Falcon

Jonathan Trinh, Aaron Honjaya

Motivation

- Existing hardware transports rely on restrictive network assumptions
  - RoCE is the current standard for performing RDMA, but has strong requirements for losslessness which is managed with Priority Flow Control
    - PFC can pause traffic which propagates congestions
    - Operating this at scale is operationally difficulty
- Software transports can introduce too much overhead
  - To guarantee reliability and to handle dropped messages and reorder we manage this in software which adds CPU overhead
- Existing transports struggle with multipath and diverse workloads
  - RoCE especially struggles with identifying packet loss vs reordering
  - Reordering increases greatly when we introduce multipathing

Goals

Requirement	Design Principle
R1,**R2**: Predictable Performance under Diverse Networks	**D1**: Delay-based CC and Multipath load balancing (§6.1)
R3: Robustness and no performance cliffs	**D4**: Hardware resource admission, Backpressure, Error-handling (§6.2)
R4: Standard ULPs, multiple ULP support	**D2**: Layered Design **D3**: Simple Interface to ULPs (§6.3)
R5: Rapid Iterations and Operational Ease	**D5**: HW-assisted Programmable Engine (§6.2)

Falcon Architecture Overview

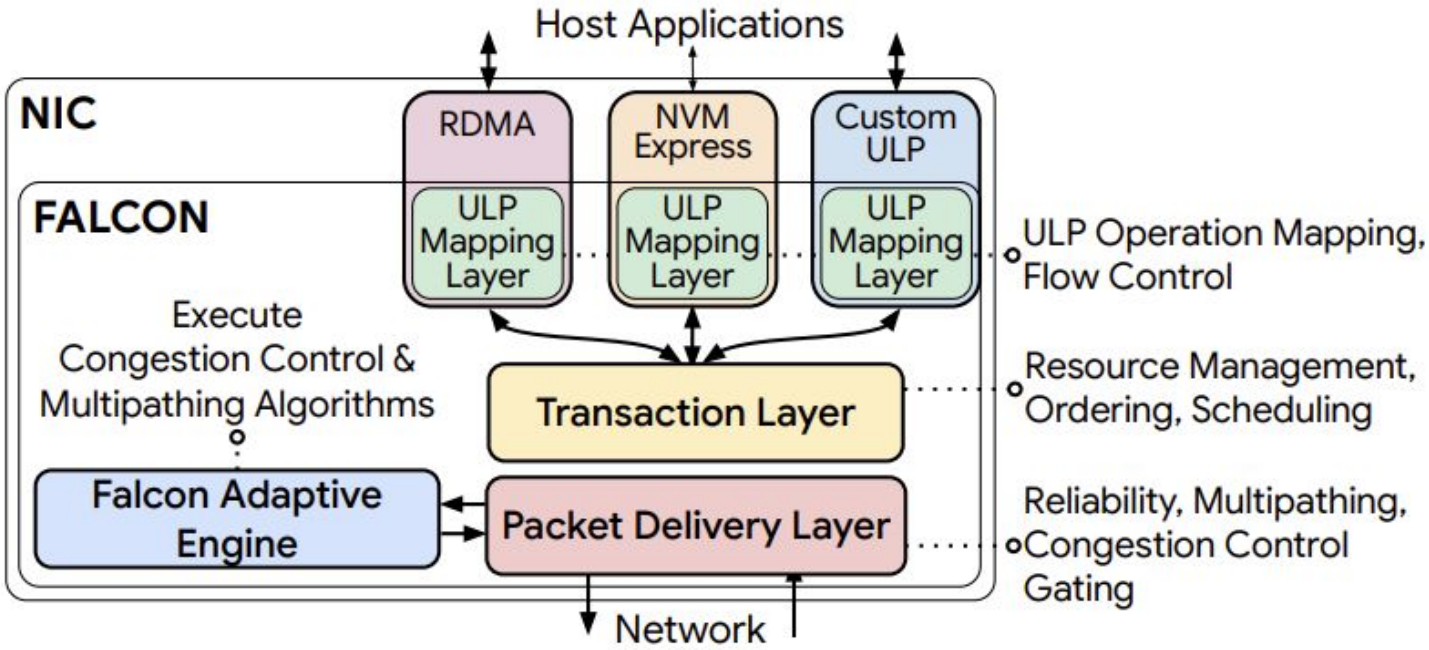
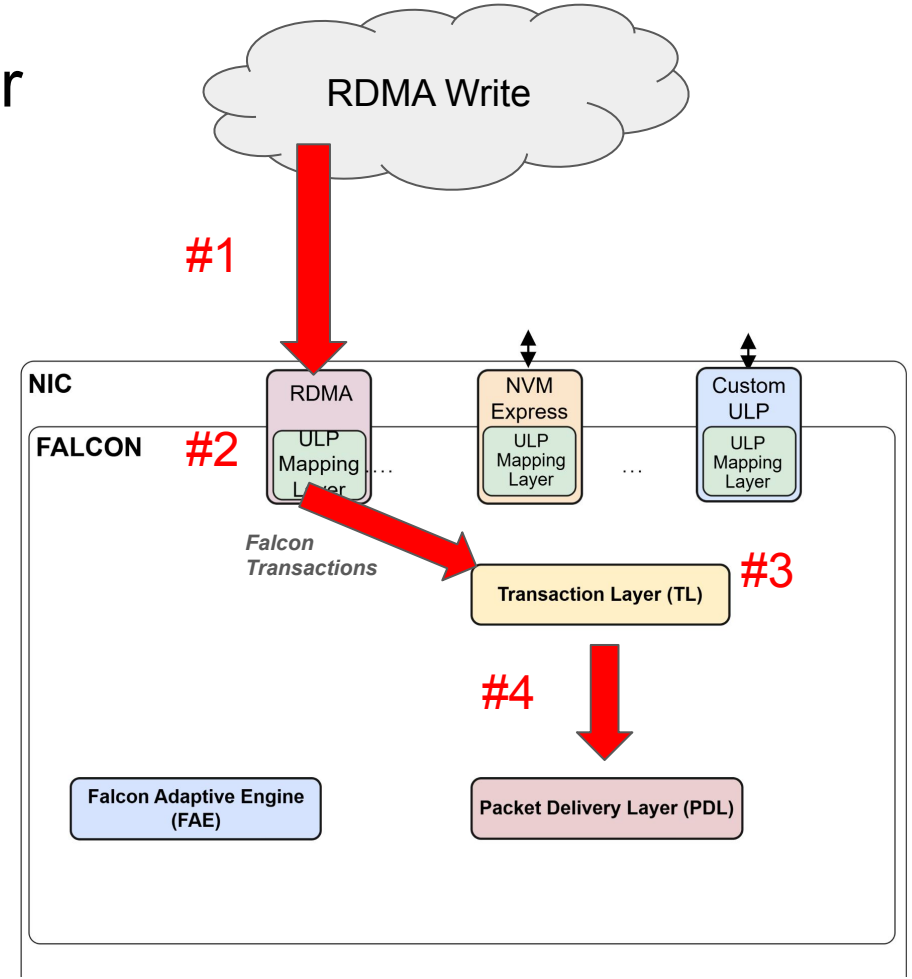


Figure 2: Falcon hardware transport layers

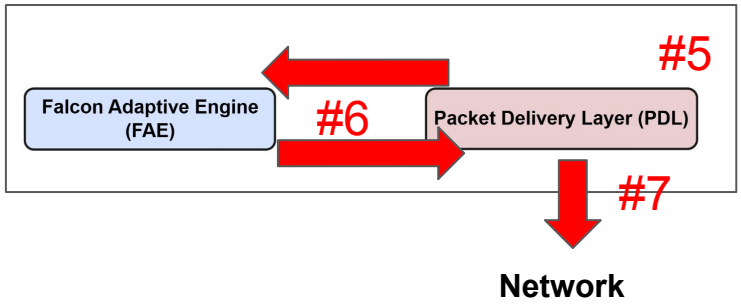
Falcon Request Flow - Sender

1. Host application Issues an operation
   a. E.g. an app makes an RDMA Write.
2. ULP translates into Falcon Transaction(s)
   a. Push or Pull
3. TL handles the Transaction
   a. RSN, Resources, Scheduling
4. Message Passed to the PDL



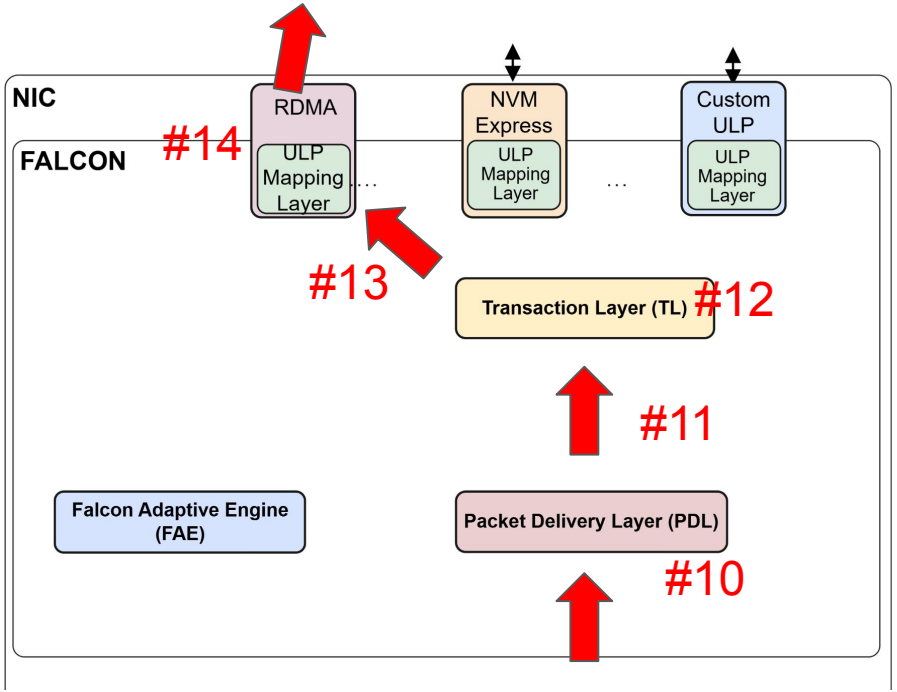
Falcon Request Flow - Sender

5. PDL Assign PSN(s)
6. FAE forwards control decisions to the PDL
    a. PDL has been sending relevant signals from previous acks to FAE
    b. FAE makes control decisions
    c. Uses information provided by PDL to for congestion control and multipathing
7. PDL enforces decisions and transmits the packet



Falcon Request Flow - Receiver

8. PDL receives packet
    a. Checks PSNs, generates ACK Info, etc.
9. PDL forwards to TL
10. TL process the transactions
    a. Manages resources, processes RSNs
11. TL forwards to ULP
12. ULP handles the operation
    a. Interprets the falcon operations and executes it, and waits for a response.

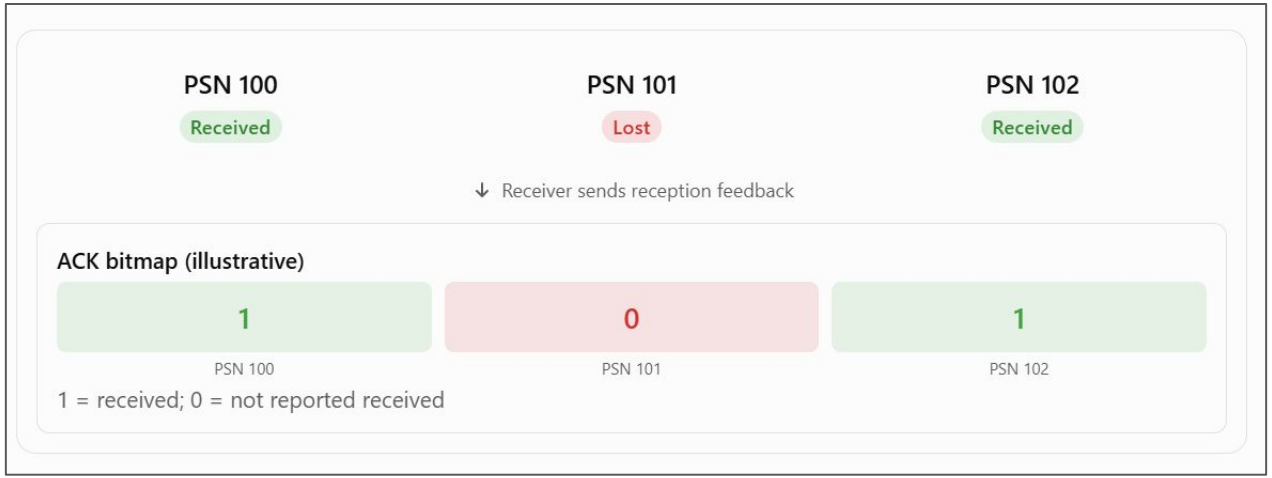


Reliability - Overview

- Falcon guarantees end-to-end reliability (like TCP)
- Packet Sequence Number (PSN)
  - Packets tracked by a sequence number attached to the header
  - For each logical connection between two endpoints, 1 PSN space for requests and 1 PSN space for responses.
- Messages are processed in order.
- Uses RACK-TLP to help recover from packet loss
- **General Idea** → take the end-to-end reliability of TCP and move it to hardware for a speed boost.

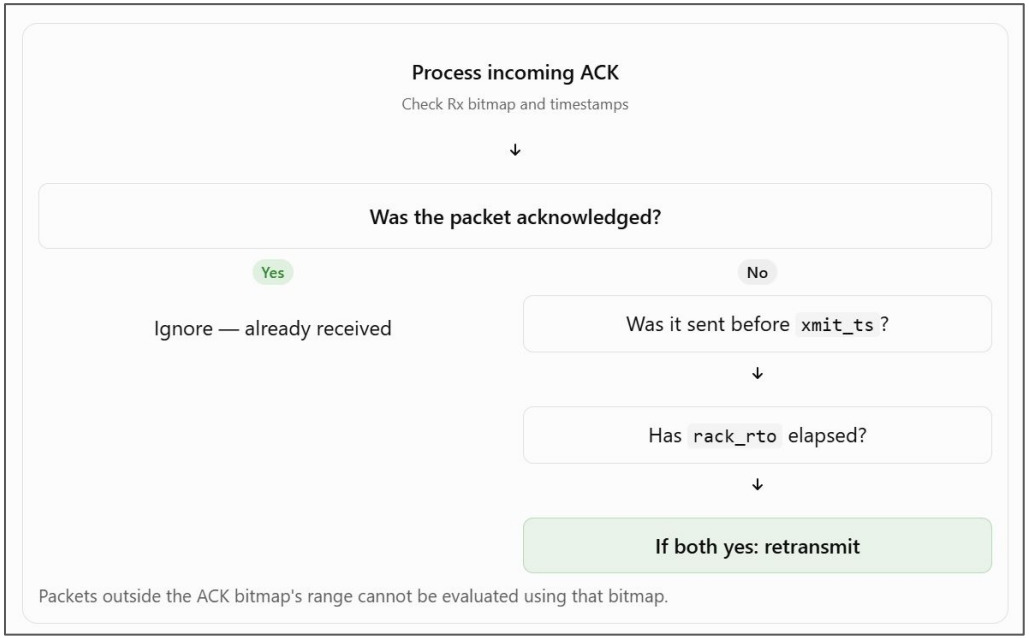
Reliability – Receiver

- Receiver's PDL tracks PSNs using a bitmap
  - The Bitmap is 128 bits
- Bitmap is attached to the Falcon Acks for the Sender
- Receiver also attaches timestamps to the ACK



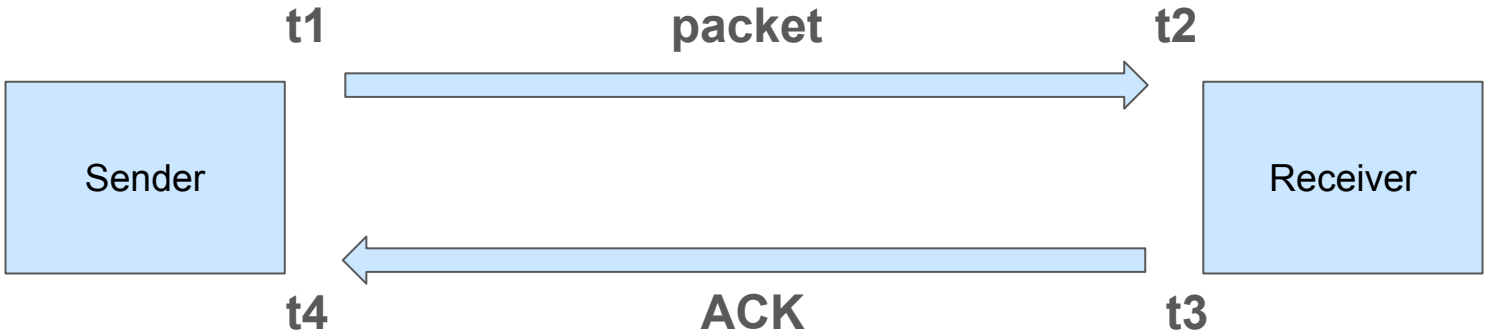
Reliability - Sender RACK-TLP

- When receiving an ACK applies RACK heuristic using
  - PSN Bitmap
  - ACK's transmission timestamp (xmit_ts)
  - Rack Timeout (rack_rto)
- TLP: When Sender detects no ACK progress, it retransmits the lowest un-ACKed PSN.
- FAE calculates timeouts (rack_rto & TLP)



Congestion Control – Calculating Delay

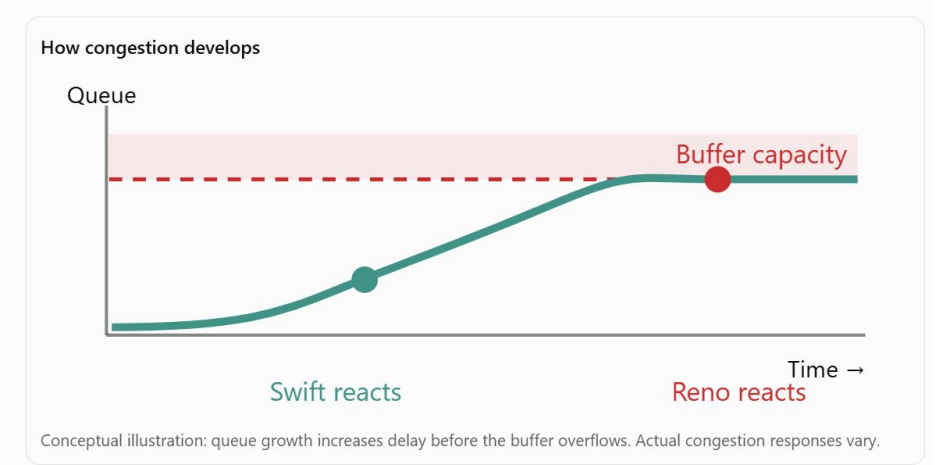
- Falcon measures NIC Rx buffer occupancy to calculate NIC congestion window (ncwnd)
- Falcon measures fabric delay to calculate the fabric congestion window (fcwnd)
- Effective congestion window is the minimum of ncwnd and fcwnd



Fabric delay = (t4 – t1) – (t3 – t2)

Congestion Control – Swift

- Swift – Google's delay-based congestion control algorithm
- Uses Additive Increase, Multiplicative Decrease
  - Delay below target → gradually increase congestion window
  - Delay above target → gradually decrease congestion window
- While TCP Reno reacts to **packet loss** but Swift reacts to an **increase in delay**
- Falcon modifies Swift to use both fcwnd and ncwnd.
- FAE computes the updated congestion windows and the PDL enforces them



Multipathing – Congestion-Aware

- Each connection can use multiple flows
- Each flow tracks its fcwnd
- Connection Scheduler – selects a connection for transmission and enforces fcwnd and ncwnd
- Flow Scheduler – selects flow with the largest open window
  - Open Window = fcwnd – Unacked Packets

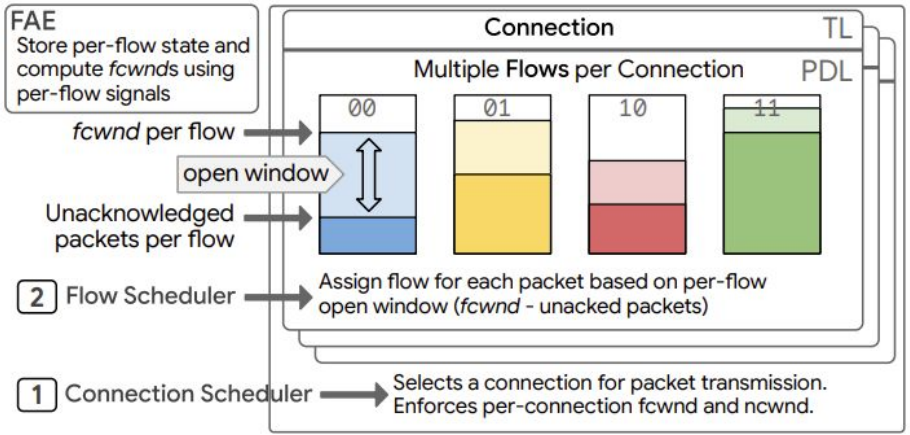
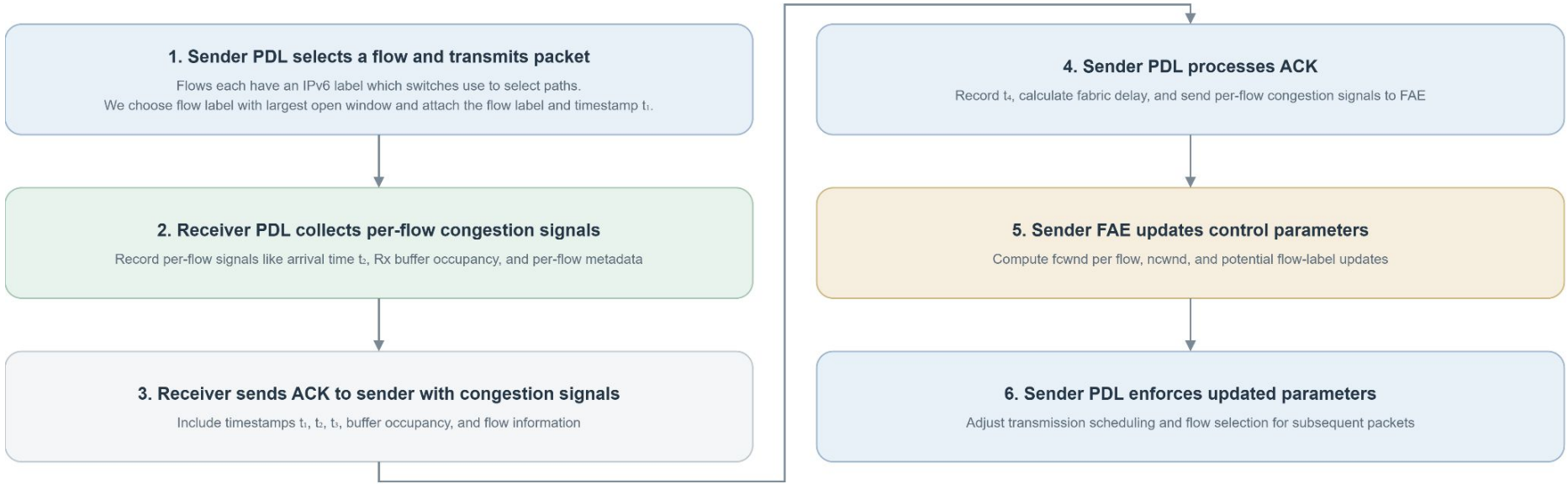


Figure 4: Congestion-aware Multipathing in Falcon.

Multipathing – Packet Transmission and Feedback



Interface to Upper Layer Protocols

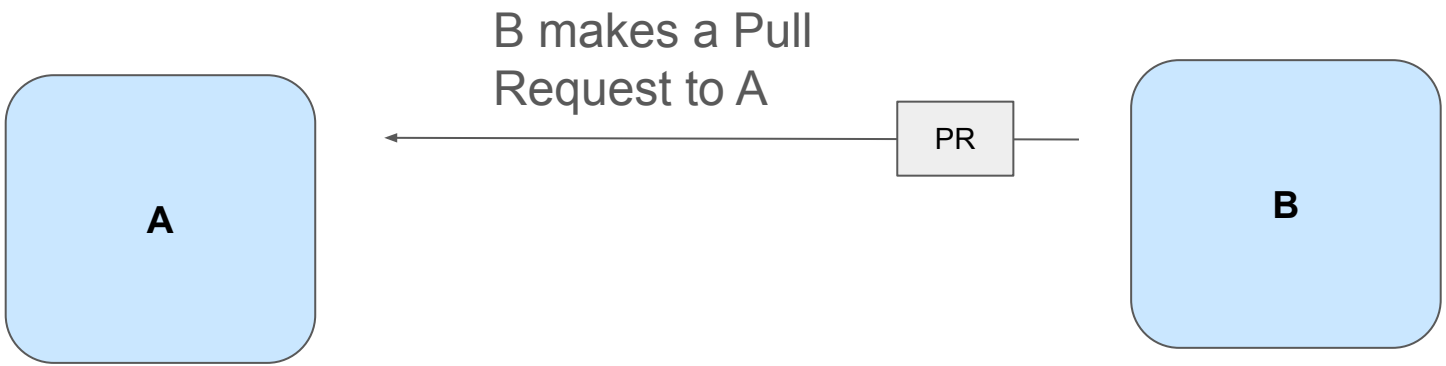
- Falcon defines a request/response transaction interface
- Each Push or Pull transaction has a request and its corresponding response
  - Push – sending data
  - Pull – receiving data
- Transaction Layer (TL) is type-aware (e.g. push/pull & request/response)

ULP	Operation	Falcon Transaction
RDMA	WRITE, SEND/RECV	Push
RDMA	READ, ATOMICS	Pull
NVMe	Read	Pull
NVMe	Write	Push and Pull

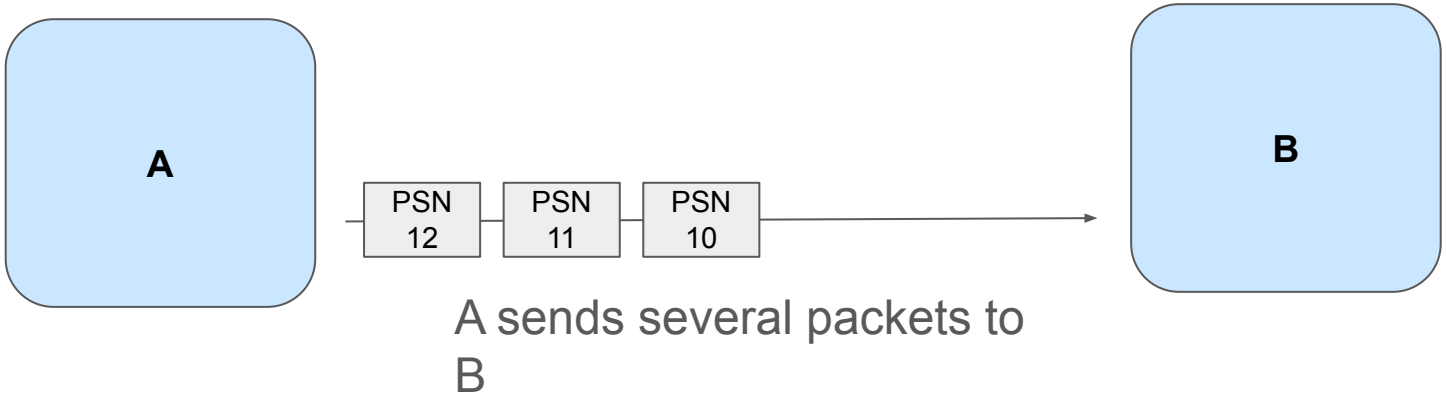
Transaction Layer – Why Type-Aware?

- Enforce congestion control based on packet types improving performance
- Put responses and requests into separate scheduling queues preserving order
- Allows the TL to assign requests and responses to **separate PSN spaces to avoid deadlock.**

Deadlock Conditions - Same PSN



Deadlock Conditions - Same PSN

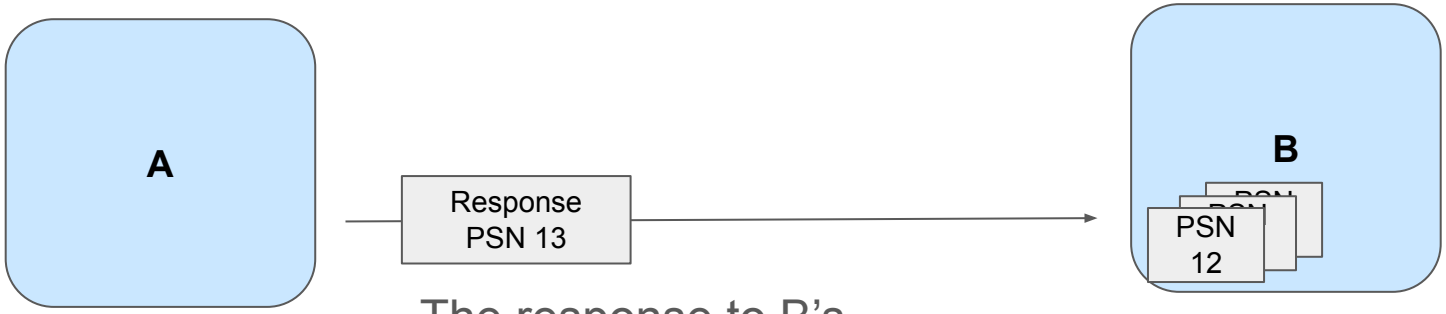


Deadlock Conditions - Same PSN



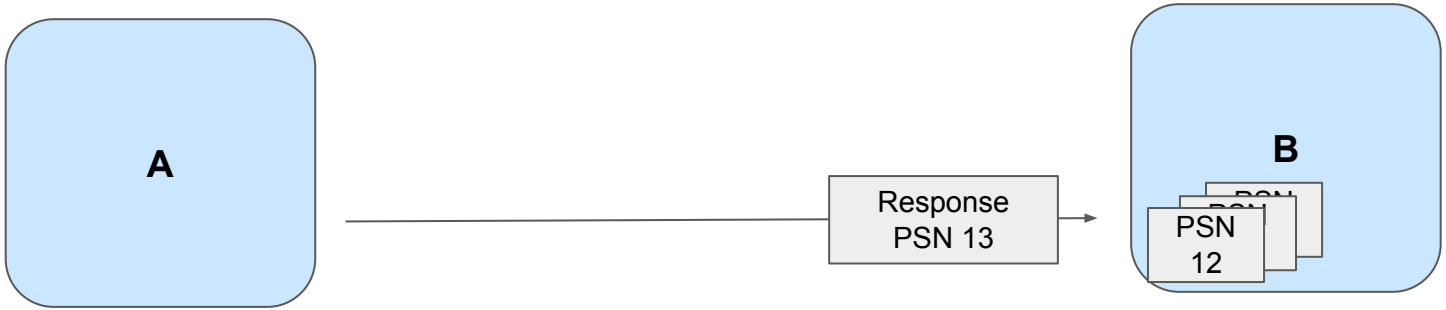
Pull Requests arrive at B, are stored, but B lacks resources to resolve them.

Deadlock Conditions - Same PSN



The response to B's request is sent with PSN 13

Deadlock Conditions - Same PSN



Processing PSN 13 would relieve the resources to resolve PSNs 10-12.
But to process PSN 13, PSNs 10-12 must be processed first.

Request Ordering

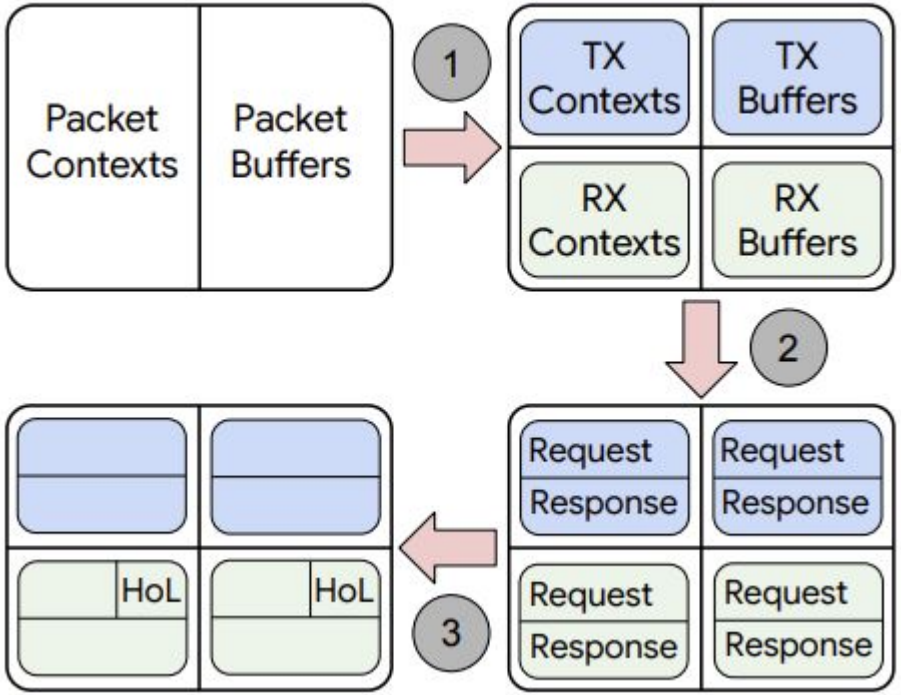
- Falcon connections can be ordered or unordered.
- Each transaction numbered by a request sequence number (RSN)
- RSN is transmitted in the packet headers.
- Falcon can accept and buffer OOO packets while waiting for earlier ones

Resource Carving

Hardware Resources:

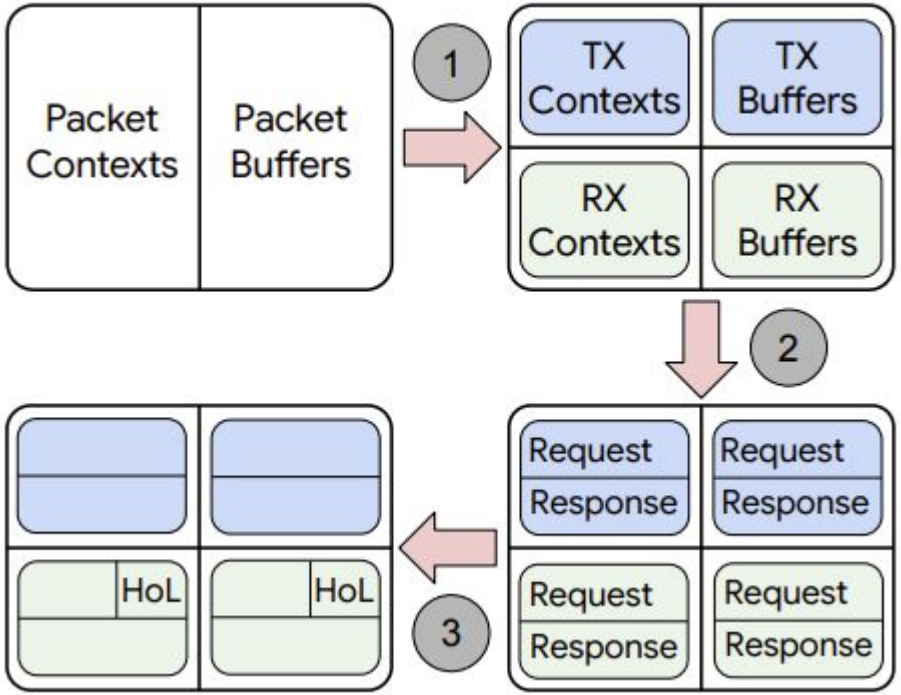
- Contexts: holds fixed size metadata
- Buffers: holds variable size metadata

1. Split hardware resources into data to be transmitted (Tx) and received (Rx)
2. Split Tx and Rx into request and response.
3. Rx Request only allows Head of Line Requests (HoL) past some threshold
    a. A request is HoL if its RSN is one more than the last received RSN.



Resource Lifecycle

- Requests from ULP result in
  - Tx resources reserved for request
  - Rx resources reserved for response
- Receivers use Rx/Tx resources when requests/responses arrive from the network.
- Rx resources released when packets delivered to ULP
- Tx resources released upon receiving corresponding ACK
- Back-pressures ULPs if there are not enough resources



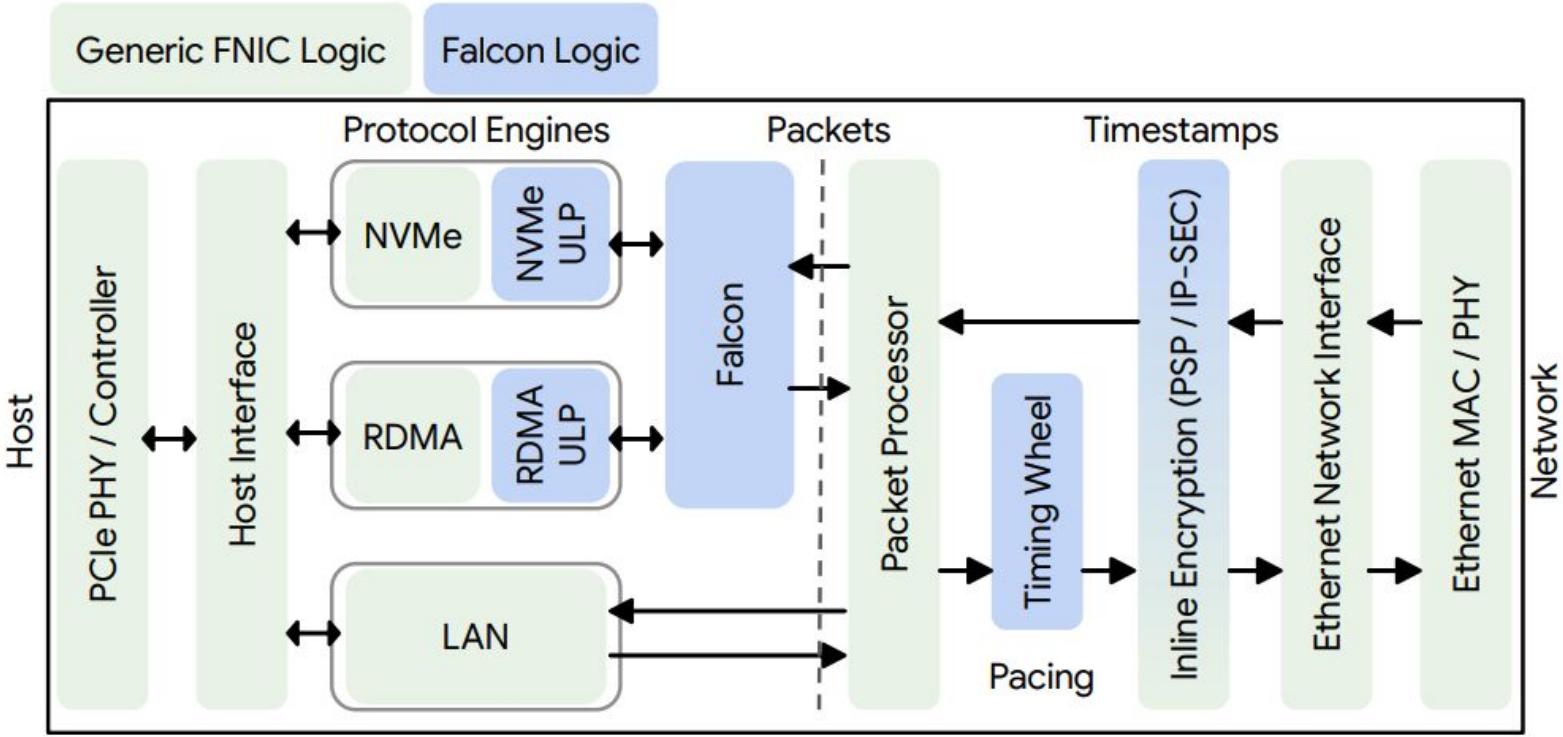
Isolation

- Falcon ensures isolation between connections – one connection's slow progress will not affect another.
- TL allows backpressure to ULPs with a per-connection Xon/Xoff flow control interface.
- Usage threshold calculated using Dynamic Thresholds (DT) Algorithm:

$$T_c = a_c * FreeResources$$

- Unlike traditional DT, a_c is not static but calculated by the FAE based on a congestion-aware variable proportional to the fcwnd and ncwnd.

Falcon's implementation in NIC



Notable Implementation Choices

- On-NIC Resources:
  - Additional on-NIC resources for buffering OOO packet arrivals
  - Packet buffering requires less than 1% of IPU die area
- Connection State Caching:
  - Second-level cache across multiple IPU blocks to improve bandwidth and latency
  - Provide sufficient memory bandwidth to retrieve connection states
- Error Handling:
  - Traditional transports handle exceptions in firmware, potentially stalling the fast path
  - Falcon handles exceptions in hardware using per-packet contexts and retransmission timeout tracking

Implementing and Scaling FAE

Hardware–Software Communication

- PDL sends events to FAE through shared-memory queues
- FAE computes new parameters and sends responses to PDL

Managing FAE State

- Stateless FAE: PDL manages state and includes it in events
  - Limited by how much state PDL can manage
- Stateful FAE: FAE maintains additional state in software
  - Cache misses can slow event processing
  - Prefetch upcoming event state to hide memory latency

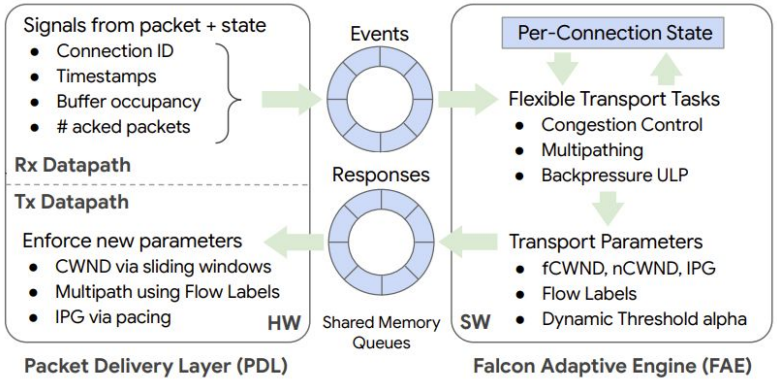
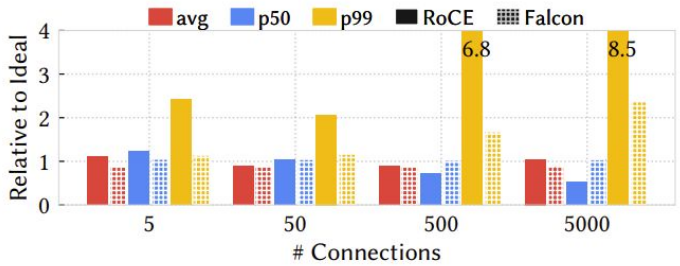


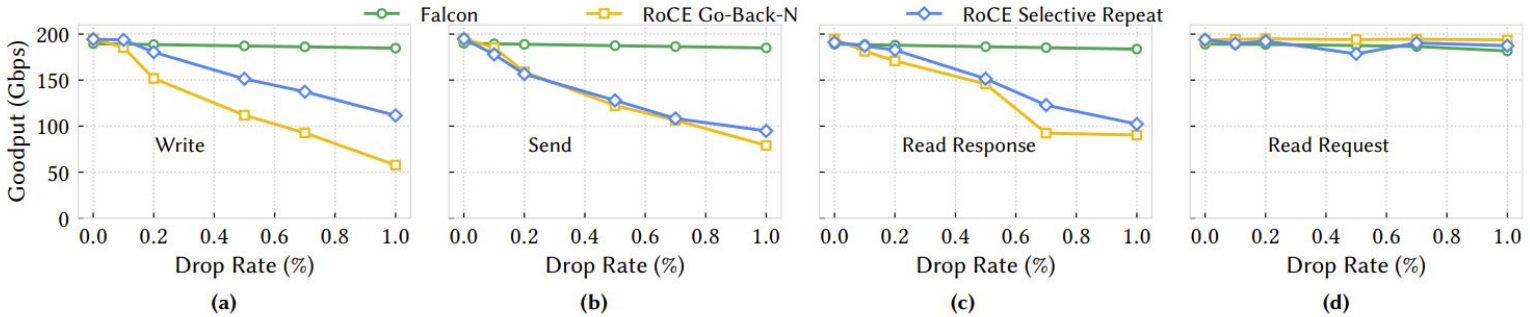
Figure 9: Falcon Adaptive Engine Implementation

Network Performance: Reliability & Congestion Control

- Falcon achieves up to 65% higher goodput under lossy conditions compared to RoCE
- Even under 5000:1 incast, Falcon P99 latency only approx 2.5x ideal, while RoCE is 8.5x ideal.



(a) Op latency relative to ideal with varying incast size



Multipath Performance

- Multipath connections offer up to 180x lower latency than single path connections
- Multipathing sustains offered load up to 90% of fabric capacity

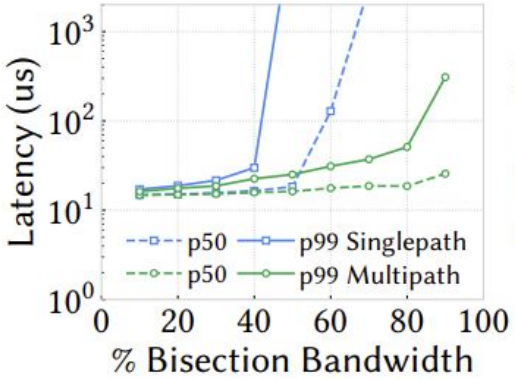


Figure 15: Multipath Latency

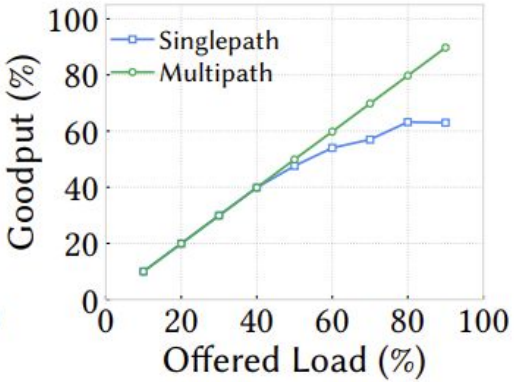


Figure 16: Multipath Goodput

Real-World Evaluation

- Evaluated RDMA-Falcon and TCP on
  - **Gromacs** – molecular dynamics simulation tool
  - **WRF** - weather forecasting model

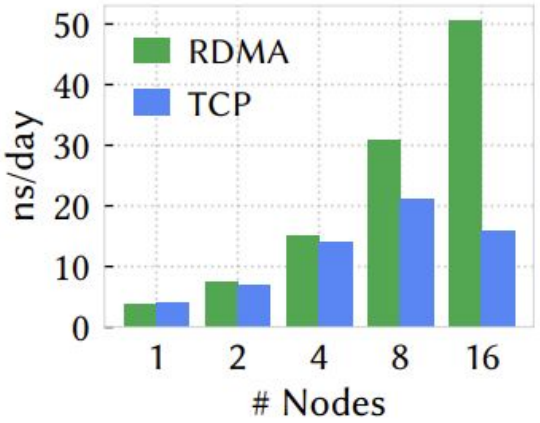


Figure 27: Gromacs

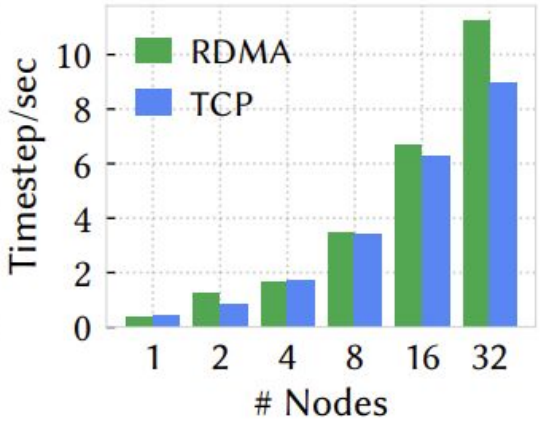
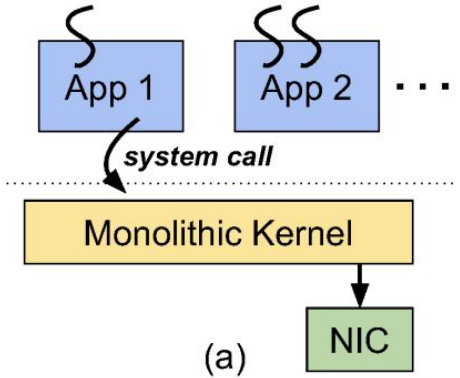


Figure 28: WRF

Snap

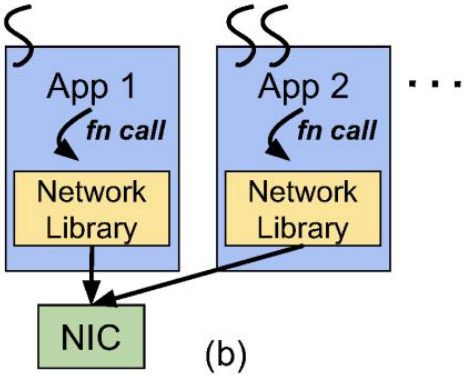
Motivation

- Kernel development is slow
- Not all functionality was covered in kernel updates



Monolithic Kernel

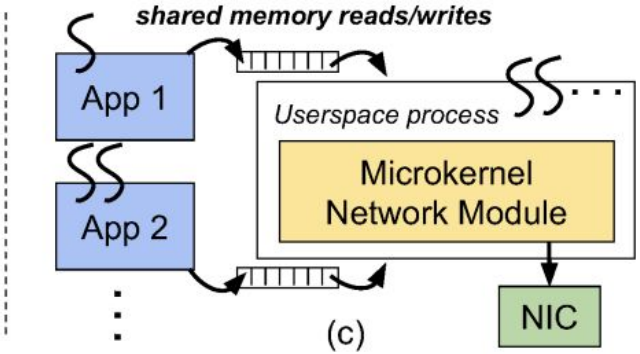
- Faster to develop and has lower latency
- Apps don’t coordinate and cause poor cpu usage



Library OS

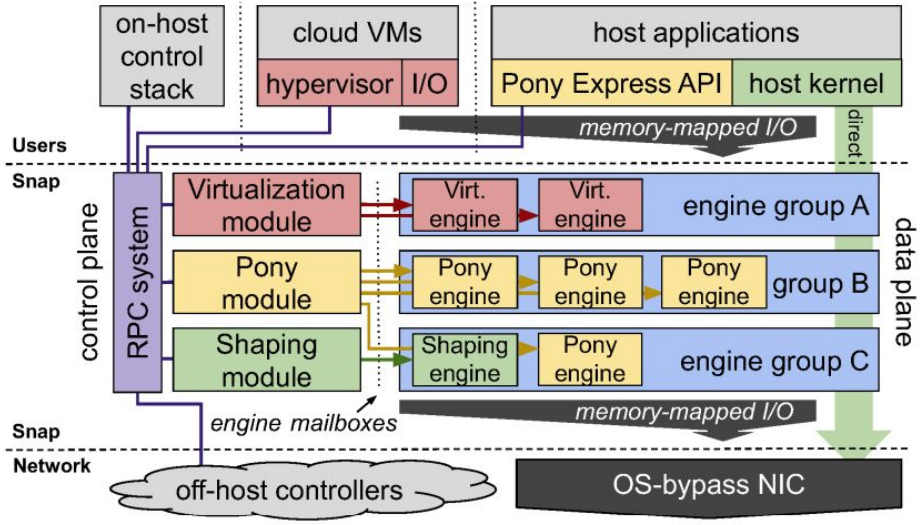
Google’s Solution: Snap

- **Microkernel approach**: move network services from OS kernel to a separate user space process
- Combines centralization benefits of (a) and userspace benefits of (b)
- Updates are separated from kernel/application releases



Snap Architecture Overview

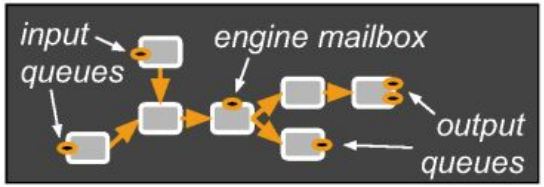
- Control Plane (Management)
  - Creates and manages Snap Engines
  - Communicates with apps using RPC or Shared Memory
  - Communicates with engines in Data Plane using mailboxes
- Data Plane (Packet processing)
  - Does the actual packet processing
  - Communicates with apps and NICs through MMIO



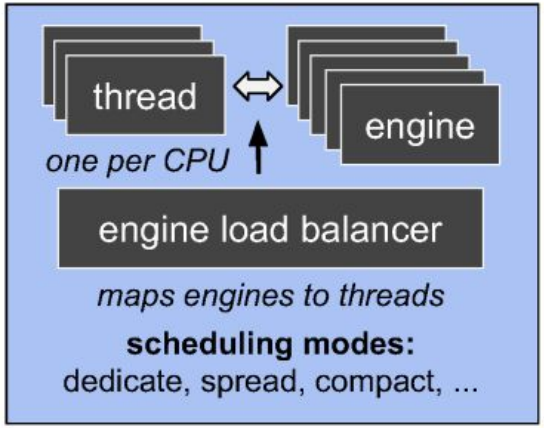
(a) Snap architecture

Engines and Modules

- Engines are stateful, single-threaded and avoid blocking synchronization.
- Engine developers determine how many engines to make and how work is distributed
- Modules create and configure engines and assign them into groups
- Modules can communicate with engines using an **engine mailbox**, a lock-free queue with only one slot.



(b) engine design

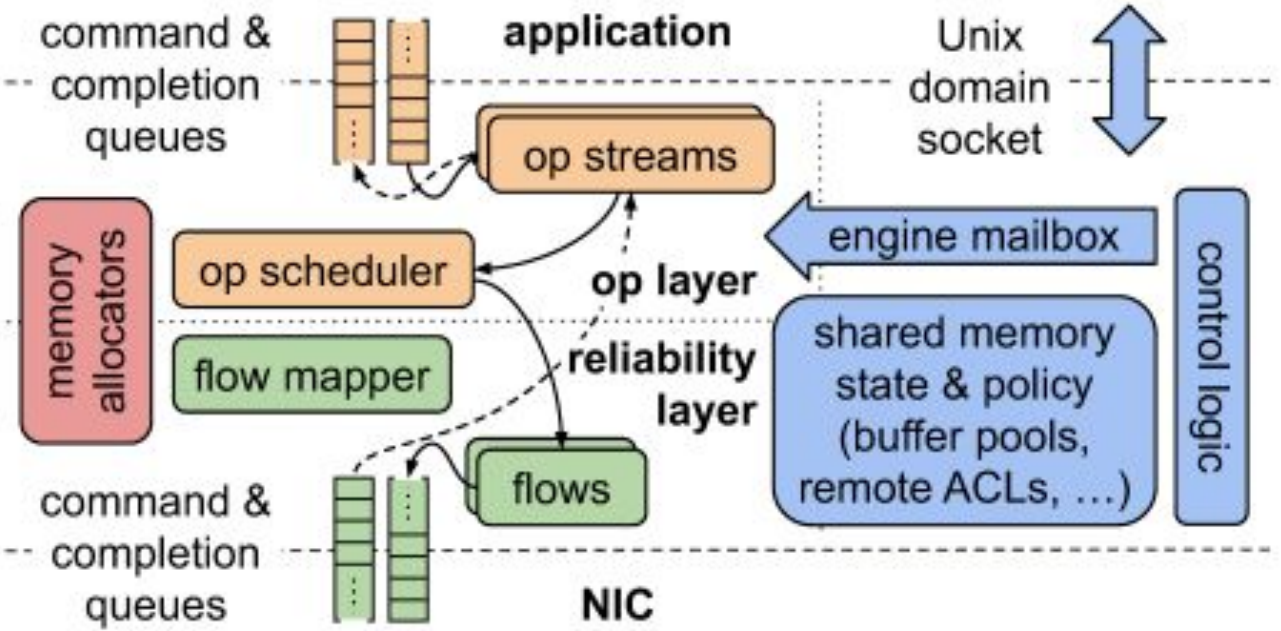


(c) engine groups and scheduling

Pony Express Overview

- An independent interface that provides asynchronous operation level commands and completions
  - Apps using this interface communicate with Pony Express through shared memory
- Supporting two-sided messaging and one sided operations like RDMA
  - Supports classic send/recv messaging for RPCs

Pony Express Architecture



One-Sided Operations

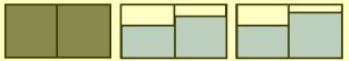
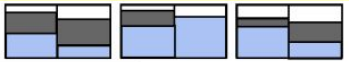
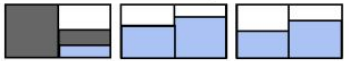
- One-sided operations are executed in the Pony Express engine to avoid overhead of scheduling the task and provides a one way trip for operations
  - Operations must run to completion
  - Without the overhead we achieve better CPU efficiency and tail latency
- We get improved latency by skipping the remote access on an application
  - We still get a request and response over the network but the remote wakeup and scheduling delay is gone
- The interface for this is defined by Pony Express and will be updated on separate releases
  - New operations go through a feature request and review process
- Flow Control is different in this case
  - Applications can’t pause senders by avoiding reads on the buffer
  - If the client over run it regardless then the requests are dropped

Hardware Offloading

- The search for points to offload and other accelerators are important as Moore's law slows down
  - CPU cores are not getting as fast as Moore's Law states so moving work elsewhere is a point to continue improving efficiency
- Primarily PE uses stateless offloads to offload memory copy operations
  - Stateless means the hardware does a self-contained job per request and keeps no connection state
  - This makes the action simpler and less bug prone than full RDMA NICs
- Snap features a kernel module to provide the capability for asynchronous interactions on DMA
  - I/OAT is a kernel managed device so the module exists to expose it in userspace
  - DMA is asynchronous so it complements Snap's continuously running packet-processing pipelines where an engine can start a copy and keep working on other packets
  - Linux traditionally features synchronous socket system calls which blocks us

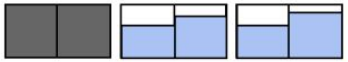
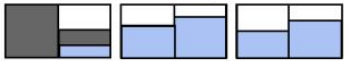
Engine Group Type – Dedicating Cores

- Engines are assigned to dedicated logical CPU cores
- Spin-Poll continuously for minimal latency
- **Trade-off –** Very low latency but poor CPU utilization when traffic is low

scheduling mode	CPU resources	scheduling latency		CPU efficiency	visualization
		median	tail		
dedicating cores	static	0-1µs	0*-100+µs	poor	
spreading engines	dynamic	3-10µs	10-30**µs	good	
compacting engines	dynamic	0-5µs	50-100**µs	excellent	

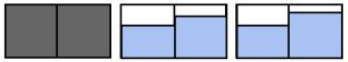
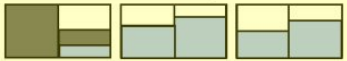
Engine Group Type – Spreading Engines

- Each engine gets its own thread
- Threads sleep when idle and wake upon interrupt
- Active threads are scheduled onto available cpu cores
- **Trade-off –** Low tail latency but interrupt wakeup can suffer from system level interference

scheduling mode	CPU resources	scheduling latency		CPU efficiency	visualization
		median	tail		
dedicating cores	static	0-1µs	0*-100+µs	poor	
spreading engines	dynamic	3-10µs	10-30**µs	good	
compacting engines	dynamic	0-5µs	50-100**µs	excellent	

Engine Group Type – Compacting Engines

- Tries to run multiple engines on as few CPU cores as possible
- Multiple engines share a single thread
- Snap monitors queueing delays and redistribute engines across threads when load increases
- **Trade-off –** Great CPU efficiency but higher tail latency due to rebalancing

scheduling mode	CPU resources	scheduling latency		CPU efficiency	visualization
		median	tail		
dedicating cores	static	0-1µs	0*-100+µs	poor	
spreading engines	dynamic	3-10µs	10-30**µs	good	
compacting engines	dynamic	0-5µs	50-100**µs	excellent	

Snap and Falcon

- Lessons carried over
  - Transport on operations rather than byte stream
    - Pony Express exposes asynchronous operation level commands
    - Falcon’s interface is aware of the transaction type (Push or Pull)
  - Two-layer transport
    - Pony express separates the operation and reliability layer
    - Falcon does the same through the transaction layer and packet delivery layer
  - Both value high velocity in software updates
    - Through pony express and the FAE respectively

Beyond the Papers – Google's Encryption Problem

Google's Problem: Software encryption took 0.7% of Google's processing power to encrypt/decrypt RPCs.

Solution: Offload encryption to hardware

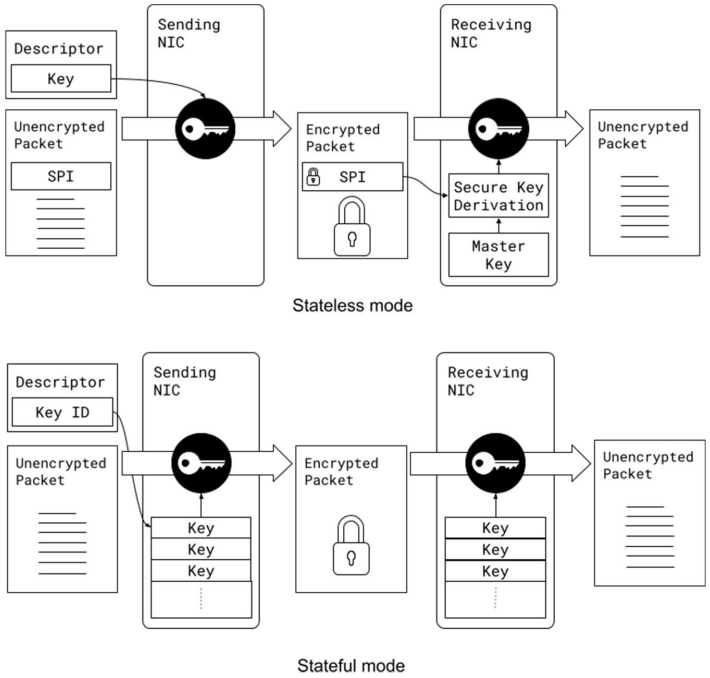
Existing Methods –

- Transport Layer Security (TLS): meets security standards but not offload-friendly due to tight coupling between kernel connection state and hardware. Also does not support UDP.
- Internet Security Protocol (IPsec): off-loadable to hardware but cannot support google's scale. IPsec hardware must store the full encryption state → for 256B entry means 5GB for 10M connections. Also does not meet security requirements.

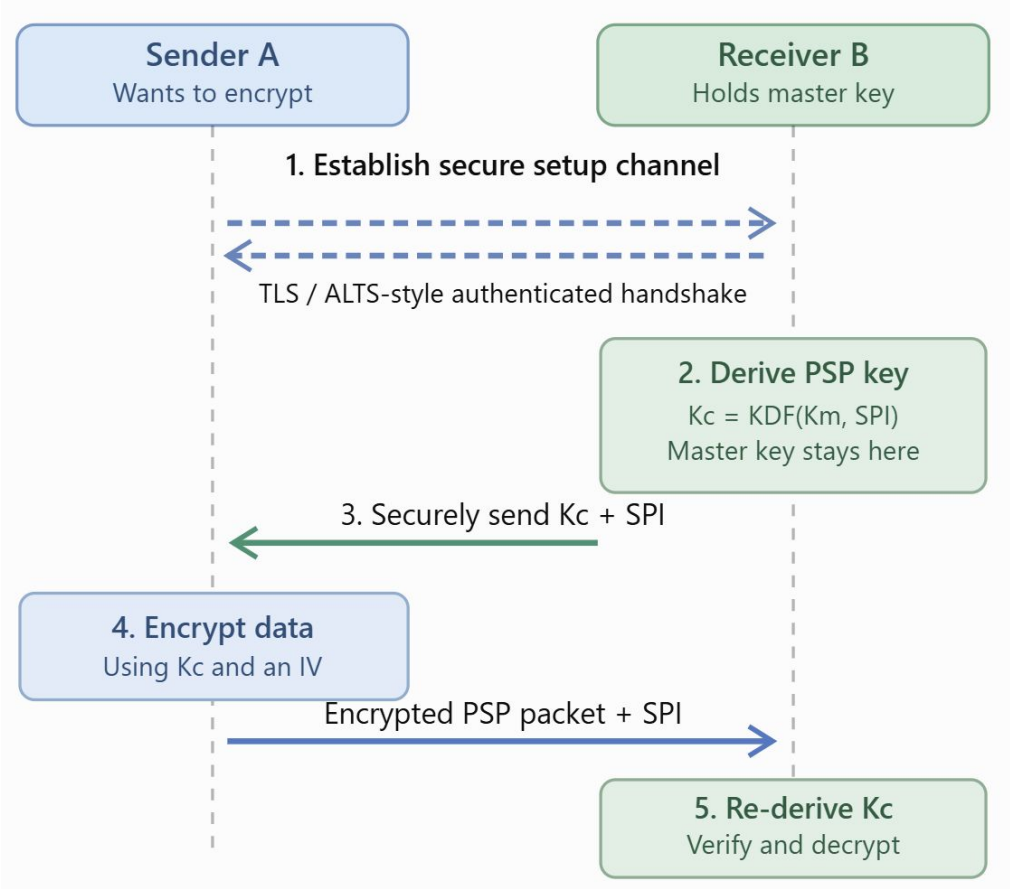
Beyond the Papers – Google's PSP

- In the normal **stateful mode**, the NIC stores a secret key for every connection
- Innovation is **stateless mode –** a function derives the key using SPI (some index) and a master key.

$$K_{secret}=F(SPI, K_{master})$$

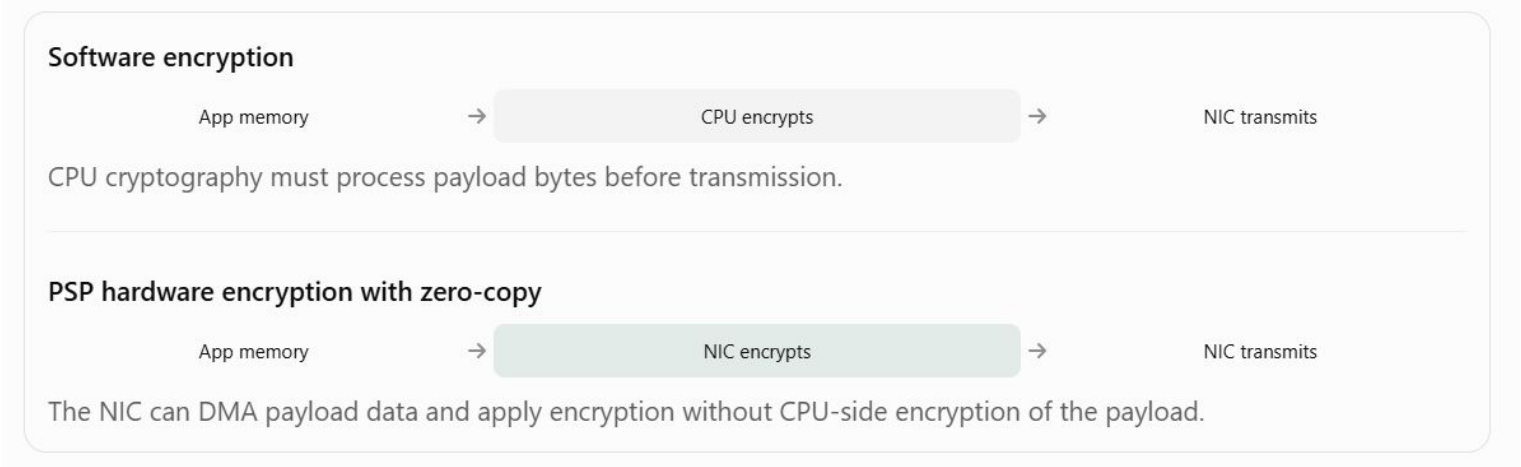


Beyond the Papers – PSP Workflow



Beyond the Papers – PSP Results

- PSP saves 0.5% of Google’s processing power
- For large 1MB RPCs, 3x speedup when combined with zero-copy techniques (e.g. Pony-Express/Snap can use zero-copy)



Beyond the Papers – PSP Next to Falcon

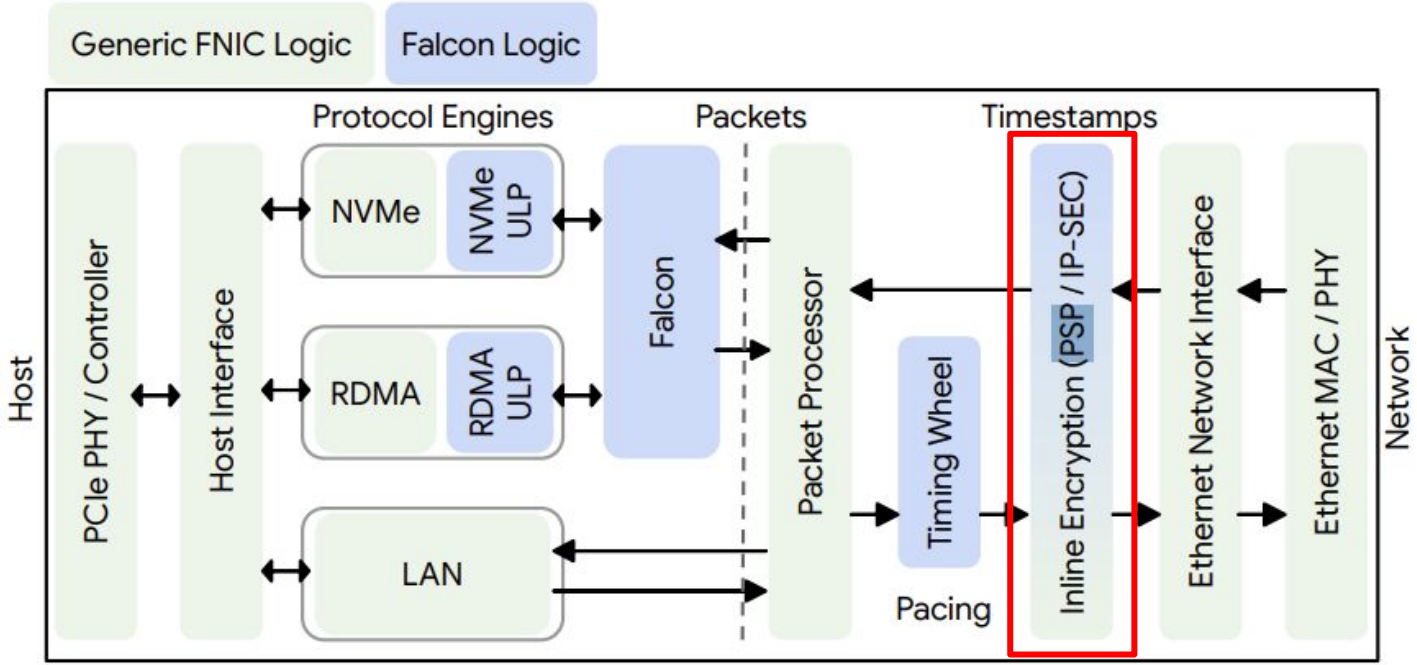


Figure 7: Falcon Implementation in a NIC

Beyond the Papers – Implementing Reinforcement Learning Datacenter Congestion Control in NVIDIA NICs

- **Train the RL agent in simulation.** Each flow is controlled independently by an agent that observes its own RTT, transmission rate, and previous actions. The agent learns when to increase or decrease its transmission rate to balance throughput and congestion.
- **Distill the policy into decision trees.** The original RL policy uses an LSTM neural network to incorporate historical information. The authors first replace the LSTM with an MLP using a sliding window of observations, then train gradient-boosted decision trees to imitate the learned policy.
- **Deploy the distilled policy on real NICs.** The resulting decision trees are implemented as simple conditional statements using NVIDIA's Programmable Congestion Control (PCC), allowing them to run directly on ConnectX-6Dx NIC processors.

Discussion

Both PSP and Falcon off-load critical functionality to hardware for a performance increase.

- What do you think of this trend?
- What other functionalities exist today that we may be able to offload to hardware? What functionalities cannot be offloaded to hardware?
- How does this relate to the end-to-end argument?