In-band Network Telemetry (INT)

(NOTE: This is a working draft. Consider using tagged versions for implementations.)

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

2018-03-14

Contents

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 transits 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 intentionally generic, and hence can enable a number of interesting high level applications, such as:

- Network troubleshooting
  - L1 traceroute, micro-burst detection, packet history (a.k.a. postcards), trajectory sampling
- Advanced congestion control
  - RCP, XCP, TIMELY
- Advanced routing
  - Utilization-aware routing (e.g., CONGA)
- Network data-plane verification

A number of use case descriptions and evaluations are described in the Millions of Little Minions paper [1].

2. Terminologies

- **INT Header**: Any packet header that carries INT information. There are two types of INT Headers – *Hop-by-hop* and *Destination* (See Section 4.1).
- **INT Packet**: Any packet containing an INT Header.
- **INT Instruction**: Embedded packet instructions indicating which INT Metadata to collect (defined below). The collected data is written into an INT Header.
- **INT Source**: A trusted entity that creates and inserts INT Headers into the packets it sends. The INT Headers contain, at minimum, INT Instructions indicating what to collect.
- **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.)
- **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 below.

[1] Millions of Little Minions: Using Packets for Low Latency Network Programming and Visibility, ACM SIGCOMM 2014.

- **INT Domain**: A set of inter-connected INT devices under the same administration. 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 and the INT technology becomes more popular, we expect to add more metadata to this INT specification.

Note the exact meaning of the following metadata (e.g., the unit of timestamp values, the precise definition of hop latency or congestion status) can vary by device 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 the users of INT will communicate the precise meanings of these metadata for each device model they use in their networks. For example, metadata units may be communicated out-of-band between each INT device and a data collection system.

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 (See section 5.3 for details). 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 adequately large number of logical ports at a 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 (See section 5.3 for details). 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 adequately large number of logical ports at a 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.

3.4. Buffer Information

- 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.
- Queue congestion status
  - The fraction (in percentage or decimal fraction) of current queue occupancy relative to the queue-size limit. This indicates how much buffer space was used relative to the maximum buffer space (instantaneously or statically) available to the queue.

4. Processing 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 either 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
  - Intermediate devices (INT Transit Hops) must process this type of INT Header.

- Destination type
  - Destination headers must only be consumed by the INT Sink; intermediate devices must ignore Destination headers.
  - Destination headers can be used for two purposes (for example):
    - To enable communication between INT Source and INT Sink.
      - INT Source can add a sequence number to detect lost INT packets.
      - INT Source can add the original IP TTL value of an INT packet. This way, an INT Sink can detect network devices on the path that do not support INT (and hence failed to add INT metadata) by checking the difference between the number of INT metadata instances in the INT Header (i.e., # of INT-compliant hops) and the decrement of the IP TTL values (i.e., # of physical IP hops).
    - To deliver follow-up INT packets to the INT Sinks (see Section 4.2)
      - Follow-up packets generated by the slow-path forwarding logic of a transit-hop switch must carry the original INT instructions but must not trigger any further INT processing by downstream devices.

4.2. Handling INT Packets

It is obviously preferable for network devices to process any INT packets strictly within the fast path, often a hardware based forwarding plane. An ideal system would be able to process INT instructions with no increase in latency or reduction in forwarding performance, but in some cases it may be required to process INT packets outside the fast path. This slow-path processing could be a CPU based control plane, some sideband or alternate hardware assisted forwarding path, or an arbitrary INT resource. Note that in the case where the INT processing is done outside the fast path, the device MUST still forward the original packet through the fast path (i.e. without processing the INT instructions). The implementation of this is not specified, though it implies the ability to make a copy of the INT packet for slow-path processing or a similar design. Following the processing of the INT packet in the normal fast-path, the forwarding plane should generate a ***trigger*** toward the slow path (e.g., either a copy of the original INT packet or a digest of it). Upon receiving the trigger, the slow path should process the INT Header appropriately, generate a new packet – called a "follow-up packet" – containing the execution results of the INT instructions. The follow-up packet is forwarded separately.

If devices in the network do perform slow-path INT processing, it is possible that a single INT packet could spawn multiple follow-up packets – and in turn each of these could spawn more INT processing downstream. Care must be taken to prevent excessive replication. To prevent the cascading generation of follow-up telemetry packets, all follow-up packets are marked with a special "exemption" flag. The presence of this flag instructs downstream devices to provide specific processing. For more specific information, see the option type for INT Source-to-sink communication messages (Destination type in Section 4.1).

The INT Header of a follow-up packet must contain all the existing INT metadata in the original packet that was added by the upstream devices, as well as its own local INT metadata. Follow-up packets must contain enough information (from outer header, inner header, or both of the original packet) so that the INT sink can correlate the follow-up packets with the original INT packet. Because INT is not tied to a particular encapsulation protocol, this spec does not dictate the exact format of a follow-up packet other than its INT portion.

To prevent potential packet ordering issues, it is recommended that an INT device ***NOT*** forward the INT packets themselves via the slow path while processing INT Headers.

5. Header Format and Location

This section specifies the format and location of INT Headers.

5.1. INT over any encapsulation

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

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

For each encapsulation format, we need to reserve a next-header type identifier (e.g., a VXLAN-GPE Next Protocol value, a Geneve Option Class value, or a TCP/UDP port number) to indicate the presence of an INT Header.

As illustrative examples, we describe three encapsulation formats, both suitable for use in virtualized data centers:

1. *INT over TCP/UDP* - This example introduces a shim header after TCP/UDP header and carry INT Headers between the 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* - Our examples use the VXLAN generic protocol extensions (draft-ietf-nvo3-vxlan-gpe) to carry INT Headers between VXLAN header and encapsulated VXLAN payload.
3. *INT over Geneve* - Geneve is an extensible tunneling framework, allowing Geneve options to be defined for INT Headers.

5.2. On-the-fly 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.

5.2.1. MTU Settings

As with any modification that potentially changes the packet size, this on-the-fly allocation may "grow" the packet size, potentially causing it to exceed the MTU on an INT switch egress link.

This may be addressed in the following ways -

- It is recommended that the MTU of links between INT sources and sinks is configured to be higher than the MTU of preceding link MTUs (server/VM NIC MTUs) by an appropriate amount. Configuring an MTU differential of [Per-hop Metadata Length*4*INT Hop Count + 12] 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. Here the 12 bytes are derived from the 4 bytes of INT shim (or option) header needed for

INT encapsulation (over TCP/UDP, VXLAN-GPE or Geneve) plus the 8 bytes of fixed INT metadata header.

- An INT source/transit switch may optionally aid in dynamic discovery of Path MTU along flows being monitored by INT by transmitting ICMP message as per Path MTU Discovery mechanisms of the corresponding IP protocol (RFC 1191 for IPv4, RFC 1981 for IPv6). An INT source or transit switch may report a conservative MTU in the ICMP message, accounting for cumulative metadata insertion at all INT hops and assuming egress MTU at downstream hops is the same as its own egress link MTU. This will help in the path MTU discovery source converging 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 needs to be able to insert 12 bytes of fixed INT shim header and metadata header, plus Per-hop Metadata Length*4 bytes of its own metadata. If inserting 12 bytes of fixed headers causes egress link MTU to be exceeded, INT will not be initiated for such packets (**TBD - handling of such scenarios**). If there is enough room to insert INT header, but not first-hop INT metadata at the source, the source must initiate INT and set the M bit at the first hop itself.

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, the INT specification requires that an INT switch **must not fragment** packets in order to append INT information to the packet.

5.2.2. Checksum Update

INT may be transported over an L4 protocol such as TCP or UDP, or over an encapsulation header that includes an L4 header, such as VXLAN. In these cases, an INT source, transit, or sink switch may be required to update the L4 Checksum of the encapsulating header.

In some cases, an L4 Checksum update is not necessary. 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. For UDP over IPv6, there are specific use cases in which it is possible to assign a zero Checksum (as defined in RFC 6936).

If an L4 Checksum update is required, an INT source/transit/sink switch may perform it 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. INT sink doesn't need to take any special action but just removing the Checksum complemenet metadata as part of decapsulating the entire INT header stack.

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 will 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.

5.3. Header Format

This subsection proposes the INT Header format. Where necessary, we use examples based on the INT-over-TCP/UDP, INT-over-VXLAN or INT-over-GENEVE encapsulation formats.

5.3.1. Header Location and Format – INT over TCP/UDP

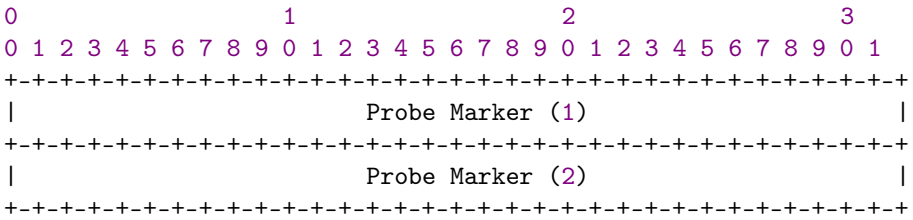
In case the packet is not encapsulated by any virtualization header, INT over VXLAN or INT over GENEVE is not helpful. Instead, one can put the INT metadata just after layer 4 headers (TCP/UDP). The scheme assumes that the non-INT devices between the INT source and the INT sink either do not parse beyond layer-4 headers or can skip through the INT stack using the Length field of 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: 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 the lower layers of Ethernet, IP, or TCP/UDP should indicate if the INT header exists after the TCP/UDP header. We propose three options:

- IPv4 DSCP or IPv6 Traffic Class field: A value or a bit will be reserved to indicate the existence of INT after TCP/UDP. INT source will put the reserved value in the field, and INT sink will remove it. The source can store the original DSCP so that the 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. The QoS engine must be programmed to ignore the designated bit position.
  - 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 queue of traffic class 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.
- TCP/UDP destination Port field: A port number in layer 4 will be reserved to indicate the existence of INT after TCP/UDP. INT source changes the destination Port, and sink must restore the original port number.
- New Probe Marker fields: arbitrary 64-bit values are inserted after IP TCP/UDP header to indicate the existence of INT after TCP/UDP. These fields should be interpreted as unsigned integer values, stored in network byte order and are initialized to a configured value. This Probe marker is a variation of an early IETF draft with existing implementations[2].

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

INT probe marker for TCP/UDP:

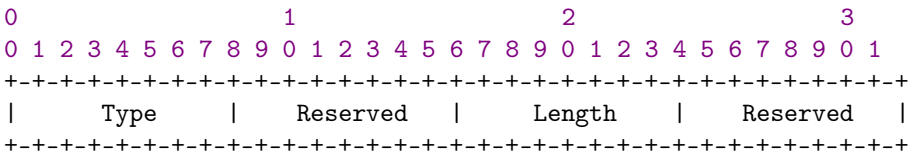


The decision to use DSCP, destination port field or probe marker option are per domain decisions that all INT capable devices should follow. The same applies to the reserved value of DSCP or port number, and the configured probe marker value, which should be consistent within a domain. It is recommended that only one option is deployed per domain. (Note: in general, encoding into DSCP field will be less intrusive compared to changing the layer 4 port field. The latter may alter ECMP behavior and can complicate ACL and network debugging. If DSCP field cannot be reserved for INT, and retaining ECMP behavior is desired, probe marker option could be followed. With arbitrary values being inserted after TCP/UDP header in probe marker option, 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, the user who deploys probe marker option could choose to check more into TCP/UDP port numbers to validate INT probe marker)

INT over TCP/UDP adds INT metadata after TCP/UDP headers as if the payload is changed. However, the sink device will remove INT headers before passing the packet to the traffic destination. Therefore, updating the TCP/UDP checksum (and UDP.length in case of UDP) is optional and a per domain decision.

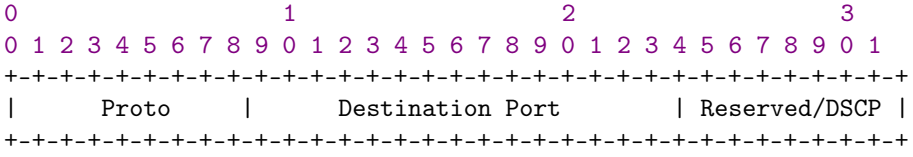
We introduce INT shim header and INT tail header for TCP/UDP, and their formats are as follows. The INT metadata header and INT metadata stack will be encapsulated between the shim and tail headers.

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).
- Length: This is the total length of INT metadata header, INT stack and the shim and tail headers in 4-byte words. A device can read this field and just skip through the whole INT headers to reach TCP/UDP payload.

INT tail header for TCP/UDP:



- Proto: A copy of IPv4.proto or IPv6.nextHeader.
- Destination Port: The original destination port of TCP/UDP protocol. In case the special destination port is used to indicate INT existence, this field is used by INT sink to restore the original value. Besides, having the proto and destination port fields in the tail header would

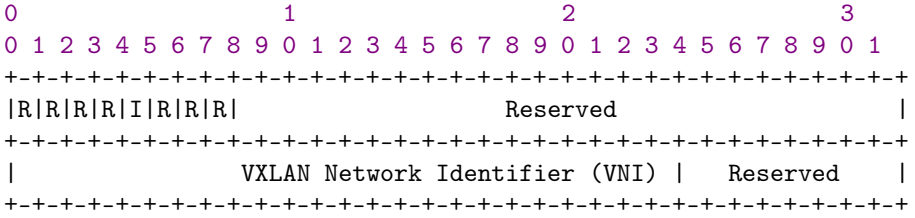
help transit and sink devices to know how to parse the rest of the packet after the INT headers.
- Reserved/DSCP: In case the IP DSCP is used to indicate INT existence, this field stores the original DSCP value.

Proto and Destination Port fields are needed in the tail header regardless of which of DSCP or Destination Port was used to indicate the existence of INT.

5.3.2. Header Location and Format – 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.

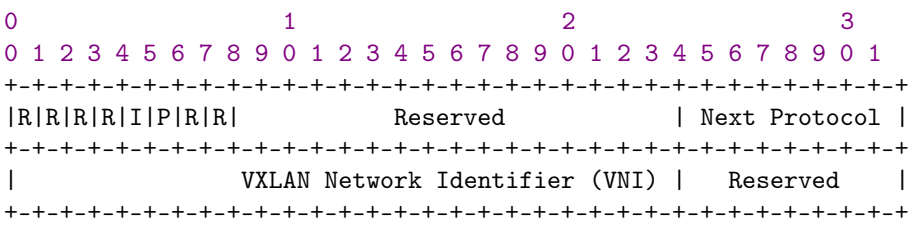
VXLAN Header:



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. This is the recommended approach as it allows for carrying more network state information along an entire path.

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-00.txt 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:



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:

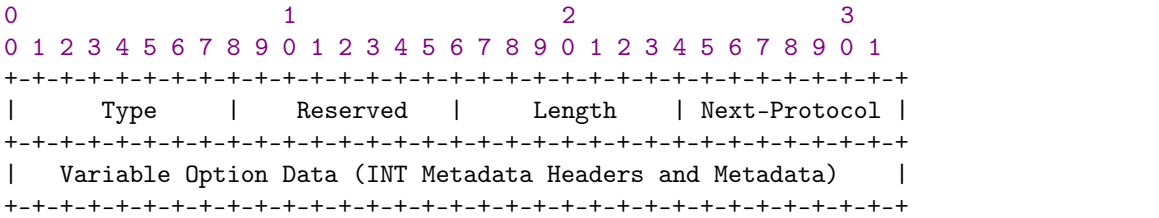
- 0x01: IPv4
- 0x02: IPv6
- 0x03: Ethernet
- 0x04: Network Service Header (NSH)
- 0x05: In-band Network Telemetry (INT) Header (the value is subject to change)

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, then all VXLAN GPE **shim** headers for the INT Headers other than the last one will carry 0x05

for their next-protocol values. And, the VXLAN GPE header for the last INT Header will carry a 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 of which format is as follows. This header follows each VXLAN GPE header for INT.

INT shim header for VXLAN GPE encapsulation:

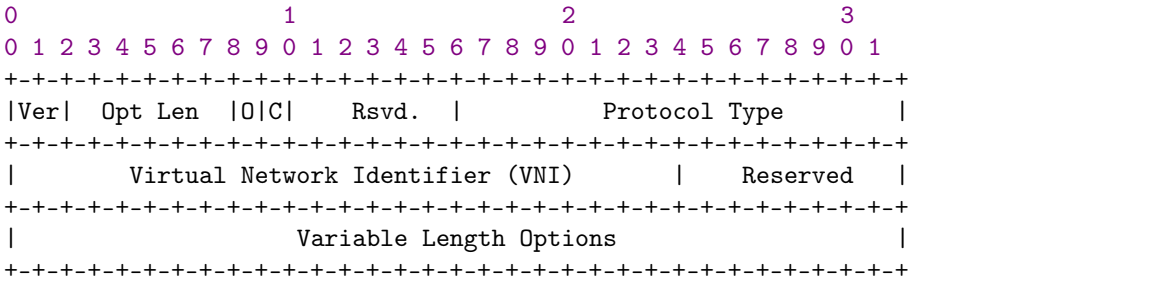


- Type: This field indicates the type of INT Header following the shim header. Two Type values are used: one for the hop-by-hop header type and the other for the destination header type (See Section 4.1).
- Length: This is the total length of the variable INT option data and the shim header in 4-byte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

5.3.3. Header Location and Format – INT over Geneve

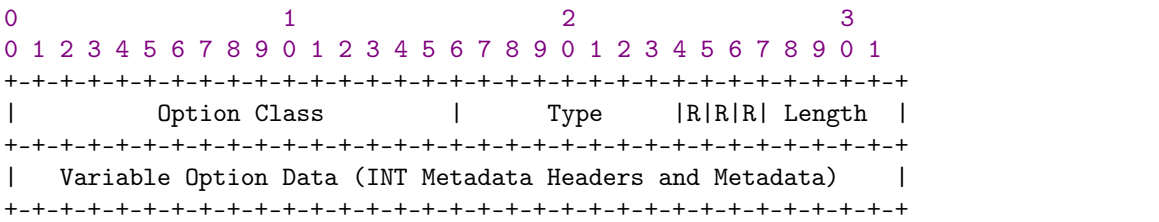
Geneve is a generic and extensible tunneling framework, allowing for current and future network virtualization implementations to carry metadata encoded in TLV format as "Option headers" in the tunnel header.

Geneve Header:



- 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.

Geneve Option Format:



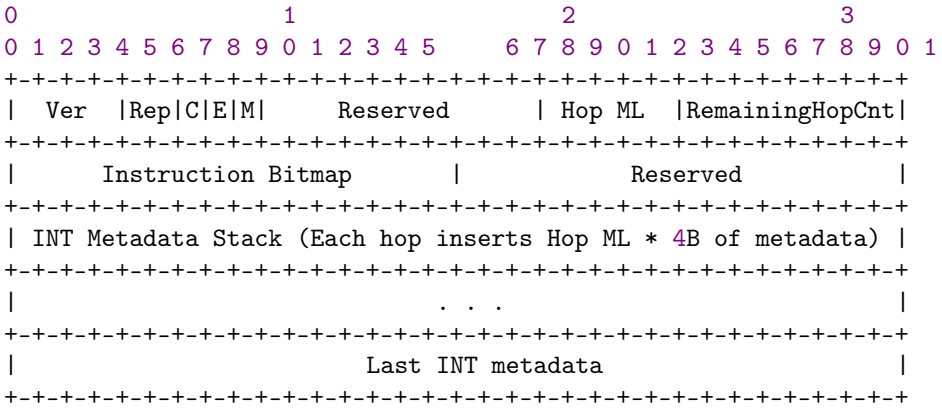
- We need to reserve a unique Option Class value for INT.
- We need to reserve two Type values associated with the Option Class for INT – one for the hop-by-hop header type and the other for the destination header type (See Section 4.1).

- The variable length option data following the Geneve Option Header carries the actual INT metadata header and metadata.
- Note 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). If 124 bytes is insufficient one could collect different, non-overlapping sets of INT metadata info across multiple packets.

5.3.4. INT Metadata Header Format

In this section, we describe the format for INT 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 zero 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. Replicate the packet to each L3 ECMP next-hop valid for the destination address and send a single copy per ECMP next-hop.
    * 3: Port- and Next-hop-level replication requested.

  - C (1b): Copy.

    * If replication is indeed 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 up the protocol stack, and drop all the replicas. The C bit must be set to

1 on each copy, whenever INT transit hop replicates a packet. The original packet must have C bit set to 0.
    - C bit must always be set to 0 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 the value 0.
  - 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, but 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 0xFFFFFFFF reserved value 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 + Egress Port ID
  - bit2: Hop latency
  - bit3: Queue ID + Queue occupancy
  - bit4: Ingress timestamp

  - bit5: Egress timestamp
  - bit6: Queue ID + Queue congestion status
  - bit7: Egress port tx utilization
  - bit8: Level 2 Ingress Port ID + Egress Port ID (4 bytes each)
  - bit15: Checksum Complement
  - The remaining bits are reserved. Each instruction requests 4 bytes of metadata to be inserted at each hop, except if bit 8 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 reserved value 0xFFFF_FFFF indicating "invalid".
  - 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 0xFFFF_FFFF or must not add any INT metadata to the packet.
  - 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, 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, Remaining Hop Count

- In an INT packet, 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 (for INT over TCP/UDP, 16 bytes; for INT over VXLAN-GPE, 12 bytes) from (shim header length * 4). For INT over Geneve it is 8 bytes subtracted from (length in Geneve tunnel option header * 4).

5.4. Examples

This section shows example INT Headers. The assumptions made for this example are as follows.

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

Two hosts (Host1 and Host2) communicate through a network path composed of three network devices – Switch1, 2, and 3. The example in this section shows INT Headers attached to data packet P forwarded by Switch3 and received by Host2; the INT Source of this data packet is Host1, and the INT Sink is Host2. The INT metadata attached to this packet are the switch ID (sw id) and queue occupancy (q len) from every switch on the forwarding path from Host1 to Host2.

While it is not the scope of this spec, we assume that Host1 may also deliver the INT metadata (queue occupancy) collected via a previous packet it received from Host2 by piggybacking the data onto packet P. We assume that Host1 uses the destination-type INT Header for that because intermediate devices must ignore such a header type. We also assume that Host1 uses the same INT metadata header format for the piggybacked INT metadata just for convenience.

The following is the detailed assumption made for this example.

- As an INT Source, Host1 wants to collect switch id and queue occupancy from each device on the path. It uses the hop-by-hop-type INT Header to do so.
- As an INT Sink, Host1 wants to piggyback the queue occupancy values collected from the Host2-to-Host1 path onto the data packet sent back to Host2. It uses the destination-type INT Header.
- There are three devices (hops) on the path, and all the devices can expose both metadata (switch id and queue occupancy).
- INT type value for the hop-by-hop INT Header type (i.e., the one that intermediate switches must process) is 1, and that the type value for the destination INT Header is 2.
- The maximum number of hops (network diameter) is 8.
- The INT metadata header uses the following field values in the metadata header, unless specified otherwise in each example.
  - Ver = 0
  - Rep = 0 (No replication)
  - C = 0
  - E = 0 (Max Hop Count not exceeded)
  - Per-hop Metadata Length = 2 (for switch id & queue occupancy)
- The piggybacked metadata header happens to use the same INT metadata header format with the following field values. Again, note this is only for example; we do not propose any designs for the piggybacked metadata format other than that this is out of the scope of this document and that we should reserve a special option type for this kind of data (i.e., INT Source-to-sink data).
  - Ver = 0
  - Rep = 0 (No replication)
  - C = 0
  - E = 0 (Max Hop Count not exceeded)
  - Per-hop Metadata Length = 1 (for queue occupancy)

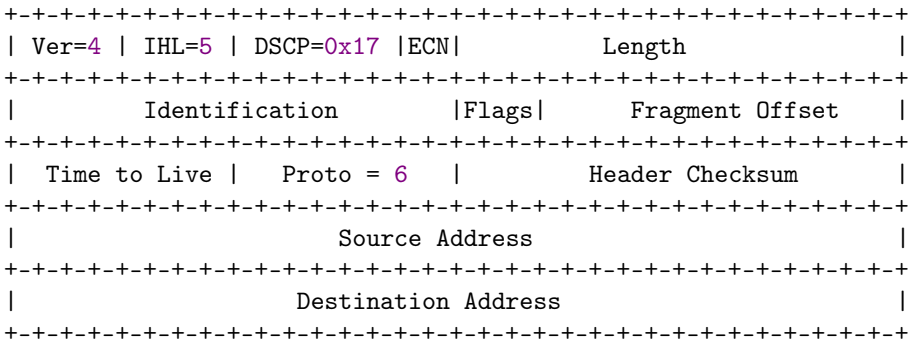
5.5. Example with INT over TCP

We consider a scenario where host1 sends a TCP packet to host2. The ToR switch of host1 (hop1) adds INT over TCP. Then an aggregate switch adds hop2 information and finally ToR switch of host2 (hop3) receives the packet and plays the role of INT sink.

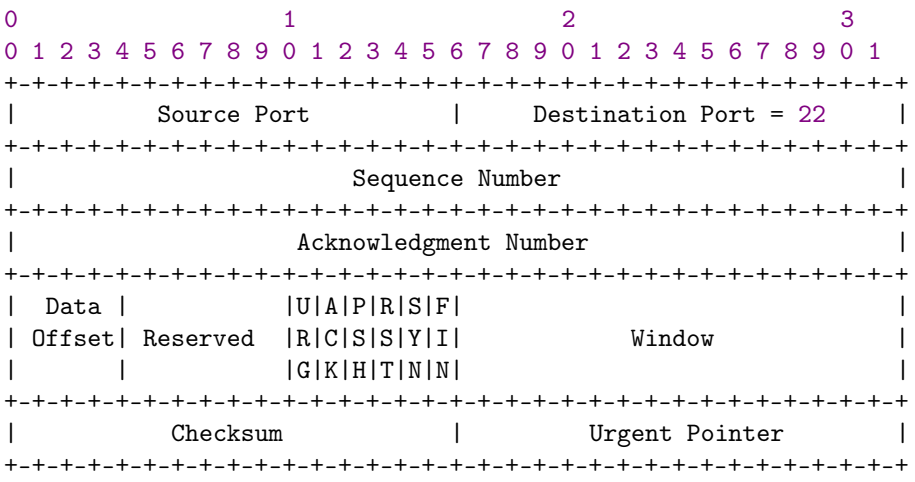
Below is the packet received by INT sink starting from the IPv4 header. We use the value of 0x17 for IPv4.DSCP to indicate the existence of INT headers. We omit the piggybacked metadata header in this example.

IP Header:

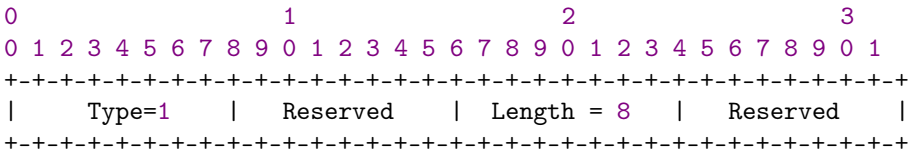
```
0                   1                   2                   3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
```



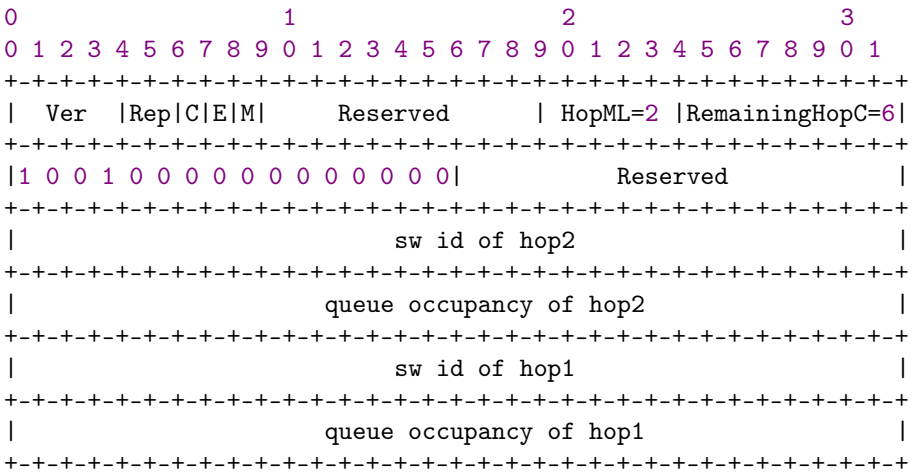
TCP Header:



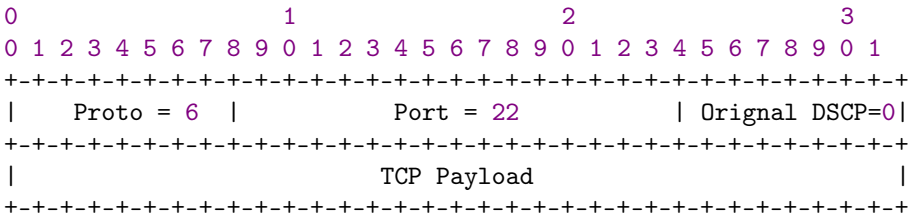
INT Shim Header for TCP/UDP, INT type is hop-by-hop:



INT Metadata Header and Metadata Stack:



INT Tail Header, followed by TCP Payload:

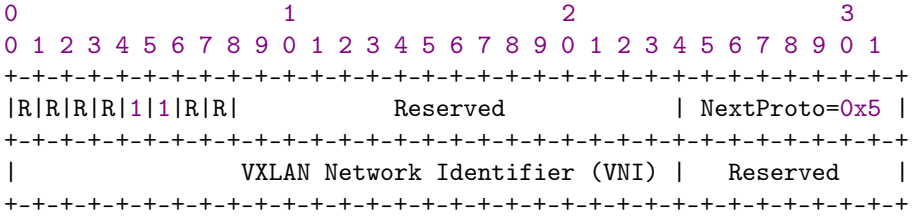


5.6. Example with VXLAN GPE encapsulation

We now consider a scenario where Host1 and Host2 use VXLAN encapsulation, intermediate switches parse through VXLAN header and the INT shim between VXLAN header and encapsulated payload and populate the INT metadata. The hop-by-hop INT stack is followed by a destination type INT stack carrying the piggybacked metadata.

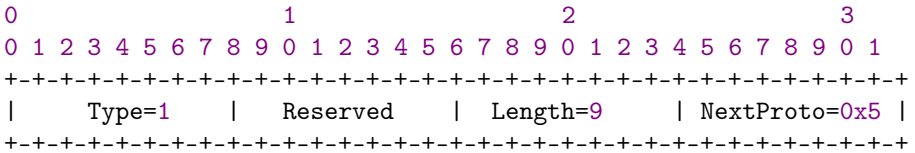
The packet headers received at Host 2 are as follows, starting with the VXLAN header (encapsulating ethernet, IP and UDP headers are not shown here):

VXLAN GPE Header:

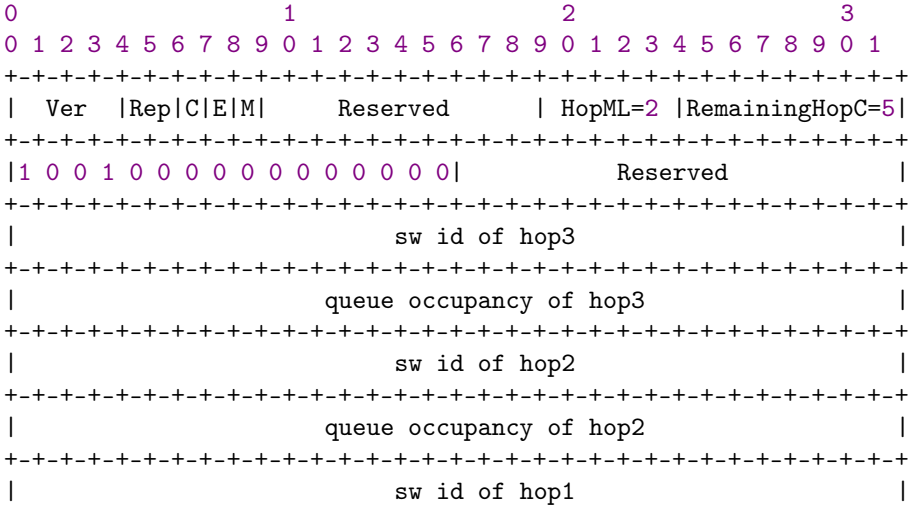


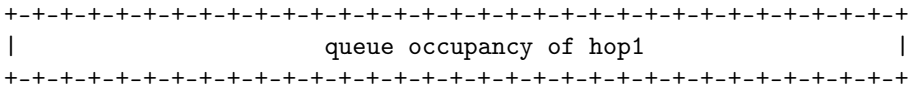
[INT Metadata - info about the path from Host1 to Host2]

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

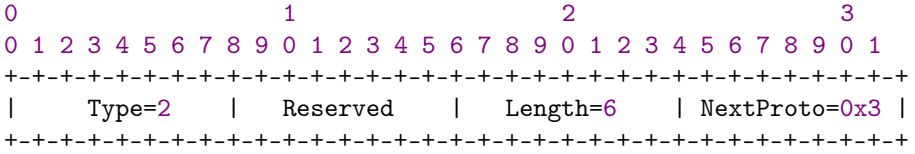


INT Metadata Header and Metadata Stack:

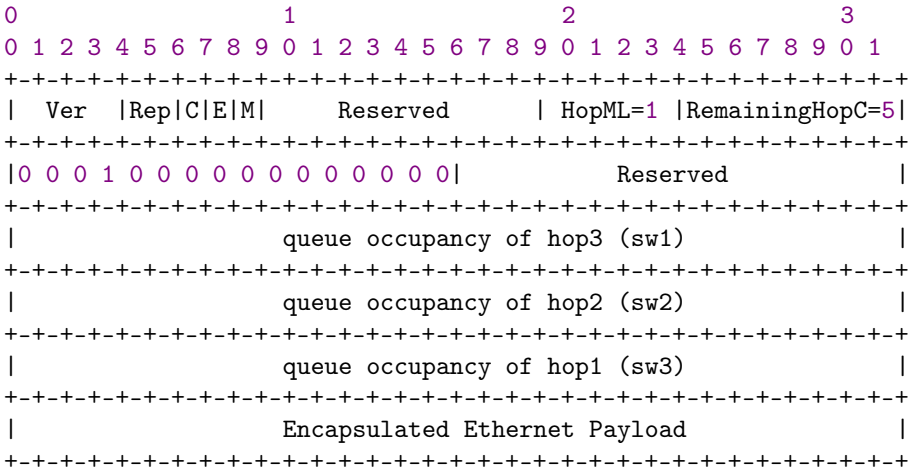




[Piggybacked metadata - info about the path from Host2 to Host1]
INT Shim Header for VXLAN-GPE, destination INT type:



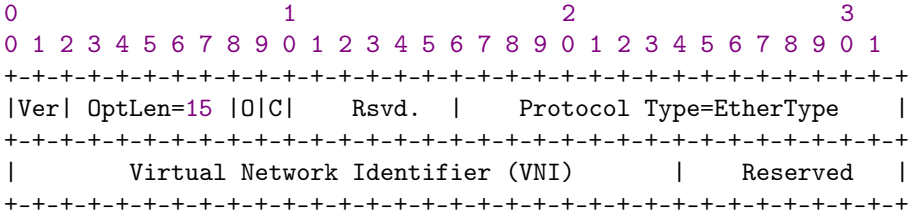
INT Metadata Header and Metadata Stack, followed by encapsulated Ethernet payload:



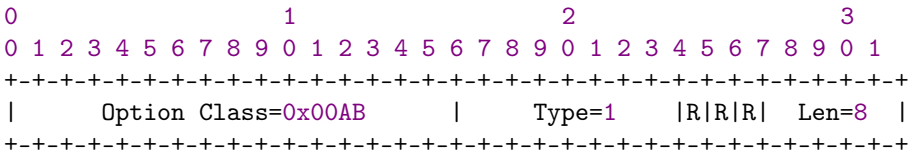
5.7. Example with Geneve encapsulation

We consider a scenario where Host1 and Host2 are using Geneve encapsulation and the intermediate switches parse the Geneve headers and populate INT metadata. We assume Geneve option class of 0x00AB for INT.

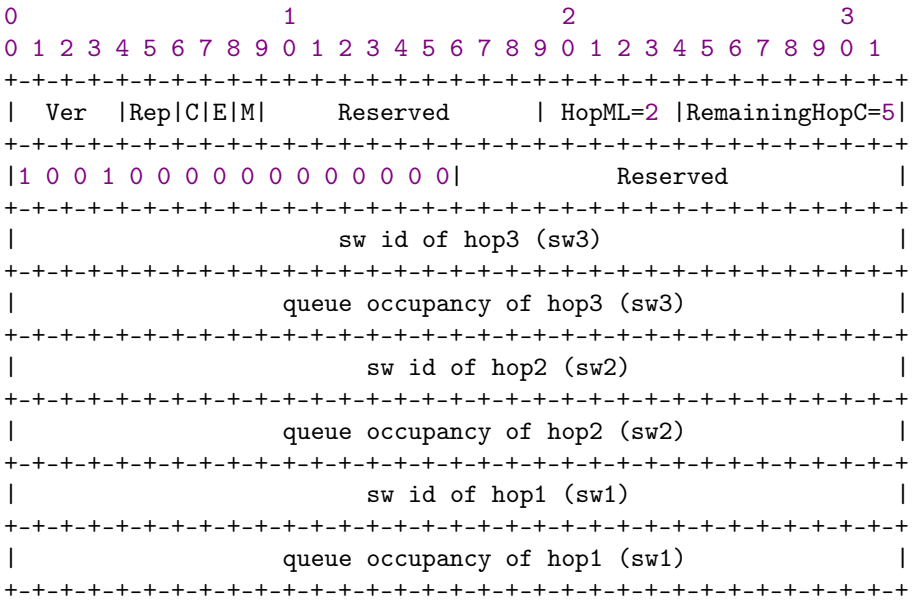
The following is the Geneve and INT Headers attached to the packet received by Host2.
Geneve Header:



[INT Metadata - info about the path from Host1 to Host2]
Geneve Option for INT, hop-by-hop type:

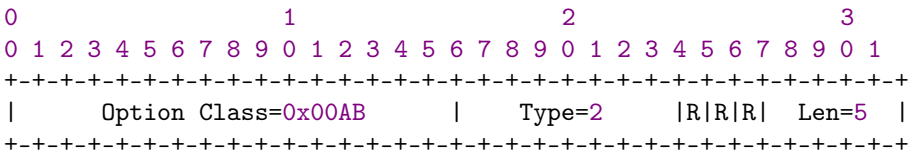


INT Metadata Header and Metadata Stack:

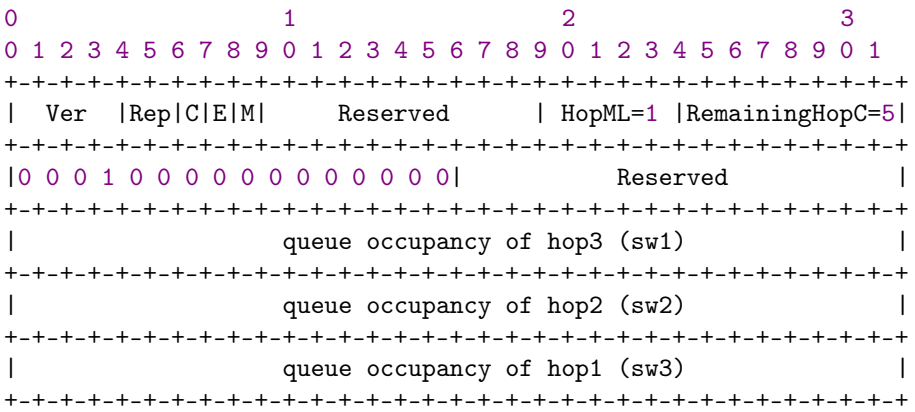


[Piggybacked metadata - info about the path from Host2 to Host1]

Geneve Option for INT, destination 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, and the match-action tables implementing the INT Transit behaviors. The program is written in P4_16 on v1model architecture, and it assumes INT over TCP as the encapsulation protocol.

This program is work in progress.

```
/*************************************************************************
 * header.p4
 *************************************************************************/

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

/* INT tail header for TCP/UDP */
header intl4_tail_t {
    bit<8>  next_proto;
    bit<16> dest_port;
    bit<8>  dscp;
}

/* INT headers */
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;  /* split the bits for lookup */
    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_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_q_congestion_t {
    bit<8>  q_id;
    bit<24> q_congestion;
}

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;
}

struct headers {
    ethernet_t              ethernet;
    ipv4_t                  ipv4;
    tcp_t                   tcp;
    intl4_shim_t            intl4_shim;
    int_header_t            int_header;
    int_switch_id_t         int_switch_id;
    int_port_ids_t          int_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_q_congestion_t      int_q_congestion;
    int_egress_port_tx_util_t int_egress_port_tx_util;
}

/* 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 metadata {
    int_metadata_t   int_metadata;
}


/*************************************************************************
 * parser.p4
 *************************************************************************/

/* indicate INT at LSB of DSCP */
const bit<6> DSCP_INT = 0x1;

parser ParserImpl(packet_in packet, out headers hdr, inout metadata meta,
                  inout standard_metadata_t standard_metadata) {
    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);
        transition select(hdr.ipv4.protocol) {
            8w0x6: parse_tcp;
            default: accept;
        }
    }
    state parse_tcp {
        packet.extract(hdr.tcp);
        transition select(hdr.ipv4.dscp) {
            DSCP_INT &&& DSCP_INT: parse_intl4_shim;
            default: accept;
        }
    }
    state parse_intl4_shim {
        packet.extract(hdr.intl4_shim);
        transition parse_int_header;
    }
    state parse_int_header {
        packet.extract(hdr.int_header);
        transition accept;
    }
}

control DeparserImpl(packet_out packet, in headers hdr) {
    apply {
        packet.emit(hdr.ethernet);
        packet.emit(hdr.ipv4);
        packet.emit(hdr.tcp);
        packet.emit(hdr.intl4_shim);
        packet.emit(hdr.int_header);
        packet.emit(hdr.int_switch_id);
        packet.emit(hdr.int_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_q_congestion);
        packet.emit(hdr.int_egress_port_tx_util);
    }
}

control VerifyChecksumImpl(in headers hdr, inout metadata meta) {
    Checksum16() ipv4_checksum;
    apply {
        if (hdr.ipv4.hdrChecksum == ipv4_checksum.get(
          {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}))
             mark_to_drop();
    }
}

control ComputeChecksumImpl(inout headers hdr, inout metadata meta) {
    Checksum16() ipv4_checksum;
    apply {
        hdr.ipv4.hdrChecksum == ipv4_checksum.get(
          {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});
    }
}


/*************************************************************************
 * int_transit.p4: tables, actions, and control flow
 *************************************************************************/

#include <core.p4>
#include <v1model.p4>

#include "header.p4"
#include "parser.p4"

control Int_metadata_insert(inout headers hdr,
        in int_metadata_t int_metadata,
        inout standard_metadata_t standard_metadata)
{
    /* 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_port_ids.setValid();
        hdr.int_port_ids.ingress_port_id =
            (bit<16>) standard_metadata.ingress_port;
        hdr.int_port_ids.egress_port_id =
            (bit<16>) standard_metadata.egress_port;
    }
    action int_set_header_2() {
        hdr.int_hop_latency.setValid();
        hdr.int_hop_latency.hop_latency =
            (bit<32>) standard_metadata.deq_timedelta;
    }
    action int_set_header_3() {
        hdr.int_q_occupancy.setValid();
        hdr.int_q_occupancy.q_id =
            (bit<8>) standard_metadata.egress_qid;
        hdr.int_q_occupancy.q_occupancy =
            (bit<24>) standard_metadata.deq_qdepth;
    }

    /* 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();
        }
    }
}

/* TBD - Check egress link MTU, do not insert any metadata and
   set M bit if adding metadata will cause egress MTU to be exceeded */
control Int_egress(inout headers hdr, inout metadata meta,
        inout standard_metadata_t standard_metadata)
{
    action int_transit(bit<32> switch_id) {
        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;
    }
    table int_prep {
```

```
        key = {}
        actions = {int_transit;}
    }

    Int_metadata_insert() int_metadata_insert;
    Int_outer_encap() int_outer_encap;

    action int_hop_cnt_decrement() {
        hdr.int_header.remaining_hop_cnt =
            hdr.int_header.remaining_hop_cnt - 1;
    }
    action int_hop_cnt_exceeded() {
        hdr.int_header.e = 1;
    }

    apply{
        if(hdr.int_header.isValid()) {
            if(hdr.int_header.remaining_hop_cnt != 0
                    && hdr.int_header.e == 0) {
                int_hop_cnt_decrement();
                int_prep.apply();
                int_metadata_insert.apply(
                    hdr, meta.int_metadata, standard_metadata);
                int_outer_encap.apply(hdr, meta.int_metadata);
            } else {
                int_hop_cnt_exceeded();
            }
        }
    }
}

control IngressImpl(inout headers hdr, inout metadata meta,
        inout standard_metadata_t standard_metadata)
{
    apply{
        /* fill in */
    }
}

control EgressImpl(inout headers hdr, inout metadata meta,
        inout standard_metadata_t standard_metadata)
{
    Int_egress() int_egress;
    apply{
        // snip
        int_egress.apply(hdr, meta, standard_metadata);
        // snip
    }
}

V1Switch(ParserImpl(),
         VerifyChecksumImpl(),
```

```
        IngressImpl(),
        EgressImpl(),
        ComputeChecksumImpl(),
        DeparserImpl()) main

/* End of Code snippet */
```

‘

‘

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

A.1. Switch-level Information

- 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 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. 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 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. 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
  - Device local time capturing when the INT packet leaves the egress 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.

- Congestion status
  - The ratio of the current queue length to the configured maximum queue limit. This value is used primarily to determine how much space is left in the queue.

- 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
- Anoop Ghanwani
- Mukesh Hira
- Hugh Holbrook
- Changhoon Kim
- Jeongkeun Lee
- Tal Mizrahi
- Masoud Moshref
- Heidi OU
- Mickey Spiegel

C. Change log

- 2015-09-28
  - Initial release

- 2016-06-19
  - Updated section 5.3.2, the Length field definition of VXLAN GPE shim header, to be consistent with the example in section 5.4.

- 2017-10-17
  - Introduced INT over TCP/UDP (section 5.3.1 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 5.3.4)

  - 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 5.4
  - Formatted the spec as a Madoko file

- 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 5.3.1).

- 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 5.3.1.
  - Relaxed the location of INT stack relative to TCP options in section 5.3.1.

- 2018-03-14
  - Added the Checksum Complement metadata.