In-band Network Telemetry (INT) Dataplane Specification

Working draft. Note: consider using tagged versions for implementation.

The P4.org Applications Working Group. Contributions from
Alibaba, Arista, Barefoot Networks, Dell, Intel, Marvell, Netronome, VMware

2018-08-17

Contents

1. Introduction 2
2. Terminology 3
3. What To Monitor 4
3.1. Switch-level Information . . . 4
3.2. Ingress Information . . . 4
3.3. Egress Information . . . 5
4. INT Headers 6
4.1. INT Header Types . . . 6
4.2. Per-Hop Header Creation . . . 6
4.3. MTU Settings . . . 6
4.4. INT over any encapsulation . . . 7
4.5. Checksum Update . . . 8
4.6. Header Location . . . 9
4.6.1. INT over TCP/UDP . . . 9
4.6.2. INT over VXLAN GPE . . . 11
4.6.3. INT over Geneve . . . 12
4.7. INT Hop-by-Hop Metadata Header Format . . . 13
5. Examples 16
5.1. Example with INT over TCP . . . 17
5.2. Example with INT over VXLAN GPE . . . 18
5.3. Example with INT over Geneve . . . 19
6. P4 program specification for INT Transit 20
A. Appendix: An extensive (but not exhaustive) set of Metadata 38
A.1. Switch-level . . . 38
A.2. Ingress . . . 38
A.3. Egress . . . 39
A.4. Buffer Information . . . 39
A.5. Miscellaneous . . . 40
B. Acknowledgements 40
C. Change log 40

1. Introduction

Inband Network Telemetry ("INT") is a framework designed to allow the collection and reporting of network state, by the data plane, without requiring intervention or work by the control plane. In the INT architectural model, packets contain header fields that are interpreted as "telemetry instructions" by network devices. These instructions tell an INT-capable device what state to collect and write into the packet as it traverses the network. INT traffic sources (applications, end-host networking stacks, hypervisors, NICs, send-side ToRs, etc.) can embed the instructions either in normal data packets or in special probe packets. Similarly, INT traffic sinks retrieve (and optionally report) the collected results of these instructions, allowing the traffic sinks to monitor the exact data plane state that the packets "observed" while being forwarded.

Some examples of traffic sink behavior are described below:

- OAM – the traffic sink might simply collect the encoded network state, then export that state

to an external controller. This export could be in a raw format, or could be combined with basic processing (such as compression, deduplication, truncation).

- Real-time control or feedback loops – traffic sinks might use the encoded data plane information to feed back control information to traffic sources, which could in turn use this information to make changes to traffic engineering or packet forwarding. (Explicit congestion notification schemes are an example of these types of feedback loops).
- Network Event Detection - If the collected path state indicates a condition that requires immediate attention or resolution (such as severe congestion or violation of certain data-plane invariances), the traffic sinks could generate immediate actions to respond to the network events, forming a feedback control loop either in a centralized or a fully decentralized fashion (a la TCP).

The INT architectural model is intended to be generic and enables a number of interesting high level applications, such as:

- Network troubleshooting
  - Traceroute, micro-burst detection, packet history (a.k.a. postcards)
- Advanced congestion control
- Advanced routing
  - Utilization-aware routing (For example, HULA[1], CLOVE[2])
- Network data plane verification

A number of use case descriptions and evaluations are described in the Millions of Little Minions paper [3].

2. Terminology

- **INT Header**: A packet header that carries INT information. There are two types of INT Headers – *Hop-by-hop* and *Destination* (See Section 4.1).
- **INT Packet**: A packet containing an INT Header.
- **INT Instruction**: Instructions embedded in INT header, indicating which INT Metadata (defined below) to collect at each INT switch. The collected data is written into the INT Header.
- **INT Source**: A trusted entity that creates and inserts INT Headers into the packets it sends.
- **INT Sink**: A trusted entity that extracts the INT Headers and collects the path state contained in the INT Headers. The INT Sink is responsible for removing INT Headers so as to make INT transparent to upper layers. (Note that this does not preclude having nested or hierarchical INT domains.)

[1] HULA: Scalable Load Balancing Using Programmable Data Planes, ACM SOSR 2016
[2] CLOVE: Congestion-Aware Load Balancing at the Virtual Edge, ACM CoNEXT 2017
[3] Millions of Little Minions: Using Packets for Low Latency Network Programming and Visibility, ACM SIGCOMM 2014.

- **INT Transit Hop**: A networking device that adds its own INT Metadata to an INT Packet by following the INT Instructions in the INT Header.
- **INT Metadata**: Information that an INT Source or an INT Transit Hop device inserts into the INT Header. Examples of metadata are described in section 3.
- **INT Domain**: A set of inter-connected INT devices under the same administration. This specification defines the behavior and packet header formats for interoperability between INT switches from different vendors in an INT domain. The INT devices within the same domain must be configured in a consistent way to ensure interoperability between the devices. Operators of an INT domain should deploy INT Sink capability at domain edges to prevent INT information from leaking out of the domain.

3. What To Monitor

In theory, one may be able to define and collect any switch-internal information using the INT approach. In practice, however, it seems useful to define a small baseline set of metadata that can be made available on a wide variety of devices: the metadata listed in this section comprises such a set. As the INT specification evolves, we expect to add more metadata to this INT specification.

The exact meaning of the following metadata (e.g., the unit of timestamp values, the precise definition of hop latency or queue occupancy) can vary from one device to another for any number of reasons, including the heterogeneity of device architecture, feature sets, resource limits, etc. Thus, defining the exact meaning of each metadata is beyond the scope of this document. Instead we assume that the semantics of metadata for each device model used in a deployment is communicated with the entities interpreting/analyzing the reported data in an out-of-band fashion.

3.1. Switch-level Information

- Switch id
  - The unique ID of a switch (generally administratively assigned). Switch IDs must be unique within an INT domain.

3.2. Ingress Information

- Ingress port identifier
  - The port on which the INT packet was received. A packet may be received on an arbitrary stack of port constructs starting with a physical port. For example, a packet may be received on a physical port that belongs to a link aggregation port group, which in turn is part of a Layer 3 Switched Virtual Interface, and at Layer 3 the packet may be received in a tunnel. Although the entire port stack may be monitored in theory, this specification allows for monitoring of up to two levels of ingress port identifiers. First level of ingress port identifier would typically be used to monitor the physical port on which the packet was received, hence a 16-bit field (half of a 4-Byte metadata) is deemed adequate. Second level of ingress port identifier occupies a full 4-Byte metadata field, which may be used to monitor a logical port on which the packet was received. A 32-bit space at the second level allows for an adequately large number of logical ports at each

network element. The semantics of port identifiers may differ across devices, each INT hop chooses the port type it reports at each of the two levels.

- Ingress timestamp
  - The device local time when the INT packet was received on the ingress physical or logical port.

3.3. Egress Information

- Egress port identifier
  - The port on which the INT packet was sent out. A packet may be transmitted on an arbitrary stack of port constructs ending at a physical port. For example, a packet may be transmitted on a tunnel, out of a Layer 3 Switched Virtual Interface, on a Link Aggregation Group, out of a particular physical port belonging to the Link Aggregation Group. Although the entire port stack may be monitored in theory, this specification allows for monitoring of up to two levels of egress port identifiers. First level of egress port identifier would typically be used to monitor the physical port on which the packet was transmitted, hence a 16-bit field (half of a 4-Byte metadata) is deemed adequate. Second level of egress port identifier occupies a full 4-Byte metadata field, which may be used to monitor a logical port on which the packet was transmitted. A 32-bit space at the second level allows for an adequately large number of logical ports at each network element. The semantics of port identifiers may differ across devices, each INT hop chooses the port type it reports at each of the two levels.
- Egress timestamp
  - The device local time when the INT packet was processed by the egress physical or logical port.
- Hop latency
  - Time taken for the INT packet to be switched within the device.
- Egress port TX Link utilization
  - Current utilization of the egress port via which the INT packet was sent out. Again, devices can use different mechanisms to keep track of the current rate, such as bin bucketing or moving average. While the latter is clearly superior to the former, the INT framework does not stipulate the mechanics and simply leaves those decisions to device vendors.
- Queue occupancy
  - The build-up of traffic in the queue (in bytes, cells, or packets) that the INT packet observes in the device while being forwarded.

4. INT Headers

This section specifies the format and location of INT Headers.

4.1. INT Header Types

There are two types of INT Headers, *hop-by-hop* and *destination*. A given INT packet may have one or both types of INT Headers. When both INT Header types are present, the hop-by-hop type must precede the destination type header.

- Hop-by-Hop type (**INT Header type 1**)
  - Intermediate devices (INT Transit Hops) must process this type of INT Header. The format of this header is defined in section 4.7
- Destination type (**INT Header type 2**)
  - Destination headers must only be consumed by the INT Sink. Intermediate devices must ignore Destination headers.
  - Destination headers can be used to enable communication between the INT Source and INT Sink. For example:
    * INT Source can add a sequence number to detect loss of INT packets.
    * INT Source can add the original values of IP TTL and INT Remaining Hop Count, thus enabling the INT sink to detect network devices on the path that do not support INT by comparing the IP TTL decrement against INT Remaining Hop Count decrement (assuming each network device is an L3 hop)
  - The data format of Destination type INT Headers is not defined in this version.

4.2. Per-Hop Header Creation

In the INT model, each device in the packet forwarding path creates additional space in the INT Header on-demand to add its own INT metadata. To avoid exhausting header space in the case of a forwarding loop or any other anomalies, it is strongly recommended to limit the number of total INT metadata fields added by devices by setting the Remaining Hop Count field in INT header appropriately.

4.3. MTU Settings

As each hop creates additional space in the INT header to add its metadata, the packet size increases. This can potentially cause egress link MTU to be exceeded at an INT switch.

This may be addressed in the following ways -

- It is recommended that the MTU of links between INT sources and sinks be configured to a value higher than the MTU of preceding links (server/VM NIC MTUs) by an appropriate amount. Configuring an MTU differential of [Per-hop Metadata Length*4*INT Hop Count + Fixed INT Header Length] bytes, based on conservative values of total number of INT hops and Per-hop Metadata Length, will prevent egress MTU being exceeded due to INT metadata insertion at INT hops. The Fixed INT Header Length is the sum of INT metadata

header length (8B) and the size of encapsulation-specific shim/option header (4B) as defined in section 4.6.

- An INT source/transit switch may optionally participate in dynamic discovery of Path MTU for flows being monitored by INT by transmitting ICMP message to the traffic source as per Path MTU Discovery mechanisms of the corresponding L3 protocol (RFC 1191 for IPv4, RFC 1981 for IPv6). An INT source or transit switch may report a conservative MTU in the ICMP message, assuming that the packet will go through the maximum number of allowed INT hops (i.e. Remaining Hop Count will decrement to zero), accounting for cumulative metadata insertion at all INT hops, and assuming that the egress MTU at all downstream INT hops is the same as its own egress link MTU. This will help the path MTU discovery source to converge to a path MTU estimate faster, although this would be a conservative path MTU estimate. Alternatively, each INT hop may report an MTU only accounting for the metadata it inserts. This would enable the path MTU discovery source converge to a precise path MTU, at the cost of receiving more ICMP messages, one from each INT hop.

Regardless of whether or not an INT transit switch participates in Path MTU discovery, if it cannot insert all requested metadata because doing so will cause the packet length to exceed egress link MTU, it must not insert any metadata and set the M bit in the INT header, indicating that egress MTU was exceeded at an INT hop.

An INT source inserts 12 bytes of fixed INT headers, and may also insert Per-hop Metadata Length*4 bytes of its own metadata. If inserting the fixed headers causes egress link MTU to be exceeded, INT cannot not be initiated for such packets. If an INT source is programmed to insert its own INT metadata, and there is enough room in a packet to insert fixed INT headers, but no additional room for its INT metadata, the source must initiate INT and set the M bit in the INT header.

In theory, an INT transit switch can perform IPv4 fragmentation to overcome egress MTU limitation when inserting its metadata. However, IPv4 fragmentation can have adverse impact on applications. Moreover, IPv6 packets cannot be fragmented at intermediate hops. Also, fragmenting packets at INT transit hops, with or without copying preceding INT metadata into fragments imposes extra complexity of correlating fragments in the INT monitoring engine. Considering all these factors, this specification requires that an INT switch must not fragment packets in order to append INT information to the packet.

4.4. INT over any encapsulation

The specific location for INT Headers is intentionally not specified: an INT Header can be inserted as an option or payload of any encapsulation type. The only requirements are that the encapsulation header provides sufficient space to carry the INT information and that all INT switches (Sources, transit hops and Sinks) agree on the location of the INT Headers. The following choices are potential encapsulations using common protocol stacks, although a deployment may choose a different encapsulation format if better suited to their needs and environment.

- INT over VXLAN (as VXLAN payload, per GPE extension)
- INT over Geneve (as Geneve option)
- INT over NSH (as NSH payload)
- INT over TCP (as payload)
- INT over UDP (as payload)

- INT over GRE (as a shim between GRE header and encapsulated payload)

4.5. Checksum Update

As described above in section 4.4, INT headers and metadata may be carried in an L4 protocol such as TCP or UDP, or in an encapsulation header that includes an L4 header, such as VXLAN. The checksum field in the TCP or UDP L4 header needs to be updated as INT switches modify the L4 payload via insertion/removal of INT headers and metadata. However, there are certain exceptions. For example, when UDP is transported over IPv4, it is possible to assign a zero checksum, causing the receiver to ignore the value of the checksum field (as defined in RFC 768). For UDP over IPv6, there are specific use cases in which it is possible to assign a zero Checksum (as defined in RFC 6936).

INT source, transit and sink devices must comply with IETF standards for Layer 4 transport protocols with respect to whether or not Layer 4 checksum is to be updated upon modification of Layer 4 payload. For example, if an INT source/transit/sink hop receives UDP traffic with zero L4 checksum, it must not update the L4 checksum in conformance with the behavior defined in relevant IETF standards such as RFC 768 and RFC 6936.

When L4 checksum update is required, an INT source/transit switch may update the checksum in one of two ways:

- Update the L4 Checksum field such that the new value is equal to the checksum of the new packet, after the INT-related updates (header additions/removals, field updates), or
- If the INT source indicates that Checksum-neutral updates are allowed by setting an instruction bit corresponding to the Checksum Complement metadata, then the INT source/transit switches may assign a value to the Checksum Complement metadata which guarantees that the existing L4 Checksum is the correct value of the packet after the INT-related updates.

The motivation for the Checksum Complement is that some hardware implementations process data packets in a serial order, which may impose a problem when INT fields and metadata that reside after the L4 Checksum field are inserted or modified. Therefore, the Checksum Complement metadata, if present, is the last metadata field in the stack.

Note that when the Checksum Complement metadata is present source/transit switches may choose to update the L4 Checksum field instead of using the Checksum Complement metadata. In this case the Checksum Complement metadata must be assigned the reserved value 0xFFFFFFFF. A host that verifies the L4 Checksum will be unaffected by whether some or all of the nodes chose not to use the Checksum Complement, since the value of the L4 Checksum should fit the Checksum of the payload in either of the cases.

INT sink cannot perform a Checksum-neutral update using Checksum Complement metadata as it removes all INT headers from the packet. Thus, an INT sink when performing a checksum update has to do so by updating the L4 Checksum field.

Regardless of whether checksum update is performed via modifying the L4 checksum field or via use of Checksum Complement metadata, performing the update based on an incremental checksum calculation (as is typically done) will ensure that any potential corruption is detected at the point of checksum validation. If full checksum computation is performed at an INT switch, it should be preceded by checksum validation so as to not mask out any corruption at preceding hops.

4.6. Header Location

We describe three encapsulation formats in this specification, covering different deployment scenarios, with and without network virtualization:

1. *INT over TCP/UDP* - A shim header is inserted following TCP/UDP header. INT Headers are carried between this shim header and TCP/UDP payload. This approach doesn't rely on any tunneling/virtualization mechanism and is versatile to apply INT to both native and virtualized traffic.
2. *INT over VXLAN* - VXLAN generic protocol extensions (draft-ietf-nvo3-vxlan-gpe) are used to carry INT Headers between the VXLAN header and the encapsulated VXLAN payload.
3. *INT over Geneve* - Geneve is an extensible tunneling framework, allowing Geneve options to be defined for INT Headers.

4.6.1. INT over TCP/UDP

In case the traffic being monitored is not encapsulated by any virtualization header, INT over VXLAN or INT over Geneve is not helpful. Instead, one can put the INT metadata just after layer 4 headers (TCP/UDP). The scheme assumes that the non-INT devices between the INT source and the INT sink either do not parse beyond layer-4 headers or can skip through the INT stack using the Length field in the INT shim header. If TCP has any options, the INT stack may come before or after the TCP options but the decision must be consistent within an INT domain.

Note that INT over UDP can be used even when the packet is encapsulated by VXLAN, Geneve, or GUE (Generic UDP Encapsulation). INT over TCP/UDP also makes it easier to add INT stack into outer, inner, or even both layers. In such cases both INT header stacks carry information for respective layers and need not be considered interfering with each other.

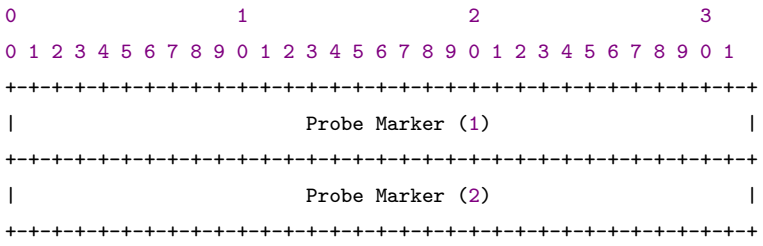
A field in Ethernet, IP, or TCP/UDP should indicate if the INT header exists after the TCP/UDP header. We propose two options.

- IPv4 DSCP or IPv6 Traffic Class field: A value or a bit can be used to indicate the existence of INT after TCP/UDP. When the INT source inserts the INT header into a packet, it sets the reserved value in the field or sets the bit. The INT source may write the original DSCP value in the INT headers so that the INT sink can restore the original value. Restoring the original value is optional.
  - Allocating a bit, as opposed to a value codepoint, will allow the rest of DSCP field to be used for QoS, hence allowing the coexistence of DSCP-based QoS and INT. If the traffic being monitored is subjected to QoS services such as rate limiting, shaping, or differentiated queueing based on DSCP field, QoS classification in the network must be programmed to ignore the designated bit position to ensure that the INT-enabled traffic receives the same treatment as the original traffic being monitored.
  - In brownfield scenarios, however, the network operator may not find a bit available to allocate for INT but may still have a fragmented space of 32 unused DSCP values. The operator can allocate an INT-enabled DSCP value for every QoS DSCP value, map the INT-enabled DSCP value to the same QoS behavior as the corresponding QoS DSCP value. This may double the number of QoS rules but will allow the co-existence of DSCP-based QoS and INT even when a single DSCP bit is not available for INT.
  - Within an INT domain, DSCP values used for INT must exclusively be used for INT.

INT transit and switches must not receive non-INT packets marked with DSCP values used for INT. Any time a switch forwards a packet into the INT domain and there is no INT header present, it must ensure that the DSCP/Traffic class value is not the same as any of the values used to indicate INT.

- Probe Marker fields: If DSCP field or values cannot be reserved for INT, probe marker option could be used. A specific 64-bit value can be inserted after the TCP/UDP header to indicate the existence of INT after TCP/UDP. These fields must be interpreted as unsigned integer values in network byte order. This approach is a variation of an early IETF draft with existing implementation[4].

INT probe marker for TCP/UDP:

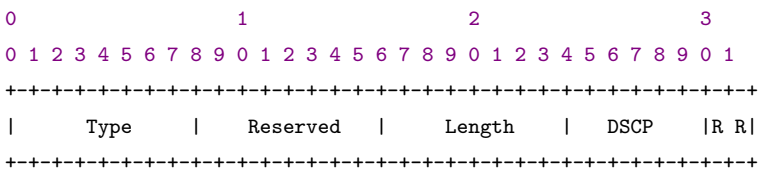


With arbitrary values being inserted after TCP/UDP header as probe markers, the likelihood of conflicting with user traffic in a data center is low, but cannot be completely eliminated. To further reduce the chance of conflict, a deployment could choose to also examine TCP/UDP port numbers to validate INT probe marker.

Any of the above options may be used in an INT domain, provided that the INT transit and sink devices in the INT domain comply with the mechanism chosen at the INT sources, and are able to correctly identify the presence and location of INT headers. The above approaches are not intended to interoperate in a mixed environment, for example it would be incorrect to mark a packet for INT using both DSCP and probe marker, as INT devices that only understand DSCP marking and do not recognize probe markers may incorrectly interpret the first four bytes of the probe marker as INT shim header. It is strongly recommended that only one option be used within an INT domain.

We introduce an INT shim header for TCP/UDP. The INT metadata header and INT metadata stack will be encapsulated between the shim header and the TCP/UDP payload.

INT shim header for TCP/UDP:



- Type: This field indicates the type of INT Header following the shim header. Two Type values are used: one for the hop-by-hop header type and the other for the destination header type (See Section 4.1).

[4] Data-plane probe for in-band telemetry collection, https://tools.ietf.org/html/draft-lapukhov-dataplane-probe-01

- Length: This is the total length of INT metadata header, INT stack and the shim header in 4-byte words. A non-INT device may read this field and skip over INT headers.
- DSCP: If IP DSCP is used to indicate INT, this field optionally stores the original DSCP value. Otherwise, this field is reserved.

All the other bits in the shim header are reserved for future use.

4.6.2. INT over VXLAN GPE

VXLAN is a common tunneling protocol for network virtualization and is supported by most software virtual switches and hardware network elements. The VXLAN header as defined in RFC 7348 is a fixed 8-byte header as shown below.

VXLAN Header:

```
0                   1                   2                   3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|R|R|Ver|I|P|B|O|                    Reserved                   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|               VXLAN Network Identifier (VNI) |   Reserved    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

The amount of free space in the VXLAN header allows for carrying minimal network state information. Hence, we embed INT metadata in a shim header between the VXLAN header and the encapsulated payload.

The VXLAN header as defined in RFC 7348 does not specify the protocol being encapsulated and assumes that the payload following the VXLAN header is an Ethernet payload. Internet draft draft-ietf-nvo3-vxlan-gpe proposes changes to the VXLAN header to allow for multi-protocol encapsulation. We use this VXLAN generic protocol extension draft and propose a new "Next-protocol" type for INT.

VXLAN GPE Header:

```
0                   1                   2                   3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|R|R|Ver|I|P|B|O|          Reserved             | Next Protocol |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|               VXLAN Network Identifier (VNI) |   Reserved    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

P bit: Flag bit 5 is defined as the Next Protocol bit. The P bit MUST be set to 1 to indicate the presence of the 8-bit next protocol field.

Next Protocol Values:

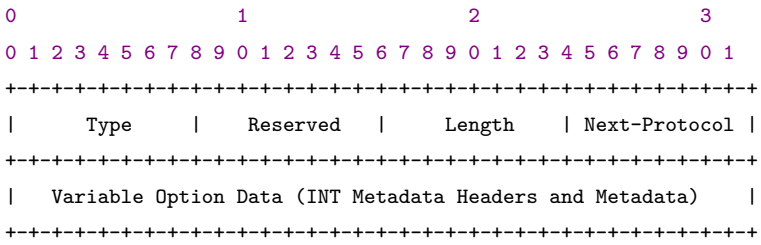
- 0x01: IPv4
- 0x02: IPv6
- 0x03: Ethernet
- 0x04: Network Service Header (NSH)
- 0x05: MPLS

- 0x06: GBP
- 0x07: vBNG
- 0x08: In-band Network Telemetry Header (This value has not been reserved by VXLAN GPE specification yet, and is hence subject to change)

When there is one INT Header in the VXLAN GPE stack, the VXLAN GPE header for the INT Header will have a next-protocol value other than INT Header indicating the payload following the INT Header - typically Ethernet. If there are multiple INT Headers in the VXLAN GPE stack (for example if both hop-by-hop and destination type INT headers are being carried), then all VXLAN GPE shim headers for the INT Headers other than the last one will carry 0x08 for their next-protocol values, and the VXLAN GPE header for the last INT Header will carry next-protocol value of the original VXLAN payload (e.g., Ethernet).

To embed a variable-length data (i.e., INT metadata) in the VXLAN GPE stack, we introduce the INT shim header. This header follows each VXLAN GPE header for INT.

INT shim header for VXLAN GPE encapsulation:

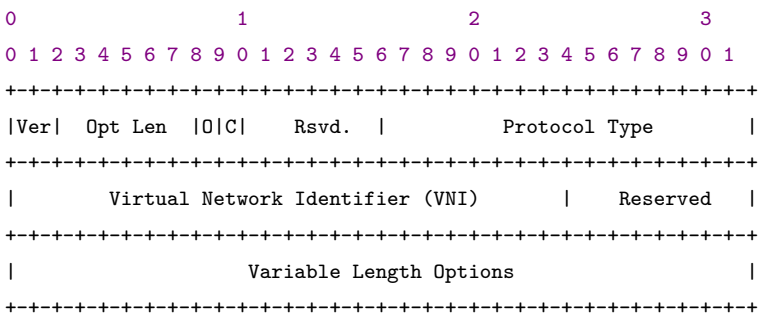


- Type: This field indicates the type of INT Header following the shim header. Two Type values are used: one for the hop-by-hop header type and the other for the destination header type (See Section 4.1).
- Length: This is the total length of the variable INT option data and the shim header in 4-byte words.

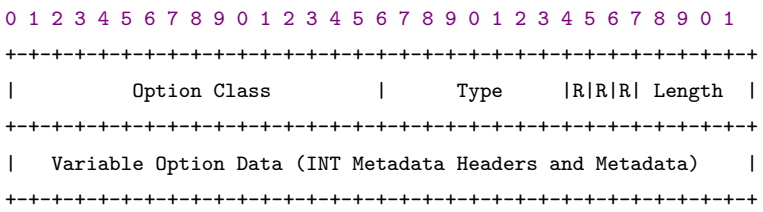
4.6.3. INT over Geneve

Geneve is a generic and extensible tunneling framework, allowing for INT metadata to be carried in TLV format as "Option headers" in the tunnel header.

Geneve Header:



Geneve Option for INT:



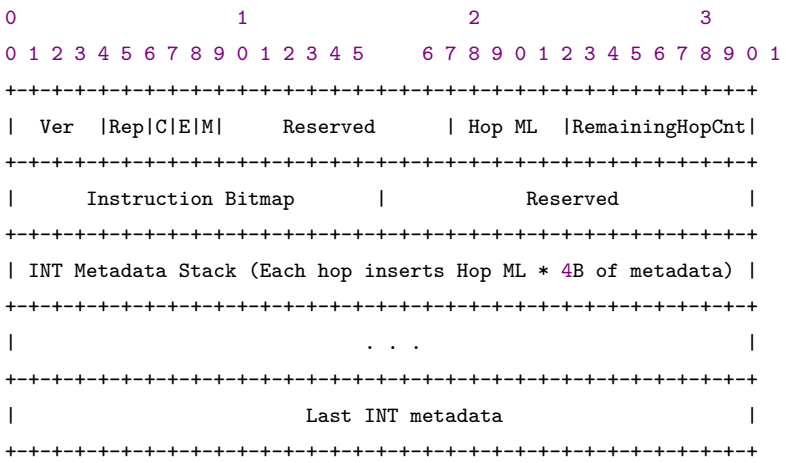
Note:

- We do not need to reserve any special values for fields in the base Geneve header for INT.
- Users may or may not use INT with Geneve along with VNI (network virtualization), though using INT with Geneve without network virtualization would be a bit wasteful.
- Specific values of Option Class and Type have not been reserved for INT yet. Deployments using INT over Geneve must use configurable values for Option Class and Type to be able to migrate to different values as and when specific values are defined for INT.
- The variable length option data following the Geneve Option Header carries the actual INT metadata header and metadata.
- The Length field of the Geneve Option header is 5-bits long, which limits a single Geneve option instance to no more than 124 bytes long (31 * 4). Remaining Hop Count in hop-by-hop INT header has to be set accordingly at the INT source to ensure that the Geneve option does not overflow. The entire hop-by-hop INT header must fit in a single Geneve option.

4.7. INT Hop-by-Hop Metadata Header Format

In this section, we define the format for INT hop-by-hop metadata headers, and the metadata itself.

INT Metadata Header and Metadata Stack:



- INT metadata header is 8 bytes long followed by a stack of INT metadata. Each metadata is either 4 bytes or 8 bytes in length. Each INT hop adds the same length of metadata. The total length of the metadata stack is variable as different packets may traverse different paths and hence different number of INT hops.

- The fields in the INT metadata header are interpreted the following way:

- Ver (4b): INT metadata header version. Should be 1 for this version.
- Rep (2b): Replication requested. Support for this request is optional. If this value is non-zero, the device may replicate the INT packet. This is useful to explore all the valid physical forwarding paths when multi-path forwarding techniques (e.g., ECMP, LAG) are used in the network. Note the Rep bits should be used judiciously (e.g., only for probe packets, not for every data packet). While we recommend that Rep bits be set only for probe packets, the INT architecture does not (and perhaps cannot) disallow use of the Rep bits for real data packets.
  * 0: No replication requested.
  * 1: Port-level (L2-level) replication requested. If the INT packet is forwarded through a logical port that is a port-channel (LAG), then replicate the packet on each physical port in the port-channel and send a single copy per physical port.
  * 2: Next-hop-level (L3-level) replication requested. Forward the packet to each L3 ECMP next-hop valid for the destination address, with INT headers replicated in each forwarded copy.
  * 3: Port-level and Next-hop-level replication requested.
- C (1b): Copy.
  * If replication is requested for data packets, the INT Sink must be able to distinguish the original packet from replicas so that it can forward only original packets up the protocol stack, and drop all the replicas. The C bit must be set to 1 on each copy, whenever an INT hop replicates a packet. The original packet must have C bit set to 0.
  * C bit must be set to 0 in the original packet by INT source
- E (1b): Max Hop Count exceeded.
  * This flag must be set if a device cannot prepend its own metadata due to the Remaining Hop Count reaching zero.
  * E bit must be set to 0 by INT source
- M (1b): MTU exceeded
  * This flag must be set if a device cannot add all of the requested metadata because doing so will cause the packet length to exceed egress link MTU. In this case, the device must not add any metadata to the packet, and set the M bit in the INT header. Note that it is possible for egress MTU limitation to prevent INT metadata insertion at multiple hops along a path. The M bit simply serves as an indication that INT metadata was not inserted at one or more hops and corrective action such as reconfiguring MTU at some links may be needed, particularly when INT switches are not participating in path MTU discovery. The M bit is not aimed at readily identifying which switch(es) did not insert INT metadata due to egress MTU limitation. In theory, if this does not occur at consecutive hops, it may be possible for the monitoring system to derive which switch(es) set the M bit based on knowledge of the network topology and "Switch ID, Ingress port ID, Egress port ID" tuples in the INT metadata stack.
- R: Reserved bits.
- Hop ML (5b): Per-hop Metadata Length, the length of metadata in 4-Byte words to be

inserted at each INT hop.

* While the largest value of Per-hop Metadata Length is 31, an INT-capable device may be limited in the maximum number of instructions it can process and/or maximum length of metadata it can insert in data packets. An INT hop that cannot process all instructions must still insert Per-hop Metadata Length * 4 bytes, with all-ones reserved value (4 or 8 bytes of 0xFF depending on the length of metadata) for the metadata corresponding to instructions it cannot process. An INT hop that cannot insert Per-hop Metadata Length * 4 bytes must skip INT processing altogether and not insert any metadata in the packet.

– Remaining Hop Count (8b): The remaining number of hops that are allowed to add their metadata to the packet.

* Upon creation of an INT metadata header, the INT Source must set this value to the maximum number of hops that are allowed to add metadata instance(s) to the packet. Each INT-capable device on the path, including the INT Source as well as INT Transit Hops, must decrement the Remaining Hop Count if and when it pushes its local metadata onto the stack.
* When a packet is received with the Remaining Hop Count equal to 0, the device must ignore the INT instruction, pushing no new metadata onto the stack, and the device must set the E bit.

- INT instructions are encoded as a bitmap in the 16-bit INT Instruction field: each bit corresponds to a specific standard metadata as specified in Section 3.

  – bit0 (MSB): Switch ID
  – bit1: Level 1 Ingress Port ID (16 bits) + Egress Port ID (16 bits)
  – bit2: Hop latency
  – bit3: Queue ID (8 bits) + Queue occupancy (24 bits)
  – bit4: Ingress timestamp
  – bit5: Egress timestamp
  – bit6: Level 2 Ingress Port ID + Egress Port ID (4 bytes each)
  – bit7: Egress port Tx utilization
  – bit15: Checksum Complement
  – The remaining bits are reserved. Each instruction requests 4 bytes of metadata to be inserted at each hop, except if bit 6 is set, which requires 8 bytes of metadata. Per-hop metadata length is set accordingly at the INT source.

- Each INT Transit device along the path that supports INT adds its own metadata values as specified in the instruction bitmap immediately after the INT metadata header.

  – When adding a new metadata, each device must prepend its metadata in front of the metadata that are already added by the upstream devices. This is similar to the push operation on a stack. Hence, the most recently added metadata appears at the top of the stack. The device must add metadata in the order of bits set in the instruction bitmap.
  – If a device is unable to provide a metadata value specified in the instruction bitmap because its value is not available, it must add a special all-ones reserved value indicating "invalid" (4 or 8 bytes of 0xFF depending on metadata length).

  - If a device cannot add all the metadata required by the instruction bitmap (irrespective of the availability of the metadata values that are asked for), it must skip processing that particular INT packet entirely. This ensures that each INT Transit device adds either zero bytes or Per-hop Metadata Length*4 bytes to the packet.
  - Reserved bits in the instruction bitmap are to be handled similarly. If an INT transit hop receives a reserved bit set in the instruction bitmap (e.g. set by a INT source that is running a newer version), the transit hop must either add corresponding metadata filled with the reserved value 0xFFFFFFFF or must not add any INT metadata to the packet. This means that an instruction bit marked reserved in this specification may be used for a 4B metadata in a subsequent minor version while still being backward compatible with this specification. However, an instruction bit marked reserved in this specification may be used for a 8B metadata only in the next major version, breaking backward compatibility and requring all INT switches to be upgraded to the new major version. For example a version 1.0 INT switch cannot operate alongside version 2.0 INT switches if a new 8B metadata is introduced in version 2.0, as the version 1.0 INT switch could insert 0xFFFFFFFF reserved value for a 8B metadata field, thus breaking the metadata stack length invariance - the length of metadata stack will not be a multiple of Per-Hop Metadata length * 4 in this case.
  - If an INT transit hop does not add metadata to a packet due to any of the above reasons, it must not decrement the remaining INT hop count in the INT metadata header.

- Summary of the field usage

  - The INT Source must set the following fields:
    * Ver, Rep, C, M, Per-hop Metadata Length, Remaining Hop Count, and Instruction Bitmap.
    * INT Source must set all reserved bits to zero.
  - Intermediate devices can set the following fields:
    * C, E, M, Remaining Hop Count

- The length (in bytes) of the INT metadata stack must always be a multiple of (Per-hop Metadata Length * 4). This length can be determined by subtracting the total INT fixed header sizes (12 bytes) from (shim header length * 4). For INT over Geneve it is 8 bytes subtracted from (length in Geneve tunnel option header * 4).

5. Examples

This section shows example INT Headers with two hosts (Host1 and Host2), communicating over a network path composed of three network devices (Switch1, Switch2 and Switch3) as shown below.

```
==> packet P travels from Host1 to Host2 ==>
Host1 --------> Switch1 ---------> Switch2 ---------> Switch3 --------> Host2
```

Detailed assumptions made for this example are as follows

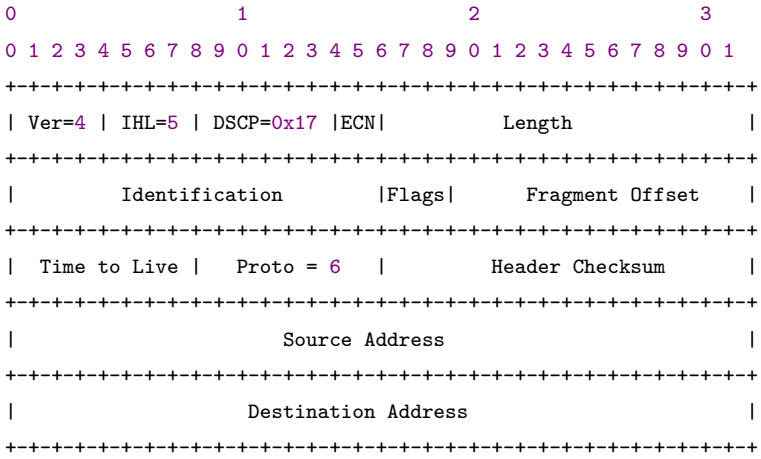
- INT source requests each INT hop to insert switch ID and queue occupancy (For the sake of illustration we only consider switch ID and queue occupancy being inserted at each hop. Queue IDs are typically defined per port, hence in a real use-case queue occupancy is likely to be collected along with egress port ID)
- There are three devices (hops) on the path, and all the devices expose both metadata (switch ID and queue occupancy).
- The maximum number of hops (network diameter) is 8.
- The values of INT metadata header fields in this example are as follows:
  - Ver = 1
  - Rep = 0 (No replication)
  - C = 0
  - E = 0 (Max Hop Count not exceeded)
  - M = 0 (MTU not exceeded at any switch)
  - Per-hop Metadata Length = 2 (for switch id & queue occupancy)
  - Remaining hop count starts at 8, decremented by 1 at each hop that inserts INT metadata

5.1. Example with INT over TCP

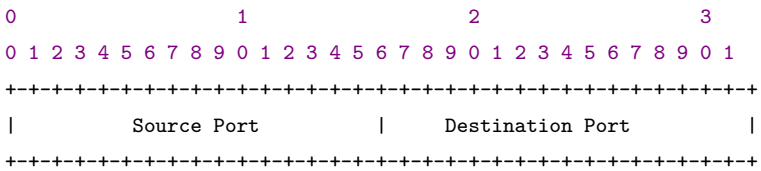
We consider a scenario where host1 sends a TCP packet to host2. The ToR switch of host1 (Switch1) acts as the INT source. It adds INT headers and its own metadata in the packet. Switch2 prepends its metadata. Finally, the ToR switch of host2 (Switch3) acts as the INT sink and removes INT headers before forwarding the packet to host2.

Below is the packet received by INT sink Switch3, starting from the IPv4 header. We use the value of 0x17 for IPv4.DSCP to indicate the existence of INT headers.

IP Header:



TCP Header:



```
|                        Sequence Number                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Acknowledgment Number                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Data |           |U|A|P|R|S|F|                               |
| Offset| Reserved  |R|C|S|S|Y|I|            Window             |
|       |           |G|K|H|T|N|N|                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           Checksum            |         Urgent Pointer        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

INT Shim Header for TCP/UDP, INT type is hop-by-hop:

```
0                   1                   2                   3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     Type=1    |   Reserved    |  Length = 8   |   DSCP    |R R|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

INT Metadata Header and Metadata Stack, followed by TCP payload:

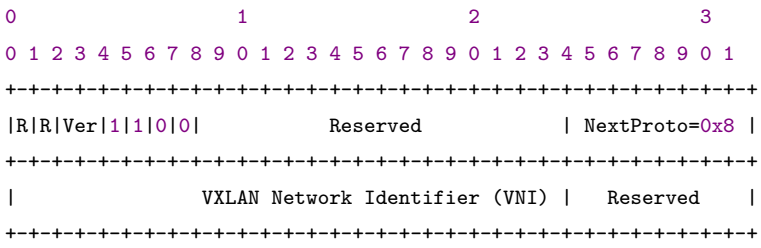
```
0                   1                   2                   3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Ver=1 | 0 |0|0|0|     Reserved      | HopML=2 |RemainingHopC=6|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0|           Reserved            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          sw id of hop2                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                     queue occupancy of hop2                   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          sw id of hop1                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                     queue occupancy of hop1                   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         TCP payload                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

5.2. Example with INT over VXLAN GPE

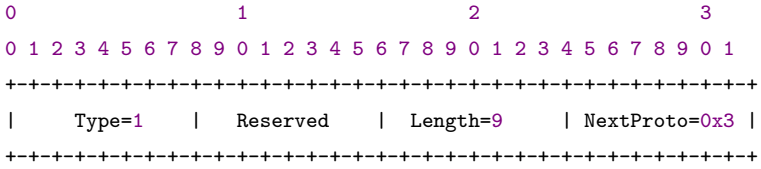
We now consider a scenario where Host1 and Host2 use VXLAN encapsulation. Host1 acts as VXLAN tunnel endpoint and INT source, inserts VXLAN and INT headers with instruction bits corresponding to the network state to be reported at intermediate switches. In this example, Host1 itself does not insert any INT metadata. Intermediate switches parse through VXLAN header and populate the INT metadata. Host2 acts as INT sink and VXLAN tunnel endpoint, removes INT and VXLAN headers.

The packet headers received at Host 2 are as follows, starting with the VXLAN header (encapsulating ethernet, IP and UDP headers are not shown here):

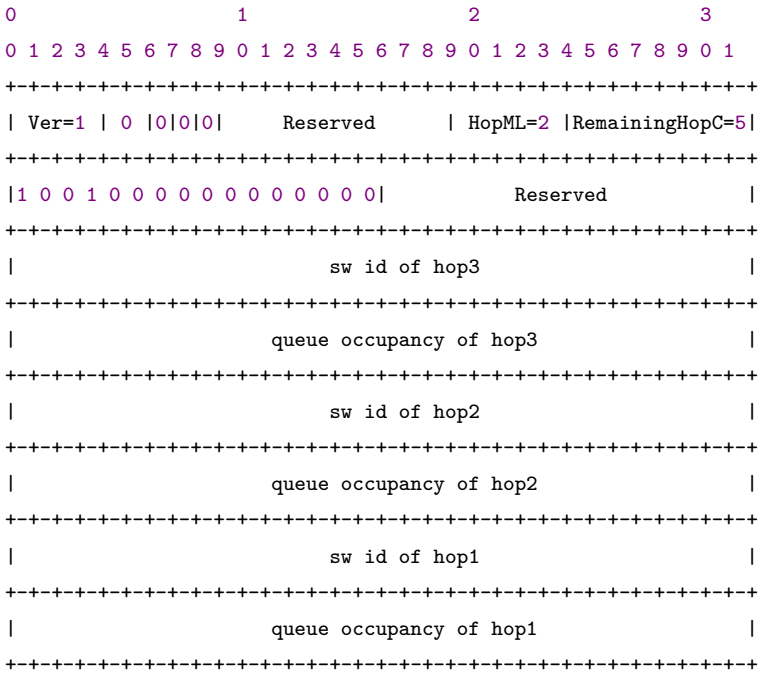
VXLAN GPE Header:



INT Shim Header for VXLAN-GPE, hop-by-hop INT type:



INT Metadata Header and Metadata Stack:



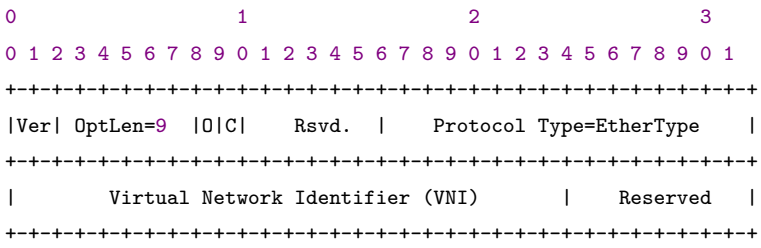
5.3. Example with INT over Geneve

Finally, we consider a scenario where Host1 and Host2 use Geneve encapsulation. Host1 acts as Geneve tunnel endpoint and INT source, inserts Geneve and INT headers with instruction bits corresponding to the network state to be reported at intermediate switches. In this example, Host1 itself does not insert any INT metadata. Intermediate switches parse through Geneve header and populate the INT metadata. Host2 acts as INT sink and Geneve tunnel endpoint, removes INT

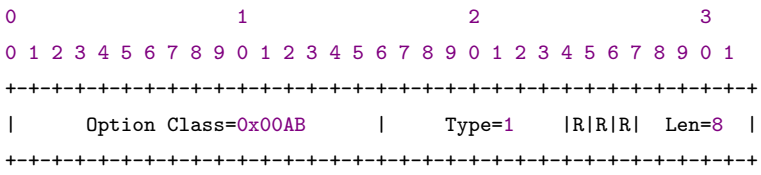
and Geneve headers.

The following is the Geneve and INT Headers attached to the packet received by Host2.

Geneve Header:



Geneve Option for INT, hop-by-hop type:



INT Metadata Header and Metadata Stack:

6. P4 program specification for INT Transit

P4 program is an unambiguous description of a protocol. Here we present a P4 program for an INT Transit device, describing the INT headers, the parsing operations for the headers, incremental checksum updates, and the match-action tables implementing the INT Transit behaviors. The program is written in P4_16 on PSA architecture, and it assumes INT over TCP/UDP as the

encapsulation protocol.

This program is work in progress.

```
#include <core.p4>
#include <psa.p4>

/*************************************************************************
 * headers and structs
 *************************************************************************/

/* INT shim header for TCP/UDP */
header intl4_shim_t {
    bit<8>  int_type;
    bit<8>  rsvd1;
    bit<8>  len;
    bit<6>  dscp;
    bit<2>  rsvd2;
}

/* INT header */
/* 16 instruction bits are defined in four 4b fields to allow concurrent
   lookups of the bits without listing 2^16 combinations */
header int_header_t {
    bit<4>  ver;
    bit<2>  rep;
    bit<1>  c;
    bit<1>  e;
    bit<1>  m;
    bit<7>  rsvd1;
    bit<3>  rsvd2;
    bit<5>  hop_metadata_len;
    bit<8>  remaining_hop_cnt;
    bit<4>  instruction_mask_0003;
    bit<4>  instruction_mask_0407;
    bit<4>  instruction_mask_0811;
    bit<4>  instruction_mask_1215;
    bit<16> rsvd3;
}

/* INT meta-value headers - different header for each value type */
header int_switch_id_t {
    bit<32> switch_id;
}

header int_level1_port_ids_t {
    bit<16> ingress_port_id;
    bit<16> egress_port_id;
```

```
}

header int_hop_latency_t {
    bit<32> hop_latency;
}

header int_q_occupancy_t {
    bit<8>  q_id;
    bit<24> q_occupancy;
}

header int_ingress_tstamp_t {
    bit<32> ingress_tstamp;
}

header int_egress_tstamp_t {
    bit<32> egress_tstamp;
}

header int_level2_port_ids_t {
    bit<32> ingress_port_id;
    bit<32> egress_port_id;
}

header int_egress_port_tx_util_t {
    bit<32> egress_port_tx_util;
}

/* standard ethernet/ip/tcp headers */
header ethernet_t {
    bit<48> dstAddr;
    bit<48> srcAddr;
    bit<16> etherType;
}

/* define diffserv field as DSCP(6b) + ECN(2b) */
header ipv4_t {
    bit<4>  version;
    bit<4>  ihl;
    bit<6>  dscp;
    bit<2>  ecn;

    bit<16> totalLen;
    bit<16> identification;
    bit<3>  flags;
    bit<13> fragOffset;
```

```
    bit<8>  ttl;
    bit<8>  protocol;
    bit<16> hdrChecksum;
    bit<32> srcAddr;
    bit<32> dstAddr;
}

header tcp_t {
    bit<16> srcPort;
    bit<16> dstPort;
    bit<32> seqNo;
    bit<32> ackNo;
    bit<4>  dataOffset;
    bit<4>  res;
    bit<8>  flags;
    bit<16> window;
    bit<16> checksum;
    bit<16> urgentPtr;
}

header udp_t {
    bit<16> srcPort;
    bit<16> dstPort;
    bit<16> length_;
    bit<16> checksum;
}

struct headers {
    ethernet_t                  ethernet;
    ipv4_t                      ipv4;
    tcp_t                       tcp;
    udp_t                       udp;
    intl4_shim_t                intl4_shim;
    int_header_t                int_header;
    int_switch_id_t             int_switch_id;
    int_level1_port_ids_t       int_level1_port_ids;
    int_hop_latency_t           int_hop_latency;
    int_q_occupancy_t           int_q_occupancy;
    int_ingress_tstamp_t        int_ingress_tstamp;
    int_egress_tstamp_t         int_egress_tstamp;
    int_level2_port_ids_t       int_level2_port_ids;
    int_egress_port_tx_util_t   int_egress_port_tx_util;
}

struct empty_metadata_t {
}
```

```
/* port id and timestamp types are defined in PSA */
struct bridged_ingress_input_metadata_t {
    PortId_t    ingress_port;
    Timestamp_t ingress_timestamp;
}

/* switch internal variables for INT logic implementation */
struct int_metadata_t {
    bit<16>  insert_byte_cnt;
    bit<8>   int_hdr_word_len;
    bit<32>  switch_id;
}

struct fwd_metadata_t {
    bit<16>  l3_mtu;
    bit<16>  checksum_state;
}

struct metadata {
    bridged_ingress_input_metadata_t bridged_istd;
    int_metadata_t                   int_metadata;
    fwd_metadata_t                   fwd_metadata;
}

error {
    BadIPv4HeaderChecksum
}

/*************************************************************************
 * parsers and deparsers
 *************************************************************************/

/* Checksum verification and update of ipv4, tcp and udp are inspired by
 * p4lang/p4-spec/blob/master/p4-16/psa/examples/psa-example-incremental-checksum2.p4
 * For checksum related details, check the notes in the PSA example.
 */

/* This reference code processes INT Transit at egress where all
 * switch metadata become available.
 * Ingress doesn't need to parse or deparse INT.
 */

parser IngressParserImpl(packet_in packet,
                         out headers hdr,
                         inout metadata meta,
```

```
                        in psa_ingress_parser_input_metadata_t istd,
                        in empty_metadata_t resubmit_meta,
                        in empty_metadata_t recirculate_meta)
{
    InternetChecksum() ck;

    state start {
        transition parse_ethernet;
    }
    state parse_ethernet {
        packet.extract(hdr.ethernet);
        transition select(hdr.ethernet.etherType) {
            16w0x800: parse_ipv4;
            default: accept;
        }
    }
    state parse_ipv4 {
        packet.extract(hdr.ipv4);

        ck.clear();
        ck.add({
                hdr.ipv4.version, hdr.ipv4.ihl, hdr.ipv4.dscp, hdr.ipv4.ecn,
                hdr.ipv4.totalLen,
                hdr.ipv4.identification,
                hdr.ipv4.flags, hdr.ipv4.fragOffset,
                hdr.ipv4.ttl, hdr.ipv4.protocol,
                hdr.ipv4.srcAddr,
                hdr.ipv4.dstAddr
            });
        verify(hdr.ipv4.hdrChecksum == ck.get(), error.BadIPv4HeaderChecksum);

        // For incremental update of TCP/UDP checksums
        // subtract out the contributions of the IPv4 'pseudo header'
        // fields that the P4 program might change
        ck.clear();
        ck.subtract({
                hdr.ipv4.srcAddr,
                hdr.ipv4.dstAddr,
                hdr.ipv4.totalLen
            });

        transition select(hdr.ipv4.protocol) {
            6 : parse_tcp;
            17: parse_udp;
            default: accept;
        }
```

```
    }

    state parse_tcp {
        packet.extract(hdr.tcp);
        ck.subtract({
                hdr.tcp.srcPort,
                hdr.tcp.dstPort,
                hdr.tcp.seqNo,
                hdr.tcp.ackNo,
                hdr.tcp.dataOffset, hdr.tcp.res,
                hdr.tcp.flags,
                hdr.tcp.window,
                hdr.tcp.checksum,
                hdr.tcp.urgentPtr
            });
        meta.fwd_metadata.checksum_state = ck.get_state();
        transition accept;
    }

    state parse_udp {
        packet.extract(hdr.udp);
        ck.subtract({
                hdr.udp.srcPort,
                hdr.udp.dstPort,
                hdr.udp.length_,
                hdr.udp.checksum
            });
        meta.fwd_metadata.checksum_state = ck.get_state();
        transition accept;
    }
}

control IngressDeparserImpl(packet_out packet,
    out empty_metadata_t clone_i2e_meta,
    out empty_metadata_t resubmit_meta,
                            out metadata normal_meta,
                            inout headers hdr,
                            in metadata meta,
                            in psa_ingress_output_metadata_t istd)
{
    apply {
        if (psa_normal(istd)) {
            normal_meta = meta;
        }
        packet.emit(hdr.ethernet);
        packet.emit(hdr.ipv4);
```

```
        packet.emit(hdr.tcp);
        packet.emit(hdr.udp);
    }
}

/* indicate INT by DSCP value */
const bit<6> DSCP_INT = 0x17;
const bit<6> DSCP_MASK = 0x3F;

parser EgressParserImpl(packet_in packet,
                        out headers hdr,
                        inout metadata meta,
                        in psa_egress_parser_input_metadata_t istd,
                        in metadata normal_meta,
                        in empty_metadata_t clone_i2e_meta,
                        in empty_metadata_t clone_e2e_meta)
{
    InternetChecksum() ck;

    state start {
        transition copy_normal_meta;
    }
    state copy_normal_meta {
        meta = normal_meta;
        transition parse_ethernet;
    }
    state parse_ethernet {
        packet.extract(hdr.ethernet);
        transition select(hdr.ethernet.etherType) {
            16w0x800: parse_ipv4;
            default: accept;
        }
    }
    state parse_ipv4 {
        packet.extract(hdr.ipv4);
        transition select(hdr.ipv4.protocol) {
            6 : parse_tcp;
            17: parse_udp;
            default: accept;
        }
    }

    state parse_tcp {
        packet.extract(hdr.tcp);
        transition select(hdr.ipv4.dscp) {
        /* &&& is a mask operator in p4_16 */
```

```
            DSCP_INT &&& DSCP_MASK: parse_intl4_shim;
            default: accept;
        }
    }

    state parse_udp {
        packet.extract(hdr.udp);
        transition select(hdr.ipv4.dscp) {
            DSCP_INT &&& DSCP_MASK: parse_intl4_shim;
            default: accept;
        }
    }

    /* INT headers are parsed first time at egress,
     * hence subtract INT header fields from checksum
     * for incremental update
     */
    state parse_intl4_shim {
        packet.extract(hdr.intl4_shim);
        ck.subtract({
                hdr.intl4_shim.int_type, hdr.intl4_shim.rsvd1,
                hdr.intl4_shim.len, hdr.intl4_shim.dscp, hdr.intl4_shim.rsvd2
            });
        transition parse_int_header;
    }
    state parse_int_header {
        packet.extract(hdr.int_header);
        ck.subtract({
            hdr.int_header.ver, hdr.int_header.rep,
            hdr.int_header.c, hdr.int_header.e,
            hdr.int_header.m, hdr.int_header.rsvd1,
            hdr.int_header.rsvd2, hdr.int_header.hop_metadata_len,
            hdr.int_header.remaining_hop_cnt,
            hdr.int_header.instruction_mask_0003,
            hdr.int_header.instruction_mask_0407,
            hdr.int_header.instruction_mask_0811,
            hdr.int_header.instruction_mask_1215,
            hdr.int_header.rsvd3
        });
        meta.fwd_metadata.checksum_state = ck.get_state();
        transition accept;
    }
}

control EgressDeparserImpl(packet_out packet,
                           out empty_metadata_t clone_e2e_meta,
```

```
                        out empty_metadata_t recirculate_meta,
                        inout headers hdr,
                        in metadata meta,
                        in psa_egress_output_metadata_t istd,
                        in psa_egress_deparser_input_metadata_t edstd)
{
    InternetChecksum() ck;
    apply {
        if (hdr.ipv4.isValid()) {
            ck.clear();
            ck.add({
                hdr.ipv4.version, hdr.ipv4.ihl, hdr.ipv4.dscp, hdr.ipv4.ecn,
                hdr.ipv4.totalLen,
                hdr.ipv4.identification,
                hdr.ipv4.flags, hdr.ipv4.fragOffset,
                hdr.ipv4.ttl, hdr.ipv4.protocol,
                hdr.ipv4.srcAddr,
                hdr.ipv4.dstAddr
            });
            hdr.ipv4.hdrChecksum = ck.get();
        }

        // TCP/UDP header incremental checksum update.
        // Restore the checksum state partially calculated in the parser.
        ck.set_state(meta.fwd_metadata.checksum_state);

        // Add back relevant header fields, including new INT metadata
        if (hdr.ipv4.isValid()) {
            ck.add({
                hdr.ipv4.srcAddr,
                hdr.ipv4.dstAddr,
                hdr.ipv4.totalLen
            });
        }

        if (hdr.intl4_shim.isValid()) {
            ck.add({
                    hdr.intl4_shim.int_type, hdr.intl4_shim.rsvd1,
                    hdr.intl4_shim.len, hdr.intl4_shim.dscp, hdr.intl4_shim.rsvd2
                });
        }

        if (hdr.int_header.isValid()) {
            ck.add({
                    hdr.int_header.ver, hdr.int_header.rep,
                    hdr.int_header.c, hdr.int_header.e,
```

```
            hdr.int_header.m, hdr.int_header.rsvd1,
            hdr.int_header.rsvd2, hdr.int_header.hop_metadata_len,
            hdr.int_header.remaining_hop_cnt,
            hdr.int_header.instruction_mask_0003,
            hdr.int_header.instruction_mask_0407,
            hdr.int_header.instruction_mask_0811,
            hdr.int_header.instruction_mask_1215,
            hdr.int_header.rsvd3
        });
}

if (hdr.int_switch_id.isValid()) {
    ck.add({hdr.int_switch_id.switch_id});
}

if (hdr.int_level1_port_ids.isValid()) {
    ck.add({
            hdr.int_level1_port_ids.ingress_port_id,
            hdr.int_level1_port_ids.egress_port_id
        });
}

if (hdr.int_hop_latency.isValid()) {
    ck.add({hdr.int_hop_latency.hop_latency});
}

if (hdr.int_q_occupancy.isValid()) {
    ck.add({
            hdr.int_q_occupancy.q_id,
            hdr.int_q_occupancy.q_occupancy
        });
}

if (hdr.int_ingress_tstamp.isValid()) {
    ck.add({hdr.int_ingress_tstamp.ingress_tstamp});
}

if (hdr.int_egress_tstamp.isValid()) {
    ck.add({hdr.int_egress_tstamp.egress_tstamp});
}

if (hdr.int_level2_port_ids.isValid()) {
    ck.add({
            hdr.int_level2_port_ids.ingress_port_id,
            hdr.int_level2_port_ids.egress_port_id
        });
```

```
}

if (hdr.int_egress_port_tx_util.isValid()) {
    ck.add({hdr.int_egress_port_tx_util.egress_port_tx_util});
}

if (hdr.tcp.isValid()) {
    ck.add({
        hdr.tcp.srcPort,
        hdr.tcp.dstPort,
        hdr.tcp.seqNo,
        hdr.tcp.ackNo,
        hdr.tcp.dataOffset, hdr.tcp.res,
        hdr.tcp.flags,
        hdr.tcp.window,
        hdr.tcp.urgentPtr
    });
    hdr.tcp.checksum = ck.get();
}

if (hdr.udp.isValid()) {
    ck.add({
        hdr.udp.srcPort,
        hdr.udp.dstPort,
        hdr.udp.length_
    });

    // If hdr.udp.checksum was received as 0, we
    // should never change it.  If the calculated checksum is
    // 0, send all 1 bits instead.
    if (hdr.udp.checksum != 0) {
        hdr.udp.checksum = ck.get();
        if (hdr.udp.checksum == 0) {
            hdr.udp.checksum = 0xffff;
        }
    }
}

packet.emit(hdr.ethernet);
packet.emit(hdr.ipv4);
packet.emit(hdr.tcp);
packet.emit(hdr.udp);
packet.emit(hdr.intl4_shim);
packet.emit(hdr.int_header);
packet.emit(hdr.int_switch_id);
packet.emit(hdr.int_level1_port_ids);
```

```
        packet.emit(hdr.int_hop_latency);
        packet.emit(hdr.int_q_occupancy);
        packet.emit(hdr.int_ingress_tstamp);
        packet.emit(hdr.int_egress_tstamp);
        packet.emit(hdr.int_level2_port_ids);
        packet.emit(hdr.int_egress_port_tx_util);
    }
}

control Int_metadata_insert(inout headers hdr,
        in int_metadata_t int_metadata,
        in bridged_ingress_input_metadata_t bridged_istd,
        in psa_egress_input_metadata_t istd)
{
    /* this reference implementation covers only INT instructions 0-3 */
    action int_set_header_0() {
        hdr.int_switch_id.setValid();
        hdr.int_switch_id.switch_id = int_metadata.switch_id;
    }
    action int_set_header_1() {
        hdr.int_level1_port_ids.setValid();
        hdr.int_level1_port_ids.ingress_port_id =
            (bit<16>) bridged_istd.ingress_port;
        hdr.int_level1_port_ids.egress_port_id =
            (bit<16>) istd.egress_port;
    }
    action int_set_header_2() {
        hdr.int_hop_latency.setValid();
        hdr.int_hop_latency.hop_latency =
            (bit<32>) (istd.egress_timestamp - bridged_istd.ingress_timestamp);
    }
    action int_set_header_3() {
        hdr.int_q_occupancy.setValid();
        // PSA doesn't support queueing metadata yet
        hdr.int_q_occupancy.q_id = 0xFF;
        hdr.int_q_occupancy.q_occupancy = 0xFFFFFF;
    }

    /* action functions for bits 0-3 combinations, 0 is msb, 3 is lsb */
    /* Each bit set indicates that corresponding INT header should be added */
    action int_set_header_0003_i0() {
    }
    action int_set_header_0003_i1() {
        int_set_header_3();
    }
    action int_set_header_0003_i2() {
```

```
        int_set_header_2();
    }
    action int_set_header_0003_i3() {
        int_set_header_3();
        int_set_header_2();
    }
    action int_set_header_0003_i4() {
        int_set_header_1();
    }
    action int_set_header_0003_i5() {
        int_set_header_3();
        int_set_header_1();
    }
    action int_set_header_0003_i6() {
        int_set_header_2();
        int_set_header_1();
    }
    action int_set_header_0003_i7() {
        int_set_header_3();
        int_set_header_2();
        int_set_header_1();
    }
    action int_set_header_0003_i8() {
        int_set_header_0();
    }
    action int_set_header_0003_i9() {
        int_set_header_3();
        int_set_header_0();
    }
    action int_set_header_0003_i10() {
        int_set_header_2();
        int_set_header_0();
    }
    action int_set_header_0003_i11() {
        int_set_header_3();
        int_set_header_2();
        int_set_header_0();
    }
    action int_set_header_0003_i12() {
        int_set_header_1();
        int_set_header_0();
    }
    action int_set_header_0003_i13() {
        int_set_header_3();
        int_set_header_1();
        int_set_header_0();
```

```
}
action int_set_header_0003_i14() {
    int_set_header_2();
    int_set_header_1();
    int_set_header_0();
}
action int_set_header_0003_i15() {
    int_set_header_3();
    int_set_header_2();
    int_set_header_1();
    int_set_header_0();
}

/* Table to process instruction bits 0-3 */
table int_inst_0003 {
    key = {
        hdr.int_header.instruction_mask_0003 : exact;
    }
    actions = {
        int_set_header_0003_i0;
        int_set_header_0003_i1;
        int_set_header_0003_i2;
        int_set_header_0003_i3;
        int_set_header_0003_i4;
        int_set_header_0003_i5;
        int_set_header_0003_i6;
        int_set_header_0003_i7;
        int_set_header_0003_i8;
        int_set_header_0003_i9;
        int_set_header_0003_i10;
        int_set_header_0003_i11;
        int_set_header_0003_i12;
        int_set_header_0003_i13;
        int_set_header_0003_i14;
        int_set_header_0003_i15;
    }
    default_action = int_set_header_0003_i0();
    size = 16;
}

/* Similar tables can be defined for instruction bits 4-7 and bits 8-11 */
/* e.g., int_inst_0407, int_inst_0811 */

apply{
    int_inst_0003.apply();
    // int_inst_0407.apply();
```

```
        // int_inst_0811.apply();
    }
}

control Int_outer_encap(inout headers hdr,
        in int_metadata_t int_metadata)
{
    action int_update_ipv4() {
        hdr.ipv4.totalLen = hdr.ipv4.totalLen + int_metadata.insert_byte_cnt;
    }
    action int_update_shim() {
        hdr.intl4_shim.len = hdr.intl4_shim.len + int_metadata.int_hdr_word_len;
    }

    apply{
        if (hdr.ipv4.isValid()) {
            int_update_ipv4();
        }
        /* Add: UDP length update if you support UDP */

        if (hdr.intl4_shim.isValid()) {
            int_update_shim();
        }
    }
}

control Int_ingress(inout metadata meta,
                    in psa_ingress_input_metadata_t  istd)
{
    action bridge_ingress_istd() {
        meta.bridged_istd.ingress_port = istd.ingress_port;
        meta.bridged_istd.ingress_timestamp = istd.ingress_timestamp;
    }
    apply{
        bridge_ingress_istd();
    }
}

control Int_egress(inout headers hdr,
                   inout metadata meta,
                   in    psa_egress_input_metadata_t  istd)
{
    action int_hop_cnt_exceeded() {
        hdr.int_header.e = 1;
    }
    action int_mtu_limit_hit() {
```

```
    hdr.int_header.m = 1;
}
action int_hop_cnt_decrement() {
    hdr.int_header.remaining_hop_cnt =
        hdr.int_header.remaining_hop_cnt - 1;
}
action int_transit(bit<32> switch_id, bit<16> l3_mtu) {
    meta.int_metadata.switch_id = switch_id;
    meta.int_metadata.insert_byte_cnt =
        (bit<16>) hdr.int_header.hop_metadata_len << 2;
    meta.int_metadata.int_hdr_word_len =
        (bit<8>) hdr.int_header.hop_metadata_len;
    meta.fwd_metadata.l3_mtu = l3_mtu;
}
table int_prep {
    key = {}
    actions = {int_transit;}
}

Int_metadata_insert() int_metadata_insert;
Int_outer_encap() int_outer_encap;

apply{
    if(hdr.int_header.isValid()) {
        if(hdr.int_header.remaining_hop_cnt == 0
                || hdr.int_header.e == 1) {
            int_hop_cnt_exceeded();
        } else if ((hdr.int_header.instruction_mask_0811 ++
                    hdr.int_header.instruction_mask_1215)
                    & 8w0xFE == 0 ) {
            /* v1.0 spec allows two options for handling unsupported
             * INT instructions. This exmple code skips the entire
             * hop if any unsupported bit (bit 8 to 14 in v1.0 spec) is set.
             */
            int_prep.apply();
            // check MTU limit
            if (hdr.ipv4.totalLen + meta.int_metadata.insert_byte_cnt
                    > meta.fwd_metadata.l3_mtu) {
                int_mtu_limit_hit();
            } else {
                int_hop_cnt_decrement();
                int_metadata_insert.apply(hdr,
                                          meta.int_metadata,
                                          meta.bridged_istd,
                                          istd);
                int_outer_encap.apply(hdr, meta.int_metadata);
```

```
                }
            }
        }
    }
}

control ingress(inout headers hdr,
                inout metadata meta,
                in    psa_ingress_input_metadata_t  istd,
                inout psa_ingress_output_metadata_t ostd)
{
    Int_ingress() int_ingress;
    apply{
        /* ... ingress code here ... */
        int_ingress.apply(meta, istd);
        /* ... ingress code here ... */
    }
}

control egress(inout headers hdr,
               inout metadata meta,
               in    psa_egress_input_metadata_t  istd,
               inout psa_egress_output_metadata_t ostd)
{
    Int_egress() int_egress;
    apply{
        /* ... egress code here ... */
        int_egress.apply(hdr, meta, istd);
        /* ... egress code here ... */
    }
}

IngressPipeline(IngressParserImpl(),
                ingress(),
                IngressDeparserImpl()) ip;

EgressPipeline(EgressParserImpl(),
               egress(),
               EgressDeparserImpl()) ep;

PSA_SWITCH(ip, ep) main;


/* End of Code snippet */
```

A. Appendix: An extensive (but not exhaustive) set of Metadata

Here we list a set of examplary metadata that future versions of the spec may support as well as those are supported in the current spec.

A.1. Switch-level

- Switch id
  - The unique ID of a switch (generally administratively assigned). SwitchIDs must be unique within a domain.
- Control plane state version number
  - Whenever a control-plane state changes (e.g., IP FIB update), the switch control plane can also update this version number in the data plane. INT packets may use these version numbers to determine which control-plane state was active at the time packets were forwarded.

A.2. Ingress

- Ingress port identifier
  - The port on which the INT packet was received. A packet may be received on an arbitrary stack of port constructs starting with a physical port. For example, a packet may be received on a physical port that belongs to a link aggregation port group, which in turn is part of a Layer 3 Switched Virtual Interface, and at Layer 3 the packet may be received in a tunnel. Although the entire port stack may be monitored in theory, this specification allows for monitoring of up to two levels of ingress port identifiers. The semantics of port identifiers may differ across devices, each INT hop chooses the port type it reports at each of the two levels.
- Ingress timestamp
  - The device local time when the INT packet was received on the ***ingress*** physical or logical port.
- Ingress port RX pkt count
  - Total # of packets received so far (since device initialization or counter reset) on the ingress physical or logical port where the INT packet was received.
- Ingress port RX byte count
  - Total # of bytes received so far on the ingress physical or logical port where the INT packet was received.
- Ingress port RX drop count
  - Total # of packet drops occurred so far on the ingress physical or logical port where the INT packet was received.

- Ingress port RX utilization
  - Current utilization of the ingress physical or logical port where the INT packet was received. The exact mechanism (bin bucketing, moving average, etc.) is device specific and while the latter is clearly superior to the former, the INT framework leaves those decisions to device vendors.

A.3. Egress

- Egress port identifier
  - The port on which the INT packet was sent out. A packet may be transmitted on an arbitrary stack of port constructs ending at a physical port. For example, a packet may be transmitted on a tunnel, out of a Layer 3 Switched Virtual Interface, on a Link Aggregation Group, out of a particular physical port belonging to the Link Aggregation Group. Although the entire port stack may be monitored in theory, this specification allows for monitoring of up to two levels of egress port identifiers. The semantics of port identifiers may differ across devices, each INT hop chooses the port type it reports at each of the two levels.
- Egress timestamp
  - The device local time when the INT packet was processed by the egress physical or logical port.
- Egress port TX pkt count
  - Total # of packets forwarded so far (since device initialization or counter reset) through the egress physical or logical port where the INT packet was also forwarded.
- Egress port TX byte count
  - Total # of bytes forwarded so far through the egress physical or logical port where the INT packet was forwarded.
- Egress port TX drop count
  - Total # of packet drops occurred so far on the egress physical or logical port where the INT packet was forwarded.
- Egress port TX utilization
  - Current utilization of the egress port via which the INT packet was sent out.

A.4. Buffer Information

- Queue id
  - The id of the queue the device used to serve the INT packet.
- Instantaneous queue length

  - The instantaneous length (in bytes, cells, or packets) of the queue the INT packet has observed in the device while being forwarded. The units used need not be consistent across an INT domain, but care must be taken to ensure that there is a known, consistent mapping of {device, queue} values to their respective unit {packets, bytes, cells}.

- Average queue length
  - The average length (in bytes, cells, or packets) of the queue via which the INT packet was served. The calculation mechanism of this value is device specific.

- Queue drop count
  - Total # of packets dropped from the queue

A.5. Miscellaneous

- Checksum Complement
  - This field enables a Checksum-neutral update when INT is encapsulated over an L4 protocol that uses a Checksum field, such as TCP or UDP.

B. Acknowledgements

We thank the following individuals for their contributions to the design, specification and implementation of this spec.

- Parag Bhide
- Dennis Cai
- Dan Daly
- Bruce Davie
- Ed Doe
- Senthil Ganesan
- Anoop Ghanwani
- Mukesh Hira
- Hugh Holbrook
- Raja Jayakumar
- Changhoon Kim
- Jeongkeun Lee
- Tal Mizrahi
- Masoud Moshref
- Michael Orr
- Heidi OU
- Mickey Spiegel
- Bapi Vinnakota

C. Change log

- 2015-09-28

  - Initial release

- 2016-06-19

  - Updated section 4.6.2, the Length field definition of VXLAN GPE shim header, to be consistent with the example in section 5.

- 2017-10-17

  - Introduced INT over TCP/UDP (section 4.6.1 and new example)
  - Removed BOS (Bottom-Of-Stack) bit at each 4B metadata, from the header definition and examples
  - Updated the INT instruction bitmap and the meaning of a few instructions (section 4.7)
  - Moved the INT transit P4 program from Appendix to the main section 6. Re-wrote the program in p4_16.

- 2017-12-11

  - Increaed the size of Version field from 2b to 4b in INT Metadata Header
  - Improved the header presentation of the examples and clarified the assumptions in section 5
  - Formatted the spec as a Madoko file
  - **Tag v0.5 spec**

- 2018-02-13

  - Elaborated on interactions between INT and MTU settings. Defined switch behavior when inserting INT metadata in a packet would result in egress link MTU to be exceeded.
  - Defined behavior of INT transit switch when it receives reserved bits set in the INT header

- 2018-02-14

  - Replaced Max Hop Count and Total Hop Count with Remaining Hop Count

- 2018-02-28

  - Added Probe Marker approach as another way to indicate the existence of INT over TCP/UDP (section 4.6.1).

- 2018-03-08

  - Added support for monitoring of two levels of ingress and egress port identifiers

- 2018-03-13

  - Defined INT domain in section 2.
  - Described a possible allocation of non-contiguous DSCP codepoints for INT over TCP/UDP in section 4.6.1.
  - Relaxed the location of INT stack relative to TCP options in section 4.6.1.

- 2018-03-14

  - Added the Checksum Complement metadata.

- 2018-03-29
  - Removed queue congestion status from the list of metadata.
  - Removed Section 4.2 (Handling INT Packets) on slow path processing using follow-up packets.
  - Removed the examples of piggybacked metadata for closed loop control.
  - The expectation is that any of these may be reintroduced in future versions of INT. They could benefit from a better understanding of use cases and some preliminary implementation experience.
- 2018-03-31
  - Defined checksum update behavior more precisely
  - Miscellaneous editorial changes in preparation for v1.0
- 2018-04-02
  - Revised the example transit code to be compliant with spec v1.0, perform incremental TCP/UDP checksum updates, and against PSA architecture instead of v1model.
- 2018-04-03
  - Some more editorial changes for v1.0
- 2018-04-10
  - Removed the option to modify L4 destination port to indicate INT over TCP/UDP.
  - Removed INT Tail header from INT over TCP/UDP encapsulation.
  - Added DSCP to the INT over TCP/UDP shim header.
- 2018-04-20
  - Some more editorial changes for v1.0
  - **Tag v1.0 spec**
- 2018-05-08
  - Fixed checksum subtract/add calls in the reference code
- 2018-08-17
  - Fixed INT DSCP mask in the reference code