# OCP NIC Core Features Specification

Version 1.0

Author: OCP Networking workgroup, OCP NIC-SW subgroup
Author: Dan Daly (Intel)
Author: Jakub Kicinski (Meta)
Author: Willem de Bruijn (Google)

# Abstract

This document standardizes common features and behavior of server-grade network interface cards.

# Contents

# License

**Notes**:

1) The above license does not apply to the Appendix or Appendices. The information in the Appendix or Appendices is for reference only and non-normative in nature.

NOTWITHSTANDING THE FOREGOING LICENSES, THIS SPECIFICATION IS PROVIDED BY OCP "AS IS" AND OCP EXPRESSLY DISCLAIMS ANY WARRANTIES (EXPRESS, IMPLIED, OR OTHERWISE), INCLUDING IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, FITNESS FOR A PARTICULAR PURPOSE, OR TITLE, RELATED TO THE SPECIFICATION. NOTICE IS HEREBY GIVEN, THAT OTHER RIGHTS NOT GRANTED AS SET FORTH ABOVE, INCLUDING WITHOUT LIMITATION, RIGHTS OF THIRD PARTIES WHO DID NOT EXECUTE THE ABOVE LICENSES, MAY BE IMPLICATED BY THE IMPLEMENTATION OF OR COMPLIANCE WITH THIS SPECIFICATION. OCP IS NOT RESPONSIBLE FOR IDENTIFYING RIGHTS FOR WHICH A LICENSE MAY BE REQUIRED IN ORDER TO IMPLEMENT THIS SPECIFICATION. THE ENTIRE RISK AS TO IMPLEMENTING OR OTHERWISE USING THE SPECIFICATION IS ASSUMED BY YOU. IN NO EVENT WILL OCP BE LIABLE TO YOU FOR ANY MONETARY DAMAGES WITH RESPECT TO ANY CLAIMS RELATED TO, OR ARISING OUT OF YOUR USE OF THIS SPECIFICATION,

# Compliance with OCP Tenets

## Openness

This specification defines in an open format a common core set of NIC features. The express goal is to obsolete and replace the historical behavior of exchanging (imprecise) requirements only under NDA one-on-one between operator and vendor. The open specification is accompanied by a publicly available and open source testsuite to enable self-certification.

## Impact

The specification creates a level playing field and open ecosystem for server NICs, by defining a precise feature specification and testsuite. This simplifies validation and acceptance of new vendors, and ensures interoperability in heterogeneous data centers.

## Scale

This specification explicitly designs for large scale deployment in "hyperscale" data center environments. It describes hyperscale workloads quantitatively, presents synthetic tests to evaluate for these workloads, and gives design directions both at high level and as concrete features. It explicitly proposes stateless offloads, for instance, to build hardware that scales O(1) with connection count, rather than the commonly observed O(N).

## Efficiency

NIC feature offloads move hot path operations from the CPU to fixed function logic. This can result in up to an order of better efficiency, as measured in transistor density.

## Sustainability

As a software specification, this document does not directly set hardware parameters that may affect sustainability, e.g., through power draw. But, the increased efficiency of fixed function offloads concomitantly conserves energy compared to CPU execution. It is imperative that offloads are implemented correctly. The authors have ample experience with offloads that have to be disabled in practice. This specification aims to avoid such situations.

# Introduction

This specification defines a core feature set that constitutes a minimum requirement for a server grade network interface card (NIC). It prescribes feature set and behavior, gives a rationale for the choices made and lists conformance tests.

Network interface cards can offload the host CPU by offloading network packet processing to fixed-function hardware on the NIC. Modern servers have come to depend on these I/O offloads, to the extent that a broken or missing implementation may disqualify a device for deployment.

## Why Standardization

Experience shows that even well known and widely deployed features have subtly different behavior between devices due to incomplete feature specifications, protocol peculiarities and unclear operating conditions. This document aims to define features in sufficient detail to avoid ambiguity, warn about common implementation bugs and model representative workloads. Fundamentally, it aims to share and codify network device expertise, in an open format that is publicly accessible, unencumbered by NDAs and based on broad input from across the industry.

## Target

Target platforms are high-end servers with many CPU cores, 100+ Gbps and 100+ Mpps. Though applicability is likely wider, the principal target is large scale deployment in data centers ("hyperscale"). That environment adds requirements for interoperability of heterogeneous hardware and monitoring at scale.

## Scope

This specification covers the core offloads that may be required of every NIC in the server domain. Sufficiently novel features that cannot be considered standard are out of scope. Sufficiently complex features are left for separate targeted specifications.

Specifically excluded complex features are inline cryptography and virtualization support, including smartNICs ("IPU", "DPU"), virtual switch abstractions, PCI Virtual Functions, SR-IOV and Scalable IOV.

Also excluded are hardware requirements, such as power usage. Those are captured by the OCP NIC hardware spec 3.0 [ref_id:ocp_nic]. The two specifications are independent. A NIC can conform to this spec without conforming to the hardware spec and vice versa.

Multi-host devices, as defined in the OCP NIC 3.0 spec[ref_id:ocp_nic], are in scope. All requirements are specified per device, and are assumed to be distributed equally across hosts.

### Workloads

Hyperscale servers are deployed in planetary scale environments, with many sites of tens of thousands or more servers. Each server can run hundreds of tasks across hundreds of cores. Each task can communicate with tens of thousands of peers. Network incast and outcast can reach millions of connections per host, tens of millions at the tail.

At this scale, connection establishment rate becomes a significant design consideration besides absolute connection count. From experimental results we derive 100K connections/sec per host as the minimum level that must be supported, and an order of magnitude higher to be future proof for the expected lifetime of new devices. At these levels, systems that scale O(1) with connection count are strongly preferred over those that scale O(N). This is the principal reason to prefer stateless device offloads.

Hyperscale servers today should be expected to scale at the 99% to at least

- connection count: 10M TCP/IP connections
- connection rate:100K connections/sec

Hyperscale workloads can be mixes of many applications. They can be generalized into three types.

- high priority, latency sensitive, such as user facing traffic
- low priority, latency insensitive, such as map-reduce style jobs
- high performance computing: dominated by machine learning workloads

Machines may run a mix of workloads to increase hardware utilization. This way they can offer assured service to high priority tasks, while scheduling low priority tasks on surplus resources at best effort. This model requires strong quality of service isolation to meet latency sensitive traffic service level objectives (SLOs).

## Interface

This specification covers behavior of the device-dependent driver as observed by the host operating system. Hardware implementation details are expressly out of scope. No specific device API is prescribed.

The host operating system is not defined. The same features can and often are used by multiple operating systems. In practice, requirements are derived from experience with Linux, the most widely used operating system in hyperscale deployments.

## Validation

The specification has an accompanying open source testsuite. Where possible, feature sections conclude with an introduction of an open source conformance test. The testsuite covers the majority of offloads, but is a work in progress. It is intended to be an on-going community effort.

## Certification

The goal is to allow for self certification: vendors can qualify their hardware against the spec and publish the results. Doing so reduces repetitive work, as it moves qualification from a large community of customers to the smaller set of vendors. By introducing a shared language and test suite, both unencumbered by legal limitations, it can also simplify communication between customers and vendors.

Devices may not conform fully to the spec. This is understood and acceptable. The specification is a starting point. A customer and vendor can agree to drop certain requirements and add or adjust others. The specification is an initial blueprint to these conversations. Vendors SHOULD publish their conformance to the specification, with an explicit list of known deviations. This can take the form of a vendor column to the list in Appendix A, plus optional clarification text for the deviations.

## Style and Terminology

This document adopts IETF style as specified in [RFC2119], [RFC2223] and [RFC7322]. The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

"Host" and "device" define the two sides of the specified network interface: the device-independent host operating system on the one hand and the network interface card including its device-dependent driver code on the other. The specification covers the *behavior* of the device in this relationship.

## Glossary

- Device: Synonym for network interface card (NIC).
- Driver: Device-dependent operating system code to interact with the device.
- Queue: Asynchronous communication channel between driver and device.

## Contact

This specification was created through the NIC software effort with the OCP Networking project by OCP member companies Google, Intel and Meta.

Comments, questions, suggestions for revisions and requests to join the standard committee can be directed to the OCP Networking mailing list. See opencompute.org/projects/networking for details.

Detailed feedback from the wider OCP networking community improved this document. The authors want to thank all participants who took the time to review the specification and give feedback.

# I/O API

A server network interface must handle tens of millions of packets per second or more. To scale to these rates (1) datapath communication between host and device uses asynchronous shared memory queues and (2) the device MUST be able to expose multiple concurrent queues for both transmit and receive processing to scale traffic processing across CPUs.

## Queues

The device and driver communicate through descriptor queue buffers in host RAM. The format of individual descriptors is device dependent, out of scope for this document.

For host to device communication, the descriptor queue is combined with a MMIO writable doorbell register exposed through a PCI BAR to write the host producer index and notify the device of new data.

For device to host communication, the device may write the device producer index to an agreed on location in host RAM. Alternatively, the explicit producer index field may be replaced by embedding a generation bit in packet descriptors to detect the head of the queue.

Each queue MUST be able to hold at least 4096 single-buffer packets at a time. The exact length should be configurable, in which case it MUST be host configurable. The device SHOULD support queue reconfiguration while the link remains up.

The asynchronous queue is associated with an IRQ for optional interrupt driven processing.

### Post and Completion Queues

Each logical queue SHOULD consist of a pair of post and completion queues. Post queues post buffers in host RAM from host to device. Completion queues post ready events from device to host.

Post queues MAY be shared between queues, such that a single post queue can supply multiple receive queues. Completion queues MAY be shared between queues, such that a single completion queue can return buffers posted to the device from multiple post queues. If shared queues are supported, then this MUST be an optional feature. All unqualified declarations of supported number of queues MUST be calculated with no sharing.

### Scatter-Gather I/O

The device MUST support scatter-gather I/O.

Transmitted packets may consist of multiple discrete host memory buffers. The device MUST support a minimum of (MTU / PAGE_SIZE) scatter-gather memory buffers for MTU sized packets, rounded up to the nearest natural number, plus a separate header buffer. For packets with segmentation offload (see below), the device must support this number times the maximum number of supported segments, with an absolute minimum of 17: the minimum number of 4KB pages to span a 64KB TSO packet. Again, plus a separate header buffer.

For the receive case, the host may choose to post buffers smaller than MTU to the receive queue. The device must support the same limits as for transmit queues: the absolute minimum of 2 buffers per packet and the relative minimum of (MTU / PAGE_SIZE) in the general case, and the absolute minimum of 17 and the relative minimum of N * (MTU / PAGE_SIZE) for large packets produced by Receive Segment Coalescing (RSC, below).

*Optimization: RAM Conservation*

A device MAY support scatter-gather I/O with multiple buffer sizes. It may support the driver posting multiple buffer sizes to the device. One approach stripes different buffers of expected header and payload sizes in the same post queue. Another is to associate multiple post queues with a receive completion queue, where each post queue posts buffers of a single size. The device then selects for each packet the smallest buffer(s) suitable for storing it. A practical example is supporting 9K jumbo frames in environments where the majority of traffic may consist of standard 1500B frames and smaller pure ACK style packets.

The device MAY also support sharing post queues among receive completion queues. This mitigates scale-out cost. In receive processing, buffers have to be posted to the device in anticipation of packet arrival. With many queues, total posted memory can add up to a significant amount of RAM allocated to the device.

Devices MAY also support an “emergency reserve” queue, a single extra queue of buffers available to use on any receive queue, if the buffers dedicated to that queue are depleted. This allows the host to post fewer dedicated buffers while avoiding the risk of transient traffic bursts leading to drops.

*Receive Header-Split*

A device SHOULD support the special case of receive scatter-gather I/O that split headers from application layer payload. It SHOULD be possible to allocate header and data buffers from separate memory pools.

All protocol header buffers for an entire queue may be allocated as one contiguous DMA region, to minimize IOTLB pressure. In this model, the host operating system will copy the headers out on packet reception, so the region need only allocate exactly as many headers as there are descriptors in the queue.

Header-split allows direct data placement (DDP) of application payload into user or device memory (e.g., GPUs), while processing protocol headers in the host operating system. The operating system is responsible for ensuring that payload is not loaded into the CPU during protocol processing. Data is placed in posted buffers in the order that it arrives. Transport layer in-order delivery in the context of DDP is out of scope for this spec.

Header-split SHOULD be implemented by protocol parsing to identify the start of payload. The protocol option space is huge in principle. This spec limits to unencapsulated TCP/IP, which covers the majority of relevant datacenter workloads (crypto is deferred to a future version of the spec). Protocol parsing can fail for many reasons, such as encountering an unknown protocol type. Then the device MUST allow falling back to splitting packets at a fixed offset. This offset SHOULD be host configurable.

Header-split MAY be implemented with only support for a fixed offset: Fixed Offset Split (FOS). This variant does not require protocol parsing and is thus simpler to implement. Workloads often have a common default protocol layout, such as Ethernet/IPv6/TCP/TSopt. Splitting at 14 + 40 + 20 + 12 will correctly cover this modal packet length and with that the majority of packets arriving on a host.  True header split is strongly preferred over FOS, and required at the advanced conformance level. If FOS is implemented, the offset MUST be host configurable.

*PCIe Cache Aligned Stores*

Stores from device to host memory SHOULD be complete cache lines when possible. The device SHOULD store the last cacheline of a packet with padding to avoid a RMW cycle. It SHOULD do the same for headers when header-split is enabled.

A partial write results in a read-modify-write (RMW) cycle across the PCIe bus, increasing latency and bus contention. With current Ethernet, PCIe and memory speeds, this has been observed to cause significant bus contention and packet drops in practice. That behavior can escape synthetic network benchmarks, but is apparent in real-world deployments, where memory and PCI see contention from other applications and devices besides networking.

## Interrupts

The device MUST support interrupt driven operation, where it signals the host of ready events by sending a hardware interrupt. The device MUST also support polling mode, where the host keeps device interrupts masked or disabled.

Interrupts in this context are understood solely as Message Signaled Interrupts (MSI-X) messages across a PCIe bus.

### Moderation

The device has to notify the host on data ready events on its completion queues. It MUST implement interrupt moderation on both transmit and receive queues. If the device supports adaptive interrupt moderation, it MUST support disabling this.

#### *Delay*

The device MUST support configuring a minimum delay before an interrupt is sent. This MAY be measured from either the previous interrupt, or the most recent unmask of interrupts. The timeout range must span at a minimum from 2 to 200 microseconds, in 2 microsecond steps or better.

#### *Count*

The device SHOULD also support configuring a maximum event count until an interrupt is sent. This triggers an interrupt when a configurable number of events since the last interrupt is reached. Each event corresponds to a single received or transmitted packet. For TSO/RSC packets, the COUNT should count each segment separately. When supporting a maximum event count, the device MUST support values in the range of [2, 128]. It then MUST send an interrupt when either of the two interrupt moderation conditions is met, whichever comes first. Reaching the maximum number of events immediately raises an interrupt regardless of remaining delay, so the delay constitutes an upper bound. Triggering an interrupt for either limit MUST lead to both counters being reset.

*Tx and Rx*

The configuration parameters for Rx and Tx queues MUST be independent. Receive interrupts are more time sensitive than transmit completion interrupts, translating in a lower interrupt moderation threshold and thus higher interrupt rate. If the interrupt moderation is performed at the level of a mixed-purpose completion queue (holding both Rx and Tx completion events) the moderation logic SHOULD remain separate per direction. Triggering an interrupt for either event type SHOULD lead to both state machines being reset.

*Timer Reset Mode*

The device MAY additionally support timer reset mode (TRM). In this mode the timer is reset on each event. In this mode a delay-based interrupt is sent only if no event occurs within a timeout period, signaling an idle queue. In this mode, maximum interrupt delay is unbounded. If the device supports a maximum event count, then it MUST also respect this in TRM mode.

### MMIO Transmit Mode

The device may offer an optional low latency transmission path that writes descriptors directly to device memory using memory mapped I/O (MMIO), bypassing the asynchronous descriptor queue and device DMA operation to fetch the descriptor contents.

The device may further offer the option to store entire packets in device memory using MMIO, bypassing both DMA descriptor and packet contents fetch. This is generally limited to small packet sizes.

## Multi Queue

A device MUST support from 1 up to 1024 logical queues per device. Number of queues MUST be host configurable. It is acceptable to require the device link to be brought down to reconfigure queue count.

*Independent Transmit and Receive Queues*

The number of Tx and Rx queues MUST be configurable independently. No relationship between the two should be assumed.

Receive and transmit processing generate CPU cycle cost in different ways. The interrupt moderation section compares interrupt frequency trade-offs. Transmit cost is sensitive to cacheline and lock contention when a Tx queue is shared between CPUs. Something that does not happen on receive. Thus Tx and Rx can have different optimal numbers of queues.

### Flow Steering

With multiple receive queues, the network interface needs to implement queue selection. It MUST support RSS load balancing and MAY advertise accelerated RFS or programmable flow steering. If it advertises either, then that implementation MUST follow the feature requirements defined here.

#### *Receive Side Scaling*

A device MUST support load balancing with flow affinity using Receive Side Scaling (RSS). This algorithm combines (a) field extraction rules for packet steering with flow affinity, (b) a hash function for uniform load balancing that incorporates a secondary hash input for DoS resistance and (c) an indirection table to optionally implement non-uniform weighted load balancing.

##### *Field Extraction*

Queue selection must be flow affine, forwarding all packets from a transport flow to the same queue, so that packets within a flow are not reordered. Transport protocol performance can degrade when packets arrive out of order, which is likely to happen with simpler round robin packet spraying.

RSS defines two rules to derive queue selection input in a flow-affine manner from packet headers. Selected fields of the headers are extracted and concatenated into a byte array. If the packet is IPv4 or IPv6, not fragmented, and followed by a transport layer protocol with ports, such as TCP and UDP, then extract the concatenated 4-field byte array { source address, destination address, source port, destination port }. Else, if the packet is IPv4 or IPv6, extract 2-field byte array { source address, destination address }. IPv4 packets are considered fragmented if the more fragments bit is set or the fragment offset field is non-zero.

If a packet contains multiple IPv4 or IPv6 headers, then RSS operates on the first IPv4 or IPv6 header and the immediately following transport header, if any. The IPv6 flowlabel field may also be included. If present, this MUST be placed immediately before the source address in the byte array.

The same fields from subsequent IPv4/IPv6 and transport headers MAY be appended to the byte array, if present. These extensions are optional and MUST be configurable if supported. A basic version of RSS without optional extensions MUST always be supported, to be able to perform explicit flow steering by reversing the algorithm.

### *Toeplitz Hash Function*

The device MUST support the Toeplitz hash function~[ref_id:toeplitz_hash] for Receive Side Scaling~[ref_id:ms_rss]. A hash function maps the byte array onto a 32-bit number with significant entropy to serve as effective input for uniform load balancing.

The Toeplitz function takes two inputs, the byte array derived from the packet P and a second byte array S of at least 40 bytes that is constant between packets. S, called the secret, is mixed into the entropy for DoS prevention. It makes queue prediction hard for a given packet unless the secret is known. Toeplitz trades off execution speed and security: it is not a cryptographically secure hash function. The secret MUST be readable and writable from the host. Explicit queue prediction has legitimate use cases, so the secret must be discoverable by trusted parties.

The secret S is converted to a Toeplitz matrix Sm, a matrix in which each left-to-right descending diagonal is constant. Due to this property, a Toeplitz matrix is fully defined by its first row and column. An N * M Toeplitz matrix is defined by an N + M - 1 length source vector.

S has to be long enough to match against each bit in the longest possible RSS input vector. That is 2 16B IPv6 addresses plus 2 2B ports, for 36B == 288b.

A minimal RSS Toeplitz matrix is a binary Toeplitz matrix of 288 x 32 bits. 288 is one row for each bit in P. 288 + 32 - 1 == 319 bits to define the matrix establishes the minimum 40B required length of secret S. The minimum secret key length MUST be no less than 40B. Due to optional extended inputs, larger secrets MAY be supported. A range of 40-60B is common. Matching MUST always begin at bit zero, regardless of configured key length. The first row consists of the left-most 32 bits of the array. The remainder define the first bit of each subsequent row.

The Toeplitz hash function performs a scalar multiplication between the Toeplitz matrix Sm and the input array P. Each bit in $P_i$ is multiplied with the 32b row $Sm_i$. The output array O is converted to a scalar value by an XOR of all elements in O. The below reference implementation demonstrates the algorithm. Care must be taken surrounding endianness and bit-order (traverse a byte from MSB to LSB). See Appendix B for validation.

```
uint32_t toeplitz(const unsigned char *P,
                  const unsigned char *S)
{
    uint32_t rxhash = 0;
    int bit;

    for (bit = 0; bit < 288; bit++)
        if (test_bit(P, bit))
            rxhash ^= word_at_bit_offset(S, bit);

    return rxhash;
}
```

The device MAY support other hash functions besides Toeplitz. Then function selection must be configurable.

#### *Receive Hash*

The computed 32b hash SHOULD be passed to the host alongside the packet. Doing so allows the host to perform additional flow steering without having to compute a hash in software, such as Linux Receive Flow Steering (RFS).

A device MAY compute a 64b field to reduce collisions. It MAY communicate this instead, as long as either the 32b Toeplitz hash can be derived or can be communicated alongside.

#### *Indirection Table*

The device MUST select a queue by reducing the hash through modulo arithmetic. It applies division to the hash value and uses the remainder as an index into a fixed number of resources. The divisor is not simply the number of receive queues. RSS specifies an additional level of indirection, the indirection table. This allows for non-uniform load balancing. The device MUST support the RSS indirection table. The device MUST lookup a queue using the following modulo operation:

```
queue_id = rss_table[rxhash % rss_table_length];
```

The table MUST be host-readable and writable. The host may configure the table with fewer slots than the configured number of receive queues, if the host wants to apply RSS to only a subset of queues. The host may configure the table with more slots than the number of receive queues, for more uniform load balancing. The device may limit the maximum supported table size. The minimum supported indirection table size MUST be 128. The minimum SHOULD be at least 4 times the number of supported receive queues. The device SHOULD allow querying the maximum supported table size by the host. The device SHOULD allow replacement of the indirection table without pausing network traffic or bringing the device down, to support dynamic rebalancing, e.g., based on CPU load.

*Accelerated RFS*

RSS does not maintain per-flow state. A device MAY also implement the stateful Accelerated RFS (ARFS) algorithm, which explicitly records a preferred queue for a given flow hash. If the device advertises this feature, it MUST be implemented as described in this section.

In Linux, Receive Flow Steering (RFS) is a software algorithm that steers receive processing of a packet to the CPU that last ran an application thread for the same flow. It identifies the flow that a packet belongs to by a flow hash, Optionally and preferably, that is the RSS hash received from the device.

RFS introduces a map from flow hash to CPU. When an application thread interacts with a flow, the host stores the CPU ID in `rfs_table[hash % rfs_table_length]`. When the host processes a packet from the receive queue, it looks up this table entry, queues the packet on a host queue for the given CPU and sends an inter-processor-interrupt (IPI) to trigger receive processing on the CPU affine with the application thread.

Accelerated RFS moves the RFS table to the device. This directly wakes the RFS affine CPU, skipping over RSS and IPI. The feature can be implemented with an explicit lookup table as described, or as a list of match/action rules that match on a hash or its source fields. In all cases, the action is to queue the packet on a specific queue or RSS context (see below). The host is responsible for storing a queue ID that results in interrupt processing on the same CPU as recorded at the application layer.

If ARFS is supported, regardless of implementation, the device MUST present a match/action API with match on L4 hash and queue selection action. It may offer an API that inserts and/or removes multiple rules at once.

If ARFS is enabled and an ARFS match for a hash is found, then this takes precedence over RSS. Else the device MUST fall back onto RSS.

ARFS is not suitable for all workloads. If connection churn or thread migration is high, it can introduce significant table management communication across the PCI bus.

#### *Self-learning ARFS*

ARFS may alternatively be implemented entirely on the device. In this case the device programs the match/action table for ingress matching based on sampling of egress traffic. This requires matching a transmit queue to a receive queue and thus assumes a M:1 mapping of transmit to receive queues. Care must be taken to ensure that self-learning ARFS does not cause packet reordering within a flow.

### *Programmable Flow Steering*

A device MAY support more complex match rules for flow steering. ARFS matching by hash can be seen as one instance of a broader match language, which may match on

- Ethernet source and/or destination address, protocol
- VLAN identifier
- MPLS label
- IPv4/IPv6 source and/or destination address
- IPv4 other header fields, including ToS, protocol
- IPv6 other header fields, including Traffic Class, next header
- UDP/TCP ports
- TCP flags
- Opaque data+mask bit arrays, with fixed offset and length.

If IPv4 flow steering is supported, then IPv6 MUST also be supported.

If flow steering rules are inserted and a rule matches a packet then this MUST take precedence over RSS and ARFS.

#### *RSS contexts*

Flow steering MAY be combined with RSS to steer a flow not to a single queue, but to a group of receive queues: an RSS context.

A device MAY support multiple RSS contexts, plus extended match/action rules that select an RSS context instead of a queue. Each context is then configured as described in the RSS section. In RSS configuration, a context is selected through an additional numeric ID. The RSS context applied to traffic that does not match any of the explicit flow steering rules MUST be context 0.

It is not required that queues are exclusively assigned to a single RSS context. The same queue may appear in the indirection tables of multiple contexts.

RSS context 0 MUST be able to program the complete RSS indirection table if no other contexts are in use. Other contexts MUST be able to program tables of up to 64 entries each. RSS context 0 SHOULD be able to program all table entries not in use by other contexts.

### Transmit Packet Scheduling

When multiple Tx queues have packets outstanding, the device must choose in which order to service the queues.

The device MUST implement equal weight deficit round robin (DRR) as default dequeue algorithm [ref_id_fq_drr]. Deficit round robin is a per-byte algorithm. Time is divided in rounds. Each queue earns a constant number of byte credits during each round, its quantum. The device services queues in a round robin order. If a queue has data outstanding when it is scanned, all packets that add up to less than the queue's quantum are sent and the credit is reduced accordingly. If one or more packets cannot be sent because the packet at the head of the queue is longer than the remaining quantum, then the remaining quantum carries over to the next round. If the queue is empty at the end of a round, the remaining quantum is reset to zero.

The device SHOULD also support DRR with non-equal weights. Then it MUST support host configuration of the weights. This specification does not prescribe a specific interface to program the weights. In Linux, this feature does not have a standard API as of writing.

The device MAY offer additional algorithms. If strict priority is supported, it SHOULD implement this mode with starvation prevention.

# Offloads

Fixed function hardware is more efficient at specific operations than a general purpose CPU. It is an effective optimization for both memory intensive operations such as checksum computation, and computationally expensive operations like encryption.

## Design principles

### Stateless

Where possible, offloads SHOULD be implemented in a stateless manner. That is, all information associated consumed or produced for a packet is communicated along with the packet.

A stateful implementation may store per-flow state on the device, requiring additional communication to keep host and device state in sync. At the scale of modern servers and hyperscale deployments, this adds complexity and additional performance limitations. It should be avoided where possible. See the performance section for a summary of the canonical workloads and their scale.

### Protocol Independence

Where possible, offloads SHOULD be implemented in a protocol independent manner. Protocol dependent offloads are fragile, in that they break when protocols are revised or replaced.

Tunneling and transport layer encapsulation are common in hyperscale systems. Protocols used may be standard, such as Generic Routing Encapsulation (GRE) as defined in IETF RFCs 2784 and 2890. But even standard protocols can be extended, for example GRE key and sequence number extensions of RFC 2890. Hyperscale providers operate in a closed world, and as such are not limited to standardized protocols. They may extend standard headers with proprietary fields or entirely replace standard protocols with custom ones. They may stacks protocols in arbitrary ways to encapsulate multiple layers of information, e.g., for routing, traffic shaping and sharing application metadata out-of-band (OOB).

*Programmable Parsers*

Devices with a programmable hardware parser allow the administrator to push firmware updates to support custom protocols. A programmable parser is still strictly less desirable than protocol independent offloads, as programmable parsers introduce correlated roll-outs between software and firmware. At hyperscale, correlated roll-outs and potential roll-backs add significant complexity and risk.

This target of protocol independence is in conflict with some features defined in this spec (RSS, header-split, etcetera). That is why the prescriptive opening sentence of this section starts with "where possible". Where features can be implemented without parsing, that design MUST be taken.

## Checksum Offload

The device MUST support TCP and UDP checksum offload, for both IPv4 and IPv6, on both receive and transmit. The device SHOULD implement these features in a protocol-independent manner, by checksumming a linear range of bytes.

### Transmit Checksum

The device MUST be able to insert a checksum in the TCP header such that the checksum conforms to IETF RFC 793 section 3.1.

The device SHOULD implement this feature in the form of a protocol independent linear ones' complement (PILOC) checksum offload. In PILOC, the host communicates the start of the byte range to sum within the packet: checksum_start. It also specifies a positive insert offset from this byte where the sum must be stored: checksum_offset. The last byte is specified implicitly: summing continues to the end of the packet payload, excluding any (e.g., link layer) trailers. The insert offset MUST support both 6B for a UDP header checksum field offset and 16B for a TCP header.

With a PILOC checksum implementation, the device does not need to be aware of protocol specific complexity, such as a pseudo header checksum. The host will insert the pseudo header sum at checksum_start + checksum_offset, to include this in the linear sum. The device therefore MUST NOT clear this field before computing the linear sum.

Legacy devices MAY implement transmit checksum offload in a protocol dependent manner, fully in hardware. In this case it parses the packet to find the start of the TCP or UDP header and the preceding IPv4 or IPv6 fields that must be included in the pseudo header. This approach is strongly discouraged, as (1) the innermost transport header may be preceded by protocol headers that the hardware cannot parse, (2) with tunnel encapsulation, a packet may contain multiple transport headers, (3) it may exclude other protocols with checksum fields, such as Generic Route Encapsulation (GRE).

*Multiple Checksums*

A packet can contain multiple transport headers, some or all of which require valid checksums. A device that implements PILOC sums MUST NOT insert multiple checksums. It only inserts a single checksum at the location requested by the host. The host can prepare all other checksums in software efficiently. Once the device inserts the innermost checksum, by definition the innermost packet sums up to zero (including pseudo header). Any preceding checksums therefore can be computed by summing over headers only. Local Checksum Offload (LCO) [ref_id:csum] computes checksums of all but the innermost transport header efficiently in software without loading any payload bytes.

*UDP Zero Checksum Conversion*

The UDP protocol as specified in IETF RFC 768 introduces a special case that MUST be handled correctly when computing checksums. A checksum that sums up to zero MUST be stored in the checksum field as negative zero in ones' complement arithmetic: 0xFFFF. A device MAY apply the same logic to all checksums in a protocol independent manner.

Transport checksums are computed with ones' complement arithmetic. In this arithmetic, a positive integer is converted to its negative complement by flipping all bits, and vice versa. Adding any number and its complement will produce all ones, or 0xFFFF. Every number thus has a complement. This includes zero: both 0x000 and 0xFFFF represent zero.

RFC 768 adds explicit support for transmitting a datagram without checksum. This is signaled by setting the checksum field to 0x0000. To distinguish this lack of checksum from a computed checksum that sums up to zero, a sum that adds up to 0 MUST be written as 0xFFFF.

### Receive Checksum

A device MUST be able to verify ones' complement checksums. The device SHOULD implement the feature in a protocol independent manner.

Protocol independent linear ones' complement (PILOC) receive checksum offload computes the ones' complement sum over the entire packet exactly as passed by the driver to the host, for every packet, excluding only the 14B Ethernet header. The sum MUST exclude the Ethernet header. It MUST include all headers after this header, including VLAN tags if present. It MUST exclude all fields not passed to the host, such as possible crypto protocol MAC footers.

It MUST be possible for the host to independently verify checksum correctness by computing the same sum in software. This is impossible if the checksum includes bytes removed by the device, such as an Ethernet FCS.

Legacy devices MAY instead return only a boolean value with the packet that signals whether a checksum was successfully verified. This approach is strongly discouraged. If this approach is chosen, then the device MUST checksum only the outermost TCP or non-zero UDP checksum (if it verifies a checksum at all) and MUST return true only if this checksum can be verified. The device SHOULD then compute the sum over the pseudo-header, L4 header and payload, including the checksum field, and verify that this sums up to zero. Note that both negative and positive zero MUST be interpreted as valid sums, for all protocols except UDP. Only for UDP does the all-zeroes checksum 0x0000 indicate that the checksum should not be verified. An implementation returning a PILOC sum does not require extra logic to address these protocol variations.

The device MUST pass all packets to the host, including those that appear to fail checksum verification. The host must be able to account, verify and report such packets.

#### *Checksum Conversion*

If a legacy device only returns a boolean value, then host software can derive from this plus the checksum field in the packet a running sum over the packet from that header onward. It can use this to verify any subsequent checksums without touching data.

### Segmentation Offload

Segmentation offload (SO) allows a host to pass the same number of bytes to the device in fewer packets. Most host transmission cost is a per-packet that is incurred as each packet traverses the software protocol stack layers. In this path, payload is not commonly accessed and thus packet size is less relevant. SO amortizes the per-packet overhead.

If a device supports SO, the host may pass it substantially larger packets than can be sent on the network. The device breaks up these SO packets into smaller packets and transmits those.

Segmentation offload depends on having checksum offload enabled, because packet checksums have to be computed after segmentation.

*Copy Headers and Split Payload*

In an abstract model of segmentation offload, the device splits SO packet payload into segment sized chunks and copies the SO packet protocol headers to each segment. We refer to this basic mechanism as copy-headers-and-split-payload (CH/SP). The host communicates an unsigned integer segment size to the device along with the packet. This field must be large enough to cover the L3 MTU range: 16b is customary, but not strictly required to meet this goal. If segment size is not a divisor of total payload length, then the last packet in the segment chain will be shorter. The device MUST NOT attempt to compute or derive segment size, because establishing that is a complex process of path MTU and transport MSS discovery, more suitable to be implemented in software in the host protocol stack.

CH/SP is a simplified model. For specific protocols, segmentation offload can have subtle exceptions in how protocol header fields must be updated after copy. This spec explicitly defines all cases that diverge from pure CH/SP. The ground truth is the software segmentation implementation in Linux v6.3. If the two disagree, that source code takes precedence.

### TCP Segmentation Offload

A device MUST support TCP Segmentation Offload (TSO), for both IPv4 and IPv6. It MUST be possible to enable or disable the feature. The device MUST support TSO with TCP options.

The device SHOULD support IPv4 options and IPv6 extension headers in between the IPv4 or IPv6 and TCP header. The device SHOULD support IPSec ESP and PSP transport-layer encryption headers between the IPv4 or IPv6 and TCP header. As with other fields, the device should treat these bytes as opaque and copy them unconditionally unless otherwise specified.

TCP is particularly suitable for segmentation offload because at the user interface TCP is defined as a bytestream. By this definition, the user may have no expectations of how data is segmented into packets, in contrast with datagrams or message based protocols.

TSO enables the host to send the largest possible IP packet to the device, ignoring any constraints on path maximum transmission unit (MTU) or negotiated TCP maximum segment size (MSS). The host TCP stack selects the current MSS for the TCP connection as segment size. This number may vary between connections and across a connection lifespan.

*TCP Header Field Adjustments*

TSO requires protocol header changes to the TCP header after CH/SP:

- Sequence number: Sequence number of previous segment + segment size.
- Flags
  - FIN, PSH are only reflected in the last segment, zero in all others
  - CWR is only reflected in the first segment, zero in all others

*IP Header Field Adjustments*

IP protocols require these changes:

- IPv4 total length is updated to match the shorter payload
- IPv6 payload length is updated to match the shorter payload
- IPv4 packets must increment IP ID unless DF bit is set
- IPv4 packet checksum is recomputed

*Extension Header Field Adjustments*

Headers between the IPv4 or IPv6 header and TCP header MUST be copied as pure CH/SP.

Authenticated encryption has to happen after SO. IPSec ESP or PSP encryption headers must be copied in a pure CH/SP manner to each segment, for further processing by downstream inline encryption logic.

### UDP Segmentation Offload

A device SHOULD support UDP Segmentation Offload (USO), for both IPv4 and IPv6. It MUST be possible to enable or disable the feature.

USO allows sending multiple UDP datagrams in a single operation. The host passes to the device a UDP packet plus segment size field. The device splits the datagram payload on segment size boundaries and replaces the UDP header to each segment.

USO is NOT the same as UDP fragmentation offload (UFO). That sends a datagram larger than MTU size, by relying on IP fragmentation. UFO is out of scope of this spec. Unlike UFO, USO does not maintain ordering. Datagrams may arrive out of order, same as if they were sent one at a time.

The device SHOULD support IPv4 options and IPv6 extension headers in between the IPv4 or IPv6 and TCP header. The device SHOULD support IPSec ESP and PSP transport-layer encryption headers between IPv4 or IPv6 header and UDP header.

UDP forms the basis for multiple high transfer rate protocols, including HTTP/3 and QUIC, and video streaming protocols like RTP. These workloads benefit from SO and form a sizable fraction of Internet workload.

*Header Field Adjustments*

Beyond CH/SP, USO requires an update of the UDP length field for the last segment if the USO payload is not an exact multiple of the segment size. It also requires the same IP and extension header field adjustments as TCP. A device SHOULD support this. Optionally, a device MAY only support USO for packets where payload is an exact multiple of segment size. The host then has to ensure to only pass such packets to the device. This mechanism forms the basis for Protocol Independent Segmentation Offload, next.

### Protocol Independent Segmentation Offload

A device SHOULD support Protocol Independent Segmentation Offload (PISO), for both IPv4 and IPv6. It MUST be possible to enable or disable the feature.

PISO codifies the core CH/SP mechanism. It extends segmentation offload to transport protocols other than TCP and UDP,  and to tunneling scenarios, where a stack of headers precede the inner transport layer. Many protocols can be supported purely with CH/SP.

In PISO, the host

1. communicates a segment size to the device along with the large packet, as in TSO/USO.
2. communicates also an inner payload offset piso_off to the device along with that packet.
3. prepares any headers before piso_off as they need to appear after segmentation.

If any of the headers include a length field, PISO requires all segments to be the same size, because the host prepares the headers exactly as they appear on the wire. PISO does not adjust them.

If a payload size leaves a remainder after dividing by segment size, the host has to send two packets to the device: one PISO packet of payload length minus remainder, and a separate no-SO packet of remainder size. This is a software concern only.

*Interaction with Checksum Offload*

piso_off is similar to, but separate from, checksum_start. It must be possible to configure both independently.

*Interaction with TSO and USO*

PISO can be combined with TSO and USO. Then piso_off points not to the start of the payload, but the start of the inner transport header, TCP or UDP. Then the protocol specific rules for the inner transport protocol must be respected. Any headers before piso_off must still be entirely ignored by the device and treated solely as CH/SP. The device cannot infer whether the offset points to a UDP or TCP header. Whether to apply pure PISO, PISO + TSO or PISO + USO will have to be communicated explicitly, e.g., with a field in a context descriptor.

PISO + TSO/USO can optionally be supported on some legacy devices that were not built with PISO in mind. If a device supports TSO with variable length IPv4 options or IPv6 extension headers, with an explicit descriptor field that passes the length of these extra headers, then this can be used to pass arbitrary headers for CH/SP processing (instead of only options or extension headers), including tunnels. In this case the device expects the outer IP or IPv6 header to be an SO header with a large length field, so not prepared for pure CH/SP. A driver can patch up this distinction from the PISO interface.

### Jumbogram Segmentation Offload

The device SHOULD support IPv4 and IPv6 jumbogram SO packets that exceed the 64 KB maximum IP packet size.

IPv6 headers have a 16-bit payload length field, so the largest possible standard IPv6 packet is 64 KB + IPv6 header (payload length includes IPv6 extension headers, if any). IPv4 headers have a 16bit total length field, so the largest possible IPv4 packet is slightly smaller: 64KB including header.

Jumbogram segmentation offload ignores the IPv6 payload length and IPv4 total length fields if zero. The host must then communicate the real length of the entire packet to the device out-of-band of the packet, likely as a descriptor field.

RFC 2675 defines an IPv6 jumbo payload option, with which IPv6 packets can support up to 4GB of payload. This configuration sets the payload length field to zero and appends a hop-by-hop next header with jumbo payload option. Unlike for IPv6 jumbograms that are sent as jumbograms on the wire, it is NOT necessary for IPv6 jumbo segmentation offload to include this jumbo payload hop-by-hop next header, as the segments themselves will not be jumbograms.

## Receive Segment Coalescing

The device MAY support Receive Segment Coalescing (RSC). If the device supports this feature, it MUST follow the below rules on packet combining.

Receive Segment Coalescing reduces packet rate from device to host by building a single large packet from multiple consecutive packet payloads in the same stream. The concept applies well to TCP, which defines payload as a contiguous byte stream.

The feature is also referred to as Large Receive Offload (LRO) and Hardware Generic Receive Offload (HW-GRO). The three mechanisms can differ in the exact rules on when and how to coalesce. RSC and LRO are originally defined only for TCP/IP. This section defines a broader set of rules. It takes the software Generic Receive Offload (GRO) in Linux v6.3 as ground truth. If the two disagree, that source code takes precedence.

Receive Segment Coalescing is the common term for this behavior. To avoid confusion we do not introduce yet another different acronym. But the RSC rules defined here differ from those previously defined by Microsoft [ref_id:msft_rsc]. At a minimum, in the following ways:

- This spec generalizes to other protocols besides IP and TCP
- This spec requires all TCP options to be supported

### *Segment size*

The device MUST pass to the host along with the large (SO) packet, a segment size field that encodes the payload length of the original packets. This field implies that packets are only coalesced if they have the same size on the wire. Coalescing stops if a packet arrives of different size. If it is larger than the previous packets, it cannot be appended. If it is smaller, it can be. If segment size is not a divisor of the SO packet payload, then the remainder encodes the payload length of this last packet.

### *Reversibility*

The segment size field is mandatory. It must be possible to reconstruct the original packet stream. This reversibility capability is a hard requirement, to be able to use RSC plus

TSO/USP/PISO for forwarding without creating externally observable changes to the packet stream compared to when both offloads are disabled.

The ground rule is that receive offload must be the exact inverse of segmentation offload. That is, if TSO/USO/PISO splits a large packet into a chain of small ones, RSC will rebuild the exact same packet. The inverse also holds. An RSC packet forwarded to a device for transmission with TSO/USO/PISO will result in the same packets on the wire as arrived before RSC coalescing.

Reconstructing the original packet stream imposes constraints on header coalescing beyond segment size. Each operation has to be reversible at segmentation offload. When fields are identical, coalescing is a trivially reversible operation. All other cases are explicitly listed below, by protocol. In exceptional cases, only where explicitly stated, do we allow information loss by coalescing packets with fields that differ.

*Stateful*

Receive Segment Coalescing is not stateless. This specification does not prescribe concrete implementation. But in an abstract design, RSC maintains a table of RSC contexts. This specification does not state a minimum required number of contexts. Each RSC context can hold one SO packet. Each flow maps onto at most one context. When a packet arrives, it is compared to all contexts. See RSS for flow matching. If a context matches a flow, the next phase enters.

A packet matches a context if it matches the flow, is consecutive to the SO packet and all header fields match. Fields match if they are the same, with some protocol-specific exceptions to this rule, all listed below.

### Context Closure

An SO context closes if a packet matches the flow, but not the other conditions. Then the data is flushed to the host and the context released. In the common case, the SO packet and incoming packet are then passed to the host as two packets. In a few specific exception cases, the incoming packet is appended to the SO packet and the single larger SO packet is passed to the host. This special case MUST happen if all fields match, except for payload size, and payload size of the incoming packet is less than the previous segments. It then forms the valid remainder of the SO packet. The same SHOULD happen also if all fields match except for PSH or FIN and either or both of these is set on the incoming packet.

*General Match Exceptions*

- Length: if shorter than previous, may be appended, then closes context.
- Checksums: must all have been verified before RO. Are ignored for packet matching.

*TCP Header Field Exceptions*

- Sequence number: must be previous plus segment size.
- Flags: FIN and PSH bit only allow appending the packet, then close context.
- Flags: all other flag differences close context without append.

To state explicitly: Ack sequence number and TCP options must match.

*IP Header Field Exceptions*

- Fragmentation: fragmented packets are not coalesced. Detection of a first fragment closes the context for a flow, if open.
- The IP ID must either increment for each segment or be the same for all segments.
    - The first is common. The second may be the result of segmentation offload.

To state explicitly: TTL, hop limit and flowlabel fields must match.

Contexts can also be closed if a maximum number of segments is reached. This maximum may be host configurable.

*Asynchronous Close*

Flows can also be closed asynchronously, due to one of two events. If the device applies RSC to a flow, it must set an expiry timer when the first packet opens an RSC context. The device must send the packet to the host no later than the timeout. The flow timeout value MUST be host readable and SHOULD be host configurable.

The host may also notify the device that it wants RSC to be disabled. Any outstanding context must then be closed asynchronously, in the same manner as if their timers expired.

### SO packet construction

The device must adjust all protocol header length fields to match the length of the combined payload.

*TCP Header Field Adjustments*

- Sequence number: Sequence number of the first segment.
- Checksum: undefined.
- Flags: FIN and PSH are set if present in the last segment.
- Flags: CWR is set if present in the first segment.

*IP Header Field Adjustments*

TSO requires protocol specific changes to the preceding IPv4 or IPv6 header of the last segment, if this is shorter than full mss:

- IPv4 total length is updated to match the SO packet.
    - Or set to zero and the below jumbo rules apply.
- IPv6 payload length is updated to match the SO packet.
    - Or set to zero and the below jumbo rules apply.
- IPv4 IP ID is the ID of the first segment.
- IPv4 checksum is valid.

*Jumbogram Receive Segmentation Offload*

Devices SHOULD support coalescing of packet streams that exceed the maximum IPv4 or IPv6 packet size. Jumbogram RSC is the inverse of Jumbogram Segmentation Offload. It solves the length field limitation in the same way: the length field MUST be set to zero and the length communicated out-of-band, likely as a descriptor field.

Jumbogram RSC MUST only be applied if total length exceeds the IPv4 total length or IPv6 payload length field.

## Timestamping

The device MUST support hardware timestamping at line rate, on both ingress and egress. Timestamps MUST be taken as defined in IEEE 802.3-2022[ref_id:802.3_2022] clause 90.

### Measurement Plane

The vendor SHOULD measure and report any constant delay between the measurement and reference plane (i.e., network), as defined there. There MUST NOT be any variable length delay between measurement and reference plane exceeding 10ns.

On ingress, this implies that timestamps must be taken before any queueing. On egress, the inverse holds. In particular, measurement of a timestamp must not be subject to (PCIe) backpressure delay on communication of the transmit descriptor to the host.

### Clock

Timestamps are measurements of a device clock. Most device clock components conform to standard requirements for stratum 3 clocks, such as G-1244-CORE[ref_id:gr_1244_core] or ITU-T G.812[ref_id:g812] type IV. The device clock SHOULD conform to one of these and report this.

Before using the common terms in this domain, we first define them:

- Resolution is the quantity below which two samples are seen as equal. It is defined as a time interval (e.g., nsec). The range of values that can be expressed is defined in terms of wrap-around time. From this, a minimum bit-width can be derived. Resolution itself is not an integer storage size, however.
- Precision is the distribution of measurements. It indicates repeatability of measurements, and is affected by read uncertainty. Precision is also expressed as a time interval.
- Accuracy is the offset from the true value. A perfectly precise measurement may have a constant offset. In this context, for instance the offset from the measurement plane from the reference plane.

Clock resolution and precision MUST be 10 ns or better. The clock MUST NOT drift more than 10 ppm. This may require a temperature controlled device (TXCO, OXCO or otherwise), but implementation is not prescribed. The clock must have a wraparound no worse than the 64-bit PTPv1 format, which is 2^32 seconds or roughly 136 years.

The counter MUST be monotonically non-decreasing. That is, causality must be maintained: any packet B measured after another packet A at the same measurement plane cannot have a timestamp lower than the timestamp of A. A packet passing through two measurement planes X and Y (such as PHY Tx and Rx when looping through a switch) must have a timestamp at Y greater than or equal to the timestamp at X. Timestamps may be equal in particular if transmission rate is higher than clock accuracy.

### Clock Synchronization

The device MUST support clock synchronization of host clock to device clock with at most 500 nsec uncertainty. Transmitting an absolute clock reading across a medium such as PCIe itself introduces variable delay that can exceed this bound. The device SHOULD bound this uncertainty, e.g., by implementing a hardware mechanism such as PCI Precision Time Measurement (PTM) [ref_id:pci_ptm]. The vendor MUST report this bound.

The device must expose a clock API to read and control the NIC clock. The device MUST expose at least operations to get absolute value, set absolute value and adjust frequency. These must match the behavior of the `gettimex64`, `adjtime` and `adjfine` or `adjfreq` operations as defined in Linux `ptp_clock_info`. The get value operation MUST be implemented as a sandwich algorithm where the device clock reading is reported in between two host clock reads, as described in the PCI PTM link protocol[ref_id:pci_express_5.0, sec 6.22.2]. The frequency adjustment operation MUST allow frequency adjustments at 1 part per billion resolution or better.

#### *PPS in and out*

The device MUST support both a Pulse Per Second (PPS) input and output signal.

### Host Communication

Timestamps may be passed to the host in a truncated format consisting of only the N least significant bits. This N-bit counter MUST have a wraparound of 1 second or greater. This allows the host to extend timestamps received during this interval to the full resolution by reading the full device clock at this timescale.

#### *Receive*

The device MAY support selective receive timestamping, where the host can install a packet filter to select a subset of packets to be timestamped. The device MUST support the option to timestamp all packets.

For RSC packets the timestamp reported MUST be the timestamp of the first segment. This extends IEEE 802.3 Ethernet timestamp measurement to Receive Segment Coalescing packets.

*Transmit*

Transmit timestamps SHOULD be passed by the device to the host in a transmit completion descriptor field. If the measurement takes place after the completion notification, the device may instead queue a separate second completion, or directly expose an MMIO timestamp register file to the host, if that design can sustain line rate measurement.

The device MAY require the host to explicitly request a timestamp for each packet, e.g., through a descriptor field.

For TSO packets, measurement happens after segmentation. As with all other timestamps, the timestamp MUST be taken for the first symbol in the message. This corresponds to the first segment.

### Applications

NIC hardware timestamping is essential to IEEE 1588 clock synchronization. Applications at hyperscale also include congestion control and distributed applications.

Delay based TCP congestion control takes network RTT as input signal. Measurement must be more precise than network delay, which in data centers can be tens of microseconds. Hyperscale deployment of advanced congestion control requires a significantly higher measurement rate than for PTP clock synchronization, since RTT estimates are per-connection and measurements taken on every packet.

NIC hardware timestamps also enable latency measurement of the NIC datapath itself. Incast is a significant concern in hyperscale environments. Concurrent connection establishment can cause queue build up in a NIC if the host CPU, memory or peripheral bus are out of resources. Latency instrumentation can give an earlier and more informative signal than drops alone.

Finally, distributed systems increasingly rely on high precision clock synchronization to offer strongly consistent scalable storage\ref_id[sundial]. Microsoft FaRMv2\ref_id[farm] and CockroachDB are two examples. Serializability in such databases depends on strict event ordering based on timestamps. Transactions can be committed only after a time uncertainty bound has elapsed. Key to scaling transaction rate is bounding this uncertainty.

## Traffic Shaping

A device MUST implement ingress traffic shaping to mitigate incast impact on high priority traffic. It MAY implement egress traffic shaping to offload this task from the host CPU.

### Ingress

An ingress queue can build up on the device due to incast. If a standing queue can build up in the device, the device SHOULD mitigate head of line blocking of high priority traffic, by prioritizing traffic based on IP DSCP bits. The device MUST offer at least two traffic bands and MUST support host configurable mapping of DSCP bits to band. The device SHOULD offer weighted round robin (WRR) dequeue with weights configurable by the host. It may implement strict priority. If so, this MUST include starvation prevention with a minimum of 10% of bandwidth for every queue.

### Egress

Transmit packet scheduling is discussed in the multi-queue section. The device may additionally offer hardware traffic shaping to offload traffic prioritization from the host CPU. This specification does not ask the device to implement explicit hardware meters, policers or priority queues.

*Earliest Departure Time*

The device MAY support hardware traffic shaping by holding packets until a packet departure time: Earliest Departure Time (EDT)[ref_id:google_carousel]. The feature allows the host to specify a time until which the device must defer packet transmission. The same mechanism is also sometimes referred to as Frame Launch time or SO_TXTIME.

EDT can be implemented efficiently in an O(1) data structure. Device implementation is out of scope for this spec, but one potential design is in the form of a two-layer hierarchical timing wheel[ref_id:varghese_tw].

This feature relies on comparing packet departure time against a device clock. It thus depends on a device hardware clock and host clock synchronization as described in the section on timestamping. It requires a transmit descriptor field to encode the departure time.

If the device supports EDT, then it MUST implement this according to the following rules. It MUST send without delay packets which have no departure time set or for which the departure time is in the past. It MUST NOT send a packet with a departure time before that departure time under any conditions. Departure time resolution MUST be 2us or smaller. The device MUST be able to accept and queue packets with a departure time up to 50 msec in the future. This "time horizon" is based on congestion control algorithms' forward looking window. The device likely also has a global maximum storage capacity. The requirement that departure times up to 50 msec must be programmable DOES NOT imply that the device has to support enough storage space to queue up to 50 msec of data: actual packet spacing may be sparse. It SHOULD NOT have a maximum per interval capacity. The vendor MUST report all such bounds. The device MAY support a special slot for queueing packets with a time beyond the time horizon, or it may choose to drop those. The device MUST expose a counter for all packets dropped by the timing wheel due to either resource exhaustion or departure time beyond the horizon. The device SHOULD signal in a transmit completion when a packet was dropped rather than sent.

# Protocol Support

## Link Layer

### Ethernet

The only required supported link layer is Ethernet.

If the device supports pause frames, it MUST be possible for the host to disable both Tx and Rx support. The device SHOULD support programmatically querying and setting link speed and link up/down state. This extends to multi-host devices: if the device uplink is down, all hosts see link-down, and hosts attached to the device can also not communicate with one another.

### MTU

The device MUST support Ethernet jumbo frames holding up to 9000B of payload (L3MTU, excluding Ethernet header and FCS) with multiple layers of encapsulation. It must support the industry-wide common maximum of 9216B including Ethernet header and FCS trailer (L2MTU) or higher, to allow for significant encapsulation.

*4KB MSS*

The device and driver SHOULD be optimized for 4KB packet payload. Hyperscalers are not bound to the public internet default MTU of 1280 or 1500. 4KB payload is a more efficient transfer size than either 1500B or maximum jumbo size because it aligns with microarchitecture page size on many platforms, which allows for more efficient data movement than copying.

For example, Linux TCP_RECEIVE_ZEROCOPY is an optimization in the receive path that replaces the copying from kernel to userspace in the recv() system call with virtual memory operations ("page flipping"). This optimization is applied only for page-sized, paged-aligned data. That requirement can be satisfied only if TCP bytestream data is written by the NIC in page size data buffers, without co-locating headers in these buffers.

MTU is 4168B when optimizing TCP/IPv6 for 4KB payload: 4096B payload + 40B IPv6 + 20B TCP + 12B TCP options (RFC 7323 timestamp). TCP options are included in MSS calculation [ref_id:rfc_6691], so this gives an MSS of 4108 and MTU of 4168. Higher values must also be supported, to support this optimization together with tunneling and transport mode encryption, both of which are common in hyperscale environments.

#### Address tables

Unicast and multicast address tables MUST support direct matching on at least 64 L2 addresses each.

The device MUST support promiscuous mode reception, where all L2 frames are delivered to the host. It SHOULD support all-multicast reception, where all multicast packets are delivered to the host.

The device does not have to implement loopback mode, where egress traffic is looped in the MAC or PHY to the ingress path.

#### VLAN

VLAN hardware acceleration is not required.

#### MPLS

The device MUST support MPLS label stacks of up to 5 labels. For any actions where parsing up to network or transport layer is required (such as receive hashing, RSS and RSC), the device MUST be able to parse packets with label stacks and reach the same result on the inner L3/L4 packet as for the case where no labels are present.

#### Network Controller Sideband Interface

Devices implementing a Network Controller Sideband Interface (NC-SI) to a Baseboard Management Controller (BMC) MUST support communication between the host and the BMC directly, without hairpinning packets through an external switch, both when the uplink connection is down and up.

### Network Layer

IPv6 and IPv4 are the only network protocols required by this specification.

### IPv6 First

Where a trade-off is to be made between IPv4 and IPv6, IPv6 MUST be the prioritized or only supported network protocol. Features MUST NOT be implemented or tested for IPv4 only, except for features that are only defined for this protocol family (such as the IP header checksum).

IPv4 is still a significant presence, but IPv6 is the standard network protocol for mobile networks and an increasingly large share of the public Internet. Hyperscalers are closed environments and can and do migrate faster than the public Internet.

### Options and Extensions

All IPv4 options and IPv6 extension headers SHOULD be supported. They are only relevant to this specification to the extent that devices must be able to parse through them for header-split and Receive Side Coalescing.

## Transport Layer

TCP and UDP are the only transport protocols required by this specification.

All TCP options MUST be supported. This is analogous to the options and extensions at the network layer.

Tunnel protocols are out of scope for this revision of the specification.

# Telemetry

The device must present the following counters, with the specified counter semantics. Per device counters may be exposed as per queue, and summed on the host. All counters must be 64-bit unsigned.

Many counters are based on Linux v6.3, in particular struct rtnl_link_stat64. The semantics of the fields in that struct are also defined in the source code as comments. The two definitions are identical. Where there is ambiguity, the Linux v6.3 source code is the ground truth. This document specifies additional fields not standardized in Linux as of writing.

*Definition: Byte Counters*

The guiding principle when deciding the increment of a counter is to allow for maximum cross-referencing between counters at different stages of processing. Since most of the pipeline does not carry the FCS, byte counters SHOULD NOT include it even when present. Similarly, since most counters deal with wire packets, stages before segmentation (on egress) or after receive combining (on ingress) SHOULD use wire packets as the unit even when dealing with super-segments.

### Device Counters

*Errors*

All dropped or discarded packets MUST be counted. Each stage in a pipeline that can drop packets SHOULD expose a separate drop counter, preferably a counter per each drop reason.

At a minimum, these packet counters MUST be present

- rx_dropped: aggregate count of packets dropped in the receive pipeline
  - excluding those reported as dropped due to host overrun (rx_missed_errors, below).
- rx_missed_errors: packets dropped due to insufficient buffer space to store the packet
  - Likely due to a slow host that does not keep up with the arrival rate.
  - Incast is a significant concern in hyperscale workloads. This is a critical counter.
- tx_dropped: aggregate count of packets dropped in the transmit pipeline

*RMON*

RFC 2819:

- histogram statistics - packet counters broken down by packet size
    - the [1k, 2k] range SHOULD be broken down into two separate buckets: up to the standard 1518/1522 MTU and above
- etherStatsUndersizePkts: packets received that were less than minimum size

*MAC*

IEEE defined:

- 30.3.4.2 aPAUSEMACCtrlFramesTransmitted
- 30.3.4.3 aPAUSEMACCtrlFramesReceived
- additional statistics measuring time spent in paused state are welcome

*PHY*

IEEE defined:

- 30.3.2.1.5 aSymbolErrorDuringCarrier
- 30.5.1.1.17 aFECCorrectedBlocks
- 30.5.1.1.18 aFECUncorrectableBlocks

Other:

- link_down_events - number of times the link lock was lost or stopped at the PHY level

### Feature-specific Counters

The following counters SHOULD be present if the relevant feature is supported

*Header Split*

- hsplit_packets: number of packets with headers placed in separate header buffer
    - helps verify that the feature is enabled and matches expected ratio of rx_packets
    - alternatively, report the inverse (non_hsplit_packets), as that is expected to be the exception

*Checksum Offload*

- rx_csum_err: packets with checksum computed and compared to header field, not matching
    - only for legacy devices that verify a checksum rather than return the linear sum
- rx_csum_none: packets that were not checksummed
    - same

*Segmentation Offload*

- lso_packets: segmentation offload packets, including TSO, UDO and PISO.

*Receive Segment Coalescing*

- receive_offload_packets: number of coalesced SO packets created by RSC

# Performance

The device must demonstrate to meet its advertised performance targets in the intended operating environment. The vendor MUST document a reproducible test setup that demonstrates to meet all performance requirements. Appendix B lists suggestions for specific performance testing on Linux. Those are not prescriptive.

Targets must be met with a relatively standard off-the-shelf server that is representative of the intended target environment. For a 100 Gbps configuration, it is suggested to have a single CPU socket with at least 16 CPU cores and a network configuration with 16 receive and transmit queues and RSS load-balancing.

Performance targets cover bitrate (bps), packet rate (pps) and NIC pipeline latency (nsec).

### Bitrate

Bitrate is the metric by which a device is often advertised, e.g., a 100 Gbps NIC.

*Variants*

All following variants SHOULD reach the advertised line rate with

- TSO on and off
  - if PISO is supported, across a UDP tunnel
- RSC on and off
- IOMMU on and off
- 1500B, 4168B and 9198B L3MTU
- Unidirectional and bidirectional traffic
- Scalability
  - 10, 100, 1K, 10K flows
  - 1, 10, NUM_CPU threads
  - 1, 10, NUM_CPU queues

*Single Flow*

Single flow MUST reach 40 Gbps with 1500B MTU and TSO. A single TCP/IP flow can reach 100 Gbps line rate when using TSO, 4KB MSS and copy avoidance\ref_id[tcp_rx_0copy], but this is a less common setup. Single flow line rate is not a hard requirement, especially as device speeds exceed 100 Gbps.

*Peak, Stress and Endurance Results*

Short test runs can show best case numbers. Deployment requires sustained performance.

Endurance tests can expose memory leaks and rare unrecoverable edge cases, e.g., those that result in device or queue timeout. Endurance tests essentially run the same testsuite over longer periods of time. Reported numbers for 1 hour runs MUST stay constant and match short term numbers.

Stress tests test specific adverse conditions. They need not be as long as endurance tests. Performance during adverse conditions may be lower than best case, but not catastrophically so. Device and driver are expected to handle overload gracefully. They MUST be resistant to Denial of Service (DoS) and incast. If max packet rate for minimal packets is less than line rate, it SHOULD be constant regardless of packet arrival rate.

*Bus Contention*

Network traffic competes with other tasks for PCIe and memory bandwidth. Some micro-architectural considerations, such as NUMA or cache sizes and partitioning, cannot be controlled. But devices can be compared to the extent that they stress the PCIe or memory bus for the same traffic: how many PCIe messages are required to transfer the same number of packets of a given size is an indicator for real world throughput under bus contention.

This efficiency is evaluated by repeating the testsuite while running a memory antagonist. An effective memory antagonist on Unix environments is a pinned dd binary copying in-memory virtual files. A device SHOULD minimize the number of PCIe messages needed (see the section on PCIe Cache Aligned Stores) to reduce sensitivity to concurrent workloads.

### Packet Rate

The vendor MUST report a maximum packet rate, and MUST demonstrate that the device reaches this rate.

*Scalability: Queue Count*

The vendor MUST report maximum packet rate BOTH with a chosen optimal configuration and with a single pair of receive and transmit queues.

The performance metrics should remain reasonably constant with queue count: packet rate at any number of queues 8 or higher SHOULD be no worse than 80% of the best case packet rate.

If this cannot be met, the vendor MUST also report the worst case queue configuration and its packet rate. This to avoid surprises as the user deploys the device and tunes configuration.

### Connection Count and Rate

Most NIC features operate below the transport layer. Where features do interact with the transport layer, the NIC has to demonstrate to be able to reach observed datacenter server workloads.

The NIC MUST scale to 10M open TCP/IP connections and 100K connection establishments + terminations (each) per second. It MUST be able to achieve this with no more than 100 CPU cores. This limit is not overly aggressive: it was chosen with significant room above what is observed in production.

### Latency

Rx and Tx pipeline latency for standard Ethernet packets SHOULD NOT exceed 2 usec each, MUST NOT exceed 4 usec at 90% and MUST NOT exceed 20 us at 99%, as measured under optimal conditions with no competing workload generating bus contention. If one of these bounds cannot be met, then this MUST be reported. This requirement must be met for a standard device configuration. That is, checksum offload and RSS must be enabled. Interrupt delay is not included, so interrupt moderation may be disabled or interrupts disabled in favor of polling. This measurement is for packets that do not exercise TSO/USO/PISO or RSC.

Tx pipeline latency is the time from when the host signals to the device that work is pending, to packet transmission as defined in the timestamping section. Rx pipeline latency is reception as defined in the timestamping section until the descriptor is first readable by the host. In practice, measurement compares hardware PHY timestamps to best case conditions for host software timestamp measurement, which always overestimates to a degree. The chosen bounds are experimentally arrived at in the same manner and take this measurement error into account. See Appendix B for more details on testing.

# Appendix A: Checklist

This section concisely summarizes the requirements for conformance to this specification. No new requirements are introduced.

Devices can conform at one of two levels, basic and advanced. A device can only conform at the advanced level if it also conforms to all requirements of the base level.

Numerical requirements can include a value or range [minimum, maximum] that must be supported.

Devices can conform to the spec with exceptions. If so, the device must list "conforms to level N, except for features A, B and C", with a clear description how the listed features diverge from the behavior specified in this spec.

| Domain | Feature | Required | Value |
|---|---|---|---|
| Queues | | | |
| | Queue Length | basic | [512, 4K] |
| | Num Queues | basic | [1, 1K] |
| | Separate Post + Completion Queues | optional | |
| | Scatter-gather I/O | basic | >= 17 |
| | Header-Split | advanced | |
| | Fixed Offset Split (unless Header-split is supported) | basic | |
| | Reconfiguration without link down | optional | |
| | MMIO Transmit Mode | optional | |
| Multi Queue | | | |
| | Independent Rx and Tx Queue Lengths | advanced | |
| | Emergency Reserve Queue | optional | |
| Interrupts | | | |
| | Num IRQs | basic | == Num Queues |

| | | | |
|---|---|---|---|
| | Moderation: Delay | basic | [2, 200] us |
| | Moderation: Event Count | optional | [2, 128] |
| | Moderation: Separate Tx and Rx | basic | |
| Flow Steering | | | |
| | RSS: Toeplitz Hash | basic | |
| | RSS: include flow label in the hash (configurable) | basic | |
| | RSS: Indirection Table - entries | basic | == Num Queues (combined space; 256 per context) |
| | RSS: Indirection Table | basic | == Num Queues |
| | RSS: Communicate Hash to Host | basic | |
| | ARFS: Table Size | optional | |
| | PFS: Num Rules | optional | >= 64 |
| | PFS: Num RSS Contexts | optional | |
| | RSS: Indirection Table: RSS contexts > 0 | optional | 64 |
| | Transmit Scheduling: DRR: Equal Weight | basic | |
| | Transmit Scheduling: DRR: Weighted | optional | |
| Checksum | | | |
| | Tx Checksum | basic | |
| | Tx Checksum: Zero Checksum Conversion | basic | |
| | Tx Checksum: Protocol Independent | advanced | |
| | Rx Checksum | basic | |
| | Rx Checksum: Protocol Independent | advanced | |
| Segmentation | | | |
| | TCP TSO for IPv4 and IPv6 | basic | |

| | | | |
|---|---|---|---|
| | TCP TSO with transport layer security (IPSec ESP, PSP) | advanced | |
| | TSO IPv6 jumbogram support | advanced | >= 256KB |
| | UDP USO for IPv4 and IPv6 | advanced | |
| | PISO | advanced | |
| Coalescing | | | |
| | RSC support | advanced | |
| | RSC with transport layer security (IPSec ESP, PSP) | advanced | |
| | RSC Num Contexts | advanced | 64 |
| | RSC Context Timeout | advanced | [2, 50] us |
| | Jumbogram RSC | optional | |
| Time | | | |
| | HW Rx timestamping | advanced | line rate |
| | HW Tx timestamping | advanced | line rate |
| | PTP clock device | advanced | |
| | PPS output | advanced | |
| | PPS input | advanced | |
| Traffic Shaping | | | |
| | Ingress Traffic Shaping, number of queues | advanced | >= 2 |
| | Egress Earliest Departure Time (EDT) Shaping | advanced | |
| Link Layer | | | |
| | Unicast Address Table Size | basic | >= 64 |
| | Multicast Address Table Size | basic | >= 64 |
| | MTU | basic | >= 9216 |
| Telemetry | | | |

| | | | |
|---|---|---|---|
| | Device Counters | basic | |
| | Feature Counters | basic | |
| Performance | | | |
| | BPS: 100 flows | basic | bidir line rate |
| | BPS: single flow | basic | 40 Gbps |
| | PPS: max | basic | line rate @ 128B |
| | PPS: single queue | basic | max pps / 8 |
| | CPS: 10M concurrent open TCP/IP connections | basic | with <= 100 CPUs |
| | CPS: 100K TCP/IP opens + closes | basic | with <= 100 CPUs |
| | NIC Rx Latency: 99% | basic | 4 us |
| | NIC Tx Latency: 99% | basic | 4 us |

# Appendix B: Validation

This section lists an optional testsuite that may help demonstrate conformance to the specification. The testsuite covers many, but not all, features. Contributions to extend and improve the testsuite are strongly welcomed.

The specification is operating system independent, but the software tests are not. All use Linux. All tools are open source and freely available.

## Configuration

The following features can be configured on Linux using ethtool. The following setters should succeed and the getters should confirm the setting.

Reading back configuration settings does not verify that the behavior matches the stated configuration. This is a necessary, but not sufficient test. The subsequent section will present functional behavior tests.

### *Queue Length*

The device must support queue lengths between 512 and 4096 slots.

```
    For VAL in 512 1024 4096:
        ethtool -G eth0 rx $VAL tx $VAL
        ethtool -g eth0
```

It should allow reconfiguration without bringing the link down. Link changes can be counted with

```
    cat /sys/class/net/eth0/carrier_up_count
```

### *Queue Count*

The device must support up to 1K queues. It must support independent configuration of receive and transmit queue count.

```
    For VAL in 1 128 1024:
        ethtool -L eth0 rx $VAL tx $VAL
        ethtool -l eth0

        ethtool -L eth0 rx 1 tx $VAL
        ethtool -l eth0

        ethtool -L eth0 rx $VAL tx 1
        ethtool -l eth0
```

*Interrupt Moderation*

The device must support [2, 200] usec delay. It should support [2, 128] events batched.

```
    For VAL in 1 2 20 32 200:
        ethtool -C eth0 rx-usecs $VAL
        ethtool -c
        ethtool -C eth0 tx-usecs $VAL
        ethtool -c

    For VAL in 1 2 20 24 128:
        ethtool -C eth0 rx-frames $VAL
        ethtool -c
        ethtool -C eth0 tx-frames $VAL
        ethtool -c
```

*Receive Side Scaling*

The device must support an indirection table with 1K slots. It may support non-equal weight load balancing

```
    ethtool -L eth0 rx 1024
    ethtool -X eth0 equal 1024
    ethtool -x eth0
    ethtool -X eth0 equal 2
    ethtool -x eth0
    ethtool -X eth0 weight 2 1
    ethtool -x eth0
```

It must allow reconfiguration without bringing the link down. Link changes can be counted with

```
    cat /sys/class/net/eth0/carrier_up_count
```

*Programmable Flow Steering*

Programmable flow steering is an optional feature. But if supported on Linux it must implement this standard interface

```
ethtool -X eth0 equal 1          # steer all non-matching traffic to queue 0
ethtool -K eth0 ntuple on
ethtool -N eth0 flow-type tcp6 src-port 8000 action 1
ethtool -N eth0 flow-type tcp4 src-port 8000 action 2
```

*Offloads*

The device must advertise the following features and they must be configurable:

```
For VAL in off on off:
        ethtool -K eth0 rx $VAL          # rx checksum offload
        ethtool -K eth0 tx $VAL          # tx checksum offload
        ethtool -K eth0 tso $VAL
        ethtool -K eth0 tx-udp-segmentation $VAL
```

The device should advertise these offloads as well:

```
For VAL in off on off:
        ethtool -K eth0 rxhash $VAL
        ethtool -K eth0 rx-gro-hw $VAL
```

The device may advertise flow steering (including ARFS) as well:

```
For VAL in off on off:
        ethtool -K eth0 ntuple $VAL
```

In all cases the configuration must be tested to be successfully reflected in ethtool -k.

*Jumbogram Segmentation Offload*

In Linux, jumbogram segmentation offload is enabled by explicitly configuring a maximum size that exceeds 64KB:

```
ip link set dev eth0 gso_max_size 262144
```

*Link Layer*

The device must support L2MTU up to 9216, which is L3MTU of 9198. It should optimize for 4KB payload: 4096 + 40B IPv6 + 20B TCP + 12B TCP options (RFC 7323 timestamp). TCP options are included in MSS calculation [ref_id:rfc_6691], so this gives an MSS of 4108 and L3MTU of 4168.

```
    For VAL in 1500 4168 9198:
        ip link set dev eth0 mtu $VAL
```

The device must support 64 concurrent unicast and multicast MAC filters:

```
    For BYTE in {1..64}:
        ip link add link eth0 dev eth0.$BYTE address 22:22:22:22:22:$BYTE type macvlan
```

The device must support promiscuous (all addresses) and allmulti (all multicast addresses) modes:

```
    For VAL in off on off:
        ip link set dev eth0 promisc $VAL
        ip link set dev eth0 allmulticast $VAL
```

## Functional

Some features are covered by open source functional tests. This testsuite is partial. We strongly encourage organizations to open source their test suites and suggest these for inclusion here.

### *Receive Header-Split*

Header-split is not a user-facing feature. But, we can verify its implementation by testing a feature that is dependent on HS. Linux TCP_RECEIVE_ZEROCOPY, as described in the MTU section, depends on header-split to store payload page-aligned in page-sized buffers. TCP_ZEROCOPY_RECEIVE further requires the administrator to choose MTU such that MSS is 4KB plus room for expected TCP options, and by posting 4KB pages as buffers to the device receive queue.

Linux v6.3 **tools/testing/selftests/net/tcp_mmap** tests this feature. Limitations are that the test does not differentiate between header-split and fixed-prefix split, and does not cover all possible TCP options.

The **hsplit_packets** device counter must also match the expected packet rate of the test.

*Toeplitz Receive Hash*

Linux v6.3 **tools/testing/selftests/net/toeplitz.sh** verifies the Toeplitz hash implementation by receiving whole packets up to userspace (using PF_PACKET sockets), along with the hash as returned by the device. The test manually recomputes the hash in software and compares the two.

Microsoft RSS documentation~[ref_id:ms_rss] lists a set of example inputs with expected output hash values. The software implementation in Linux kernel tools/testing/selftests/toeplitz.c and the example code in this spec have been verified to pass that test. A vendor may test these exact packets, or test arbitrary packets against the toeplitz.c software implementation.

*Receive Side Scaling*

Linux v6.3 **tools/testing/selftests/net/toeplitz.sh** also verifies RSS queue selection if argument -rss is passed. In this mode it uses Linux PACKET_FANOUT_CPU to detect the CPU on which packets arrive. Beyond the Toeplitz algorithm, this test also verifies the modulo operation that the device must use to convert the 32-bit Toeplitz hash value into a queue number.

*Checksum Offload*

Linux v6.3 **tools/testing/selftests/net/csum.c** verifies receive and transmit checksum offload. On receive, it tests both correct and corrupted checksums. On transmit, it tests checksum offload, UDP zero checksum conversion, checksum disabled and transport mode encapsulation. The test covers IPv4 and IPv6, TCP and UDP. The process is started on two machines. See comments in the source file header for invocation details.

*TCP Segmentation Offload*

**github.com/wdebruij/kerneltools/blob/master/tests/tso.c** can deterministically craft TSO packets of specific size and content. A full testsuite that uses this to test all interesting cases is a work in progress.

Common case TSO packets can be tested by generating a TCP stream and measuring byte- and packet counts.

This requires care: packet count as reported by the device must report counts after segmentation, so cannot be used to report the number of TSO packets. On Linux, packets observed with tcpdump or similar tools are observed before TSO, but also before possible software segmentation, so cannot be trusted. The correct counter is the device **lso_counter**. Additionally it is possible to count the number of ndo_start_xmit invocations using ftrace.

*UDP Segmentation Offload*

Linux v6.3 **tools/testing/selftests/net/udpgso_bench_[rt]x.c** implements both a sender and receiver side for a two-machine test. It can send UDP traffic with and without USO and thus can be used to exercise the USO hardware block.

Tests can be performed with various optional features at multiple segmentation sizes. We recommend testing boundary conditions along with common MSS (1500B, 4K, jumbo). The accompanying **udpgso.sh** and **udpgso_bench.sh** scripts list the relevant boundary test cases (but exercise the software implementation themselves).

*Receive Segment Coalescing*

Linux v6.3 **tools/testing/selftests/net/gro.c** implements both a sender and receiver side for a two-machine test. The sender sends a train of packets in quick succession, the receiver verifies that they are coalesced or not, in accordance with the RSC rules. Test cases cover IP (e.g., ToS), TCP (e.g., sequence number) and more. RSC tests are inherently timing dependent. We recommend testing with the maximum RSC context timeout available.

*Earliest Departure Time*

Linux v6.3 **tools/testing/selftests/net/so_txtime.c** implements both a sender and receiver side for a transmit test. The sender transmits UDP packets with an earliest delivery time configured with the SO_TXTIME socket option. The receiver validates arrival rate against expectations.

Testing delivery time requires tight clock synchronization. The test can be run on a single machine. Either with two devices, or with a single device by configuring two IPVLAN virtual devices in separate network namespaces. Aside from exercising physical hardware, the setup is similar to that in the accompanying **so_txtime.sh**, which only differs by using entirely virtual devices to test software pacing.

Testing delivery time requires packets arriving with EDT timestamp unmodified at the driver. This requires EDT support from the Linux queueing discipline (qdisc). **so_txtime.sh** demonstrates software EDT with the FQ and ETF qdiscs. ETF also supports hardware offload in Linux v6.3. At the time of writing hardware offload for FQ is in development.

*Timestamping*

Linux v6.3 **tools/testing/selftests/net** contains multiple tests that exercise the control and datapath Linux timestamping APIs, SIOC[GS]HWTSTAMP and SO_TIMESTAMPING. But it currently lacks a true regression test for hardware timestamps. This is a gap in this test suite that still needs to be completed.
**github.com/wdebruij/kerneltools/blob/master/tests/tstamp.c** requests and parses software and hardware, receive and transmit, IPv4 and IPv6, TCP and UDP timestamps. It needs to be run for all these cases.

*Correctness*

Failure to maintain causality can be detected, both for multiple packets at the same measurement point, and for a single packet at multiple measurement points. Clock drift can be measured. Open source conformance tests for these invariants are also future work.

*Telemetry*

Packet and byte counters can be observed with `ip -s -s link show dev $DEV`. At a minimum, these counters must be verified to match the actual data transmission. One way is to compare this data with data obtained with ftrace instrumentation on ndo_start_xmit.

## Performance

### Benchmark Suite

Hardware limits are demonstrated at the transport layer where possible, as this is the highest application independent layer.

This TCP/IP transport layer testsuite uses neper [ref_id:neper] as the benchmark tool. Neper is similar to other transport layer benchmarks tools, such as netperf and iperf. It differentiates itself by having native support for scaling threads and flows, aggregate statistics reporting including median and tail numbers, and epoll support for scalable socket processing. Neper supports IPv4 and IPv6, TCP and UDP and streaming ("tcp_stream"), echo request/response ("tcp_rr") and connection establishment ("tcp_crr") style workloads.

### Reproducible Results

Presented results must be the median of at least 5 runs, with interquartile range (Q3-Q1) less than 10% of the median (Q2). If the interquartile range exceeds this number, it must be listed explicitly. Especially single flow and latency tests may be noisy.

Many factors can add noise on modern systems. Evaluators are encouraged to apply the following system settings to minimize variance:

- Disable CPU sleep states (C-states), frequency scaling (P-states) and turbo modes.
- Disable hyperthreading
- Disable IOMMU
- Pin process threads
- Memory distance: pin threads and IRQ handlers to the same NUMA node or cache partition
  - Select the NUMA node to which the NIC is connected
- Move other workloads away from selected cores (e.g., using isolcpus on Linux).
- Disable adaptive interrupt moderation and software load balancing (e.g., Linux RPS/RFS).

### Isolating Hardware and Software

Transport and application layer tests exercise both software on the host and the device hardware. One strategy to help disambiguate bottlenecks between the two environments is to run the host at multiple (fixed) clock rates. For latency tests, this will result in measurements

RTT1 = cpu_cycles * freq1 + hw_pipeline_latency
RTT2 = cpu_cycles * freq2 + hw_pipeline_latency
RTT3 = cpu_cycles * freq3 + hw_pipeline_latency
etc.

Solving this system of equations generates an estimate of hardware pipeline latency.

## Metrics

### Bitrate

The performance section suggests a standard configuration to demonstrate reaching advertised line rate. This configuration expressed as a neper command is

```
tcp_stream -6 -T 10 -F 10 -B 65536 [-r -w] [-c -H $SERVER]
```

Transport layer metrics will report goodput without protocol header overhead. A 100 Gbps device should report approximately 94 Gbps of goodput.

### Transaction Rate

A request/response workload on an otherwise idle system can both test latency at the application level (with a single flow), and small packet throughput (when run with parallel flows and threads).

A single flow avoids all contention and queue build-up. But idle systems may enter low power modes from which wake-up adds latency. Report both single flow and a setup that gives maximum throughput, for instance.

```
tcp_rr [-c -H $SERVER]
tcp_rr -T $NUM_CPU -F 10000 [-c -H $SERVER]
```

Neper reports 50/90/99% application-level latency.

### Packet Rate

Packet rates at possibly hundreds of Mpps expose software bottlenecks. It is unlikely that advertised minimum packet line rate (PPS) can be demonstrated with transport layer benchmarks like tcp_rr.

A userspace network stack optimized for packet processing, such as DPDK, is reasonable to stress this hardware limit.

A pure Linux solution for packet processing can be built using eXpress Data Path (XDP). Packets must be generated on the host as close to the device as possible. A device that supports AF_XDP, in native driver mode, with copy avoidance and busy polling, has been shown to reach 30 Mpps on a 40 Gbps NIC using the rx_drop benchmark that ships with the Linux kernel. Over 100 Mpps has been demonstrated on 100 Gbps NICs, but these results are not publicly published.

### Connection Rate

Neper tcp_crr ("connect-request-response") can demonstrate connection establishment and termination rate. The expressed target is 100K TCP/IP connections per second, with no more than 100 CPU cores. tcp_crr is invoked similar to tcp_rr, but created a separate connection for each request. Demonstrate with the boundary number of CPUs or fewer. Ideal

```
tcp_crr -T $NUM_CPU -F $NUM_FLOWS [-c -H $SERVER]
```

### Connection Count

There is no current test to demonstrate reaching 10M concurrent connection count

### Latency

Pipeline latency can be measured at the link layer with an echo request/response ("ping") workload. An upper bound of pipeline latency can be established by measuring RTT between two devices that are directly connected (i.e., without an intermediate switch). In a two-machine test, full RTT is recorded to avoid requiring clocks synchronized at sub microsecond precision. A single machine setup with two devices (or ports on a single device) can report half-RTT. The reported value is an upper bound. 50/90/99 percentiles should be reported.

Low event rate tests can be sensitive to cache and power scaling effects, as discussed in the section on reproducible results. This test alone cannot differentiate Rx from Tx latency.

Hardware timestamps can be used to compute the time spent in the cable (and switch or wider fabric), to reduce the overestimation in the L2 ping test, and to disambiguate Rx from Tx latency.

# Appendix C: Revision History

| Version | Date | Comment |
|---|---|---|
| 0.9 | 2023-05-16 | Initial public draft |
| 0.9.1 | 2023-07-13 | Pre-publication: incorporated public feedback |
| 1.0 | 2023-08-09 | OCP approved v1.0 |

# References

[ref_id:fq_drr] M. Shreedhar and G. Varghese, "Efficient fair queuing using deficit round-robin," in IEEE/ACM Transactions on Networking, vol. 4, no. 3, pp. 375-385, June 1996, doi: 10.1109/90.502236.
[ref_id:csum] "Checksum Offloads", https://www.kernel.org/doc/html/latest/networking/checksum-offloads.html, retrieved on 2022-11-07.
[ref_id:neper] "Neper, a Linux networking performance tool", https://github.com/google/neper, retrieved on 2022-12-30.
[ref_id:ocp_nic] , "OCP NIC 3.0 Design Specification version 1.00", https://www.opencompute.org/documents/ocp-nic-3-0-r1v00-20191219a-tn-no-cb-pdf, retrieved on 2023-1-24.
[ref_id:msft_rsc], "RSS Hashing Types", https://learn.microsoft.com/en-us/windows-hardware/drivers/network/rss-hashing-types, retrieved on 2023-04-8.
[ref_id:pci_ptm], "Precision Time Measurement (PTM), Revision 1.0a", https://pcisig.com/specifications/pciexpress/specifications/ECN_PTM_Revision1a_31_Mar_2013.pdf, retrieved on 2023-1-26.
[ref_id:802.3-2022], "IEEE Standard for Ethernet", https://standards.ieee.org/ieee/802.3/10422, retrieved on 2023-1-31.
[ref_id:gr_1244_core], "Clocks for the Synchronized Network: Common Generic Criteria", https://telecom-info.njdepot.ericsson.net/site-cgi/ido/docs.cgi?ID=SEARCH&DOCUMENT=GR-1244, retrieved on 2023-2-17.
[ref_id:itu-t_g.812], "G.812 : Timing requirements of slave clocks suitable for use as node clocks in synchronization networks", https://www.itu.int/rec/T-REC-G.812, retrieved on 2023-2-23.
[ref_id:farm], "Fast General Distributed Transactions with Opacity", SOSP 2015
[ref_id:sundial], "Sundial: Fault-tolerant Clock Synchronization for Datacenters", OSDI 2020
[ref_id:tcp_rx_0copy], "PATH to TCP 4K MTU and RX zerocopy", Netdev 0x14, August 2020, https://legacy.netdevconf.info/0x14/session.html?talk-the-path-to-tcp-4k-mtu-and-rx-zerocopy
[ref_id:toeplitz_hash], "New Hash Functions for Message Authentication", H. Krawczyk, EUROCRYPT ’95, Springer Verlag LNCS volume 921.
[ref_id:ms_rss], "Verifying the RSS Hash Calculation", https://learn.microsoft.com/en-us/windows-hardware/drivers/network/verifying-the-rss-hash-calculation, retrieved on 2023-3-20/