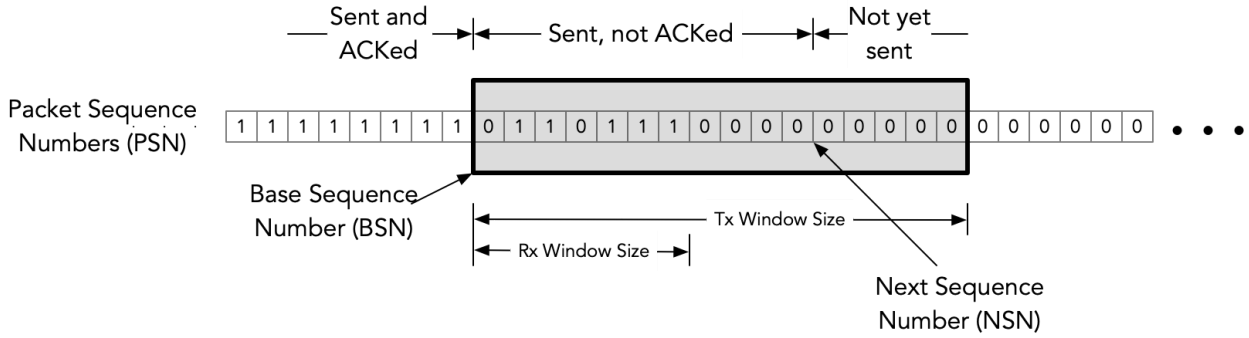


Figure 6: Sliding Window Protocol Concepts

The three main functionalities of the packet delivery layer are reliable packet delivery,congestion control, and multipath load-balancing from a Falcon transmitter to a Falcon receiver; their detailed mechanisms are elucidated later in this specification.

The packet delivery sublayer uses two sliding windows to separate out requests and data. The request sliding window is responsible for reliable delivery of Pull Requests, and the data sliding window is responsible for reliable delivery of Pull Data and Push Data. The two sliding windows are needed to avoid protocol deadlocks as detailed in the Deadlock Avoidance section.

The above figure illustrates the key concepts of the sliding window protocol that operates independently for most parts across the two windows. The transmitter assigns a Packet

Sequence Number (PSN) to each packet in the range of 0 to 2^{32} -1. The PSN is incremented by 1 for each packet transmitted and is included in the Falcon base header.

At the receiver, each packet has two states: received and acknowledged. Packet acknowledgment is tied to ULP acknowledgment and thus a received packet may not yet be acknowledged. The receiver conveys the received and acknowledged states of packets to the sender by sending an ACK packet. ACKs sent by the receiver contain an indication of the sequence numbers that have been successfully received. The receiver is permitted to use cumulative ACKs and can also piggyback ACKs onto packets flowing in the opposite direction. In the event of packet loss or reordering, the receiver sends Extended ACK (EACK) which includes bitmaps of received/acknowledged PSNs, such that the sender can detect loss or reordering and retransmit lost packets.

At all times, the transmitter maintains a set of sequence numbers that the transmitter is permitted to send. These sequence numbers define a sliding window (shown by the gray box in figure above). The left edge of the window is the Base Sequence Number (BSN) and represents the oldest packet sent that is yet to be acknowledged by the receiver. The right edge of the window is defined by the BSN + Tx Window Size. The transmitter is permitted to send a packet only if the PSN assigned to that packet falls within the transmitter's sliding window. The Next Sequence Number (NSN) represents the next PSN value that must be assigned to the packet that is to be transmitted.

6.1.3 Falcon Peer Entities

Initiator and target are transaction layer concepts, whereas, transmitter and receiver are packet delivery layer concepts. An initiator (and target) can be both a transmitter and a receiver. For example, the initiator of a Pull Request is the transmitter of the Pull Request packet, but the receiver of the Pull Data packet; the target is the receiver of the Pull Request packet, and the transmitter of the Pull Data packet.

6.2 Connections

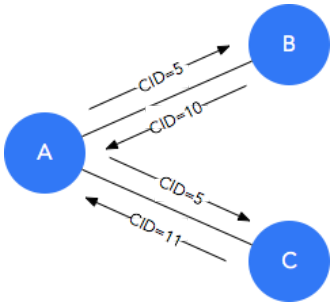


Figure 7: Example of CID Allocation

A connection is an end-to-end construct that describes a bidirectional communication channel between two Falcon peers. A packet is the basic unit of communication across a connection. A packet can be up to MTU size in length. A connection is identified by a pair of Connection IDs (CID), one in each direction. CIDs are allocated by the receiver during the connection setup process and have no global significance. The CID assigned by the receiver is called the Destination CID and is carried in the Falcon header.

The figure above shows an example of allocation of CIDs. The connection between A and B uses CID value 5 from A to B and CID value 10 from B to A. The connection between A and C uses CID value 5 from A to C and CID value 11 from C to A. The CID values need not be unique at the transmitter end as illustrated in this example where A uses the same CID value of 5 to communicate with both B and C. To disambiguate between the two connections, the transmitter side uses a locally unique identifier called the Source CID. The Source CID is not carried in the Falcon header.

6.3 Congestion Control

For single path connections, Falcon uses per-connection congestion control. The Falcon datapath hardware is responsible for measuring congestion signals and enforcing the computed congestion window and rate on a per-connection basis. The congestion control algorithm itself is implemented in a Rate Update Engine (RUE) that is separate from the main Falcon datapath. In case of a multipath Falcon connection, congestion control is done on a per-path basis.

Falcon triggers RUE operation on certain events such as ACK/NACK reception, packet retransmissions, etc. RUE computes congestion control output (congestion window aka cwnd, and rate) in response to these events and returns them to Falcon. fcwnd tracks fabric congestion, while ncwnd tracks end-point or NIC congestion. The architectural separation between Falcon and RUE permits the implementation of a class of congestion control algorithms such as Swift described below, without changing the main Falcon datapath.

Swift is a well-known delay-based congestion control for datacenters delivering low tail latency with near-zero packet drops under high bandwidths. The combination of the following makes Swift a high-performant congestion control algorithm:

- Harnesses per-packet NIC hardware-timestamps and uses them effectively for congestion window and rate computations.
- Rapidly reactive algorithms handle massive-incasts and flow collisions in the network.
- Disambiguates and responds aptly to both fabric as well as end-host/NIC congestion by decoupling cwnd (or rate) computations for fabric and end-host components.

6.4 Multipath Load-Balancing

Network load balancing is critical for minimizing congestion hotspots in the fabric. For multipath connections, Falcon distributes traffic for a single connection across multiple network paths simultaneously to improve fabric utilization and tail latency. This is achieved by assigning a set of flow labels to the connection rather than just one. Network switches perform ECMP hashing over packet header fields(including the IPv6 Flow Label; varying the flow label across packets of the same connection directs packets onto distinct network paths. Thus, a multipath connection consists of multiple flows (up to four in this specification), where each flow corresponds to a different network path. Throughout this specification, *per-flow* refers to *path-specific* handling. The Falcon datapath uses a scheduler to select a flow label for each outgoing packet, thereby directing it onto a specific path.

The congestion control logic for multipathing is handled by RUE on a per-flow basis as described above. The RUE computes a cwnd for each flow (fcwnd is on a per-flow basis, while ncwnd remains per-connection). and the Packet Delivery Layer aggregates these per-flow cwnds into a combined cwnd for the connection. To support this, RUE events and responses contain additional per-flow information (such as flow cwnd and flow labels). While multipathing distributes a single connection's traffic across several paths concurrently, Falcon also uses Protective Load Balancing (PLB) to reactively change a network path in response to sustained congestion on the path. PLB leverages congestion signals from Swift to determine when to reroute traffic by changing the flow label.

In prior versions of this specification (Falcon v1.1), multipath scheduling in Falcon's datapath operated in an *open-loop* manner using a Weighted Round-Robin (WRR) scheme. In open-loop scheduling, the RUE computed per-path fabric congestion windows ($fcwnd_i$) and converted them into coarse, static flow weights ($W_i \propto fcwnd_i / \sum fcwnd$). The datapath transmitted packets across flows strictly in proportion to these static weights, regardless of which paths were actively returning acknowledgments. Falcon Specification 1.2.0 replaces the open-loop WRR scheduler with a *Max-Open-Window Scheduler*. This scheduler operates as a closed-loop hardware mechanism that couples packet departures directly to path-specific acknowledgment feedback at line rate:

1) Per-Path Credit Tracking: For each path, the hardware datapath tracks the in-flight backlog ($outstanding_packets_i$) and computes the instantaneous available capacity as: $open_fcwnd_i = flow_fcwnd_i - outstanding_packets_i$. Transmitting a packet immediately consumes credit on that path. Only when an ACK (or EACK) confirming receipt of a packet on

that specific path returns to the sender is that path's in-flight counter decremented, selectively reopening capacity on that path.
2) Max-Capacity Prioritization: Outgoing packets are dispatched to the path possessing the maximum open window ($max(open_fcwnd)$), directing traffic to paths with the highest available capacity. If a path experiences congestion or increased queuing delay, ACKs on that path slow down, leaving $outstanding_packets_i$ high and causing $open_fcwnd_i$ to drop to one or below, the scheduler immediately blocks transmissions to that congested path without waiting for software RUE recalculations, instantaneously shifting traffic to uncongested paths.

Implementations may optionally retain the legacy open-loop WRR scheduler behind a hardware configuration switch, but the closed-loop scheduler is the normative plan of record.

6.5 Congestion Signaling (CSIG)

6.5.1 Protocol summary

CSIG provides precise, multi-bit bottleneck congestion signals to end hosts via a transport-neutral design. It utilizes a fixed-size L2 Ethernet header, using line-rate compare-and-replace operations to record real-time congestion metrics in every data packet.

CSIG protocol consists of two primary components:

- **CSIG Tag:** Positioned as the final tag in the Layer 2 header, where switches record real-time bottleneck metrics. It is available in two variants carrying the congestion signal and locator information: a **Compact 4-byte tag** or a **Wide 8-byte tag**. The placement ensures compatibility with end-to-end encryption and L3+ tunneling. UEC Specification v1.1 outlines the formats for both compact and wide CSIG L2 tags.

- **CSIG Reflection:** The mechanism by which signals are returned to the sender via L4 transport protocols, such as Falcon.

The decoupling of switch-side signaling from end-point side L4 reflection, the CSIG protocol allows transports such as Falcon to evolve its control loop responses independently of switch hardware. Falcon Specification 1.2.0 supports the CSIG Compact 4-byte tag, and its reflection header is described in Section 7.7 and Section 7.8.

6.5.2 Realization in Falcon

Falcon realizes CSIG using a division of responsibilities between the RUE and the HW datapath, operating entirely within the Falcon layer and independent of upper-layer protocols. RUE serves as the central control plane for CSIG: it performs signal selection, and also processes reflected

CSIG signals in the ACKs via an enhanced RUE event interface to drive congestion control and telemetry.

At the sender, the Packet Delivery layer constructs the CSIG tag (4-byte Compact format). On the receiving end, the ACK handling logic (including coalescing) maintains a dedicated state to store the latest received L2 CSIG tag. To support Multipath Load Balancing, this state is maintained on a per-flow basis, allowing the hardware to accurately relay the collected signals back to the sender via any subsequent ACK or NACK events.

To ensure interoperability with prior Falcon versions, CSIG capability is first negotiated between endpoints using standard Falcon negotiation. Note that Falcon assumes the support for Compact CSIG recognition within the underlying network fabric and does not explicitly negotiate for switch support. CSIG should be configured and used only if the fabric switches support CSIG end-to-end on packet paths.

6.6 Sender-Driven Acknowledgement Packet Generation Controls

ULP Op Boundary ACKs: An initiator Falcon can request an explicit and immediate ACK from the target Falcon for the last packet of a large multi-MTU operation issued by the initiator ULP. This prevents ACK coalescing delays at the target and improves op latency at low offered loads. The ULP must indicate to Falcon that a given transaction is the last (or only) packet of an operation, so that Falcon can set the ack request bit in the header appropriately.

RUE-driven Ack Request: Falcon supports fine-grained control over the number of explicit ACK packets generated on a per-connection basis. This is useful for modulating the number of ACKs for connections operating in different regimes. For e.g., a slow connection with a small congestion window prefers an ACK for every packet, whereas a fast connection with a larger congestion window can operate with an ACK every 8 or 16 packets. Falcon supports RUE-driven control over explicit ACK generation frequency via the **ar_rate** field, which encodes a power of 2 (N) to specify that an explicit ACK be requested every 2^N packets.

ACK Rate Limiter: The two enhancements described above can result in an excessive number of ACK packets being generated over the network if not controlled appropriately. For e.g., if all packets are less than an MTU, the number of ACKs will be equal to the number of packets in the network, which is suboptimal. Similarly, if RUE miscomputes the 'ar_rate' for a fast connection, it can lead to a large number of ACKs as well. Therefore, Falcon supports rate-limiting the total number of ACKs generated by a Falcon endpoint. This is configured at a global level after considering deployment tradeoffs.

6.7 Error Handling

Falcon supports error handling semantics of the ULP. In particular, Falcon supports the ULP concept of a "receiver not ready (RNR)" indication. The RNR indication allows the target ULP to specify a time after which the transaction must be retried. Falcon implements the RNR retry transparent to ULP. Push transactions are retried from the initiator side. Pull transactions are retried from the target side. RNR NACK processing is described in more detail in later sections.

In addition, Falcon also supports a more graceful recovery with "complete in error and continue (CIE)" semantics. The CIE semantics allow for a more graceful handling of various errors (such as memory protection errors in the RDMA ULP) without requiring such errors to be treated as fatal within the ULP. The target ULP can reply back with a CIE indication to a push or a pull transaction. For a pull transaction, the ULP must return a zero-length pull response with the CIE error code. Falcon processes the zero-length pull response in the same way as any other pull response. For a push transaction, the ULP must return an explicit CIE indication with an error code. Falcon will transmit a CIE NACK packet with the ULP error code from the target to the initiator. The initiator will complete the transaction in error and return the CIE error code to the initiator ULP.

Although RNR NACK and CIE NACK are unreliable packets, Falcon has a mechanism to recover from lost NACKs.

Within the Falcon block itself, error handling is implemented entirely in hardware. The goal of the error handling is to contain the impact scope to only the connection that is experiencing the error and to not impact the performance of other connections. Protocol errors such as packet losses are handled using packet timeouts and retransmissions. Repeated retransmission failures beyond a configurable threshold are signaled to software so that corrective action can be taken, such as re-establishing the connection.

All resources allocated to a transaction within Falcon can be reclaimed after a configurable timeout. All resources within Falcon can be considered "soft state" in the sense that resource cleanup happens after timeouts on both the initiator and target sides.

6.8 Security Model

Falcon employs the PSP Security Protocol (PSP) or the IP-SEC ESP protocol for authentication and encryption of all data transferred over a connection. Each connection is associated with two Security Associations (SA) one in each direction. Multiple connections can share the same SA. On transmit, the SA index into the Security Association Database (SADB) is obtained from the connection state and is provided to the encryption hardware. On receive, the PSP decryption hardware derives the SA data (including the decryption key) using only the data in

the received packet, plus a secret kept internally to the NIC. Received packets must also pass the PSP SPI check before they are accepted. In the case of IP-SEC ESP, there is a SADB lookup in the receive direction to obtain the decryption key and other SA parameters.

This use of encryption in transit is orthogonal to additional security mechanisms employed by the upper layer protocol. For example, the NVMe upper layer protocol may employ encryption at rest (using AES-XTS), IB Verbs RDMA upper layer protocol uses protection domains and per memory region keys on top of the transport level PSP encryption.

Falcon implements replay protection similar to TCP PAWS to guard against packet sequence number rollover. Received packets that fail the replay protection check are dropped and do not generate ACKs or NACKs. The replay protection scheme uses the timestamp carried in the IV field of the PSP/ESP header. Falcon maintains the most recent timestamp received on a per-connection basis. A packet is accepted if its timestamp is within a time window of the most recent timestamp for that connection. This time window permits Falcon to accept packets that are reordered by the fabric.

6.9 Example Packet Flows

The following sections describe the transaction flows between a Falcon initiator and a Falcon target. The flows depict the life of a push transaction and a pull transaction under normal and error conditions.

6.9.1 RDMA Read Flow

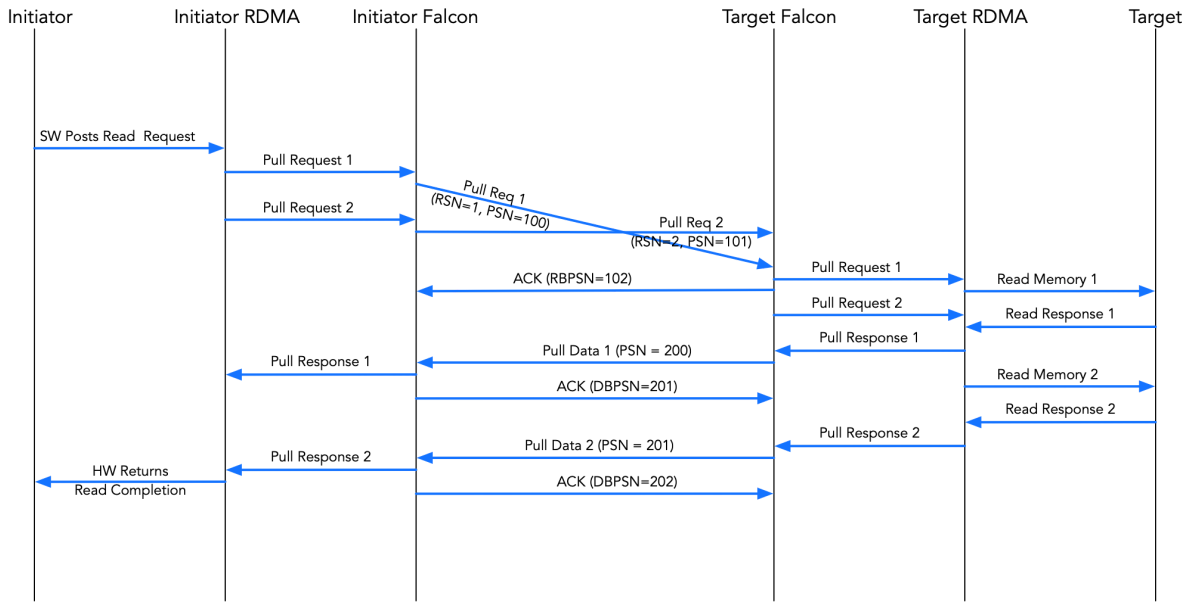


Figure 8: Life of a RDMA Read Transaction

The life of a RDMA Read transaction is shown in the figure above. The connection is assumed to be an ordered connection. The following sequence of operations are performed at the initiator and target sides:

1. Software posts a RDMA read request to a send queue. This results in the RDMA protocol engine issuing two pull requests to Falcon. The two pull requests are created because each pull request is limited to one MTU in length and the original read request is larger than one MTU.
2. The initiator creates two pull request packets (with RSN=1 and RSN=2) and transmits them to the target. The packet delivery sublayer assigns PSN=100 and PSN=101 to the two pull request packets.
3. The two pull request packets are reordered by the network and arrive out of order at the target. The target reorders the two pull requests by RSN and delivers them to the RDMA engine. The target also triggers the generation of the ACK packet acknowledging the receipt of the two pull request packets. In the figure above, a single coalesced ACK is sent from the target to the initiator.
4. The RDMA block performs the memory read operation for each pull request and returns pull responses to the target.
5. The target creates two pull data packets and transmits them to the initiator. The packet delivery sublayer assigns PSN=200 and PSN=201 to the pull responses.
6. The initiator receives the two pull data packets and creates pull responses back to the RDMA block. The initiator also triggers the generation of ACK packets by the packet delivery sublayer acknowledging the receipt of the pull data packets.
7. After the RDMA block at the initiator receives both pull responses, it creates a read completion and posts it to the completion queue.

6.9.2 RDMA Write Flow

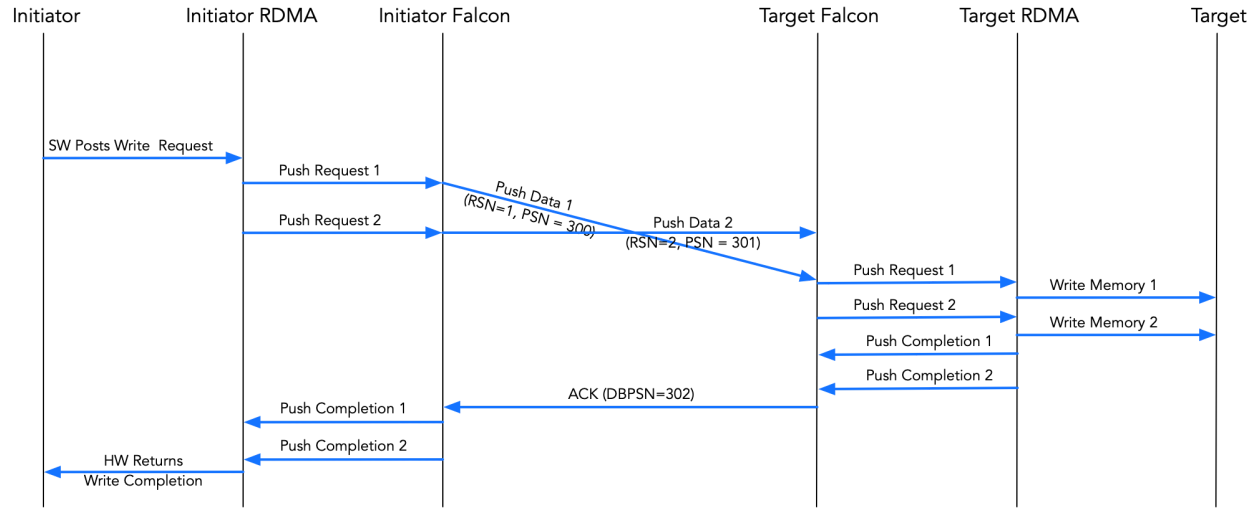


Figure 9: Life of a RDMA Write Transaction

The life of a RDMA Write transaction is shown in the figure above. The connection is assumed to be an ordered connection. The following sequence of operations are performed at the initiator and target sides:

1. Software posts a RDMA write request to a send queue. This results in the RDMA protocol engine issuing two push requests to Falcon. The two push requests are created because each push request is limited to one MTU in length and the original write request is larger than one MTU.
2. The initiator creates two push data packets (with RSN=1 and RSN=2) and transmits them to the target. The packet delivery sublayer assigns PSN=300 and PSN=301 to the two push data packets.
3. The two push data packets are reordered by the network and arrive out of order at the target. The target reorders the two push data packets by RSN and delivers them to the RDMA engine.
4. The RDMA block performs the memory write operation for each push request and returns push completions to Falcon. After receiving the push completions, the target Falcon triggers the generation of the ACK packet acknowledging the receipt of the two push data packets. In the figure above, a single coalesced ACK is sent from the target to the initiator.
5. Upon receiving the ACK for the push data packets, the initiator delivers two push completions back to the RDMA block.
6. After the RDMA block at the initiator receives both push completions, the RDMA block creates a write completion and posts the completion to the completion queue.

6.9.3 Push Transaction Receiver Not Ready (RNR) NACK Flow

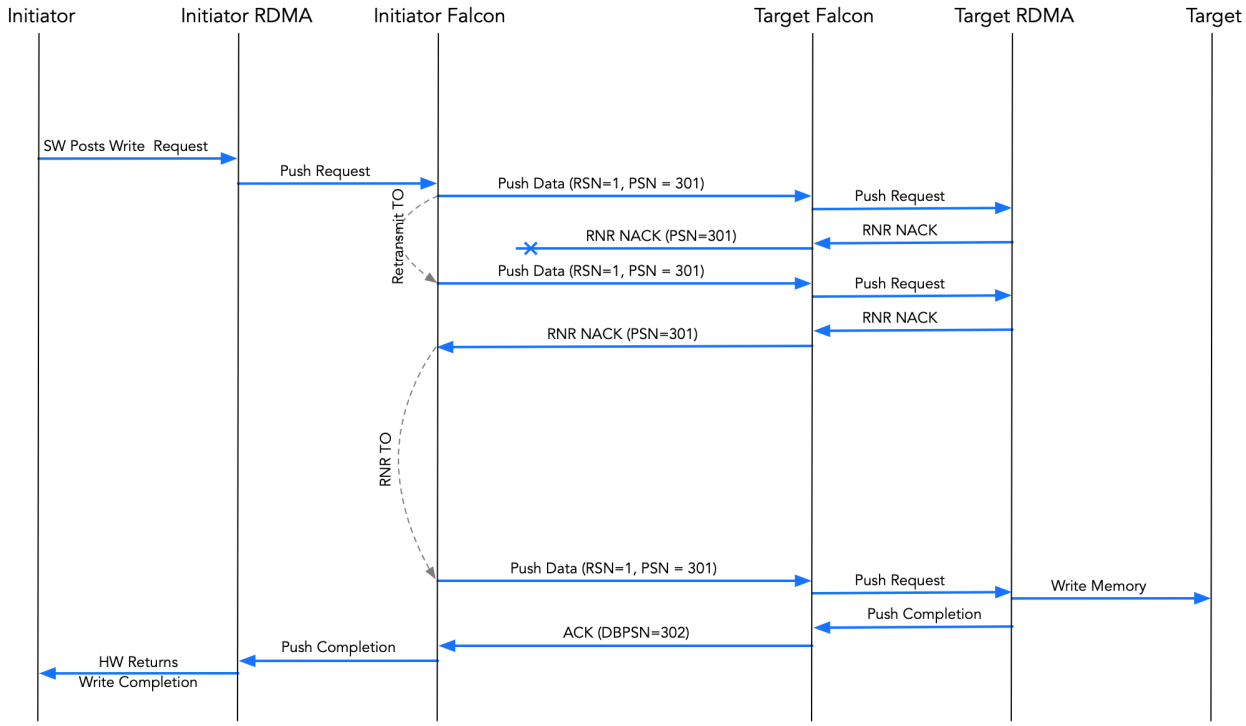


Figure 10: RNR NACK Flow for a Push Transaction

The RNR NACK flow for a push transaction is shown in the figure above. The connection is assumed to be an ordered connection. RNR NACK handling and associated packet retransmission is entirely handled within Falcon. This simplifies the ULP implementations since the ULP layers can assume that reliable delivery is completely handled at the Falcon layer. The following sequence of operations are performed at the initiator and target sides:

1. Software posts a RDMA write request to a send queue. This results in the RDMA protocol engine issuing a push request to Falcon.
2. The initiator creates a push data packet (with RSN=1) and transmits the packet to the target. The packet delivery sublayer assigns PSN=301 to the push data packet.
3. The target receives the push data packet and delivers a push request to the RDMA engine.
4. The RDMA engine cannot accept the push transaction and replies back with an RNR NACK. The target creates a RNR NACK packet and sends the packet to the initiator. This RNR packet is dropped in the network.

5. Since the push data packet sent in step 2 has not been acknowledged, the retransmit timeout for the packet fires at the initiator and the packet delivery layer at the initiator retransmits the packet to the target.
6. The target receives the retransmitted push data packet and delivers a push request to the RDMA engine. The RDMA engine again cannot accept the push request and replies back with an RNR NACK. The target creates a RNR NACK packet and sends this packet to the initiator.
7. As shown in the figure, this second attempt is successful and the packet delivery layer at the initiator adjusts the retransmit timeout for the push data packet to the timeout value in the RNR NACK packet.
8. After the retransmit timer fires (after the RNR NACK delay), the packet delivery sublayer at the initiator retransmits the packet to the target.
9. The target receives the retransmitted push data packet and delivers a push request to the RDMA engine. This time the RDMA engine at the target accepts the push request, performs a memory write and returns a push completion.
10. After receiving the push completions, the target triggers the generation of the ACK packet acknowledging the receipt of the push data packet.
11. Upon receiving the ACK for the push data packets, the initiator delivers a push completion back to the RDMA block.
12. After the RDMA block at the initiator receives the push completion, it creates a write completion and posts the write completion to the completion queue.

6.9.4 Pull Transaction RNR NACK Flow

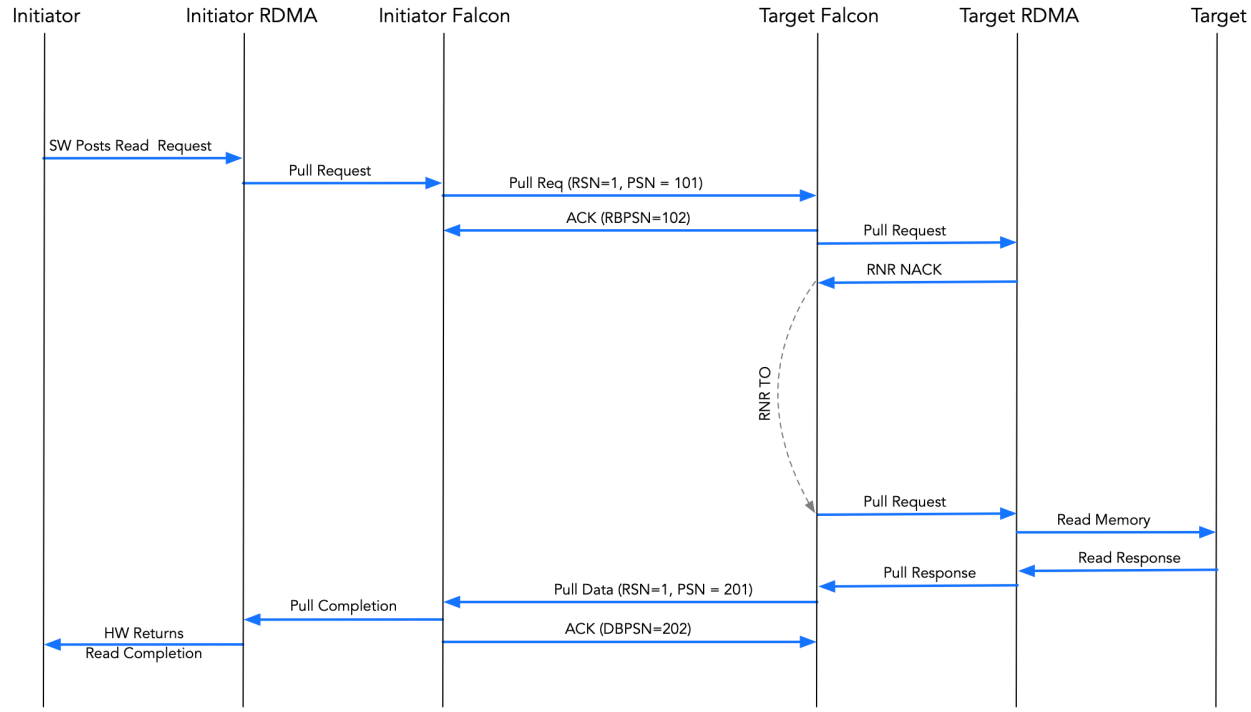


Figure 11: RNR NACK Flow for a Pull Transaction

The RNR NACK flow for a pull transaction is shown in the figure above. The connection is assumed to be an ordered connection. The following sequence of operations are performed at the initiator and target sides:

1. Software posts a RDMA read request to a send queue. This results in the RDMA protocol engine issuing a pull request to Falcon.
2. The initiator creates a pull request packet (with RSN=1) and transmits it to the target. The packet delivery sublayer assigns PSN=101 to the pull request packet.
3. The target receives the pull request by RSN and delivers the pull request to the RDMA engine. The target also triggers the generation of the ACK packet acknowledging the receipt of the pull request packet.
4. The RDMA engine cannot accept the pull request and replies with an RNR NACK. The target starts a retransmit timer with the timeout set to the value indicated in the RNR NACK indication from the RDMA engine.
5. When the retransmit timer expires, the target retransmits the pull request to the RDMA block.
6. This time, the RDMA block accepts the retransmitted pull request and performs the memory read operation and returns a pull response to Falcon.

7. The target Falcon creates a pull data packet and transmits this packet to the initiator. The packet delivery sublayer assigns PSN=201 to the pull data packet.
8. The initiator receives the pull data packet and creates a pull response back to the RDMA block. The initiator also triggers the generation of the ACK packet by the packet delivery sublayer acknowledging the receipt of the pull data packet.
9. After the RDMA block at the initiator receives the pull response, the RDMA block creates a read completion and posts the completion to the completion queue.

6.9.5 Push Transaction Complete In Error (CIE) NACK Flow

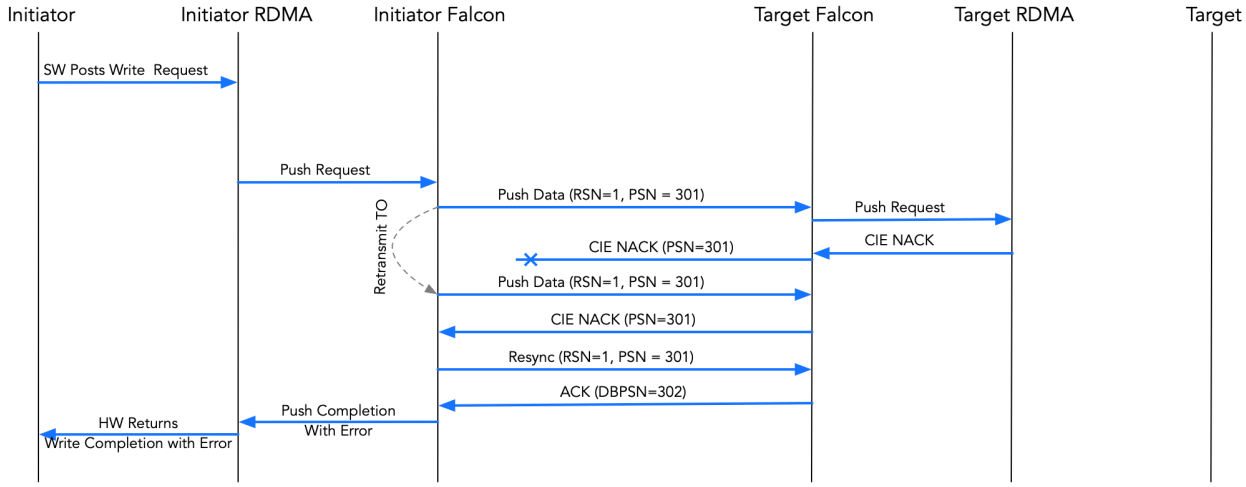


Figure 12: CIE NACK Flow for a Push Transaction

The CIE NACK flow for a push transaction is shown in the figure above. The connection is assumed to be an ordered connection. The following sequence of operations are performed at the initiator and target sides:

1. Software posts a RDMA write request to a send queue. This results in the RDMA protocol engine issuing a push request to Falcon.
2. The initiator creates a push data packet (with RSN=1) and transmits this to the target. The packet delivery sublayer assigns PSN=301 to the push data packet.
3. The target receives the push data packet and delivers a push request to the RDMA engine.
4. The RDMA engine does not accept the push request and returns a CIE NACK to Falcon. The target Falcon triggers the transmission of a CIE NACK packet to the initiator. In the scenario described in the above figure, the CIE NACK packet is dropped in the network.
5. After a retransmission timeout, the initiator retransmits the push data packet to the target.

6. The target receives the retransmitted push data packet and determines that the transaction was previously NACKed with a CIE indication. The target triggers the retransmission of the same CIE NACK packet (with the same error code as before) by the packet delivery layer.
7. The CIE NACK packet is received by the initiator. The initiator triggers the transmission of a Resync packet to advance the data sliding window at the receiver with the PSN of the push data packet.
8. The target receives the Resync packet and advances its data sliding window. The target triggers the transmission of an ACK packet acknowledging the receipt of the Resync packet.
9. Upon receiving the acknowledgement for the Resync packet, the initiator completes the push request by sending push completion (with the error indication) back to the RDMA engine.
10. After the RDMA block at the initiator receives the push completion, the RDMA block creates a write completion (with the error indication) and posts it to the completion queue.

6.9.6 Early Retransmission using Extended ACK (EACK)

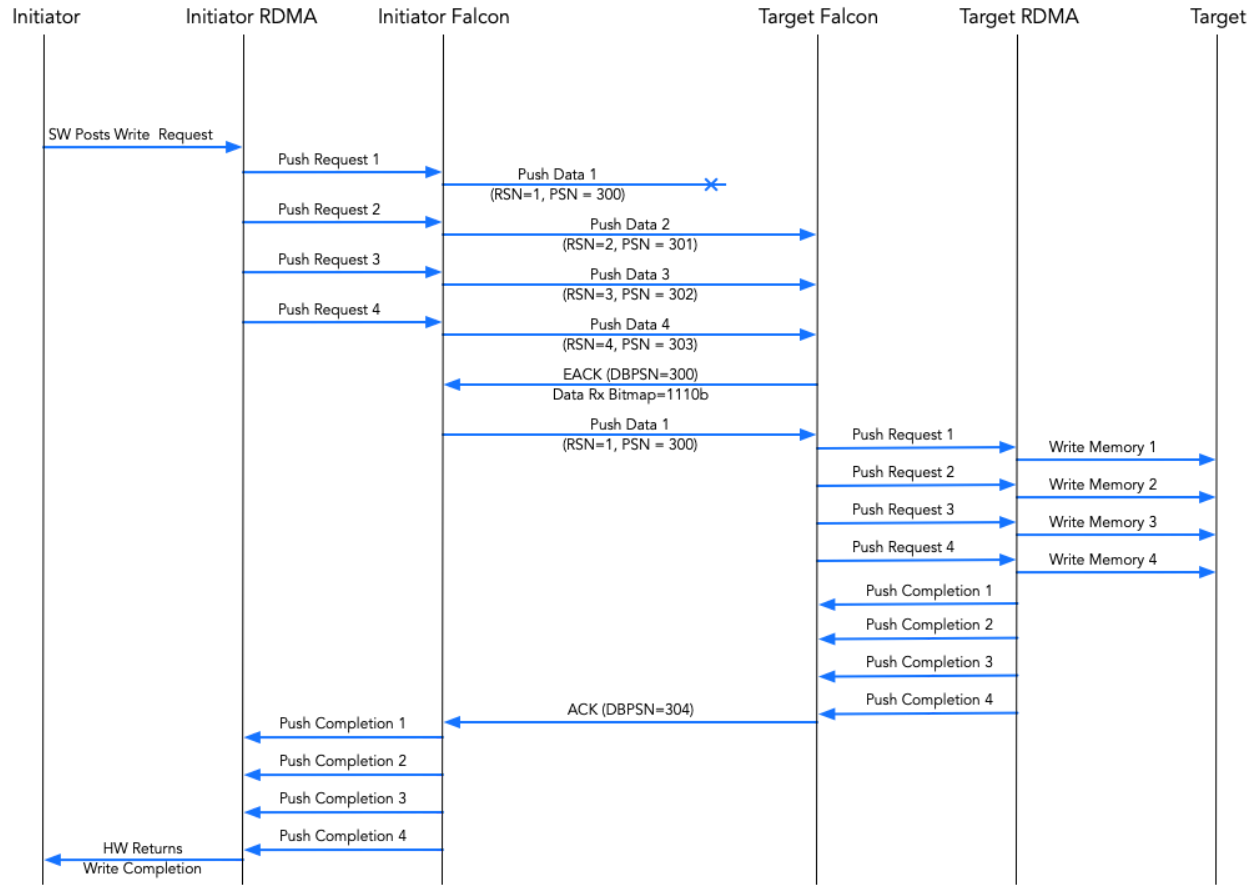


Figure 13: Extended ACK (EACK) flow for RDMA Write Request

EACK (Extended ACK) is transmitted when there are out-of-order arrivals, which may trigger early retransmission. The EACK flow is shown in the figure above. The connection is assumed to be an ordered connection. The following sequence of operations are performed at the initiator and target sides:

1. Software posts a RDMA write request to a send queue. This results in the RDMA protocol engine issuing 4 push requests to Falcon. The four 4 requests are created because each push request is limited to one MTU in length and the original write request is larger than 3 MTUs.
2. The initiator creates 4 push data packets (with RSN=1 to 4) and transmits them to the target. The packet delivery sublayer assigns PSN=300 to 303 to the 4 push data packets.

3. The 1st push data packet is lost in the network, and the latter 3 push data packets arrive at the target. The target holds the three without delivering them to the RDMA engine because they are out of order.
4. The target Falcon generates an EACK whose Data Rx Bitmap = 1110b, which indicates a hole at data sliding window PSN 300.
5. The initiator, upon receiving the EACK, triggers early retransmission of data sliding window PSN 300.
6. The target receives the retransmitted push data and delivers them to the RDMA engine in order by RSN.
7. The RDMA block performs the memory write operation for each push request and returns push completions to Falcon. After receiving the push completions, the target Falcon triggers the generation of the ACK packet acknowledging the receipt of the 4 push data packets. In the figure above, a single coalesced ACK is sent from the target to the initiator.
8. Upon receiving the ACK for the push data packets, the initiator delivers 4 push completions back to the RDMA block.
9. After the RDMA block at the initiator receives all 4 push completions, the RDMA block creates a write completion and posts the write completion to the completion queue.

7. Packet Formats

7.1 Falcon Packet

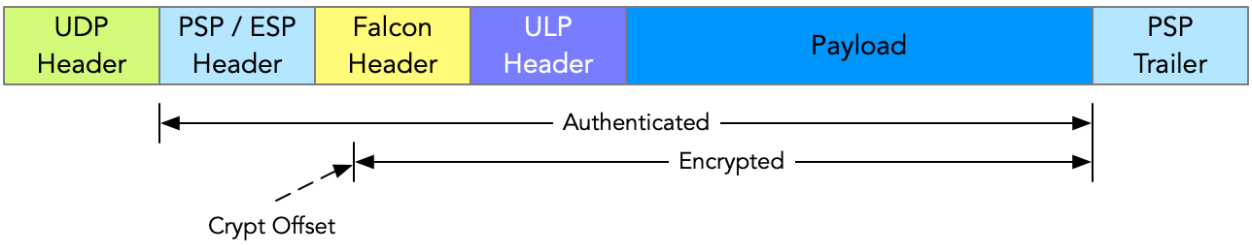


Figure 14: Wire Format of a Falcon Packet

The wire format of a Falcon packet is shown in the figure above. Falcon packets are encapsulated as the payload of an encryption protocol such as PSP or IPSEC ESP that provides authentication and encryption of the Falcon packet. The Falcon header follows the PSP/ESP header (indicated by a Next Header value of 252 in the PSP header). The payload of the Falcon packet contains the Upper Layer Protocol (ULP) packet payload with the Protocol Type field in the Falcon header indicating the ULP protocol. Immediately following the Falcon header is the ULP protocol-specific header. In the case of PSP, the PSP crypt offset field is configured to point beyond the first 4 bytes of the Falcon header (leaving the connection ID field in the clear).

7.1.1 Transport Mode

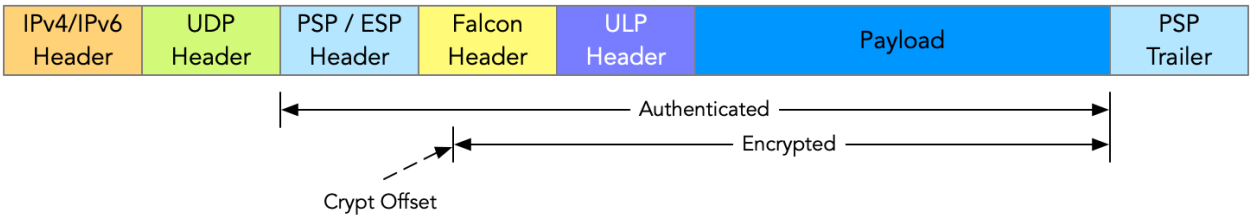


Figure 15: Falcon Packet Format in Transport Mode

The most common packet format for Falcon in the transport mode is shown in the figure above. The transport mode format has a single IP header where the source and destination IP addresses represent physical IP addresses of machines in the network. The transport mode format is used in non-virtualized deployments for both RDMA RC and RD modes.

7.1.2 Tunnel Mode

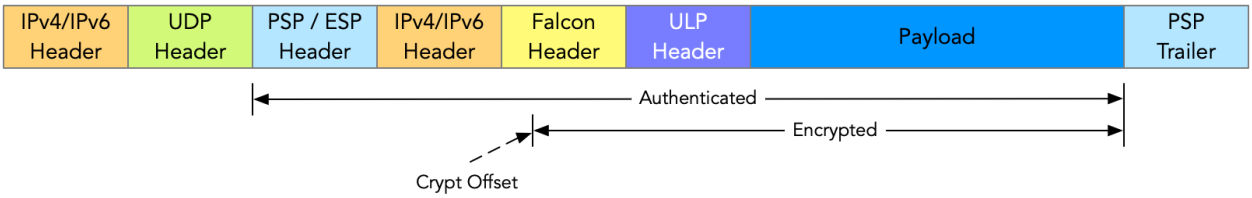


Figure 16: Falcon Packet Format in Tunnel Mode

Falcon can also be used in Tunnel mode (shown in figure above) which is useful in virtualized environments. In the tunnel mode, the PSP/ESP payload carries an inner IPv4/IPv6 packet.

7.2 Falcon Base Header

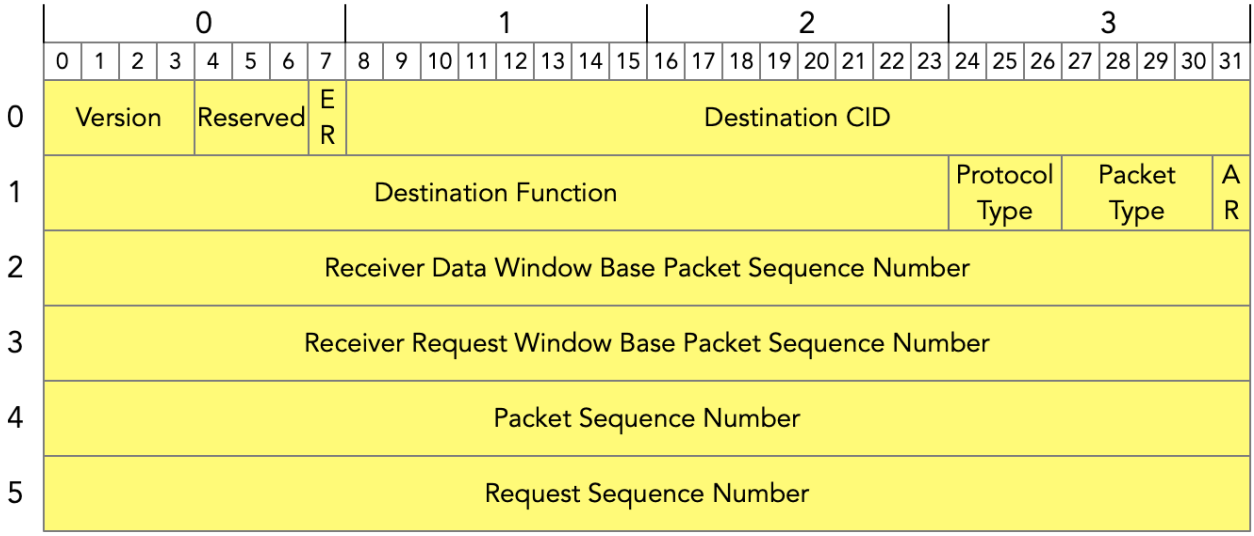


Figure 17: Falcon Base Header Format

All Falcon packets except the ACK and NACK packets carry the Falcon Base header. The format of the Falcon header is shown in the figure above and specifications of the various fields of the header are provided in the table below.

Table 1: Falcon Base Header Fields

Field	Bit Width	Description
Version	4	This field encodes the Falcon protocol version and must be set to 1.
EACK Request (ER)	1	This bit must be set to 1b by the transmitter to request an immediate Extended ACK (EACK) for this packet by the receiver.
Destination CID	24	This field encodes the destination connection id.
Destination Function	24	This field encodes the <Host, PF, VF> tuple of the function at the destination host that must receive the packet.
Protocol Type	3	This field specifies the upper layer protocol associated with this packet and is encoded as shown below: 000b Reserved 001b Reserved

Field	Bit Width	Description
		010b RDMA 011b NVMe 100b-111b Reserved
Packet Type	4	This field specifies the type of the packet and is encoded as shown below: 0000b Pull Request 0001b Reserved 0010b Reserved 0011b Pull Data 0100b Reserved 0101b Push Data 0110b Resync 0111b Reserved 1001b-1111b Reserved ACK and NACK's own headers also have this Packet Type field at the same location. They are encoded as shown below: 1000b NACK 1001b BACK 1010b EACK
Ack Req	1	This bit must be set to 1b by the transmitter to request an immediate acknowledgement for this packet by the receiver.

Field	Bit Width	Description
Receiver Data Window Base PSN	32	This field encodes the base sequence number maintained by the receiver's data sliding window for the connection and is used to piggyback acknowledgements for the data sliding window.
Receiver Request Window Base PSN	32	This field encodes the base sequence number maintained by the receiver's request sliding window for the connection and is used to piggyback acknowledgements for the request sliding window.
Packet Sequence Number	32	This field encodes the sequence number assigned to the packet by the transmitter according to the sliding window protocol. For Pull Request packets this is the request sliding window PSN. For Push Data, Pull Data, and Resync packets this is the data sliding window PSN.
Request Sequence Number	32	This field encodes the unique identifier assigned to a ULP request by the transaction sublayer. For an ordered connection, this field is used by the receiver to reorder requests before they are passed to the ULP layer.

7.3 Pull Request Packet

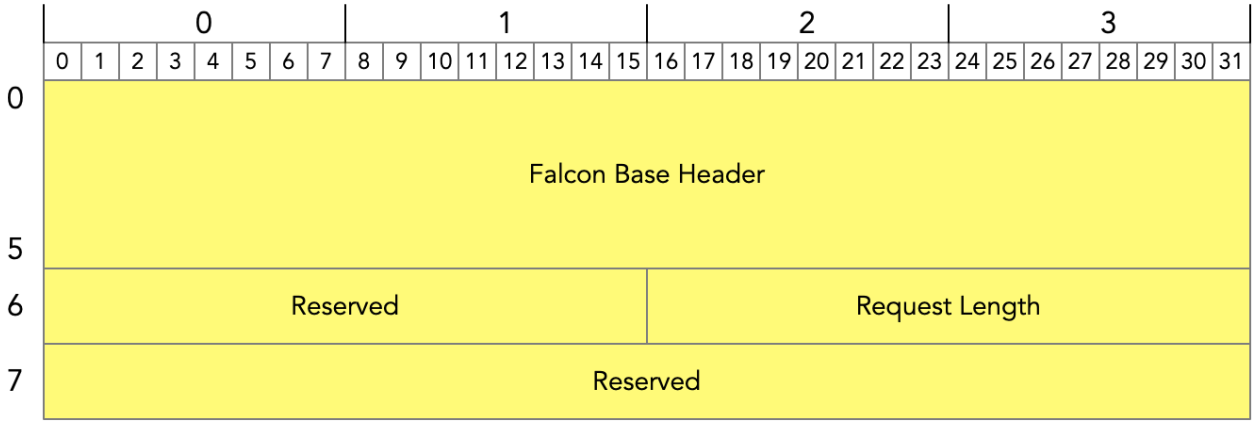


Figure 18: Pull Request Packet Format

A pull request packet is sent by the initiator to the target in response to a pull transaction initiated by the ULP. At the target the pull request packet is delivered to the target ULP and the ULP must complete the pull transaction by sending a pull response. The format of the pull request packet is shown in the figure above and is defined by the table below.

Table 2: Pull Request Packet Fields

Field	Bit Width	Description
Falcon Base Header	192	The Falcon Base Header is described here.
Request Length	16	This field encodes the size of the request in bytes. The payload of the corresponding pull data packet must match this request length. Therefore, the request length must include the ULP headers, ULP padding and the size of the ULP request.
Reserved	32	This field is reserved and must be encoded as 0.

7.4 Pull Data Packet

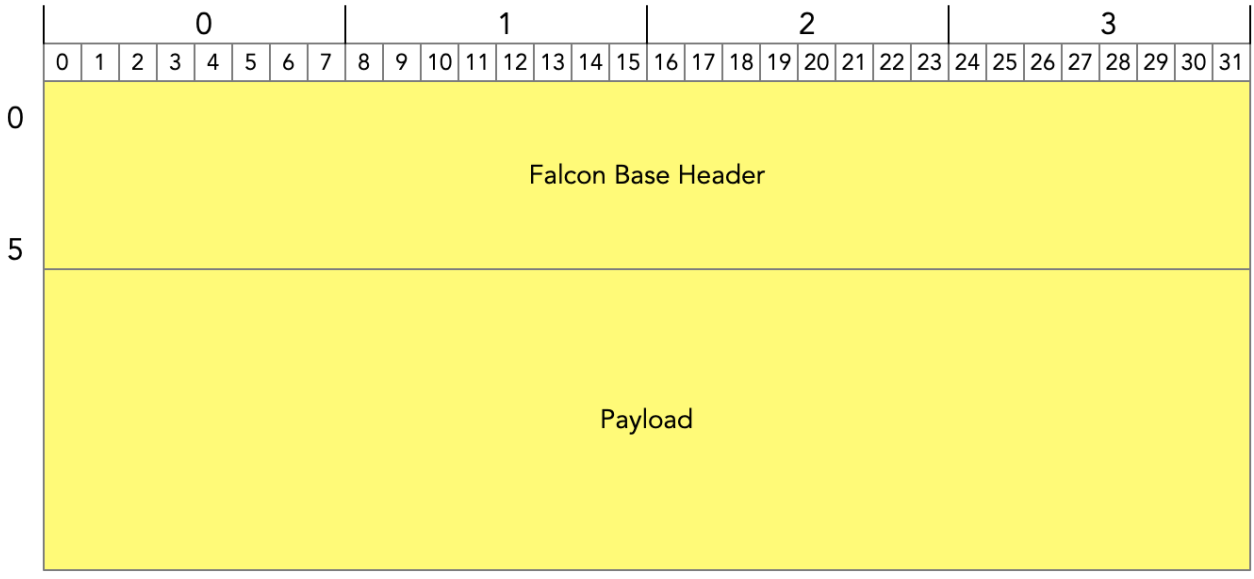


Figure 19: Pull Data Packet Format

A pull data packet is sent by ULP at the target to the initiator in response to a pull request received by the target. The format of the pull data packet is shown in the figure above and is defined by the table below.

Table 3: Pull Data Packet Fields

Field	Bit Width	Description
Falcon Base Header	192	The Falcon Base Header is described here. The Request Sequence Number (RSN) field equals the corresponding Pull Request's RSN.
Payload	<N>	The payload of the pull data packet includes the ULP headers, ULP trailers (if any) and the ULP payload bytes.

7.5 Push Data Packet

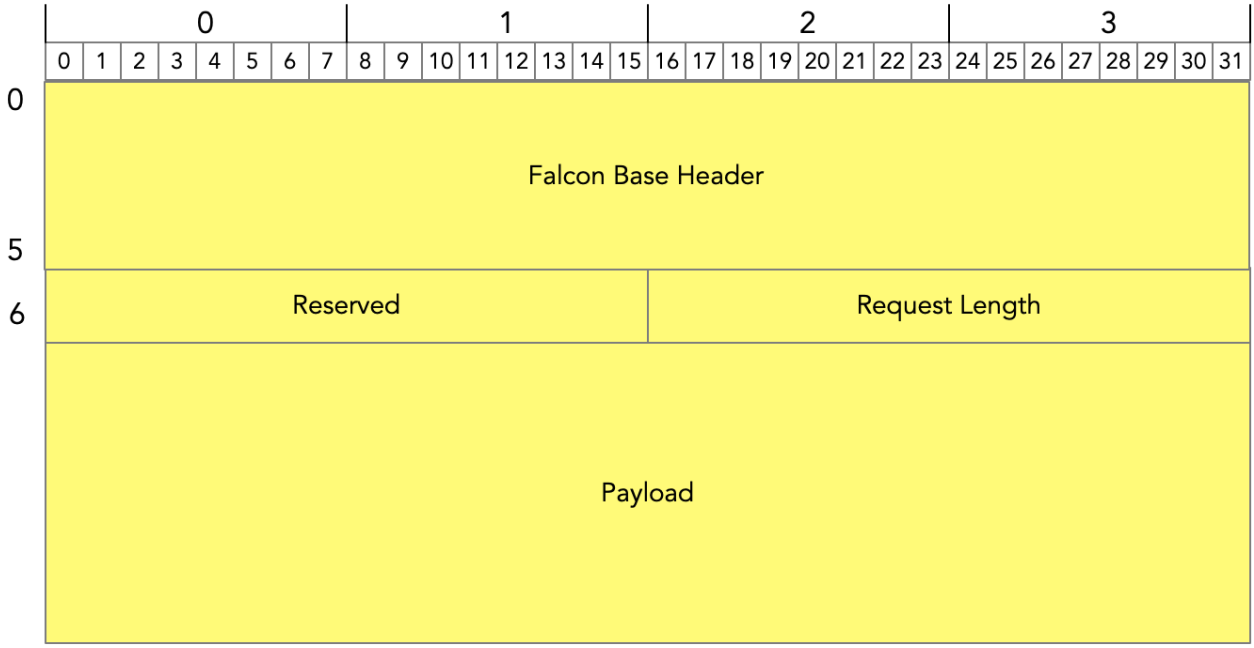


Figure 20: Push Data Packet Format

A push data packet is sent by the initiator to the target in response to a push transaction initiated by the ULP. The format of the push data packet is shown in the figure above and is defined by the table below.

Table 4: Push Data Packet Fields

Field	Bit Width	Description
Falcon Base Header	192	The Falcon Base Header is described here.
Request Length	16	This field encodes the size of the push request in bytes. The payload size of the push data packet must match this request length.
Payload	<N>	The payload of the push data packet includes the ULP headers, ULP trailers (if any) and the ULP payload bytes.

7.6 Resync Packet

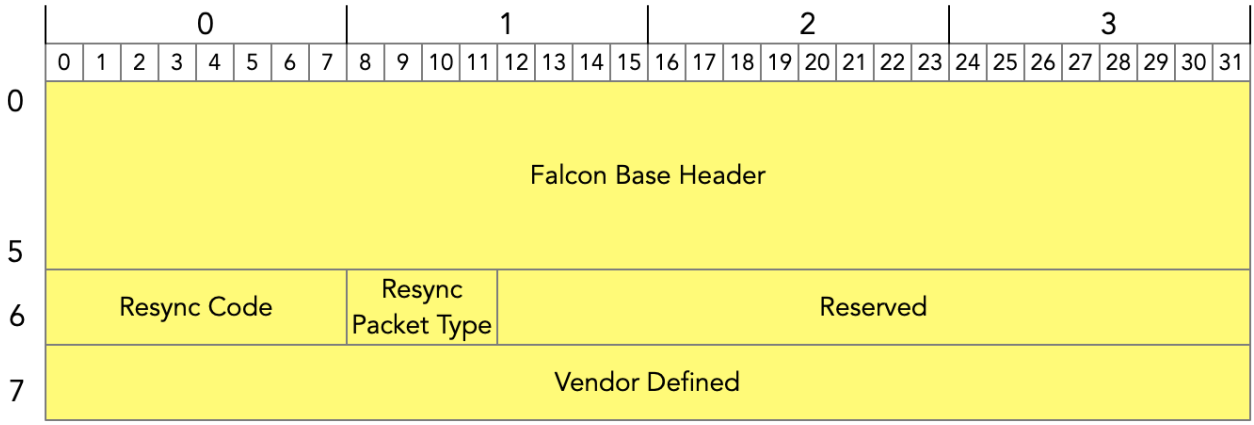


Figure 21: Resync Packet Format

The Resync packet is sent from the Falcon transmitter to the Falcon receiver to synchronize the sliding window state (i.e., PSN). The format of the resync packet is shown in the figure above and is defined by the table below.

Table 5: Resync Packet Fields

Field	Bit Width	Description
Falcon Base Header	192	The Falcon Base Header is encoded as described here.

Field	Bit Width	Description
Resync Code	8	This field specifies the reason for the Resync operation and is encoded as follows: 0x0 Reserved 0x1 Target ULP complete-in-error 0x2 Local xLR flow 0x3 Packet (exhausting) Retransmission Error 0x4 Transaction timed out 0x5 Remote xLR flow 0x6 Target ULP non-recoverable error 0x7 Target ULP invalid CID error 0x8-0xff Reserved
Resync Packet Type	4	Packet Type value of original packet that is the reason for the resync operation.
Vendor Defined	32	This is an opaque vendor-defined field; encodes to 0 when unused.

7.7 Acknowledgement (ACK) Packet

There are two types of ACK packets: Base ACK (BACK) and Extended ACK (EACK). In this spec, an ACK packet may refer to either BACK or EACK if not specified, depending on the context.

7.7.1 Base ACK (BACK) Packet

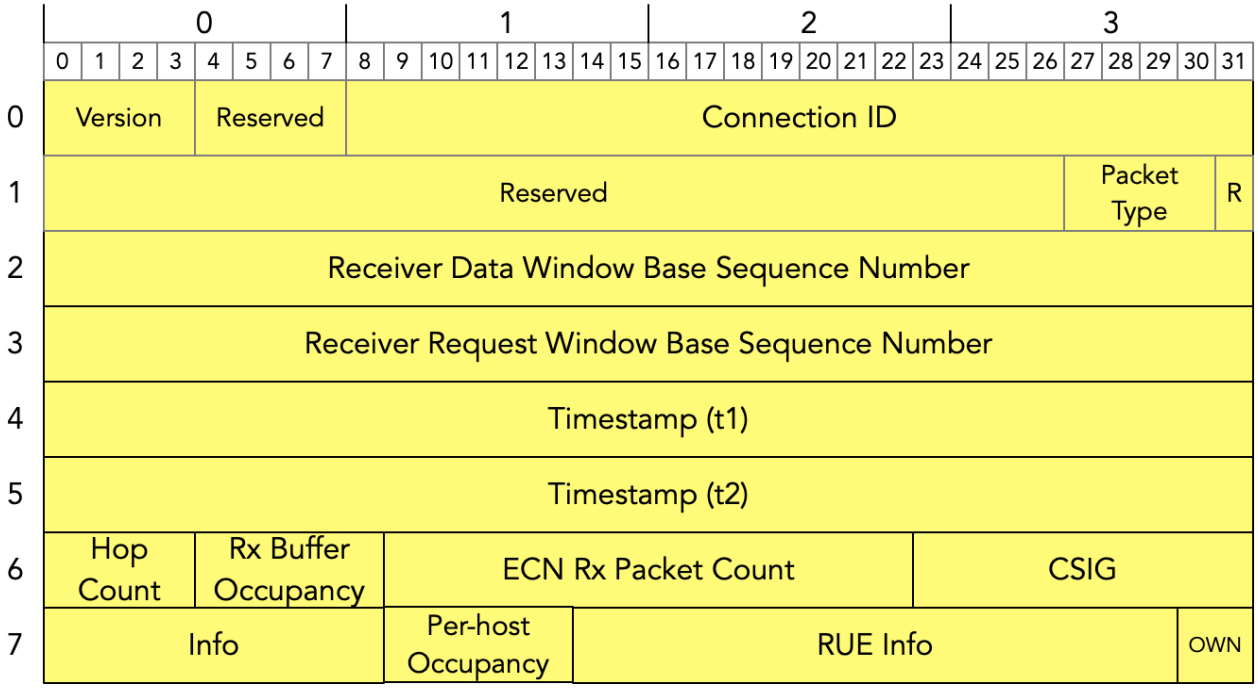


Figure 22: Base ACK (BACK) Packet Format

The BACK packet is sent by the Falcon receiver to the Falcon transmitter to acknowledge the receipt of packets. Acknowledgements may be coalesced by the receiver. Therefore, a BACK packet may acknowledge multiple packets that were received previously. The format of the BACK packet is shown in the figure above and is defined by the table below.

Table 6: Base ACK (BACK) Packet Fields

Field	Bit Width	Description
Version	4	This field encodes the Falcon protocol version and must be set to 1.
Connection ID	24	This field encodes the ID of the connection associated with this BACK packet.
Packet Type	4	This field must be encoded as 1001b.
Receiver Data Window Base PSN	32	This field encodes the base sequence number maintained by the receiver's data sliding window.

Field	Bit Width	Description
Receiver Request Window Base PSN	32	This field encodes the base sequence number maintained by the receiver's request sliding window.
Timestamp (t1)	32	This field encodes the timestamp value contained in the PSP/ESP header IV field of a packet received by the receiver. The timestamp value is encoded in 131.072 ns units.
Timestamp (t2)	32	This field encodes the time when a packet is received by the receiver. The timestamp is encoded in 131.072 ns units.
Hop Count	4	This field encodes the forward path hop-count of a packet received by the receiver.
Rx Buffer Occupancy	5	This field encodes the quantized EMA receiver buffer occupancy level.
ECN Rx Packet Count	14	This field encodes the cumulative count of ECN marked packets received.
CSIG Info	17	The first bit of this field indicates whether CSIG data is valid or not. Remaining 16b carry the CSIG data received from the Compact CSIG tag of incoming packet.
Host Rx Buffer Occupancy	5	This field encodes the quantized EMA receiver buffer occupancy level corresponding to the receiver host.
RUE Info	17	This field encodes the value driven by the Rate Update Engine (RUE).
Out of Window Notification (OWN)	2	This field indicates that the receiver dropped a packet due to lack of space in the sliding window: Bit 0 is for the Request Window Bit 1 is for the Data Window.

With multipathing, the receiver maintains separate sets of congestion control metadata, such as timestamps and hop counts, for each path within a connection. When an acknowledgement is generated for a packet arriving on a certain path, the acknowledgement is populated with congestion control metadata to that path. Acknowledgement coalescing is also performed at a per-flow granularity.

7.7.2 Extended ACK (EACK) Packet

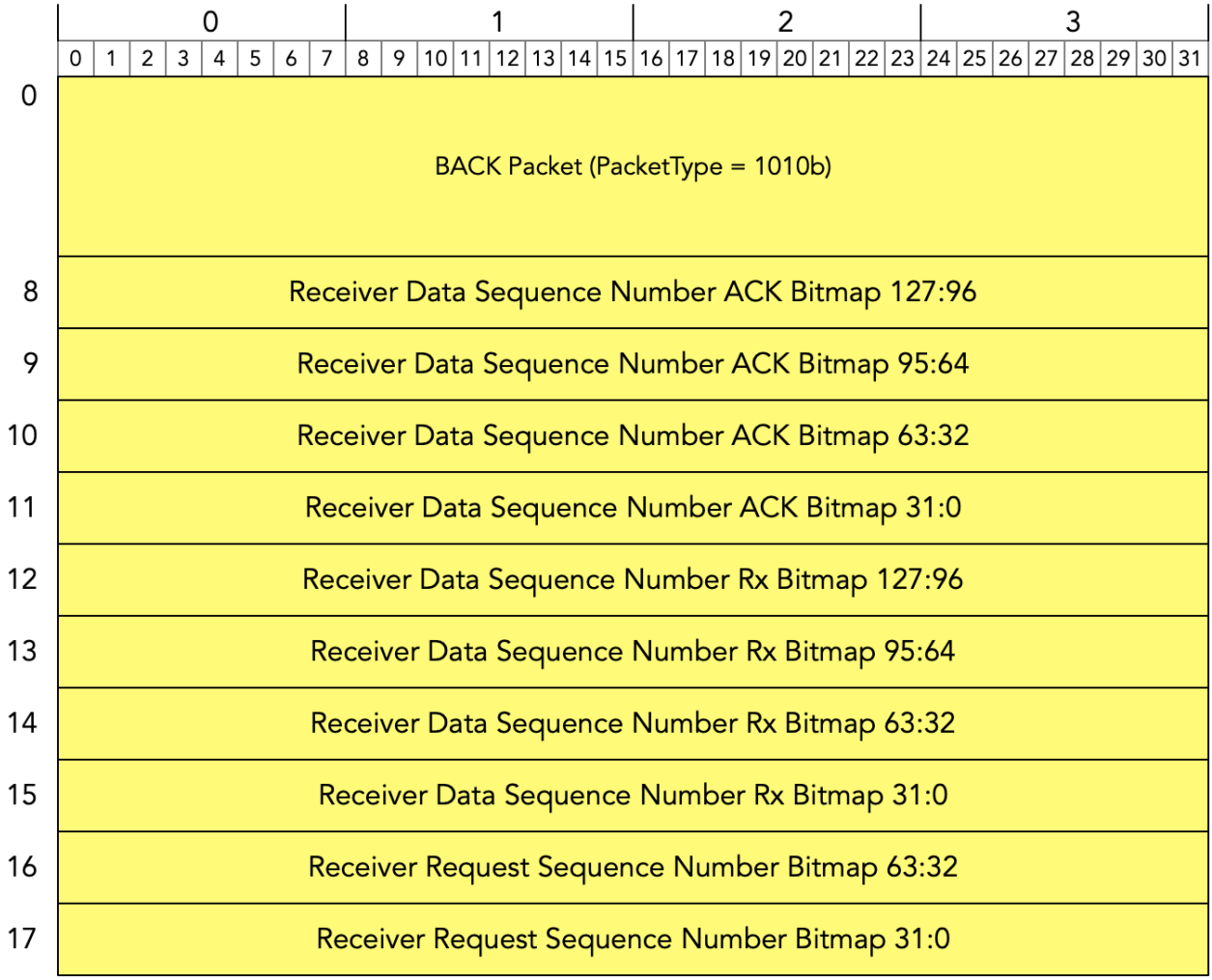


Figure 23: Extended ACK (EACK) Packet Format

The EACK packet extends BACK with 3 bitmaps: Receiver Data Sequence Number ACK Bitmap (data-ack-bitmap), Receiver Data Sequence Number Rx Bitmap (data-rx-bitmap), Receiver Request Sequence Number Bitmap (req-bitmap). The reason for having these three bitmaps is explained in the packet delivery sublayer.

Table 7: Extended ACK (EACK) Packet Fields

Field	Bit Width	Description
BACK Header	256	The BACK Header is encoded as described before, except the packet type field, which should be 1010b to indicate it to be an EACK.
Receiver Data Sequence Number ACK Bitmap (data-ack-bitmap)	128	This bitmap contains a snapshot of the receiver's sliding window bitmap for the ACK-state of data packets when this EACK packet was generated.
Receiver Data Sequence Number Rx Bitmap (data-rx-bitmap)	128	This bitmap contains a snapshot of the receiver's sliding window bitmap for the received-state (not necessarily acknowledged) of data packets when this EACK packet was generated.
Receiver Request Sequence Number Bitmap (req-bitmap)	64	This bitmap contains a snapshot of the receiver's sliding window bitmap for request packets when this EACK packet was generated.

7.8 Negative Acknowledgement (NACK) Packet

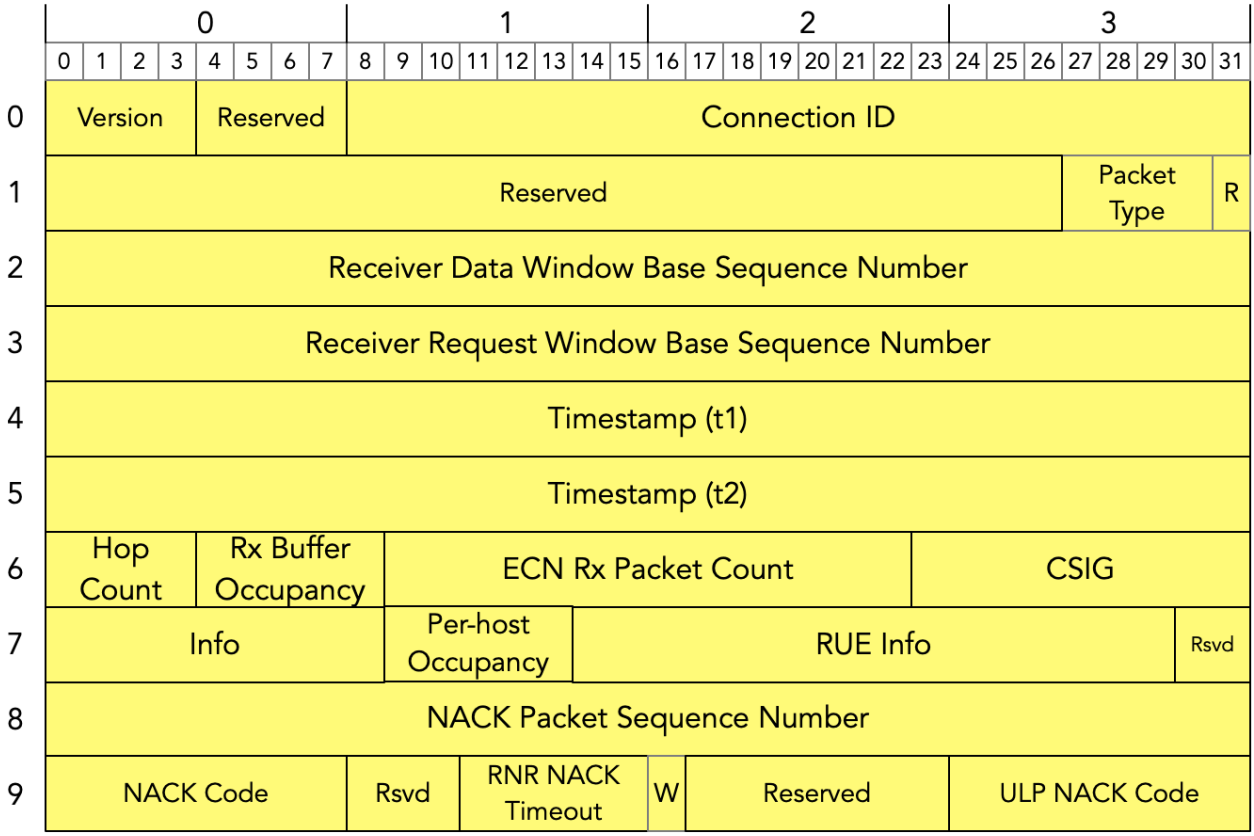


Figure 24: Negative Acknowledgement (NACK) Packet Format

The NACK packet is sent by the Falcon receiver to the Falcon transmitter in response to errors signaled by the ULP or to packets received out of order beyond the receiver sliding window. The format of the NACK packet is shown in the figure above and is defined by the table below.

Table 8: Negative Acknowledgement (NACK) Packet Fields

Field	Bit Width	Description
Version	4	This field encodes the Falcon protocol version and must be set to 1.
Connection ID	24	This field encodes the ID of the connection associated with this NACK packet.
Packet Type	4	This field must be encoded as 1000b.

<table>
<tr><th>Field</th><th>Bit Width</th><th>Description</th></tr>
<tr><td>Receiver Data Window Base PSN</td><td>32</td><td>This field encodes the base sequence number maintained by the receiver's data sliding window.</td></tr>
<tr><td>Receiver Request Window Base PSN</td><td>32</td><td>This field encodes the base sequence number maintained by the receiver's request sliding window for the connection.</td></tr>
<tr><td>NACK PSN</td><td>32</td><td>This field encodes the PSN of the packet that is being NACKed.</td></tr>
<tr><td>NACK Code</td><td>8</td><td>This field specifies the reason for sending the NACK packet and is encoded as shown below:
<table>
<tr><th>Value</th><th>Definition</th></tr>
<tr><td>0</td><td>Reserved</td></tr>
<tr><td>1</td><td>Request dropped due to insufficient resources at the receiver</td></tr>
<tr><td>2</td><td>ULP Receiver Not Ready (RNR)</td></tr>
<tr><td>3</td><td>Reserved</td></tr>
<tr><td>4</td><td>xLR drop indication</td></tr>
<tr><td>5</td><td>Reserved</td></tr>
<tr><td>6</td><td>ULP complete in error indication</td></tr>
<tr><td>7</td><td>ULP non-recoverable error indication</td></tr>
<tr><td>8</td><td>Invalid CID error indication</td></tr>
<tr><td>9-255</td><td>Reserved</td></tr>
</table></td></tr>
<tr><td>RNR NACK Timeout</td><td>5</td><td>This field encodes the timeout value for the Receiver Not Ready (RNR) NACK as shown below:</td></tr>
</table>

<table>
<tr><th>Field</th><th>Bit Width</th><th>Description</th></tr>
<tr><td></td><td></td><td>
<table>
<tr><th>Value</th><th>Delay in milliseconds</th><th>Value</th><th>Delay in milliseconds</th></tr>
<tr><td>00000b</td><td>655.36</td><td>10000b</td><td>2.56</td></tr>
<tr><td>00001b</td><td>0.01</td><td>10001b</td><td>3.84</td></tr>
<tr><td>00010b</td><td>0.02</td><td>10010b</td><td>5.12</td></tr>
<tr><td>00011b</td><td>0.03</td><td>10011b</td><td>7.68</td></tr>
<tr><td>00100b</td><td>0.04</td><td>10100b</td><td>10.24</td></tr>
<tr><td>00101b</td><td>0.06</td><td>10101b</td><td>15.36</td></tr>
<tr><td>00110b</td><td>0.08</td><td>10110b</td><td>20.48</td></tr>
<tr><td>00111b</td><td>0.12</td><td>10111b</td><td>30.72</td></tr>
<tr><td>01000b</td><td>0.16</td><td>11000b</td><td>40.96</td></tr>
<tr><td>01001b</td><td>0.24</td><td>11001b</td><td>61.44</td></tr>
<tr><td>01010b</td><td>0.32</td><td>11010b</td><td>81.92</td></tr>
<tr><td>01011b</td><td>0.48</td><td>11011b</td><td>122.88</td></tr>
<tr><td>01100b</td><td>0.64</td><td>11100b</td><td>163.84</td></tr>
<tr><td>01101b</td><td>0.96</td><td>11101b</td><td>245.76</td></tr>
<tr><td>01110b</td><td>1.28</td><td>11110b</td><td>327.68</td></tr>
<tr><td>01111b</td><td>1.92</td><td>11111b</td><td>491.52</td></tr>
</table>
</td></tr>
<tr><td>Window (W)</td><td>1</td><td>This bit must be encoded as 1b for request sliding window and 0b for data sliding window.</td></tr>
<tr><td>ULP NACK code</td><td>8</td><td>This field encodes the NACK reason provided by the ULP receiver. This field is opaque to Falcon and is passed to the ULP when the NACK is received by the transmitter.</td></tr>
</table>

Field	Bit Width	Description
Timestamp (t1)	32	Forward path packet TX time at sender, in 131.072ns unit.
Timestamp (t2)	32	Forward path packet receival time at receiver, in 131.072ns unit.
Hop Count	4	This field encodes the forward path hop-count of a packet received by the receiver.
Rx Buffer Occupancy	5	This field encodes the quantized EMA receiver buffer occupancy level.
Per-host Occupancy	5	This field encodes the quantized EMA receiver buffer occupancy level corresponding to packets targeted to the destination host associated with this connection.
ECN Rx Packet Count	14	This field encodes the cumulative count of ECN marked packets received.
CSIG Info	17	The first bit of this field indicates whether CSIG data is valid or not. Remaining 16b carry the CSIG data received from the compact CSIG tag of incoming packet.
Host Rx Buffer Occupancy	5	This field encodes the quantized EMA receiver buffer occupancy level corresponding to the receiver host.
RUE Info	17	This field encodes the value driven by the Rate Update Engine (RUE).

8. Transaction Sublayer

The transaction sublayer is responsible for transaction ordering and management of resources held by each transaction during its lifetime in a manner that guarantees forward progress and avoids protocol deadlock. The following sections detail the behavior of the transaction sublayer in terms of the initiator and target behaviors. Each end of a Falcon connection is simultaneously both an initiator and a target.

8.1 Transaction Types

The transaction sublayer supports two types of transactions: Push and Pull. A transaction may result in one or more packets that are delivered reliably between the initiator and the target.

- Pull transaction: A Pull transaction is a 2-phase transaction where the initiator sends a Pull Request packet to the target and the target replies with a Pull Data packet. The request packet specifies the length of the data that is expected in response. The Pull Data packet payload must match the request length of the corresponding pull request packet. The Pull Data packet also serves as the completion returned to the ULP.
- Push transaction: A Push transaction is a 2-phase transaction where the initiator sends a Push Data packet to the target and the target acknowledges the receipt of the data packet. The target can acknowledge the received Push Data packet only after its ULP completes and acknowledges the received transaction. This dependency is introduced because the ACK of the Push Data packet will result in a completion being delivered to the ULP at the initiator.

8.2 Resource Management

Each transaction holds various resources within the transaction sublayer during its lifetime. This section describes the minimum resource management policies that must be implemented by the transaction sublayer to avoid protocol deadlock.

8.2.1 Avoiding Deadlocks

There are two factors that can cause protocol deadlocks. First, protocol deadlocks can occur due to the dependencies between transactions introduced by the Falcon protocol architecture. Second, Falcon transactions must allocate and hold resources at the initiator and the target. Protocol deadlock can occur if resources are allocated in a way that does not guarantee forward progress of transactions.

Implementations may choose different resource management strategies. The forward progress rules that need to be followed depend on whether Falcon resources are constrained or not. In unconstrained implementations where each connection has enough resources, only forward progress rules that arise due to dependencies between Falcon transactions need to be adhered to. For constrained implementations, additional forward progress rules that arise due to limited Falcon resources also need to be adhered to.

We now describe the forward progress rules for both unconstrained and constrained implementations.

8.2.1.1 Unconstrained Resource (UR) Implementations

In an unconstrained implementation, deadlocks that arise due to Falcon transaction dependencies need to be avoided.

The Falcon protocol architecture introduces dependency between transactions:

- The packet delivery layer cannot acknowledge a push data packet until the ULP at the target accepts and acknowledges the push transaction. This is required because the ACK of the push data packet results in a push completion being delivered to the ULP at the initiator. The push transaction cannot be completed until there is confirmation at the target that the ULP has accepted and acknowledged the push transaction.
- Consequently, this can lead to the advancement of the receiver sliding window at the target to be blocked by transaction ordering (using RSN) on an ordered connection. This can happen when the packet at the head of the sliding window is a push data packet and it cannot be delivered to the ULP because the head-of-line (HoL) transaction on that connection has not yet been delivered to the ULP. The HoL transaction may be another push data packet or a pull request.

An unconstrained implementation must follow the rules (henceforth referred to as **UR rules**) specified below to ensure guaranteed forward progress and protocol deadlock freedom:

- For an ordered connection, the sliding window layer must transmit (and retransmit) packets in RSN order.
- The ULP must guarantee that forward progress of transactions received on one connection is not dependent on the forward progress of transactions received on a different connection.

8.2.1.2 Constrained Resource (CR) Implementations

Implementations with constrained resources have **additional** rules to adhere to given the limited Falcon resources. Any given Falcon transaction holds up one or more of the following resources during its lifetime:

- ULP resources: The ULP typically holds resources corresponding to a request it initiates until it receives the corresponding response or completion. Likewise, for requests that the ULP receives from Falcon, it holds resources until the corresponding responses are generated and handed over to Falcon, e.g., for RDMA reads received from the network, Incoming Read Requests Queue slots are held until the responses are generated.
- Falcon resources: On receiving a request/response packet from the ULP, each packet holds up Falcon resources until the remote Falcon receiver acknowledges the packet.

Likewise, when Falcon receives packets from the network, each packet holds up Falcon resources until the packet is delivered to and acknowledged by the ULP.

These resources need to be managed to ensure they do not cause protocol deadlocks and block forward progress, through the following rules (henceforth referred to as **CR Rules**):

1. **Independent Resource Allocation for Incoming and Outgoing Packets:** For a **Falcon resource constrained** implementation where Falcon resources are limited, Falcon needs to ensure that incoming packets and outgoing packets allocate resources independent of each other. This rule applies to both ordered and unordered connections.

2. **Independent Handling of Same Direction Initiator and Target Generated Packets:** For a ULP and/or Falcon resource constrained implementation, Falcon needs to ensure that initiator generated packets and target generated packets in the same direction are handled independent of each other. This rule and its following corollaries apply to both ordered and unordered connections.

Table 9: Mapping of Initiator and Target generated packets by Transaction Type

Transaction Type	Initiator Generated Packet	Target Generated Packet
Pull	Pull Request	Pull Data
Push	Push Data	ACK (which implies completion)

   - For a ULP and Falcon resource constrained implementation, pull requests and pull data in the same direction need to use separate sliding window sequence number spaces which we describe further in the Packet Delivery section.
   - For a Falcon resource constrained implementation, Falcon needs to ensure that pull requests and pull data or push data and push data completion in the same direction should allocate resources independent of each other.

3. **The Target and Initiator must always accept HoL transactions and ensure that HoL transactions make unconditional forward progress for ordered connections.** For a Falcon resource constrained implementation, Falcon needs to ensure that if a HoL push data or pull request cannot be admitted into the target, another HoL push data or pull request (from a different connection) must already be admitted into the target. Likewise, the initiator needs to provide similar guarantees for pull data.

8.2.2 Resource Pools

For resource constrained implementation, we describe a method via which the resources within the transaction sublayer are carved into multiple resource pools. Implementations may need to implement additional resource management policies based on implementation constraints to meet performance isolation requirements. Transaction resources within the transaction sublayer can be classified into four resource pools as shown in the figure below:

- Tx Packet Pool: This pool implements resources that hold the transmit packet context. This includes the transaction metadata, packet headers and any other per-packet resources that are necessary for packet transmission. Every packet generated by the transaction sublayer requires a single resource in the Tx packet pool.
- Tx Buffer Pool: This pool implements the transmit packet buffers. The resources consumed by a transaction from this pool is variable and depends on the request length. Not every packet generated by the transaction sublayer requires a resource in the Tx buffer pool. Implementations typically implement the Tx buffer pool using buffers of fixed size units called cells. The size of the cell defines the allocation granularity of the Tx buffer pool.
- Rx Packet Pool: This pool implements resources that hold the receive packet context. This includes request packets received from the network as well completions to be delivered to the ULP. Every packet / completion generated by the transaction sublayer requires a single resource in the Rx packet pool.
- Rx Buffer Pool: This pool implements the receive packet buffers. The resources consumed by a transaction from this pool is variable and depends on the request length. Not every packet received from the network requires a resource in the Rx buffer pool. Implementations typically implement the Rx buffer pool using buffers of a fixed size unit called cells. The size of the cell defines the allocation granularity of the Rx buffer pool. Implementations may use different cell sizes for the Tx and Rx buffer pools.

This separation of resources between Tx and Rx pools is required to ensure independent resource allocation for outgoing and incoming packets (CR Rule#1) as well as to guarantee HoL transaction forward progress for ordered connections (UR Rule).

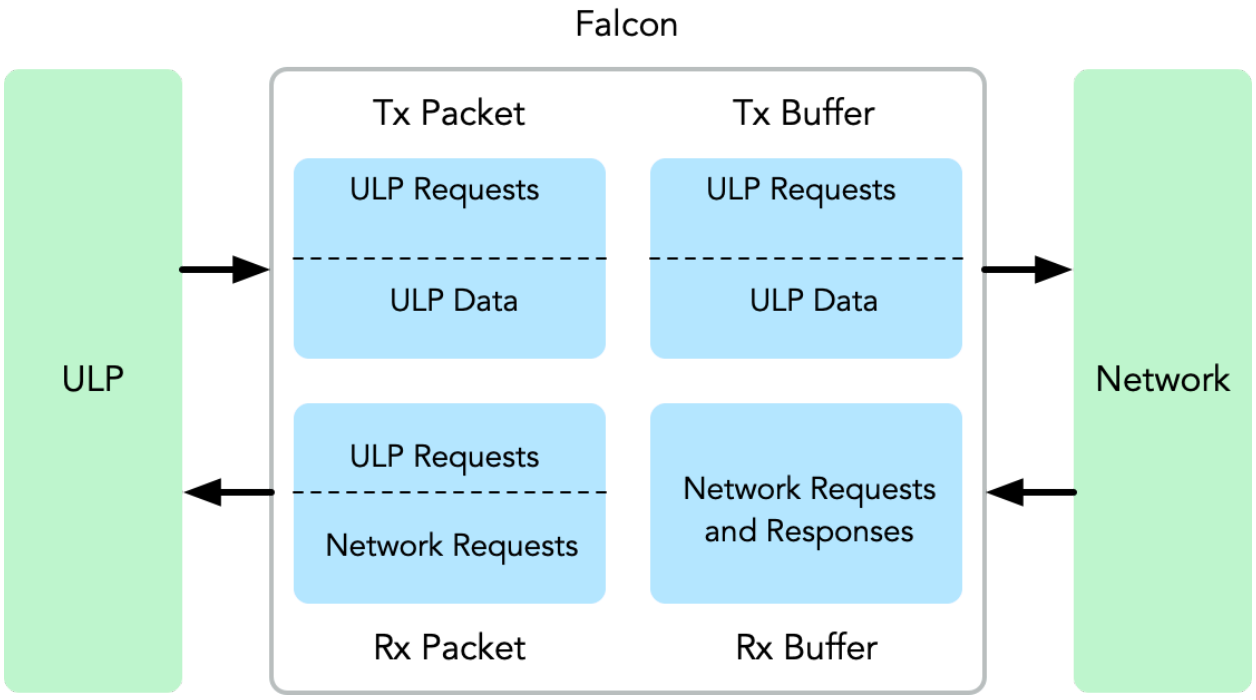


Figure 25: Resource Pools in Transaction Sublayer

Each resource pool within the transaction sublayer must be carved into different regions to provide dedicated resources for ULP generated requests and data packets and packets received from the network. This resource carving is required to ensure forward progress of ULP generated and network received transactions (satisfies CR Rule#1). The minimum resource carving that is required is shown in the figure above and is described below. Implementations may choose to provide additional resource carving to provide isolation between ULPs.

- ULP Requests: Tx buffer, Tx packet and Rx packet resource pools must provide a dedicated resource carve out for requests generated by the ULP. This region is used to implement flow control between ULP and the transaction sublayer as described in the next section.
- ULP Data: The Tx resource pools (Tx packet and Tx buffer) must provide a dedicated resource carve out for data packets generated by the ULP (i.e. Pull Data). This region is required to ensure outgoing requests and data are handled independently, satisfying CR Rule#2, and avoiding protocol deadlock.
- Network Requests: The Rx packet resource pool must provide a dedicated resource carve out for requests received from the network to ensure independent handling of ULP requests and network requests and thus avoiding protocol deadlock (satisfies CR Rule#1). Within the Network Requests region (both packets and buffers) further carve out must be provided for HoL transactions to avoid protocol deadlocks (UR Rule).

- Network Responses: The Rx buffer resource pool must provide carve out for HoL responses to avoid protocol deadlocks (UR Rule).

The resources allocated for each packet type transmitted and received by the transaction sublayer is shown in the table below. Resources for a particular transaction are proactively allocated on the initiator during the first phase when the request is received from ULP to ensure protocol correctness. For example, the pull request packet proactively allocates all the resources required by the subsequent pull data phase of the transaction at the initiator. As shown in the table below (Row B), the pull requests proactively allocate Rx packet resources from the ULP Requests region. These Rx resources need to be proactively allocated to hold metadata required for protocol correctness (e.g., allows matching the incoming pull data with its corresponding request). The resources allocated by each packet type is described in further detail in a later section.

Table 10: Resource Allocation per Packet Type

Packet Type	Initiator Side			Target Side	
	ULP Req Tx (Col 1)	ULP Req Rx (Col 2)	Net Resp Rx (Col 3)	ULP Data Tx (Col 4)	Net Req Rx (Col 5)
Push Data (Row A)	Yes	Yes	No	No	Yes
Pull Req (Row B)	Yes	Yes	No	No	Yes
Pull Data (Row C)	No	Use B2	Yes	Yes	No

8.2.3 ULP Resource Management

The transaction sublayer employs per-connection credit based flow control to prevent ULP from overrunning transaction resources. This flow control is applied only when there is a 1:1 association between a Falcon connection and ULP queue (e.g., RDMA Reliable Connection QPs). No flow control is used when there is a M:N association between Falcon connections and ULP queues (e.g. RDMA Unreliable Datagram QPs). In this case, the transaction sublayer must reject ULP transactions if it is out of resources and issue a “complete in error and continue” indication back to the ULP.

The flow control model permits per-connection, per-function and per-host allocation of the transmit and receive resource pools described in the previous section. Resource allocation can be oversubscribed. For example, the sum of per-connection credits can exceed the total

amount of transaction resources available within the transaction sublayer. To prevent buffer overrun when oversubscription is used, a set of global credit counters are used.

The unit of credit allocation for per-packet resources is 1 per transaction and resource type. The unit of credit allocation for per-buffer resources is N bytes where N is the minimum sized buffer that can be allocated to a transaction. If a transaction has a request length of L bytes, the number of buffer credits required by that transaction for the request itself is 1 and for the response data is ceiling(L/N).

The ULP must track resource allocation at the per-connection, per-function and per-host level using the following five credit counters given the above resource carving required for avoiding protocol deadlock:

- Request Tx packets (Req Pkt Tx) and Request Tx buffers (Req Buf Tx). These counters are used for Push and Pull Requests.
- Request Rx packets (Req Pkt Rx). This counter is used for Push and Pull Requests.
- Data Tx packets (Res Pkt Tx) and Data Tx buffers (Res Buf Tx). These counters are used for Pull Data.

These counters are initialized by the Connection Manager (CM) when the connection is configured. The ULP must decrement these counters depending on the transaction type as shown in the table below. When transaction resources are freed up, the transaction sublayer must return credits to the ULP.

Table 11: ULP Resource Counters

Transaction Type	ULP Req Pkt Tx	ULP Req Buf Tx	ULP Req Pkt Rx	ULP Data Pkt Tx	ULP Data Buf Tx
Push Data	1	ceiling(L/N)	1	0	0
Pull Request	1	1	1	0	0
Pull Response	0	0	0	1	ceiling(L/N)

8.2.4 Network Resource Management

The transaction sublayer must allocate resources from the network requests region for packets received from the network. The resources that must be allocated for packets received are shown in the table below. As with ULP resource management, the unit of credit allocation for per-packet resources is 1 per transaction and resource type. The unit of credit allocation for

per-buffer resources is N bytes where N is the minimum sized buffer that can be allocated to a transaction. If a transaction has a request length of L bytes, the number of request credits required by that transaction is 1 and the number of buffer credits required for the response data is ceiling(L/N).

Table 12: Network Resource Counters

Transaction Type	Net Req Pkt Rx	Net Buf Rx
Push Data	1	ceiling(L/N)
Pull Request	1	1
Pull Response	0 (Use Ulp Req Pkt Rx)	ceiling(L/N)

Additionally, a Falcon resource constrained implementation can always ensure HoL forward progress (UR Rule) by further carving the network pool into two resource occupancy zones - green and red (see figure below). Specifically, if the buffer occupancy crosses a threshold, Falcon enters the red zone and only accepts HoL packets. Otherwise, Falcon is in the green zone and accepts all packets (HoL or non-HoL).

For the network Rx buffer pool, the implementation should have separate red regions for requests and responses.

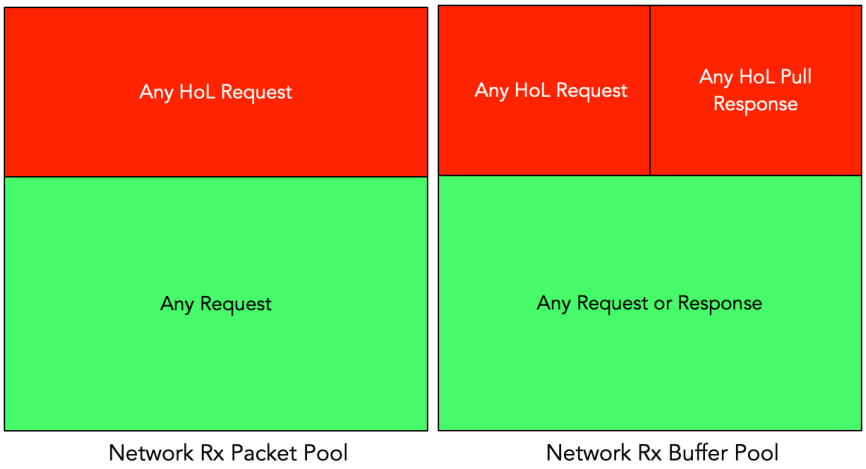


Figure 26: Network Pool Carving. (a) Packet and (b) Buffer

8.2.5 Connection-level Resource Carving

Falcon v1.1 introduces resource isolation between connections. Without isolation, a connection making slow progress will consume more hardware resources and for longer duration, depriving other fast connections of hardware resources and hindering their performance.

Falcon exerts fine-grained backpressure to ULPs via a per-connection Xon/Xoff flow control interface whenever a connection's resource usage exceeds a dynamically computed threshold. This enables Falcon to backpressure slow connections, preventing them from sending new transactions and consuming excess resources, allowing other faster connections to continue making progress.

Falcon calculates the backpressure thresholds dynamically and takes a similar approach to the Dynamic Thresholds (DT) algorithm that is commonly used to manage datacenter switch buffers. If the connection's current resource usage exceeds the dynamic threshold, it asserts Xoff for that connection to the ULP. Falcon calculates a connection's dynamic threshold for each resource as follows:

$$DynamicThreshold = \alpha \times FreeResource \quad (1)$$
$$FreeResource = TotalAllocation - Headroom - CurrentOccupancy \quad (2)$$

The parameter α (alpha) of a connection determines its resource limit relative to other connections, a larger value implying a larger share of the resources. Unlike DT, wherein α is static, RUE calculates $\alpha_{request}$ and $\alpha_{response}$ for a connection dynamically as follows:

$$\alpha_{request/response} = \beta_{request/response} \times \alpha_{base} \quad (3)$$

where $\beta_{request/response}$ is a congestion-aware variable proportional to fcwnd and ncwnd, but inversely proportional to fabric delay and target buffer occupancy. For example, in the case of fabric (host) congestion, if the fcwnd (ncwnd) is low and/or the fabric delay (buffer occupancy) is high, it indicates the connection is making slow progress and requires fewer resources. Falcon scales down α_{base} accordingly to decrease the connection's backpressure threshold relative to a congestion-free connection.

Falcon exerts separate Xoff signals for Request and Response transactions. Request Xoff should be calculated using $\alpha_{request}$, while response Xoff should use $\alpha_{response}$. Both of these α values are calculated in RUE and thus the RUE output format is augmented to include them. These values are expressed in fractional powers of 2 to simplify hardware implementation.

8.3 Connection Scheduler

The connection scheduler is responsible for transmit scheduling of packets generated by the transaction sublayer. The connection scheduler must implement a two level scheduling hierarchy: across transaction-type and across connections. The connection scheduler may implement QoS policies to provide prioritized handling of connections or transaction types. At a minimum, the connection scheduler must implement round-robin arbitration at both levels. Each connection has two queues that hold packets of a particular type:

- Pull Requests : This queue holds Pull requests and Push Data for both ordered and unordered connections. All packets in this queue are generated by the initiator.
- Pull Data: This queue holds the Pull data packets for both ordered and unordered connections. All packets in this queue are generated by the target.

The two queues are necessary to ensure that initiator generated packets and target generated packets are handled independently which is required to avoid protocol deadlock (CR Rule#2) The other level of arbitration selects a connection from the set of active connections. The connection scheduler may take into account additional implementation-defined constraints to determine whether a connection is eligible to transmit a packet. For example, the congestion control information from the packet delivery sublayer may be fed back into the connection scheduler to mask out connections who have exhausted their congestion window. A connection may also be masked out of the second level of arbitration if it is out of resources provisioned for it by the implementation. As another example, the connection scheduler may arbitrate across the three different packet types as shown in the figure below.

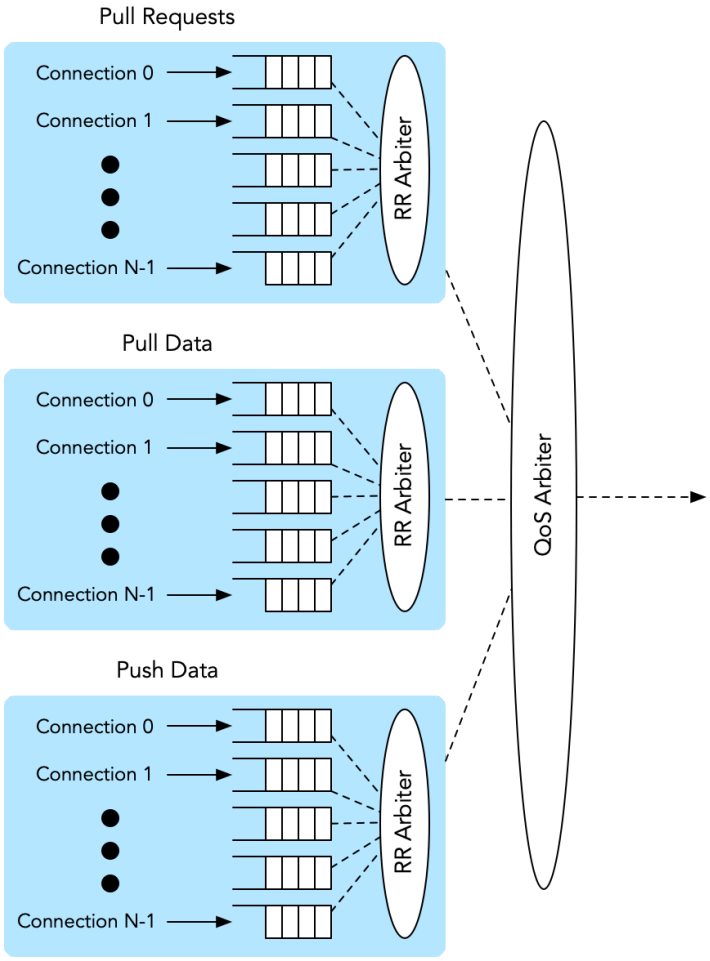


Figure 27: Connection Scheduler Architecture

8.4 Initiator Implementation

The following subsections describe the behavior of the initiator.

8.4.1 Initiator State

The initiator maintains the following per-connection variables that are updated dynamically during data transfer:

- Base Request Sequence Number (BRSN): This is the RSN value of the oldest transaction that is yet to be completed back to the ULP.
    - BRSN is initialized by the CM when the connection is established.
    - When a transaction is completed to the ULP with a RSN value of BRSN, BRSN must be updated to the next highest RSN of the transaction that has not yet been completed (using modulo 2^{32} arithmetic).

- Next Request Sequence Number (NRSN): This is the RSN value that must be assigned to the next transaction received from the ULP for this connection.
    - NRSN is initialized by the CM when the connection is established.
    - NRSN is incremented when a transaction is received from the ULP as follows: NRSN = (NRSN + 1) mod 2^{32}.

The initiator must maintain the following variables per transaction:

- Transaction State: This represents the current state of the transaction at the initiator. The possible values for the transaction state at the initiator are shown in the table below.

Table 13: Initiator Transaction States

Transaction Type	Initiator Transaction State	Description
Pull	PullReqUlpRx	Pull request transaction received from ULP.
	PullReqTx	Pull request packet sent to the target.
	PullReqAckd	Pull request packet ACKed by the target.
	PullDataRx	Pull data packet received from the target.
	PullDataUlpTx	Pull data delivered to the ULP.
Push	PushReqUlpRx	Push transaction received from ULP.
	PushDataTx	Push data packet sent to the target.
	PushDataAckd	Push data packet ACKed by the target.
	PushCplTx	Push completion delivered to the ULP.

8.4.2 Transaction Ordering

Apart from the ordering rules (UR Rule) required to avoid deadlocks for ordered connections, the initiator must return completions to the ULP in RSN order to match the ordering semantics expected by the ULP. Since Pull Data packet serves as an implicit completion for Pull transactions, the initiator must implement completion ordering by reordering received Pull Data packets with Push completion events generated by receiving ACKs for the Push Data packets.

There are no transaction ordering requirements for unordered connections.

8.4.3 Transaction Processing

The following subsections describe how different packet types are processed on the initiator side.

8.4.3.1 Pull Request

The initiator must process pull requests received from the ULP according to the following rules:

- The initiator must use the Source CID supplied by the ULP to lookup the connection context.
- The initiator must update the transaction state to PullReqUlpRx.
- The initiator must assign the RSN value to the pull request packet based on the NRSN value in the connection context and increment NRSN by one.
- The initiator must assign resources to the pull transaction as described in the resource management section.
- The pull request packet must be queued to the connection scheduler and scheduled based on congestion control criteria exposed by the packet delivery sublayer.
- After the packet is scheduled by the connection scheduler, the initiator must create the Falcon base header based on packet header fields from the connection context.
- The initiator must hand the pull request packet to the packet delivery layer.
- The initiator must update the transaction state to PullReqTx.
- When the target ACKs the pull request packet, the initiator must update the transaction state to PullReqAckd.

8.4.3.2 Pull Data

The initiator must process pull data packets received from the target according to the following rules:

- The initiator must use the Destination CID from the Falcon header to lookup the connection context.
- The initiator must check the RSN in the pull data packet against the RSNs of outstanding pull requests. If a matching pull request is not found, the pull data must be discarded.
- The initiator must check the request length of the matching pull request against the payload length of the pull data packet. If there is a mismatch, the pull data packet must be discarded.
- The initiator must allocate the Rx buffer resources for the pull data packet from the network region and use the Rx packet resource already allocated when the pull request was sent out. If Rx buffers are unavailable, the initiator must discard the pull data packet and generate a NACK event to the packet delivery sublayer.

- The initiator must update the transaction state to PullDataRx.
- The initiator must reorder the pull data packet based on RSN on an ordered connection.
- The initiator must deliver the pull data packet to the ULP. On an ordered connection, the pull data must be delivered to the ULP in RSN order.
- The initiator must update the transaction state to PullDataUlpTx.

8.4.3.3 Push Data

The initiator must process ULP push requests according to the following rules:

- The initiator must update the transaction state to PushReqUlpRx.
- The initiator must use the Source CID supplied by the ULP to lookup the connection context.
- The initiator must assign the RSN value to the push data packet based on the NRSN value in the connection context and increment NRSN by one.
- The initiator must assign resources to the push transaction as described in the resource management section.
- The push data packet must be queued to the connection scheduler.
- After the packet is scheduled by the connection scheduler, the initiator must create the Falcon base header based on packet header fields from the connection context.
- The initiator must hand the push data packet to the packet delivery layer.
- The initiator must update the transaction state to PushDataTx.
- When the target ACKs the push data packet, the initiator must update the transaction state to PushDataAckd.
- The initiator must deliver a push completion to the ULP. On an ordered connection, the push completion must be delivered to the ULP in RSN order.
- The initiator must update the transaction state to PushCplTx.

8.4.3.4 RNR NACK

Receiver-Not-Ready (RNR) NACKs are sent by the target to the initiator to request a delayed retransmission of push data packets. Push data transmission will be delayed by the amount, RNR timeout, which is specified in the NACK packet. RNR NACKs for pull transactions are handled locally on the target side as described later. The initiator must process RNR NACK received from the network for a push transaction according to the following rules:

- The initiator must use the RSN delivered to it by the packet delivery layer along with the RNR NACK indication to identify the push transaction.
- The initiator must signal to the packet delivery layer to adjust the retransmit timeout for the push data packet associated with the push transaction to the RNR NACK timeout value.

8.4.3.5 CIE NACK

Complete-in-Error (CIE) NACKs are sent by the target to the initiator to indicate that a push transaction encountered an error and could not be completed. The initiator must process the CIE NACK received from the network according to the following rules:

- The initiator must use the RSN delivered to it by the packet delivery layer along with the CIE NACK indication to identify the push transaction.
- The initiator must trigger the sending of a Resync packet by the packet delivery sublayer to advance the receiver base PSN.
- The initiator must deliver a push completion to the ULP with a complete-in-error indication and include the ULP NACK code received from the packet delivery layer. On an ordered connection, the push completion must be delivered to the ULP in RSN order.

8.5 Target Implementation

The following subsections describe the behavior of the target.

8.5.1 Target State

The target maintains the following per-connection variables that are updated dynamically during data transfer:

- Base Request Sequence Number (BRSN): This is the RSN value of the oldest transaction that has been delivered to the ULP but not yet acknowledged by the ULP.
    - BRSN is initialized by the CM when the connection is established.
    - When a transaction is acknowledged by the ULP with a RSN value of BRSN, BRSN must be updated to the next highest RSN of the transaction that has not yet been acknowledged (using modulo 2^{32} arithmetic).
- Next Request Sequence Number (NRSN): This is the RSN value that must be delivered to the ULP.
    - NRSN is initialized by the CM when the connection is established.
    - NRSN is incremented when a transaction with RSN equal to NRSN is delivered to the ULP as follows:
      NRSN = (NRSN + 1) mod 2^{32}.

The target must maintain the following variables per transaction:

- Transaction State: This represents the current state of the transaction at the target. The possible values for the transaction state at the target are shown in the table below.

Table 14: Target Transaction States

Transaction Type	Target Transaction State	Description
Pull	PullReqRx	Pull request packet received from the network.
	PullReqUlpTx	Pull request transaction delivered to the ULP.
	PullReqUlpAckd	Pull request ACKed by the ULP.
	PullDataUlpRx	Pull data received from the ULP.
	PullDataTx	Pull data packet sent to the initiator.
	PullDataAckd	Pull data packet ACKed by the initiator.
Push	PushDataRx	Push data packet received from the network.
	PushReqUlpTx	Push request transaction delivered to the ULP.
	PushReqUlpAckd	Push request ACKed by the ULP.

8.5.2 Transaction Ordering

To match the ordering semantics expected by the ULP, the target must maintain transaction ordering on an ordered connection according the following rules:

- The target must deliver requests received from the initiator to the ULP in RSN order. The target must implement this request ordering by reordering received Push Data with received Pull Requests.

There are no transaction ordering requirements for unordered connections.

8.5.3 Transaction Processing

The following subsections describe how different packet types are processed on the target side.

8.5.3.1 Pull Request

The target must process pull request packets received from the initiator according to the following rules:

- The target must use the Destination CID from the Falcon header to lookup the connection context.
- The target must allocate the Rx resources for the pull request packet from the network requests region. If Rx resources are unavailable, the target must discard the pull request packet and generate a NACK event to the packet delivery sublayer.
- The target must update the transaction state to PullReqRx.
- The target must reorder the pull request packet based on RSN on an ordered connection.
- The target must deliver the pull request packet to the ULP. On an ordered connection, the pull request must be delivered to the ULP in RSN order.
- The target must update the transaction state to PullReqUlpTx.
- When the ULP acknowledges the pull request, the target must update the transaction state to PullReqUlpAckd.

8.5.3.2 Pull Data

The target must process pull data packets received from the ULP according to the following rules:

- The target must update the transaction state to PullDataUlpRx.
- The target must use the Source CID supplied by the ULP to lookup the connection context.
- The target must assign the RSN value to the pull data packet based on the RSN value of the corresponding pull request that was delivered to ULP.
- The target must assign resources to the pull data packet as described in the resource management section.
- The pull data packet must be queued to the connection scheduler.
- After the packet is scheduled by the connection scheduler, the target must create the Falcon base header based on packet header fields from the connection context.
- The target must hand the pull request packet to the packet delivery layer.
- The target must update the transaction state to PullDataTx.
- When the initiator ACKs the pull data packet, the target must update the transaction state to PullDataAckd.

8.5.3.3 Push Data

The target must process push data packets received from the initiator according to the following rules:

- The target must use the Destination CID from the Falcon header to lookup the connection context.

- The target must allocate the Rx resources for the push data packet from the network requests region. If Rx resources are unavailable, the target must discard the push data packet and generate a NACK event to the packet delivery sublayer.
- The target must update the transaction state to PushDataRx.
- The target must reorder the push data packet based on RSN on an ordered connection.
- The target must deliver the push data packet to the ULP. On an ordered connection, the push data packet must be delivered to the ULP in RSN order.
- The target must update the transaction state to PushReqUlpTx.
- When the ULP acknowledges the push data packet, the target must update the transaction state to PushReqUlpAckd.
- The target must trigger an ACK generation from the packet delivery sublayer for the push data packet.

8.5.3.4 RNR NACK

The ULP may respond to a push or a pull transaction delivered to it by the target with a Receiver-Not-Ready (RNR) NACK. Along with the RNR NACK response, the ULP provides a timeout value that specifies when the transaction must be retransmitted to the ULP. The handling of RNR NACK by the target depends on the type of the transaction. Since pull requests are already acknowledged back to the initiator, pull transactions must be retransmitted from the target side. On the other hand, push data packets are not acknowledged back to the initiator until they are acknowledged by the ULP. Hence, push transactions must be retransmitted from the initiator side. On an ordered connection, if a transaction with RSN value of N is RNR NACKed by the ULP, all subsequent transactions (with RSN greater than N) continue to be RNR NACKed until the ULP accepts the retransmitted transaction with RSN value of N.

The target must process RNR NACK indication from the ULP for a pull transaction according to the following rules:

- The target must look up the pull transaction corresponding to the RNR NACK response from the ULP based on the RSN.
- The target must initialize a retransmit timer for the pull transaction based on the RNR NACK timeout value.
- When the retransmit timer expires, the target must redeliver the pull request to the ULP.
- When the ULP acknowledges the pull request, the target must update the transaction state to PullReqUlpAckd.

The target must process RNR NACK indication from the ULP for a push transaction according to the following rules:

- The target must look up the push transaction corresponding to the RNR NACK response from the ULP based on the RSN.
- The target must generate a RNR NACK event and deliver it to the packet delivery layer triggering the sending of a NACK packet to the initiator.

8.5.3.5 CIE NACK

The ULP may respond to a push transaction delivered to it by the target with a Complete-In-Error (CIE) NACK. Pull transactions that must complete in error are signaled to the initiator by the ULP on the target side sending a zero-length pull data packet. For a push transaction, along with the CIE NACK response, the ULP provides a NACK code that specifies the reason why the transaction is being completed in error.

The target must process CIE NACK indication from the ULP for a push transaction according to the following rules:

- The target must look up the push transaction corresponding to the CIE NACK response from the ULP based on the RSN.
- The target must generate a CIE NACK event and deliver it to the packet delivery layer triggering the sending of a NACK packet to the initiator.

9. Packet Delivery Sublayer

The main responsibilities of the packet delivery sublayer are reliable packet delivery, congestion control and multipath load balancing.

The packet delivery sublayer implements two sliding windows to separate out requests and data. The request sliding window is responsible for reliable delivery of Pull Request, while the data sliding window is responsible for the reliable delivery of Push Data and Pull Data. Having the Pull Request and Pull Data in separate sliding windows is important for resolving deadlocks – CR Rule#2.

Since datacenter fabrics do not guarantee packet ordering, packets may arrive at the receiver out of order. The receiver maintains its own version of the sliding window. If a packet is received with a PSN that falls within the receiver's sliding window (greater than or equal to BSN and less than or equal to BSN + Rx Window Size), the packet is accepted and delivered to the transaction sublayer of the connection. The receiver then sends an ACK back to the transmitter.

Because Falcon covers reliable delivery across not only the fabric, but also the ULP which could also NACK packets, each packet has 2 states: received and acknowledged. All packets are

received when they reach the receiver and fall in the receiver's sliding window. But when they are acknowledged depends on the packet types: for a Push Data packet, the receiver waits for ULP acknowledgement; for all other packet types, the receiver marks a packet as acknowledged immediately.

Therefore, the receiver maintains 3 bitmaps – 2 for the data window received and acknowledged states, and 1 for the request window because its received and acknowledged states are identical. The bit is set to 1 if the packet is received/acknowledged and to 0 otherwise. The size of the receiver request and data sliding windows is 512, which is sized for the extent of out of order packets that can be handled by the receiver.

The receiver sends ACK packets to the transmitter, to convey the received and acknowledged states. The ACK is normally BACK to save bandwidth, and is only EACK when the 3 bitmaps have useful information. The receiver is permitted to coalesce ACKs. That is, when multiple packets are received, the receiver is permitted to send a single ACK packet with the latest snapshot of bitmaps. The receiver is also permitted to piggyback ACKs onto packets flowing in the opposite direction. The combination of ACK coalescing and ACK piggybacking helps to keep the count of ACK packets down and reduce the network bandwidth consumed by ACK packets. With multipathing, while an ACK packet's header and congestion control metadata correspond to a specific path, its acknowledgment bitmaps are cumulative, covering all packets received across all paths within the connection. Note that the sizes of the req-bitmap, data-ack-bitmap, and data-rx-bitmap fields carried in EACKs remain as specified in EACK Packet format.

The transmitter sliding window is configurable and must be sized considering the network RTT for the connection. The transmitter must also keep a bitmap of all packets within each sliding window where a bit is set to 1 if that packet was acknowledged by the receiver. Since packets may be lost in transit, the transmitter must retain packets (or packet header and metadata necessary to construct the full packet) for possible retransmission until they are acknowledged by the receiver.

In Falcon, there are two primary retransmission mechanisms. Most packets should be retransmitted very quickly (within one or a few RTTs) by Falcon's early retransmission. The transmitter utilizes one of two heuristics to infer which packets are likely lost and retransmit them immediately upon bitmap updates: the first is based on out-of-order information from the bitmaps and receiver out-of-window drop notifications, and the second is the RACK-TLP algorithm. Recent Acknowledgement (RACK) retransmits quickly using time-based inferences derived from ACK feedback, and Tail Loss Probe (TLP) sends a probe packet to trigger ACK feedback to avoid retransmission timeouts. Implementations must support both the out-of-order distance heuristic and the RACK-TLP algorithm to drive early retransmissions, providing the flexibility to select either method at initialization time for the entire system.

Where the early retransmission heuristics fail to recover a loss, the backup timeout mechanism kicks in. The transmitter maintains a retransmit timer for every packet within its sliding window. The timer is started (reset) when the packet is transmitted (retransmitted) and is cleared if the packet is ACKed by the receiver. If the timer expires (based on a configurable retransmit timeout), the transmitter must retransmit that packet with the same PSN value as the original transmission.

ACK packets sent by the receiver may also be lost in transit. The transmitter recovers from this scenario when it receives a subsequent ACK from the receiver or when the retransmit timer fires for packets that were supposed to be ACKed by the lost ACK packet.

The following sections detail the behavior of the packet delivery sublayer in terms of the transmitter and receiver behaviors. Each end of a Falcon connection is simultaneously both a transmitter and a receiver.

9.1 Transmitter Implementation

The following subsections detail the behavior of the transmitter.

9.1.1 Transmitter State

The transmitter maintains the following per-connection variables that are updated dynamically during data transfer:

- Request Window Base PSN (RBPSN): This is the PSN value of the oldest packet in the request sliding window that is yet to be acknowledged by the receiver.
    - RBPSN is initialized by the connection manager (CM) when the connection is established.
    - When an ACK is received that acknowledges the receipt of a request sliding window packet with PSN value of RBPSN, RBPSN must be updated to the next highest PSN of the packet that has not been acknowledged (using modulo 2^{32} arithmetic). When RBPSN is updated, the bits in the sequence number bitmap corresponding to packets from the previous RBPSN to the updated RBPSN must be cleared.
- Request Window Next PSN (RNPSN): This is the PSN value that must be assigned to the next packet in the request sliding window transmitted on the connection.
    - RNPSN is initialized to RBPSN by the CM when the connection is established.
    - RNPSN is incremented when a packet is transmitted as follows: RNPSN = (RNPSN + 1) mod 2^{32}.
- Data Window Base PSN (DBPSN): This is the PSN value of the oldest packet in the data sliding window that is yet to be acknowledged by the receiver.

    - DBPSN is initialized by the CM when the connection is established.
    - When an ACK is received that acknowledges the receipt of a data sliding window packet with PSN value of DBPSN, DBPSN must be updated to the next highest PSN of the packet that has not been acknowledged (using modulo 2^{32} arithmetic). When DBPSN is updated, the bits in the sequence number bitmap corresponding to packets from the previous DBPSN to the updated DBPSN must be cleared.
- Data Window Next PSN (DNPSN): This is the PSN value that must be assigned to the next packet in the data sliding window transmitted on the connection.
    - DNPSN is initialized to DBPSN by the CM when the connection is established.
    - DNPSN is incremented when a packet is transmitted as follows: DNPSN = (DNPSN + 1) mod 2^{32}.
- Request Window Sequence Number Bitmap: A bitmap of the same width as the req-bitmap field of the EACK packet, where each bit represents a packet within the request sliding window. The bit is set to 1 if the packet has been ACKed. Otherwise the bit is 0. Bit 0 represents a PSN value of RBPSN and bit n represents a PSN value of (RBPSN+n).
    - This bitmap is initialized to all 0s when the connection is configured.
    - When an ACK is received, this bitmap is updated based on the receiver sequence number bitmap contained in the ACK packet.
- Data Window Sequence Number ACK Bitmap: A bitmap of the same width as the data-ack-bitmap field of the EACK packet, where each bit represents a packet within the data sliding window. The bit is set to 1 if the packet has been ACKed. Otherwise the bit is 0. Bit 0 represents a PSN value of DBPSN and bit n represents a PSN value of (DBPSN+n).
    - This bitmap is initialized to all 0s when the connection is configured.
    - When an ACK is received, this bitmap is updated based on the data-ack-bitmap field contained in the ACK packet.
- Data Window Sequence Number Receive Bitmap: A bitmap of the same width as the data-rx-bitmap field of the EACK packet, where each bit represents a packet within the data sliding window. The bit is set to 1 if the packet has been received. Otherwise the bit is 0. Bit 0 represents a PSN value of DBPSN and bit n represents a PSN value of (DBPSN+n).
    - This bitmap is initialized to all 0s when the connection is configured.
    - When an ACK is received, this bitmap is updated based on the data-rx-bitmap field contained in the ACK packet.
- NIC Congestion Window (ncwnd): This represents the number of packets that start new ULP transactions (Pull Request or Push Data) that the transmitter is permitted to send based on congestion at the receiving NIC, per sliding window. The ncwnd is computed by the RUE dynamically based on receiver buffer occupancy.

- Fabric Congestion Window (fcwnd): This represents the number of packets (any types of reliable packets) that the transmitter is permitted to send based on fabric congestion, per sliding window. The fcwnd is computed by the RUE dynamically based on congestion in the fabric. For a multipath connection, fcwnd is the summation of per-flow fcwnds.
- Request Window Outstanding Requests Counter (RORC): This is a count of the number of Pull Request packets that have been sent by the transmitter but have not yet been acknowledged by the receiver.
- Data Window Outstanding Requests Counter (DORC): This is a count of the number of Push Data packets that have been sent by the transmitter but have not yet been acknowledged by the receiver.
- Retransmit Timeout (RTO): This timeout value is used to initiate retransmission of packets to recover from lost packets or ACKs. When the retransmit timer for a packet matches this timeout value the packet must be retransmitted.
    - RTO must be initialized by the CM based on expected round-trip latencies for the connection through the fabric.
    - RTO may be dynamically adjusted by the RUE based on congestion in the network.
- RACK.xmit_ts: This variable represents the most recent transmit timestamp of an acknowledged packet, updated whenever an ACK, EACK, or NACK is received. This variable is used by the RACK heuristic to early retransmit packets.

The transmitter maintains the following variable on a per-flow basis:

- ACK Coalescing Timer: This timer determines when a coalesced ACK must be transmitted according to the following rules:
    - The timer must be started when a packet corresponding to a path is received and falls within or beyond the receiver's sliding window or when a packet is acknowledged by ULP.
    - The timer must be reset when an ACK is sent.
    - The timer must be reset when another packet piggybacks all useful acknowledged and received states (when bitmaps are all zero).
    - Implementations must support 4 timers to support a multipathing degree of 4.
- RACK RTO: This timeout value is a dynamically calculated reordering offset provided by Rate Update Engine (RUE). A packet is retransmitted if its elapsed time exceeds this value.
- TLP PTO: The probe timeout value is a dynamically calculated value by RUE which determines when probes need to be sent after a period of inactivity.

The transmitter maintains the following variables per packet:

- Retransmit Timer: This timer determines when a retransmission attempt must be made. The timer must be started when a packet is transmitted. When the timer expires, the packet must be retransmitted and the timer must be restarted.
- Timeout-based Retransmit Counter: This counter keeps track of the number of times the packet has been timeout-retransmitted. The retransmit counter should not include early retransmissions.
- Early Retransmission Counter: This counter keeps track of the number of times the packet has been retransmitted via early retransmissions.
- Associated Path: This variable keeps track of the path taken by the packet. Path taken is identified by the lower 2 bits of the flow label associated with the packet. We refer to the lower 2 bits of the flow label as the path-id in the rest of the specification. With multipathing, the flow labels are constrained so that each path has a unique path-id. That is, the 2 LSBs for the 4 flow labels must be 00b, 01b, 10b and 11b. This allows the receiver to map a packet to its path-id by extracting the path-id from the flow label.

The following variables are initialized once when the connection is configured and do not change during the lifetime of the connection:

- Max Timeout-Based Retransmit Attempts: This variable determines how many times a packet may be timeout-based retransmitted. When the retransmit counter for a packet matches this value, a fatal error is signaled to the CM and will result in teardown of the connection.
- Max Early Retransmit Attempts: This variable determines how many times a packet may be early retransmitted. When the retransmit counter for a packet matches this value, a fatal error is signaled to the CM and will result in teardown of the connection.
- ACK Coalescing Timeout: This timeout value determines the maximum amount of time the transmitter waits for coalescing before transmitting an ACK. When the ACK coalescing timer value equals this timeout value, an ACK is sent on the connection. If the ACK Coalescing timeout is too long relative to the RACK timeout, the transmitter might prematurely conclude that a packet is lost simply because the acknowledgment is being held in the receiver's coalescing buffer. To prevent these spurious retransmissions, the ACK Coalescing timeout is usually kept small (microsecond level) to ensure that RACK_RTO can remain aggressive enough to trigger retransmissions within a few RTTs.
- ACK Request Percent: This percent value determines the fraction of packets that have the AR bit marked.
- Out-of-order Distance Threshold: This threshold determines how aggressive the early retransmission is. The transmitter can early-retransmit a non-received PSN if in the same sliding window there is a received PSN, corresponding to the same path, that is sufficiently larger – this threshold defines how much larger is sufficient. The threshold in essence is used as a buffer to disambiguate a packet loss versus reordering in the

network. Packets reordered in the network within the Out-of-order Distance Threshold are not interpreted to be lost in the network.

9.1.2 Congestion Control Tx Gating Function

The mental model for congestion control gating function is simple: a packet is eligible for transmission so long as Next Packet Sequence Number is lower than the Base Sequence Number + fcwnd. This guarantees that there are at most an fcwnd number of packets at any point in time.

All initial packet transmissions are subject to the following congestion control gating function at the transmitter depending on the packet type:

- Pull Request:
  PSN < RBPSN + fcwnd and
  RORC < ncwnd
- Pull Data:
  PSN < DBPSN + fcwnd
  DORC < ncwnd
- Push Data:
  PSN < DBPSN + fcwnd and
  DORC < ncwnd

All packet retransmissions are subject to the following congestion control gating function at the transmitter depending on the packet type. The gating is explained using another variable Outstanding Retransmitted Request Count (ORRC) per sliding window (RORRC counts retransmitted Pull Requests and DORRC counts retransmitted Push Data):

- Pull Request:
  PSN < RBPSN + fcwnd
  RORRC < ncwnd
- Pull Data:
  PSN < DBPSN + fcwnd
  DORRC < ncwnd
- Push Data:
  PSN < DBPSN + fcwnd
  DORRC < ncwnd
- [R/D]ORRC is maintained as follows:
  - [R/D]ORRC is incremented by 1 when retransmitted request sent out
  - [R/D]ORRC is decremented by 1 when a retransmitted request is ACKed

- [R/D]ORRC is decremented by 1 when a retransmitted request becomes eligible for retransmission again (becoming eligible for retransmission implies a retransmission at a later time). When the request is retransmitted again the [R/D]ORRC is incremented again at the time the retransmitted request is sent out.

For a multipath Falcon connection, fcwnd above is the summation of the per-flow fcwnds.

9.1.3 Packet Transmission

The transmitter must follow the rules given below when it receives a packet transmit request from the transaction sublayer:

- The transmitter must set the PSN of the packet to NSN of request or data sliding window depending on the packet type and then increment NSN as described earlier.
- The transmitter must determine whether the packet is eligible for transmission using the Tx gating functions.
- On an ordered connection, the transmitter must transmit the packet in RSN order across both the request and data sliding windows. Within each sliding window, the transmitter must transmit the packet in PSN order.
- After the packet is transmitted, it must be buffered by the transmitter until it is acknowledged by the receiver.
- The transmitter must start the retransmit timer after the packet is transmitted.

9.1.3.1 Ack Req (AR) generation

The transmitter can decide when to mark the AR bit in the header, to request the receiver to generate an ACK. The transmitter decides the frequency of the AR bit for congestion control purposes.

For example, as a guideline of implementation, AR bit needs to be more frequent when congestion window is lower, to ensure timely congestion control actions. Specifically, when the fcwnd is smaller than or equal to the ACK Request Fcwnd Threshold, every packets' AR bit is marked; when fcwnd is larger than the ACK Request Fcwnd Threshold, a ACK-Request-Percent fraction of packets' AR bit is marked, to make sure CC reaction frequency is sufficient.

9.1.3.2 Path Association

With multipathing enabled, the transmitter datapath associates each outgoing packet with a specific path (or flow) by assigning a 20-bit flow label from an ordered list (flow_label[3:0]) provided by the Rate Update Engine (RUE). The selection is performed dynamically by the Max-Open-Window scheduler located in the transmitter datapath of the Packet Delivery

Sublayer that establishes an ACK-clocked, line-rate closed-loop mechanism that continuously steers traffic to the paths possessing the highest instantaneous capacity.

The scheduler resides within the Packet Delivery Sublayer datapath and interfaces with the RUE via a pair of Event/Response queues. Congestion signals measured by the datapath, including RTT timestamps, are tagged with the packet's 2-bit path-id and posted to the RUE via the CC Event Queue. In response, the RUE executes congestion control computations per flow and delivers updated parameters through the CC Result Queue. For each flow update, the RUE provides the target flow_id (0 to 3), the integer and fractional congestion window components (fcwnd_integer and fcwnd_fraction), the connection-level cwnd_carryover, and the active flow labels (flow_label[3:0]) alongside their validity bitmask (flow_label_valid[3:0]). In contrast to Falcon 1.1.0 open-loop scheduler, the datapath directly tracks each path's congestion window and executes path arbitration autonomously in hardware.

The scheduler operates on a closed-loop credit principle driven by the instantaneous available capacity of each path. When an outgoing packet is cleared for transmission across either the request or data sliding window by the Tx Gating Function, the scheduler computes the instantaneous open window for each valid flow as:

open_fcwnd[i] = flow_fcwnd[i] – outstanding_packets[i]

where outstanding_packets[i] tracks the number of unacknowledged packets currently in flight on path i. Flows with open_fcwnd[i] ≤ 0 have fully saturated their allocated window and are masked out from scheduling. If the maximum available capacity across all valid flows (max_open_fcwnd = max_i(open_fcwnd[i])) is zero or negative, no path has available credit and transmission is deferred until an incoming ACK arrives or the RUE expands a window.

To ensure a balanced path utilization under uncongested conditions, the scheduler incorporates threshold detection into its arbitration logic. A policy that strictly selects max(open_fcwnd) creates an unintended positive feedback loop under light-to-moderate offered loads: a flow that is selected and quickly acknowledged receives an additive increase, granting it a slightly larger open window (even by just 1 packet) and causing the scheduler to pick it repeatedly while other paths starve. Because a difference of a single packet prevents a tie from being recognized under strict equality, the scheduler relaxes the tie condition using a configurable parameter, tie_breaking_threshold, that is defined as relative fraction of max_open_fcwnd). All valid flows whose available window is strictly positive and within the threshold of the maximum—satisfying open_fcwnd[i] > 0 and open_fcwnd[i] ≥ (max_open_fcwnd – tie_breaking_threshold)—are admitted into a candidate tied set. By combining maximum-open-window with threshold-relaxed round-robin tie-breaking, the scheduler preserves balanced flow distribution even under light load.

9.1.4 Early Retransmission

The transmitter can retransmit packets when it receives an EACK from the receiver. The bitmap information, the timestamp and out-of-window notifications of EACK can help the transmitter decide which packets are likely dropped and should be retransmitted. The policies of early retransmission are specified below. For multipath connections, this heuristic is applied on a per-path basis.

OOO-distance heuristic: Once receiving an EACK, the transmitter can infer which packets are likely lost based on the bitmaps. If some PSNs are not received but larger PSNs are received, these PSN are either lost or reordered in the fabric. To disambiguate the losses from reordering, one can use the following intuition: if a PSN is not received but a sufficiently larger PSN is received, this PSN is likely lost. The out-of-order distance threshold (ooo_threshold) defines how much larger is sufficient. For multipath connections, this heuristic is applied on a per-flow basis.

Recency check: We must guarantee at least an RTT has passed since the last (re)transmission before retransmitting a packet, to give enough time for the last (re)transmission to be acknowledged. Otherwise, a lost packet may be spuriously retransmitted multiple times, once per EACK, before the retransmission getting acknowledged.

9.1.4.1 EACK Algorithm

The following algorithm implements the above policies.

```
for bitmap in [request-bitmap, data-rx-bitmap]
  psn_high = highest '1' PSN in bitmap
  for all '0' in bitmap whose PSN < psn_high - ooo_threshold
    if the last transmission time + rtt < now: // Recency check
      Retransmit this PSN
```

Early retransmission uses data-rx-bitmap, not data-ack-bitmap, because early retransmission is responsible for recovering non-ULP drops. ULP NACKed packets have their own recovery approaches. With multipathing, EACK is applied on a per-flow basis when scanning the bitmaps. For this, the stored path-id for each packet is used and the PSN distance is computed per path-id.

OWN heuristic: Recall that Out of Window Notification (OWN) is a field in Base ACK indicating if the receiver dropped a packet due to lack of space in its data or request sliding window. On receiving an EACK with OWN bit set (henceforth known as EACK-OWN), the transmitter retransmits packets within and beyond the received bitmap that have not been retransmitted

recently, within an RTT. The following algorithm at the Falcon transmitter implements this heuristic for a given window (request or data) which receives EACK-OWN.

```
// Loop over transmit sliding window worth packets. Includes
packets within and beyond the receive sliding window.
for psn in [bpsn, bpsn + tx_window_size]:
    bitmap_end_psn = highest PSN in bitmap
    // Do not retransmit packets if already received at target.
    if psn <= bitmap_end_psn and bitmap[psn] == 1:
        continue;
    packet = packets_metadata[psn] //holds packet metadata
    if packet is not retransmitted recently within an RTT:
      Retransmit this packet
```

On receiving an EACK-OWN, the intuition of the above heuristic is to retransmit out-of-window packets, i.e., packets dropped to the right of the sliding window, at most once every RTT, while leveraging the bitmap to avoid spuriously retransmitting packets which have already been received at the target.

For multipath connections, retransmissions use the same path-id as used during the initial transmission. However, the remaining bits of the flow-label can be changed.

9.1.4.2 Recent Acknowledgement

Recent Acknowledgment (RACK) is a time-based early retransmission strategy that detects packet loss using chronological timing rather than sequence counts. When an EACK is received, the sender processes the accompanying Rx bitmap and T1 timestamp as follows:

- Scope of scanning: RACK evaluates only unreceived packets that fall within the range of the received Rx bitmap, as the status of packets beyond this horizon remains unknown to the sender. Packets already marked as received within this range are excluded from further loss analysis.
- Recency Check: To prevent premature retransmissions, the sender only considers outstanding packets transmitted at or before RACK.xmit_ts which is the transmit time of the most recently acknowledged segment; any packets sent after this time are considered too fresh to be retransmitted and are ignored.
- Inference of Loss or Reordering: For eligible packets, the sender evaluates the time elapsed since their original transmission – If a packet's elapsed time equals or exceeds the RACK_RTO, it is marked as lost and retransmitted immediately. Otherwise, the packet is treated as potentially reordered, and this additional 'time cushion' of RACK_RTO allows for its delayed arrival without triggering a premature retransmission.

```
for window in {request_window, data_window}:
  window.rack_psn_limit =
    max (window.rack_psn_limit,
      EACK.{R/D}BPSN + {R/D}_ACK_BITMAP_SIZE)
  for packet in window.outstanding_packets:
    // Consider packets only within the RACK PSN limit.
    if packet.psn <= window.rack_psn_limit:
      // Consider not received packets that pass the recency check.
      if packet is not received and packet.tx_time <= RACK.xmit_ts:
        if packet.elapsed_time >= rack_rto:
          Retransmit this packet
        else:
          Setup retransmit timer with expiry at rack_rto
```

RACK.xmit_ts is updated on receiving an ACK, EACK or NACK:

```
RACK.xmit_ts = max(RACK.xmit_ts, {E/N}ACK.T1)
```

RACK_RTO is calculated by the Rate Update Engine as follows on ACK, EACK or NACK:

```
rack_rto = (ACK.T4 - ACK.T1) +
               (smoothed_rtt * reordering_window_rtt_factor)
```

9.1.4.3 Tail Loss Probe

Tail Loss Probe (TLP) enables early recovery from tail losses. After a period of inactivity, e.g., due to tail losses or lost ACKs, the sender retransmits a probe packet per window. The probe prompts the receiver to generate an ACK. On receiving this ACK, the sender uses the RACK heuristic to trigger early retransmissions if warranted. The lowest unacknowledged PSN packet in each window is sent as a probe. Probe is sent out when the period of inactivity (no packets transmitted or received) exceeds the TLP timeout tlp_pto. The EACK Request (ER) bit and ACK Request (AR) bit in the Falcon Base Header is set in outgoing TLP.

tlp_pto is calculated by the Rate Update Engine as follows on:

```
tlp_pto = max((smoothed_rtt * tlp_pto_rtt_factor), min_tlp_pto)
```

Probe is sent out when the period of inactivity (no packets transmitted or received) exceeds the TLP timeout tlp_pto.

9.1.5 Timeout Based Retransmission

The transmitter must process the retransmit timer expiry events according to the following rules:

- The retransmit counter for the packet must be incremented. If the retransmit counter exceeds the maximum retransmit attempts, the transmitter must signal a fatal error on the connection to the CM.
- The transmitter must determine whether the packet is eligible for retransmission using the Tx gating functions described here.
- The transmitter must retransmit the packet with the same PSN and path-id as the original packet.
- On an ordered connection, the transmitter must retransmit the packet in RSN order across both request and data sliding windows. Within each sliding window, the transmitter must retransmit the packet in PSN order.
- If an ACK packet is received in the middle of a retransmission, the transmitter is permitted to complete the transmission with no regard to the received ACK packet.
- The transmitter must start the retransmit timer after the packet is retransmitted.

9.1.6 ACK Generation

The packet delivery sublayer supports timer-based coalescing of ACKs to reduce the overhead of ACK packets. With multipathing, ACK coalescing is done per path. ACKs can also be piggy backed onto packets flowing in the reverse direction. The transmitter must implement ACK generation according to the following rules:

- When receiving a packet without the AR bit set, the transmitter must start the ACK coalescing timer corresponding to the packet's path, if it has not already started.
- When receiving a packet with the AR bit set, the transmitter must send an ACK packet when the packet is acknowledged (when ULP acknowledges for PushData and when received for other packets); and the transmitter must stop the ACK coalescing timer corresponding to the packet's path, if the timer has started.
- If the transmitter has an outgoing packet, the transmitter must advance the RBPSN and DBPSN to the latest value and piggyback them onto the outgoing packet. The transmitter must stop the ACK coalescing timer if
    - Receiver Request Sequence Number Bitmap is zero, and
    - Receiver Data Sequence Number ACK Bitmap is zero, and
    - Receiver Data Sequence Number Rx Bitmap is zero, and
    - Receiver R-OWN and D-OWN bits are zero.

- When the ACK coalescing timer corresponding to a path equals the ACK coalescing timeout value, the transmitter must send a coalesced ACK packet and stop the ACK coalescing timer.
- The ACK packet must include the latest values of the receiver's RBPSN and DBPSN.
- The ACK packet must include the latest values of t1 and t2 maintained by the receiver for RTT measurement as described in a later section. These values are maintained on a per-flow basis.
- The ACK packet must include the latest snapshot of the congestion control metadata from the transmitter state. The congestion control metadata is maintained on a per-flow basis.
- The path-id corresponding to the received packet is extracted as the two LSBs from the incoming packet's flow label. The ACK packet uses the same flow label as the corresponding path and so the flow label of the incoming packet is also stored.

Whether the ACK should be a BACK or an EACK depends on the bitmaps and ER bit:

- If the EACK Request bit is set, EACK must be sent.
- If the Receiver Request Sequence Number Bitmap is non-zero, EACK must be sent.
- If the Receiver Data Sequence Number ACK Bitmap is non-zero, EACK must be sent.
- If the Receiver Data Sequence Number Rx Bitmap has missing PSNs ('1's are non-consecutive from LSB), EACK must be sent.
- If the Request Out-of-Window Notification (R-OWN) or Data Out-of-Window Notification (D-OWN) bits are set, then EACK must be sent.
- Otherwise, BACK can be sent.
    - This is not strict – it is always legitimate to send EACK.

9.1.7 NACK Generation

NACKs may be generated in response to NACK events generated by the transaction sublayer or to NACK events generated by the sliding window receiver. The transmitter must implement NACK generation according to the following rules:

- The NACK packet must include the latest values of the receiver's RBPSN and DBPSN.
- The NACK packet must include the PSN of the received packet that is triggering the NACK event and an indication of whether the packet is in the request sliding window or the data sliding window.
- For NACK events generated by the transaction sublayer, the NACK packet must include the ULP NACK code and the RNR NACK timeout if appropriate.
- The NACK packet must include the latest values of t1 and t2 maintained by the receiver for RTT measurement as described in a later section. As with ACK, these values are maintained per-flow and the NACK carries this data for the corresponding path.

- The NACK packet must include the latest snapshot of the congestion control metadata from the transmitter state. As with ACK, these values are maintained per-flow and the NACK carries this data for the corresponding path.

9.1.8 Resync Generation

The packet delivery sublayer uses Resync packets to synchronize the PSN between the transmitter and receiver. Under certain error scenarios (such as ULP NACK or xLR drop) a packet sent by the transmitter may be lost permanently or NACKed by the receiver creating a hole in the sequence number space. Under these conditions Resync packets are sent by the transmitter to the receiver to fill the PSN hole and advance the receiver sliding window. Resync packets are themselves delivered reliably and use the PSN of the original packet that was lost or NACKed. The transmitter must implement Resync generation according to the following rules:

- The transmitter must transmit the Resync packet with the PSN, RSN and path-id of the original packet that was lost.
- The transmitter must include the packet type of the original packet in the Resync packet.
- The transmitter must include the reason code in the Resync packet.
- The transmitter must determine whether the packet is eligible for transmission using the Tx gating functions described here.

9.2 Receiver Implementation

The following subsections detail the behavior of the receiver.

9.2.1 Receiver State

The receiver maintains the following per-connection variables that are updated dynamically during data transfer:

- Request Window Base PSN (RBPSN): This is the PSN value of the oldest packet in the request sliding window that is yet to be received by the receiver.
    - RBPSN is initialized by the CM when the connection is configured.
    - When a request sliding window packet is received with a PSN value of RBPSN, RBPSN must be updated to the next highest PSN of the packet that has not been received (using modulo 2^{32} arithmetic). The update of the BSN must also clear the bits in the sequence number bitmap corresponding to packets from the previous RBPSN to the updated RBPSN.
- Data Window Base PSN (DBPSN): This is the PSN value of the oldest packet in the data sliding window that is yet to be received by the receiver.

    - DBPSN is initialized by the CM when the connection is configured.
    - When a data sliding window packet is received with a PSN value of DBPSN, DBPSN must be updated to the next highest PSN of the packet that has not been received (using modulo 2^{32} arithmetic). The update of the BSN must also clear the bits in the sequence number bitmap corresponding to packets from the previous DBPSN to the updated DBPSN.
- Request Window Sequence Number Bitmap: This is a bitmap whose width in bits equals the receiver request sliding window size, where each bit represents a packet within the request sliding window. The bit is set to 1 if the packet has been received and acknowledged by the receiver. Otherwise the bit is 0. Bit 0 represents a PSN value of RBPSN and Bit n represents a PSN value of (RBPSN+n).
    - This bitmap is initialized to all 0s when the connection is configured.
    - When a packet is received and acknowledged by the receiver, this bitmap is updated based on the PSN of the packet. For some packet types (Push Request), an acknowledgement is sent by the receiver only after the packet is delivered to the ULP and is acknowledged by the ULP.
- Data Window Sequence Number ACK Bitmap: This is a bitmap whose width in bits equals the receiver data sliding window size, where each bit represents a packet within the data sliding window. The bit is set to 1 if the packet has been acknowledged by the receiver. Otherwise the bit is 0. Bit 0 represents a PSN value of DBPSN and Bit n represents a PSN value of (DBPSN+n).
    - This bitmap is initialized to all 0s when the connection is configured.
    - When a packet is acknowledged by the receiver, this bitmap is updated based on the PSN of the packet. For some packet types (Push Data), an acknowledgment is sent by the receiver only after the packet is delivered to the ULP and is acknowledged by the ULP.
- Data Window Sequence Number Rx Bitmap: This is a bitmap whose width in bits equals the receiver data sliding window size, where each bit represents a packet within the data sliding window. The bit is set to 1 if the packet has been received by the receiver. Otherwise the bit is 0. Bit 0 represents a PSN value of DBPSN and Bit n represents a PSN value of (DBPSN+n).
    - This bitmap is initialized to all 0s when the connection is configured.
    - When a packet is received by the receiver, this bitmap is updated based on the PSN of the packet.
- Request Out-of-Window Notification (R-OWN): This bit is set to 1 if the receiver has to drop packets beyond the right side of the request sliding window. Otherwise the bit is 0. This bit is reset to 0 when an outgoing EACK carries the notification.
- Data Out-of-Window Notification (D-OWN): This bit is set to 1 if the receiver has to drop packets beyond the right side of the data sliding window. Otherwise the bit is 0. This bit is reset to 0 when an outgoing EACK carries the notification.

- Current SPI: This variable represents the current value of the Security Parameters Index (SPI) field used to derive the PSP encryption key. This variable is checked against the SPI field in the PSP header of the received packet as described in the following section.
    - Current SPI is initialized by the CM when the connection is configured.
    - Current SPI is updated by the CM every time the PSP key is rotated.
- Current SPI Generation: This variable represents the generation of the PSP key that corresponds to the current SPI.
    - Current SPI Generation is initialized by the CM when the connection is configured.
    - Current SPI Generation is updated by the CM every time the PSP key is rotated.
- Previous SPI: This variable represents the previous value of the SPI field. This variable is checked against the SPI field in the PSP header of the received packet as described in the following section.
    - The previous SPI is initialized by the CM to the same value as the current SPI when the connection is configured.
    - Previous SPI is updated by the CM every time the PSP key is rotated.
- Remote Time: This variable represents the receiver's estimate of the transmitter's time.
    - Remote Time is initialized by the CM to a value that is conservatively older than the time of the remote transmitter when the connection is configured and every time the PSP keys are rotated.
    - Remote Time is set to the value of the timestamp in the PSP.IV field of a received packet if the PSP.IV field is larger than the remote time field using modulo arithmetic.
- PSP Connection ID (PCID): This variable is a 64b value that uniquely identifies a PSP tunnel. This variable is used when multiple Falcon connections share a common PSP tunnel. The PCID is carried in the virtualization cookie (VC) field of the PSP header and is verified against the PCID value stored in the connection context by the receiver as part of the security checks of incoming packets.

The following variables are global across all connections and are initialized once when the receiver is initialized and do not change during the lifetime of the connection:

- Replay Protection Margin: This variable is used for replay protection and specifies a constant window of time older prior to the remote time value where the receiver will still admit packets as described in the next section. This variable is initialized by the CM and must be large enough to accommodate network jitter introduced by reordering but also much smaller than the smallest possible PSN wrap-around time.
- Current SPI Generation: This variable represents the current generation of the PSP master key.
    - Current SPI Generation is configured by the CM upon initial boot of the NIC.
    - Current SPI Generation is incremented by 1 with every PSP key rotation.

- Allow Previous SPI Generation: This boolean variable is used to control whether the receiver accepts or rejects packets with an older PSP key generation. It is configured by the CM during a key rotation flow as described in the section below on SPI check for received packets.
- PCID Match Enable: This boolean variable must be set to true to enable the PCID check by the receiver. It is configured by the CM based on whether multiple Falcon connections share a single PSP tunnel.

9.2.2 Packet Acceptance Checks

The receiver must implement a variety of packet acceptance checks as described in the following subsections. A packet that passes all the acceptance checks must be delivered to the transaction sublayer.

9.2.2.1 Packet Integrity Check

The receiver must implement packet integrity checks for every received packet according to the following rules:

- If the packet fails the header parsing checks or if the packet is received with a CRC error or if the packet has an invalid checksum, the receiver must discard the packet.
- The receiver must validate the length in the UDP header against the number of bytes received and must discard the packet if there is a length mismatch.
- If the packet fails decryption, the receiver must discard the packet. Optionally, the packet must be logged and an error indication must be sent to the CM.

9.2.2.2 Replay Protection Check

The packet delivery sublayer implements replay protection check similar to that implemented by TCP PAWS. The replay protection check relies on the remote timestamp carried in the IV field of the PSP header. The receiver compares the remote timestamp contained in the PSP header IV field against its most recent estimate of the transmitter's time maintained in the remote time field. The receiver must implement the replay protection check according to the following rules:

- If the value of (Remote Time - Replay Protection Margin) is less than the PSP header IV field value using modulo arithmetic, the packet must be accepted. Otherwise the packet must be dropped.
- If the packet is dropped due to failing the check in the previous step, the receiver must not generate an ACK or a NACK generation event.

The replay protection implemented by the packet delivery sublayer differs from TCP PAWS in that the receiver will accept packets with an older timestamp value in the PSP header IV field than the remote time variable. This is because the replay protection margin allows the receiver to admit packets that were sent earlier by the transmitter but were delayed in the network.

The remote time variable could wrap around if the connection is idle for a long time. The packet delivery sublayer does not employ a keep-alive mechanism. Therefore, the remote time variable must have a wrap-around time that is much larger than the PSP key rotation intervals. When the PSP keys are rotated, the CM must update the remote time variable based on the current estimate of the transmitter's time.

9.2.2.3 SPI Check

The receiver must apply the SPI check to all packets including ACK/NACK packets. The SPI check verifies that the packets received on a connection are from the expected PSP tunnel. If the SPI check fails, the receiver must signal an error to the CM. The receiver must not generate ACK/NACK packets for received packets that fail the SPI check.

To allow for PSP key rotations during the lifetime of an active connection, the receiver maintains the current SPI and the previous SPI variables per connection. The receiver also maintains the current SPI generation and computes the previous SPI generation. When a PSP key rotation event occurs, the CM must perform the following steps:

1. Increment the global current SPI generation and allow previous SPI generation packets to be received:
    - Global.AllowPreviousSPIGen = 1
    - Global.CurrentSPIGen = Global.CurrentSPIGen + 1
2. Iterate over all established connections and update the SPI variables as follows:
    - Connection.PreviousSPI = Connection.CurrentSPI
    - Connection.CurrentSPI = new SPI assigned by PSP key rotation
    - Connection.CurrentSPIGen = Connection.CurrentSPIGen + 1
3. Communicate new encryption keys and SPI to the remote transmitters of all active connections.
4. (Optional) After an appropriate waiting period for the new keys to take effect at the remote transmitters (few network RTTs), iterate over all established connections and remove the previous SPI as follows:
    - Connection.PrevSPI = Connection.CurrentSPI
5. Before next PSP key rotation (indicated by a notification from IMC to CM), disallow accepting packets from the previous generation:
    - Global.AllowPreviousSPIGen = 0

The receiver must implement the SPI check on all packets according to the following pseudocode:

```
bool PacketPassesSPICheck(connection.curSPI, connection.prevSPI,
                          connection.curSPIGen, connection.PCID,
                          global.curSPIGen, global.allowPrevSPIGen,
                          global.pcidMatchEnable) {
  if (connection.curSPI == 0) {
    // SPI = 0 means connection is not configured
    return false
  }
  prevSpiGen = global.curSPIGen - 1
  curSpiGenMatches = global.allowPrevSPIGen ?
                     ((connection.curSPIGen == global.curSPIGen) ||
                      (connection.curSPIGen == prevSPIGen)) :
                     (connection.curSPIGen == global.curSPIGen)

  // curSpi is equal and generation for curSpi is allowed
  curSpiMatches = (connection.curSPI == psp_hdr.spi) &&
                  curSpiGenMatches

  prevSpiSameGenMatches =
    (MSB(connection.curSPI) == MSB(connection.prevSPI)) &&
    // prevSpi is equal, non-zero and generation for prevSpi is allowed
    ((connection.prevSPI == psp_hdr.spi) && (connection.prevSPI != 0)) &&
    // since MSB is equal, prevSpi has same generation as curSpi
    curSpiGenMatches

 prevSpiPrevGenMatches =
    (MSB(connection.curSPI) != MSB(connection.prevSPI)) &&
     global.allowPrevSPIGen &&
    // prevSpi is equal, non-zero and generation for prevSpi is allowed
    ((connection.prevSPI == psp_hdr.spi) && (connection.prevSPI != 0)) &&
    // since MSB is not equal, prevSpi has curSpiGen - 1
    (connection.curSPIGen - 1 == prevSpiGen)

  pcidMatches = (~global.pcidMatchEnable) ||
                (psp_hdr.vcookie == connection.PCID)

  return pcidMatches &&
    (curSpiMatches || prevSpiSameGenMatches || prevSpiPrevGenMatches)
}
```

9.2.2.4 Sliding Window Check

The receiver must implement the following sliding window acceptance checks for every packet. These checks apply to the request or data sliding window depending on the type of the packet.

- If the PSN value of the received packet is less than RBPSN / DBPSN, the packet is considered old and duplicate and the receiver must discard the packet.
- If the PSN value of the received packet is greater or equal to (RBPSN + request sliding window) for request packets or (DBPSN + data sliding window) for data packets, the receiver must discard the packet and set the corresponding R-OWN/D-OWN bit indicating out-of-window drops.
- If the PSN value falls within the receiver's sliding window, and the bit in the corresponding req-bitmap/data-ack-bitmap at location (PSN - RBPSN/DBPSN) is 1b, the packet is considered a duplicate packet and must be discarded.
- Note: discarded packet should also trigger ACK coalescing, but its AR bit is ignored.
- If the above checks pass, the packet must be delivered to the transaction sublayer. If the packet type is Push Data, the bit in the receiver data-ack-bitmap at location (PSN - DBPSN) must not be set to 1b until the packet is explicitly acknowledged by the ULP. For all other packet types, the bit in the corresponding req-bitmap/data-ack-bitmap at location (PSN - RBPSN/DBPSN) must be set to 1b and triggers ACK coalescing.
- If the PSN to be acknowledged is equal to RBPSN/DBPSN, the RBPSN/DBPSN value must be updated to be equal to the next highest PSN that hasn't been acknowledged. After the RBPSN/DBPSN value is updated, the bitmaps must be shifted by the amount of RBPSN/DBPSN advancement.

9.2.3 ACK Processing

Received ACK packets result in changes to the transmitter's sliding window state for the connection indicated by the CID in the ACK packet. The receiver must process a received ACK packet, be it BACK or EACK, according to the following rules:

- If the RBPSN in the ACK packet is smaller than the transmitter's RBPSN, the receiver must discard the ACK packet.
- If the receiver RBPSN in the ACK packet is larger than the transmitter's RBPSN, the receiver must update the transmitter's RBPSN to match the RBPSN in the ACK packet.
- If the DBPSN in the ACK packet is smaller than the transmitter's DBPSN, the receiver must discard the ACK packet.
- If the receiver DBPSN in the ACK packet is larger than the transmitter's DBPSN, the receiver must update the transmitter's DBPSN to match the DBPSN in the ACK packet.
- The receiver must trigger the freeing up of the resources allocated to all acknowledged packets in the retransmit buffer.
- The receiver must use the timestamps T1 and the ACK's Rx timestamp to compute the RTT samples on a per-flow basis.
- Compute the number of newly acknowledged packets for each path by inspecting each acknowledged PSN. The path-id used for the packet was stored in the transmit context at the time of packet transmission.

Additionally, if the ACK is an EACK, following additional rules are required:

- The receiver must OR the Receiver Request Sequence Number Bitmap in the EACK packet with the transmitter's Request Sequence Number Bitmap after adjusting the RBPSN.
- The receiver must OR the Receiver Data Sequence Number ACK Bitmap in the EACK packet with the transmitter's Data Sequence Number Rx Bitmap after adjusting the DBPSN.
- The receiver must trigger the freeing up of the resource allocated to all packets acknowledged by the bitmaps.
- The receiver should trigger early retransmission based on Receiver Request Sequence Number Bitmap, Receiver Data Sequence Number Rx Bitmap, Receiver Request Out-Of-Window Notification, Receiver Data Out-of-Window Notification in the EACK.

9.2.4 NACK Processing

Received NACK packets may result in changes to the transmitter's sliding window state for the connection indicated by the CID in the NACK packet. The receiver must process received NACK packets according to the following rules:

- If the RBPSN in the NACK packet is smaller than the transmitter's RBPSN, the receiver must discard the NACK packet.
- If the receiver RBPSN in the NACK packet is larger than the transmitter's RBPSN, the receiver must update the transmitter's RBPSN to match the RBPSN in the NACK packet.
- If the DBPSN in the NACK packet is smaller than the transmitter's DBPSN, the receiver must discard the NACK packet.
- If the receiver DBPSN in the NACK packet is larger than the transmitter's DBPSN, the receiver must update the transmitter's DBPSN to match the DBPSN in the NACK packet.
- The receiver must trigger the freeing up of the resources allocated to all acknowledged packets in the retransmit buffer.
- The receiver must use the timestamps T1 and the NACK's Rx timestamp to compute the RTT samples on a per-flow basis.

In addition to the above rules, the receiver must process NACK packets according to the NACK code as described in the table below:

Table 15: NACK Error Codes and Associated Recovery Actions

NACK Code	NACK Handling
Resource Drop	Receiver must indicate the resource drop to the RUE and trigger an adjustment of the ncwnd. Transmitter will retransmit the packet as normal.
Receiver Not Ready	Receiver must trigger a RNR NACK event with the timeout value specified in the NACK packet to the transaction sublayer. The transaction sublayer will delay the packet transmission according to the timeout value.
xLR Drop	Receiver must trigger a NACK drop event to the transmitter. The transmitter must remove the packet from the retransmit queue and send a Resync packet.
Complete in Error	Receiver must trigger a NACK drop event to the transmitter. The transmitter must remove the packet from the retransmit queue and send a Resync packet. The receiver must also trigger a NACK complete-in-error event to the transaction sublayer with the RSN of the packet that was dropped.
Non-Recoverable Error	Receiver must trigger a NACK drop event to the transmitter. The transmitter must remove the packet from the retransmit queue and send a Resync packet. The receiver must also trigger a NACK non-recoverable error event to the transaction sublayer with the RSN of the packet that was dropped.
Invalid CID	Receiver must trigger a NACK drop event to the transmitter. The transmitter must remove the packet from the retransmit queue and send a Resync packet. The receiver must also trigger a NACK invalid-CID error event to the transaction sublayer with the RSN of the packet that was dropped.

9.2.5 Resync Processing

The receiver must process received Resync packets according to the following rules:

- The bit in the corresponding receiver sequence number bitmap at location (PSN - RBPSN/DBPSN) must be set to 1b and an ACK generation event must be sent to the transmitter.
- If the PSN of the received packet is equal to RBPSN/DBPSN, the RBPSN/DBPSN value must be updated to be equal to the next highest PSN that hasn't been received. After

the RBPSN/DBPSN value is updated, the bitmaps must be shifted by the amount of RBPSN/DBPSN advancement.

10. Congestion Control

Congestion control (CC) is implemented on a per-connection basis and is part of the packet delivery sublayer. The CC algorithm is implemented in a Rate Update Engine (RUE), which is a logical block that is separate from the main data path of the packet delivery sublayer. The packet delivery sublayer provides various congestion signals and state to the RUE including accurate measurements of delay, ECN marking, receive buffer occupancy, number of ACKed and NACKed packets etc. With multipathing enabled, congestion signals are provided on a per-flow basis.The interface between the packet delivery sublayer and RUE uses producer-consumer queues in each direction and is described in the following sections. The RUE logical block may be implemented entirely in hardware, or in software running on an embedded CPU on the NIC.

The separation of congestion control functions between the packet delivery sublayer and RUE has at least two advantages:

- Regardless of SW or HW RUE, the measurement of congestion control signals, generating acknowledgement packets, and the enforcement of congestion window and rate for a connection are implemented in Falcon hardware close to the wire. Such a separation makes CC more responsive to network congestion via fast measurement of signals and enforcement of rates.

- Separation of RUE and the rest of Falcon protocol allows for tuning and iterating on CC without impacting the main datapath of the packet delivery sublayer.

In the following, we describe the Swift congestion control algorithm, and protective load balancing (PLB). The design of Swift is described in this paper - Swift: Delay is Simple and Effective for Congestion Control in the Datacenter. The design of PLB is described in this paper – PLB: Congestion Signals are Simple and Effective for Network Load Balancing. In what follows, we focus on the implementation aspects in the following order:

1. A methodology for obtaining precise measurements of delay signal;
2. A description of Swift pseudocode;
3. A description of PLB pseudocode;
4. Configuration of Swift and PLB, and
5. RUE architecture including the API between RUE and Falcon hardware.

We expect that the RUE ↔ Falcon API and methodology for delay measurements will carry over across hardware generations while the details of Swift algorithm as instantiated in RUE will be iterated on over time.

10.1 Delay Measurement

The packet delivery sublayer provides precise delay measurements. This is important for low-latency, delay-based congestion control algorithms such as Swift. Below we describe how the delay measurements are used to compute the delay experienced in the network.

The delay measurement uses transmit timestamps taken close to the wire in the encryption HW block that implements PSP or IPSEC ESP encryption protocols and receive timestamps taken by the packet delivery sublayer as shown in the figure below.

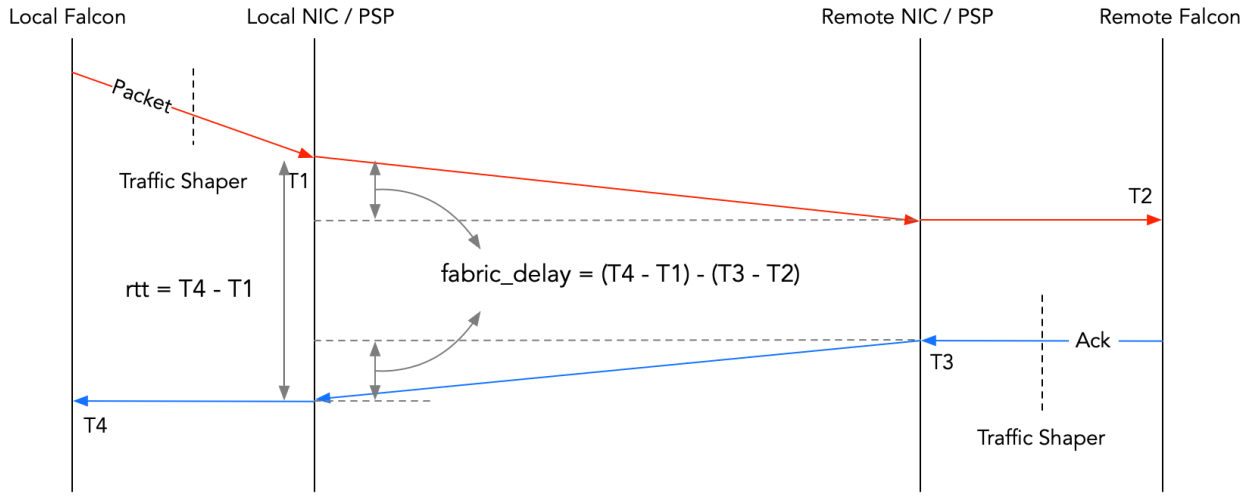


Figure 28: Timestamps for Round-trip time Measurements

As shown in the above figure, a timestamp T_1 is taken by the encryption block when a packet is transmitted. The sliding window receiver takes the timestamp T_2 when the packet is received. Similarly, timestamp T_3 is taken when an ACK or NACK packet is transmitted and the timestamp T_4 is taken when the ACK/NACK packet is received by the packet delivery layer. The receiver includes the timestamps T_1 and T_2 in every ACK/NACK packet sent to the transmitter. These allow calculation of the following two delays:

- rtt == $T_4 - T_1$
- fabric_delay == $(T_4 - T_1) - (T_3 - T_2)$

Additionally, if clocks of the NICs are synchronized, we can calculate one-way delays:

- forward_delay == $T_2 - T_1$

- reverse_delay == T_4 - T_3

The assumption here is that the time delta between packet receipt at remote NIC and that at remote Falcon is very small.

When ACK coalescing happens at the receiver, the receiver must include the T_1 and T_2 timestamps from the most recently received packet in the generated ACK packet. This ensures that the fabric_delay measurement measures the fabric RTT precisely and does not reflect the ACK coalescing delays. The meaning of the four timestamps involved in the fabric_delay computation is described in the table below:

Table 16: Timestamp Meanings

Timestamp	Meaning
T_1	This timestamp defines the time when a packet sent by the packet delivery sublayer is transmitted by the NIC. This timestamp is indicated by the timestamp included in the IV field of the PSP or IPSEC ESP headers by the encryption block. It is assumed that there are no significant queueing delays between the encryption block and the wire.
T_2	This timestamp defines the time when a packet was received by the packet delivery layer at the receiver. It is assumed that there are no significant queueing delays between the wire and the Falcon block on receive.
T_3	This timestamp defines the time when a ACK/NACK packet sent by the packet delivery sublayer is transmitted by the NIC. This timestamp is indicated by the timestamp included in the IV field of the PSP or IPSEC ESP headers by the encryption block. It is assumed that there are no significant queueing delays between the encryption block and the wire.
T_4	This timestamp defines the time when a ACK/NACK packet was received by the packet delivery layer at the receiver. It is assumed that there are no significant queueing delays between the wire and the Falcon block on receive.

Sometimes delay needs to be smoothed by using an exponentially weighted moving average (EWMA) algorithm. The EWMA algorithm computes the smoothed delay using the following equation:

$$smoothed_delay = (1 - \alpha) \times smoothed_delay + \alpha \times delay_{sample}$$

Where α is the averaging constant of the EWMA algorithm. Setting α to one gives the instantaneous delay.

With multipathing enabled, the timestamps are tracked per path by relying on the path-id encoded in the packet's flow label.

10.2 NIC Buffer Occupancy Measurement

A Falcon receiver maintains an EMA of occupancy for each of the 2 network request resource pools: network Rx packet, network Rx buffer. The EMA of occupancy is updated each time the occupancy changes:

$$ema_occupancy = (1 - \alpha) \times ema_occupancy + \alpha \times current_occupancy$$

Where α is the averaging constant, which can be configured separately for the 2 pools. Setting $\alpha = 1$ is equivalent to getting instantaneous occupancy.

When sending an ACK/NACK, it carries the quantized buffer level of Falcon. Each of the 2 pools has 32 levels of quantization, based on 31 quantization levels: if $level_i \leq ema_occupancy \leq level_{i+1}$, the quantized buffer level is i. The 2 pools have separate sets of quantization levels. The ACK/NACK carries the maximum quantized buffer level across the 2 pools.

10.3 Swift Congestion Control Algorithm

Swift is a congestion control algorithm that aims to control queuing in the network by reacting to measured delay signals, control remote NIC buffer occupancy by reacting to the measured remote EMA buffer occupancy, as well as reacting to other events such as NACKs and retransmissions.

Swift controls its rate using 2 congestion windows (fcwnd and ncwnd, for controlling fabric and NIC congestion respectively) with an ability to operate in a pacing mode for fcwnd < 1. When in pacing mode, the rate is enforced using the Timing Wheel, by inserting an inter-packet gap delay. In this section, we elucidate Swift's pseudo code and its implementation aspects.

In the following descriptions, we use fabric_delay as the congestion signal. But one can replace it with forward_delay if clocks are synchronized. The EMA buffer occupancy is denoted by rx_buffer_level.

10.3.1 Process Ack and Nack that is not NIC-resource-exhaustion

```
ComputeAckFabricCongestionWindow():
  prev_fcwnd = fcwnd
```

```
  delay_target = FabricDelayTarget()
  if smoothed_delay <= delay_target:
    if fcwnd >= 1:
      fcwnd += fabric_additive_increment * num_packets_acked / fcwnd // ai = 1
      // The goal here is to increase the cwnd by ai per RTT, this
      // can be implemented differently where you accumulate
      // num_packets_acked and increment fcwnd by ai once num_packets_acked >
      // fcwnd and reset num_packets_acked.
    else:
      fcwnd += fabric_additive_increment * num_packets_acked
  else if HasRttElapsed(fabric_window_time_marker):
        fcwnd *= max(1 - fabric_multiplicative_decrease_factor *
                  (smoothed_delay - delay_target) / smoothed_delay, 1 -
                   max_fabric_multiplicative_decrease_factor)
  fcwnd = clamp(min_fcwnd, fcwnd, max_fcwnd)

  if fcwnd < prev_fcwnd or fcwnd == min_fcwnd:
    fabric_window_time_marker = now
  else:
    UpdateTimeMarker(fabric_window_time_marker)
  return fcwnd;

ComputeAckNICCongestionWindow():
  prev_ncwnd = ncwnd

  // The goal is to decrease or increase the ncwnd (by
  // nic_additive_increment) at most once per RTT.
  if rx_buffer_level < target_rx_buffer_level:
    if ncwnd_direction == kDecrease or HasRttElapsed(nic_window_time_marker):
      # Previous state was Decrease.
      ncwnd += nic_additive_increment
      ncwnd_direction = kIncrease
  else if HasRttElapsed(nic_window_time_marker):
    rx_buffer_delta = rx_buffer_level - target_rx_buffer_level
    ncwnd *= max(1 - rx_buffer_delta / rx_buffer_level, 1 -
max_nic_multiplicative_decrease_factor)
    ncwnd_direction = kDecrease
  ncwnd = clamp(min_ncwnd, ncwnd, max_ncwnd)

  if ncwnd < prev_ncwnd or ncwnd == min_ncwnd:
    nic_window_time_marker = now
  else if: ncwnd > prev_ncwnd or ncwnd == max_ncwnd:
    nic_window_time_marker = now
  else:
```

```
    UpdateTimeMarker(nic_window_time_marker)
  return [ncwnd, ncwnd_direction]

ProcessAckNackCommon():
  smoothed_rtt = GetSmoothed(rtt_smoothing_alpha, smoothed_rtt, rtt)
  smoothed_delay = GetSmoothed(delay_smoothing_alpha, smoothed_delay, delay)
  fcwnd = ComputeAckFabricCongestionWindow()
  inter_packet_gap = ComputeInterPacketGap()
  ncwnd = ComputeAckNICCongestionWindow()
  return [fcwnd, inter_packet_gap, ncwnd]
```

10.3.2 On a Retransmit

```
ProcessRetransmit():

  if Event != kRetransmit:
    retransmit_count = 1, state = kRetransmit
  else:
    retransmit_count += 1

  if retransmit_count == 1 and HasRttElapsed(fabric_window_time_marker):
    fcwnd *= (1 - max_fabric_multiplicative_decrease_factor)
    fabric_window_time_marker = now
  else if  retransmit_count >= retransmit_limit:
    fcwnd = min_fcwnd
    fabric_window_time_marker = now

   retransmit_timeout = max(retransmit_timeout_scalar * smoothed_rtt,
min_retransmission_timeout)
  return [fcwnd, inter_packet_gap, ncwnd, retransmit_timeout]
```

10.3.3 On NACK that indicates NIC resource exhaustion

```
ProcessNackResourceExhaustion():
  // fcwnd is computed in the same way as other ACK/NACK
  fcwnd = ComputeAckFabricCongestionWindow()
  inter_packet_gap = ComputeInterPacketGap()

  // ncwnd is reduced by maximum
  if ncwnd_direction == kIncrease or HasRttElapsed(nic_window_time_marker):
    ncwnd *= (1 - max_nic_multiplicative_decrease_factor)
    nic_window_time_marker = now
```

10.3.4 Common Methods

```
Initialize():
  flow_scaling_alpha = max_flow_scaling / (sqrt(min_flow_scaling_window) -
sqrt(max_flow_scaling_window))
  flow_scaling_beta = -1.0 * flow_scaling_alpha /
sqrt(max_flow_scaling_window)

FabricDelayTarget():
  flow_scaling_delay_adder = flow_scaling_alpha / sqrt (fcwnd) +
flow_scaling_beta
  flow_scaling_delay_adder = clamp(0, flow_scaling_delay_adder,
max_flow_scaling)

  topology_scaling_delay_adder = topology_scaling_per_hop * forward_hops

  return base_delay_target + flow_scaling_delay_adder +
topology_scaling_delay_adder

GetSmoothed(alpha, smoothed, delay):
  return smoothed * alpha + delay * (1 - alpha);

HasRttElapsed(time_marker):
  delta = now - time_marker
  return delta >= rtt

UpdateTimeMarker(time_marker):
  time_delta = now - time_marker
  if time_delta > rtt:
     time_marker = now - rtt

ComputeInterPacketGap():
  if fcwnd > 1:
    return 0
  if inter_packet_gap == 0:
    return rtt / fcwnd
  else:
    return inter_packet_gap / fcwnd
```

10.3.5 Congestion Windows

Swift uses two congestion windows for each connection. fcwnd and ncwnd are modulated based on congestion in the fabric and NIC respectively. When the effective cwnd is greater

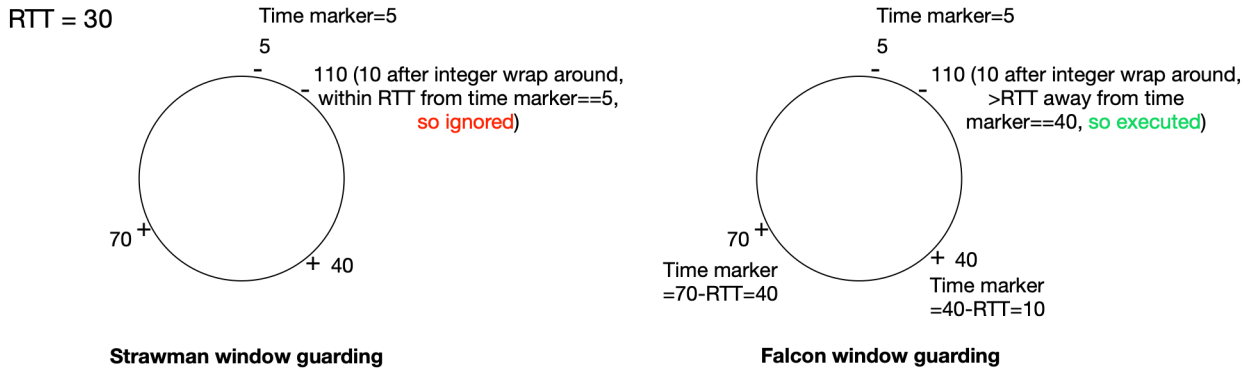
than or equal to one, Falcon limits the number of outstanding packets to the integer portion of fcwnd. When fcwnd or ncwnd is less than one the number of outstanding packets is fixed to one and packet pacing is enabled based on the inter-packet gap specified by RUE. ncwnd is always integer.

With multipathing enabled, fcwnd is computed by the RUE on a per-flow basis using congestion signals collected for each specific path. These per-flow fcwnd values are directly enforced in the datapath by the Max-Open-Window scheduler to gate packet transmissions per path, and are aggregated into a connection-level fcwnd as the sum of per-flow integer windows plus cwnd_carryover. When one or more flows' fcwnd falls below one, packets on those flows are paced using a per-flow inter-packet gap calculated by RUE after taking into account the flow's RTT and fractional fcwnd. Only the packets assigned to such sub-unity fcwnd flows are paced by the datapath hardware.

10.3.6 Window Guarding

Swift avoids reducing fcwnd multiple times within one round-trip and avoids changing ncwnd multiple times in the same direction within one round-trip. Thus, each connection needs to save two time markers, one for each window. The time marker is used to mark a time, from which within an RTT period fcwnd or ncwnd cannot be changed in certain directions (increase or decrease).

However, Falcon uses a limited number of bits for timestamps (e.g. 24-bit numbers) where integer wrap-around can be frequent. This brings challenges because if two events indicates fcwnd or ncwnd changes in the same direction (e.g., two fcwnd decrease events) are apart by more than the wrap-around epoch, the second event may be treated as within an RTT of the previous event, and thus is mistakenly ignored, as the strawman window guarding in the figure below shows (the circle represents the time and its wrap-around), where events happen in their timestamp order (5, 40, 70, then 110).



2 decrease events (marked by "-") and 2 increase events (marked by "+"), at different times. The circle represents the time (and its wrap around). The integer wraps around every 100 time units.
With strawman window guarding (left), the decrease event at 110 cannot be executed, because after integer wrap around, its timestamp is 10, within an RTT of the previous decrease event at 5. This is not a rare case because in a non-congested network, fcwnd decrease events are infrequent, even though there can be many increase events in between.
Falcon's window guarding (right) solves this problem, by leveraging the increase events in the middle — each increase event also updates the time marker with its own timestamp - RTT (if it is RTT later than previous time marker). So the time marker keeps moving but guarantees correctness.

Figure 29: Window Guarding with Integer Wrap-around

Hence, in Falcon the window guarding is improved to be more tolerant to integer wrap-around. It leverages the events in the middle that indicate changes in the different directions, as shown in the figure above, to keep the time marker moving while guaranteeing correctness. So as long as the distance between two sequential events (regardless of their directions) is smaller than the wrap-around epoch, Falcon window guarding works successfully. This can significantly reduce the number of mistakenly ignored events, because even though one particular type of event may be infrequent (e.g., decrease events in a non-congested network), the frequency of events in both directions should be mostly within the wrap-around epoch.

The following describes the definitions and update policies of the two time markers.

fabric_window_time_marker

- Definition: During the RTT period from fabric_window_time_marker, fcwnd cannot decrease.
- Updates:
    - If fcwnd is decreased, fabric_window_time_marker = now.
    - If fcwnd is increased, and now is > RTT later than fabric_window_time_marker, fabric_window_time_marker = now - RTT.

nic_window_time_marker

- Definition: During the RTT period from fabric_window_time_marker,

    - ncwnd cannot decrease
    - ncwnd cannot increase if the last ncwnd change was also increase.
- Updates:
    - If ncwnd is decreased or increased, nic_window_time_marker = now.
    - If the last event didn't change ncwnd, and now is > RTT later than nic_window_time_marker, nic_window_time_marker = now - RTT.

Additionally, for robustness, if fcwnd/ncwnd are not changed by an event but are at their max/min values, they are treated as increased (if at max) or decreased (if at min) for time marker update purposes.

Below are examples of how to set the fcwnd time marker (ncwnd time marker can be drawn similarly according to the policies above). Situation 1 is for an event that occurs within an RTT of the last time we decreased the fcwnd, thus fcwnd cannot be decreased for this event. In this situation, the time marker is not updated. Situation 2 is for an event that occurs after an RTT has elapsed, thus fcwnd could be decreased if needed. The new time marker depends on whether fcwnd is decreased or not in the event. If fcwnd is decreased the time marker will be set to the current time. If fcwnd is not decreased the time marker will be set to the current time minus one RTT.

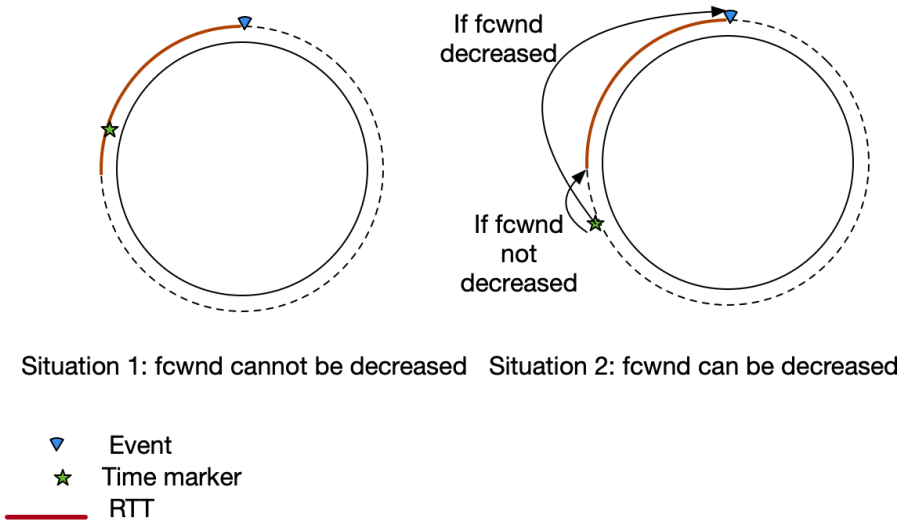


Figure 30: Examples of Setting fcwnd Time Marker

10.4 PLB: Protective Load Balancing

PLB tries to balance the load across different network paths. It runs at every ACK event and primarily involves 3 key calculations:

- Decides if an ACK reflects congestion (smoothed_delay > delay_target * plb_target_delay_multiplier)
- Count the fraction of acknowledged packets that experience congestion within an RTT. If the fraction is no less than plb_congestion_threshold, the RTT is considered congested.
- Count consecutive congested RTTs. If there is no less than plb_attempt_threshold consecutive congested RTT, reroute.

```
ComputePlb(old_window_size): // old_window size is the minimum of fcwnd and
ncwnd before Swift adjusts them at the current ACK event.
  reroute = false
  new_packets_acknowledged = plb_packets_acknowledged + num_packets_acked
  new_packets_congestion_acknowledged = plb_packets_congestion_acknowledged
  if (smoothed_delay > delay_target * plb_target_delay_multiplier):
    new_packets_congestion_acknowledged += num_packets_acked
  if (new_packets_acknowledged >= old_window_size):
    congested_frac = new_packets_congestion_acknowledged /
new_packets_acknowledged
    if (congested_frac < plb_congestion_threshold):
      plb_reroute_attempted = 0
    else:
      Plb_reroute_attempted += 1
      if (plb_reroute_attempted >= plb_attempt_threshold):
        reroute = true
        plb_reroute_attempted = 0
    plb_packets_congestion_acknowledged = 0
    plb_packets_acknowledged = 0
  else:
    plb_packets_congestion_acknowledged = new_packets_congestion_acknowledged
    plb_packets_acknowledged = new_packets_acknowledged
  return reroute
```

10.5 Parameters and Variables

10.5.1 Fabric congestion window settings

Table 17: Fabric Congestion Window Settings

Parameter	
max_fcwnd	Maximum fabric congestion window.
min_fcwnd	Minimum fabric congestion window

Parameter	
fabric_additive_increment	Additive increment used in the AIMD calculation of the fabric congestion window.
fabric_multiplicative_decrease_factor	Multiplicative decrease factor used in the AIMD calculation of the fabric congestion window.
max_fabric_multiplicative_decrease_factor	The maximum multiplicative decrease for the fabric cwnd.
base_delay_target	The base delay target, before flow scaling and topo scaling.

10.5.2 NIC congestion window settings

Table 18: NIC Congestion Window Settings

Parameter	
max_ncwnd	Maximum NIC congestion window.
min_ncwnd	Minimum NIC congestion window
nic_additive_increment	Additive increment used in the AIMD calculation of the NIC congestion window.
nic_multiplicative_decrease_factor	Multiplicative decrease factor used in the AIMD calculation of the NIC congestion window.
max_nic_multiplicative_decrease_factor	The maximum multiplicative decrease for the NIC cwnd.
target_rx_buffer_level	Target NIC RX buffer occupancy level.

10.5.3 PLB setting

Table 19: PLB Settings

Parameter	
plb_target_delay_multiplier	Decide the threshold of fabric delay, above which the ACK is considered to reflect congestion by PLB
plb_congestion_threshold	A threshold for fractions of acknowledged packets within an RTT that are congested, above which PLB considers the RTT as congested.

Parameter	
plb_attempt_threshold	A threshold for consecutive congested RTTs, above which PLB conducts a rerouting.

10.5.4 Common parameters

Table 20: Common Parameters

Parameter	
rtt_smoothing_alpha	The alpha used in EWMA of RTT
delay_smoothing_alpha	The alpha used in EWMA of Swift delay signal
topology_scaling_per_hop	Per-hop scaling factor
retransmit_timeout_scalar	The retransmit timeout is RTT * this scalar
retransmit_limit	Number of retransmission events at which the fabric congestion window is set to the minimum value.
min_retransmission_timeout	The minimum duration for an RTO.
min_flow_scaling_window	Minimum congestion window size used for Swift flow scaling.
max_flow_scaling_window	Maximum congestion window size used for Swift flow scaling.
max_flow_scaling	Maximum addition to target delay by flow scaling.
tie_breaking_threshold	The maximum difference in open_fcwnd between an eligible flow and the maximum open_fcwnd across flows to be treated as a tie for Round Robin path selection in the Max-Open-Window scheduler for multipathed flows.

10.5.5 Measurements

Table 21: Measurements

Parameter	
rtt	Total RTT measured using hardware timestamps.

Parameter	
delay	Delay signal (fabric_delay or forward_delay) measured using hardware timestamps.
forward_hops	Number of hops in the forward direction.
num_packets_acked	Number of acked packets since last Ack.
rx_buffer_level	NIC RX buffer level.

10.5.6 State

Table 22: State Variables

Parameter	
smoothed_rtt	Smoothed RTT using EWMA.
smoothed_delay	Smoothed delay signal (fabric_delay or forward_delay) using EWMA
fcwnd	Fabric congestion window.
ncwnd	NIC congestion window.
ncwnd_direction	An enum indicating the direction of the NIC cwnd (increase or decrease).
fabric_window_time_marker	Fabric congestion window time marker. See time marker section for details.
nic_window_time_marker	NIC congestion window time marker. See time marker section for details.
retransmit_count	Number of successive retransmission events on the connection.
plb_packets_acknowledged	Number of packets acknowledged in the current RTT.
plb_packets_congestion_acknowledged	Number of packets acknowledged in the current RTT that are congested.
plb_reroute_attempted	Number of consecutive RTTs that are congested.

10.5.7 Outputs

Table 23: Outputs

Parameter	
smoothed_rtt	Smoothed RTT using EMWA.
smoothed_delay	Smoothed delay signal (fabric_delay or forward_delay) using EWMA.
fcwnd	Fabric congestion window.
ncwnd	NIC congestion window.
inter_packet_gap	Inter-packet time gap used to pace packets.
fabric_window_time_marker	Fabric congestion window time marker. See time marker section for details.
nic_window_time_marker	NIC congestion window time marker. See time marker section for details.
ncwnd_direction	An enum indicating the direction of the NIC cwnd (increase or decrease).
randomize_path	A boolean value indicating a path change, using the IPv6 flow label.
retransmit_timeout	Retransmission timeout for the connection.

10.6 RUE Architecture

The Rate Update Engine (RUE) is responsible for computing the rate (or congestion window) for a given connection. It computes that based on a set of signals that it gets from the packet delivery sublayer through a queue of events, defined below. The output of the rate update engine is in general a rate (or a congestion window) for the connection. There are other detailed outputs described below.

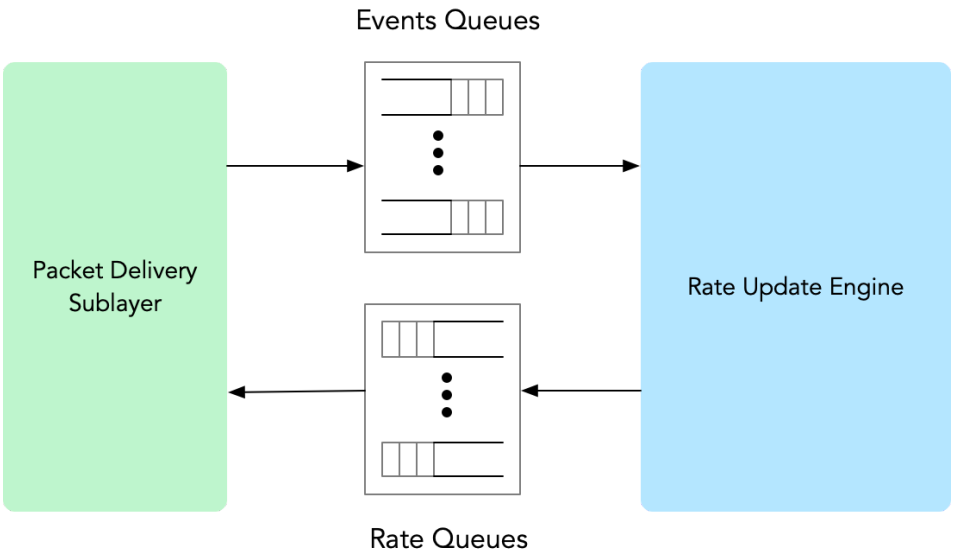


Figure 31: RUE and Packet Delivery Sublayer Interaction

10.6.1 Triggering RUE Event Policy

The Rate Update Engine is triggered with certain events such as receiving an ACK, NACK or a retransmission timeout. With a limited computation in RUE which is a fraction of the datapath, we need to decide when to generate events for RUE and when we should not.

An important goal of congestion control is to provide bandwidth fairness of the network and this is guaranteed to be achieved if we have an update for each active connection within 1 round-trip-time (RTT). The number of packets is reduced as we increase the packet size because we become bandwidth-limited. For example (shown in the chart below), at packet size of 1667 B, the maximum number of packets is 15 Mpps which can be handled by an embedded core, and in that case, no event scheduling is necessary.

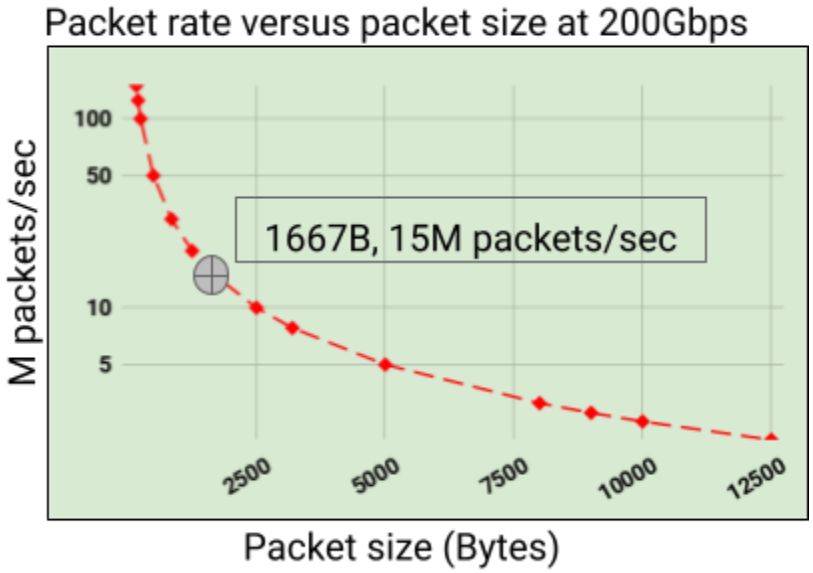


Figure 32: Maximum Packet Rate vs Packet Size

However, traffic rarely has uniform sized operations and Acks are always below that limit. Therefore, we need a policy to achieve the CC goals.

- Bandwidth fairness.
- Quick recovery after the congestion goes away.

We devise the following policy: we set three thresholds against the event queue occupancy. Using two predicates, the packet delivery layer decides whether to enqueue an event or not.

Predicate 1 (P1): Time since the last event is greater than a configuration threshold. This predicate ensures that a connection with low cwnd, can recover as fast as possible when the congestion event disappears.

Predicate 2 (P2): Bytes/packets acked since the last event is greater than a threshold. This predicate ensures that a connection with a lot of small operations does not starve a connection with large ops. While bytes acked is a preferred metric, implementation may choose another scheme (e.g., packets acked, cells acked, etc.) as a complexity tradeoff.

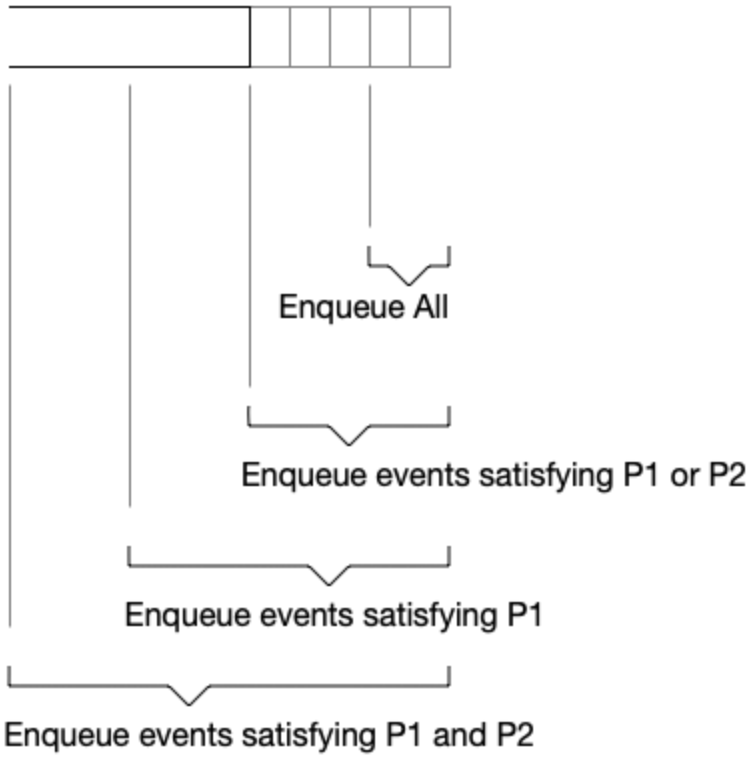


Figure 33: RUE Event Triggering Predicates

10.6.2 RUE Interface

The interface between RUE and the packet delivery sublayer is defined by a pair of producer consumer queues: CC event queue and CC result queue. The following sections define the format of the messages exchanged via these queues.

10.6.2.1 CC Event Queue

The packet delivery sublayer is the producer for the CC event queue and the RUE is the consumer. The packet delivery sublayer produces a CC event into this queue when a ACK or a NACK packet is received or when a retransmission event occurs (either due to early retransmission or due to timeout based retransmission). The format of the event posted to the CC event queue is described in the table below.

Table 24: CC Event Queue Fields

Field	Bit Width	Type	Description
connection_id	24	Int	Connection ID.

Field	Bit Width	Type	Description
event_type	2	Enum	Enum: 0x0 = kRueEventTypeAck Enum: 0x1 = kRueEventTypeNack Enum: 0x2 = kRueEventTypeRetx
t1	24	Int	Valid for ACK and NACK triggered events. Forward path packet TX time
t2	24	Int	Valid for ACK and NACK triggered events. Forward path packet RX time
t3	24	Int	Valid for ACK and NACK triggered events. Reverse path packet TX time
t4	24	Int	Valid for ACK and NACK triggered events. Reverse path packet RX time
rx_buffer_level	5	Int	Quantized RX Falcon network request buffer occupancy level. Valid if event_type is kRueEventTypeAck or kRueEventTypeNack.
eack	1	Int	Set to 0x1 for EACK packet events
nack_code	8	Enum	Refer to the Falcon header NACK packet.
forward_hops	4	Int	ACK packet header CC_meta[63:60], equal to initial TTL - rxPkt.TTL
retransmit_count	8	Int	Valid when event_type is packet retransmission triggered event, and it is cumulative retx_count.
retransmit_reason	1	Enum	Valid when event_type is packet retransmission. Enum: 0x0 = kRetxReasonRto Enum: 0x1 = kRetxReasonEarly
eack_drop	1	Int	Set to 0x1 if potential drop is detected based on the received bitmap in EACK packet.
eack_own	2	Int	Out of window drop notification for either of the receive sliding window.
cc_metadata	22	Custom	Used to exchange information between two RUE. Corresponds to the Rue Info field in the ACK/NACK packet. Valid only if event_type is kRueEventTypeAck or kRueEventTypeNack

Field	Bit Width	Type	Description
fcwnd_fraction	10	Frac	Fractional portion of per-flow Fabric Congestion Window (fcwnd), needed for cwnd incrementals changes.
fcwnd_integer	11	Int	Integer portion of per-flow Fabric Congestion Window (fcwnd). When fcwnd_integer > 0: integer number of outstanding packets allowed for the connection, use cwnd and bypass timing wheel; otherwise, only one outstanding packet for the connection and also use rate_inverse and timing wheel to delay TX packet departure.
ncwnd_fraction	10	Frac	Fractional portion of NIC Congestion Window (ncwnd), needed for cwnd incrementals changes.
ncwnd_integer	11	Int	Allowed maximum number of requests outstanding in each TX sliding window.
is_ipv4	1	Int	Whether the connection is ipv4 or ipv6. Tells which bits (16 or 20) are valid in flowlabel.
multipath_enable	1	Int	Whether multipathing is enabled.
flow_label	20	Int	Flowlabel extracted from IPv6 of UDP (for IPv4).
cc_opaque	18	Int	RUE intermediate variables for RUE own usage, managed by SW RUE. HW Swift implementation does not use these variables.
fabric_window_time_marker	24	Int	Guard for the fabric congestion window. See the Window guarding subsection for details.
nic_window_time_marker	24	Int	Guard for the NIC congestion window. See the Window guarding subsection for details.
base_delay_target	24	Int	The base target delay used in the scaling target delay calculation.
delay_state	24	Int	The delay state used for smoothing and when delay information is not available.
plb_state	24	Int	CC state related to Protective Load Balancing.
delay_select	2	Int	Select form of delay for RUE computation. Enum: 0x0 = kRueDelayFull, delay = t4 - t1 Enum: 0x1 = kRueDelayFabric, delay = (t4-t1) - (t3-t2)

Field	Bit Width	Type	Description
			Enum: 0x2 = kRueDelayFwd, delay = t2 - t1 Enum: 0x3 = kRueDelayRev, delay = t4 - t3
event_queue_select	2	Int	RUE event queue to use.
reserved_0	4	Int	Reserved.
rtt_state	24	Int	The RTT state used for smoothing and when RTT information is not available.
ecn_accumulated	14	Int	Accumulated number of ECN packets received by the other end.
num_packets_acked	10	Int	Number of packets ACK-ed between two RUE events. This means the counter reset to zero when any RUE event is generated. Counter saturates without wrapping around.
csig_enable	1	Int	Whether CSIG is enabled for this connection (both CSIG-tagging and CSIG-reflection).
csig	16	Int	CSIG tag reflected back from the remote side.
connection_queue_size	14	Int	Count of the total number of packets remaining in the scheduler queues for this connection.
reserved	48	N/A	Reserved.
gen	1	Int	G-bit is for software polling new entry

10.6.2.2 CC Result Queue

The RUE is the producer for the CC result queue and the packet delivery sublayer is the consumer. The RUE produces a CC result into this queue for every CC event it consumes from the CC event queue. The format of the result posted to the CC result queue is described in the table below.

Table 25: CC Result Queue Fields

Field	Bit Width	Type	Description
connection_id	24	Int	Connection id, RUE pass through from event queue
fcwnd_fraction	10	Fraction	fractional part of the per-flow cwnd.
fcwnd	11	Int	RUE generated for HW datapath consumption, see event queue format for field description.

Field	Bit Width	Type	Description
fabric_inter_packet_gap	20	Int	Inter-packet gap used to calculate packet pacing delay (packet TX time for timing wheel), required minimum rates: 800Kbps, targeted max rate when cwnd==0 is 8Gbps. All higher rates can be achieved by changing cwnd when cwnd>0. HW set packet departure time as now + ipg (in unit of TS granularity). To achieve R(pps) rate, RUE needs to set ipg = 1/R - target_RTT(in unit of TS granularity)
ncwnd_frac	10	Fraction	Fractional part of ncwnd.
ncwnd	11	Int	Allowed maximum number of requests outstanding in each TX sliding window.
nic_inter_packet_gap	20	Int	Inter-packet gap used to calculate packet pacing delay based on ncwnd.
cc_metadata	22	Custom	Corresponds to the Rue Info field in the ACK/NACK packet.
alpha_request	4	Int	Alpha value for requests to calculate per-connection backpressure to ULPs.
alpha_response	4	Int	Alpha value for responses to calculate per-connection backpressure to ULPs.
fabric_window_time_marker	24	Int	Guard for the fabric congestion window. See the Window guarding subsection for details.
nic_window_time_marker	24	Int	Guard for the NIC congestion window. See the Window guarding subsection for details.
base_delay_target	24	Int	The base target delay used in the scaling target delay calculation.
delay_state	24	Int	The delay state used for smoothing and when delay information is not available.
plb_state	24	Int	CC state related to Protective Load Balancing.
cc_opaque	18	Custom	RUE intermediate variables for RUE own usage. Owned and Updated by Software RUE. Same size as the field in RUE event
delay_select	2	Int	Select form of delay for RUE computation. Same as RUE event.
ar_rate	4	Int	N where every 2^N packet is marked with ack request bit set to true.

Field	Bit Width	Type	Description
rtt_state	24	Int	The RTT state used for smoothing and when RTT information is not available.
flow_label[3:0]	80	Int	Four 20-bit flow label values corresponding to the multiple paths. The 2 LSBs must be the path-ids (00b, 01b, 10b, 11b).
flow_weights[3:0]	16	Int	*(Deprecated in v1.2.0)* Four 4-bit path weights corresponding to the multiple paths.
flow_label_valid[3:0]	4	Int	Each bit indicates whether the corresponding value in flow_label[3:0] is valid or not for the datapath to use.
wrr_restart_round	1	Int	*(Deprecated in v1.2.0)* If set, the WRR immediately resets the credits to be the flow_weights. Else, credits will be reloaded at the end of WRR scheduling round.
flow_id	2	Int	Indicated which of the 4 flows this RUE Response is updating.
csig_enable	1	Int	Whether CSIG tagging is enabled for this connection.
csig_sel	3	Int	Identifies the signal type to be requested over the outgoing packet. The datapath indexes this 3-bit selector into a global 8-entry table to initialize all L2 CSIG tag fields on the outgoing packet. Each 2B entry in this table initializes the signal type (3b), signal value (5b), locator metadata (7b), and reserved bit (1b).
retransmit_timeout	24	Int	Retransmission time-out in Falcon unit-time. Applies both future packets and existing retransmission.
event_queue_select	2	Int	RUE event queue to use.
tlp_probe_timeout	24	Int	TLP probe timeout in Falcon unit-time, to be used for the flow indicated by the flow_id.
rack_retransmit_timeout	24	Int	RACK retransmission timeout in Falcon unit-time, to be used for the flow indicated by the flow_id.
cwnd_carryover	2	Int	Sum of fractional fcwnds of all flows. Added to the sum of integer fcwnds of all flows to calculate connection-level fcwnd.
Reserved	25	N/A	Padding

11. Error Handling and Resource Reclaim

Falcon error handling includes both packet delivery sublayer and transaction layer error cases. Related resources include both physical resources, such as packet and buffer resources, and virtual resources such as sliding window sequence numbers. All Falcon physical resources are ultimately guarded and reclaimed by timeout. The transaction sublayer is responsible for reclaiming the RX packet and buffer resource; and the transaction sublayer must cooperate with the packet delivery sublayer for TX packet and buffer resource reclamation. To prevent race conditions between TX and RX resource reclamations, it is recommended that transaction sublayer resource reclamation timeout to be significantly larger than that of the packet delivery sublayer.

At the packet delivery sublayer, Falcon retransmission for reliability is described in the section above. Falcon guarantees packet delivery sublayer forward progress through the resync process. And the tables below summarize all abnormal cases for sender and receiver, respectively.

Table 26: Sender Error Scenarios and associated Actions

Sender Error Scenario		Sender Action
Transmit packet timeout	Original transmission packet below Max Retransmit Attempts	Retransmit same packet with same PSN
	Original transmission packet at Max Retransmit Attempts	Transmit resync packet with same PSN, and reset Retransmit Counter
	Resync packet at Max Retransmit Attempts	Connection fatal interrupt; Reclaim TX packet and buffer resources
Receive NACK with reason code	Remote xLR drop, or remote ULP error completion (NACK-CIE, NACK-NRE, NACK-CID)	Reset Retransmit Counter, and transmit resync packet with same PSN
	Remote ULP RNR-NACK	Set retransmission timeout to max(RTO, RNR-NACK-timeout)
	Remote resource drop, or receiver sliding window drop	Notify congestion control
Pending packet timeout in connection scheduler		Reclaim TX packet and buffer resources
Connection not alive		Reclaim TX packet and buffer resources

Table 27: Receiver Error Scenarios

Receiver Error Scenario	Receiver Action	
	Process piggyback ACK	Other actions
Violating packets integrity check or security check	No	Drop packet, and raise interrupt
Duplicate packet that is previously acknowledged	Yes	Drop packet and send ACK
Duplicate packet that is previously received and it is not a resync packet	Yes	Drop packet and send ACK
Duplicate packet that is previously received (PSN received state is True), and it is a resync packet	Yes	Drop and send ACK for the ingress PSN
Requiring new receiver resources and exceeding per NIC, per function, per host or per connection resource limit	Yes	Drop packet and send resource drop NACK
Match xLR drop filter	Yes	Drop packet and send xLR drop NACK
Connection not alive	No	Drop packet

At the transaction sublayer, the following table summarizes abnormal cases for initiator and target, respectively.

Table 28: Initiator Error Scenarios and associated Actions

Initiator Error Scenario	Initiator Action	
	Completion Code to ULP	RX Resources Reclamation
Pull or push request transaction timeout[1]	kCmplLocalTO	Upon timeout

[1] Note this encompasses a wide range of errors. In one example, if the pull request packet exceeds Max Retransmit Attempts, the transaction sublayer may get signaled by the packet delivery sublayer. In another example, if the pull request is already acknowledged in the packet delivery sublayer, the transaction sublayer must independently timeout the transaction.

Initiator Error Scenario	Initiator Action	
	Completion Code to ULP	RX Resources Reclamation
Match local xLR drop filter	kCmplXlr	Upon completing to ULP
Pull Data packet size larger than requested[2]	kCmplOpErr Or kCmplLocalTO	Upon completing to ULP immediately or timeout
Remote xLR drop	kCmplRemoteErr	Upon completing to ULP
Remote ULP NACK-CIE for push transactions[3]	kCmplTargetCIE	Upon completing to ULP
Remote ULP NACK-NRE for push transactions	kCmplTargetNRE	Upon completing to ULP
Remote ULP NACK-CID for push transactions	kCmplTargetNackCID	Upon completing to ULP
Exceeding initiator resource pool per NIC, per host, per function or per connection limit	kCmplLocalRsrc	Upon completing to ULP
Connection not alive	kCmplDeadConn	Upon completing to ULP

Table 29: Target Error Scenarios and associated Actions

Target Error Scenario	Target Action
Match local xLR drop filter	Drop, no resource reservation
Target ULP NACK-CIE for push	Resync for packet delivery sublayer progress and reclaim RX resources
Target ULP NACK-NRE for push	Resync for packet delivery sublayer progress and reclaim RX resources
Target ULP NACK-CID for push	Resync for packet delivery sublayer progress and reclaim RX resources

[2] Note that the received pull data packet smaller than requested is a Falcon-level normal case and not a Falcon-level error case.
[3] Note that (1) remote ULP NACK is not applicable to initiator pull transactions, and (2) remote ULP NACK-RNR for push transactions is handled at the packet delivery layer.

Target Error Scenario	Target Action
Target ULP NACK-RNR for push	Resend to target ULP after RNR timeout, or upon receiving a retransmitted push data packet from the network
Target ULP NACK-CIE for pull	N/A
Target ULP NACK-NRE for pull	N/A
Target ULP NACK-CID for pull	Reclaim RX packet and buffer resources
Target ULP NACK-RNR for pull	Resend to target ULP after RNR timeout.

Table 30: Completion Error Codes

Completion Error Code	Value (4-bit)	Completion Error Code	Value (4-bit)
kCmplAck	0x0	kCmplOpErr	0x9
kCmplTargetNackCIE	0x1	kCmplDeadConn	0xA
kCmplTargetNackNRE	0x3	kCmplLocalRsrc	0xB
kCmplTargetNackCID	0x4	kUlpCmplCdtErr	0xC
kCmplLocalTO	0x8	kCmplRemoteErr	0xD
Reserved	0x2, 0x5, 0x6, 0x7, 0xE	kCmplLocalXlr	0xF

12. Interoperability

Feature	**Version Introduced**	**Scope**	**Interop with v1.1 Peer**	**Interop with v1.0 Peer**
RACK-TLP (time-based early retransmission and tail loss probe)	v1.2.0	Sender	Compatible; RACK works as is; TLP falls back to AR-triggered ACK	Compatible; Same behavior as with a v1.1 peer.

Feature	Version Introduced	Scope	Interop with v1.1 Peer	Interop with v1.0 Peer
			because the peer ignores the ER bit.	
EACK-Request (ER) bit	v1.2.0	Packet Wire Format (base header)	Compatible; but the peer ignores the bit. EACK may not be returned immediately.	Compatible; same behavior as with a v1.1 peer.
Rx window expansion (512)	v1.2.0	Receiver	Compatible; Cap the Tx window at the v1.1 receiver's window size (256) for performance.	Compatible
Max-Open-Window multipath scheduler (replaces WRR)	v1.2.0	Sender	Compatible; The receiver's behavior is unchanged.	Not applicable as multipathing must be disabled.
On-demand pull response resource management	v1.2.0	Sender and Receiver	Compatible; for performance target must gate Pull Responses on ncwnd.	Compatible; Same behavior as with a v1.1 peer.
Multipathing (up to 4 paths, path ID in flow label LSBs)	v1.1	Sender and Receiver	Compatible	Incompatible
Congestion Signaling (CSIG compact 4-byte tag)	v1.1	Packet Wire Format	Compatible	Incompatible
ACK/NACK field layout (Host Rx Buffer Occupancy, RUE Info cut from 22 to 17 bits)	v1.1	Packet Wire Format	Compatible	Compatible; v1.0 RUE must interpret the new field.

Feature	Version Introduced	Scope	Interop with v1.1 Peer	Interop with v1.0 Peer
Sender-driven ACK controls (op-boundary AR, ar_rate, ACK rate limiter)	v1.1	Sender	Compatible	Compatible.
Connection-level resource carving (per-connection Xon/Xoff, RUE alpha)	v1.1	Sender	Compatible	Compatible
RUE Event/Result format enhancements	v1.1 / v1.2.0	Sender HW-to-RUE interface	Compatible. The RUE interface only has to match the local hardware.	Compatible. The RUE interface only has to match the local hardware.

13. Acknowledgments

We thank our many collaborators across Google, Intel, and NVIDIA who worked on implementing Falcon in NICs. We also thank the Open Compute Project community for their feedback on the Falcon specification, and the Ultra Ethernet Consortium for standardizing the Congestion Signaling tag formats that Falcon uses.