Scale UP EtheRnet protocol
SUPERp

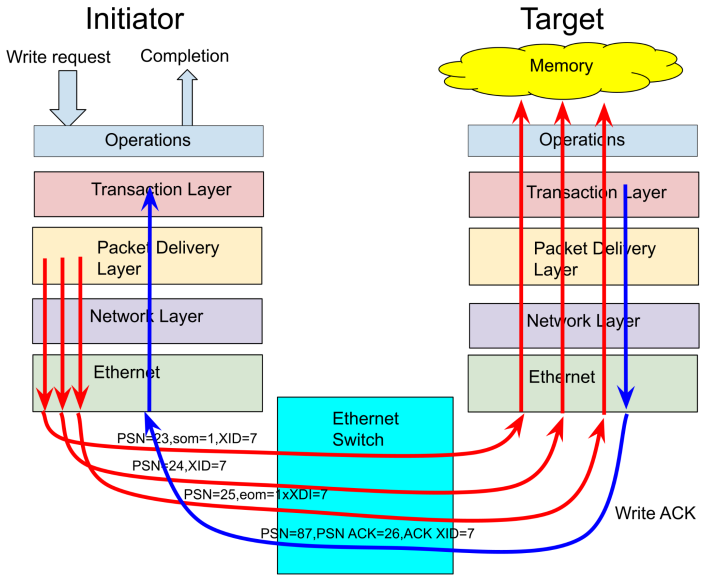
Version 0.4

Tom Herbert <tom@xdpnet.ai>

This document describes the ***Scale UP EtheRnet protocol***, or *SUPERp*. SUPERp is a networking protocol for Scale Up that runs over Ethernet. The primary purpose is to support Remote Memory Access (RMA) operations and other HPC operations among a set of tightly coupled systems in AI/ML Scale Up networks. For instance, several co-located racks of GPUs, or more generically XPUs or DSAs, may be interconnected to create a scalable cluster.

A major goal of SUPERp is that it's able to be implemented by a programmable device without sacrificing performance that could be attained by fixed function hardware. Programmability affords flexibility and allows the protocol to be changed or features added as needed. The protocol is purposely kept simple to be amenable to programmability and the programming model encourages the use of hardware accelerations to maintain high performance.

Protocol Architecture



SUPERp layers and communications example

SUPERp provides a lightweight, reliable, transaction oriented protocol for Scale Up networks. The protocol carries Remote Memory Access (RMA) operations (e.g. read or write remote memory) and is extensible for other HPC operations as well. There are five primary layers in SUPERp: Ethernet Layer, Network Layer, Packet Delivery Layer (PDL), Transaction Layer

XDPnet

(TAL), and the Operations Layer (OPL). These layers provide a complete protocol stack for request/response operations.

The Ethernet layer

The expected deployment for a Scale Up network is a flat topology with a single switch with all nodes connected so that there is one hop between all nodes. Additional switches may be added to allow multi-path and packet spraying while still maintaining the one hop property. Two level or a small number of switched levels are supported. SUPERp works fine with plain 802.3 framing without VLANs. We assume Jumbo frames are supported by the Ethernet switches. SUPERp is extensible to support Priority Flow Control (PFC), Credit Based Flow Control (CBFC) being standardized by the Ultra Ethernet Consortium, and Link Layer Retry (LLR).

The Network Layer

The Network Layer provides compatibility and routability in Layer 3 deployments. There are two modes for SUPERp: *UDP/IP encapsulation* and *SUNH encapsulation*.

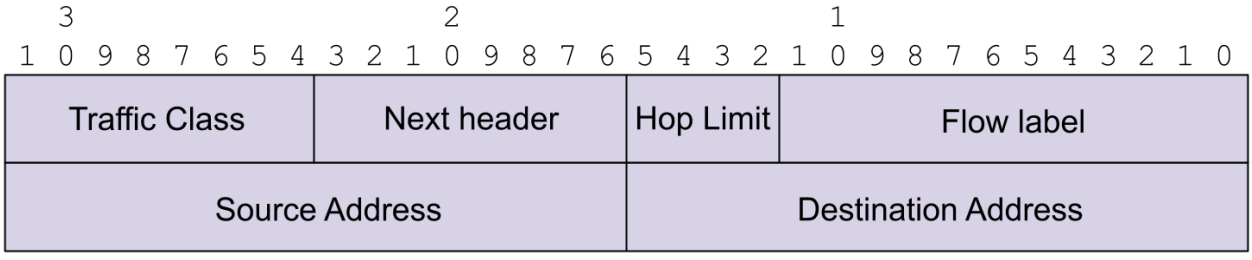
UDP/IP encapsulation

UDP encapsulation is a canonical protocol encapsulation of SUPERp over UDP in IPv4 or IPv6. The destination port will be a number assigned for SUPERp, and the source port is used as an entropy value for ECMP. The UDP encapsulation format for SUPERp is shown in a section below.

SUNH encapsulation

Scale Up Network Header, or *SUNH* is an eight byte compressed network header for IPv4 or IPv6 that carries a SUPERp packet in its payload.

A SUPERp domain is defined by an IPv4 or IPv6 “SUPERp subnet” where the host part is less than or equal to sixteen bits. Within the subnet, each node can be identified by the host portion of its IP address. This allows using an on-the-wire address of sixteen bits that is effectively a compressed IP address (i.e. the fully qualified address is the SUPERp subnet prefix with the sixteen bit address). The SUNH header is an eight byte network header with its own EtherType. The format is:



Fields:

- **Traffic Class** 8-bit Traffic Class field equivalent to the IPv6 Traffic Class field and has the same semantics. The current use of the Traffic Class field for Differentiated Services and Explicit Congestion Notification is specified in [RFC2474] and [RFC3168].
- **Next Header** 8-bit selector. Identifies the type of header immediately following the SUNH header. Uses the same values as the IPv4 Protocol field [IANA-PN].
- **Hop Limit** 4-bit integer Hop Limit field. This is equivalent to the IPv6 Hop Limit field except that the SUNH Hop Limit is four bits. The field value is decremented by 1 by each node that forwards the packet. When forwarding, the packet is discarded if Hop Limit was zero when received or is decremented to zero. A node that is the destination of a packet SHOULD NOT discard a packet with Hop Limit equal to zero; it SHOULD process the packet normally.
- **Flow Label** 12-bit flow label. This is smaller version of the IPv6 Flow Label and is otherwise set and processed with the same semantics
- **Source Address** 16-bit SUNH address of the originator of the packet. Normally this is the host part of the uncompressed source IP address (IPv4 or IPv6)
- **Destination Address** 16-bit SUNH destination address of the intended recipient. Normally this is the host part of the uncompressed destination IP address (IPv4 or IPv6)

The SUNH network header provides three primary benefits:

1. SUNH reduces the amount of on-the-wire protocol overhead
2. SUNH simplifies route lookup to be more efficient with lower
3. SUNH retains the properties and benefits of a Network Layer

The SUNH header allows switches to do layer 3 switching and has the added benefit that there are no hacks to the Ethernet header. A SUNH header can be decompressed into a plain UDP/IP packet: The full Destination address and Source addresses are derived by prepending the SUPERp subnet prefix to the respective sixteen bit SUNH addresses, the Hop Limit and Traffic Class fields are directly mapped, the IPv6 flow label and/or the UDP source port are derived from the SUNH Flow Label. The Next header field allows carrying different protocols in the payload; this could just be a normal IP protocol number where one value is reserved for SUPERp.

The SUPERp subnet is a limited domain of connected nodes. The common SUPERp prefix is shared among all nodes of the domain. Packets with an SUNH network header are identified by a SUPERp EtherType that is distinct from IPv4 and IPv6. The use of a different EtherType also provides routing isolation so that SUNH packets won't escape the domain and packets from other domains won't enter. The payload of the SUNH is a SUPERp packet starting with a Packet Delivery Header (Next header indicates SUPERp).

When a switch switches on a SUNH destination address that entails performing a forwarding lookup on a sixteen bit address. The lookup can be efficiently done with a simple array lookup in

SRAM where the sixteen bit address is used as the index eschewing the need for a complex CAM or TCAM. The number of bits in the lookup, and hence the maximum number of nodes in the SUPERp domain, may be limited to reduce the size of SRAM needed.

The Packet Delivery Layer

The *Packet Delivery Layer*, or *PDL*, provides packet transport and reliability.

Connections

For two communicating endpoints a connection is established. Connection establishment and management is performed out-of-band by the *SUPERp Connection Manager*. Connections are represented globally by a 4-tuple of:

<IPaddress1>, *<IPaddress2>*, *<CID1>*, *<CID2>*

We list the numerically “lesser” address first to normalize the 4-tuple to avoid having two equivalent representations for a connection. *CID1* is the local connection identifier for the node with *IPaddress1*, and *CID2* is the local connection identifier for the node with *IPaddress2*.

From the perspective of a node, a connection is represented by:

<local_address>, <remote_address>, <local_cid>, <remote_cid>

When a packet is sent, the *remote_cid* is set in the *DCID* field of the Packet Delivery Header. A receiver interprets the *DCID* as *local_cid* and maps it to the state for the connection.

Packet Sequence Numbers (PSNs)

Each SUPERp packet sent contains a *Packet Sequence Number* (*PSN*). The per-connection PSN is initialized to zero and incremented for every packet sent (excepted for retransmission). PSNs are fundamental to the protocol and are used in flow control and retransmissions.

Flow control

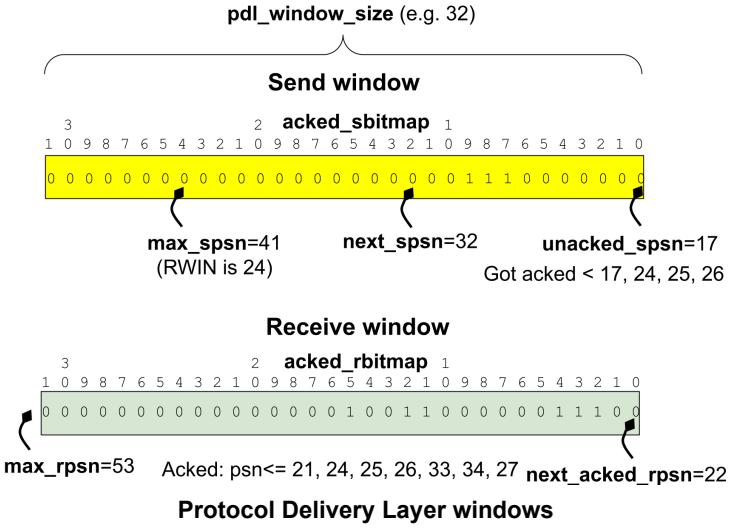
Packet Delivery flow control is based on a per connection *send window* of some number of packets. The receiver maintains a corresponding per connection *receive window* with the same number of packets. The send window is a sliding window with acknowledgements. The right edge of the window is the lowest PSN, **unacked_spsn**, that has not been acknowledged. The left edge, **max_spsn**, represents the greatest PSN that might be sent plus one. **max_spsn** is equal to **unacked_spsn** plus the minimum of the PDL window size and the last received RWIN:

max_spsn = **unacked_spsn** + **MIN**(**pdl_window_size**, **last_received_RWIN**)

PSNs must be sent in order without holes, the **next_spsn** variable indicates the next PSN to use for sending a packet. The right edge of the receive window, **next_acked_rpsn**, is the next

PSN the receiver expects to receive and acknowledge. The left edge of the window, **max_rpsn**, is the maximum PSN plus one that the receiver may receive and still be within the window. Packet Delivery flow control is symmetric, both sides of a connection maintain their own send and receive windows.

Each SUPERp packet includes an "**ACK PSN**" that is the greatest PSN acknowledged by a receiver (sent as **next_acked_rpsn** - 1). All PSNs through the ACK PSN are received and acknowledged by the node. If an ACK PSN is received that is greater than or equal to **unacked_spsn** then the window advances forward by the number of PSNs acknowledged.



Protocol Delivery Layer windows

SACK bitmap

Out-of-order packet delivery and fast retransmits are facilitated by a SACK bitmap. The SACK bitmap conveys PSNs that were received out-of-order and are being acknowledged. When a receiver receives a packet with a PSN that is not equal to **next_acked_rpsn**, but is otherwise in the window, the receiver sets a corresponding bit in the SACK bitmap (at bitmap index PSN minus **next_acked_rpsn**). The SACK bitmap is sent back to the peer in return packets. Upon receiving a packet with a non-zero SACK bitmap, a node may do a fast retransmit of the "holes" in the bitmap. Note that the internal bitmaps may be bigger in number of bits than thirty-two which is the size of the SACK bitmap on the wire so that the sent SACK bitmap is the lower thirty two bits.

NACKs

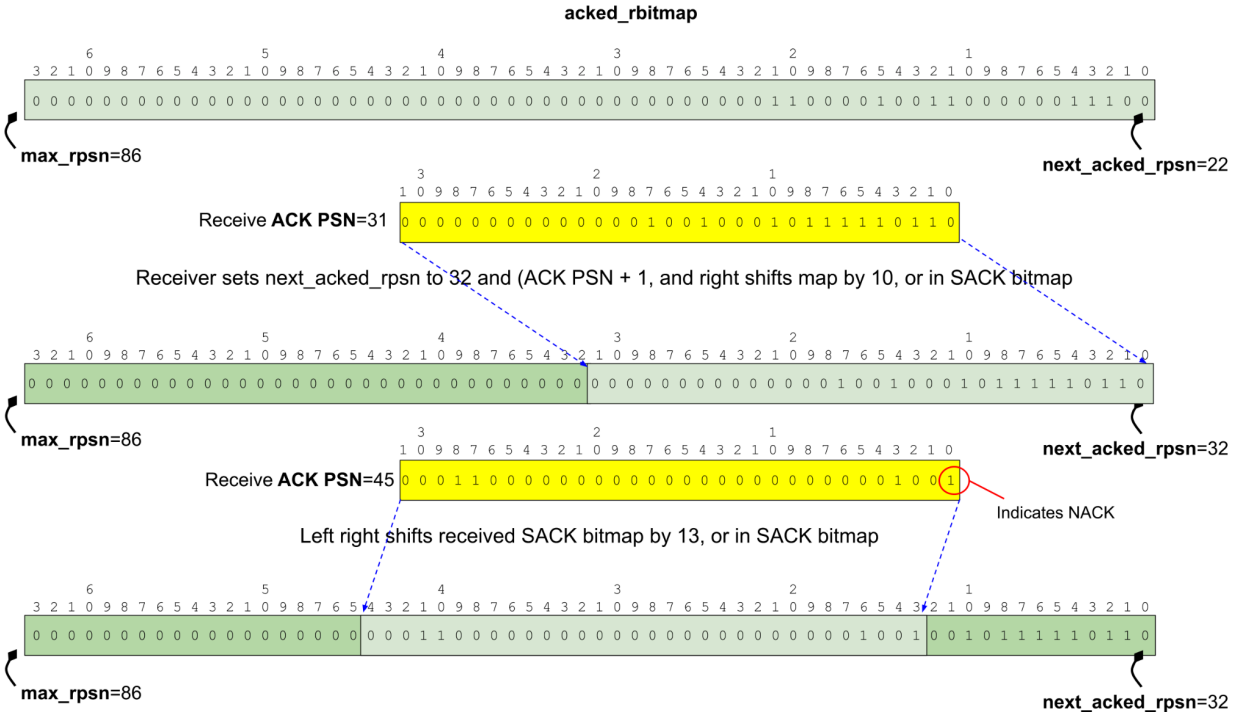
The first bit of the SACK bitmap is always sent as zero since it's the first unacked PSN. If it's set to one then that indicates a NACK packet. In this case the ACK PSN acknowledges a PSN out of order as a selective acknowledgement. If the NACK is within the window then the receiver interprets this as a negative acknowledgement where all all PSNs from next_acked_rpsn through the value of ACK PSN are not acknowledged and the value of ACK PSN is a selective acknowledgement. The SACK bitmap may also indicate other PSNs being selectively

acknowledged beyond the one in ACK PSN. The first bit in the bitmap refers to the PSN in ACK PSN.

The use of NACKs is optional and should only be used to make selective acknowledgments of PSN that are greater than or equal to ACK PSN plus thirty-two.

SACK bitmap processing

The diagram below illustrates receiver processing of a SACK bitmap for a normal ACK PSN and a NACK PSN.



Protection against wrap-around PSNs

At high data rates, sequence number wrap-around poses a potential problem. For SUPERp we address wrap-around with a large PSN space of 32-bits. At 1Tbit speed and assuming sixty-four byte packets, at most one point 1.9 billion packets can be transmitted per second. Assuming a Maximum Segment Lifetime of 1 second (which should be overkill in a Scale UP network), within the standard 2MSL time 3.9 billion packets can be sent. We adopt the rule that a PSN cannot be reused for two seconds, and based on this math it's not possible for the PSN to wrap around in less than two seconds for up to Terabit speeds.

This rule is sufficient protection in SUPERp against wrap-around PSNs. As an extra sanity check, when a PSN is being assigned to a packet the last send time of the PSN is checked that it is at least two seconds in the past (if it's not then an error is reported to the connection manager).

Incast congestion control

The Packet Delivery Layer has two models of in-cast congestion control:

1) Each node is configured with a window of outstanding packets of some known size (all receivers share the same window size). Each receiver allocates a number of maximum sized data buffers for each peer. For instance, if there are 16 nodes in the network, window size is 16 packets, and maximum payload size is 8K, then each node allocates ~2Mbytes of receive buffers. With this allocation, we are assured that each transmitter can send its full window without seeing an ACK (assuming no retransmissions).
2) A node implements receiver based congestion control. Each PDL header includes a window advertisement (**RWIN**). **max_spsn** can be capped by the last received RWIN as described above. Based on buffer availability, a receiver can modulate the receive window advertisements to throttle senders. The minimum advertised receive window is one to ensure that a sender can always send at least one packet to avoid deadlock.

Traffic classes

Each SUPERp connection may be assigned to a traffic class. Internally, the traffic class is used for mapping to buffer pools and classful packet transmission queues. Class information is carried in the Traffic Class bits of the encapsulating IP or SUNH headers.

Connection ordering

SUPERp connections may be *ordered* or *unordered*. For an ordered connection, packets are delivered to the Transaction Layer as they are received in order by their PSN. If a PSN is received out of order but still in the window, then the packet is held in the Packet Delivery Layer until all packets with lesser PSNs have been received. Ordered connections enable strict in order processing of transactions, although it's still at the discretion of the Transaction whether to process transactions and send replies in order.

For an unordered connection, valid packets are delivered to the Transaction Layer regardless of the PSN order. In this mode transactions and packets of individual transactions are expected to be processed out of order.

Connection ordering, i.e. ordered or unordered, is a property of each connection that is set at connection creation by the connection manager. The two directions of a connection may have different ordering, however for each direction both sides must agree on the order in each direction.

The Transaction Layer

The *Transaction Layer, TAL*, manages transactions. Transactions are initiated by an initiator to a target. A transaction is composed of some number of operations. An opcode in the Transaction Layer header specifies the specific operation to perform. A transaction may be composed of

multiple operations. The operations of a transaction may be split over multiple packets, and a single packet may carry multiple transactions. All operations of a transaction share the same opcode.

Transactions

An initiator initiates a transaction by making a request (this may be an RDMA operation started on a Queue Pair for instance). A transaction state is created. The request is sent to the target on a connection in one or more SUPERp packets. The request will be sent in multiple packets if the data for the request is bigger than one MTU. Each packet is marked with a transaction identifier (**XID**) and sequence number (**Seqno**) that is the packet number of the transaction. The last packet sent in the sequence is marked with **eom** for end-of-message. The initiator's XID is initialized to zero and incremented for every new transaction. The **Seqno** is initialized to zero at the start of each transaction and incremented for every sent packet of the transaction.

When the target receives the first packet of a transaction from an initiator, it creates its own transaction state. If the transaction is contained in one packet (**Seqno** is zero and **eom** is set) the request is processed. If the transaction is contained in multiple packets then the target waits to receive all the packets before completing the request. Depending on the operation, the sub-operations in each packet may be processed before receiving all packets for a transaction (e.g. an RMA write with early data placement).

A SUPERp packet may contain multiple operations. A single transaction is defined by the packets of the transaction and the operations within each of the packets. Operations may have included data in the packet.

When the target has received a full initiator request, it completes the operation and commences to send a reply back to the initiator. The reply is sent in one or more SUPERp packets, each with a sequence number (**Seqno**) starting from zero and the last packet is marked **eom**. The **XID** in reply packets is set to the initiator’s XID for the transaction and a “target reply” Opcode is set.

Once all the packets for a transaction have been sent and acknowledged by the target, the transaction is “delivered” to the ULP. In the case of a read operation this would entail sending a message to the calling application or ULP. In the case of a write this may entail the target sending an optional “write complete” message to the peer. When the initiator gets a “write complete” message it can send a completion message to the application or ULP.

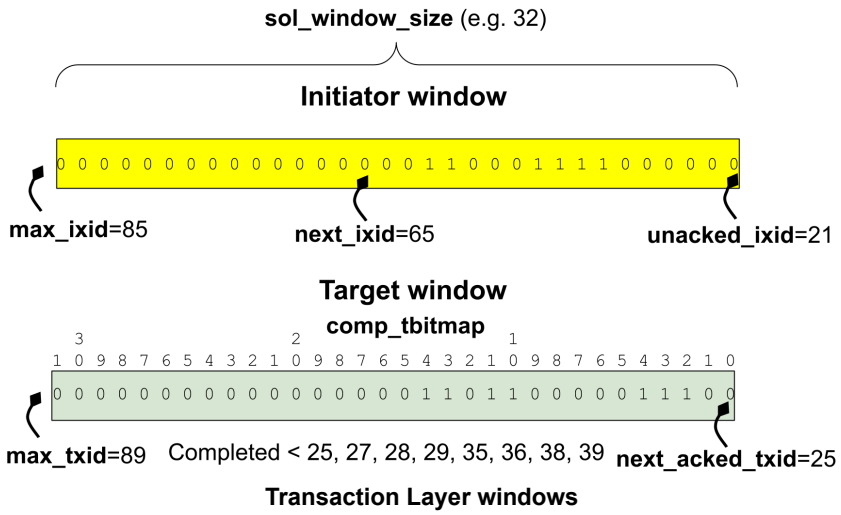
Note that all operations of a transaction must have the same operation type. Other than that requirement, processing of each operation in a transaction is idempotent. For instance, if a transaction contains multiple write operations, each operation has its own target address to write and can be of a different length (for operations in different packets). The target must validate each operation to check for permissions to write to the memory per the address and length of the packet without regard to other operations in the transaction.

Flow control

Similar to the Packet Delivery Layer, the Transaction Layer has flow control, however it's simpler. Flow control at this layer is only to manage the number of outstanding transactions on a connection, it is not used for packet level flow control or reliability. Each endpoint of a connection maintains two windows based on transaction identifiers: an initiator window and a target window. The Transaction Layer does not employ bitmaps sent on the wire.

The right edge of the initiator window, **unacked_ixid**, is the lowest transaction identifier that is unacknowledged. The left edge of the window is the greatest transaction identifier plus one that is within the window, **max_ixid**. The **next_ixid** variable is the next XID that may be used for a new transaction. If the **next_ixid** is equal to **max_ixid** then no new transactions may be started pending reception of an XID ACK that shifts the window. The right edge of the target window, **next_acked_txid**, is the next XID the target expects to complete and acknowledge. The left edge of the window, **max_txid**, is the maximum XID that the target may receive and is still within the window. The target maintains a bitmap to track transactions completed out-of-order.

Each SUPERp packet includes an "**ACK XID**" that is the greatest XID acknowledged by the target (sent as **next_acked_txid** - 1). All XIDs up to and including the ACK XID have been completed and acknowledged by the target. When an ACK XID received by an initiator is greater than or equal to **unacked_ixid** then the window moves forward by the number of XIDs acknowledged.



OOO transactions

Packets for transactions and transactions themselves may be completed out-of-order within a connection. For an ordered connection packets are guaranteed to be delivered in PSN order to the Transaction Layer, for unordered connections packets may be delivered to the Transaction Layer out of order. In the former case, the Transaction Layer may choose to process transactions out of order even though the connection is ordered. If out-of-order transaction processing is allowed, that is either the connection is unordered or the Transaction Layer might

process transactions out of order, then if there are dependencies between two transactions such that out-of-order processing would lead to incorrect results then a “memory barrier” can be used. At a memory barrier an initiator pauses initiating new transactions until all previous transactions are complete. A “write memory barrier” would pause just memory write transactions.

XID interpretation

The XID in a Transaction Layer header is interpreted depending on the Opcode in the header.

- If the Opcode is for an initiator request then XID is that for the request
- If the Opcode is for a target reply then XID is set the original XID in the initiator’s request

Operations

Zero or more Operation Headers follow the Transaction Layer header. The number of operations in a packet is specified in the **num_ops** field of the Transaction Layer header. The operation type of all operations in a packet is specified in the **opcode** field of the Transaction Layer header (all operations in a packet are for the same operation type).

The format and content of each operation header is operation specific. Operations may have associated data as part of the operation. The data for operations follows the last Operation header. The data is divided up into equal sized blocks, one block for each operation header.

Comparison to other RMA-like or HPC transports

Why Ethernet?

For Scale Up networks there are a few choices. UALink and NVLink are specialized high speed interconnects and protocols for connecting AI accelerators. Another candidate would be to use PCIe directly with a PCIe switch. These alternatives have a few advantages and have seen some deployment, however going with venerable Ethernet technology has many advantages.

Ethernet is a well established technology and high speed links and high-capacity switches are well deployed. The ecosystem is well developed with well-understood operational methods. Multiple industry groups are currently involved in developing networking technologies for AI based networks that extend Ethernet or use some of its components as building blocks. Ethernet also has very good scalability in the number of attached devices.

One of the concerns with Ethernet compared to UAlink is higher switch latency and power. This is due to FEC and the need for a CAM to perform lookups on long addresses (MAC addresses or IP addresses). Ethernet currently uses four-way interleaved codewords for FEC whereas UAlink uses one-way or two-interleaving which results in lower latency. IEEE is defining standards for short run Ethernet that could use one-way or two-way interleaving to reduce latency. The CAM lookup cost can be eliminated by defining a small destination identifier that could be used as a simple index to a lookup table (SUNH defines such an identifier).

Given the advantages and ubiquity of Ethernet, we have selected Ethernet as the basis for the Scale Up protocol SUPERp defined in this document.

Broadcom's Scale Up Ethernet (SUE)

Broadcom's Scale Up Ethernet Transport (SUE-T) is another protocol defined for scale up Ethernet however it has a number of deficiencies and potential issues:

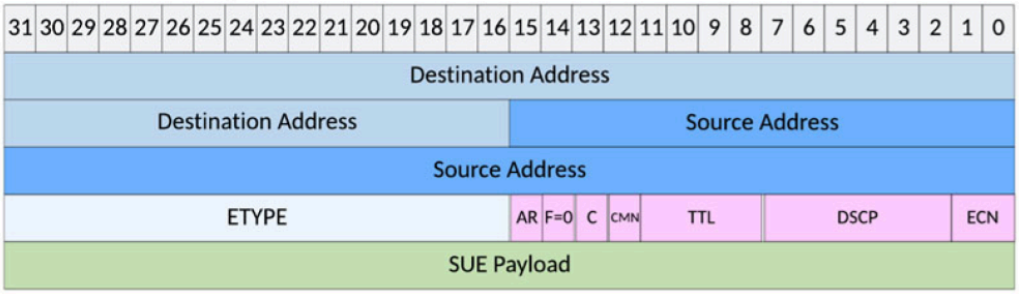
- SUE-T is incomplete in that it doesn't define any protocol beyond a packet delivery layer, whereas SUPERp defines an extensible transaction and operations layer
- The AFH headers in SUE-T are incompatible with deployed switches and NICs because of modifications to the Ethernet header. SUNH in SUPERp does not modify the Ethernet headers and retains compatibility with legacy switches and NICs
- SUPERp specifies connection management, whereas SUE-T does not
- SUE-T employs Go-back-N for retransmission, SUPERp includes a Selective ACK bitmap for more precise and efficient retransmissions
- SUPERp eschews the **ver** and **op** fields which are unnecessary
- SUPERp defines error handling and recovery procedures, SUE-T does not
- SUPERp allows out of order delivery, SUE-T does not
- SUE-T does not consider wrap-around sequence numbers, SUPERp does
- SUE-T hacks up the Ethernet header, SUPERp specifies a small new network layer header that allows L3 switching and doesn't touch the Ethernet header itself
- SUE-T mentions the ability to pack multiple operations into a single packet but does not specify how to do that like SUPERp does

AFH headers

SUE-T defines "AI Forwarding Headers" of AFH. These repurpose bits in the Ethernet header by applying the Structured Local Address Plan in IEEE 802.c-2017.

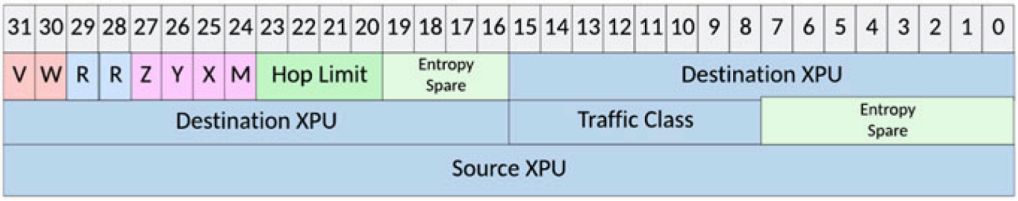
The AFH Gen 1 format uses the standard Ethernet header but fewer bits may be used from the MAC addresses to perform forwarding lookups thereby reducing complexity of the lookup engine. The EtherType specifies AFH Gen 1. A two byte shim header is defined to carry some fields from the IP header, a second EtherType presumably following the shim header would disambiguate the Ethernet payload (the two bytes would also be needed to maintain expected alignment of the Ethernet payload.

An AFH Gen 1 format with the shim header is shown below.



In the AFH Gen 2 format the forwarding information is reduced to 6 or 12 bytes. The remaining bytes in the Ethernet destination address and source address can be used for user-defined functionality. The Ethertype field is used to disambiguate multiple transport headers.

The first twelve bytes of AFH Gen 2 header with 12 bytes of forwarding information are below.



AFH versus SUNH

The goal of AFH is to reduce the overhead of packets in AI. Specifically, it seems that the primary intention is to eliminate the IP and UDP headers that would otherwise be needed. In both AFH Gen 1 and Gen 2 end to end addressing is via XPU addresses that are eight sixteen bits or eight bits. Other pertinent information from the IP/UDP header (Hop Limit, Traffic class, flow label) are either in the AFH Gen 1 shim header or repurposed fields in the Ethernet header in AFH Gen 2. The AFH headers effectively replace the IP/UDP headers in SUE and therefore AFH Gen 1 saves twenty-four bytes in IPv4 and forty-four bytes in IPv6, and AFH Gen 2 would save twenty-eight bytes in IPv4 and forty-eight bytes in IPv6. Those savings come at the cost of non-standard modifications to the Ethernet header that require explicit support in both Ethernet switches and end hosts.

The alternative proposed in the document is SUNH. This is a compressed network header that is treated as its own Ethernet payload with its own EtherType. In particular, SUNH *does not require* changes to either switches or end hosts. The format works with commodity Layer 2 switching as there are no changes to the Ethernet header which is transparent to L2 switching. Packets may be Layer 3 switched with switch support that is very similar to how IP is L3 switched. The switch simply needs to identify the SUNH EtherType, extract the sixteen bit destination address for a fixed location in the Ethernet payload, and then perform the forwarding lookup. SUNH contains all the information needed by the switch for forwarding network layer packets (ECN marking, a flow label for ECMP, time to live, and traffic class).

SUNH replaces the IP/UDP header needed for SUPERp with an eight byte header. So for IPv4 SUNH saves twenty bytes and for IPv6 SUNH saves and forty bytes. The net difference between AFH and SUNH is that SUNH has four more bytes of overhead than AFH Gen 1 and eight more bytes than AFH Gen 2. We believe that difference is insignificant in the grand scheme of things, and that benefits of a much less invasive solution that retains compatibility with deployed hardware outweigh a slight reduction in packet overhead.

Transactions and operations packing

SUE-T defers on the structure of the “SUE payload” and how commands are sent on the wire. From the specification: “The set operations and structure of the commands are XPU-specific and transparent to SUE”. Without defining this layer, SUE-T is only solving part of the problem. And while SUE-T offers no specifics on the format or contents of commands, it does specify that commands may be packed together into a single packet but again doesn’t give the details on how this could efficiently work. In particular, it seems like unpacking commands would require parsing deep inside the packet to find the commands which is very inefficient.

SUPERp defines both a transaction layer (for transaction flow control) and explicit operations with an extensible format. Having a complete stack allows optimizations to work across levels instead of doing it piecemeal like SUE does. SUPERp also supports operation packing, but unlike SUE-T the operations, commands in SUE terminology, are in packet headers followed by data blocks for the operations yielding a much more efficient format to process.

UEC’s UET

The Ultra Ethernet Transport (UET) from the Ultra Ethernet consortium is a protocol for Scale Out networks. It is feature rich, however quite complicated and not particularly appropriate for Scale Up networking. SUPERp advantages over UET are:

- UET has many different packet formats with variable sizes, whereas SUPERp has just one format of a fixed size for efficient processing
- UET has an elaborate scheme for congestion control, whereas SUPERp has fairly rudimentary mechanisms. Scale Up networks tend to have simple topologies with homogeneous nodes such that elaborate congestion control would be overkill
- UET has many different modes of operation, request types, and different profiles for broad applicability. SUPERp is far simpler, but still extensible as it defines a base protocol for reliability and transactions upon which various operations are supported. .
- SUPERp does not force a specific ULP API like UET does with *libfabric*
- UET employs UDP/IP encapsulation, SUPERp similarly employs UDP/IP encapsulation but also defines a lightweight SUNH encapsulation that maintains key properties of a Layer 3 protocol
- UET has a concept of unordered and ordered connections, in SUPERp packets and transactions may be processed out-of-order with memory barriers when ordered semantics are required

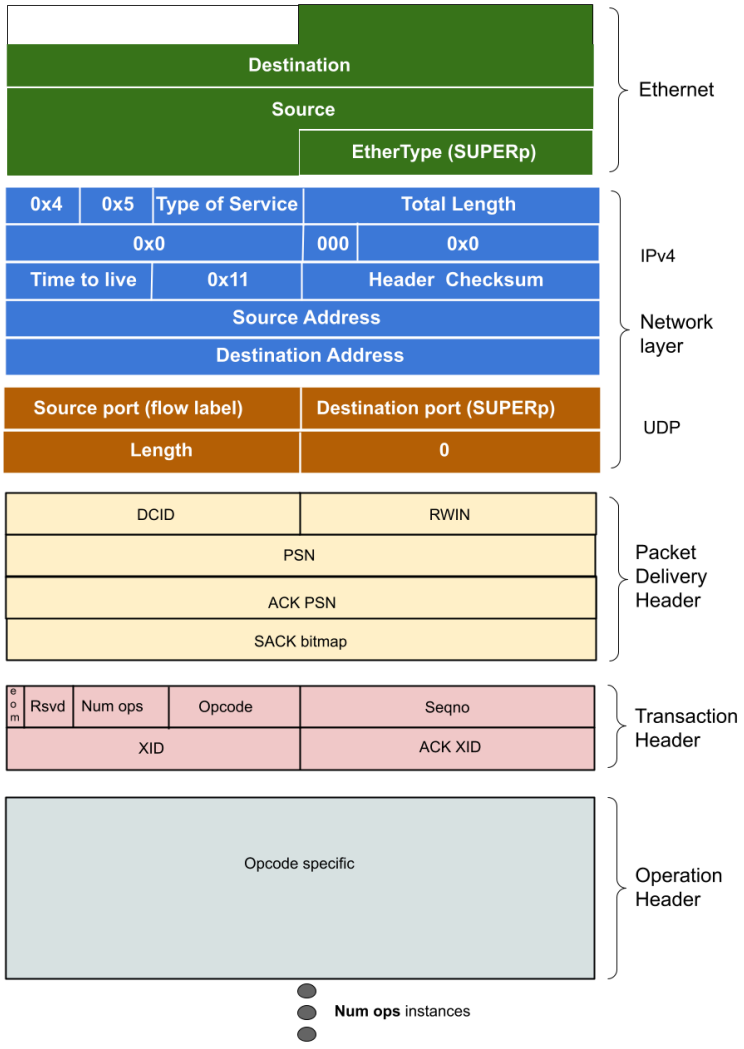
Google’s Falcon

Falcon is another Scale Out designed to be a reliable transport for RDMA Operations and NVMe commands. It is simpler than UET, but more complex than SUE. SUPERp advantages over Falcon are:

- Like UET, Falcon has an elaborate protocol for congestion control including protocol fields to carry congestion information, SUPERp keeps this simple for Scale Up
- Falcon has two SACK bitmaps in a packet, whereas SUPERp only needs one
- Falcon headers are variable length depending on the packet type (makes header/data split more complicated), all SUPERp headers are a common fixed length
- Falcon employs UDP/IP encapsulation, SUPERp uses UDP/IP or SUNH encapsulation
- Falcon has a very elaborate and rich scheme for buffer management and resource pools, SUPERp assumes a simple model of pre-allocated resources for all connections
- Falcon has NAKs. SUPERp does not need them (the SACK bitmap suffices)
- Falcon requires timestamps sent on the wire for protection against wrapped sequence number (sent in ESP or PSP headers), SUPERp doesn’t employ timestamps
- Falcon is a big endian protocol whereas SUPERp is little endian for performance

Protocol format

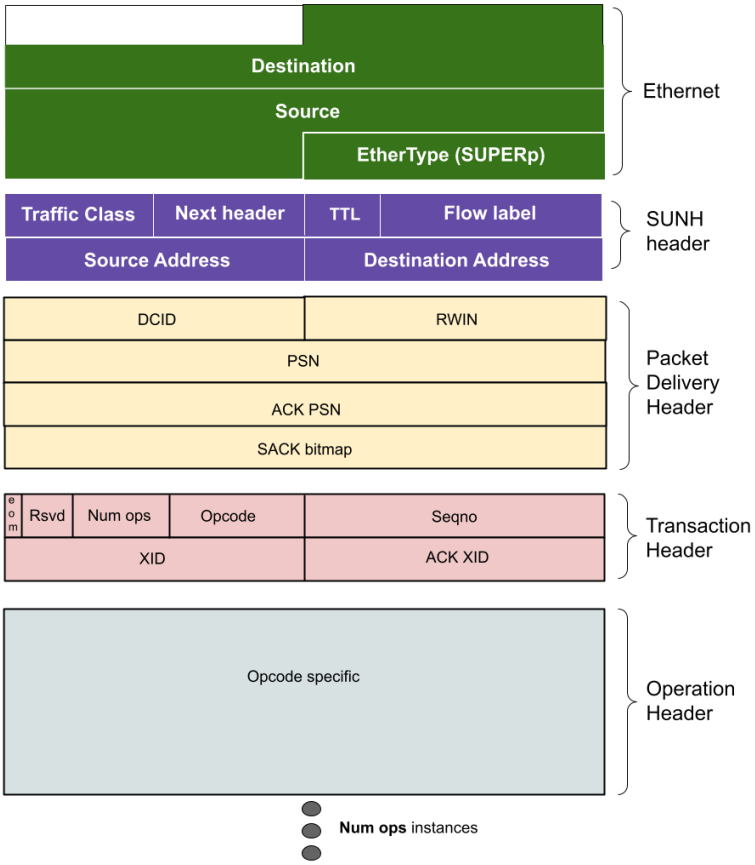
SUPERp messages, starting with the Packet Delivery Layer, share a common protocol headers format. The format of SUPERp encapsulated in UDP/IP is shown below:



SUPERp encapsulated in UDP/IP.

XDPnet

The format of SUPERp encapsulated in SUNH is shown below:



SUPERp encapsulated SUNH (SUPERp Network Header)

Fields are:

Ethernet header

- **Destination address**: Ethernet address of destination (normal semantics)
- **Source address**: Ethernet address of source (normal semantics)
- **EtherType**: For UDP/IP encapsulation this is 0x0800 for IPv4 or 0x86DD for IPv6. For SUNH this is pending assignment by IEEE, Experimental types 0x88B5 and 0x88B6 can be used for protocol development.

Network header

- UDP/IP encapsulation
  - Fields in the IPv4 or IPv6 header are set normally. The IPv6 flow should be set as a flow entropy value for the connection
  - **UDP Source port**: Flow entropy value for the connection
  - **UDP Destination port**: Set to the SUPERp port number pending assignment by IANA
  - **UDP Length**: Set to the length of the packet (no surplus area)
  - **UDP checksum**: optional for IPv4 and must be set for IPv6

- SUNH encapsulation
    - **Traffic Class**(8 bits): Differentiated services/ECN
    - **Next header** (8 bits): Value reserved for SUPERp
    - **Hop Limit** (4 bits): Hop limit like in IPv6
    - **Flow label** (12 bits): Flow label
    - **Source address** (16 bits): Set to the host bits of the IP address for the source
    - **Destination address** (16 bits): Set to the host bits of the IP address for the destination

Packet Delivery header (16 bytes)

- **DCID** (16 bits): Destination connection Identifier. This is the identifier of the connection used by the receiver to lookup the context for the connection.
- **RWIN** (16 bits): Receive window advertisement (window minus one)
- **PSN** (32 bits): Packet sequence number
- **ACK PSN** (32 bits): The PSN being acknowledged
- **SACK bitmap** (32 bits): The selective acknowledgment (SACK) bitmap

Transaction Header (8 bytes)

- **eom** (1 bit): End of message
- **rsvd** (3 bits); Reserver
- **Num ops** (4 bits): Number of operations in the message
- **Opcode** (8 bits): Opcode for the operations in the message
- **XID** (16 bits): Transaction identifier (semantics based on opcode)
- **Seqno** (16 bits): Packet sequence number for transaction
- **ACK XID** (16 bits): Acknowledged transaction identifier

Operation Header (variable length depending on Opcode) (Occurs **Num ops** times)

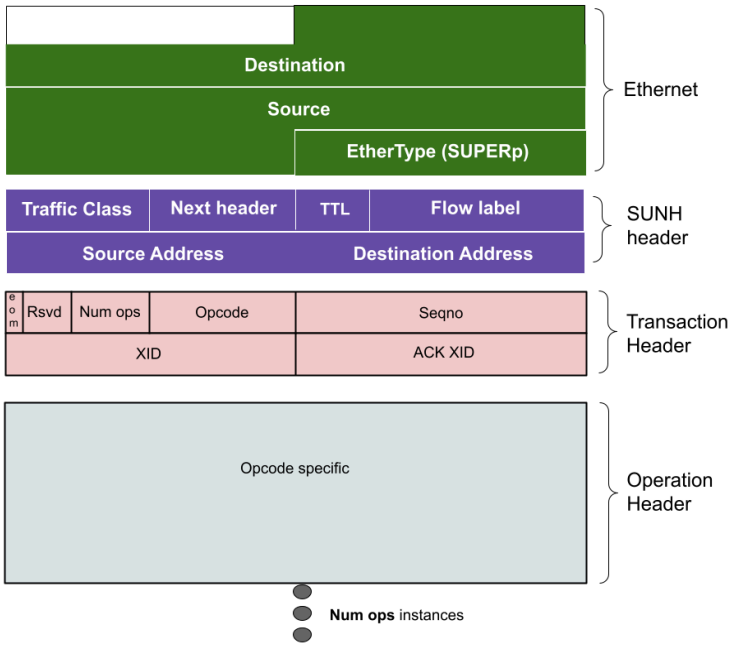
Notes:

- Multi-byte fields in the PDL, TAL, and operation headers are **little endian** (not network byte order).
- There is no payload length field in SUNH or PDL. The length is deduced by the reported packet length.
- The Packet Delivery Header and Transaction header are fixed size and present in all SUPERp packets. The Ethernet Header may include VLANs.
- With a single operation using SUNH encapsulation, all headers fit into a sixty-four byte cacheline (i.e. a parsing buffer of sixty-four byte suffices).
- The Packet Delivery Header should be aligned to eight bytes in the receive buffer. As shown above, the Ethernet header starts at a two byte offset to satisfy this requirement. When VLANs are in use the Ethernet header offset should be adjusted accordingly.
- End nodes may be configured to do blind header/data split on a packet with SUNH EtherType if all packets contain exactly one operation. The split happens at the end of the Operation header. The header data goes to header buffers and the payload goes to packet buffers. If there is more than one operation in a packet then header data split will

need more intelligence. In order to easily distinguish the cases in a simple device, a separate Ethertype could be used when there is exactly one operation in packet.
- The PDL window may be larger than thirty-two bits. A node can maintain an internal SACK bitmap larger than that and set SACK bitmap in a packet from the low-order bits

SUPERp-lite

If the Ethernet layer is completely lossless, reliable, and provides sufficient flow control then the Packet Delivery Layer may be eliminated in SUPERp packets within the SUPERp domain. The SUPERp-lite format without PDL is shown below



In SUPERp-lite there is no connection identifier in packets, instead connections are identified by the SUNH address so that there is at most one connection between any two nodes. Within a SUPERp domain employing SUPERp-lite connections are represented globally by a 4-tuple of:

<SUNH Address1>, *<SUNH Address2>*

A receiver can use the source address for connection lookup in the same manner that it would use the DCID in the PDL header.

The flow control mechanisms of the Transaction Layer remain operative and ordered and unordered transactions are supported. An initiator should use a timer to ensure that transactions are acknowledged, for instance the connectivity might be broken in the network. If a timer fires then that is a fatal error that must be reported to the connection manager.

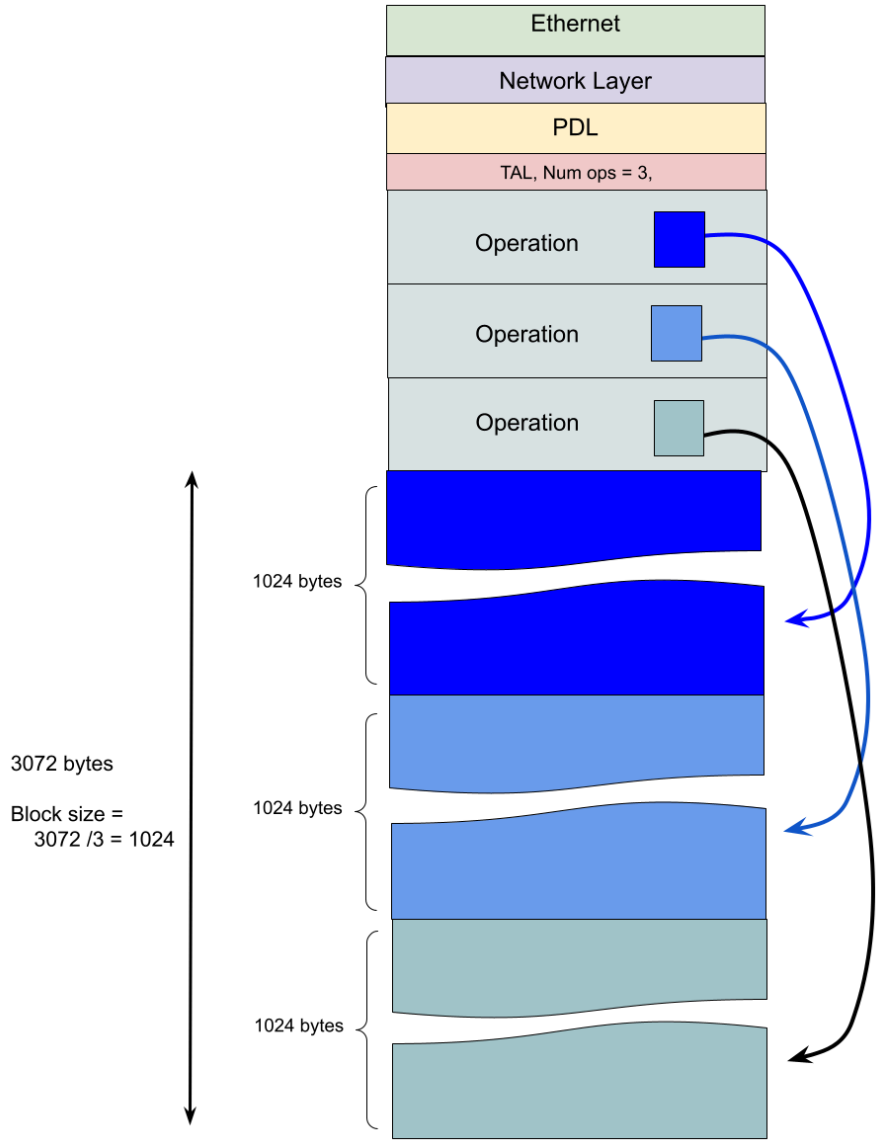
Other than these modifications, the Transaction Layer and operations work the same way in SUPERp-lite as they do in plain SUPERp.

SUPERp packets

SUPERp packet consists of an Ethernet header, a network layer header of either UDP/IP or SUNH, a Packet Delivery Layer header, a Transaction Layer header, zero to fifteen Operation headers, followed by some number of data blocks corresponding to operations that have included data. All data blocks within a packet have the same size. The block size is calculated as the length of the data region following the last operation divided by the number of operations:

Block_size = (*Packet_length* - *Offset_of_data_region*) / *Num_ops*

as indicated by the **Block_size** in the Transaction Layer header. An example of a SUPERp packet is illustrated below:



Example SUPERp packet. The packet contains four operations (**Num ops** equals 3), each operation has an associated data block of 1024 bytes

XDPnet

Small packets

Ethernet frames have a minimum size of sixty-four bytes such that the minimum Ethernet payload size is forty-six bytes. Care must be taken to ensure that the minimum requirement is meant.

The SUNH header is eight bytes in size, the Protocol Delivery Layer is sixteen bytes, and the Transaction Layer is eight bytes. If a SUPERp packet is set without any operations then the Ethernet payload length is thirty-two bytes which is fourteen bytes short of the minimum. Fourteen zero bytes can be safely appended after the TAL header. The receiver will ignore the padding bytes in that case.

If a SUPERp packet includes at least one operation then the Ethernet payload length is at least fourty-eight bytes so the minimum requirement is met.

In a SUPERp-lite packet with one operation and no data the Ethernet payload length is thirty-two bytes which is fourteen bytes short of the minimum size. To avoid any misinterpretation that the minimum block size for operations that have associated data in the packet is sixteen bytes. A sender MUST ensure that this minimum block size requirement is met, and a receiver MAY check that the block size in a packet is greater than or equal to sixteen bytes and if it's not then a transaction error is reported and the error is also reported to the connection manager who may take further action.

Protocol operations

In this section we cover the core protocol operations of SUPERp.

Packet Delivery Layer

Nodes maintain some variables for the Packet Delivery Layer:

- **pdl_window_size**: The per connection window size set by the connection manager. The window size is symmetric on both sides of the connection
- **next_spsn**: The next PSN to set in a packet
- **unacked_spsn**: The lowest PSN that has not been ACKed
- **max_spsn**: The maximum PSN plus one that may be sent in the window. This is **unacked_spsn** + **MIN**(**pdl_window_size**, *last_received_RWIN*)
- **acked_sbitmap**: A sender bitmap of PSNs ACKed out-of-order (via received SACK bitmap). Base is **unacked_spsn**
- **spsn[pdl_window_size]**: State for each sent PSN in-flight. Indexed by PSN % **pdl_window_size**. Information includes:
    - A reference to the transaction state
    - A reference to the packet data sufficient to reconstruct the packet if it's retransmitted

    - A timestamp when the packet was sent
    - The number of retransmissions

- **next_acked_rpsn**: The lowest unacknowledged PSN
- **acked_rbitmap**: Bitmap of PSNs received out-of-order. Base is **next_acked_rpsn**
- **max_rpsn**: The maximum PSN plus one that may be received in the window. This is **next_acked_rpsn** + **pdl_window_size**

Sending a packet

A new packet may be sent if the send window is not full. This is true if **next_spsn** is less than **max_spsn**. If the send window is not full then procedures for sending a new PSN are:

1. The **next_spsn** is assigned to the packet. The **spsn** state is updated to list the associated transaction ID, the starting offset and length of transaction data (needed for retransmitting the packet for the PSN), and the time the packet is being sent. The **spsn** array is indexed by **next_spsn** % **pdl_window_size**. **next_spsn** is incremented.

2. A PVbuf is created that holds the headers and packet data

    a. The packet data is cloned from the original PVbuf for the transaction if the transaction includes data to send (e.g. a write request or read response data). The cloned data is from the starting offset in the original PVbuf for the current packet being sent though the length of the data being sent (either MTU length or to the end of the data).

    b. The “headers” buffer is allocated with a sixty-four byte pbuf that contains an Ethernet Header, Packet Delivery Header, and a Semantic Operations header. The headers pbuf is prepended to the PVbuf to create a fully qualified packet. The buffer can be initialized by copying from a template maintained in the connection context. The template includes constant fields for the connection (the whole Ethernet Header, and the DCID), as well as fields set in receive path processing (ACK PSN, RWIN, the SACK bitmap, and the ACK XID). The rest of the fields are set with per packet specific values. Since packet creation is a relatively expensive and frequent operation, a *Packet Builder* accelerator may be developed, this would include creating the headers pbuf from template and the payload data from an input PVbuf.

3. The packet is queued for delivery on the appropriate device queue. If there is no available space on the device queue then the enqueue is deferred.

4. Eventually the packet is sent and a completion is reported by the device queue. The PVbuf may be freed once the device acknowledges the packet has been sent, or the implementation may wait until the PSN is ACKed to free the Pvbuf. In the former case, If the PSN must be retransmitted then a new packet is created; in the latter case the same PVbuf can be used if the packet is retransmitted.

Retransmitting a packet

If a PSN is to be retransmitted, the **spsn** entry for the PSN is accessed to provide the necessary parameters. The send time is updated and the number of retransmissions is incremented. The packet is sent following the procedures above start from #2.

Receiving a packet

When a packet is received it is processed. The procedure for receiving a packet are:

1. The PSN is checked against the receive window.
    a. If PSN < **acked_rbitmap** then it's to the right of the receive window so the packet is an old PSN and the packet is dropped with no further action
    b. If PSN >= **max_rpsn** is greater than the left edge of the window then the packet is out-of-window. This is considered an error. The packet is dropped and a message is sent to the connection manager that might take further action (minimally the event should be logged).
    c. If PSN == **acked_rbitmap** the receive window moves with these procedures:
        i. The number of PSNs to be ACKed is one plus the number of consecutive bits set **acked_rbitmap** starting at index 1
        ii. **next_acked_rpsn** is incremented by the number of PSNs being acknowledged
        iii. The **next_acked_rpsn** bitmap is right shifted by the number of PSNs acknowledged
    d. The PSN is in the window but not equal to **next_acked_rpsn**. The bit in the **next_acked_rpsn** bitmap at index PSN - **next_acked_rpsn** is set and no further action is taken
2. If it wasn't dropped, the packet is processed by TAL processing. Processing depends on the opcode.

Packet Delivery Layer Flow control

Sending flow control

Flow control from the senders perspective works as follows:

1. If a new packet is being sent (not a retransmission of a PSN) and the **next_psn** is less than **max_spsn** then the PSN may be reserved for the packet. If **next_spsn** is equal to the last available packet in the send window, **max_spsn** - 1, that it may only be reserved if the ACK PSN sent that would be set is greater than **unacked_spsn,** the ACK PSN in the first packet in the window (this is needed to prevent deadlock where the send windows are full on both sides and there's no window availability to send an ACK). The current time is recorded in the **spsn** context for the PSN.
2. If this is the first PSN being sent in the window then the retransmit timer is started.
3. When a packet is received from the peer, ACK processing is performed

a. The ACK PSN is checked.
    i. If ACK PSN is greater than or equal to **unacked_spsn**, and less than **next_spsn**, then it is valid. Proceed to b.
    ii. If ACK PSN is less than **unacked_spsn** then the ACK PSN contains no new information so it is ignored with no further processing
    iii. If ACK PSN is greater than or equal to **next_spsn** then the ACK PSN is out-of-window. That is considered an error. The packet is dropped and a message is sent to the connection manager that can take further action
b. ACK PSN is valid and bit 0 of the bitmap is not set
    i. PSNs from **unacked_spsn** through the ACK PSN are acknowledged
    ii. **unacked_spsn** is incremented by the number of PSNs acknowledged
    iii. Bitmap **acked_sbitmap** is right shifted by the number of PSNs acknowledged
c. If the window was previously full then pending transactions are checked to send new packets. The per connection retransmission timer is reset as follows:
    i. If there are no longer any unacknowledged PSNs then the retransmission timer is disabled
    ii. Otherwise, the retransmission timer is reset to timeout based on the first unacknowledged packet:
        timeout = config.RTO_timeout - (current_time - packet_sent_time)
d. ACK processing: Process the received SACK bitmap if bit 0 of the bitmap is not set
    i. The **acked_sbitmap** bitmap is or'ed with the SACK bitmap in the packet (**acked_sbitmap** |= SACK bitmap).
    ii. If the SACK bitmap is non-zero then the peer is acknowledging packets received out of order. The node may perform a fast retransmit:
        1. The node may immediately retransmit unacknowledged packets that precede an acknowledged PSN up to a configurable limit of number of packets.
        2. The node may reset the retransmission timer to a small timeout to avoid small delays due to variances that created out-of-order packets
e. NACK processing: Process the received SACK bitmap if bit 0 of the bitmap not set
    i. The the received SACK bitmap is left shifted by ACK PSN value minus **next_acked_rspn**
    ii. **acked_sbitmap** bitmap is or'ed with the shift SACK bitmap in the packet (**acked_sbitmap** |= SACK bitmap).
    iii. The node may perform a fast retransmit:
        1. The node may immediately retransmit unacknowledged packets that precede an acknowledged PSN up to a configurable limit of number of packets.

2. The node may reset the retransmission timer to a small timeout to avoid small delays due to variances that created out-of-order packets

4. When the retransmission timer fires the following procedures are performed:
    a. The **spsn** state for **unacked_psn** is checked. It the number of retransmissions exceed a configurable threshold then the connection is declared broken
    b. The PSN is retransmitted and the number of retransmissions is incremented
    c. Unacknowledged PSNs, per bits not being set in **acked_sbitmap**, following **unacked_spsn** may be retransmitted if their send time is a configurable delta for the send time of **unacked_spsn**
    d. The per connection retransmit timer is reset

Receiver flow control

Flow control from the receiver's perspective works as follows:

1. When the receiver receives a PSN there are four possibilities.
    a. The PSN lies to the left of the window (PSN >= **max_rspn**). The sender has sent a packet with a PSN that is greater than the extent of the window. This is considered an error. The packet is dropped and a message is sent to the connection manager to decide on an action (minimally the event should be logged)
    b. The PSN lies to the right of the window (PSN < **next_acked_rspn**). The PSN has already been ACKed so it is considered a duplicate. The PSN ACK and XID may still be processed (e.g. these packets might be retransmissions that would advance the windows)
    c. The PSN is equal to the right edge of the window (PSN == **next_acked_rpsn**). The sliding window moves following these procedures:
        i. The number of PSNs to acknowledge is one plus the number of consecutive bits set in the **acked_rbitmap** starting from index 1
        ii. **next_acked_rpsn** is incremented by the number of PSNs to acknowledge
        iii. **acked_rbitmap** is right shifted by the number of PSNs to acknowledge
    d. The PSN is within the window but not equal to **next_acked_rpsn** (**max_rpsn** < PSN < **next_acked_rpsn**). The corresponding bit in the SACK bit is set in **acked_rbitmap** and no other action is taken (bitmap index is PSN - **next_acked_rpsn**)
2. Whenever a node sends a packet, the PSN ACK is set to the **next_acked_rspn** minus one. The low order thirty-two bits of **acked_rbitmap** are set in the SACK bitmap field
3. Pure ACKs are sent when there are no packets being sent to piggyback ACKs. There are are two instances for sending pure ACKs:
    a. *N* consecutive PSNs have been received with out sending back an ACK (these are so-called "stretch ACKs" in TCP parlance

    b. A delayed ACK timer fires. When a new PSN is received a delayed ACK timer is started for the connection for some configurable timeout. When the timer fires an ACK is sent. The delayed ACK timer is reset any time an ACK is sent so it's likely the only time the delayed ACK timer fires as at the tail of a communication
4. A receiver may throttle the sender by returning a received window advertisement (RWIN) that is less than **pdl_window_size**.

Transaction Layer

Nodes maintain some variables for the Transaction Layer.

- **tal_window_size**: The per connection transaction window size set by the connection manager. The window size is symmetric on both sides of the connection
- **next_ixid**: The next XID to send in a new transaction request
- **unacked_ixid**: The lowest XID that has not been ACKed by the target
- **max_ixid**: The maximum XID plus that may be used in a new transaction (equals **unacked_ixid** + **tal_window_size**)
- **ixid[tal_window_size]**: State for each transaction in-flight
    - A reference to the packet data sufficient to reconstruct the packet if it's retransmitted
    - The number of packets
    - Glue to communicate with the ULP (like RDMA QP)
- **retired_ibitmap**: Bitmap of retried XIDs

- **next_acked_txid**: The lowest unacknowledged XID
- **max_txid**: The maximum XID plus one that may be received in the window. This is equal to **next_acked_txid** + **tal_window_size**
- **comp_tbitmap**: Bitmap of completed out-of-order XIDs
- **txid[tal_window_size]**: State for each target transaction in-flight. Indexed by XID % **tal_window_size**. Information include:
    - **num_packets**: the number of packets for the transaction (initialized to zero and learned from **eom**)
    - **seqno_bitmap**: bitmap of received sequence numbers

Flow control

Similar to the Packet Delivery Layer, the Transaction Layer has flow control, however it's for a different purpose and the procedures are much simpler. Flow control at this layer is only to manage the number of outstanding transactions on a connection, it is not used for packet level flow control or reliability. Each endpoint of a connection maintains two windows based on transaction identifiers: an initiator window and a target window.

Initiating a new transaction

An initiator initiates new transactions. An initiator may start a new transaction if the initiator window is not full. This is true if **next_ixid** is less than **max_ixid**. If the window is not full then procedures for initiating a new request are:

1. **next_ixid** is assigned to the transaction. The corresponding entry in **ixid** is initialized with information about the transaction including the opcode, associated data to send, and caller information. The **ixid** array is indexed by **next_ixid** % **tal_window_size**. **next_ixid** is incremented.
2. The Transaction Layer tries to send packets for the transaction over the Packet Delivery Layer. Packets may be sent up to the point that the PDL send window becomes full. If the window is full and there are more packets to send for the transaction then sending is deferred for the transaction. When the PDL window is no longer full (received ACKs move the send window) then a callback will done to send more packets for the transaction
3. For each packet sent, the XID and Seqno are set accordingly. The XDI ACK is copied from **next_acked_txid**.
4. All packets for the transaction are eventually sent and the last one has **eom** set

Receive target processing

When a packet is received and passed from the Packet Delivery Layer, target receive processing is invoked if Opcode indicates delivery to the target. Procedures are:

1. The XID is checked
    a. If XID is greater than or equal to **max_txid** or XID is less than **next_acked_txid** then the XID is out of the window. This is considered an error. The packet is dropped and a message is sent to the connection manager who may take further action
    b. If the bit for the XID is set in **comp_tbitmap** then this is considered an error (bitmap index is XID minus **next_acked_txid**). The target has received an XID that it believes has been completed. The packet is dropped and a message is sent to the connection manager who may take further action
2. The **txid** state for the XID checked
    a. If the state is “closed” then this is the first packet received for a new transaction. Change the **txid** state to “open”. Initialize the **txid** state as necessary
    b. If the state is already “open” then check the **seqno_bitmap** in the **txid** state. If the bit is set then this sequence number has already been seen and that’s considered an error. The packet is dropped and a message is sent to the connection manager who may take further action
3. If the **eom** bit is set then:
    a. If **num_packets** in the **txid** state is non-zero then the **eom** flag has already been seen for the transaction. This is considered an error. The packet is dropped and a message is sent to the connection manager who may take further action
    b. Otherwise, **num_packets** in the **txid** state is set to Seqno
4. The sub-operation is processed

5. If all packets for the transaction have been received then transaction reception is complete (i.e. **weight**(**seqno_bitmap**) equals **num_packets**).
    a. The bit for the XID in the **comp_tbitmap** is set to one (bitmap index is XID - **next_acked_txid**)
    b. The target completes the request and may commence sending a reply to the initiator
    c. If the transaction does not require a response and XID == **next_acked_txid** then the transaction retirement procedures below are followed

Transaction replies

A target may send a response to an initiator request in one or more packets. Procedures are:

1. The Transaction Layer tries to send reply packets for the transaction over the PDL layer. Packets may be sent up to the point that the PDL send window becomes. If the window is full and there are more packets to send for the transaction then sending is deferred for the transaction. When the PDL window is no longer full (received ACKs move the send window) then a callback will done to send more packets for the transaction
2. For each packet sent, the XID and Seqno are set accordingly. The XID ACK is copied from **next_acked_txid**. Seqno is set from **seqno** in the **txid** state and in incremented for each packet
3. All packets for the transaction are eventually sent and the last one has **eom** set. Once all packets for the transaction have been sent the transaction can be retired per the procedures below

Target transaction retirement

When a target transaction is finished, that is the request has been completed and any replies have been sent, the transaction can be retired. The procedures are:

1. Set the corresponding **txid** to “closed” and any ancillary resources are freed
2. If the XID being retired equals **next_acked_tid**, the window slides following these procedures:
    a. The number of XIDs to retire is one plus the number of consecutive bits set in the **retired_tbitmap** starting from index 1
    b. **next_acked_txid** is incremented by the number of XIDs to retire
    c. **comp_tbitmap** is right shifted by the number of XIDs to acknowledge
3. Else when the XID being retired does not equal **next_acked_id**, the corresponding bit in the **retired_tbitmap** is set to one (bitmap index is XID % **tal_window_size**

Receive initiator processing

When a packet is received and passed from the Packet Delivery Layer, initiator receive processing is invoked if Opcode indicates delivery to the initiator. Procedures are:

1. The XID is checked

    a. If XID is greater than or equal to **next_ixid** or XID is less than **unacked_ixid** then the XID is out of the window. This is considered an error. The packet is dropped and a message is sent to the connection manager who may take further action
    b. If the bit for the XID is set in **retired_ibitmap** then this is considered an error (bitmap index is XID minus **unacked_ixid**). The initiator has received an XID that it believes has been completed. The packet is dropped and a message is sent to the connection manager who may take further action
2. The **ixid** state for the XID checked
    a. If there are packets still outstanding to send in the request then the target has sent a response before it was allowed to. This is considered an error. The packet is dropped and a message is sent to the connection manager who may take further action
    b. The **seqno_bitmap** in the **ixid** state is checked. If the bit is set then this sequence number has already been seen and that's considered an error. The packet is dropped and a message is sent to the connection manager who may take further action
3. If the **eom** bit is set then:
    a. If **num_packets** in the **ixid** state is non-zero then the **eom** flag has already been seen for the transaction. This is considered an error. The packet is dropped and a message is sent to the connection manager who may take further action
    b. Otherwise, **num_packets** in **ixid** state is set to Seqno
4. The corresponding bit in **seqno_bitmap** is set and the sub-operation is processed
5. If all reply packets for the transaction have been received then transaction reception is complete (i.e. **weight**(**seqno_bitmap**) equals **num_packets**). Once any additional processing is complete, like posting a response on a receive queue, then the transaction can be retired per the procedures below

Initiator transaction retirement

When an initiator transaction is finished, that is the request has been completed and any replies have been received, the transaction can be retired. The procedures are:

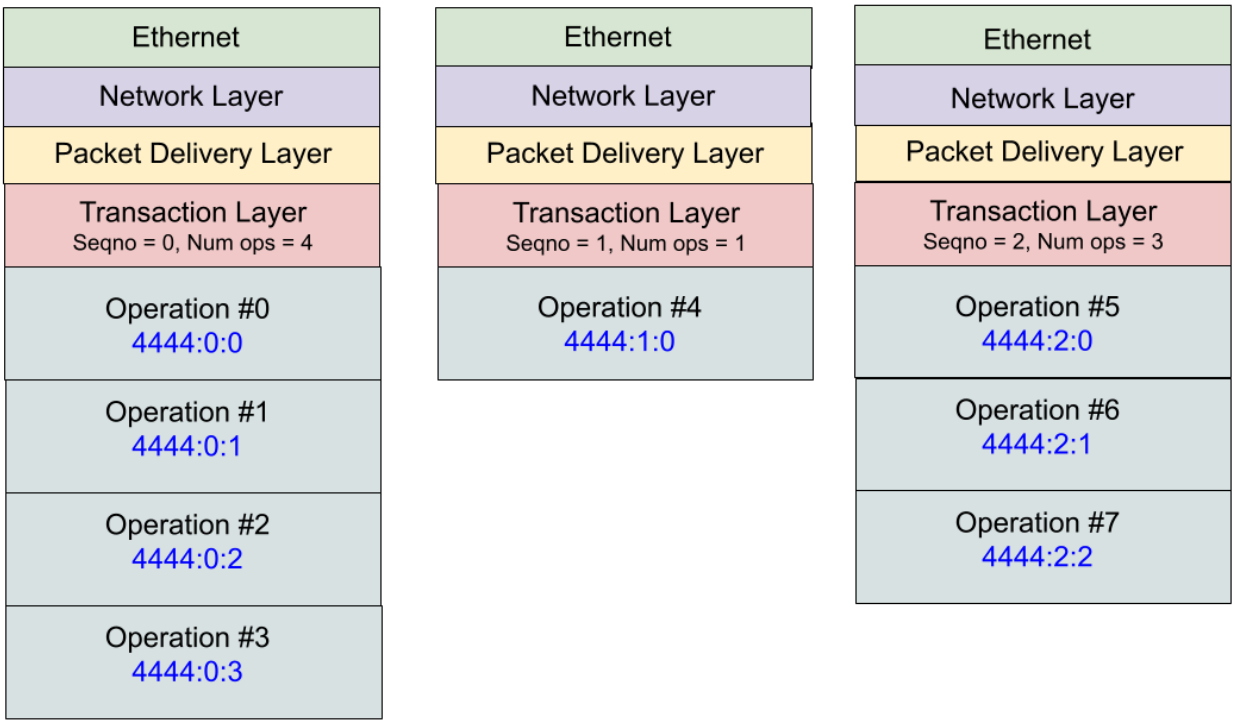
1. Set the corresponding **ixid** to "closed" and any ancillary resources are freed
2. If the XID being retired equals **unacked_ixid**, the window slides following these procedures:
    a. The number of XIDs to retire is one plus the number of consecutive bits set in the **retired_ibitmap** starting from index 1
    b. **unacked_ixid** is incremented by the number of XIDs to retire
    c. **retired_ibitmap** is right shifted by the number of XIDs to acknowledge
3. Else when the XID being retired does not equal **unacked_xid**, the corresponding bit in the **retired_ibitmap** is set to one (bitmap index is XID % **tal_window_size**

Operations identification

Operations are identified by a combination of the XID for the transaction, the sequence number for the packet the operation is contained in, and the number of the operation in the packet determined by the position of the operation header. The identifier can be denoted by:

<XID>:*<Seqno>*:*<OpInPacket>*

An example of operation identifiers is shown below.



Operation counting. This image shows three SUPERp for a transaction with XID equal to 4444. The entire transaction consists of eight operations spread across three packets (numbered 0 to 7). The identifiers for each operation are shown in blue with the format <XID>:<Seqno>:<OpInPacket>.

Connection management

Connections in SUPERp are assumed to be long-lived and static once created at initialization time. Connection management is done out-of-band.

Connection management

A connection manager manages the connections and nodes of a SUPERp network. Each host runs a SUPERp agent that manages the local SUPERp instance. The connection manager connects to each agent via a standard connection (e.g. RESTful APIs over TCP).

The connection manager is responsible for managing the SUPERp endpoints and topology. The connection manager maintains the list of global configuration parameters described in the parameters section. A database is maintained for all the connections in the network. The database contains an entry for each connection consisting of the normalized 4-tuple and connection specific parameters such as the traffic class.

System boot

The following procedures are performed to start SUPERp operations

1) The connection manager establishes a management connection with each of the SUPERp agents. Each agent provides its capabilities including memory space for RMA.
2) The connection manager sends the list of global parameters to each node. The parameters may include the programs to be run by a programmable device. The parameters also include per node RMA attributes such as ACLs. If a global address space is being used, the connection manager will provide the base address for the node.
3) The connection manager sends the per node list of connections to each node. This includes the 4-tuple with the addresses, local CID, and remote CID. Other connection specific parameters are set.
4) Once the connection agent has all the configuration and list of connections, it configures the SUPERp device. This entails loading the necessary programs and setting up the connection states to their initial values. Targets are enabled at this time.
5) When a node completes initialization it replies to the connection manager. Once all nodes have replied, the connection sends a “start operations” message to all the nodes
6) Upon receiving a “start message” each node can start applications and commence RMA initiator operations

Hard connection shutdown

A shutdown is necessary when there is a “hard error” that is unrecoverable. When a hard error is detected we take the conservative approach and assume that the whole system is corrupted such that everything needs to be restarted. The actions are:

1) An error is reported to the local connection agent. The connection agent immediately ceases any operations on the connection. Any packets received are dropped, timers are cancelled, an error is reported to the application for any pending initiator requests, and any completions from the DMA engine are ignored
2) The local agent sends a message to the connection manager. The connection manager determines if the error is isolated to just the one connection or if it has compromised the whole system. In the latter case, procedures to shut down everything and restart the world. In the former case, the connection manager sends a “shutdown connection” to the connection peer to perform an immediate shutdown
3) Once the peer completes the shutdown, it sends a message back to the connection manager and the connection manager can then proceed to restart the connection

Soft connection shutdown

The connection manager may decide to restart a connection that is not in error. For a “soft shutdown” the system attempts to complete any outstanding transactions. Procedures are:

1) The connection manager sends a “soft shutdown” message to the connection agents for each endpoint of a connection.
2) Upon receiving the message the connection agent transitions the connection to “shutting down” state. No new initiator requests are accepted (new requests can either result in an error or they can remain in the work queue if the connection is being restarted).
3) A “LAST NULL” initiator request is sent by both sides. Any prior requests will complete and then the LAST NULL request will be acknowledged (XID ACKed), at this point there are no initiator requests pending. Once both sides have seen the final acknowledgement then the connection is quiescent.
4) If the connection is being restarted then the procedures in system boot are applied. If the restart is transparent to the application then operations can continue and any pending requests on work queues may be processed.

System errors

A system level error is a serious unrecoverable error in the network. For example, one of the nodes may have completely lost connectivity. We assume that this is a rare event such that when such an error occurs everything including the applications are restarted.

Procedures are:

1) The connection manager send a “shutdown node” message to each node (at least those that are operation)
2) Upon receiving the message the agent ceases all SUPERp operations, frees resources and kills all applications that are SUPERp initiators (the application may have some checkpoint features to expedite a restart). The SUPERp agent sends a “node shutdown complete” message to the connection manager.
3) Once the connection manager receives a “node shutdown complete” message from all nodes it can commence restart procedures presumably with a different configuration.

Operations

Generic Operation opcodes

Three generic opcodes is defined:

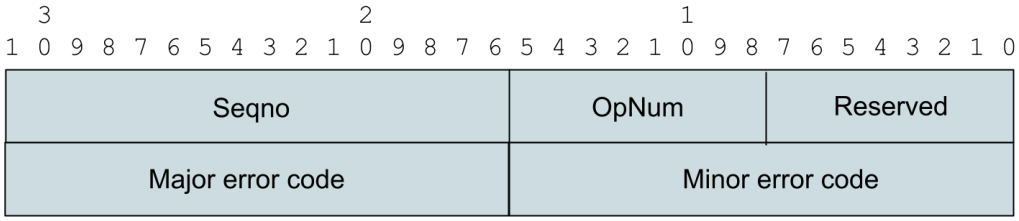
- **0: No-op** ((initiator->target). The XID is set for an initiator transaction.
- **1: Last NULL** (initiator->target). The XID is set for an initiator transaction.
- **2: Transaction Error** (target->initiator). The XID is set to the XID in the initiator’s original request

No-op and Last message format

For No-op and Last Message num_ops is set to zero and there is no format for these operations.

Transaction Error

The Opcode Specific field contains a sixteen bit major and minor error code. Seqno and OpNum identify the particular operation that caused the error.



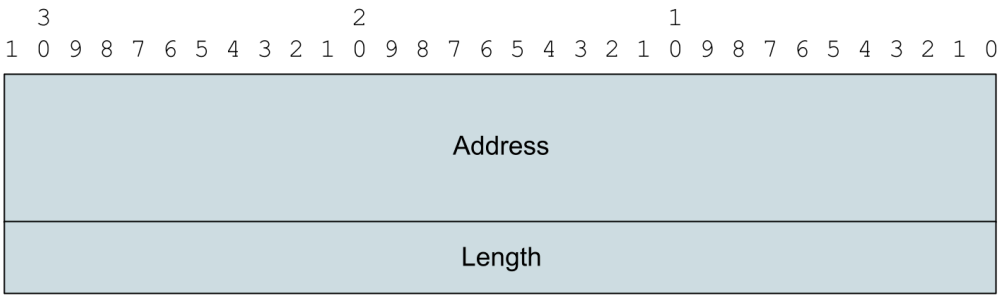
RMA operations

Basic RMA operations are . RMA opcodes are:

- **8: RMA Read** (initiator->target): RMA Read request. XID is set as the initiator request
- **9: RMA Write** (initiator->target): RMA write request. XID is set as the initiator request
- **10: Read response** (target->initiator): Response data for RMA read request. XID is set to that of the original initiator request
- **11: Send** (initiator->target): Send unrequested data to the target. A key provides a receiving context for the target
- **12: Send to queue pair** (initiator->target): Send data on a queue pair to target. There is no offset so the operation is expected to be constrained to a single transaction.

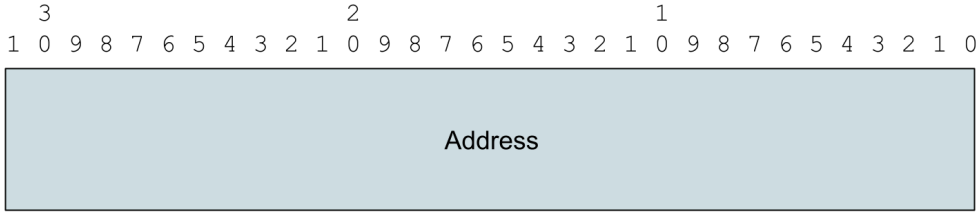
Read format

The Opcode Specific field contains a sixty-four bit address to read from and a thirty-two bit length. There is no data attached for the read operation.



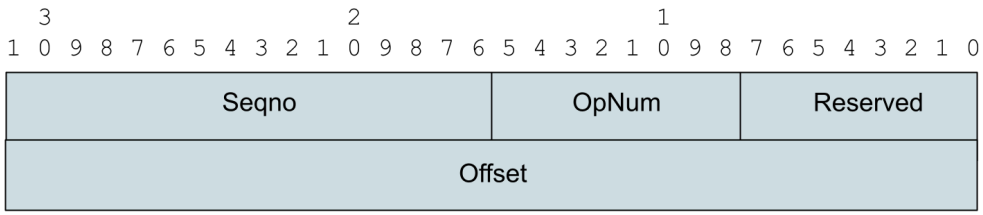
Write format

The Opcode Specific field contains a sixty-four bit address to write. The data to write is contained in the data block for the operation.



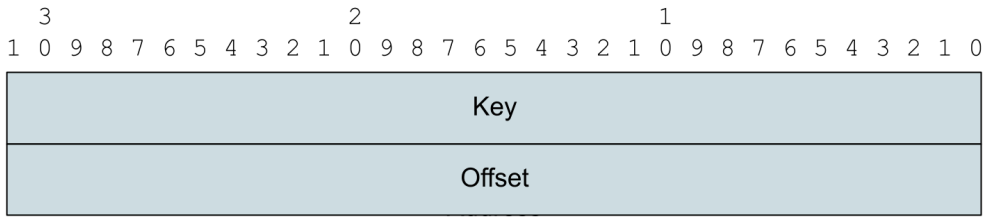
Read response format

The Opcode Specific field contains the offset of read data being returned for a read operation. **Seqno** and **OpNum** identify the original read operation (with the XID) if there were multiple read operations in the original initiator transaction. The read data is contained in the data block or the operation.



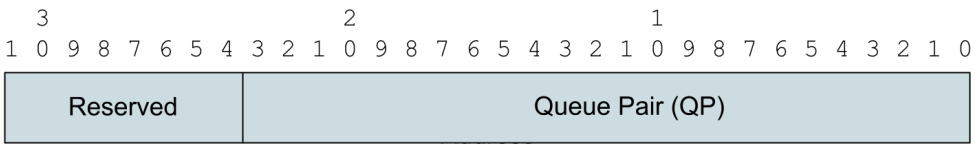
Send format

The Opcode Specific field contains a thirty-two bit key to read from and a thirty-two bit offset. The send data is contained in the data block or the operation.



Send to QP

The Opcode Specific field contains a twenty four-bit Queue Pair number. The send data is contained in the data block or the operation.



Application API

A *libverbs* API will be presented to applications for RDMA operations. A library translates libverbs requests to work queue requests and completions will be sent back following the appropriate callbacks. The API will expose queue pairs to userspace for highest performance.

Queue Pairs

The command channel for SUPERp has canonical RDMA Queue Pairs (QPs) that are exposed to Upper Layer Protocols or applications. Each QP maps to one SUPERp connection, and multiple QPs in different applications may be mapped to the same SUPERp connection. Demultiplexing of transactions back to their QP is done by the transaction identifier (XID). Different connections might also be used if isolation is needed. Applications enqueue request messages on a Send Queue (SQ), and completions are posted on the Receive Queue (RQ) by SUPERp. Transactions are mapped to the associated QP by the transaction identifier.

Initiator processing

A new transaction commences when the Transaction Layer processing dequeues a request from an Send Queue. The Transaction Layer will only dequeue a request if there is an available slot in the pending transactions array in the SUPERp connection context. When a transaction request is seen, the transaction is first validated and permissions are checked. If the validations fail then an error is returned on the RQ.

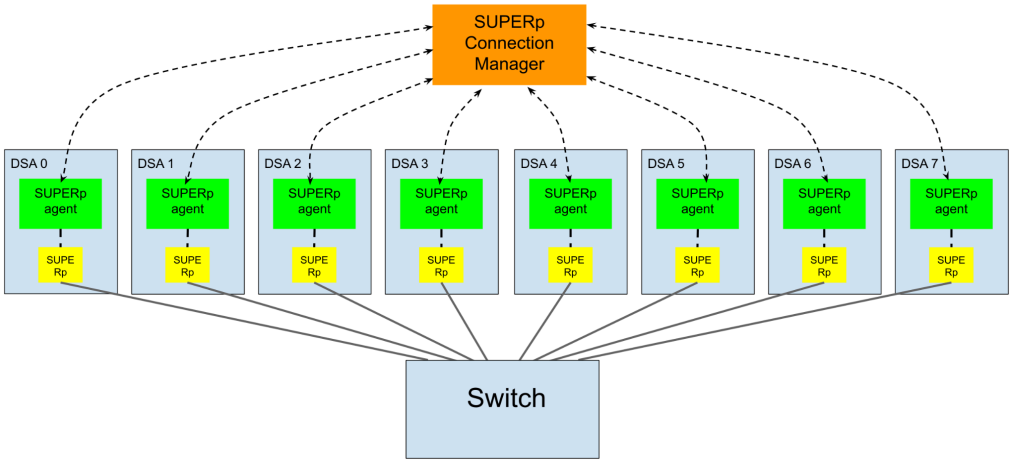
If the transaction request is valid the next available transaction state in the connection context for the connection associated with the QP is reserved. The transaction state is initialized. If the request includes data then a PVbuf is allocated and populated with scatter-gather data.

If there is availability in the connection's send window, then packets for the transaction may be immediately sent via the PDL layer up to the point that the send window is full. When the send window is full, then sending packets for the transaction is deferred. When an ACK is received then new packets may be sent. A packet scheduler selects which of the deferred transactions may now send packets.

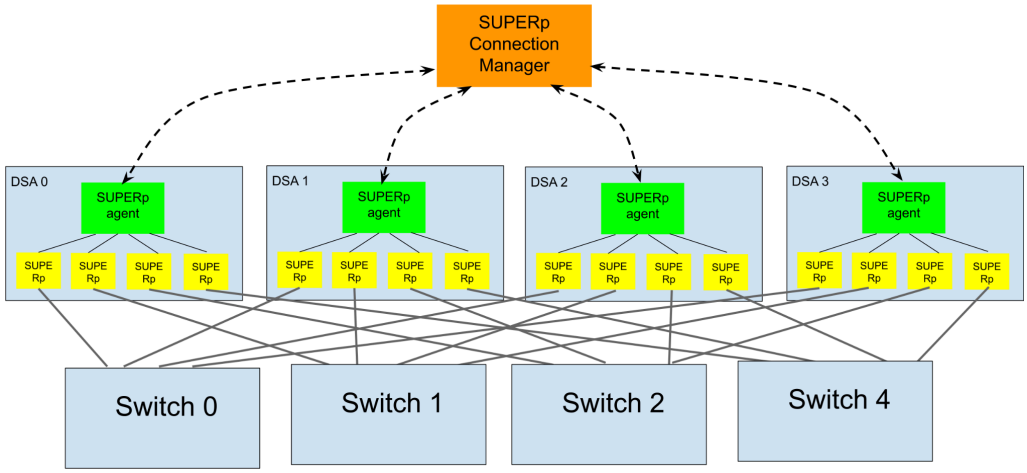
Deployment scenarios

Each SUPERp provides network connectivity that can be configured as one, two, or four ports. For example, an 800G SUPERp instance supports 1x 800G, 2x 400Gbps, or 4x 200Gbps. The

primary network configuration is expected to be a single switch hop. The port configuration is chosen based on the required switch radix and SerDes count, as well as redundancy and failover considerations. A topology for single hop/single DSA connection is shown below.

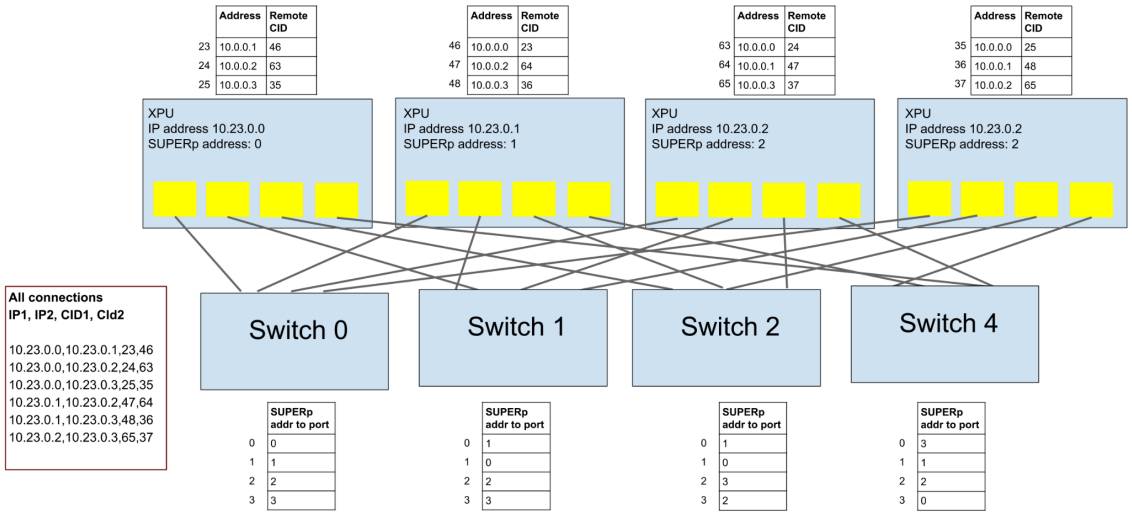


SUPERp is designed to enable multiple instances of SUPERp per DSA. For example, a single DSA can include 4, 8, or 16 SUPERp instances. The figure below shows an example configuration of 4 DSAs with four 800G SUPERp instances per DSA. Four Ethernet switches are used with 4 ports each. In this example, with fully switched connectivity, any pair of XPUs has up to 3.2 Tb/s (4x 800G) bandwidth between them.



SUPERp subnet

The diagram below shows a SUPERp subnet with four XPUs and four switches.



In the example, each of the four XPUs have a connection to each of the other XPUs. This results in twelve connections within the subnet that are shown in the connections list at the left. Each XDPU has a table indexed by its local CID for each of the connections to its peers. The tables are shown above each of the XPUs.

All the XPUs are in the subnet 10.23/16 which serves as the SUPERp subnet. XPUs can send SUPERp messages using UDP encapsulation or SUNH encapsulation. For instance, the third XPU can send a message in UDP/IP to the second one with the IP destination address set to 10.23.0.1, the source address set to 10.23.0.2, the PDL DCID set to 47, and UDP destination port set to the number of SUPERp. The same message can be sent in SUNH by setting the Destination address in SUNH header to 1 and the PDL DCID is still set to 47.

Configuration parameters

There are a number of configuration parameters for SUPERp. It is expected that all nodes in a SUPERp domain share the same configuration.

Parameter	Range	Default	Description
pdl_retrans_time_out	100 to 10^{30}	500	Retransmission timeout in nanoseconds
pdl_rtx_time_win	0 to 10^{30}	0	Proactively retransmit at timeout if packet send time is within window
pdl_rtx_time_win	0 to 64	8	Number of packets to proactively retransmit
pdl_max_rtx	1 to 8	4	Maximum number of retransmissions

			for a packet
pdl_delayed_ack_tim	100 to 10^{30}	500	Delayed ACK timeout
pdl_window_size	16 to 1024	32	Number of PSNs in send and receive windows
pdl_conns_per_xpu	1 to 4096	64	Maximum number of connections per XPU
pdl_receive_win	1 to 1024	32	Default receive window for a connection
nh_udp_port	0 to 65535	7777	UDP port for SUPERp
nh_num_caddr_bits	8 to 16	12	Number of a bits in SUNH addresses
nh_hop_limit	1 to 15	15	Default Hop Limit for sending
tal_trans_window	1 to 1024	32	Size of initiator and target transaction windows
tal_max_pkt_per_trans	1 to 4096	32	Maximum packet per transaction
tal_max_ops_per_pkt	1-15	8	Maximum operations per packet
tal_transaction_timeout	0 to 2^{64}	0	Transaction timeout in units of 10nsecs. 0 means no timeout
cm_address	IPaddr	None	Address of the connection manager
cm_port	0 to 65536	7778	Port number of connection manager