

Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: April 28, 2015

J. Gross
T. Sridhar
VMware
P. Garg
Microsoft
C. Wright
Red Hat
I. Ganga
Intel
P. Agarwal
Broadcom
K. Duda
Arista
D. Dutt
Cumulus
J. Hudson
Brocade
October 25, 2014

Geneve: Generic Network Virtualization Encapsulation
draft-gross-geneve-02

Abstract

Network virtualization involves the cooperation of devices with a wide variety of capabilities such as software and hardware tunnel endpoints, transit fabrics, and centralized control clusters. As a result of their role in tying together different elements in the system, the requirements on tunnels are influenced by all of these components. Flexibility is therefore the most important aspect of a tunnel protocol if it is to keep pace with the evolution of the system. This draft describes Geneve, a protocol designed to recognize and accommodate these changing capabilities and needs.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <http://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any

time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on April 28, 2015.

Copyright Notice

Copyright (c) 2014 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	3
1.1. Requirements Language	4
1.2. Terminology	4
2. Design Requirements	5
2.1. Control Plane Independence	6
2.2. Data Plane Extensibility	7
2.2.1. Efficient Implementation	7
2.3. Use of Standard IP Fabrics	8
3. Geneve Encapsulation Details	9
3.1. Geneve Frame Format Over IPv4	9
3.2. Geneve Frame Format Over IPv6	10
3.3. UDP Header	12
3.4. Tunnel Header Fields	13
3.5. Tunnel Options	14
3.5.1. Options Processing	16
4. Implementation and Deployment Considerations	16
4.1. Encapsulation of Geneve in IP	16
4.1.1. IP Fragmentation	16
4.1.2. DSCP and ECN	17
4.1.3. Broadcast and Multicast	17
4.2. NIC Offloads	18
4.3. Inner VLAN Handling	18
5. Interoperability Issues	19
6. Security Considerations	19
7. IANA Considerations	20
8. Acknowledgements	20

9. References	20
9.1. Normative References	20
9.2. Informative References	21
Authors' Addresses	22

1. Introduction

Networking has long featured a variety of tunneling, tagging, and other encapsulation mechanisms. However, the advent of network virtualization has caused a surge of renewed interest and a corresponding increase in the introduction of new protocols. The large number of protocols in this space, ranging all the way from VLANs [IEEE.802.1Q-2011] and MPLS [RFC3031] through the more recent VXLAN [RFC7348], NVGRE [I-D.sridharan-virtualization-nvgre], and STT [I-D.davie-stt], often leads to questions about the need for new encapsulation formats and what it is about network virtualization in particular that leads to their proliferation.

While many encapsulation protocols seek to simply partition the underlay network or bridge between two domains, network virtualization views the transit network as providing connectivity between multiple components of an integrated system. In many ways this system is similar to a chassis switch with the IP underlay network playing the role of the backplane and tunnel endpoints on the edge as line cards. When viewed in this light, the requirements placed on the tunnel protocol are significantly different in terms of the quantity of metadata necessary and the role of transit nodes.

Current work such as [VL2] and the NVO3 working group [I-D.ietf-nvo3-dataplane-requirements] have described some of the properties that the data plane must have to support network virtualization. However, one additional defining requirement is the need to carry system state along with the packet data. The use of some metadata is certainly not a foreign concept - nearly all protocols used for virtualization have at least 24 bits of identifier space as a way to partition between tenants. This is often described as overcoming the limits of 12-bit VLANs, and when seen in that context, or any context where it is a true tenant identifier, 16 million possible entries is a large number. However, the reality is that the metadata is not exclusively used to identify tenants and encoding other information quickly starts to crowd the space. In fact, when compared to the tags used to exchange metadata between line cards on a chassis switch, 24-bit identifiers start to look quite small. There are nearly endless uses for this metadata, ranging from storing input ports for simple security policies to service based context for interposing advanced middleboxes.

Existing tunnel protocols have each attempted to solve different aspects of these new requirements, only to be quickly rendered out of date by changing control plane implementations and advancements. Furthermore, software and hardware components and controllers all have different advantages and rates of evolution - a fact that should be viewed as a benefit, not a liability or limitation. This draft describes Geneve, a protocol which seeks to avoid these problems by providing a framework for tunneling for network virtualization rather than being prescriptive about the entire system.

1.1. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

In this document, these words will appear with that interpretation only when in ALL CAPS. Lower case uses of these words are not to be interpreted as carrying RFC-2119 significance.

1.2. Terminology

The NVO3 framework [RFC7365] defines many of the concepts commonly used in network virtualization. In addition, the following terms are specifically meaningful in this document:

Checksum offload. An optimization implemented by many NICs which enables computation and verification of upper layer protocol checksums in hardware on transmit and receive, respectively. This typically includes IP and TCP/UDP checksums which would otherwise be computed by the protocol stack in software.

Clos network. A technique for composing network fabrics larger than a single switch while maintaining non-blocking bandwidth across connection points. ECMP is used to divide traffic across the multiple links and switches that constitute the fabric. Sometimes termed "leaf and spine" or "fat tree" topologies.

ECMP. Equal Cost Multipath. A routing mechanism for selecting from among multiple best next hop paths by hashing packet headers in order to better utilize network bandwidth while avoiding reordering a single stream.

Geneve. Generic Network Virtualization Encapsulation. The tunnel protocol described in this draft.

LRO. Large Receive Offload. The receive-side equivalent function of LSO, in which multiple protocol segments (primarily TCP) are coalesced into larger data units.

NIC. Network Interface Card. A NIC could be part of a tunnel endpoint or transit device and can either process Geneve packets or aid in the processing of Geneve packets.

OAM. Operations, Administration, and Management. A suite of tools used to monitor and troubleshoot network problems.

Transit device. A forwarding element along the path of the tunnel making up part of the Underlay Network. A transit device MAY be capable of understanding the Geneve frame format but does not originate or terminate Geneve packets.

LSO. Large Segmentation Offload. A function provided by many commercial NICs that allows data units larger than the MTU to be passed to the NIC to improve performance, the NIC being responsible for creating smaller segments with correct protocol headers. When referring specifically to TCP/IP, this feature is often known as TSO (TCP Segmentation Offload).

Tunnel endpoint. A component encapsulating packets, such as Ethernet frames or IP datagrams, in Geneve headers and vice versa. As the ultimate consumer of any tunnel metadata, endpoints have the highest level of requirements for parsing and interpreting tunnel headers. Tunnel endpoints may consist of either software or hardware implementations or a combination of the two. Endpoints are frequently a component of an NVE but may also be found in middleboxes or other elements making up an NVO3 Network.

VM. Virtual Machine.

2. Design Requirements

Geneve is designed to support network virtualization use cases, where tunnels are typically established to act as a backplane between the virtual switches residing in hypervisors, physical switches, or middleboxes or other appliances. An arbitrary IP network can be used as an underlay although Clos networks composed using ECMP links are a common choice to provide consistent bisectional bandwidth across all connection points. Figure 1 shows an example of a hypervisor, top of rack switch for connectivity to physical servers, and a WAN uplink connected using Geneve tunnels over a simplified Clos network. These tunnels are used to encapsulate and forward frames from the attached components such as VMs or physical links.

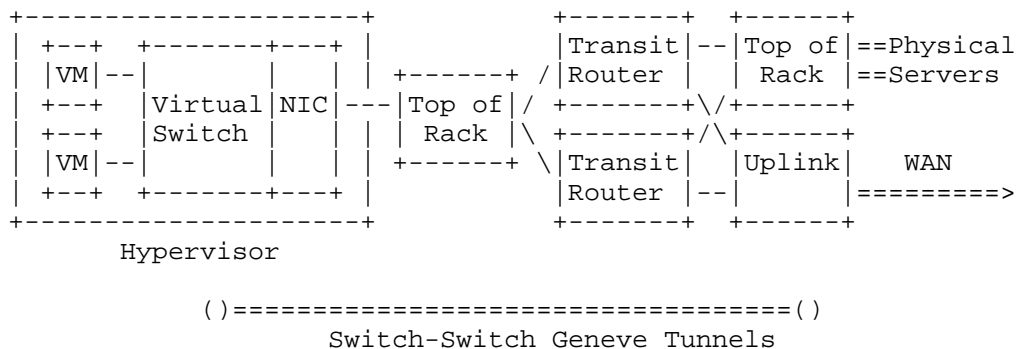


Figure 1: Sample Geneve Deployment

To support the needs of network virtualization, the tunnel protocol should be able to take advantage of the differing (and evolving) capabilities of each type of device in both the underlay and overlay networks. This results in the following requirements being placed on the data plane tunneling protocol:

- o The data plane is generic and extensible enough to support current and future control planes.
- o Tunnel components are efficiently implementable in both hardware and software without restricting capabilities to the lowest common denominator.
- o High performance over existing IP fabrics.

These requirements are described further in the following subsections.

2.1. Control Plane Independence

Although some protocols for network virtualization have included a control plane as part of the tunnel format specification (most notably, the original VXLAN spec prescribed a multicast learning-based control plane), these specifications have largely been treated as describing only the data format. The VXLAN frame format has actually seen a wide variety of control planes built on top of it.

There is a clear advantage in settling on a data format: most of the protocols are only superficially different and there is little advantage in duplicating effort. However, the same cannot be said of control planes, which are diverse in very fundamental ways. The case for standardization is also less clear given the wide variety in requirements, goals, and deployment scenarios.

As a result of this reality, Geneve aims to be a pure tunnel format specification that is capable of fulfilling the needs of many control planes by explicitly not selecting any one of them. This simultaneously promotes a shared data format and increases the chances that it will not be obsoleted by future control plane enhancements.

2.2. Data Plane Extensibility

Achieving the level of flexibility needed to support current and future control planes effectively requires an options infrastructure to allow new metadata types to be defined, deployed, and either finalized or retired. Options also allow for differentiation of products by encouraging independent development in each vendor's core specialty, leading to an overall faster pace of advancement. By far the most common mechanism for implementing options is Type-Length-Value (TLV) format.

It should be noted that while options can be used to support non-wirespeed control frames, they are equally important on data frames as well to segregate and direct forwarding (for instance, the examples given before of input port based security policies and service interposition both require tags to be placed on data packets). Therefore, while it would be desirable to limit the extensibility to only control frames for the purposes of simplifying the datapath, that would not satisfy the design requirements.

2.2.1. Efficient Implementation

There is often a conflict between software flexibility and hardware performance that is difficult to resolve. For a given set of functionality, it is obviously desirable to maximize performance. However, that does not mean new features that cannot be run at that speed today should be disallowed. Therefore, for a protocol to be efficiently implementable means that a set of common capabilities can be reasonably handled across platforms along with a graceful mechanism to handle more advanced features in the appropriate situations.

The use of a variable length header and options in a protocol often raises questions about whether it is truly efficiently implementable in hardware. To answer this question in the context of Geneve, it is important to first divide "hardware" into two categories: tunnel endpoints and transit devices.

Endpoints must be able to parse the variable header, including any options, and take action. Since these devices are actively participating in the protocol, they are the most affected by Geneve.

However, as endpoints are the ultimate consumers of the data, transmitters can tailor their output to the capabilities of the recipient. As new functionality becomes sufficiently well defined to add to endpoints, supporting options can be designed using ordering restrictions and other techniques to ease parsing.

Transit devices MAY be able to interpret the options and participate in Geneve packet processing. However, as non-terminating devices, they do not originate or terminate the Geneve packet. The participation of transit devices in Geneve packet processing is OPTIONAL.

Further, either tunnel endpoints or transit devices MAY use offload capabilities of NICs such as checksum offload to improve the performance of Geneve packet processing. The presence of a Geneve variable length header SHOULD NOT prevent the tunnel endpoints and transit devices from using such offload capabilities.

2.3. Use of Standard IP Fabrics

IP has clearly cemented its place as the dominant transport mechanism and many techniques have evolved over time to make it robust, efficient, and inexpensive. As a result, it is natural to use IP fabrics as a transit network for Geneve. Fortunately, the use of IP encapsulation and addressing is enough to achieve the primary goal of delivering packets to the correct point in the network through standard switching and routing.

In addition, nearly all underlay fabrics are designed to exploit parallelism in traffic to spread load across multiple links without introducing reordering in individual flows. These equal cost multipathing (ECMP) techniques typically involve parsing and hashing the addresses and port numbers from the packet to select an outgoing link. However, the use of tunnels often results in poor ECMP performance without additional knowledge of the protocol as the encapsulated traffic is hidden from the fabric by design and only endpoint addresses are available for hashing.

Since it is desirable for Geneve to perform well on these existing fabrics, it is necessary for entropy from encapsulated packets to be exposed in the tunnel header. The most common technique for this is to use the UDP source port, which is discussed further in Section 3.3.

3. Geneve Encapsulation Details

The Geneve frame format consists of a compact tunnel header encapsulated in UDP over either IPv4 or IPv6. A small fixed tunnel header provides control information plus a base level of functionality and interoperability with a focus on simplicity. This header is then followed by a set of variable options to allow for future innovation. Finally, the payload consists of a protocol data unit of the indicated type, such as an Ethernet frame. The following subsections provide examples of Geneve frames transported (for example) over Ethernet along with an Ethernet payload.

3.1. Geneve Frame Format Over IPv4

```

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

```

Outer Ethernet Header:

```

+++++
|                                     Outer Destination MAC Address                                     |
+++++
| Outer Destination MAC Address | Outer Source MAC Address |
+++++
|                                     Outer Source MAC Address                                     |
+++++
| Optional Ethertype=C-Tag 802.1Q | Outer VLAN Tag Information |
+++++
|                                     Ethertype=0x0800                                     |
+++++

```

Outer IPv4 Header:

```

+++++
|Version| IHL |Type of Service| Total Length |
+++++
| Identification | Flags | Fragment Offset |
+++++
| Time to Live | Protocol=17 UDP | Header Checksum |
+++++
|                                     Outer Source IPv4 Address                                     |
+++++
|                                     Outer Destination IPv4 Address                               |
+++++

```

Outer UDP Header:

```

+-----+-----+-----+-----+-----+-----+-----+-----+
|           Source Port = xxxx           |           Dest Port = 6081           |
+-----+-----+-----+-----+-----+-----+-----+-----+
|           UDP Length           |           UDP Checksum           |
+-----+-----+-----+-----+-----+-----+-----+-----+

```

Geneve Header:

```

+-----+-----+-----+-----+-----+-----+-----+-----+
| Ver | Opt Len | O | C |      Rsvd.      |           Protocol Type           |
+-----+-----+-----+-----+-----+-----+-----+-----+
|           Virtual Network Identifier (VNI)           |           Reserved           |
+-----+-----+-----+-----+-----+-----+-----+-----+
|           Variable Length Options           |
+-----+-----+-----+-----+-----+-----+-----+-----+

```

Inner Ethernet Header (example payload):

```

+-----+-----+-----+-----+-----+-----+-----+-----+
|           Inner Destination MAC Address           |
+-----+-----+-----+-----+-----+-----+-----+-----+
| Inner Destination MAC Address | Inner Source MAC Address |
+-----+-----+-----+-----+-----+-----+-----+-----+
|           Inner Source MAC Address           |
+-----+-----+-----+-----+-----+-----+-----+-----+
| Optional Ethertype=C-Tag 802.1Q | Inner VLAN Tag Information |
+-----+-----+-----+-----+-----+-----+-----+-----+

```

Payload:

```

+-----+-----+-----+-----+-----+-----+-----+-----+
| Ethertype of Original Payload |
+-----+-----+-----+-----+-----+-----+-----+-----+
|           Original Ethernet Payload           |
+-----+-----+-----+-----+-----+-----+-----+-----+
| (Note that the original Ethernet Frame's FCS is not included) |
+-----+-----+-----+-----+-----+-----+-----+-----+

```

Frame Check Sequence:

```

+-----+-----+-----+-----+-----+-----+-----+-----+
|           New FCS (Frame Check Sequence) for Outer Ethernet Frame           |
+-----+-----+-----+-----+-----+-----+-----+-----+

```

3.2. Geneve Frame Format Over IPv6

```

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

```

Outer Ethernet Header:

```

+-----+
|                                     Outer Destination MAC Address                                     |
+-----+
| Outer Destination MAC Address | Outer Source MAC Address |
+-----+
|                                     Outer Source MAC Address                                     |
+-----+
| Optional Ethertype=C-Tag 802.1Q | Outer VLAN Tag Information |
+-----+
|                               Ethertype=0x86DD                               |
+-----+

```

Outer IPv6 Header:

```

+-----+
| Version | Traffic Class | Flow Label |
+-----+
| Payload Length | NxtHdr=17 UDP | Hop Limit |
+-----+
|
+
|
+
|                                     Outer Source IPv6 Address                                     |
+
|
+-----+
|
+
|                                     Outer Destination IPv6 Address                               |
+
|
+-----+

```

Outer UDP Header:

```

+-----+
| Source Port = xxxx | Dest Port = 6081 |
+-----+
| UDP Length | UDP Checksum |
+-----+

```

Geneve Header:

```

+-----+
|Ver|  Opt Len  |O|C|    Rsvd.   |          Protocol Type          |
+-----+
|          Virtual Network Identifier (VNI)          |      Reserved      |
+-----+
|          Variable Length Options          |
+-----+

```

Inner Ethernet Header (example payload):

```

+-----+
|          Inner Destination MAC Address          |
+-----+
| Inner Destination MAC Address | Inner Source MAC Address |
+-----+
|          Inner Source MAC Address          |
+-----+
|Optional Ethertype=C-Tag 802.1Q| Inner VLAN Tag Information |
+-----+

```

Payload:

```

+-----+
| Ethertype of Original Payload |
+-----+
|          Original Ethernet Payload          |
|
| (Note that the original Ethernet Frame's FCS is not included) |
+-----+

```

Frame Check Sequence:

```

+-----+
| New FCS (Frame Check Sequence) for Outer Ethernet Frame |
+-----+

```

3.3. UDP Header

The use of an encapsulating UDP [RFC0768] header follows the connectionless semantics of Ethernet and IP in addition to providing entropy to routers performing ECMP. The header fields are therefore interpreted as follows:

Source port: A source port selected by the ingress tunnel endpoint. This source port SHOULD be the same for all packets belonging to a single encapsulated flow to prevent reordering due to the use of different paths. To encourage an even distribution of flows across multiple links, the source port SHOULD be calculated using a hash of the encapsulated packet headers using, for example, a traditional 5-tuple. Since the port represents a flow identifier

rather than a true UDP connection, the entire 16-bit range MAY be used to maximize entropy.

Dest port: IANA has assigned port 6081 as the fixed well-known destination port for Geneve. This port MUST be used in both directions of a flow. Although the well-known value should be used by default, it is RECOMMENDED that implementations make this configurable.

UDP length: The length of the UDP packet including the UDP header.

UDP checksum: The checksum MAY be set to zero on transmit for packets encapsulated in both IPv4 and IPv6 [RFC6935]. When a packet is received with a UDP checksum of zero it MUST be accepted and decapsulated. If the ingress tunnel endpoint optionally encapsulates a packet with a non-zero checksum, it MUST be a correctly computed UDP checksum. Upon receiving such a packet, the egress endpoint MUST validate the checksum. If the checksum is not correct, the packet MUST be dropped, otherwise the packet MUST be accepted for decapsulation. It is RECOMMENDED that the UDP checksum be computed to protect the Geneve header and options in situations where the network reliability is not high and the packet is not protected by another checksum or CRC.

3.4. Tunnel Header Fields

Ver (2 bits): The current version number is 0. Packets received by an endpoint with an unknown version MUST be dropped. Non-terminating devices processing Geneve packets with an unknown version number MUST treat them as UDP packets with an unknown payload.

Opt Len (6 bits): The length of the options fields, expressed in four byte multiples, not including the eight byte fixed tunnel header. This results in a minimum total Geneve header size of 8 bytes and a maximum of 260 bytes. The start of the payload headers can be found using this offset from the end of the base Geneve header.

O (1 bit): OAM frame. This packet contains a control message instead of a data payload. Endpoints MUST NOT forward the payload and transit devices MUST NOT attempt to interpret or process it. Since these are infrequent control messages, it is RECOMMENDED that endpoints direct these packets to a high priority control queue (for example, to direct the packet to a general purpose CPU from a forwarding ASIC or to separate out control traffic on a NIC). Transit devices MUST NOT alter forwarding behavior on the basis of this bit, such as ECMP link selection.

C (1 bit): Critical options present. One or more options has the critical bit set (see Section 3.5). If this bit is set then tunnel endpoints MUST parse the options list to interpret any critical options. On devices where option parsing is not supported the frame MUST be dropped on the basis of the 'C' bit in the base header. If the bit is not set tunnel endpoints MAY strip all options using 'Opt Len' and forward the decapsulated frame. Transit devices MUST NOT drop or modify packets on the basis of this bit.

Rsvd. (6 bits): Reserved field which MUST be zero on transmission and ignored on receipt.

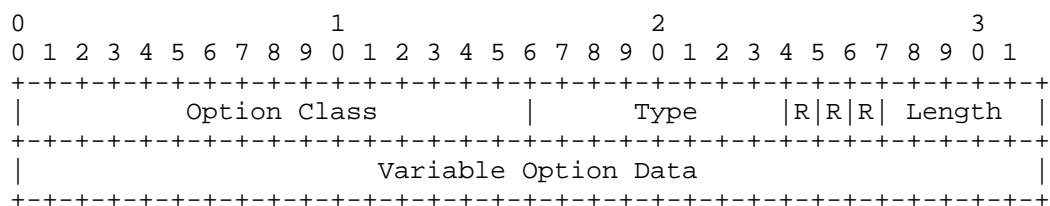
Protocol Type (16 bits): The type of the protocol data unit appearing after the Geneve header. This follows the EtherType [ETYPES] convention with Ethernet itself being represented by the value 0x6558.

Virtual Network Identifier (VNI) (24 bits): An identifier for a unique element of a virtual network. In many situations this may represent an L2 segment, however, the control plane defines the forwarding semantics of decapsulated packets. The VNI MAY be used as part of ECMP forwarding decisions or MAY be used as a mechanism to distinguish between overlapping address spaces contained in the encapsulated packet when load balancing across CPUs.

Reserved (8 bits): Reserved field which MUST be zero on transmission and ignored on receipt.

Transit devices MUST maintain consistent forwarding behavior irrespective of the value of 'Opt Len', including ECMP link selection. These devices SHOULD be able to forward packets containing options without resorting to a slow path.

3.5. Tunnel Options



Geneve Option

The base Geneve header is followed by zero or more options in Type-Length-Value format. Each option consists of a four byte option

header and a variable amount of option data interpreted according to the type.

Option Class (16 bits): Namespace for the 'Type' field. IANA will be requested to create a "Geneve Option Class" registry to allocate identifiers for organizations, technologies, and vendors that have an interest in creating types for options. Each organization may allocate types independently to allow experimentation and rapid innovation. It is expected that over time certain options will become well known and a given implementation may use option types from a variety of sources. In addition, IANA will be requested to reserve specific ranges for standardized and experimental options.

Type (8 bits): Type indicating the format of the data contained in this option. Options are primarily designed to encourage future extensibility and innovation and so standardized forms of these options will be defined in a separate document.

The high order bit of the option type indicates that this is a critical option. If the receiving endpoint does not recognize this option and this bit is set then the frame MUST be dropped. If the critical bit is set in any option then the 'C' bit in the Geneve base header MUST also be set. Transit devices MUST NOT drop packets on the basis of this bit. The following figure shows the location of the 'C' bit in the 'Type' field:

```

0 1 2 3 4 5 6 7 8
+---+---+---+---+---+
|C|      Type      |
+---+---+---+---+---+

```

The requirement to drop a packet with an unknown critical option applies to the entire tunnel endpoint system and not a particular component of the implementation. For example, in a system comprised of a forwarding ASIC and a general purpose CPU, this does not mean that the packet must be dropped in the ASIC. An implementation may send the packet to the CPU using a rate-limited control channel for slow-path exception handling.

R (3 bits): Option control flags reserved for future use. MUST be zero on transmission and ignored on receipt.

Length (5 bits): Length of the option, expressed in four byte multiples excluding the option header. The total length of each option may be between 4 and 128 bytes. Packets in which the total length of all options is not equal to the 'Opt Len' in the base

header are invalid and MUST be silently dropped if received by an endpoint.

Variable Option Data: Option data interpreted according to 'Type'.

3.5.1. Options Processing

Geneve options are intended to be originated and processed by tunnel endpoints. Options MAY be processed by transit devices along the tunnel path as well. This document only details the handling of options by tunnel endpoints. A future version of this document will provide details of options processing by transit devices. Transit devices not processing Geneve options SHOULD process Geneve frame as any other UDP frame and maintain consistent forwarding behavior.

In tunnel endpoints, the generation and interpretation of options is determined by the control plane, which is out of the scope of this document. However, to ensure interoperability between heterogeneous devices two requirements are imposed on endpoint devices:

- o Receiving endpoints MUST drop packets containing unknown options with the 'C' bit set in the option type.
- o Sending endpoints MUST NOT assume that options will be processed sequentially by the receiver in the order they were transmitted.

4. Implementation and Deployment Considerations

4.1. Encapsulation of Geneve in IP

As an IP-based tunnel protocol, Geneve shares many properties and techniques with existing protocols. The application of some of these are described in further detail, although in general most concepts applicable to the IP layer or to IP tunnels generally also function in the context of Geneve.

4.1.1. IP Fragmentation

To prevent fragmentation and maximize performance, the best practice when using Geneve is to ensure that the MTU of the physical network is greater than or equal to the MTU of the encapsulated network plus tunnel headers. Manual or upper layer (such as TCP MSS clamping) configuration can be used to ensure that fragmentation never takes place, however, in some situations this may not be feasible.

It is strongly RECOMMENDED that Path MTU Discovery ([RFC1191], [RFC1981]) be used by setting the DF bit in the IP header when Geneve packets are transmitted over IPv4 (this is the default with IPv6).

The use of Path MTU Discovery on the transit network provides the encapsulating endpoint with soft-state about the link that it may use to prevent or minimize fragmentation depending on its role in the virtualized network.

Note that some implementations may not be capable of supporting fragmentation or other less common features of the IP header, such as options and extension headers.

4.1.2. DSCP and ECN

When encapsulating IP (including over Ethernet) frames in Geneve, there are several options for propagating DSCP and ECN bits from the inner header to the tunnel on transmission and the reverse on reception.

[RFC2983] lists considerations for mapping DSCP between inner and outer IP headers. Network virtualization is typically more closely aligned with the Pipe model described, where the DSCP value on the tunnel header is set based on a policy (which may be a fixed value, one based on the inner traffic class, or some other mechanism for grouping traffic). Aspects of the Uniform model (which treats the inner and outer DSCP value as a single field by copying on ingress and egress) may also apply, such as the ability to remark the inner header on tunnel egress based on transit marking. However, the Uniform model is not conceptually consistent with network virtualization, which seeks to provide strong isolation between encapsulated traffic and the physical network.

[RFC6040] describes the mechanism for exposing ECN capabilities on IP tunnels and propagating congestion markers to the inner packets. This behavior SHOULD be followed for IP packets encapsulated in Geneve.

4.1.3. Broadcast and Multicast

Geneve tunnels may either be point-to-point unicast between two endpoints or may utilize broadcast or multicast addressing. It is not required that inner and outer addressing match in this respect. For example, in physical networks that do not support multicast, encapsulated multicast traffic may be replicated into multiple unicast tunnels or forwarded by policy to a unicast location (possibly to be replicated there).

With physical networks that do support multicast it may be desirable to use this capability to take advantage of hardware replication for encapsulated packets. In this case, multicast addresses may be allocated in the physical network corresponding to tenants,

encapsulated multicast groups, or some other factor. The allocation of these groups is a component of the control plane and therefore outside of the scope of this document. When physical multicast is in use, the 'C' bit in the Geneve header may be used with groups of devices with heterogeneous capabilities as each device can interpret only the options that are significant to it if they are not critical.

4.2. NIC Offloads

Modern NICs currently provide a variety of offloads to enable the efficient processing of packets. The implementation of many of these offloads requires only that the encapsulated packet be easily parsed (for example, checksum offload). However, optimizations such as LSO and LRO involve some processing of the options themselves since they must be replicated/merged across multiple packets. In these situations, it is desirable to not require changes to the offload logic to handle the introduction of new options. To enable this, some constraints are placed on the definitions of options to allow for simple processing rules:

- o When performing LSO, a NIC MUST replicate the entire Geneve header and all options, including those unknown to the device, onto each resulting segment. However, a given option definition may override this rule and specify different behavior in supporting devices. Conversely, when performing LRO, a NIC MAY assume that a binary comparison of the options (including unknown options) is sufficient to ensure equality and MAY merge packets with equal Geneve headers.
- o Option ordering is not significant and packets with the same options in a different order MAY be processed alike.
- o NICs performing offloads MUST NOT drop packets with unknown options, including those marked as critical.

There is no requirement that a given implementation of Geneve employ the offloads listed as examples above. However, as these offloads are currently widely deployed in commercially available NICs, the rules described here are intended to enable efficient handling of current and future options across a variety of devices.

4.3. Inner VLAN Handling

Geneve is capable of encapsulating a wide range of protocols and therefore a given implementation is likely to support only a small subset of the possibilities. However, as Ethernet is expected to be widely deployed, it is useful to describe the behavior of VLANs inside encapsulated Ethernet frames.

As with any protocol, support for inner VLAN headers is OPTIONAL. In many cases, the use of encapsulated VLANs may be disallowed due to security or implementation considerations. However, in other cases trunking of VLAN frames across a Geneve tunnel can prove useful. As a result, the processing of inner VLAN tags upon ingress or egress from a tunnel endpoint is based upon the configuration of the endpoint and/or control plane and not explicitly defined as part of the data format.

5. Interoperability Issues

Viewed exclusively from the data plane, Geneve does not introduce any interoperability issues as it appears to most devices as UDP frames. However, as there are already a number of tunnel protocols deployed in network virtualization environments, there is a practical question of transition and coexistence.

Since Geneve is a superset of the functionality of the three most common protocols used for network virtualization (VXLAN, NVGRE, and STT) it should be straightforward to port an existing control plane to run on top of it with minimal effort. With both the old and new frame formats supporting the same set of capabilities, there is no need for a hard transition - endpoints directly communicating with each other use any common protocol, which may be different even within a single overall system. As transit devices are primarily forwarding frames on the basis of the IP header, all protocols appear similar and these devices do not introduce additional interoperability concerns.

To assist with this transition, it is strongly suggested that implementations support simultaneous operation of both Geneve and existing tunnel protocols as it is expected to be common for a single node to communicate with a mixture of other nodes. Eventually, older protocols may be phased out as they are no longer in use.

6. Security Considerations

As UDP/IP packets, Geneve does not have any inherent security mechanisms. As a result, an attacker with access to the underlay network transporting the IP frames has the ability to snoop or inject packets. Legitimate but malicious tunnel endpoints may also spoof identifiers in the tunnel header to gain access to networks owned by other tenants.

Within a particular security domain, such as a data center operated by a single provider, the most common and highest performing security mechanism is isolation of trusted components. Tunnel traffic can be carried over a separate VLAN and filtered at any untrusted

boundaries. In addition, tunnel endpoints should only be operated in environments controlled by the service provider, such as the hypervisor itself rather than within a customer VM.

When crossing an untrusted link, such as the public Internet, IPsec [RFC4301] may be used to provide authentication and/or encryption of the IP packets. If the remote tunnel endpoint is not completely trusted, for example it resides on a customer premises, then it may also be necessary to sanitize any tunnel metadata to prevent tenant-hopping attacks.

7. IANA Considerations

IANA has allocated UDP port 6081 as the well-known destination port for Geneve. Upon publication, the registry should be updated to cite this document. The original request was:

Service Name: geneve
Transport Protocol(s): UDP
Assignee: Jesse Gross <jgross@vmware.com>
Contact: Jesse Gross <jgross@vmware.com>
Description: Generic Network Virtualization Encapsulation (Geneve)
Reference: This document
Port Number: 6081

In addition, IANA is requested to create a "Geneve Option Class" registry to allocate Option Classes. This shall be a registry of 16-bit hexadecimal values along with descriptive strings. The identifiers 0x0-0xFF are to be reserved for standardized options for allocation by IETF Review [RFC5226] and 0xFFFF for Experimental Use. Otherwise, identifiers are to be assigned to any organization with an interest in creating Geneve options on a First Come First Served basis. There are no initial registry assignments.

8. Acknowledgements

The authors wish to thank Martin Casado, Bruce Davie and Dave Thaler for their input, feedback, and helpful suggestions.

9. References

9.1. Normative References

- [RFC0768] Postel, J., "User Datagram Protocol", STD 6, RFC 768, August 1980.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

- [RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, May 2008.

9.2. Informative References

- [ETYPES] The IEEE Registration Authority, "IEEE 802 Numbers", 2013, <<http://www.iana.org/assignments/ieee-802-numbers/ieee-802-numbers.xml>>.
- [I-D.davie-stt] Davie, B. and J. Gross, "A Stateless Transport Tunneling Protocol for Network Virtualization (STT)", draft-davie-stt-06 (work in progress), April 2014.
- [I-D.ietf-nvo3-dataplane-requirements] Bitar, N., Lasserre, M., Balus, F., Morin, T., Jin, L., and B. Khasnabish, "NVO3 Data Plane Requirements", draft-ietf-nvo3-dataplane-requirements-03 (work in progress), April 2014.
- [I-D.sridharan-virtualization-nvgre] Sridharan, M., Greenberg, A., Wang, Y., Garg, P., Venkataramiah, N., Duda, K., Ganga, I., Lin, G., Pearson, M., Thaler, P., and C. Tumuluri, "NVGRE: Network Virtualization using Generic Routing Encapsulation", draft-sridharan-virtualization-nvgre-06 (work in progress), October 2014.
- [IEEE.802.1Q-2011] IEEE, "IEEE Standard for Local and metropolitan area networks -- Media Access Control (MAC) Bridges and Virtual Bridged Local Area Networks", IEEE Std 802.1Q, 2011.
- [RFC1191] Mogul, J. and S. Deering, "Path MTU discovery", RFC 1191, November 1990.
- [RFC1981] McCann, J., Deering, S., and J. Mogul, "Path MTU Discovery for IP version 6", RFC 1981, August 1996.
- [RFC2983] Black, D., "Differentiated Services and Tunnels", RFC 2983, October 2000.
- [RFC3031] Rosen, E., Viswanathan, A., and R. Callon, "Multiprotocol Label Switching Architecture", RFC 3031, January 2001.
- [RFC4301] Kent, S. and K. Seo, "Security Architecture for the Internet Protocol", RFC 4301, December 2005.

- [RFC6040] Briscoe, B., "Tunnelling of Explicit Congestion Notification", RFC 6040, November 2010.
- [RFC6935] Eubanks, M., Chimento, P., and M. Westerlund, "IPv6 and UDP Checksums for Tunneled Packets", RFC 6935, April 2013.
- [RFC7348] Mahalingam, M., Dutt, D., Duda, K., Agarwal, P., Kreeger, L., Sridhar, T., Bursell, M., and C. Wright, "Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks", RFC 7348, August 2014.
- [RFC7365] Lasserre, M., Balus, F., Morin, T., Bitar, N., and Y. Rekhter, "Framework for Data Center (DC) Network Virtualization", RFC 7365, October 2014.
- [VL2] Greenberg et al, , "VL2: A Scalable and Flexible Data Center Network", 2009.

Proc. ACM SIGCOMM 2009

Authors' Addresses

Jesse Gross
VMware, Inc.
3401 Hillview Ave.
Palo Alto, CA 94304
USA

Email: jgross@vmware.com

T. Sridhar
VMware, Inc.
3401 Hillview Ave.
Palo Alto, CA 94304
USA

Email: tsridhar@vmware.com

Pankaj Garg
Microsoft Corporation
1 Microsoft Way
Redmond, WA 98052
USA

Email: pankajg@microsoft.com

Chris Wright
Red Hat Inc.
1801 Varsity Drive
Raleigh, NC 27606
USA

Email: chrisw@redhat.com

Ilango Ganga
Intel Corporation
2200 Mission College Blvd.
Santa Clara, CA 95054
USA

Email: ilango.s.ganga@intel.com

Puneet Agarwal
Broadcom Corporation
3151 Zanker Road
San Jose, CA 95134
USA

Email: pagarwal@broadcom.com

Kenneth Duda
Arista Networks
5453 Great America Parkway
Santa Clara, CA 95054
USA

Email: kduda@arista.com

Dinesh G. Dutt
Cumulus Networks
140C S. Whisman Road
Mountain View, CA 94041
USA

Email: ddutt@cumulusnetworks.com

Jon Hudson
Brocade Communications Systems, Inc.
130 Holger Way
San Jose, CA 95134
USA

Email: jon.hudson@gmail.com

Internet Draft
<draft-herbert-gue-03.txt>
Category: Standard track
Expires September 2015

T. Herbert
Google
L. Yong
Huawei USA
O. Zia
Microsoft
March 6, 2015

Generic UDP Encapsulation
<draft-herbert-gue-03.txt>

Status of this Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

The list of current Internet-Drafts can be accessed at
<http://www.ietf.org/ietf/lid-abstracts.txt>

The list of Internet-Draft Shadow Directories can be accessed at
<http://www.ietf.org/shadow.html>

This Internet-Draft will expire on September 7, 2015.

Copyright Notice

Copyright (c) 2014 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document.

Abstract

This specification describes Generic UDP Encapsulation (GUE), which is a scheme for using UDP to encapsulate packets of arbitrary IP protocols for transport across layer 3 networks. By encapsulating packets in UDP, specialized capabilities in networking hardware for efficient handling of UDP packets can be leveraged. GUE specifies basic encapsulation methods upon which higher level constructs, such as tunnels and overlay networks for network virtualization, can be constructed. GUE is extensible by allowing optional data fields as part of the encapsulation, and is generic in that it can encapsulate packets of various IP protocols.

Table of Contents

1. Introduction	3
2. Packet formats	4
2.1. GUE header preamble	4
2.2. GUE header	5
2.3. Flags and optional fields	6
2.4 Private data	7
3. Message types	7
3.1. Control messages	7
3.2. Data messages	8
4. Operation	8
4.1. Network tunnel encapsulation	9
4.2. Transport layer encapsulation	9
4.3. Encapsulator operation	9
4.4. Decapsulator operation	9
4.5. Router and switch operation	10
4.6. Middlebox interactions	10
4.7. NAT	10
4.8. Checksum Handling	11
4.8.1. Checksum requirements	11
4.8.2. GUE header checksum	11
4.8.3. UDP Checksum with IPv4	11
4.8.4. UDP Checksum with IPv6	12
4.9. MTU and fragmentation issues	13
4.10 Congestion control	13
5. Inner flow identifier properties	13
5.1. Flow classification	13
5.2. Inner flow identifier properties	14
6. Motivation for GUE	15
7. Security Considerations	16
7.1. GUE security fields	17
7.2. GUE and IPsec	17
8. IANA Consideration	17
9. Acknowledgements	18

10. References	18
10.1. Normative References	18
10.2. Informative References	18
Appendix A: NIC processing for GUE	19
A.1. Receive multi-queue	20
A.2. Checksum offload	20
A.2.1. Transmit checksum offload	20
A.2.2. Receive checksum offload	21
A.3. Transmit Segmentation Offload	22
A.4. Large Receive Offload	22
Appendix B: Privileged ports	23
Appendix C: Inner flow identifier as a route selector	23
Appendix D: Hardware protocol implementation considerations	23
Authors' Addresses	24

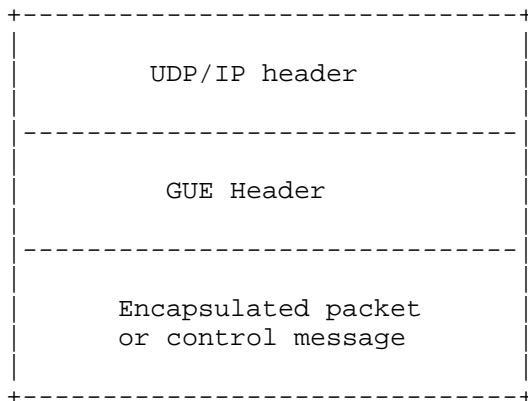
1. Introduction

This specification describes Generic UDP Encapsulation (GUE) which is a general method for encapsulating packets of arbitrary IP protocols within User Datagram Protocol (UDP) [RFC0768] packets. Encapsulating packets in UDP facilitates efficient transport across networks. Networking devices widely provide protocol specific processing and optimizations for UDP (as well as TCP) packets. Packets for atypical IP protocols (those not usually parsed by networking hardware) can be encapsulated in UDP packets to maximize deliverability and to leverage flow specific mechanisms for routing and packet steering.

GUE provides an extensible header format for including optional data in the encapsulation header. This data potentially covers items such as virtual networking identifier, security data for validating or authenticating the GUE header, congestion control data, etc. GUE also allows private optional data in the encapsulation header. This feature can be used by a site or implementation to define local custom optional data, and allows experimentation of options that may eventually become standard.

2. Packet formats

A GUE packet is comprised of a UDP packet whose payload is a GUE header followed by a payload which is either an encapsulated packet of some IP protocol or a control message (like an OAM message). A GUE packet has the general format:



The GUE header is variable length as determined by the presence of optional fields.

2.1. GUE header preamble

The first byte of the GUE header provides the GUE protocol version number, indicator of a control or data message, and header length:

```

0
0 1 2 3 4 5 6 7
+---+---+---+---+---+---+
|Ver|C|   Hlen  |
+---+---+---+---+---+---+

```

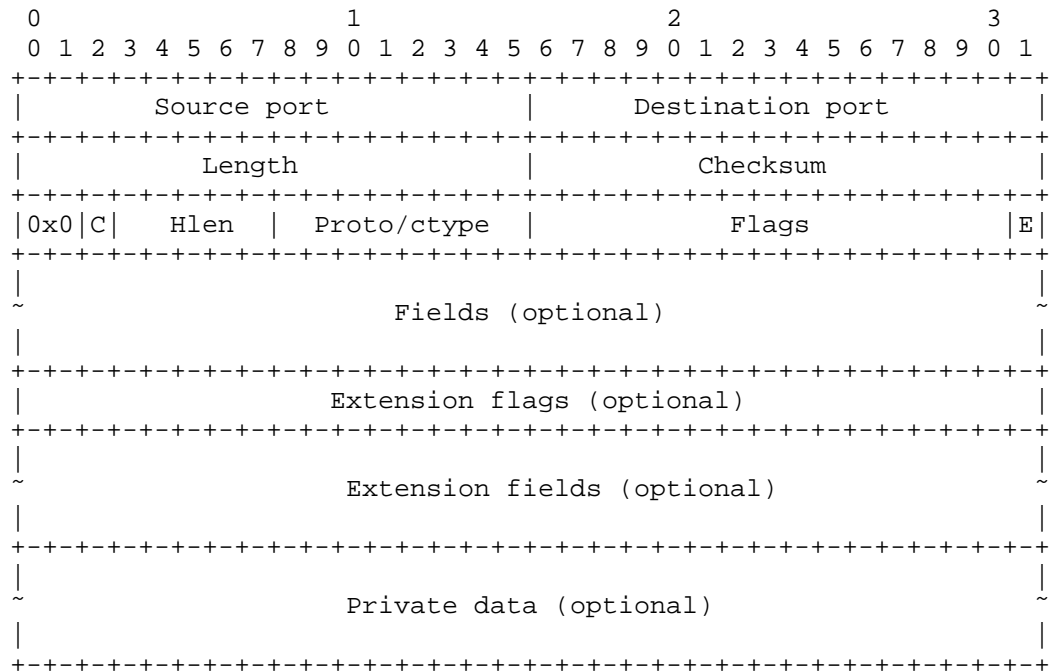
Contents are:

- o Ver: GUE protocol version. The rest of the fields after the preamble are defined based on the version. This field is two bits allowing four possible values.
- o Control flag: When set indicates a control message, not set indicates a data message.
- o Hlen: Length in 32-bit words of the GUE header, including optional fields but not the first four bytes of the header. Computed as $(\text{header_len} - 4) / 4$. All GUE headers are a multiple

of four bytes in length. Maximum header length is 132 bytes.

2.2. GUE header

The header format for version 0x0 of GUE in UDP is:



The contents of the UDP header are:

- o Source port (inner flow identifier): This should be set to a value that represents the encapsulated flow. The properties of the inner flow identifier are described below.
- o Destination port: The GUE assigned port number, 6080.
- o Length: Canonical length of the UDP packet (length of UDP header and payload).
- o Checksum: Standard UDP checksum.

The GUE header consists of:

- o Preamble byte: Version number (0x0), C bit, and header length.

- o Proto/ctype: When the C bit is set this field contains a control message type for the payload. When C bit is not set, the field holds the IP protocol number for the encapsulated packet in the payload. The control message or encapsulated packet begins at the offset provided by Hlen.
- o Flags. Header flags that may be allocated for various purposes and may indicate presence of optional fields. Undefined header flag bits must be set to zero on transmission.
- o 'E' Extension flag. Indicates presence of extension flags option in the optional fields.
- o Fields: Optional fields whose presence is indicated by corresponding flags.
- o Extension flags: An optional field indicated by the E bit. This field provides an additional set of 32 header bit flags for the header.
- o Extension fields: Optional fields whose presence is indicated by corresponding extension flags.
- o Private data: Optional private data. If private data is present it immediately follows that last field present in the header. The length of this data is determined by subtracting the starting offset from the header length.

2.3. Flags and optional fields

Flags and associated optional fields are the primary mechanism of extensibility in GUE. There are sixteen flag bits in the primary GUE header with one being reserved to indicate that an optional extension flags field is present. The extension flags field contains an additional thirty-two flag bits.

A flag may indicate presence of optional fields. The size of an optional field indicated by a flag must be fixed.

Flags may be paired together to allow different lengths for an optional field. For example, if two flag bits are paired, a field may possibly be three different lengths. Regardless of how flag bits may be paired, the lengths and offsets of optional fields corresponding to a set of flags must be well defined.

Optional fields are placed in order of the flags. New flags should be allocated from high to low order bit contiguously without holes. Flags allow random access, for instance to inspect the field

corresponding to the Nth flag bit, an implementation only considers the previous N-1 flags to determine the offset. Flags after the Nth flag are not pertinent in calculating the offset of the Nth flag.

Flags (or paired flags) are idempotent such that new flags should not cause reinterpretation of old flags. Also, new flags should not alter interpretation of other elements in the GUE header nor how the message is parsed (for instance, in a data message the proto/ctype field always holds an IP protocol number as an invariant).

2.4 Private data

An implementation may use private data for its own use. The private data immediately follows the last field in the GUE header and is not a fixed length. This data is considered part of the GUE header and must be accounted for in header length (Hlen). The length of the private data must be a multiple of four and is determined by subtracting the offset of private data in the GUE header from the header length. Specifically:

$$\text{Private_length} = (\text{Hlen} * 4) - \text{Length}(\text{flags})$$

Where "Length(flags)" returns the sum of lengths of all the optional fields present in the GUE header. When there is no private data present, length of the private data is zero.

The semantics and interpretation of private data are implementation specific. A encapsulator and decapsulator MUST agree on the meaning of private data before using it. The private data may be structured as necessary, for instance it might contain its own set of flags and optional fields.

If a decapsulator receives a GUE packet with private data, it MUST validate the private data appropriately. If a decapsulator does not expect private data from an encapsulator the packet MUST be dropped. If a decapsulator cannot validate the contents of private data per the provided semantics the packet MUST also be dropped. An implementation may place security data in GUE private data which must be verified for packet acceptance.

3. Message types

3.1. Control messages

Control messages are indicated in the GUE header when the C bit is set. The payload is interpreted as a control message with type specified in the proto/ctype field. The format and contents of the control message are indicated by the type and can be variable length.

Other than interpreting the proto/ctype field as a control message type, the meaning and semantics of the rest of the elements in the GUE header are the same as that of data messages. Forwarding and routing of control messages should be the same as that of a data message with the same outer IP and UDP header and GUE flags-- this ensures that a control message can be created which follows the same path as a data message.

Control messages can be defined for OAM type messages. For instance, an echo request and corresponding echo reply message may be defined to test for liveness.

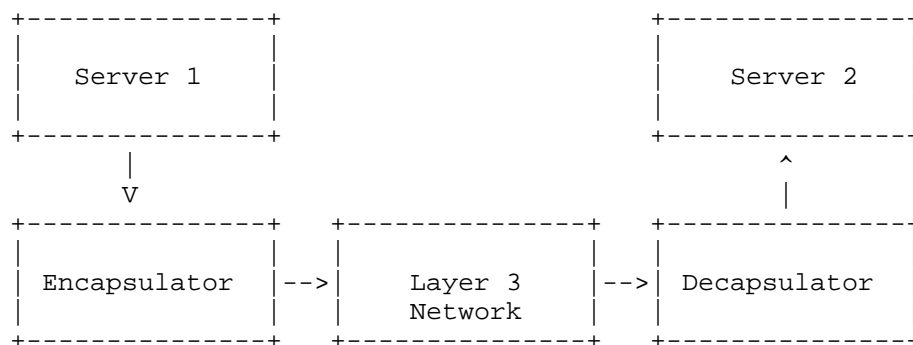
3.2. Data messages

Data messages are indicated in GUE header with C bit not set. The payload of a data message is interpreted as an encapsulated packet of an IP protocol indicated in the proto/ctype field. The packet immediately follows the GUE header.

Data messages are a primary means of encapsulation and can be used to create tunnels for overlay networks.

4. Operation

The figure below illustrates the use of GUE encapsulation between two servers. Server 1 is sending packets to server 2. An encapsulator performs encapsulation of packets from server 1. These encapsulated packets traverse the network as UDP packets. At the decapsulator, packets are decapsulated and sent on to server 2. Packet flow in the reverse direction need not be symmetric; GUE encapsulation is not required in the reverse path.



The encapsulator and decapsulator may be co-resident with the corresponding servers, or may be on separate nodes in the network.

4.1. Network tunnel encapsulation

Network tunneling can be achieved by encapsulating layer 2 or layer 3 packets. In this case the encapsulator and decapsulator nodes are the tunnel endpoints. These could be routers that provide network tunnels on behalf of communicating servers.

4.2. Transport layer encapsulation

When encapsulating layer 4 packets, the encapsulator and decapsulator should be co-resident with the servers. In this case, the encapsulation headers are inserted between the IP header and the transport packet. The addresses in the IP header refer to both the endpoints of the encapsulation and the endpoints for terminating the the transport protocol.

4.3. Encapsulator operation

Encapsulators create GUE data messages, set the source port to the inner flow identifier, set flags and optional fields in the GUE header, and forward packets to a decapsulator.

An encapsulator may be an end host originating the packets of a flow, or may be a network device performing encapsulation on behalf of servers (routers implementing tunnels for instance). In either case, the intended target (decapsulator) is indicated by the outer destination IP address.

If an encapsulator is tunneling packets, that is encapsulating packets of layer 2 or layer 3 protocols (e.g. EtherIP, IP/IP, ESP tunnel mode), it should follow standard conventions for tunneling of one IP protocol over another. Diffserv interaction with tunnels is described in [RFC2983], ECN propagation for tunnels is described in [RFC6040].

4.4. Decapsulator operation

A decapsulator performs decapsulation of GUE packets. A decapsulator is addressed by the outer destination IP address of a GUE packet. The decapsulator validates packets, including fields of the GUE header. If a packet is acceptable, the UDP and GUE headers are removed and the packet is resubmitted for IP protocol processing or control message processing if it is a control message.

If a decapsulator receives a GUE packet with an unsupported version, unknown flag, bad header length (too small for included optional fields), unknown control message type, or an otherwise malformed header, it must drop the packet and may log the event. No error

message is returned back to the encapsulator. Note that set flags in GUE that are unknown to a decapsulator MUST NOT be ignored. If a GUE packet is received by a decapsulator with unknown flags, the packet MUST be dropped.

4.5. Router and switch operation

Routers and switches should forward GUE packets as standard UDP/IP packets. The outer five-tuple should contain sufficient information to perform flow classification corresponding to the flow of the inner packet. A switch should not normally need to parse a GUE header, and none of the flags or optional fields in the GUE header should affect routing.

A router should not modify a GUE header when forwarding a packet. It may encapsulate a GUE packet in another GUE packet, for instance to implement a network tunnel. In this case the router takes the role of an encapsulator, and the corresponding decapsulator is the logical endpoint of the tunnel.

4.6. Middlebox interactions

A middle box may interpret some flags and optional fields of the GUE header for classification purposes, but is not required to understand all flags and fields in GUE packets. A middle box should not drop a GUE packet because there are flags unknown to it. The header length in the GUE header allows a middlebox to inspect the payload packet without needing to parse the flags or optional fields.

A middlebox may infer bidirectional connection semantics to a UDP flow. For instance a stateful firewall may create a five-tuple rule to match flows on egress, and a corresponding five-tuple rule for matching ingress packets where the roles of source and destination are reversed for the IP addresses and UDP port numbers. To operate in this environment, a GUE tunnel must assume connected semantics defined by the UDP five tuple and the use of GUE encapsulation must be symmetric between both endpoints. The source port set in the UDP header must be the destination port the peer would set for replies.

4.7. NAT

IP address and port translation can be performed on the UDP/IP headers adhering to the requirements for NAT with UDP [RFC478]. In the case of stateful NAT, connection semantics must be applied to a GUE tunnel as described above.

When using transport mode encapsulation and traversing a NAT, the IP addresses may be changed such that the pseudo header checksum used

for checksum calculation is modified and the checksum will be found invalid at the receiver. To compensate for this, A GUE option can be added which contains the checksum over the source and destination addresses when the packet is transmitted. Upon receiving this option, the delta of the pseudo header checksum is computed by subtracting the checksum over the source and destination addresses from the checksum value in the option. The resultant value is then added into checksum calculation when validating the inner transport checksum.

4.8. Checksum Handling

This section describes the requirements around the UDP checksum and GUE header checksum. Checksums are an important consideration in that they can provide end to end validation and protect against packet mis-delivery. The latter is allowed by the inclusion of a pseudo header that covers the IP addresses and UDP ports of the encapsulating headers.

4.8.1. Checksum requirements

The potential for mis-delivery of packets due to corruption of IP, UDP, or GUE headers must be considered. One of the following requirements must be met:

- o UDP checksums are enabled (for IPv4 or IPv6).
- o The GUE header checksum is used.
- o Zero UDP checksums are used in accordance with applicable requirements in [GREUDP], [RFC6935], and [RFC6936].

4.8.2. GUE header checksum

The GUE header checksum provides a UDP-lite [RFC3828] type of checksum capability as an optional field of the GUE header. The GUE header checksum minimally covers the GUE header and a GUE pseudo header. The GUE pseudo header includes the corresponding IP addresses as well as the UDP ports of the encapsulating headers. This checksum should provide adequate protection against address corruption in IPv6 when the UDP checksum is zero. Additionally, the GUE checksum provides protection of the GUE header when the UDP checksum is set to zero with either IPv4 or IPv6. The GUE header checksum is defined in [GUECSUM].

4.8.3. UDP Checksum with IPv4

For UDP in IPv4, the UDP checksum MUST be processed as specified in [RFC768] and [RFC1122] for both transmit and receive. An

encapsulator MAY set the UDP checksum to zero for performance or implementation considerations. The IPv4 header includes a checksum which protects against mis-delivery of the packet due to corruption of IP addresses. The UDP checksum potentially provides protection against corruption of the UDP header, GUE header, and GUE payload. Enabling or disabling the use of checksums is a deployment consideration that should take into account the risk and effects of packet corruption, and whether the packets in the network are already adequately protected by other, possibly stronger mechanisms such as the Ethernet CRC. If an encapsulator sets a zero UDP checksum for IPv4 it SHOULD use the GUE header checksum as described in section 4.8.2.

When a decapsulator receives a packet, the UDP checksum field MUST be processed. If the UDP checksum is non-zero, the decapsulator MUST verify the checksum before accepting the packet. By default a decapsulator SHOULD accept UDP packets with a zero checksum. A node MAY be configured to disallow zero checksums per [RFC1122]; this may be done selectively, for instance disallowing zero checksums from certain hosts that are known to be sending over paths subject to packet corruption. If verification of a non-zero checksum fails, a decapsulator lacks the capability to verify a non-zero checksum, or a packet with a zero-checksum was received and the decapsulator is configured to disallow, the packet MUST be dropped and an event MAY be logged.

4.8.4. UDP Checksum with IPv6

For UDP in IPv6, the UDP checksum MUST be processed as specified in [RFC768] and [RFC2460] for both transmit and receive. Unlike IPv4, there is no header checksum in IPv6 that protects against mis-delivery due to address corruption. Therefore, when GUE is used over IPv6, either the UDP checksum must be enabled or the GUE header checksum must be used. An encapsulator MAY set a zero UDP checksum for performance or implementation reasons, in which case the GUE header checksum MUST be used or applicable requirements for using zero UDP checksums in [GREUDP] MUST be met. If the UDP checksum is enabled, then the GUE header checksum should not be used since it is mostly redundant.

When a decapsulator receives a packet, the UDP checksum field MUST be processed. If the UDP checksum is non-zero, the decapsulator MUST verify the checksum before accepting the packet. By default a decapsulator MUST only accept UDP packets with a zero checksum if the GUE header checksum is used and is verified. If verification of a non-zero checksum fails, a decapsulator lacks the capability to verify a non-zero checksum, or a packet with a zero-checksum and no GUE header checksum was received, the packet MUST be dropped and an

event MAY be logged.

4.9. MTU and fragmentation issues

Standard conventions for handling of MTU (Maximum Transmission Unit) and fragmentation in conjunction with networking tunnels (encapsulation of layer 2 or layer 3 packets) should be followed. Details are described in MTU and Fragmentation Issues with In-the-Network Tunneling [RFC4459]

If a packet is fragmented before encapsulation in GUE, all the related fragments must be encapsulated using the same source port (inner flow identifier). An operator may set MTU to account for encapsulation overhead and reduce the likelihood of fragmentation.

4.10 Congestion control

Per requirements of [RFC5405], if the IP traffic encapsulated with GUE implements proper congestion control no additional mechanisms should be required.

In the case that the encapsulated traffic does not implement any or sufficient control, or it is not known rather a transmitter will consistently implement proper congestion control, then congestion control at the encapsulation layer must be provided. Note this case applies to a significant use case in network virtualization in which guests run third party networking stacks that cannot be implicitly trusted to implement conformant congestion control.

Out of band mechanisms such as rate limiting, Managed Circuit Breaker, or traffic isolation may used to provide rudimentary congestion control. For finer grained congestion control that allow alternate congestion control algorithms, reaction time within an RTT, and interaction with ECN, in band mechanisms may warranted.

DCCP may be used to provide congestion control for encapsulated flows. In this case, the protocol stack for an IP tunnel may be IP-GUE-DCCP-IP. Alternatively, GUE can be extended to include congestion control (related data carried in GUE optional fields). Congestion control mechanisms for GUE will be elaborated in other specifications.

5. Inner flow identifier properties

5.1. Flow classification

A major objective of using GUE is that a network device can perform flow classification corresponding to the flow of the inner

encapsulated packet based on the contents in the outer headers.

Hardware devices commonly perform hash computations on packet headers to classify packets into flows or flow buckets. Flow classification is done to support load balancing (statistical multiplexing) of flows across a set of networking resources. Examples of such load balancing techniques are Equal Cost Multipath routing (ECMP), port selection in Link Aggregation, and NIC device Receive Side Scaling (RSS). Hashes are usually either a three-tuple hash of IP protocol, source address, and destination address; or a five-tuple hash consisting of IP protocol, source address, destination address, source port, and destination port. Typically, networking hardware will compute five-tuple hashes for TCP and UDP, but only three-tuple hashes for other IP protocols. Since the five-tuple hash provides more granularity, load balancing can be finer grained with better distribution. When a packet is encapsulated with GUE, the source port in the outer UDP packet is set to reflect the flow of the inner packet. When a device computes a five-tuple hash on the outer UDP/IP header of a GUE packet, the resultant value classifies the packet per its inner flow.

To support flow classification, the source port of the UDP header in GUE is set to a value that maps to the inner flow. This is referred to as the inner flow identifier. The inner flow identifier is set by the encapsulator; it can be computed on the fly based on packet contents or retrieved from a state maintained for the inner flow.

Examples of deriving an inner flow identifier are:

- o If the encapsulated packet is a layer 4 packet, TCP/IPv4 for instance, the inner flow identifier could be based on the canonical five-tuple hash of the inner packet.
- o If the encapsulated packet is an AH transport mode packet with TCP as next header, the inner flow identifier could be a hash over a three-tuple: TCP protocol and TCP ports of the encapsulated packet.
- o If a node is encrypting a packet using ESP tunnel mode and GUE encapsulation, the inner flow identifier could be based on the contents of clear-text packet. For instance, a canonical five-tuple hash for a TCP/IP packet could be used.

5.2. Inner flow identifier properties

The inner flow identifier is the value set in the UDP source port of a GUE packet. The inner flow identifier should adhere to the following properties:

- o The value set in the source port should be within the ephemeral port range. IANA suggests this range to be 49152 to 65535, where the high order two bits of the port are set to one. This provides fourteen bits of entropy for the inner flow identifier.
- o The inner flow identifier should have a uniform distribution across encapsulated flows.
- o An encapsulator may occasionally change the inner flow identifier used for an inner flow per its discretion (for security, route selection, etc). Changing the value should happen no more than once every thirty seconds.
- o Decapsulators, or any networking devices, should not attempt any interpretation of the inner flow identifier, nor should they attempt to reproduce any hash calculation. They may use the value to match further receive packets for steering decisions, but cannot assume that the hash uniquely or permanently identifies a flow.
- o Input to the inner flow identifier is not restricted to ports and addresses; input could include flow label from an IPv6 packet, SPI from an ESP packet, or other flow related state in the encapsulator that is not necessarily conveyed in the packet.
- o The assignment function for inner flow identifiers should be randomly seeded to mitigate denial of service attacks. The seed may be changed periodically.

6. Motivation for GUE

This section presents the motivation for GUE with respect to other encapsulation methods.

A number of different encapsulation techniques have been proposed for the encapsulation of one protocol over another. EtherIP [RFC3378] provides layer 2 tunneling of Ethernet frames over IP. GRE [RFC2784], MPLS [RFC4023], and L2TP [RFC2661] provide methods for tunneling layer 2 and layer 3 packets over IP. NVGRE [NVGRE] and VXLAN [RFC7348] are proposals for encapsulation of layer 2 packets for network virtualization. IPIP [RFC2003] and Generic packet tunneling in IPv6 [RFC2473] provide methods for tunneling IP packets over IP.

Several proposals exist for encapsulating packets over UDP including ESP over UDP [RFC3948], TCP directly over UDP [TCPUDP], VXLAN, LISP [RFC6830] which encapsulates layer 3 packets, and Generic UDP Encapsulation for IP Tunneling (GRE over UDP)[GREUDP]. Generic UDP tunneling [GUT] is a proposal similar to GUE in that it aims to

tunnel packets of IP protocols over UDP.

GUE has the following discriminating features:

- o UDP encapsulation leverages specialized network device processing for efficient transport. The semantics for using the UDP source port as an identifier for an inner flow are defined.
- o GUE permits encapsulation of arbitrary IP protocols, which includes layer 2, 3, and 4 protocols. This potentially allows nearly all traffic within a data center to be normalized to be either TCP or UDP on the wire.
- o Multiple protocols can be multiplexed over a single UDP port number. This is in contrast to techniques to encapsulate protocols over UDP using a protocol specific port number (such as ESP/UDP, GRE/UDP, SCTP/UDP). GUE provides a uniform and extensible mechanism for encapsulating all IP protocols in UDP with minimal overhead (four bytes of additional header).
- o GUE is extensible. New flags and optional fields can be defined.
- o The GUE header includes a header length field. This allows a network node to inspect an encapsulated packet without needing to parse the full encapsulation header.
- o Private data in the encapsulation header allows local customization and experimentation while being compatible with processing in network nodes (routers and middleboxes).
- o GUE includes both data messages (encapsulation of packets) and control messages (such as OAM).

7. Security Considerations

Encapsulation of IP protocols within GUE should not increase security risk, nor provide additional security in itself. As suggested in section 5 the source port for of UDP packets in GUE should be randomly seeded to mitigate some possible denial service attacks.

GUE is most useful when it is in the outermost header of a packet which allows for flow hash calculation as well as making GUE header data (such as virtual network identifier) visible to switches and middleboxes. GUE must be amenable to encapsulating (and being encapsulated within) IPsec. Also, we allow provisions to secure the GUE header itself without external protocol.

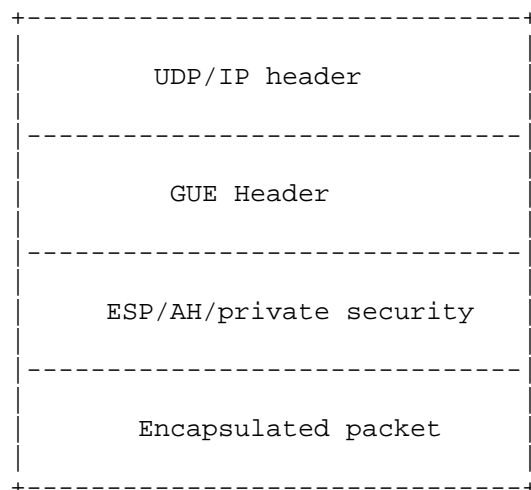
Security for Generic UDP Encapsulation is described in more detail in [GUESEC].

7.1. GUE security fields

Security fields should be used to provide integrity and authentication of the GUE header. Security negotiation (algorithms, interpretation of security field, key management, etc.) is expected to be done out of band between hosts.

7.2. GUE and IPsec

GUE may be used to encapsulate IPsec packets. This allows the benefits of deriving a flow hash for the inner, potentially encrypted, packet. In this case the protocol stack may be:



Note that IPsec would not cover the GUE header in this case (does not authenticate it for instance). GUE security optional fields may be used to provide authentication or integrity of the GUE header.

8. IANA Consideration

A user UDP port number assignment for GUE has been assigned:

Service Name: gue
Transport Protocol(s): UDP
Assignee: Tom Herbert <therbert@google.com>
Contact: Tom Herbert <therbert@google.com>
Description: Generic UDP Encapsulation

Reference: draft-herbert-gue
Port Number: 6080
Service Code: N/A
Known Unauthorized Uses: N/A
Assignment Notes: N/A

9. Acknowledgements

The authors would like to thank David Liu for valuable input on this draft.

10. References

10.1. Normative References

[RFC0768] Postel, J., "User Datagram Protocol", STD 6, RFC 768, August 1980.

[RFC2434] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", RFC 2434, October 1998.

[RFC2983] Black, D., "Differentiated Services and Tunnels", RFC 2983, October 2000.

[RFC6040] Briscoe, B., "Tunnelling of Explicit Congestion Notification", RFC 6040, November 2010.

[RFC6936] Fairhurst, G. and M. Westerlund, "Applicability Statement for the Use of IPv6 UDP Datagrams with Zero Checksums", RFC 6936, April 2013.

[RFC4459] Savola, P., "MTU and Fragmentation Issues with In-the-Network Tunneling", RFC 4459, April 2006.

10.2. Informative References

[RFC2003] Perkins, C., "IP Encapsulation within IP", RFC 2003, October 1996.

[RFC3948] Huttunen, A., Swander, B., Volpe, V., DiBurro, L., and M. Stenberg, "UDP Encapsulation of IPsec ESP Packets", RFC 3948, January 2005.

[RFC6830] Farinacci, D., Fuller, V., Meyer, D., and D. Lewis, "The Locator/ID Separation Protocol (LISP)", RFC 6830, January 2013.

[RFC3378] Housley, R. and S. Hollenbeck, "EtherIP: Tunneling Ethernet Frames in IP Datagrams", RFC 3378, September 2002.

[RFC2784] Farinacci, D., Li, T., Hanks, S., Meyer, D., and P. Traina, Generic Routing Encapsulation (GRE)", RFC 2784, March 2000.

[RFC4023] Worster, T., Rekhter, Y., and E. Rosen, Ed., "Encapsulating MPLS in IP or Generic Routing Encapsulation (GRE)", RFC 4023, March 2005.

[RFC2661] Townsley, W., Valencia, A., Rubens, A., Pall, G., Zorn, G., and B. Palter, "Layer Two Tunneling Protocol "L2TP"", RFC 2661, August 1999.

[RFC5925] Touch, J., Mankin, A., and R. Bonica, "The TCP Authentication Option", RFC 5925, June 2010.

[RFC3828] Larzon, L-A., Degermark, M., Pink, S., Jonsson, L-E., Ed., and G. Fairhurst, Ed., "The Lightweight User Datagram Protocol (UDP-Lite)", RFC 3828, July 2004, <<http://www.rfc-editor.org/info/rfc3828>>.

[RFC7348] Mahalingam, M., Dutt, D., Duda, K., Agarwal, P., Kreeger, L., Sridhar, T., Bursell, M., and C. Wright, "Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks", RFC 7348, August 2014, <<http://www.rfc-editor.org/info/rfc7348>>.

[NVGRE] NVGRE: Network Virtualization using Generic Routing Encapsulation draft-sridharan-virtualization-nvgre-03

[TCPUDP] Encapsulation of TCP and other Transport Protocols over UDP draft-cheshire-tcp-over-udp-00

[GREUDP] Generic UDP Encapsulation for IP Tunneling draft-yong-tsvwg-gre-in-udp-encap-02

[GUESEC] Yong, L., Herbert, T., "Generic UDP Encapsulation (GUE) for Secure Transport", draft-hy-gue-4-secure-transport-00, work in progress.

[GUT] Generic UDP Tunnelling (GUT) draft-manner-tsvwg-gut-02.txt

[REMCSUM] Remote Checksum Offload draft-herbert-remotecsumoffload-00

Appendix A: NIC processing for GUE

This appendix provides some guidelines for Network Interface Cards

(NICs) to implement common offloads and accelerations to support GUE. Note that most of this discussion is generally applicable to other methods of UDP based encapsulation.

A.1. Receive multi-queue

Contemporary NICs support multiple receive descriptor queues (multi-queue). Multi-queue enables load balancing of network processing for a NIC across multiple CPUs. On packet reception, a NIC must select the appropriate queue for host processing. Receive Side Scaling is a common method which uses the flow hash for a packet to index an indirection table where each entry stores a queue number. Flow Director and Accelerated Receive Flow Steering (aRFS) allow a host to program the queue that is used for a given flow which is identified either by an explicit five-tuple or by the flow's hash.

GUE encapsulation should be compatible with multi-queue NICs that support five-tuple hash calculation for UDP/IP packets as input to RSS. The inner flow identifier (source port) ensures classification of the encapsulated flow even in the case that the outer source and destination addresses are the same for all flows (e.g. all flows are going over a single tunnel).

By default, UDP RSS support is often disabled in NICs to avoid out of order reception that can occur when UDP packets are fragmented. As discussed above, fragmentation of GUE packets should be mitigated by fragmenting packets before entering a tunnel, path MTU discovery in higher layer protocols, or operator adjusting MTUs. Other UDP traffic may not implement such procedures to avoid fragmentation, so enabling UDP RSS support in the NIC should be a considered tradeoff during configuration.

A.2. Checksum offload

Many NICs provide capabilities to calculate standard ones complement payload checksum for packets in transmit or receive. When using GUE encapsulation there are at least two checksums that may be of interest: the encapsulated packet's transport checksum, and the UDP checksum in the outer header.

A.2.1. Transmit checksum offload

NICs may provide a protocol agnostic method to offload transmit checksum (`NETIF_F_HW_CSUM` in Linux parlance) that can be used with GUE. In this method the host provides checksum related parameters in a transmit descriptor for a packet. These parameters include the starting offset of data to checksum, the length of data to checksum, and the offset in the packet where the computed checksum is to be

written. The host initializes the checksum field to pseudo header checksum.

In the case of GUE, the checksum for an encapsulated transport layer packet, a TCP packet for instance, can be offloaded by setting the appropriate checksum parameters.

NICs typically can offload only one transmit checksum per packet, so simultaneously offloading both an inner transport packet's checksum and the outer UDP checksum is likely not possible. In this case setting UDP checksum to zero (per above discussion) and offloading the inner transport packet checksum might be acceptable.

If an encapsulator is co-resident with a host, then checksum offload may be performed using remote checksum offload [REMCSUM]. Remote checksum offload relies on NIC offload of the simple UDP/IP checksum which is commonly supported even in legacy devices. In remote checksum offload the outer UDP checksum is set and the GUE header includes an option indicating the start and offset of the inner "offloaded" checksum. The inner checksum is initialized to the pseudo header checksum. When a decapsulator receives a GUE packet with the remote checksum offload option, it completes the offload operation by determining the packet checksum from the indicated start point to the end of the packet, and then adds this into the checksum field at the offset given in the option. Computing the checksum from the start to end of packet is efficient if checksum-complete is provided on the receiver.

A.2.2. Receive checksum offload

GUE is compatible with NICs that perform a protocol agnostic receive checksum (CHECKSUM_COMPLETE in Linux parlance). In this technique, a NIC computes a ones complement checksum over all (or some predefined portion) of a packet. The computed value is provided to the host stack in the packet's receive descriptor. The host driver can use this checksum to "patch up" and validate any inner packet transport checksum, as well as the outer UDP checksum if it is non-zero.

Many legacy NICs don't provide checksum-complete but instead provide an indication that a checksum has been verified (CHECKSUM_UNNECESSARY in Linux). Usually, such validation is only done for simple TCP/IP or UDP/IP packets. If a NIC indicates that a UDP checksum is valid, the checksum-complete value for the UDP packet is the "not" of the pseudo header checksum. In this way, checksum-unnecessary can be converted to checksum-complete. So if the NIC provides checksum-unnecessary for the outer UDP header in an encapsulation, checksum conversion can be done so that the checksum-complete value is derived and can be used by the stack to validate checksums in the encapsulated packet.

A.3. Transmit Segmentation Offload

Transmit Segmentation Offload (TSO) is a NIC feature where a host provides a large (>MTU size) TCP packet to the NIC, which in turn splits the packet into separate segments and transmits each one. This is useful to reduce CPU load on the host.

The process of TSO can be generalized as:

- Split the TCP payload into segments which allow packets with size less than or equal to MTU.
- For each created segment:
 1. Replicate the TCP header and all preceding headers of the original packet.
 2. Set payload length fields in any headers to reflect the length of the segment.
 3. Set TCP sequence number to correctly reflect the offset of the TCP data in the stream.
 4. Recompute and set any checksums that either cover the payload of the packet or cover header which was changed by setting a payload length.

Following this general process, TSO can be extended to support TCP encapsulation in GUE. For each segment the Ethernet, outer IP, UDP header, GUE header, inner IP header if tunneling, and TCP headers are replicated. Any packet length header fields need to be set properly (including the length in the outer UDP header), and checksums need to be set correctly (including the outer UDP checksum if being used).

To facilitate TSO with GUE it is recommended that optional fields should not contain values that must be updated on a per segment basis-- for example the GUE fields should not include checksums, lengths, or sequence numbers that refer to the payload. If the GUE header does not contain such fields then the TSO engine only needs to copy the bits in the GUE header when creating each segment and does not need to parse the GUE header.

A.4. Large Receive Offload

Large Receive Offload (LRO) is a NIC feature where packets of a TCP connection are reassembled, or coalesced, in the NIC and delivered to the host as one large packet. This feature can reduce CPU utilization in the host.

LRO requires significant protocol awareness to be implemented correctly and is difficult to generalize. Packets in the same flow need to be unambiguously identified. In the presence of tunnels or network virtualization, this may require more than a five-tuple match (for instance packets for flows in two different virtual networks may have identical five-tuples). Additionally, a NIC needs to perform validation over packets that are being coalesced, and needs to fabricate a single meaningful header from all the coalesced packets.

The conservative approach to supporting LRO for GUE would be to assign packets to the same flow only if they have identical five-tuple and were encapsulated the same way. That is the outer IP addresses, the outer UDP ports, GUE protocol, GUE flags and fields, and inner five tuple are all identical.

Appendix B: Privileged ports

Using the source port to contain an inner flow identifier value disallows the security method of a receiver enforcing that the source port be a privileged port. Privileged ports are defined by some operating systems to restrict source port binding. Unix, for instance, considered port number less than 1024 to be privileged.

Enforcing that packets are sent from a privileged port is widely considered an inadequate security mechanism and has been mostly deprecated. To approximate this behavior, an implementation could restrict a user from sending a packet destined to the GUE port without proper credentials.

Appendix C: Inner flow identifier as a route selector

An encapsulator generating an inner flow identifier may modulate the value to perform a type of multipath source routing. Assuming that networking switches perform ECMP based on the flow hash, a sender can affect the path by altering the inner flow identifier. For instance, a host may store a flow hash in its PCB for an inner flow, and may alter the value upon detecting that packets are traversing a lossy path. Changing the inner flow identifier for a flow should be subject to hysteresis (at most once every thirty seconds) to limit the number of out of order packets.

Appendix D: Hardware protocol implementation considerations

A low level protocol, such is GUE, is likely interesting to being supported by high speed network devices. Variable length header (VLH) protocols like GUE are often considered difficult to efficiently implement in hardware. In order to retain the important characteristics of an extensible and robust protocol, hardware

vendors may practice "constrained flexibility". In this model, only certain combinations or protocol header parameterizations are implemented in hardware fast path. Each such parameterization is fixed length so that the particular instance can be optimized as a fixed length protocol. In the case of GUE this constitutes specific combinations of GUE flags, fields, and next protocol. The selected combinations would naturally be the most common cases which form the "fast path", and other combinations are assumed to take the "slow path".

In time, needs and requirements of the protocol may change which may manifest themselves as new parameterizations to be supported in the fast path. To allow allow this extensibility, a device practicing constrained flexibility should allow the fast path parameterizations to be programmable.

Authors' Addresses

Tom Herbert
Google
1600 Amphitheatre Parkway
Mountain View, CA
US

EMail: tom@herbertland.com

Lucy Yong
Huawei USA
5340 Legacy Dr.
Plano, TX 75024
US

Osama Zia
Microsoft
osamaz@microsoft.com

February 29, 2016

Remote checksum offload for encapsulation
draft-herbert-remotecsumoffload-02

Abstract

This document describes remote checksum offload for encapsulation, which is a mechanism that provides checksum offload of encapsulated packets using rudimentary offload capabilities found in most Network Interface Card (NIC) devices. The outer header checksum e.g. that in UDP or GRE) is enabled in packets and, with some additional meta information, a receiver is able to deduce the checksum to be set for an inner encapsulated packet. Effectively this offloads the computation of the inner checksum. Enabling the outer checksum in encapsulation has the additional advantage that it covers more of the packet than the inner checksum including the encapsulation headers.

Status of this Memo

This Internet-Draft is submitted to IETF in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

The list of current Internet-Drafts can be accessed at
<http://www.ietf.org/lid-abstracts.html>

The list of Internet-Draft Shadow Directories can be accessed at
<http://www.ietf.org/shadow.html>

Copyright and License Notice

Copyright (c) 2016 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1	Introduction	3
2	Checksum offload background	3
2.1	The Internet checksum	3
2.2	Transmit checksum offload	4
2.2.1	Generic transmit offload	4
2.2.2	Local checksum offload	4
2.2.3	Protocol specific transmit offload	5
2.3	Receive checksum offload	5
2.3.1	CHECKSUM_COMPLETE	6
2.3.2	CHECKSUM_UNNECESSARY	6
3.0	Remote checksum offload	6
3.1	Option format	6
3.2	Transmit operation	7
3.3	Receiver operation	8
3.4	Interaction with TCP segmentation offload	9
4	Security Considerations	9
5	IANA Considerations	9
6	References	9
6.1	Normative References	9
6.2	Informative References	10
	Authors' Addresses	10

1 Introduction

Checksum offload is a capability of NICs where the checksum calculation for a transport layer packet (TCP, UDP, etc.) is performed by a device on behalf of the host stack. Checksum offload is applicable to both transmit and receive, where on transmit the device writes the computed checksum into the packet, and on receive the device provides the computed checksum of the packet or an indication that specific transport checksums were validated. This feature saves CPU cycles in the host and has become ubiquitous in modern NICs.

A host may both source transport packets and encapsulate them for transit over an underlying network. In this case checksum offload is still desirable, but now must be done on an encapsulated packet. Many deployed NICs are only capable of providing checksum offload for simple TCP or UDP packets. Such NICs typically use protocol specific mechanisms where they must parse headers in order to perform checksum calculations. Updating these NICs to perform checksum offload for encapsulation requires new parsing logic which is likely infeasible or at cost prohibitive.

In this specification we describe an alternative that uses rudimentary NIC offload features to support offloading checksum calculation of encapsulated packets. In this design, the outer checksum is enabled on transmit, and meta information indicating the location of the checksum field being offloaded and its starting point for computation are sent with a packet. On receipt, after the outer checksum is verified, the receiver sets the offloaded checksum field per the computed packet checksum and the meta data.

2 Checksum offload background

In this section we provide some background into checksum offload operation.

2.1 The Internet checksum

The Internet checksum [RFC0791] is used by several Internet protocols including IP [RFC1122], TCP [RFC0793], UDP [RFC0768] and GRE [RFC2784]. Efficient checksum calculation is critical to good performance [RFC1071], and the mathematical properties are useful in incrementally updating checksums [RFC1624]. An early approach to implementing checksum offload in hardware is described in [RFC1936].

TCP and UDP checksums cover a pseudo header which is composed of the source and destination addresses of the corresponding IP packet, upper layer packet length, and protocol. The checksum pseudo header

is defined in [RFC0768] and [RFC0793] for IPv4, and in [RFC2460] for IPv6.

2.2 Transmit checksum offload

In transmit checksum offload, a host network stack defers the calculation and setting of a transport checksum in the packet to the device. A device may provide checksum offload only for specific protocols, or may provide a generic interface. In either case, support for only one offloaded checksum per packet is typical.

When using transmit checksum offload, a host stack must initialize the checksum field in the packet. This is done by setting to zero (GRE) or to the bitwise not of the pseudo header (UDP or TCP). The device proceeds by computing the packet checksum from the start of the transport header through to the end of the packet. The bitwise not of the resulting value is written in the checksum field of the transport packet.

2.2.1 Generic transmit offload

A device can provide a generic interface for transmit checksum offload. Checksum offload is enabled by setting two fields in the transmit descriptor for a packet: start offset and checksum offset. The start offset indicates the byte in the packet where the checksum calculation should start. The checksum offset indicates the offset in the packet where the checksum value is to be written.

The generic interface is protocol agnostic, however only supports one offloaded checksum per packet. While it is conceivable that a NIC could provide offload for more checksums by defining more than one checksum start/offset pair in the transmit descriptor, a more general and efficient solution is Local Checksum Offload.

2.2.2 Local checksum offload

Local Checksum Offload [LCO] (or LCO) is a technique for efficiently computing the outer checksum of an encapsulated datagram when the inner checksum is due to be offloaded. The ones-complement sum of a correctly checksummed TCP or UDP packet is equal to the sum of the pseudo header, since everything else gets 'cancelled out' by the checksum field. This property holds since the sum was complemented before being written to the checksum field. More generally, this holds in any case where the Internet one's complement checksum is used, and thus any checksum that generic transmit offload supports. That is, if we have set up transmit checksum offload with a start/offset pair, we know that after the device has filled in that checksum the one's complement sum from checksum start to the end of

the packet will be equal to whatever value is set in the checksum field beforehand. This property allows computing the outer checksum without considering at the payload per the algorithm:

- 1) Compute the checksum from the outer packet's checksum start offset to the inner packet's checksum start offset.
- 2) Add the bit-wise not of the pseudo header checksum for the inner packet.
- 3) The result is the checksum from the outer packet's start offset to the end of the packet. Taking into account the pseudo header for the outer checksum allows the outer checksum field to be set without offload processing.

Step 1) requires that some checksum calculation is performed on the host stack, however this is only done over some portion of packet headers which is typically much smaller than the payload of the packet.

LCO can be used for nested encapsulations; in this case, the outer encapsulation layer will sum over both its own header and the 'middle' header. Thus, if the device has the capability to offload an inner checksum in encapsulation, any number of outer checksums can be efficiently calculated using this technique.

2.2.3 Protocol specific transmit offload

Some devices support transmit checksum offload for very specific protocols. For instance, many legacy devices can only perform checksum offload for UDP/IP and TCP/IP packets. These devices parse transmitted packets in order to determine the checksum start and checksum offset. They may also ignore the value in the checksum field by setting it to zero for checksum computation and computing the checksum of the pseudo header themselves.

Protocol specific transmit offload is limited to the protocols a device supports. To support checksum offload of an encapsulated packet, a device must be able to parse the encapsulation layer in order to locate the inner packet.

2.3 Receive checksum offload

Upon receiving a packet, a device may perform a checksum calculation over the packet or part of the packet depending on the protocol. A result of this calculation is returned in the meta data of the receive descriptor for the packet. The host stack can apply the result in verifying checksums as it processes the packet. The intent

is that the offload will obviate the need for the networking stack to perform its own checksum calculation over the packet.

There are two basic methods of receive checksum offload: CHECKSUM_COMPLETE and CHECKSUM_UNNECESSARY.

2.3.1 CHECKSUM_COMPLETE

A device may calculate the checksum of a whole packet (layer 2 payload) and return the resultant value to the host stack. The host stack can subsequently use this value to validate checksums in the packet. As the packet is parsed through various layers, the calculated checksum is updated to correspond to each layer (subtract out checksum for preceding bytes for a given header).

CHECKSUM_COMPLETE is protocol agnostic and does not require any protocol awareness in the device. It works for any encapsulation and supports an arbitrary number of checksums in the packet.

2.3.2 CHECKSUM_UNNECESSARY

A device may explicitly validate a checksum in a packet and return a flag in the receive descriptor that a transport checksum has been verified (host performing checksum computation is unnecessary). Some devices may be capable of validating more than one checksum in the packet, in which case the device returns a count of the number verified. Typically, only a positive signal is returned, if the device was unable to validate a checksum it does not return any information and the host will generally perform its own checksum computation. If a device returns a count of validations, this must refer to consecutive checksums that are present and validated in a packet (checksums cannot be skipped).

CHECKSUM_UNNECESSARY is protocol specific, for instance in the case of UDP or TCP a device needs to consider the pseudo header in checksum validation. To support checksum offload of an encapsulated packet, a device must be able to parse the encapsulation layer in order to locate the inner packet.

3.0 Remote checksum offload

This section describes the remote checksum offload mechanism. This is primarily useful with UDP based encapsulation where the UDP checksum is enabled (not set to zero on transmit). The same technique could be applied to GRE encapsulation where the GRE checksum is enabled.

3.1 Option format

Remote checksum offload requires the sending of optional data with an encapsulated packet. This data is a pair of checksum start and checksum offset values. More than one offloaded checksum could be supported if multiple pairs are sent.

The logical data format for remote checksum offload is:

```

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
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|           Checksum start           |           Checksum offset           |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+

```

- o Checksum start: starting offset for checksum computation relative to the start of the encapsulated packet. This is typically the offset of a transport header (e.g. UDP or TCP).
- o Checksum offset: Offset relative to the start of the encapsulated packet where the derived checksum value is to be written. This typically is the offset of the checksum field in the transport header (e.g. UDP or TCP).

Support for remote checksum offload with specific encapsulation protocols is outside the scope of this document, however any encapsulation format that supports some reasonable form of optional meta data should be amenable. In Generic UDP Encapsulation [GUE] this would entail defining an optional field, in Geneve [GENEVE] a TLV would be defined, for NSH [NSH] the meta data can either be in a service header or within a TLV. In any scenario, what the offsets in the meta data are relative to must be unambiguous.

3.2 Transmit operation

The typical actions to set remote checksum offload on transmit are:

- 1) Transport layer creates a packet and indicates in internal packet meta data that checksum is to be offloaded to the NIC (normal transport layer processing for checksum offload). The checksum field is populated with the bitwise not of the checksum of the pseudo header or zero as appropriate.
- 2) Encapsulation layer adds its headers to the packet including the offload meta data. The start offset and checksum offset are set accordingly.
- 3) Encapsulation layer arranges for checksum offload of the outer header checksum (e.g. UDP).

- 4) Packet is sent to the NIC. The NIC will perform transmit checksum offload and set the checksum field in the outer header. The inner header and rest of the packet are transmitted without modification.

3.3 Receiver operation

The typical actions a host receiver does to support remote checksum offload are:

- 1) Receive packet and validate outer checksum following normal processing (e.g. validate non-zero UDP checksum).
- 2) Deduce full checksum for the IP packet. This is directly provided if device returns the packet checksum in CHECKSUM_COMPLETE. If the device returned CHECKSUM_UNNECESSARY, then the complete checksum can be trivially derived as either zero (GRE) or the bitwise not of the outer pseudo header (UDP).
- 3) From the packet checksum, subtract the checksum computed from the start of the packet (outer IP header) to the offset in the packet indicated by checksum start in the meta data. The result is the deduced checksum to set in the checksum field of the encapsulated transport packet.

In pseudo code:

```
csum: initialized to checksum computed from start (outer IP
      header) to the end of the packet
start_of_packet: address of start of packet
encap_payload_offset: relative to start_of_packet
csum_start: value from meta data
checksum(start, len): function to compute checksum from start
                    address for len bytes

csum -= checksum(start_of_packet, encap_payload_offset +
                    csum_start)
```

- 4) Write the resultant checksum value into the packet at the offset provided by checksum offset in the meta data.

In pseudo code:

```
csum_offset: offset of checksum field

*(start_of_packet + encap_payload_offset +
  csum_offset) = csum
```


- 5) Checksum is verified at the transport layer using normal processing. This should not require any checksum computation over the packet since the complete checksum has already been provided.

3.4 Interaction with TCP segmentation offload

Remote checksum offload may be useful with TCP Segmentation Offload (TSO) in order to avoid host checksum calculations at the receiver. This can be implemented on a transmitter as follows:

- 1) Host stack prepares a large segment for transmission including adding of encapsulation headers and the remote checksum option which refers to the encapsulated transport checksum in the large segment.
- 2) TSO is performed by the device taking encapsulation into account. The outer checksum is computed and written for each packet. The inner checksum is not computed, and the encapsulation header (including checksum meta data) is replicated for each packet.
- 3) At the receiver remote checksum offload processing occurs as normal for each packet.

4 Security Considerations

Remote checksum offload should not impact protocol security.

5 IANA Considerations

There are no IANA considerations in this specification. The remote checksum offload meta data may require an option number or type in specific encapsulation formats that support it.

6 References

6.1 Normative References

- [RFC0791] Postel, J., "Internet Protocol", STD 5, RFC 791, September 1981.
- [RFC1122] Braden, R., Ed., "Requirements for Internet Hosts - Communication Layers", STD 3, RFC 1122, October 1989.
- [RFC0793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, September 1981.

- [RFC0768] Postel, J., "User Datagram Protocol", STD 6, RFC 768, August 1980.
- [RFC2784] Farinacci, D., Li, T., Hanks, S., Meyer, D., and P. Traina, "Generic Routing Encapsulation (GRE)", RFC 2784, March 2000.
- [RFC2460] Deering, S. and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification", RFC 2460, December 1998.

6.2 Informative References

- [RFC1071] Braden, R., Borman, D., and C. Partridge, "Computing the Internet checksum", RFC1071, September 1988.
- [RFC1624] Rijssinghani, A., Ed., "Computation of the Internet Checksum via Incremental Update", RFC1624, May 1994.
- [RFC1936] Touch, J. and B. Parham, "Implementing the Internet Checksum in Hardware", RFC1936, April 1996.
- [GUE] Herbert, T., Yong, L, and Zia, O., "Generic UDP Encapsulation". draft-ietf-nvo3-gue-02
- [GENEVE] Gross, J. and Gango, I., "Geneve: Generic Network Virtualization Encapsulation", draft-ietf-nvo3-geneve-01, January 1, 2016
- [NSH] Quinn, P. and Elzur, U., "Network Service Header", draft-ietf-sfc-nsh-02.txt, January 19, 2016
- [LOC] Cree, E. Checksum Offloads in the Linux Networking Stack, Linux documentation: Documentation/networking/checksum-offloads.txt

Authors' Addresses

Tom Herbert
Facebook
1 Hacker Way
Menlo Park, CA
US

EMail: tom@herbertland.com

Network Working Group
Internet-Draft
Intended status: Standard Track

L. Yong
Huawei USA
T. Herbert
Facebook
O. Zia
Microsoft

Expires: April 2017

October 28, 2016

Generic UDP Encapsulation (GUE) for Network Virtualization Overlay
draft-hy-nvo3-gue-4-nvo-04

Abstract

This document describes network virtualization overlay encapsulation scheme by use of Generic UDP Encapsulation (GUE) [GUE]. It allocates one GUE optional flag and defines a 32 bit field for Virtual Network Identifier (VNID).

Status of This Document

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <http://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on April 28, 2017.

Copyright Notice

Copyright (c) 2016 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with

respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction.....	3
2. Terminology.....	3
2.1. Requirements Language.....	3
3. Generic UDP Encapsulation (GUE) for NVO3.....	3
4. Encapsulation/Decapsulation Operations.....	5
4.1. Multi-Tenant Segregation.....	5
4.2. Tenant Broadcast and Multicast Packets.....	6
4.3. Fragmentation.....	6
4.4. GUE Header Security.....	6
4.5. Tenant Packet Encryption.....	6
5. IANA Considerations.....	7
6. Security Considerations.....	7
7. References.....	7
7.1. Normative References.....	7
7.2. Informative Reference.....	8
8. Authors' Addresses.....	8

1. Introduction

Network Virtualization Overlay (NVO3) [RFC7365] provides a framework for a virtual network solution over an IP network in a DC with multi-tenant environment. Virtual network packets, i.e. tenant packets, between any pair of Network Virtualization Edges (NVE) are encapsulated at ingress NVE, sent from ingress NVE to egress NVE as IP packets, and decapsulated at egress NVE. This is known as a tunnel mechanism. This draft specifies use of Generic UDP Encapsulation (GUE) [GUE] for NVO3 packet encapsulation.

GUE [GUE] as a generic UDP encapsulation provides several merits for NVO3 encapsulation. Hence, underlay IP network treats it the same as other UDP applications, that are well supported by both IPv4 and IPv6 underlay networks. GUE provides strong security transport options [GUEEXT] that NVO3 can leverage. In addition, GUE supports other options that NVO3 may use such as private data and extensibility. In addition, GUE control flag can be used for NVO3 OAM message.

This document requests one flag (1 bit) from GUE optional flag field for Network Virtualization Overlay (NVO3) indication and specifies a 32 bit field for virtual network identifier in GUE optional fields. It describes use of GUE security options in NVO3.

2. Terminology

The terms defined in [GUE], [RFC7365] are used in this document.

2.1. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

3. Generic UDP Encapsulation (GUE) for NVO3

Generic UDP Encapsulation adds a 32 bit basic GUE header after UDP header. GUE header contains some key fields that a UDP tunnel application needs. These key fields are version, control message indication (c), Header Length (HLen), and Protocol Type (or ctype). It also contains some optional flags that are reserved for optional features at a UDP tunnel.

This document proposes to allocate one flag bit from GUE optional flags for the Network Virtualization Overlay (NVO3) and defines a 32 bit field for NVO3 in GUE optional fields when the flag bit is set. GUE based NVO3 encapsulation format is shown in Figure 1.

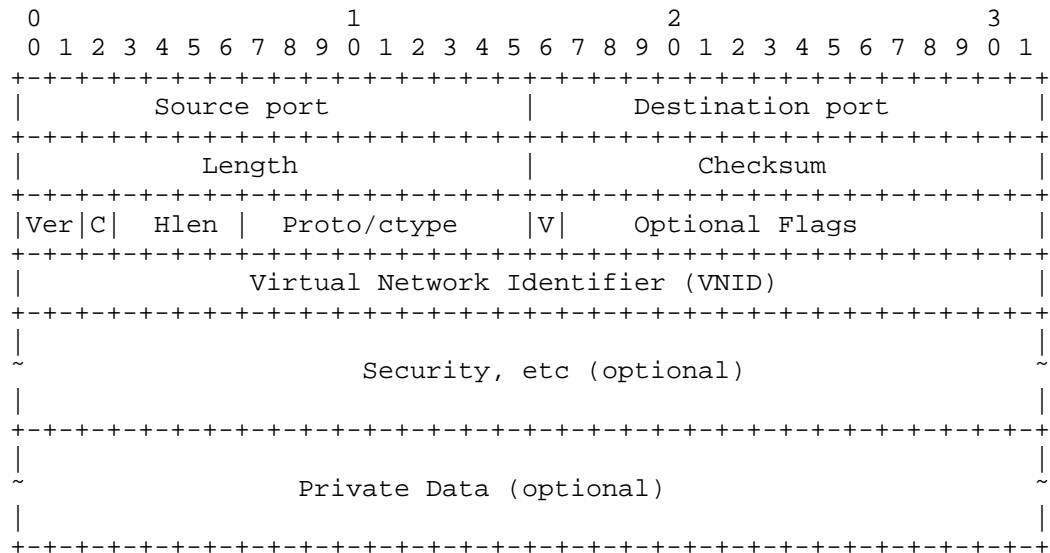


Figure 1 GUE based NVO3 Encapsulation Format

- o 'V': Virtualization flag. Indicates presence of the Virtual Network Identifier (VNID) field in GUE optional fields. This flag MUST be set when GUE is used for network virtualization overlay (NVO3).
- o Virtual Network ID (4 octets): a 32 bit field is used to identify a virtual network that the packet belongs to. This field MUST be present when 'V' virtualization flag is set; and MUST NOT present when 'V' flag is clear.

NVO3 implementation may carry private data in the private data field. It must follow the rules specified in [GUE] when inserting private data in GUE header.

NVO3 may allocate other flags and fields in GUE header for NVO3 purpose and MUST follow the flag/field allocation rules specified in [GUE].

The usage of the key fields in the GUE header [GUE] for NVO3 encapsulation is described as below:

- o Ver: Set to 0x0 to indicate version zero of GUE. Packets received by the decapsulator with non-zero version MUST be dropped.
- o Control flag: When set, indicates that the packet contains a control message. OAM packets for the virtual network instance can be carried as a control message. NOV3 OAM packet format and mechanisms will be specified in a separated document.
- o Hlen: length of (optional fields + private field (byte))/4.
- o Proto/ctype: Contain the protocol of the encapsulated payload packet, i.e. next header or control message type (ctype) when Control flag is set. The next header begins at the offset provided by Hlen. For network virtualization, the payload protocol can be Ethernet, IPv4, IPv6, or 59 (NULL). The VNID can be used with ctype to direct control message for the VN layer.

UDP header field MUST be set per [GUE]. The checksum and length implementation MUST be compliant with GUE implementation [GUE].

NVO3 can use GUE specified optional functions to improve the transport such as GUE security option [GUEEXT], GUE checksum option [GUEEXT], etc. When using a GUE specified option, NVO3 implementation MUST be compliant with the corresponding specification.

4. Encapsulation/Decapsulation Operations

The network virtualization encapsulation by use of GUE applies to both IPv4 and IPv6 underlay networks. The outer source and destination IP addresses MUST be ingress NVE and egress NVE IP addresses respectively. Ingress NVE adds UDP and GUE headers on the payload packet with the required fields as described in Section 3. NVE encapsulation and decapsulation process MUST be compliant with GUE implementation specification [GUE]. If ingress and egress NVE implement GUE options, they MUST be compliant with the corresponding GUE option specification.

4.1. Multi-Tenant Segregation

Ingress NVE MUST set option 'V' and insert Virtual Network Identifier (VNID) into the corresponding option field when encapsulating tenant packets. A GUE tunnel can carry the payload packets that are from different tenant networks simultaneously.

Egress NVE MUST use the VNID in GUE header to identify the tenant network that the payload packet is associated to and forward to the packet to corresponding tenant network. All 32 bits can be used for VNID.

4.2. Tenant Broadcast and Multicast Packets

If tenant packet is L2 broadcast/multicast, or L3 multicast packet, depending on which multicast solution NVO3 deploys [NVO3MFRWK], the packet may be carried by a set of point-to-point GUE tunnels, or a point-to-multipoint GUE tunnel. In the latter case, multicast IP address is used as outer destination address.

The mapping of inner broadcast/multicast group to IP multicast group can be manually configured or based on an algorithm, which is outside the scope of this document.

4.3. Fragmentation

To gain the performance and simplification, NVO3 SHOULD avoid packet fragmentation. Manual configuration or negotiation with tenant systems can ensure that the MTU of the physical networks is greater than or equal to the MTU of the encapsulated network plus GUE header. It is strongly RECOMMENDED Path MTU Discovery [RFC1191] [RFC1981] to be used by setting the DF bit in the IP header when GUE packets are carried by IPv4 (this is the default with IPv6). In a case, it can't avoid packet fragmentation; GUE fragmentation option can be used [GUEEXT].

4.4. GUE Header Security

NVO3 is expected to operate in multi-tenant environment, so security is extreme important. Security can be provided by DC networking and/or by NVO3. NVO3 can use GUE security options [GUEEXT]. When NVO3 use GUE security option, ingress NVE has to set the security flag and insert a key value in the security field [GUEEXT], egress NVE has to validate the key prior to packet decapitation process. If the key validation fails, the packet will be dropped [GUEEXT]. The key value used between ingress and egress NVE can be managed by NVA or generated algorithm at NVEs. This mechanism will be described in future version.

4.5. Tenant Packet Encryption

To prevent tenant packet from eavesdropping, tampering, or message forgery, NVO3 can adopt GUE payload encryption mechanism. To encrypt tenant packets, ingress NVE sets GUE payload transform flag and adds

32 bit payload transform field in GUE header. The payload type MUST be filled at the payload transform field and the protocol field in GUE base header MUST be set to 59 "No next header"[GUEEXT]. Both ingress NVE and egress NVE MUST implement the encryption mechanism as described in [GUEEXT].

5. IANA Considerations

The document request IANA to allocate the first bit in the registry of GUE optional flag fields for Network Virtualization Flag and register 32 bit field on GUE option field registry for Network Virtualization Identifier (VNID).

6. Security Considerations

When Network Virtualization Edge (NVE) uses the UDP tunnel mechanism specified in GUE [GUE], it faces the same security concern stated in Section of Security Considerations in [GUE] and can leverage GUE secure transport mechanisms [GUEEXT] for secure transport over the underlay IP network.

GUE provides two optional security functions. One is origin authentication and integrity protection between encapsulator and decapsulator, which protects Denial of Service (DoS) attacks; another is GUE payload encryption, which prevents the payload from eavesdropping, tampering, or message forgery. The two functions can be used together or independently according to the deployment environment. NVO3 virtual network identifier (VNID) is encoded in GUE header that can be protected by origin authentication.

7. References

7.1. Normative References

- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC2119, March 1997.
- [RFC7365] Lasserre, M., et al, "Framework for Data Center (DC) Network Virtualization".
- [GUE] Herbert T., Yong, L., Zia, O., "Generic UNP Encapsulation", draft-ietf-nvo3-gue, work in progress.

[GUEEXT] Herbert, T., Yong, L., Templin, F., "Extensions for Generic UDP Encapsulation", draft-herbert-gue-extensions, work in progress.

7.2. Informative Reference

[RFC1191] Mogul, J. and S. Deering, "Path MTU discovery", RFC 1191, November 1990.

[RFC1981] McCann, J., Deering, S., and J. Mogul, "Path MTU Discovery for IP version 6", RFC 1981, August 1996.

[NVO3MFRWK] Ghanwani, A., Dunbar, L., et al "A Framework for Multicast in Network Virtualization Overlays", draft-ietf-nov3-mcast-framework, work in progress.

8. Authors' Addresses

Lucy Yong
Huawei USA
5340 Legacy Dr.
Plano, TX 75024
US

Email: lucy.yong@huawei.com

Tom Herbert
Google
1600 Amphitheatre Parkway
Mountain View, CA
US

Email: therbert@google.com

Osama Zia
Microsoft
1 Microsoft Way
Redmond, WA 98029
US

Email: osamaz@microsoft.com

Internet Engineering Task Force
Internet-Draft
Intended status: Informational
Expires: March 24, 2017

D. Black
Dell EMC
J. Hudson
Independent
L. Kreeger
Cisco
M. Lasserre
Independent
T. Narten
IBM
September 20, 2016

An Architecture for Data Center Network Virtualization Overlays (NVO3)
draft-ietf-nvo3-arch-08

Abstract

This document presents a high-level overview architecture for building data center network virtualization overlay (NVO3) networks. The architecture is given at a high-level, showing the major components of an overall system. An important goal is to divide the space into individual smaller components that can be implemented independently with clear inter-component interfaces and interactions. It should be possible to build and implement individual components in isolation and have them interoperate with other independently implemented components. That way, implementers have flexibility in implementing individual components and can optimize and innovate within their respective components without requiring changes to other components.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <http://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on March 24, 2017.

Copyright Notice

Copyright (c) 2016 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	3
2. Terminology	3
3. Background	4
3.1. VN Service (L2 and L3)	6
3.1.1. VLAN Tags in L2 Service	7
3.1.2. Packet Lifetime Considerations	7
3.2. Network Virtualization Edge (NVE)	8
3.3. Network Virtualization Authority (NVA)	9
3.4. VM Orchestration Systems	10
4. Network Virtualization Edge (NVE)	11
4.1. NVE Co-located With Server Hypervisor	11
4.2. Split-NVE	12
4.2.1. Tenant VLAN handling in Split-NVE Case	13
4.3. NVE State	13
4.4. Multi-Homing of NVEs	14
4.5. Virtual Access Point (VAP)	15
5. Tenant System Types	15
5.1. Overlay-Aware Network Service Appliances	15
5.2. Bare Metal Servers	16
5.3. Gateways	16
5.3.1. Gateway Taxonomy	17
5.3.1.1. L2 Gateways (Bridging)	17
5.3.1.2. L3 Gateways (Only IP Packets)	17
5.4. Distributed Inter-VN Gateways	18
5.5. ARP and Neighbor Discovery	19
6. NVE-NVE Interaction	19
7. Network Virtualization Authority	20
7.1. How an NVA Obtains Information	20
7.2. Internal NVA Architecture	21
7.3. NVA External Interface	21
8. NVE-to-NVA Protocol	23

8.1. NVE-NVA Interaction Models	23
8.2. Direct NVE-NVA Protocol	24
8.3. Propagating Information Between NVEs and NVAs	24
9. Federated NVAs	25
9.1. Inter-NVA Peering	28
10. Control Protocol Work Areas	28
11. NVO3 Data Plane Encapsulation	28
12. Operations, Administration and Maintenance (OAM)	29
13. Summary	30
14. Acknowledgments	30
15. IANA Considerations	30
16. Security Considerations	30
17. Informative References	31
Authors' Addresses	33

1. Introduction

This document presents a high-level architecture for building data center network virtualization overlay (NVO3) networks. The architecture is given at a high-level, showing the major components of an overall system. An important goal is to divide the space into smaller individual components that can be implemented independently with clear inter-component interfaces and interactions. It should be possible to build and implement individual components in isolation and have them interoperate with other independently implemented components. That way, implementers have flexibility in implementing individual components and can optimize and innovate within their respective components without requiring changes to other components.

The motivation for overlay networks is given in "Problem Statement: Overlays for Network Virtualization" [RFC7364]. "Framework for DC Network Virtualization" [RFC7365] provides a framework for discussing overlay networks generally and the various components that must work together in building such systems. This document differs from the framework document in that it doesn't attempt to cover all possible approaches within the general design space. Rather, it describes one particular approach that the NVO3 WG has focused on.

2. Terminology

This document uses the same terminology as [RFC7365]. In addition, the following terms are used:

NV Domain A Network Virtualization Domain is an administrative construct that defines a Network Virtualization Authority (NVA), the set of Network Virtualization Edges (NVEs) associated with that NVA, and the set of virtual networks the NVA manages and supports. NVEs are associated with a (logically centralized) NVA,

and an NVE supports communication for any of the virtual networks in the domain.

NV Region A region over which information about a set of virtual networks is shared. The degenerate case of a single NV Domain corresponds to an NV region corresponding to that domain. The more interesting case occurs when two or more NV Domains share information about part or all of a set of virtual networks that they manage. Two NVAs share information about particular virtual networks for the purpose of supporting connectivity between tenants located in different NV Domains. NVAs can share information about an entire NV domain, or just individual virtual networks.

Tenant System Interface (TSI) Interface to a Virtual Network as presented to a Tenant System (TS, see [RFC7365]). The TSI logically connects to the NVE via a Virtual Access Point (VAP). To the Tenant System, the TSI is like a Network Interface Card (NIC); the TSI presents itself to a Tenant System as a normal network interface.

VLAN Unless stated otherwise, the terms VLAN and VLAN Tag are used in this document to denote a C-VLAN [IEEE-802.1Q] and the terms are used interchangeably to improve readability.

3. Background

Overlay networks are an approach for providing network virtualization services to a set of Tenant Systems (TSs) [RFC7365]. With overlays, data traffic between tenants is tunneled across the underlying data center's IP network. The use of tunnels provides a number of benefits by decoupling the network as viewed by tenants from the underlying physical network across which they communicate. Additional discussion of some NVO3 use cases can be found in [I-D.ietf-nvo3-use-case].

Tenant Systems connect to Virtual Networks (VNs), with each VN having associated attributes defining properties of the network, such as the set of members that connect to it. Tenant Systems connected to a virtual network typically communicate freely with other Tenant Systems on the same VN, but communication between Tenant Systems on one VN and those external to the VN (whether on another VN or connected to the Internet) is carefully controlled and governed by policy. The NVO3 architecture does not impose any restrictions to the application of policy controls even within a VN.

A Network Virtualization Edge (NVE) [RFC7365] is the entity that implements the overlay functionality. An NVE resides at the boundary

between a Tenant System and the overlay network as shown in Figure 1. An NVE creates and maintains local state about each Virtual Network for which it is providing service on behalf of a Tenant System.

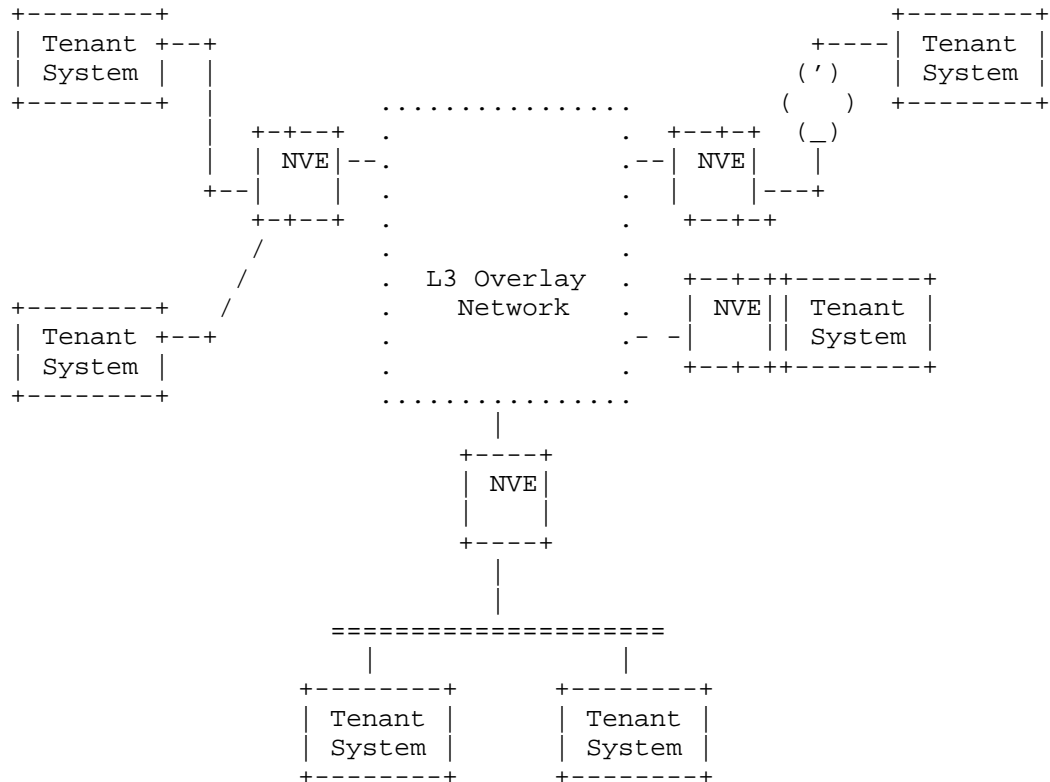


Figure 1: NVO3 Generic Reference Model

The following subsections describe key aspects of an overlay system in more detail. Section 3.1 describes the service model (Ethernet vs. IP) provided to Tenant Systems. Section 3.2 describes NVEs in more detail. Section 3.3 introduces the Network Virtualization Authority, from which NVEs obtain information about virtual networks. Section 3.4 provides background on Virtual Machine (VM) orchestration systems and their use of virtual networks.

3.1. VN Service (L2 and L3)

A Virtual Network provides either L2 or L3 service to connected tenants. For L2 service, VNs transport Ethernet frames, and a Tenant System is provided with a service that is analogous to being connected to a specific L2 C-VLAN. L2 broadcast frames are generally delivered to all (and multicast frames delivered to a subset of) the other Tenant Systems on the VN. To a Tenant System, it appears as if they are connected to a regular L2 Ethernet link. Within the NVO3 architecture, tenant frames are tunneled to remote NVEs based on the MAC addresses of the frame headers as originated by the Tenant System. On the underlay, NVO3 packets are forwarded between NVEs based on the outer addresses of tunneled packets.

For L3 service, VNs are routed networks that transport IP datagrams, and a Tenant System is provided with a service that supports only IP traffic. Within the NVO3 architecture, tenant frames are tunneled to remote NVEs based on the IP addresses of the packet originated by the Tenant System; any L2 destination addresses provided by Tenant Systems are effectively ignored by the NVEs and overlay network. For L3 service, the Tenant System will be configured with an IP subnet that is effectively a point-to-point link, i.e., having only the Tenant System and a next-hop router address on it.

L2 service is intended for systems that need native L2 Ethernet service and the ability to run protocols directly over Ethernet (i.e., not based on IP). L3 service is intended for systems in which all the traffic can safely be assumed to be IP. It is important to note that whether an NVO3 network provides L2 or L3 service to a Tenant System, the Tenant System does not generally need to be aware of the distinction. In both cases, the virtual network presents itself to the Tenant System as an L2 Ethernet interface. An Ethernet interface is used in both cases simply as a widely supported interface type that essentially all Tenant Systems already support. Consequently, no special software is needed on Tenant Systems to use an L3 vs. an L2 overlay service.

NVO3 can also provide a combined L2 and L3 service to tenants. A combined service provides L2 service for intra-VN communication, but also provides L3 service for L3 traffic entering or leaving the VN. Architecturally, the handling of a combined L2/L3 service within the NVO3 architecture is intended to match what is commonly done today in non-overlay environments by devices providing a combined bridge/router service. With combined service, the virtual network itself retains the semantics of L2 service and all traffic is processed according to its L2 semantics. In addition, however, traffic requiring IP processing is also processed at the IP level.

The IP processing for a combined service can be implemented on a standalone device attached to the virtual network (e.g., an IP router) or implemented locally on the NVE (see Section 5.4 on Distributed Gateways). For unicast traffic, NVE implementation of a combined service may result in a packet being delivered to another Tenant System attached to the same NVE (on either the same or a different VN) or tunneled to a remote NVE, or even forwarded outside the NV domain. For multicast or broadcast packets, the combination of NVE L2 and L3 processing may result in copies of the packet receiving both L2 and L3 treatments to realize delivery to all of the destinations involved. This distributed NVE implementation of IP routing results in the same network delivery behavior as if the L2 processing of the packet included delivery of the packet to an IP router attached to the L2 VN as a Tenant System, with the router having additional network attachments to other networks, either virtual or not.

3.1.1. VLAN Tags in L2 Service

An NVO3 L2 virtual network service may include encapsulated L2 VLAN tags provided by a Tenant System, but does not use encapsulated tags in deciding where and how to forward traffic. Such VLAN tags can be passed through, so that Tenant Systems that send or expect to receive them can be supported as appropriate.

The processing of VLAN tags that an NVE receives from a TS is controlled by settings associated with the VAP. Just as in the case with ports on Ethernet switches, a number of settings are possible. For example, C-TAGs can be passed through transparently, they could always be stripped upon receipt from a Tenant System, they could be compared against a list of explicitly configured tags, etc.

Note that that there are additional considerations when VLAN tags are used to identify both the VN and a Tenant System VLAN within that VN, as described in Section 4.2.1 below.

3.1.2. Packet Lifetime Considerations

For L3 service, Tenant Systems should expect the IPv4 TTL (Time to Live) or IPv6 Hop Limit in the packets they send to be decremented by at least 1. For L2 service, neither the TTL nor the Hop Limit (when the packet is IP) are modified. The underlay network manages TTLs and Hop Limits in the outer IP encapsulation - the values in these fields could be independent from or related to the values in the same fields of tenant IP packets.

3.2. Network Virtualization Edge (NVE)

Tenant Systems connect to NVEs via a Tenant System Interface (TSI). The TSI logically connects to the NVE via a Virtual Access Point (VAP) and each VAP is associated with one Virtual Network as shown in Figure 2. To the Tenant System, the TSI is like a NIC; the TSI presents itself to a Tenant System as a normal network interface. On the NVE side, a VAP is a logical network port (virtual or physical) into a specific virtual network. Note that two different Tenant Systems (and TSIs) attached to a common NVE can share a VAP (e.g., TS1 and TS2 in Figure 2) so long as they connect to the same Virtual Network.

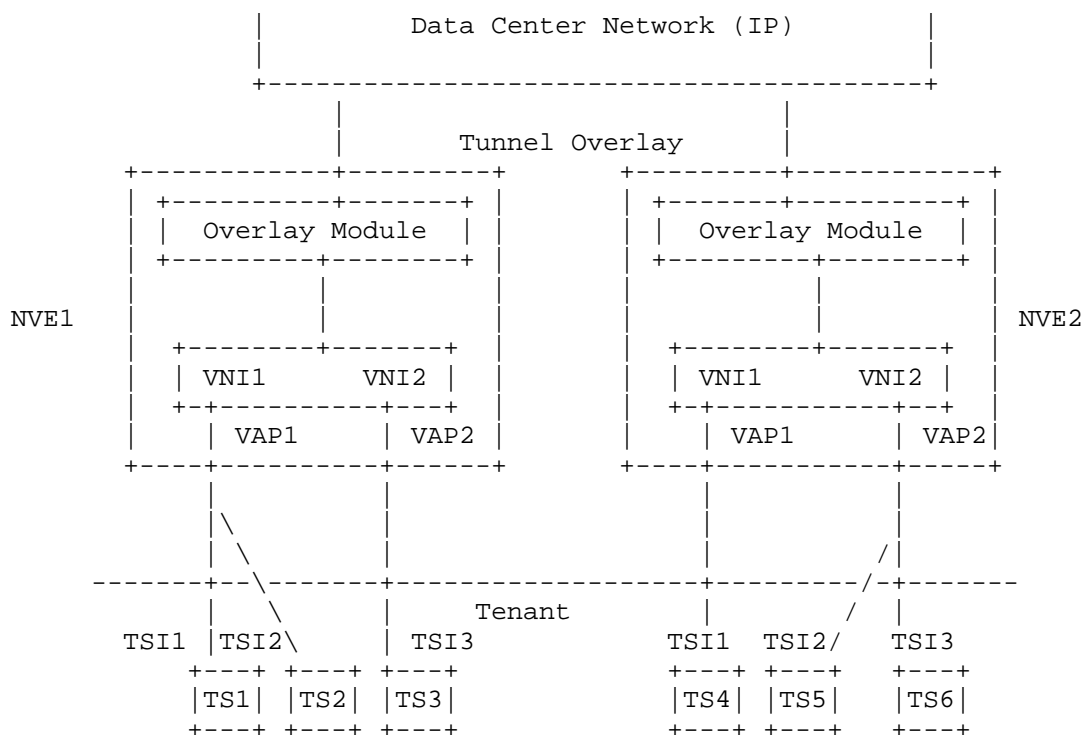


Figure 2: NVE Reference Model

The Overlay Module performs the actual encapsulation and decapsulation of tunneled packets. The NVE maintains state about the virtual networks it is a part of so that it can provide the Overlay Module with such information as the destination address of the NVE to tunnel a packet to and the Context ID that should be placed in the

encapsulation header to identify the virtual network that a tunneled packet belongs to.

On the data center network side, the NVE sends and receives native IP traffic. When ingressing traffic from a Tenant System, the NVE identifies the egress NVE to which the packet should be sent, adds an overlay encapsulation header, and sends the packet on the underlay network. When receiving traffic from a remote NVE, an NVE strips off the encapsulation header, and delivers the (original) packet to the appropriate Tenant System. When the source and destination Tenant System are on the same NVE, no encapsulation is needed and the NVE forwards traffic directly.

Conceptually, the NVE is a single entity implementing the NVO3 functionality. In practice, there are a number of different implementation scenarios, as described in detail in Section 4.

3.3. Network Virtualization Authority (NVA)

Address dissemination refers to the process of learning, building and distributing the mapping/forwarding information that NVEs need in order to tunnel traffic to each other on behalf of communicating Tenant Systems. For example, in order to send traffic to a remote Tenant System, the sending NVE must know the destination NVE for that Tenant System.

One way to build and maintain mapping tables is to use learning, as 802.1 bridges do [IEEE-802.1Q]. When forwarding traffic to multicast or unknown unicast destinations, an NVE could simply flood traffic. While flooding works, it can lead to traffic hot spots and can lead to problems in larger networks (e.g., excessive amounts of flooded traffic).

Alternatively, to reduce the scope of where flooding must take place, or to eliminate it all together, NVEs can make use of a Network Virtualization Authority (NVA). An NVA is the entity that provides address mapping and other information to NVEs. NVEs interact with an NVA to obtain any required address mapping information they need in order to properly forward traffic on behalf of tenants. The term NVA refers to the overall system, without regards to its scope or how it is implemented. NVAs provide a service, and NVEs access that service via an NVE-to-NVA protocol as discussed in Section 8.

Even when an NVA is present, Ethernet bridge MAC address learning could be used as a fallback mechanism, should the NVA be unable to provide an answer or for other reasons. This document does not consider flooding approaches in detail, as there are a number of benefits in using an approach that depends on the presence of an NVA.

For the rest of this document, it is assumed that an NVA exists and will be used. NVAs are discussed in more detail in Section 7.

3.4. VM Orchestration Systems

VM orchestration systems manage server virtualization across a set of servers. Although VM management is a separate topic from network virtualization, the two areas are closely related. Managing the creation, placement, and movement of VMs also involves creating, attaching to and detaching from virtual networks. A number of existing VM orchestration systems have incorporated aspects of virtual network management into their systems.

Note also, that although this section uses the term "VM" and "hypervisor" throughout, the same issues apply to other virtualization approaches, including Linux Containers (LXC), BSD Jails, Network Service Appliances as discussed in Section 5.1, etc.. From an NVO3 perspective, it should be assumed that where the document uses the term "VM" and "hypervisor", the intention is that the discussion also applies to other systems, where, e.g., the host operating system plays the role of the hypervisor in supporting virtualization, and a container plays the equivalent role as a VM.

When a new VM image is started, the VM orchestration system determines where the VM should be placed, interacts with the hypervisor on the target server to load and start the VM and controls when a VM should be shutdown or migrated elsewhere. VM orchestration systems also have knowledge about how a VM should connect to a network, possibly including the name of the virtual network to which a VM is to connect. The VM orchestration system can pass such information to the hypervisor when a VM is instantiated. VM orchestration systems have significant (and sometimes global) knowledge over the domain they manage. They typically know on what servers a VM is running, and meta data associated with VM images can be useful from a network virtualization perspective. For example, the meta data may include the addresses (MAC and IP) the VMs will use and the name(s) of the virtual network(s) they connect to.

VM orchestration systems run a protocol with an agent running on the hypervisor of the servers they manage. That protocol can also carry information about what virtual network a VM is associated with. When the orchestrator instantiates a VM on a hypervisor, the hypervisor interacts with the NVE in order to attach the VM to the virtual networks it has access to. In general, the hypervisor will need to communicate significant VM state changes to the NVE. In the reverse direction, the NVE may need to communicate network connectivity information back to the hypervisor. Examples of deployed VM orchestration systems include VMware's vCenter Server, Microsoft's

System Center Virtual Machine Manager, and systems based on OpenStack and its associated plugins (e.g., Nova and Neutron). Each can pass information about what virtual networks a VM connects to down to the hypervisor. The protocol used between the VM orchestration system and hypervisors is generally proprietary.

It should be noted that VM orchestration systems may not have direct access to all networking related information a VM uses. For example, a VM may make use of additional IP or MAC addresses that the VM management system is not aware of.

4. Network Virtualization Edge (NVE)

As introduced in Section 3.2 an NVE is the entity that implements the overlay functionality. This section describes NVEs in more detail. An NVE will have two external interfaces:

Tenant System Facing: On the Tenant System facing side, an NVE interacts with the hypervisor (or equivalent entity) to provide the NVO3 service. An NVE will need to be notified when a Tenant System "attaches" to a virtual network (so it can validate the request and set up any state needed to send and receive traffic on behalf of the Tenant System on that VN). Likewise, an NVE will need to be informed when the Tenant System "detaches" from the virtual network so that it can reclaim state and resources appropriately.

Data Center Network Facing: On the data center network facing side, an NVE interfaces with the data center underlay network, sending and receiving tunneled packets to and from the underlay. The NVE may also run a control protocol with other entities on the network, such as the Network Virtualization Authority.

4.1. NVE Co-located With Server Hypervisor

When server virtualization is used, the entire NVE functionality will typically be implemented as part of the hypervisor and/or virtual switch on the server. In such cases, the Tenant System interacts with the hypervisor and the hypervisor interacts with the NVE. Because the interaction between the hypervisor and NVE is implemented entirely in software on the server, there is no "on-the-wire" protocol between Tenant Systems (or the hypervisor) and the NVE that needs to be standardized. While there may be APIs between the NVE and hypervisor to support necessary interaction, the details of such an API are not in-scope for the NVO3 WG at the time of publication of this memo.

Implementing NVE functionality entirely on a server has the disadvantage that server CPU resources must be spent implementing the NVO3 functionality. Experimentation with overlay approaches and previous experience with TCP and checksum adapter offloads suggests that offloading certain NVE operations (e.g., encapsulation and decapsulation operations) onto the physical network adapter can produce performance advantages. As has been done with checksum and/or TCP server offload and other optimization approaches, there may be benefits to offloading common operations onto adapters where possible. Just as important, the addition of an overlay header can disable existing adapter offload capabilities that are generally not prepared to handle the addition of a new header or other operations associated with an NVE.

While the exact details of how to split the implementation of specific NVE functionality between a server and its network adapters is an implementation matter and outside the scope of IETF standardization, the NVO3 architecture should be cognizant of and support such separation. Ideally, it may even be possible to bypass the hypervisor completely on critical data path operations so that packets between a Tenant System and its VN can be sent and received without having the hypervisor involved in each individual packet operation.

4.2. Split-NVE

Another possible scenario leads to the need for a split NVE implementation. An NVE running on a server (e.g. within a hypervisor) could support NVO3 service towards the tenant, but not perform all NVE functions (e.g., encapsulation) directly on the server; some of the actual NVO3 functionality could be implemented on (i.e., offloaded to) an adjacent switch to which the server is attached. While one could imagine a number of link types between a server and the NVE, one simple deployment scenario would involve a server and NVE separated by a simple L2 Ethernet link. A more complicated scenario would have the server and NVE separated by a bridged access network, such as when the NVE resides on a top of rack (ToR) switch, with an embedded switch residing between servers and the ToR switch.

For the split NVE case, protocols will be needed that allow the hypervisor and NVE to negotiate and setup the necessary state so that traffic sent across the access link between a server and the NVE can be associated with the correct virtual network instance. Specifically, on the access link, traffic belonging to a specific Tenant System would be tagged with a specific VLAN C-TAG that identifies which specific NVO3 virtual network instance it connects to. The hypervisor-NVE protocol would negotiate which VLAN C-TAG to

use for a particular virtual network instance. More details of the protocol requirements for functionality between hypervisors and NVEs can be found in [I-D.ietf-nvo3-nve-nva-cp-req].

4.2.1. Tenant VLAN handling in Split-NVE Case

Preserving tenant VLAN tags across an NVO3 VN as described in Section 3.1.1 poses additional complications in the split-NVE case. The portion of the NVE that performs the encapsulation function needs access to the specific VLAN tags that the Tenant System is using in order to include them in the encapsulated packet. When an NVE is implemented entirely within the hypervisor, the NVE has access to the complete original packet (including any VLAN tags) sent by the tenant. In the split-NVE case, however, the VLAN tag used between the hypervisor and offloaded portions of the NVE normally only identifies the specific VN that traffic belongs to. In order to allow a tenant to preserve VLAN information from end to end between Tenant Systems in the split-NVE case, additional mechanisms would be needed (e.g., carry an additional VLAN tag by carrying both a C-Tag and an S-Tag as specified in [IEEE-802.1Q] where the C-Tag identifies the tenant VLAN end-to-end and the S-Tag identifies the VN locally between each Tenant System and the corresponding NVE).

4.3. NVE State

NVEs maintain internal data structures and state to support the sending and receiving of tenant traffic. An NVE may need some or all of the following information:

1. An NVE keeps track of which attached Tenant Systems are connected to which virtual networks. When a Tenant System attaches to a virtual network, the NVE will need to create or update local state for that virtual network. When the last Tenant System detaches from a given VN, the NVE can reclaim state associated with that VN.
2. For tenant unicast traffic, an NVE maintains a per-VN table of mappings from Tenant System (inner) addresses to remote NVE (outer) addresses.
3. For tenant multicast (or broadcast) traffic, an NVE maintains a per-VN table of mappings and other information on how to deliver tenant multicast (or broadcast) traffic. If the underlying network supports IP multicast, the NVE could use IP multicast to deliver tenant traffic. In such a case, the NVE would need to know what IP underlay multicast address to use for a given VN. Alternatively, if the underlying network does not support multicast, a source NVE could use unicast replication to deliver

traffic. In such a case, an NVE would need to know which remote NVEs are participating in the VN. An NVE could use both approaches, switching from one mode to the other depending on such factors as bandwidth efficiency and group membership sparseness. [I-D.ietf-nvo3-mcast-framework] discusses the subject of multicast handling in NVO3 in further detail.

4. An NVE maintains necessary information to encapsulate outgoing traffic, including what type of encapsulation and what value to use for a Context ID to identify the VN within the encapsulation header.
5. In order to deliver incoming encapsulated packets to the correct Tenant Systems, an NVE maintains the necessary information to map incoming traffic to the appropriate VAP (i.e., Tenant System Interface).
6. An NVE may find it convenient to maintain additional per-VN information such as QoS settings, Path MTU information, ACLs, etc.

4.4. Multi-Homing of NVEs

NVEs may be multi-homed. That is, an NVE may have more than one IP address associated with it on the underlay network. Multihoming happens in two different scenarios. First, an NVE may have multiple interfaces connecting it to the underlay. Each of those interfaces will typically have a different IP address, resulting in a specific Tenant Address (on a specific VN) being reachable through the same NVE but through more than one underlay IP address. Second, a specific tenant system may be reachable through more than one NVE, each having one or more underlay addresses. In both cases, NVE address mapping functionality needs to support one-to-many mappings and enable a sending NVE to (at a minimum) be able to fail over from one IP address to another, e.g., should a specific NVE underlay address become unreachable.

Finally, multi-homed NVEs introduce complexities when source unicast replication is used to implement tenant multicast as described in Section 4.3. Specifically, an NVE should only receive one copy of a replicated packet.

Multi-homing is needed to support important use cases. First, a bare metal server may have multiple uplink connections to either the same or different NVEs. Having only a single physical path to an upstream NVE, or indeed, having all traffic flow through a single NVE would be considered unacceptable in highly-resilient deployment scenarios that seek to avoid single points of failure. Moreover, in today's

networks, the availability of multiple paths would require that they be usable in an active-active fashion (e.g., for load balancing).

4.5. Virtual Access Point (VAP)

The VAP is the NVE-side of the interface between the NVE and the TS. Traffic to and from the tenant flows through the VAP. If an NVE runs into difficulties sending traffic received on the VAP, it may need to signal such errors back to the VAP. Because the VAP is an emulation of a physical port, its ability to signal NVE errors is limited and lacks sufficient granularity to reflect all possible errors an NVE may encounter (e.g., inability reach a particular destination). Some errors, such as an NVE losing all of its connections to the underlay, could be reflected back to the VAP by effectively disabling it. This state change would reflect itself on the TS as an interface going down, allowing the TS to implement interface error handling, e.g., failover, in the same manner as when a physical interfaces becomes disabled.

5. Tenant System Types

This section describes a number of special Tenant System types and how they fit into an NVO3 system.

5.1. Overlay-Aware Network Service Appliances

Some Network Service Appliances [I-D.ietf-nvo3-nve-nva-cp-req] (virtual or physical) provide tenant-aware services. That is, the specific service they provide depends on the identity of the tenant making use of the service. For example, firewalls are now becoming available that support multi-tenancy where a single firewall provides virtual firewall service on a per-tenant basis, using per-tenant configuration rules and maintaining per-tenant state. Such appliances will be aware of the VN an activity corresponds to while processing requests. Unlike server virtualization, which shields VMs from needing to know about multi-tenancy, a Network Service Appliance may explicitly support multi-tenancy. In such cases, the Network Service Appliance itself will be aware of network virtualization and either embed an NVE directly, or implement a split NVE as described in Section 4.2. Unlike server virtualization, however, the Network Service Appliance may not be running a hypervisor and the VM orchestration system may not interact with the Network Service Appliance. The NVE on such appliances will need to support a control plane to obtain the necessary information needed to fully participate in an NV Domain.

5.2. Bare Metal Servers

Many data centers will continue to have at least some servers operating as non-virtualized (or "bare metal") machines running a traditional operating system and workload. In such systems, there will be no NVE functionality on the server, and the server will have no knowledge of NVO3 (including whether overlays are even in use). In such environments, the NVE functionality can reside on the first-hop physical switch. In such a case, the network administrator would (manually) configure the switch to enable the appropriate NVO3 functionality on the switch port connecting the server and associate that port with a specific virtual network. Such configuration would typically be static, since the server is not virtualized, and once configured, is unlikely to change frequently. Consequently, this scenario does not require any protocol or standards work.

5.3. Gateways

Gateways on VNs relay traffic onto and off of a virtual network. Tenant Systems use gateways to reach destinations outside of the local VN. Gateways receive encapsulated traffic from one VN, remove the encapsulation header, and send the native packet out onto the data center network for delivery. Outside traffic enters a VN in a reverse manner.

Gateways can be either virtual (i.e., implemented as a VM) or physical (i.e., as a standalone physical device). For performance reasons, standalone hardware gateways may be desirable in some cases. Such gateways could consist of a simple switch forwarding traffic from a VN onto the local data center network, or could embed router functionality. On such gateways, network interfaces connecting to virtual networks will (at least conceptually) embed NVE (or split-NVE) functionality within them. As in the case with Network Service Appliances, gateways may not support a hypervisor and will need an appropriate control plane protocol to obtain the information needed to provide NVO3 service.

Gateways handle several different use cases. For example, one use case consists of systems supporting overlays together with systems that do not (e.g., bare metal servers). Gateways could be used to connect legacy systems supporting, e.g., L2 VLANs, to specific virtual networks, effectively making them part of the same virtual network. Gateways could also forward traffic between a virtual network and other hosts on the data center network or relay traffic between different VNs. Finally, gateways can provide external connectivity such as Internet or VPN access.

5.3.1. Gateway Taxonomy

As can be seen from the discussion above, there are several types of gateways that can exist in an NVO3 environment. This section breaks them down into the various types that could be supported. Note that each of the types below could be implemented in either a centralized manner or distributed to co-exist with the NVEs.

5.3.1.1. L2 Gateways (Bridging)

L2 Gateways act as layer 2 bridges to forward Ethernet frames based on the MAC addresses present in them.

L2 VN to Legacy L2: This type of gateway bridges traffic between L2 VNs and other legacy L2 networks such as VLANs or L2 VPNs.

L2 VN to L2 VN: The main motivation for this type of gateway to create separate groups of Tenant Systems using L2 VNs such that the gateway can enforce network policies between each L2 VN.

5.3.1.2. L3 Gateways (Only IP Packets)

L3 Gateways forward IP packets based on the IP addresses present in the packets.

L3 VN to Legacy L2: This type of gateway forwards packets between L3 VNs and legacy L2 networks such as VLANs or L2 VPNs. The original sender's destination MAC address in any frames that the gateway forwards from a legacy L2 network would be the MAC address of the gateway.

L3 VN to Legacy L3: The type of gateway forwards packets between L3 VNs and legacy L3 networks. These legacy L3 networks could be local the data center, in the WAN, or an L3 VPN.

L3 VN to L2 VN: This type of gateway forwards packets on between L3 VNs and L2 VNs. The original sender's destination MAC address in any frames that the gateway forwards from a L2 VN would be the MAC address of the gateway.

L2 VN to L2 VN: This type of gateway acts similar to a traditional router that forwards between L2 interfaces. The original sender's destination MAC address in any frames that the gateway forwards from any of the L2 VNs would be the MAC address of the gateway.

L3 VN to L3 VN: The main motivation for this type of gateway to create separate groups of Tenant Systems using L3 VNs such that the gateway can enforce network policies between each L3 VN.

5.4. Distributed Inter-VN Gateways

The relaying of traffic from one VN to another deserves special consideration. Whether traffic is permitted to flow from one VN to another is a matter of policy, and would not (by default) be allowed unless explicitly enabled. In addition, NVAs are the logical place to maintain policy information about allowed inter-VN communication. Policy enforcement for inter-VN communication can be handled in (at least) two different ways. Explicit gateways could be the central point for such enforcement, with all inter-VN traffic forwarded to such gateways for processing. Alternatively, the NVA can provide such information directly to NVEs, by either providing a mapping for a target Tenant System (TS) on another VN, or indicating that such communication is disallowed by policy.

When inter-VN gateways are centralized, traffic between TSs on different VNs can take suboptimal paths, i.e., triangular routing results in paths that always traverse the gateway. In the worst case, traffic between two TSs connected to the same NVE can be hair-pinned through an external gateway. As an optimization, individual NVEs can be part of a distributed gateway that performs such relaying, reducing or completely eliminating triangular routing. In a distributed gateway, each ingress NVE can perform such relaying activity directly, so long as it has access to the policy information needed to determine whether cross-VN communication is allowed. Having individual NVEs be part of a distributed gateway allows them to tunnel traffic directly to the destination NVE without the need to take suboptimal paths.

The NVO3 architecture supports distributed gateways for the case of inter-VN communication. Such support requires that NVO3 control protocols include mechanisms for the maintenance and distribution of policy information about what type of cross-VN communication is allowed so that NVEs acting as distributed gateways can tunnel traffic from one VN to another as appropriate.

Distributed gateways could also be used to distribute other traditional router services to individual NVEs. The NVO3 architecture does not preclude such implementations, but does not define or require them as they are outside the scope of the NVO3 architecture.

5.5. ARP and Neighbor Discovery

For an L2 service, strictly speaking, special processing of Address Resolution Protocol (ARP) [RFC0826] (and IPv6 Neighbor Discovery (ND) [RFC4861]) is not required. ARP requests are broadcast, and an NVO3 can deliver ARP requests to all members of a given L2 virtual network, just as it does for any packet sent to an L2 broadcast address. Similarly, ND requests are sent via IP multicast, which NVO3 can support by delivering via L2 multicast. However, as a performance optimization, an NVE can intercept ARP (or ND) requests from its attached TSs and respond to them directly using information in its mapping tables. Since an NVE will have mechanisms for determining the NVE address associated with a given TS, the NVE can leverage the same mechanisms to suppress sending ARP and ND requests for a given TS to other members of the VN. The NVO3 architecture supports such a capability.

6. NVE-NVE Interaction

Individual NVEs will interact with each other for the purposes of tunneling and delivering traffic to remote TSs. At a minimum, a control protocol may be needed for tunnel setup and maintenance. For example, tunneled traffic may need to be encrypted or integrity protected, in which case it will be necessary to set up appropriate security associations between NVE peers. It may also be desirable to perform tunnel maintenance (e.g., continuity checks) on a tunnel in order to detect when a remote NVE becomes unreachable. Such generic tunnel setup and maintenance functions are not generally NVO3-specific. Hence, the NVO3 architecture expects to leverage existing tunnel maintenance protocols rather than defining new ones.

Some NVE-NVE interactions may be specific to NVO3 (and in particular be related to information kept in mapping tables) and agnostic to the specific tunnel type being used. For example, when tunneling traffic for TS-X to a remote NVE, it is possible that TS-X is not presently associated with the remote NVE. Normally, this should not happen, but there could be race conditions where the information an NVE has learned from the NVA is out-of-date relative to actual conditions. In such cases, the remote NVE could return an error or warning indication, allowing the sending NVE to attempt a recovery or otherwise attempt to mitigate the situation.

The NVE-NVE interaction could signal a range of indications, for example:

- o "No such TS here", upon a receipt of a tunneled packet for an unknown TS.

- o "TS-X not here, try the following NVE instead" (i.e., a redirect).
- o Delivered to correct NVE, but could not deliver packet to TS-X.

When an NVE receives information from a remote NVE that conflicts with the information it has in its own mapping tables, it should consult with the NVA to resolve those conflicts. In particular, it should confirm that the information it has is up-to-date, and it might indicate the error to the NVA, so as to nudge the NVA into following up (as appropriate). While it might make sense for an NVE to update its mapping table temporarily in response to an error from a remote NVE, any changes must be handled carefully as doing so can raise security considerations if the received information cannot be authenticated. That said, a sending NVE might still take steps to mitigate a problem, such as applying rate limiting to data traffic towards a particular NVE or TS.

7. Network Virtualization Authority

Before sending to and receiving traffic from a virtual network, an NVE must obtain the information needed to build its internal forwarding tables and state as listed in Section 4.3. An NVE can obtain such information from a Network Virtualization Authority.

The Network Virtualization Authority (NVA) is the entity that is expected to provide address mapping and other information to NVEs. NVEs can interact with an NVA to obtain any required information they need in order to properly forward traffic on behalf of tenants. The term NVA refers to the overall system, without regards to its scope or how it is implemented.

7.1. How an NVA Obtains Information

There are two primary ways in which an NVA can obtain the address dissemination information it manages. The NVA can obtain information either from the VM orchestration system, and/or directly from the NVEs themselves.

On virtualized systems, the NVA may be able to obtain the address mapping information associated with VMs from the VM orchestration system itself. If the VM orchestration system contains a master database for all the virtualization information, having the NVA obtain information directly to the orchestration system would be a natural approach. Indeed, the NVA could effectively be co-located with the VM orchestration system itself. In such systems, the VM orchestration system communicates with the NVE indirectly through the hypervisor.

However, as described in Section 4 not all NVEs are associated with hypervisors. In such cases, NVAs cannot leverage VM orchestration protocols to interact with an NVE and will instead need to peer directly with them. By peering directly with an NVE, NVAs can obtain information about the TSs connected to that NVE and can distribute information to the NVE about the VNs those TSs are associated with. For example, whenever a Tenant System attaches to an NVE, that NVE would notify the NVA that the TS is now associated with that NVE. Likewise when a TS detaches from an NVE, that NVE would inform the NVA. By communicating directly with NVEs, both the NVA and the NVE are able to maintain up-to-date information about all active tenants and the NVEs to which they are attached.

7.2. Internal NVA Architecture

For reliability and fault tolerance reasons, an NVA would be implemented in a distributed or replicated manner without single points of failure. How the NVA is implemented, however, is not important to an NVE so long as the NVA provides a consistent and well-defined interface to the NVE. For example, an NVA could be implemented via database techniques whereby a server stores address mapping information in a traditional (possibly replicated) database. Alternatively, an NVA could be implemented in a distributed fashion using an existing (or modified) routing protocol to maintain and distribute mappings. So long as there is a clear interface between the NVE and NVA, how an NVA is architected and implemented is not important to an NVE.

A number of architectural approaches could be used to implement NVAs themselves. NVAs manage address bindings and distribute them to where they need to go. One approach would be to use Border Gateway Protocol (BGP) [RFC4364] (possibly with extensions) and route reflectors. Another approach could use a transaction-based database model with replicated servers. Because the implementation details are local to an NVA, there is no need to pick exactly one solution technology, so long as the external interfaces to the NVEs (and remote NVAs) are sufficiently well defined to achieve interoperability.

7.3. NVA External Interface

Conceptually, from the perspective of an NVE, an NVA is a single entity. An NVE interacts with the NVA, and it is the NVA's responsibility for ensuring that interactions between the NVE and NVA result in consistent behavior across the NVA and all other NVEs using the same NVA. Because an NVA is built from multiple internal components, an NVA will have to ensure that information flows to all internal NVA components appropriately.

One architectural question is how the NVA presents itself to the NVE. For example, an NVA could be required to provide access via a single IP address. If NVEs only have one IP address to interact with, it would be the responsibility of the NVA to handle NVA component failures, e.g., by using a "floating IP address" that migrates among NVA components to ensure that the NVA can always be reached via the one address. Having all NVA accesses through a single IP address, however, adds constraints to implementing robust failover, load balancing, etc.

In the NVO3 architecture, an NVA is accessed through one or more IP addresses (or IP address/port combination). If multiple IP addresses are used, each IP address provides equivalent functionality, meaning that an NVE can use any of the provided addresses to interact with the NVA. Should one address stop working, an NVE is expected to failover to another. While the different addresses result in equivalent functionality, one address may respond more quickly than another, e.g., due to network conditions, load on the server, etc.

To provide some control over load balancing, NVA addresses may have an associated priority. Addresses are used in order of priority, with no explicit preference among NVA addresses having the same priority. To provide basic load-balancing among NVAs of equal priorities, NVEs could use some randomization input to select among equal-priority NVAs. Such a priority scheme facilitates failover and load balancing, for example, allowing a network operator to specify a set of primary and backup NVAs.

It may be desirable to have individual NVA addresses responsible for a subset of information about an NV Domain. In such a case, NVEs would use different NVA addresses for obtaining or updating information about particular VNs or TS bindings. A key question with such an approach is how information would be partitioned, and how an NVE could determine which address to use to get the information it needs.

Another possibility is to treat the information on which NVA addresses to use as cached (soft-state) information at the NVEs, so that any NVA address can be used to obtain any information, but NVEs are informed of preferences for which addresses to use for particular information on VNs or TS bindings. That preference information would be cached for future use to improve behavior - e.g., if all requests for a specific subset of VNs are forwarded to a specific NVA component, the NVE can optimize future requests within that subset by sending them directly to that NVA component via its address.

8. NVE-to-NVA Protocol

As outlined in Section 4.3, an NVE needs certain information in order to perform its functions. To obtain such information from an NVA, an NVE-to-NVA protocol is needed. The NVE-to-NVA protocol provides two functions. First it allows an NVE to obtain information about the location and status of other TSs with which it needs to communicate. Second, the NVE-to-NVA protocol provides a way for NVEs to provide updates to the NVA about the TSs attached to that NVE (e.g., when a TS attaches or detaches from the NVE), or about communication errors encountered when sending traffic to remote NVEs. For example, an NVE could indicate that a destination it is trying to reach at a destination NVE is unreachable for some reason.

While having a direct NVE-to-NVA protocol might seem straightforward, the existence of existing VM orchestration systems complicates the choices an NVE has for interacting with the NVA.

8.1. NVE-NVA Interaction Models

An NVE interacts with an NVA in at least two (quite different) ways:

- o NVEs embedded within the same server as the hypervisor can obtain necessary information entirely through the hypervisor-facing side of the NVE. Such an approach is a natural extension to existing VM orchestration systems supporting server virtualization because an existing protocol between the hypervisor and VM orchestration system already exists and can be leveraged to obtain any needed information. Specifically, VM orchestration systems used to create, terminate and migrate VMs already use well-defined (though typically proprietary) protocols to handle the interactions between the hypervisor and VM orchestration system. For such systems, it is a natural extension to leverage the existing orchestration protocol as a sort of proxy protocol for handling the interactions between an NVE and the NVA. Indeed, existing implementations can already do this.
- o Alternatively, an NVE can obtain needed information by interacting directly with an NVA via a protocol operating over the data center underlay network. Such an approach is needed to support NVEs that are not associated with systems performing server virtualization (e.g., as in the case of a standalone gateway) or where the NVE needs to communicate directly with the NVA for other reasons.

The NVO3 architecture will focus on support for the second model above. Existing virtualization environments are already using the first model. But they are not sufficient to cover the case of

standalone gateways -- such gateways may not support virtualization and do not interface with existing VM orchestration systems.

8.2. Direct NVE-NVA Protocol

An NVE can interact directly with an NVA via an NVE-to-NVA protocol. Such a protocol can be either independent of the NVA internal protocol, or an extension of it. Using a purpose-specific protocol would provide architectural separation and independence between the NVE and NVA. The NVE and NVA interact in a well-defined way, and changes in the NVA (or NVE) do not need to impact each other. Using a dedicated protocol also ensures that both NVE and NVA implementations can evolve independently and without dependencies on each other. Such independence is important because the upgrade path for NVEs and NVAs is quite different. Upgrading all the NVEs at a site will likely be more difficult in practice than upgrading NVAs because of their large number - one on each end device. In practice, it would be prudent to assume that once an NVE has been implemented and deployed, it may be challenging to get subsequent NVE extensions and changes implemented and deployed, whereas an NVA (and its associated internal protocols) are more likely to evolve over time as experience is gained from usage and upgrades will involve fewer nodes.

Requirements for a direct NVE-NVA protocol can be found in [I-D.ietf-nvo3-nve-nva-cp-req]

8.3. Propagating Information Between NVEs and NVAs

Information flows between NVEs and NVAs in both directions. The NVA maintains information about all VNs in the NV Domain, so that NVEs do not need to do so themselves. NVEs obtain from the NVA information about where a given remote TS destination resides. NVAs in turn obtain information from NVEs about the individual TSs attached to those NVEs.

While the NVA could push information relevant to every virtual network to every NVE, such an approach scales poorly and is unnecessary. In practice, a given NVE will only need and want to know about VNs to which it is attached. Thus, an NVE should be able to subscribe to updates only for the virtual networks it is interested in receiving updates for. The NVO3 architecture supports a model where an NVE is not required to have full mapping tables for all virtual networks in an NV Domain.

Before sending unicast traffic to a remote TS (or TSes for broadcast or multicast traffic), an NVE must know where the remote TS(es) currently reside. When a TS attaches to a virtual network, the NVE

obtains information about that VN from the NVA. The NVA can provide that information to the NVE at the time the TS attaches to the VN, either because the NVE requests the information when the attach operation occurs, or because the VM orchestration system has initiated the attach operation and provides associated mapping information to the NVE at the same time.

There are scenarios where an NVE may wish to query the NVA about individual mappings within an VN. For example, when sending traffic to a remote TS on a remote NVE, that TS may become unavailable (e.g., because it has migrated elsewhere or has been shutdown, in which case the remote NVE may return an error indication). In such situations, the NVE may need to query the NVA to obtain updated mapping information for a specific TS, or verify that the information is still correct despite the error condition. Note that such a query could also be used by the NVA as an indication that there may be an inconsistency in the network and that it should take steps to verify that the information it has about the current state and location of a specific TS is still correct.

For very large virtual networks, the amount of state an NVE needs to maintain for a given virtual network could be significant. Moreover, an NVE may only be communicating with a small subset of the TSs on such a virtual network. In such cases, the NVE may find it desirable to maintain state only for those destinations it is actively communicating with. In such scenarios, an NVE may not want to maintain full mapping information about all destinations on a VN. Should it then need to communicate with a destination for which it does not have mapping information, however, it will need to be able to query the NVA on demand for the missing information on a per-destination basis.

The NVO3 architecture will need to support a range of operations between the NVE and NVA. Requirements for those operations can be found in [I-D.ietf-nvo3-nve-nva-cp-req].

9. Federated NVAs

An NVA provides service to the set of NVEs in its NV Domain. Each NVA manages network virtualization information for the virtual networks within its NV Domain. An NV domain is administered by a single entity.

In some cases, it will be necessary to expand the scope of a specific VN or even an entire NV domain beyond a single NVA. For example, multiple data centers managed by the same administrator may wish to operate all of its data centers as a single NV region. Such cases are handled by having different NVAs peer with each other to exchange

mapping information about specific VNs. NVAs operate in a federated manner with a set of NVAs operating as a loosely-coupled federation of individual NVAs. If a virtual network spans multiple NVAs (e.g., located at different data centers), and an NVE needs to deliver tenant traffic to an NVE that is part of a different NV Domain, it still interacts only with its NVA, even when obtaining mappings for NVEs associated with a different NV Domain.

Figure 3 shows a scenario where two separate NV Domains (1 and 2) share information about Virtual Network "1217". VM1 and VM2 both connect to the same Virtual Network 1217, even though the two VMs are in separate NV Domains. There are two cases to consider. In the first case, NV Domain B (NVB) does not allow NVE-A to tunnel traffic directly to NVE-B. There could be a number of reasons for this. For example, NV Domains 1 and 2 may not share a common address space (i.e., require traversal through a NAT device), or for policy reasons, a domain might require that all traffic between separate NV Domains be funneled through a particular device (e.g., a firewall). In such cases, NVA-2 will advertise to NVA-1 that VM1 on Virtual Network 1217 is available, and direct that traffic between the two nodes go through IP-G. IP-G would then decapsulate received traffic from one NV Domain, translate it appropriately for the other domain and re-encapsulate the packet for delivery.

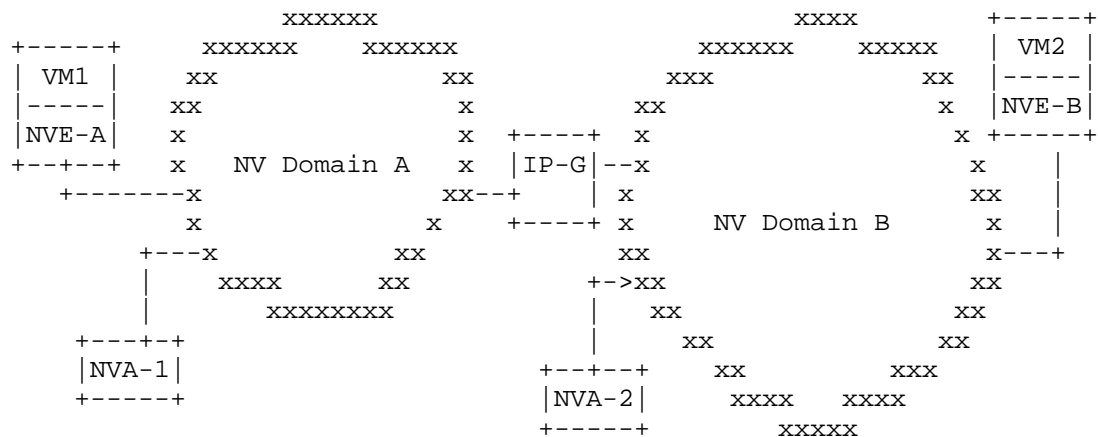


Figure 3: VM1 and VM2 are in different NV Domains.

NVAs at one site share information and interact with NVAs at other sites, but only in a controlled manner. It is expected that policy and access control will be applied at the boundaries between different sites (and NVAs) so as to minimize dependencies on external NVAs that could negatively impact the operation within a site. It is an architectural principle that operations involving NVAs at one site

not be immediately impacted by failures or errors at another site. (Of course, communication between NVEs in different NV domains may be impacted by such failures or errors.) It is a strong requirement that an NVA continue to operate properly for local NVEs even if external communication is interrupted (e.g., should communication between a local and remote NVA fail).

At a high level, a federation of interconnected NVAs has some analogies to BGP and Autonomous Systems. Like an Autonomous System, NVAs at one site are managed by a single administrative entity and do not interact with external NVAs except as allowed by policy. Likewise, the interface between NVAs at different sites is well defined, so that the internal details of operations at one site are largely hidden to other sites. Finally, an NVA only peers with other NVAs that it has a trusted relationship with, i.e., where a VN is intended to span multiple NVAs.

Reasons for using a federated model include:

- o Provide isolation among NVAs operating at different sites at different geographic locations.
- o Control the quantity and rate of information updates that flow (and must be processed) between different NVAs in different data centers.
- o Control the set of external NVAs (and external sites) a site peers with. A site will only peer with other sites that are cooperating in providing an overlay service.
- o Allow policy to be applied between sites. A site will want to carefully control what information it exports (and to whom) as well as what information it is willing to import (and from whom).
- o Allow different protocols and architectures to be used for intra- vs. inter-NVA communication. For example, within a single data center, a replicated transaction server using database techniques might be an attractive implementation option for an NVA, and protocols optimized for intra-NVA communication would likely be different from protocols involving inter-NVA communication between different sites.
- o Allow for optimized protocols, rather than using a one-size-fits all approach. Within a data center, networks tend to have lower-latency, higher-speed and higher redundancy when compared with WAN links interconnecting data centers. The design constraints and tradeoffs for a protocol operating within a data center network are different from those operating over WAN links. While a single

protocol could be used for both cases, there could be advantages to using different and more specialized protocols for the intra- and inter-NVA case.

9.1. Inter-NVA Peering

To support peering between different NVAs, an inter-NVA protocol is needed. The inter-NVA protocol defines what information is exchanged between NVAs. It is assumed that the protocol will be used to share addressing information between data centers and must scale well over WAN links.

10. Control Protocol Work Areas

The NVO3 architecture consists of two major distinct entities: NVEs and NVAs. In order to provide isolation and independence between these two entities, the NVO3 architecture calls for well defined protocols for interfacing between them. For an individual NVA, the architecture calls for a logically centralized entity that could be implemented in a distributed or replicated fashion. While the IETF may choose to define one or more specific architectural approaches to building individual NVAs, there is little need for it to pick exactly one approach to the exclusion of others. An NVA for a single domain will likely be deployed as a single vendor product and thus there is little benefit in standardizing the internal structure of an NVA.

Individual NVAs peer with each other in a federated manner. The NVO3 architecture calls for a well-defined interface between NVAs.

Finally, a hypervisor-to-NVE protocol is needed to cover the split-NVE scenario described in Section 4.2.

11. NVO3 Data Plane Encapsulation

When tunneling tenant traffic, NVEs add encapsulation header to the original tenant packet. The exact encapsulation to use for NVO3 does not seem to be critical. The main requirement is that the encapsulation support a Context ID of sufficient size. A number of encapsulations already exist that provide a VN Context of sufficient size for NVO3. For example, VXLAN [RFC7348] has a 24-bit VXLAN Network Identifier (VNI). NVGRE [RFC7637] has a 24-bit Tenant Network ID (TNI). MPLS-over-GRE provides a 20-bit label field. While there is widespread recognition that a 12-bit VN Context would be too small (only 4096 distinct values), it is generally agreed that 20 bits (1 million distinct values) and 24 bits (16.8 million distinct values) are sufficient for a wide variety of deployment scenarios.

12. Operations, Administration and Maintenance (OAM)

The simplicity of operating and debugging overlay networks will be critical for successful deployment.

Overlay networks are based on tunnels between NVEs, so the OAM (Operations, Administration and Maintenance) [RFC6291] framework for overlay networks can draw from prior IETF OAM work for tunnel-based networks, specifically L2VPN OAM [RFC6136]. RFC 6136 focuses on Fault Management and Performance Management as fundamental to L2VPN service delivery, leaving the Configuration, Management, Accounting Management and Security Management components of the OSI "FCAPS" taxonomy [M.3400] for further study. This section does likewise for NVO3 OAM, but those three areas continue to be important parts of complete OAM functionality for NVO3.

The relationship between the overlay and underlay networks is a consideration for fault and performance management - a fault in the underlay may manifest as fault and/or performance issues in the overlay. Diagnosing and fixing such issues are complicated by NVO3 abstracting the underlay network away from the overlay network (e.g., intermediate nodes on the underlay network path between NVEs are hidden from overlay VNs).

NVO3-specific OAM techniques, protocol constructs and tools are needed to provide visibility beyond this abstraction to diagnose and correct problems that appear in the overlay. Two examples are underlay-aware traceroute [I-D.nordmark-nvo3-transcending-traceroute], and ping protocol constructs for overlay networks [I-D.jain-nvo3-vxlan-ping] [I-D.kumar-nvo3-overlay-ping].

NVO3-specific tools and techniques are best viewed as complements to (i.e., not as replacements for) single-network tools that apply to the overlay and/or underlay networks. Coordination among the individual network tools (for the overlay and underlay networks) and NVO3-aware dual-network tools is required to achieve effective monitoring and fault diagnosis. For example, the defect detection intervals and performance measurement intervals ought to be coordinated among all tools involved in order to provide consistency and comparability of results.

For further discussion of NVO3 OAM requirements, see [I-D.ashwood-nvo3-oam-requirements].

13. Summary

This document presents the overall architecture for Network Virtualization Overlays (NVO3). The architecture calls for three main areas of protocol work:

1. A hypervisor-to-NVE protocol to support Split NVEs as discussed in Section 4.2.
2. An NVE to NVA protocol for disseminating VN information (e.g., inner to outer address mappings).
3. An NVA-to-NVA protocol for exchange of information about specific virtual networks between federated NVAs.

It should be noted that existing protocols or extensions of existing protocols are applicable.

14. Acknowledgments

Helpful comments and improvements to this document have come from Alia Atlas, Abdussalam Baryun, Spencer Dawkins, Linda Dunbar, Stephen Farrell, Anton Ivanov, Lizhong Jin, Suresh Krishnan, Mirja Kuehlwind, Greg Mirsky, Carlos Pignataro, Dennis (Xiaohong) Qin, Erik Smith, Takeshi Takahashi, Ziye Yang and Lucy Yong.

15. IANA Considerations

This memo includes no request to IANA.

16. Security Considerations

The data plane and control plane described in this architecture will need to address potential security threats.

For the data plane, tunneled application traffic may need protection against being misdelivered, modified, or having its content exposed to an inappropriate third party. In all cases, encryption between authenticated tunnel endpoints (e.g., via use of IPsec [RFC4301]) and enforcing policies that control which endpoints and VNs are permitted to exchange traffic can be used to mitigate risks.

For the control plane, between NVAs, the NVA and NVE as well as between different components of the split-NVE approach, a combination of authentication and encryption can be used. All entities will need to properly authenticate with each other and enable encryption for their interactions as appropriate to protect sensitive information.

Leakage of sensitive information about users or other entities associated with VMs whose traffic is virtualized can also be covered by using encryption for the control plane protocols and enforcing policies that control which NVO3 components are permitted to exchange control plane traffic.

Control plane elements such as NVEs and NVAs need to collect performance and other data in order to carry out their functions. This data can sometimes be unexpectedly sensitive, for example, allowing non-obvious inferences as to activity within a VM. This provides a reason to minimise the data collected in some environments in order to limit potential exposure of sensitive information. As noted briefly in RFC 6973 [RFC6973] and RFC 7258 [RFC7258] there is an inevitable tension between being privacy sensitive and network operations that needs to be taken into account in nvo3 protocol development

See the NVO3 framework security considerations in RFC 7365 [RFC7365] for further discussion.

17. Informative References

[I-D.ashwood-nvo3-oam-requirements]

Chen, H., Ashwood-Smith, P., Xia, L., Iyengar, R., Tsou, T., Sajassi, A., Boucadair, M., Jacquenet, C., Daikoku, M., Ghanwani, A., and R. Krishnan, "NVO3 Operations, Administration, and Maintenance Requirements", draft-ashwood-nvo3-oam-requirements-04 (work in progress), October 2015.

[I-D.ietf-nvo3-mcast-framework]

Ghanwani, A., Dunbar, L., McBride, M., Bannai, V., and R. Krishnan, "A Framework for Multicast in Network Virtualization Overlays", draft-ietf-nvo3-mcast-framework-05 (work in progress), May 2016.

[I-D.ietf-nvo3-nve-nva-cp-req]

Kreeger, L., Dutt, D., Narten, T., and D. Black, "Network Virtualization NVE to NVA Control Protocol Requirements", draft-ietf-nvo3-nve-nva-cp-req-05 (work in progress), March 2016.

[I-D.ietf-nvo3-use-case]

Yong, L., Dunbar, L., Toy, M., Isaac, A., and V. Manral, "Use Cases for Data Center Network Virtualization Overlays", draft-ietf-nvo3-use-case-09 (work in progress), September 2016.

- [I-D.jain-nvo3-vxlan-ping]
Jain, P., Singh, K., Balus, F., Henderickx, W., and V. Bannai, "Detecting VXLAN Segment Failure", draft-jain-nvo3-vxlan-ping-00 (work in progress), June 2013.
- [I-D.kumar-nvo3-overlay-ping]
Kumar, N., Pignataro, C., Rao, D., and S. Aldrin, "Detecting NVO3 Overlay Data Plane failures", draft-kumar-nvo3-overlay-ping-01 (work in progress), January 2014.
- [I-D.nordmark-nvo3-transcending-traceroute]
Nordmark, E., Appanna, C., Lo, A., Boutros, S., and A. Dubey, "Layer-Transcending Traceroute for Overlay Networks like VXLAN", draft-nordmark-nvo3-transcending-traceroute-03 (work in progress), July 2016.
- [IEEE-802.1Q]
IEEE Std 802.1Q-2014, , "IEEE Standard for Local and metropolitan area networks: Bridges and Bridged Networks", November 2014.
- [M.3400]
ITU-T Recommendation M.3400, , "TMN management functions", February 2000.
- [RFC0826]
Plummer, D., "Ethernet Address Resolution Protocol: Or Converting Network Protocol Addresses to 48.bit Ethernet Address for Transmission on Ethernet Hardware", STD 37, RFC 826, DOI 10.17487/RFC0826, November 1982, <<http://www.rfc-editor.org/info/rfc826>>.
- [RFC4301]
Kent, S. and K. Seo, "Security Architecture for the Internet Protocol", RFC 4301, DOI 10.17487/RFC4301, December 2005, <<http://www.rfc-editor.org/info/rfc4301>>.
- [RFC4364]
Rosen, E. and Y. Rekhter, "BGP/MPLS IP Virtual Private Networks (VPNs)", RFC 4364, DOI 10.17487/RFC4364, February 2006, <<http://www.rfc-editor.org/info/rfc4364>>.
- [RFC4861]
Narten, T., Nordmark, E., Simpson, W., and H. Soliman, "Neighbor Discovery for IP version 6 (IPv6)", RFC 4861, DOI 10.17487/RFC4861, September 2007, <<http://www.rfc-editor.org/info/rfc4861>>.
- [RFC6136]
Sajassi, A., Ed. and D. Mohan, Ed., "Layer 2 Virtual Private Network (L2VPN) Operations, Administration, and Maintenance (OAM) Requirements and Framework", RFC 6136, DOI 10.17487/RFC6136, March 2011, <<http://www.rfc-editor.org/info/rfc6136>>.

- [RFC6291] Andersson, L., van Helvoort, H., Bonica, R., Romascanu, D., and S. Mansfield, "Guidelines for the Use of the "OAM" Acronym in the IETF", BCP 161, RFC 6291, DOI 10.17487/RFC6291, June 2011, <<http://www.rfc-editor.org/info/rfc6291>>.
- [RFC6973] Cooper, A., Tschofenig, H., Aboba, B., Peterson, J., Morris, J., Hansen, M., and R. Smith, "Privacy Considerations for Internet Protocols", RFC 6973, DOI 10.17487/RFC6973, July 2013, <<http://www.rfc-editor.org/info/rfc6973>>.
- [RFC7258] Farrell, S. and H. Tschofenig, "Pervasive Monitoring Is an Attack", BCP 188, RFC 7258, DOI 10.17487/RFC7258, May 2014, <<http://www.rfc-editor.org/info/rfc7258>>.
- [RFC7348] Mahalingam, M., Dutt, D., Duda, K., Agarwal, P., Kreeger, L., Sridhar, T., Bursell, M., and C. Wright, "Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks", RFC 7348, DOI 10.17487/RFC7348, August 2014, <<http://www.rfc-editor.org/info/rfc7348>>.
- [RFC7364] Narten, T., Ed., Gray, E., Ed., Black, D., Fang, L., Kreeger, L., and M. Napierala, "Problem Statement: Overlays for Network Virtualization", RFC 7364, DOI 10.17487/RFC7364, October 2014, <<http://www.rfc-editor.org/info/rfc7364>>.
- [RFC7365] Lasserre, M., Balus, F., Morin, T., Bitar, N., and Y. Rekhter, "Framework for Data Center (DC) Network Virtualization", RFC 7365, DOI 10.17487/RFC7365, October 2014, <<http://www.rfc-editor.org/info/rfc7365>>.
- [RFC7637] Garg, P., Ed. and Y. Wang, Ed., "NVGRE: Network Virtualization Using Generic Routing Encapsulation", RFC 7637, DOI 10.17487/RFC7637, September 2015, <<http://www.rfc-editor.org/info/rfc7637>>.

Authors' Addresses

David Black
Dell EMC

Email: david.black@dell.com

Jon Hudson
Independent

Email: jon.hudson@gmail.com

Lawrence Kreeger
Cisco

Email: kreeger@cisco.com

Marc Lasserre
Independent

Email: mmlasserre@gmail.com

Thomas Narten
IBM

Email: narten@us.ibm.com

Internet Engineering Task Force
Internet Draft
Intended status: Informational
Expires: Oct 2014

Nabil Bitar
Verizon

Marc Lasserre
Florin Balus
Alcatel-Lucent

Thomas Morin
France Telecom Orange

Lizhong Jin

Bhumip Khasnabish
ZTE

April 15, 2014

NVO3 Data Plane Requirements
draft-ietf-nvo3-dataplane-requirements-03.txt

Status of this Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <http://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on Oct 15, 2014.

Copyright Notice

Copyright (c) 2013 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents

(<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Abstract

Several IETF drafts relate to the use of overlay networks to support large scale virtual data centers. This draft provides a list of data plane requirements for Network Virtualization over L3 (NVO3) that have to be addressed in solutions documents.

Table of Contents

1. Introduction.....	3
1.1. Conventions used in this document.....	3
1.2. General terminology.....	3
2. Data Path Overview.....	3
3. Data Plane Requirements.....	5
3.1. Virtual Access Points (VAPs).....	5
3.2. Virtual Network Instance (VNI).....	5
3.2.1. L2 VNI.....	5
3.2.2. L3 VNI.....	6
3.3. Overlay Module.....	7
3.3.1. NVO3 overlay header.....	8
3.3.1.1. Virtual Network Context Identification.....	8
3.3.1.2. Quality of Service (QoS) identifier.....	8
3.3.2. Tunneling function.....	9
3.3.2.1. LAG and ECMP.....	9
3.3.2.2. DiffServ and ECN marking.....	10
3.3.2.3. Handling of BUM traffic.....	11
3.4. External NVO3 connectivity.....	11
3.4.1. Gateway (GW) Types.....	12
3.4.1.1. VPN and Internet GWs.....	12
3.4.1.2. Inter-DC GW.....	12
3.4.1.3. Intra-DC gateways.....	12
3.4.2. Path optimality between NVEs and Gateways.....	12
3.4.2.1. Load-balancing.....	13

3.4.2.2. Triangular Routing Issues.....	14
3.5. Path MTU.....	14
3.6. Hierarchical NVE dataplane requirements.....	15
3.7. Other considerations.....	15
3.7.1. Data Plane Optimizations.....	15
3.7.2. NVE location trade-offs.....	15
4. Security Considerations.....	16
5. IANA Considerations.....	16
6. References.....	16
6.1. Normative References.....	16
6.2. Informative References.....	16
7. Acknowledgments.....	17

1. Introduction

1.1. Conventions used in this document

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC-2119 [RFC2119].

In this document, these words will appear with that interpretation only when in ALL CAPS. Lower case uses of these words are not to be interpreted as carrying RFC-2119 significance.

1.2. General terminology

The terminology defined in [NVO3-framework] is used throughout this document. Terminology specific to this memo is defined here and is introduced as needed in later sections.

BUM: Broadcast, Unknown Unicast, Multicast traffic

TS: Tenant System

2. Data Path Overview

The NVO3 framework [NVO3-framework] defines the generic NVE model depicted in Figure 1:

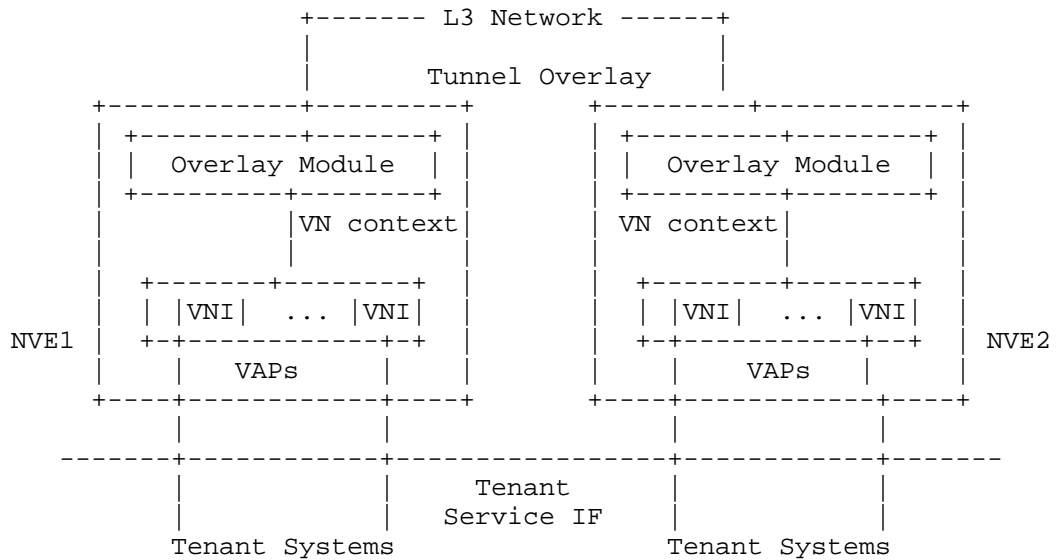


Figure 1 : Generic reference model for NV Edge

When a frame is received by an ingress NVE from a Tenant System over a local VAP, it needs to be parsed in order to identify which virtual network instance it belongs to. The parsing function can examine various fields in the data frame (e.g., VLANID) and/or associated interface/port the frame came from.

Once a corresponding VNI is identified, a lookup is performed to determine where the frame needs to be sent. This lookup can be based on any combinations of various fields in the data frame (e.g., destination MAC addresses and/or destination IP addresses). Note that additional criteria such as Ethernet 802.1p priorities and/or DSCP markings might be used to select an appropriate tunnel or local VAP destination.

Lookup tables can be populated using different techniques: data plane learning, management plane configuration, or a distributed control plane. Management and control planes are not in the scope of this document. The data plane based solution is described in this document as it has implications on the data plane processing function.

The result of this lookup yields the corresponding information needed to build the overlay header, as described in section 3.3. This information includes the destination L3 address of the egress NVE. Note that this lookup might yield a list of tunnels such as when ingress replication is used for BUM traffic.

The overlay header **MUST** include a context identifier which the egress NVE will use to identify which VNI this frame belongs to.

The egress NVE checks the context identifier and removes the encapsulation header and then forwards the original frame towards the appropriate recipient, usually a local VAP.

3. Data Plane Requirements

3.1. Virtual Access Points (VAPs)

The NVE forwarding plane **MUST** support VAP identification through the following mechanisms:

- Using the local interface on which the frames are received, where the local interface may be an internal, virtual port in a virtual switch or a physical port on a ToR switch
- Using the local interface and some fields in the frame header, e.g. one or multiple VLANs or the source MAC

3.2. Virtual Network Instance (VNI)

VAPs are associated with a specific VNI at service instantiation time.

A VNI identifies a per-tenant private context, i.e. per-tenant policies and a FIB table to allow overlapping address space between tenants.

There are different VNI types differentiated by the virtual network service they provide to Tenant Systems. Network virtualization can be provided by L2 and/or L3 VNIs.

3.2.1. L2 VNI

An L2 VNI **MUST** provide an emulated Ethernet multipoint service as if Tenant Systems are interconnected by a bridge (but instead by using a set of NVO3 tunnels). The emulated bridge could be 802.1Q enabled (allowing use of VLAN tags as a VAP). An L2 VNI provides per tenant virtual switching instance with MAC addressing isolation and L3 tunneling. Loop avoidance capability **MUST** be provided.

Forwarding table entries provide mapping information between tenant system MAC addresses and VAPs on directly connected VNIs and L3 tunnel destination addresses over the overlay. Such entries could be populated by a control or management plane, or via data plane.

Unless a control plane is used to disseminate address mappings, data plane learning **MUST** be used to populate forwarding tables. As frames arrive from VAPs or from overlay tunnels, standard MAC learning procedures are used: The tenant system source MAC address is learned against the VAP or the NVO3 tunneling encapsulation source address on which the frame arrived. Data plane learning implies that unknown unicast traffic will be flooded (i.e. broadcast).

When flooding is required, either to deliver unknown unicast, or broadcast or multicast traffic, the NVE **MUST** either support ingress replication or multicast.

When using underlay multicast, the NVE **MUST** have one or more underlay multicast trees that can be used by local VNIs for flooding to NVEs belonging to the same VN. For each VNI, there is at least one underlay flooding tree used for Broadcast, Unknown Unicast and Multicast forwarding. This tree **MAY** be shared across VNIs. The flooding tree is equivalent with a multicast (*,G) construct where all the NVEs for which the corresponding VNI is instantiated are members.

When tenant multicast is supported, it **SHOULD** also be possible to select whether the NVE provides optimized underlay multicast trees inside the VNI for individual tenant multicast groups or whether the default VNI flooding tree is used. If the former option is selected the VNI **SHOULD** be able to snoop IGMP/MLD messages in order to efficiently join/prune Tenant System from multicast trees.

3.2.2. L3 VNI

L3 VNIs **MUST** provide virtualized IP routing and forwarding. L3 VNIs **MUST** support per-tenant forwarding instance with IP addressing isolation and L3 tunneling for interconnecting instances of the same VNI on NVEs.

In the case of L3 VNI, the inner TTL field **MUST** be decremented by (at least) 1 as if the NVO3 egress NVE was one (or more) hop(s) away. The TTL field in the outer IP header **MUST** be set to a value appropriate for delivery of the encapsulated frame to the tunnel exit point. Thus, the default behavior **MUST** be the TTL pipe model where the overlay network looks like one hop to the sending NVE. Configuration of a "uniform" TTL model where the outer tunnel TTL is

set equal to the inner TTL on ingress NVE and the inner TTL is set to the outer TTL value on egress MAY be supported. [RFC2983] provides additional details on the uniform and pipe models.

L2 and L3 VNIs can be deployed in isolation or in combination to optimize traffic flows per tenant across the overlay network. For example, an L2 VNI may be configured across a number of NVEs to offer L2 multi-point service connectivity while a L3 VNI can be co-located to offer local routing capabilities and gateway functionality. In addition, integrated routing and bridging per tenant MAY be supported on an NVE. An instantiation of such service may be realized by interconnecting an L2 VNI as access to an L3 VNI on the NVE.

When underlay multicast is supported, it MAY be possible to select whether the NVE provides optimized underlay multicast trees inside the VNI for individual tenant multicast groups or whether a default underlay VNI multicasting tree, where all the NVEs of the corresponding VNI are members, is used.

3.3. Overlay Module

The overlay module performs a number of functions related to NVO3 header and tunnel processing.

The following figure shows a generic NVO3 encapsulated frame:

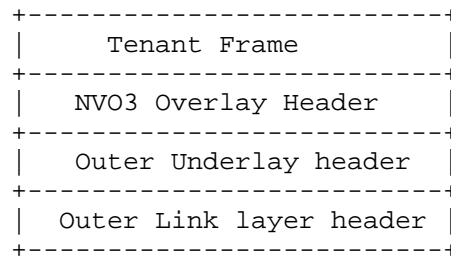


Figure 2 : NVO3 encapsulated frame

where

- . Tenant frame: Ethernet or IP based upon the VNI type

- . NVO3 overlay header: Header containing VNI context information and other optional fields that can be used for processing this packet.
- . Outer underlay header: Can be either IP or MPLS
- . Outer link layer header: Header specific to the physical transmission link used

3.3.1. NVO3 overlay header

An NVO3 overlay header **MUST** be included after the underlay tunnel header when forwarding tenant traffic.

Note that this information can be carried within existing protocol headers (when overloading of specific fields is possible) or within a separate header.

3.3.1.1. Virtual Network Context Identification

The overlay encapsulation header **MUST** contain a field which allows the encapsulated frame to be delivered to the appropriate virtual network endpoint by the egress NVE.

The egress NVE uses this field to determine the appropriate virtual network context in which to process the packet. This field **MAY** be an explicit, unique (to the administrative domain) virtual network identifier (VNID) or **MAY** express the necessary context information in other ways (e.g. a locally significant identifier).

In the case of a global identifier, this field **MUST** be large enough to scale to 100's of thousands of virtual networks. Note that there is typically no such constraint when using a local identifier.

3.3.1.2. Quality of Service (QoS) identifier

Traffic flows originating from different applications could rely on differentiated forwarding treatment to meet end-to-end availability and performance objectives. Such applications may span across one or more overlay networks. To enable such treatment, support for multiple Classes of Service (Cos) across or between overlay networks **MAY** be required.

To effectively enforce CoS across or between overlay networks without Deep Packet Inspection (DPI) repeat, NVEs **MAY** be able to map

CoS markings between networking layers, e.g., Tenant Systems, Overlays, and/or Underlay, enabling each networking layer to independently enforce its own CoS policies. For example:

- TS (e.g. VM) CoS
 - o Tenant CoS policies MAY be defined by Tenant administrators
 - o QoS fields (e.g. IP DSCP and/or Ethernet 802.1p) in the tenant frame are used to indicate application level CoS requirements
- NVE CoS: Support for NVE Service CoS MAY be provided through a QoS field, inside the NVO3 overlay header
 - o NVE MAY classify packets based on Tenant CoS markings or other mechanisms (eg. DPI) to identify the proper service CoS to be applied across the overlay network
 - o NVE service CoS levels are normalized to a common set (for example 8 levels) across multiple tenants; NVE uses per tenant policies to map Tenant CoS to the normalized service CoS fields in the NVO3 header
- Underlay CoS
 - o The underlay/core network MAY use a different CoS set (for example 4 levels) than the NVE CoS as the core devices MAY have different QoS capabilities compared with NVEs.
 - o The Underlay CoS MAY also change as the NVO3 tunnels pass between different domains.

3.3.2. Tunneling function

This section describes the underlay tunneling requirements. From an encapsulation perspective, IPv4 or IPv6 MUST be supported, both IPv4 and IPv6 SHOULD be supported, MPLS MAY be supported.

3.3.2.1. LAG and ECMP

For performance reasons, multipath over LAG and ECMP paths MAY be supported.

LAG (Link Aggregation Group) [IEEE 802.1AX-2008] and ECMP (Equal Cost Multi Path) are commonly used techniques to perform load-balancing of microflows over a set of a parallel links either at

Layer-2 (LAG) or Layer-3 (ECMP). Existing deployed hardware implementations of LAG and ECMP uses a hash of various fields in the encapsulation (outermost) header(s) (e.g. source and destination MAC addresses for non-IP traffic, source and destination IP addresses, L4 protocol, L4 source and destination port numbers, etc). Furthermore, hardware deployed for the underlay network(s) will be most often unaware of the carried, innermost L2 frames or L3 packets transmitted by the TS.

Thus, in order to perform fine-grained load-balancing over LAG and ECMP paths in the underlying network, the encapsulation needs to present sufficient entropy to exercise all paths through several LAG/ECMP hops.

The entropy information can be inferred from the NVO3 overlay header or underlay header. If the overlay protocol does not support the necessary entropy information or the switches/routers in the underlay do not support parsing of the additional entropy information in the overlay header, underlay switches and routers should be programmable, i.e. select the appropriate fields in the underlay header for hash calculation based on the type of overlay header.

All packets that belong to a specific flow MUST follow the same path in order to prevent packet re-ordering. This is typically achieved by ensuring that the fields used for hashing are identical for a given flow.

The goal is for all paths available to the overlay network to be used efficiently. Different flows should be distributed as evenly as possible across multiple underlay network paths. For instance, this can be achieved by ensuring that some fields used for hashing are randomly generated.

3.3.2.2. DiffServ and ECN marking

When traffic is encapsulated in a tunnel header, there are numerous options as to how the Diffserv Code-Point (DSCP) and Explicit Congestion Notification (ECN) markings are set in the outer header and propagated to the inner header on decapsulation.

[RFC2983] defines two modes for mapping the DSCP markings from inner to outer headers and vice versa. The Uniform model copies the inner DSCP marking to the outer header on tunnel ingress, and copies that outer header value back to the inner header at tunnel egress. The Pipe model sets the DSCP value to some value based on local policy

at ingress and does not modify the inner header on egress. Both models SHOULD be supported.

[RFC6040] defines ECN marking and processing for IP tunnels.

3.3.2.3. Handling of BUM traffic

NVO3 data plane support for either ingress replication or point-to-multipoint tunnels is required to send traffic destined to multiple locations on a per-VNI basis (e.g. L2/L3 multicast traffic, L2 broadcast and unknown unicast traffic). It is possible that both methods be used simultaneously.

There is a bandwidth vs state trade-off between the two approaches. User-configurable settings MUST be provided to select which method(s) gets used based upon the amount of replication required (i.e. the number of hosts per group), the amount of multicast state to maintain, the duration of multicast flows and the scalability of multicast protocols.

When ingress replication is used, NVEs MUST maintain for each VNI the related tunnel endpoints to which it needs to replicate the frame.

For point-to-multipoint tunnels, the bandwidth efficiency is increased at the cost of more state in the Core nodes. The ability to auto-discover or pre-provision the mapping between VNI multicast trees to related tunnel endpoints at the NVE and/or throughout the core SHOULD be supported.

3.4. External NVO3 connectivity

It is important that NVO3 services interoperate with current VPN and Internet services. This may happen inside one DC during a migration phase or as NVO3 services are delivered to the outside world via Internet or VPN gateways (GW).

Moreover the compute and storage services delivered by a NVO3 domain may span multiple DCs requiring Inter-DC connectivity. From a DC perspective a set of GW devices are required in all of these cases albeit with different functionalities influenced by the overlay type across the WAN, the service type and the DC network technologies used at each DC site.

A GW handling the connectivity between NVO3 and external domains represents a single point of failure that may affect multiple tenant

services. Redundancy between NVO3 and external domains MUST be supported.

3.4.1. Gateway (GW) Types

3.4.1.1. VPN and Internet GWs

Tenant sites may be already interconnected using one of the existing VPN services and technologies (VPLS or IP VPN). If a new NVO3 encapsulation is used, a VPN GW is required to forward traffic between NVO3 and VPN domains. Internet connected Tenants require translation from NVO3 encapsulation to IP in the NVO3 gateway. The translation function SHOULD minimize provisioning touches.

3.4.1.2. Inter-DC GW

Inter-DC connectivity MAY be required to provide support for features like disaster prevention or compute load re-distribution. This MAY be provided via a set of gateways interconnected through a WAN. This type of connectivity MAY be provided either through extension of the NVO3 tunneling domain or via VPN GWs.

3.4.1.3. Intra-DC gateways

Even within one DC there may be End Devices that do not support NVO3 encapsulation, for example bare metal servers, hardware appliances and storage. A gateway device, e.g. a ToR switch, is required to translate the NVO3 to Ethernet VLAN encapsulation.

3.4.2. Path optimality between NVEs and Gateways

Within an NVO3 overlay, a default assumption is that NVO3 traffic will be equally load-balanced across the underlying network consisting of LAG and/or ECMP paths. This assumption is valid only as long as: a) all traffic is load-balanced equally among each of the component-links and paths; and, b) each of the component-links/paths is of identical capacity. During the course of normal operation of the underlying network, it is possible that one, or more, of the component-links/paths of a LAG may be taken out-of-service in order to be repaired, e.g.: due to hardware failure of cabling, optics, etc. In such cases, the administrator may configure the underlying network such that an entire LAG bundle in the underlying network will be reported as operationally down if there is a failure of any single component-link member of the LAG bundle, (e.g.: N = M configuration of the LAG bundle), and, thus, they know that traffic will be carried sufficiently by alternate, available (potentially ECMP) paths in the underlying network. This is a likely

an adequate assumption for Intra-DC traffic where presumably the costs for additional, protection capacity along alternate paths is not cost-prohibitive. In this case, there are no additional requirements on NVO3 solutions to accommodate this type of underlying network configuration and administration.

There is a similar case with ECMP, used Intra-DC, where failure of a single component-path of an ECMP group would result in traffic shifting onto the surviving members of the ECMP group. Unfortunately, there are no automatic recovery methods in IP routing protocols to detect a simultaneous failure of more than one component-path in a ECMP group, operationally disable the entire ECMP group and allow traffic to shift onto alternative paths. This problem is attributable to the underlying network and, thus, out-of-scope of any NVO3 solutions.

On the other hand, for Inter-DC and DC to External Network cases that use a WAN, the costs of the underlying network and/or service (e.g.: IPVPN service) are more expensive; therefore, there is a requirement on administrators to both: a) ensure high availability (active-backup failover or active-active load-balancing); and, b) maintaining substantial utilization of the WAN transport capacity at nearly all times, particularly in the case of active-active load-balancing. With respect to the dataplane requirements of NVO3 solutions, in the case of active-backup fail-over, all of the ingress NVE's need to dynamically adapt to the failure of an active NVE GW when the backup NVE GW announces itself into the NVO3 overlay immediately following a failure of the previously active NVE GW and update their forwarding tables accordingly, (e.g.: perhaps through dataplane learning and/or translation of a gratuitous ARP, IPv6 Router Advertisement). Note that active-backup fail-over could be used to accomplish a crude form of load-balancing by, for example, manually configuring each tenant to use a different NVE GW, in a round-robin fashion.

3.4.2.1. Load-balancing

When using active-active load-balancing across physically separate NVE GW's (e.g.: two, separate chassis) an NVO3 solution SHOULD support forwarding tables that can simultaneously map a single egress NVE to more than one NVO3 tunnels. The granularity of such mappings, in both active-backup and active-active, MUST be specific to each tenant.

3.4.2.2. Triangular Routing Issues

L2/ELAN over NVO3 service may span multiple racks distributed across different DC regions. Multiple ELANs belonging to one tenant may be interconnected or connected to the outside world through multiple Router/VRF gateways distributed throughout the DC regions. In this scenario, without aid from an NVO3 or other type of solution, traffic from an ingress NVE destined to External gateways will take a non-optimal path that will result in higher latency and costs, (since it is using more expensive resources of a WAN). In the case of traffic from an IP/MPLS network destined toward the entrance to an NVO3 overlay, well-known IP routing techniques MAY be used to optimize traffic into the NVO3 overlay, (at the expense of additional routes in the IP/MPLS network). In summary, these issues are well known as triangular routing (a.k.a. traffic tromboning).

Procedures for gateway selection to avoid triangular routing issues SHOULD be provided.

The details of such procedures are, most likely, part of the NVO3 Management and/or Control Plane requirements and, thus, out of scope of this document. However, a key requirement on the dataplane of any NVO3 solution to avoid triangular routing is stated above, in Section 3.4.2, with respect to active-active load-balancing. More specifically, an NVO3 solution SHOULD support forwarding tables that can simultaneously map a single egress NVE to more than one NVO3 tunnel.

The expectation is that, through the Control and/or Management Planes, this mapping information may be dynamically manipulated to, for example, provide the closest geographic and/or topological exit point (egress NVE) for each ingress NVE.

3.5. Path MTU

The tunnel overlay header can cause the MTU of the path to the egress tunnel endpoint to be exceeded.

IP fragmentation SHOULD be avoided for performance reasons.

The interface MTU as seen by a Tenant System SHOULD be adjusted such that no fragmentation is needed. This can be achieved by configuration or be discovered dynamically.

Either of the following options MUST be supported:

- o Classical ICMP-based MTU Path Discovery [RFC1191] [RFC1981] or Extended MTU Path Discovery techniques such as defined in [RFC4821]
- o Segmentation and reassembly support from the overlay layer operations without relying on the Tenant Systems to know about the end-to-end MTU
- o The underlay network MAY be designed in such a way that the MTU can accommodate the extra tunnel overhead.

3.6. Hierarchical NVE dataplane requirements

It might be desirable to support the concept of hierarchical NVEs, such as spoke NVEs and hub NVEs, in order to address possible NVE performance limitations and service connectivity optimizations.

For instance, spoke NVE functionality may be used when processing capabilities are limited. In this case, a hub NVE MUST provide additional data processing capabilities such as packet replication.

3.7. Other considerations

3.7.1. Data Plane Optimizations

Data plane forwarding and encapsulation choices SHOULD consider the limitation of possible NVE implementations, specifically in software based implementations (e.g. servers running virtual switches)

NVE SHOULD provide efficient processing of traffic. For instance, packet alignment, the use of offsets to minimize header parsing, padding techniques SHOULD be considered when designing NV03 encapsulation types.

The NV03 encapsulation/decapsulation processing in software-based NVEs SHOULD make use of hardware assist provided by NICs in order to speed up packet processing.

3.7.2. NVE location trade-offs

In the case of DC traffic, traffic originated from a VM is native Ethernet traffic. This traffic can be switched by a local VM switch or ToR switch and then by a DC gateway. The NVE function can be embedded within any of these elements.

The NVE function can be supported in various DC network elements such as a VM, VM switch, ToR switch or DC GW.

The following criteria SHOULD be considered when deciding where the NVE processing boundary happens:

- o Processing and memory requirements
 - o Datapath (e.g. lookups, filtering, encapsulation/decapsulation)
 - o Control plane processing (e.g. routing, signaling, OAM)
- o FIB/RIB size
- o Multicast support
 - o Routing protocols
 - o Packet replication capability
- o Fragmentation support
- o QoS transparency
- o Resiliency

4. Security Considerations

This requirements document does not raise in itself any specific security issues.

5. IANA Considerations

IANA does not need to take any action for this draft.

6. References

6.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

6.2. Informative References

[NVOPS] Narten, T. et al, "Problem Statement: Overlays for Network Virtualization", draft-narten-nvo3-overlay-problem-statement (work in progress)

- [NVO3-framework] Lasserre, M. et al, "Framework for DC Network Virtualization", draft-lasserre-nvo3-framework (work in progress)
- [OVCPREQ] Kreeger, L. et al, "Network Virtualization Overlay Control Protocol Requirements", draft-kreeger-nvo3-overlay-cp (work in progress)
- [FLOYD] Sally Floyd, Allyn Romanow, "Dynamics of TCP Traffic over ATM Networks", IEEE JSAC, V. 13 N. 4, May 1995
- [RFC4364] Rosen, E. and Y. Rekhter, "BGP/MPLS IP Virtual Private Networks (VPNs)", RFC 4364, February 2006.
- [RFC1191] Mogul, J. "Path MTU Discovery", RFC1191, November 1990
- [RFC1981] McCann, J. et al, "Path MTU Discovery for IPv6", RFC1981, August 1996
- [RFC4821] Mathis, M. et al, "Packetization Layer Path MTU Discovery", RFC4821, March 2007
- [RFC2983] Black, D. "Diffserv and tunnels", RFC2983, October 2000
- [RFC6040] Briscoe, B. "Tunnelling of Explicit Congestion Notification", RFC6040, November 2010
- [RFC6438] Carpenter, B. et al, "Using the IPv6 Flow Label for Equal Cost Multipath Routing and Link Aggregation in Tunnels", RFC6438, November 2011
- [RFC6391] Bryant, S. et al, "Flow-Aware Transport of Pseudowires over an MPLS Packet Switched Network", RFC6391, November 2011

7. Acknowledgments

In addition to the authors the following people have contributed to this document:

Shane Amante, David Black, Dimitrios Stiliadis, Rotem Salomonovitch, Larry Kreeger, Eric Gray and Erik Nordmark.

This document was prepared using 2-Word-v2.0.template.dot.

Authors' Addresses

Nabil Bitar
Verizon
40 Sylvan Road
Waltham, MA 02145
Email: nabil.bitar@verizon.com

Marc Lasserre
Alcatel-Lucent
Email: marc.lasserre@alcatel-lucent.com

Florin Balus
Alcatel-Lucent
777 E. Middlefield Road
Mountain View, CA, USA 94043
Email: florin.balus@alcatel-lucent.com

Thomas Morin
France Telecom Orange
Email: thomas.morin@orange.com

Lizhong Jin
Email : lizho.jin@gmail.com

Bhumip Khasnabish
ZTE
Email : Bhumip.khasnabish@zteusa.com

Network Working Group
Internet-Draft
Intended status: Experimental
Expires: August 29, 2015

P. Quinn
Cisco Systems, Inc.
R. Manur
Broadcom
L. Kreeger
D. Lewis
F. Maino
M. Smith
Cisco Systems, Inc.
P. Agarwal

L. Yong
Huawei USA
X. Xu
Huawei Technologies
U. Elzur
Intel
P. Garg
Microsoft
D. Melman
Marvell
February 25, 2015

Generic Protocol Extension for VXLAN
draft-quinn-vxlan-gpe-04.txt

Abstract

This draft describes extending Virtual eXtensible Local Area Network (VXLAN), via changes to the VXLAN header, with three new capabilities: support for multi-protocol encapsulation, operations, administration and management (OAM) signaling and explicit versioning.

Status of this Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <http://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference

material or to cite them other than as "work in progress."

This Internet-Draft will expire on August 29, 2015.

Copyright Notice

Copyright (c) 2015 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	4
2. VXLAN Without Protocol Extension	5
3. Generic Protocol Extension for VXLAN (VXLAN GPE)	7
3.1. VXLAN GPE Header	7
3.2. Multi Protocol Support	8
3.3. OAM Support	8
3.4. Version Bits	8
4. Outer Encapsulations	9
4.1. Inner VLAN Tag Handling	13
4.2. Fragmentation Considerations	13
5. Backward Compatibility	14
5.1. VXLAN VTEP to VXLAN GPE VTEP	14
5.2. VXLAN GPE VTEP to VXLAN VTEP	14
5.3. VXLAN GPE UDP Ports	14
5.4. VXLAN GPE and Encapsulated IP Header Fields	14
6. VXLAN GPE Examples	15
7. Security Considerations	17
8. Acknowledgments	18
9. IANA Considerations	19
9.1. UDP Port	19
9.2. VXLAN GPE Next Protocol	19
9.3. VXLAN GPE Flag and Reserved Bits	19
10. References	20
10.1. Normative References	20
10.2. Informative References	20
Authors' Addresses	21

1. Introduction

Virtual eXtensible Local Area Network VXLAN [RFC7348] defines an encapsulation format that encapsulates Ethernet frames in an outer UDP/IP transport. As data centers evolve, the need to carry other protocols encapsulated in an IP packet is required, as well as the need to provide increased visibility and diagnostic capabilities within the overlay. The VXLAN header does not specify the protocol being encapsulated and therefore is currently limited to encapsulating only Ethernet frame payload, nor does it provide the ability to define OAM protocols. In addition, [RFC6335] requires that new transports not use transport layer port numbers to identify tunnel payload, rather it encourages encapsulations to use their own identifiers for this purpose. VXLAN GPE is intended to extend the existing VXLAN protocol to provide protocol typing, OAM, and versioning capabilities.

The Version and OAM bits are introduced in Section 3, and the choice of location for these fields is driven by minimizing the impact on existing deployed hardware.

In order to facilitate deployments of VXLAN GPE with hardware currently deployed to support VXLAN, changes from legacy VXLAN have been kept to a minimum. Section 5 provides a detailed discussion about how VXLAN GPE addresses the requirement for backward compatibility with VXLAN.

2. VXLAN Without Protocol Extension

VXLAN provides a method of creating multi-tenant overlay networks by encapsulating packets in IP/UDP along with a header containing a network identifier which is used to isolate tenant traffic in each overlay network from each other. This allows the overlay networks to run over an existing IP network.

Through this encapsulation, VXLAN creates stateless tunnels between VXLAN Tunnel End Points (VTEPs) which are responsible for adding/removing the IP/UDP/VXLAN headers and providing tenant traffic isolation based on the VXLAN Network Identifier (VNI). Tenant systems are unaware that their networking service is being provided by an overlay.

When encapsulating packets, a VTEP must know the IP address of the proper remote VTEP at the far end of the tunnel that can deliver the inner packet to the Tenant System corresponding to the inner destination address. In the case of tenant multicast or broadcast, the outer IP address may be an IP multicast group address, or the VTEP may replicate the packet and send it to all known VTEPs. If multicast is used in the underlay network to send encapsulated packets to remote VTEPs, Any Source Multicast is used and each VTEP serving a particular VNI must perform a (*, G) join to the same group IP address.

Inner to outer address mapping can be determined in two ways. One is source based learning in the data plane, and the other is distribution via a control plane.

Source based learning requires a receiving VTEP to create an inner to outer address mapping by gleaning the information from the received packets by correlating the inner source address to the outer source IP address. When a mapping does not exist, a VTEP forwards the packets to all remote VTEPs participating in the VNI by using IP multicast in the IP underlay network. Each VTEP must be configured with the IP multicast address to use for each VNI. How this occurs is out of scope.

The control plane used to distribute inner to outer mappings is also out of scope. It could use a centralized authority or be distributed, or use a hybrid.

The VXLAN Network Identifier (VNI) provides scoping for the addresses in the header of the encapsulated PDU. If the encapsulated packet is an Ethernet frame, this means the Ethernet MAC addresses are only unique within a given VNI and may overlap with MAC addresses within a different VNI. If the encapsulated packet is an IP packet, this

means the IP addresses are only unique within that VNI.

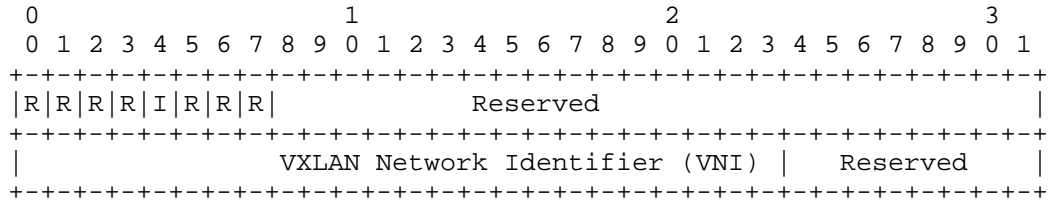


Figure 1: VXLAN Header

3. Generic Protocol Extension for VXLAN (VXLAN GPE)

3.1. VXLAN GPE Header

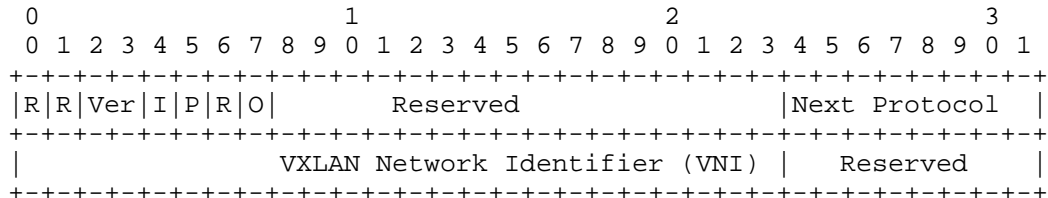


Figure 2: VXLAN GPE Header

Flags (8 bits): The first 8 bits of the header are the flag field. The bits designated "R" above are reserved flags. These MUST be set to zero on transmission and ignored on receipt.

Version (Ver): Indicates VXLAN GPE protocol version. The initial version is 0. If a receiver does not support the version indicated it MUST drop the packet.

Instance Bit (I bit): The I bit MUST be set to indicate a valid VNI.

Next Protocol Bit (P bit): The P bit is set to indicate that the Next Protocol field is present.

OAM Flag Bit (O bit): The O bit is set to indicate that the packet is an OAM packet.

Next Protocol: This 8 bit field indicates the protocol header immediately following the VXLAN GPE header.

VNI: This 24 bit field identifies the VXLAN overlay network the inner packet belongs to. Inner packets belonging to different VNIs cannot communicate with each other (unless explicitly allowed by policy).

Reserved: Reserved fields MUST be set to zero on transmission and ignored on receipt.

3.2. Multi Protocol Support

This draft defines the following two changes to the VXLAN header in order to support multi-protocol encapsulation:

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. When P=1, the destination UDP port MUST be 4790.

P = 0 indicates that the payload MUST conform to VXLAN as defined in [RFC7348], including destination UDP port.

Flag bit 5 was chosen as the P bit because this flag bit is currently reserved in VXLAN.

Next Protocol Field: The lower 8 bits of the first word are used to carry a next protocol. This next protocol field contains the protocol of the encapsulated payload packet. A new protocol registry will be requested from IANA, see section 9.2.

This draft defines the following Next Protocol values:

0x1 : IPv4
0x2 : IPv6
0x3 : Ethernet
0x4 : Network Service Header [NSH]

3.3. OAM Support

Flag bit 7 is defined as the O bit. When the O bit is set to 1, the packet is an OAM packet and OAM processing MUST occur. Other header fields including Next Protocol MUST adhere to the definitions in section 3. The OAM protocol details are out of scope for this document. As with the P-bit, bit 7 is currently a reserved flag in VXLAN.

3.4. Version Bits

VXLAN GPE bits 2 and 3 are defined as version bits. These bits are reserved in VXLAN. The version field is used to ensure backward compatibility going forward with future VXLAN GPE updates.

The initial version for VXLAN GPE is 0.

4. Outer Encapsulations

In addition to the VXLAN GPE header, the packet is further encapsulated in UDP and IP. Data centers based on Ethernet, will then send this IP packet over Ethernet.

Outer UDP Header:

Destination UDP Port: IANA has assigned the value 4790 for the VXLAN GPE UDP port. This well-known destination port is used when sending VXLAN GPE encapsulated packets.

Source UDP Port: The source UDP port is used as entropy for devices forwarding encapsulated packets across the underlay (ECMP for IP routers, or load splitting for link aggregation by bridges). Tenant traffic flows should all use the same source UDP port to lower the chances of packet reordering by the underlay for a given flow. It is recommended for VTEPs to generate this port number using a hash of the inner packet headers.

UDP Checksum: Source VTEPs MAY either calculate a valid checksum, or if this is not possible, set the checksum to zero. When calculating a checksum, it MUST be calculated across the entire packet (outer IP header, UDP header, VXLAN GPE header and payload packet). All receiving VTEPs must accept a checksum value of zero. If the receiving VTEP is capable of validating the checksum, it MAY validate a non-zero checksum and MUST discard the packet if the checksum is determined to be invalid.

Outer IP Header:

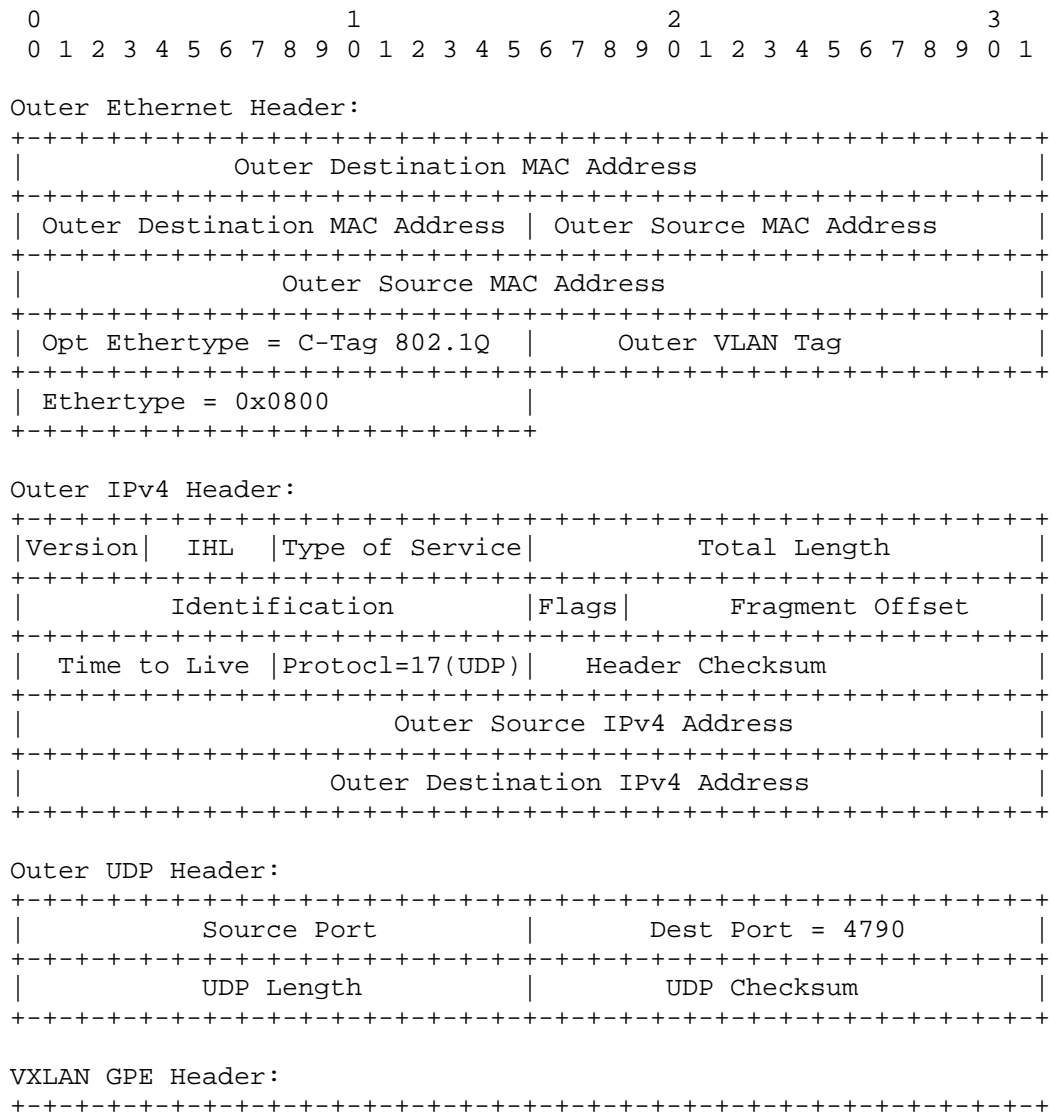
This is the header used by the underlay network to deliver packets between VTEPs. The destination IP address can be a unicast or a multicast IP address. The source IP address must be the source VTEP IP address which can be used to return tenant packets to the tenant system source address within the inner packet header.

When the outer IP header is IPv4, VTEPs MUST set the DF bit.

Outer Ethernet Header:

Most data centers networks are built on Ethernet. Assuming the outer IP packet is being sent across Ethernet, there will be an Ethernet header used to deliver the IP packet to the next hop, which could be the destination VTEP or be a router used to forward the IP packet towards the destination VTEP. If VLANs are in use within the data center, then this Ethernet header would also contain a VLAN tag.

The following figures show the entire stack of protocol headers that would be seen on an Ethernet link carrying encapsulated packets from a VTEP across the underlay network for both IPv4 and IPv6 based underlay networks.



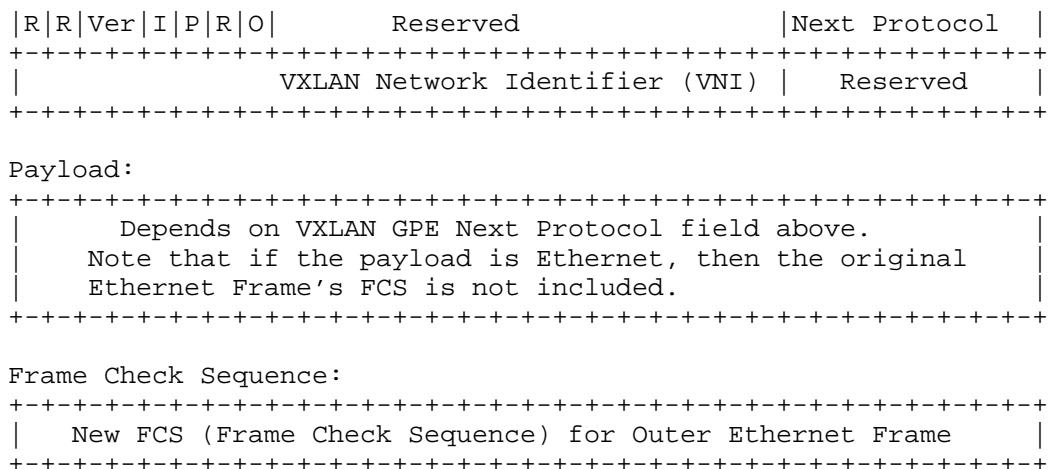
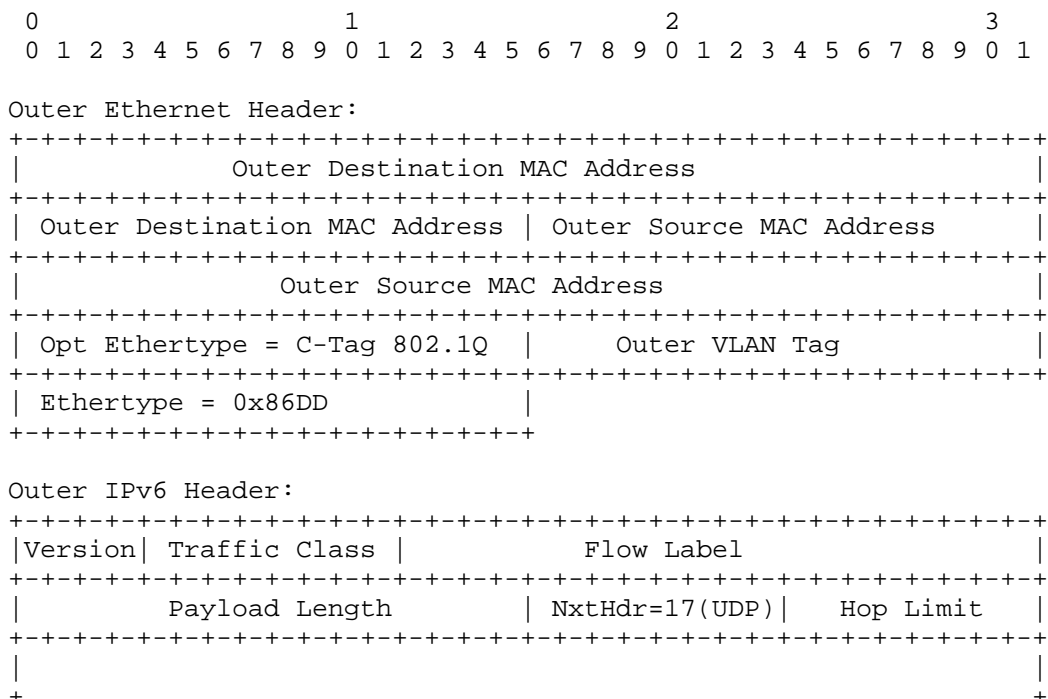


Figure 3: Outer Headers for VXLAN GPE over IPv4



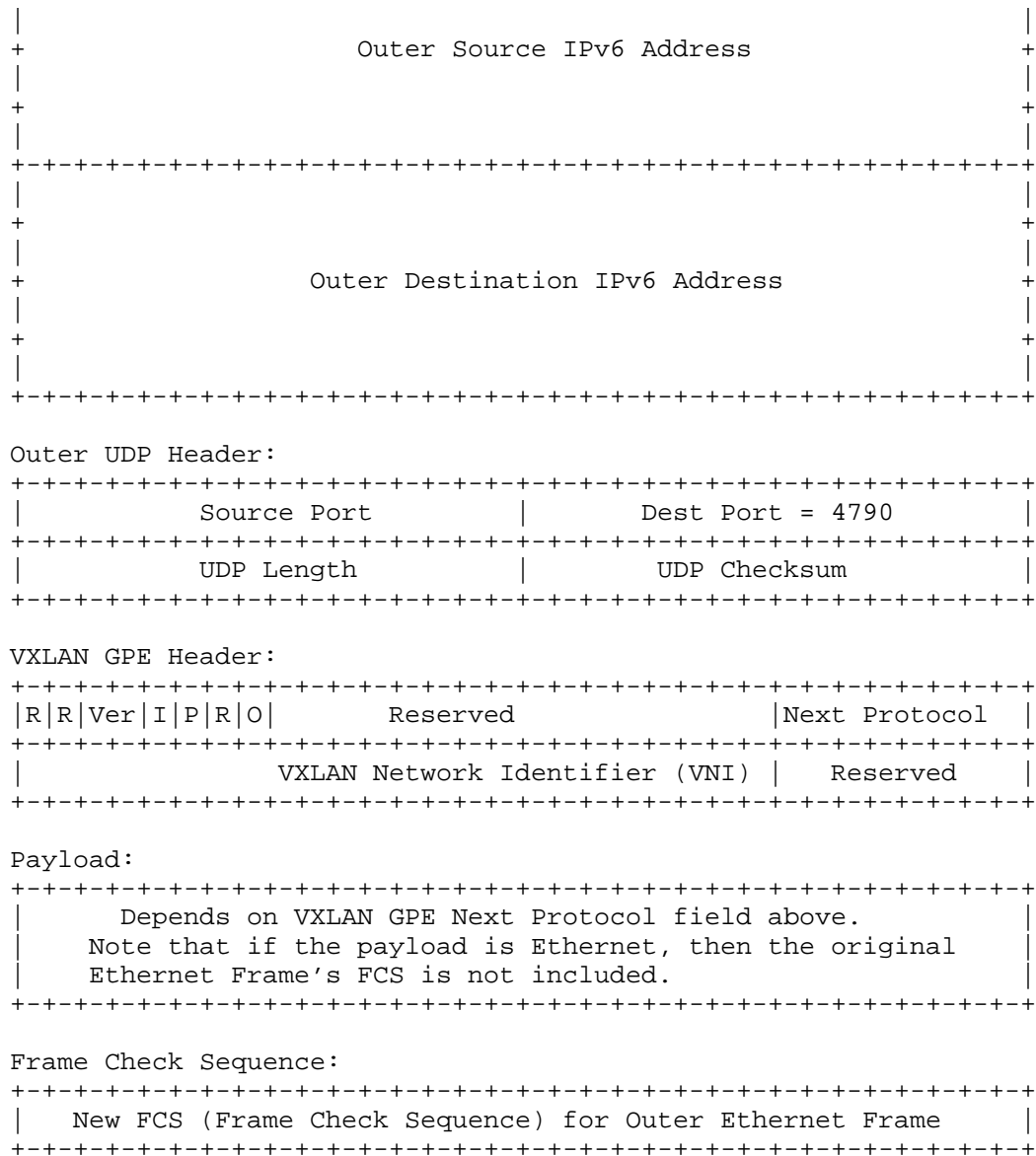


Figure X: Outer Headers for VXLAN GPE over IPv6

Figure 4: Outer Headers for VXLAN GPE over IPv6

4.1. Inner VLAN Tag Handling

If the inner packet (as indicated by the VXLAN GPE Next Protocol field) is an Ethernet frame, it is recommended that it does not contain a VLAN tag. In the most common scenarios, the tenant VLAN tag is translated into a VXLAN Network Identifier. In these scenarios, VTEPs should never send an inner Ethernet frame with a VLAN tag, and a VTEP performing decapsulation should discard any inner frames received with a VLAN tag. However, if the VTEPs are specifically configured to support it for a specific VXLAN Network Identifier, a VTEP may support transparent transport of the inner VLAN tag between all tenant systems on that VNI. The VTEP never looks at the value of the inner VLAN tag, but simply passes it across the underlay.

4.2. Fragmentation Considerations

VTEPs MUST never fragment an encapsulated VXLAN GPE packet, and when the outer IP header is IPv4, VTEPs MUST set the DF bit in the outer IPv4 header. It is recommended that the underlay network be configured to carry an MTU at least large enough to accommodate the added encapsulation headers. It is recommended that VTEPs perform Path MTU discovery [RFC1191] [RFC1981] to determine if the underlay network can carry the encapsulated payload packet.

5. Backward Compatibility

5.1. VXLAN VTEP to VXLAN GPE VTEP

A VXLAN VTEP conforms to VXLAN frame format and uses UDP destination port 4789 when sending traffic to VXLAN GPE VTEP. As per VXLAN, reserved bits 5 and 7, VXLAN GPE P and O-bits respectively must be set to zero. The remaining reserved bits must be zero, including the VXLAN GPE version field, bits 2 and 3. The encapsulated payload MUST be Ethernet.

5.2. VXLAN GPE VTEP to VXLAN VTEP

A VXLAN GPE VTEP MUST NOT encapsulate non-Ethernet frames to a VXLAN VTEP. When encapsulating Ethernet frames to a VXLAN VTEP, the VXLAN GPE VTEP MUST conform to VXLAN frame format and hence will set the P bit to 0, the Next Protocol to 0 and use UDP destination port 4789. A VXLAN GPE VTEP MUST also set O = 0 and Ver = 0 when encapsulating Ethernet frames to VXLAN VTEP. The receiving VXLAN VTEP will treat this packet as a VXLAN packet.

A method for determining the capabilities of a VXLAN VTEP (GPE or non-GPE) is out of the scope of this draft.

5.3. VXLAN GPE UDP Ports

VXLAN GPE uses a IANA assigned UDP destination port, 4790, when sending traffic to VXLAN GPE VTEPs.

5.4. VXLAN GPE and Encapsulated IP Header Fields

When encapsulating and decapsulating IPv4 and IPv6 packets, certain fields, such as IPv4 Time to Live (TTL) from the inner IP header need to be considered. VXLAN GPE IP encapsulation and decapsulation utilizes the techniques described in [RFC6830], section 5.3.

6. VXLAN GPE Examples

This section provides three examples of protocols encapsulated using the Generic Protocol Extension for VXLAN described in this document.

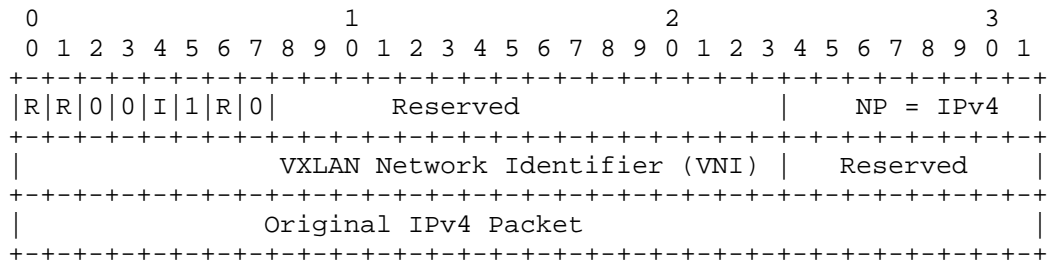


Figure 5: IPv4 and VXLAN GPE

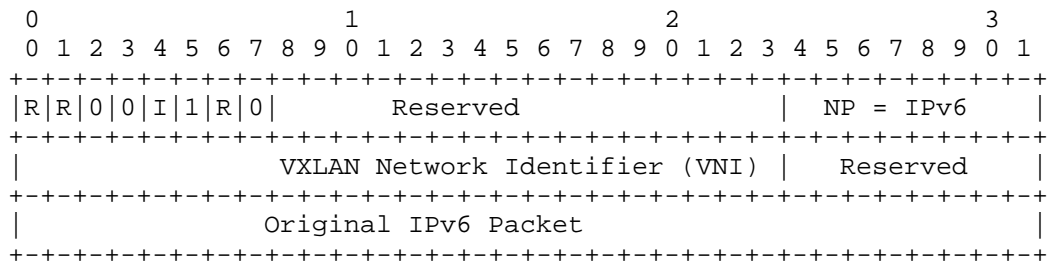


Figure 6: IPv6 and VXLAN GPE

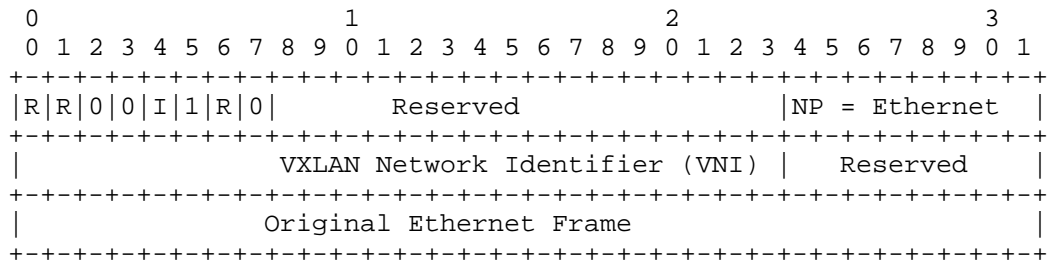


Figure 7: Ethernet and VXLAN GPE

7. Security Considerations

VXLAN's security is focused on issues around L2 encapsulation into L3. With VXLAN GPE, issues such as spoofing, flooding, and traffic redirection are dependent on the particular protocol payload encapsulated.

8. Acknowledgments

A special thank you goes to Dino Farinacci for his guidance and detailed review.

9. IANA Considerations

9.1. UDP Port

UDP 4790 port has been assigned by IANA for VXLAN GPE.

9.2. VXLAN GPE Next Protocol

IANA is requested to set up a registry of "Next Protocol". These are 8-bit values. Next Protocol values 0, 1, 2, 3 and 4 are defined in this draft. New values are assigned via Standards Action [RFC5226].

Next Protocol	Description	Reference
0	Reserved	This document
1	IPv4	This document
2	IPv6	This document
3	Ethernet	This document
4	NSH	This document
5..253	Unassigned	

Table 1

9.3. VXLAN GPE Flag and Reserved Bits

There are ten flag bits at the beginning of the VXLAN GPE header, followed by 16 reserved bits and an 8-bit reserved field at the end of the header. New bits are assigned via Standards Action [RFC5226].

Bits 0-1 - Reserved
Bits 2-3 - Version
Bit 4 - Instance ID (I bit)
Bit 5 - Next Protocol (P bit)
Bit 6 - Reserved
Bit 7 - OAM (O bit)
Bits 8-23 - Reserved
Bits 24-31 in the 2nd Word -- Reserved

Reserved bits/fields MUST be set to 0 by the sender and ignored by the receiver.

10. References

10.1. Normative References

- [RFC0768] Postel, J., "User Datagram Protocol", STD 6, RFC 768, August 1980.
- [RFC0791] Postel, J., "Internet Protocol", STD 5, RFC 791, September 1981.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.
- [RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, May 2008.

10.2. Informative References

- [NSH] Quinn, P. and et al. , "Network Service Header", 2014.
- [RFC1191] Mogul, J. and S. Deering, "Path MTU discovery", RFC 1191, November 1990.
- [RFC1700] Reynolds, J. and J. Postel, "Assigned Numbers", RFC 1700, October 1994.
- [RFC1981] McCann, J., Deering, S., and J. Mogul, "Path MTU Discovery for IP version 6", RFC 1981, August 1996.
- [RFC6335] Cotton, M., Eggert, L., Touch, J., Westerlund, M., and S. Cheshire, "Internet Assigned Numbers Authority (IANA) Procedures for the Management of the Service Name and Transport Protocol Port Number Registry", BCP 165, RFC 6335, August 2011.
- [RFC6830] Farinacci, D., Fuller, V., Meyer, D., and D. Lewis, "The Locator/ID Separation Protocol (LISP)", RFC 6830, January 2013.
- [RFC7348] Mahalingam, M., Dutt, D., Duda, K., Agarwal, P., Kreeger, L., Sridhar, T., Bursell, M., and C. Wright, "Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks", RFC 7348, August 2014.

Authors' Addresses

Paul Quinn
Cisco Systems, Inc.

Email: paulq@cisco.com

Rajeev Manur
Broadcom

Email: rmanur@broadcom.com

Larry Kreeger
Cisco Systems, Inc.

Email: kreeger@cisco.com

Darrel Lewis
Cisco Systems, Inc.

Email: darlewis@cisco.com

Fabio Maino
Cisco Systems, Inc.

Email: fmaino@cisco.com

Michael Smith
Cisco Systems, Inc.

Email: michsmit@cisco.com

Puneet Agarwal

Email: puneet@acm.org

Lucy Yong
Huawei USA

Email: lucy.yong@huawei.com

Xiaohu Xu
Huawei Technologies

Email: xuxiaohu@huawei.com

Uri Elzur
Intel

Email: uri.elzur@intel.com

Pankaj Garg
Microsoft

Email: Garg.Pankaj@microsoft.com

David Melman
Marvell

Email: davidme@marvell.com

Network Working Group
Internet-Draft
Intended status: Experimental
Expires: September 6, 2015

F. Xia
B. Sarikaya
Huawei Technologies Co., Ltd.
S. Fan
China Telecom
March 5, 2015

Quality of Service Marking and Framework in Overlay Networks
draft-xia-nvo3-vxlan-qosmarking-04.txt

Abstract

Overlay networks such as The Virtual eXtensible Local Area Network enable multiple tenants to operate in a data center. Each tenant needs to be assigned a priority group to prioritize their traffic using tenant based quality of service marking. Also, cloud carriers wish to use quality of service to differentiate different applications, i.e. legacy, traffic based marking. For these purposes, Quality of Service bits are assigned in the Virtual eXtensible Local Area Network outer Ethernet header. How these bits are assigned and are processed in the network are explained in detail. Also the document presents a quality of service framework for overlay networks such as Virtual eXtensible Local Area Network.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <http://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on September 6, 2015.

Copyright Notice

Copyright (c) 2015 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	2
2. Terminology	3
3. Problem Statement	3
4. QoS Marking Schemes in VXLAN	4
4.1. Tenant Based Marking	5
4.2. Application Based Marking	6
4.3. QoS Bits in Outer IP Header	6
5. Quality of Service Operation at VXLAN Decapsulation Point . .	7
6. Quality of Service Operation at VXLAN Encapsulation Point . .	7
7. QoS processing for VXLAN outer IP header	8
8. Security Considerations	9
9. IANA considerations	9
10. Acknowledgements	9
11. References	9
11.1. Normative References	9
11.2. Informative References	10
Authors' Addresses	10

1. Introduction

Data center networks are being increasingly used by telecom operators as well as by enterprises. An important requirement in data center networks is multitenancy, i.e. multiple tenants each with their own isolated network domain. Virtual eXtensible Local Area Network (VXLAN) is a solution that is gaining popularity in industry [RFC7348]. VXLAN overlays a Layer 2 network over a Layer 3 network. Each overlay is identified by the VXLAN Network Identifier (VNI). VXLAN tunnel end point (VTEP) can be hosted at the the hypervisor on the server or higher above in the network. VXLAN encapsulation with a UDP header is only known to the VTEP, the Virtual Machines (VM) never sees it.

It should be noted that in this document, VTEP plays the role of the Network Virtualization Edge (NVE) according to NVO3 architecture for overlay networks like VXLAN or NVGRE defined in [I-D.ietf-nvo3-arch].

NVE interfaces the tenant system underneath with the L3 network called the Virtual Network (VN).

Since VXLAN allows multiple tenants to operate, data center operators are facing the problem of treating their traffic. There is interest to provide different quality of service to the tenants based on their service level agreements, i.e. tenant-based QoS.

Cloud carriers have interest in different quality of service to different applications such as voice, video, network control applications, etc. In this case, quality of service marking can be done using deep packet inspection (DPI) in order to detect the type of application in each packet, i.e. application based QoS.

In this document, we develop Quality of Service marking solution for VXLAN as part of the Quality of Service framework for overlay multi-tenant networks as such it complements VXLAN architecture defined in [RFC7348]. The solution is compatible with IP level Differentiated Services model or diffserv described in [RFC2474] and [RFC2475]. Configuration guidelines are described in [RFC4594]. Diffserv interconnection classes and interconnection practice are described in [I-D.geib-tsvwg-diffserv-intercon].

2. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119]. The terminology in this document is based on the definitions in [RFC7348].

3. Problem Statement

In an overlay network such as VXLAN network multiple tenants are supported. There is interest in assigning different priority to each tenant's traffic based on the premium that tenant pays, etc. In another words, cloud carriers would like to categorize tenants into different traffic classes such as diamond, gold, silver and bronze classes.

Cloud carriers wish to categorize the traffic based on the application such as voice, video, etc. Based on the type of the application different traffic classes may be identified and different priority levels can be assigned to each.

In order to do these, quality of service marking is needed in the overlay network.

The solution developed in this document is based on the Network Virtualization Edge (NVE) to do the marking at the encapsulation layer. The marking, especially the tenant based QoS marking has to be done when the frames are introduced by the virtual machines (VM) because it is the encapsulation layer that adds the tenant identification, e.g. VXLAN Network Identifier (VNI) to each frame coming from the VMs. Because of this tenant based Quality of Service marking SHOULD be done at the encapsulation layer.

4. QoS Marking Schemes in VXLAN

Three bits are reserved in VXLAN Outer Ethernet Header's C-Tag 802.1Q field/VLAN Tag Information field's priority code point (PCP) field shown as QoS-flag in Figure 1.

Three bits called QoS-flag are reserved to indicate the quality of service class that this packet belongs. These bits will be assigned according to the type of traffic carried in this flow, e.g. video, voice, critical application, etc. These assignments in relation to IP level Differentiated Services model, diffserv bits or DS field, are discussed in Section 7.

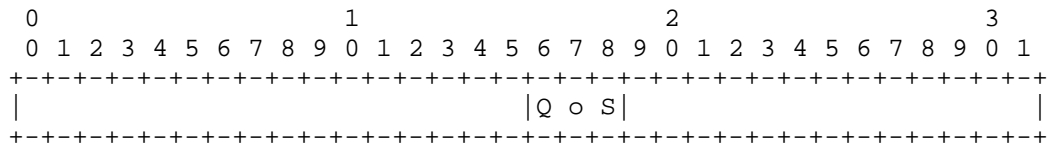


Figure 1: QoS Flag

Priority code point bits are assigned as follows in IEEE 802.1p [IEEE802.1D]:

- 001 - BK or background traffic
- 000 - BE or best effort traffic
- 010 - EE or Excellent Effort
- 011 - CA or Critical Applications
- 100 - VI or Video
- 101 - VO or Voice
- 110 - IC or Internetwork Control

111 - NC or Network Control

'111' has the highest priority while '001' has the lowest, for example, video traffic has higher priority than web surfing which is best effort traffic.

IEEE 802.1p [IEEE802.1D] based PCP marking is supported by most switches currently deployed that have the QoS capabilities.

We propose two different mappings to make use of the QoS field, tenant based marking or application based marking. Both of these markings are compatible with IEEE 802.1p PCP marking as well as the class selector codepoints defined by diffserv.

4.1. Tenant Based Marking

Tenant based marking is based on tenancy priorities. The cloud carrier categorizes its tenants into different groups such as diamond, gold, silver, bronze, standard and so on. All traffic for a diamond tenant has a high priority to be forwarded regardless of application types. The below is the mapping proposed in this document.

001 - Reserved

000 - Standard

010 - Bronze

011 - Silver

100 - Gold

101 - Diamond

110 - Emergency

111 - Reserved

The sender SHOULD assign bits 16-18 with bits assigned values as above if the quality of service treatment is needed on this packet. The sender SHOULD assign the same bit pattern to all the packets of the same tenant, i.e. the same VNI value.

4.2. Application Based Marking

This marking is based on application priorities. NVE uses some mechanism such as Deep Packet Inspection (DPI) to identify application types, and fills in the QoS field based on the identified application types. The below is a possible mapping.

001 - Reserved
000 - ftp/email
010 - Facebook, Twitter, Social Media
011 - Instant Message
100 - video
101 - voice
110 - High Performance computation
111 - Reserved

The sender SHOULD assign bits 16-18 with bits assigned values as above if the quality of service treatment is needed on this packet. The sender SHOULD assign the same bit pattern to all the packets of the same flow.

4.3. QoS Bits in Outer IP Header

Outer IPv4 header in VXLAN has type of service (TOS) field of 8 bits. Outer IPv6 header in VXLAN has traffic class field of 8 bits.

In case virtual machines are differentiated services capable, inner IP header contains per hop behavior (PHB) settings inline with diffserv RFCs. VXLAN NVE SHOULD copy inner IP header QoS bits into outer IP header bits.

In case virtual machines are not differentiated services capable, outer IP header QoS bits MUST be assigned by VXLAN NVE. VXLAN NVE SHOULD assign one of the class selector (CS) codepoints with value 'xxx000' corresponding to the marking explained above if the packet is a unicast packet. For more details, see Section 7.

5. Quality of Service Operation at VXLAN Decapsulation Point

There are two types of VXLAN packets receivers, that is, a VXLAN enabled NVE or a VXLAN gateway [I-D.sarikaya-nvo3-proxy-vxlan].

When the VXLAN enabled NVE receives the packet, it decapsulates the packet and delivers it downstream to a corresponding VM. If there are multiple packets to be processed, packets with high priority (that is higher QoS value) should be processed first.

The QoS operation is different for the VXLAN gateway processing. The gateway which provides VXLAN tunnel termination functions could be ToR/access switches or switches higher up in the data center network topology. For incoming frames on the VXLAN connected interface, the gateway strips out the VXLAN header and forwards to a physical port based on the destination MAC address of the inner Ethernet frame. If inner VLAN is included in the VXLAN frame or a VLAN is supposed to be added based on configuration, the VXLAN gateway decapsulates the VXLAN packet and remarks the QoS field of the outgoing Ethernet frame based on VXLAN Outer Ethernet Header QoS bits. The switch SHOULD copy the Q-Flags of VXLAN Outer Ethernet Header into IEEE 802.1p Priority Code Point (PCP) field in VLAN tag.

6. Quality of Service Operation at VXLAN Encapsulation Point

There are two types of VXLAN packet senders, that is, a VXLAN enabled NVE or a VXLAN gateway.

For a VXLAN enabled NVE, the upstream procedure is:

Reception of Frames

The VXLAN enabled NVE receives an Ethernet packet from a hosting VM.

Lookup

Making use of the destination of the Ethernet packet, the VXLAN enabled NVE looks up MAC-NVE mapping table, and retrieves IP address of destination NVE.

Acquisition of QoS parameters

There are two different ways to acquire QoS parameters for VXLAN encapsulation. The first is a dynamic one which requires a VXLAN enabled NVE has Deep Packet Inspection (DPI) capability and can identify different application types. The second is a static one

which requires a VM manager to assign QoS parameters to different VNIs based on premium that different tenancies pay.

Encapsulation of frames

The NVE then encapsulates the packet using VXLAN format with acquired QoS parameters and VNI. The specific format is given in Section 4. After the frame is encapsulated it is sent out upstream to the network.

For a VXLAN gateway, packets are encapsulated using VXLAN format with QoS field in a similar way. Once the VXLAN gateway receives a packet from a non-VXLAN domain, it encapsulates the packet with QoS parameters which are acquired through DPI or priorities of tenancies.

7. QoS processing for VXLAN outer IP header

QoS is user experience of end-to-end network operation. A packet from VM A to VM B normally traverses such network entities sequentially as virtual switch A which is co-located with VM A, TOR switch A, aggregation switch A, a core switch, aggregation switch B, TOR switch B, virtual switch B. VXLAN processing only takes place in virtual switches, and all other network entities only execute IP forwarding. VXLAN QoS mapping to outer IP header at virtual switch A is needed to achieve end-to-end QoS.

Six bits of the Differentiated Services Field (DS field) are used as a codepoint (DSCP) to select the per hop behaviour (PHB) a packet experiences at each node in a Differentiated Services Domain [RFC2474]. DS field is 8 bits long, 6 bits of it are used as DSCP and two bits are unused. DS field is carried in both IPv4 and IPv6 packet headers.

Using 6 bits of DS field, it is possible to define 64 codepoints. A pool of 32 RECOMMENDED codepoints called Pool 1 is standardized by IANA. 8 of these 32 codepoints are called class selector (CS) codepoints, from CS0 corresponding to '000000' to CS7 corresponding to '111000' codepoints. There are also 12 assured forwarding (AF) codepoints [RFC2597], an expedited forwarding (EF) per hop behavior [RFC3246] and VOICE-ADMIT codepoint for capacity-admitted traffic [RFC5865]. In this document we use class selector codepoints. Other codepoints are out of scope.

When a packet forwarded from non-VXLAN domain to VXLAN domain through a VXLAN gateway, DSCP field of outer IP header for unicast packets should be marked based on VXLAN QoS using the class selector codepoints, i.e. 'xxx000' that corresponds to VXLAN QoS marking, i.e.

with xxx assigned based on static or dynamic QoS markings defined above in Section 4.

8. Security Considerations

Special security considerations in [RFC7348] are applicable.

9. IANA considerations

None.

10. Acknowledgements

The authors are grateful to Brian Carpenter, David Black, Erik Nordmark for their constructive comments on our work.

11. References

11.1. Normative References

- [RFC0826] Plummer, D., "Ethernet Address Resolution Protocol: Or converting network protocol addresses to 48.bit Ethernet address for transmission on Ethernet hardware", STD 37, RFC 826, November 1982.
- [RFC0791] Postel, J., "Internet Protocol", STD 5, RFC 791, September 1981.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.
- [RFC2474] Nichols, K., Blake, S., Baker, F., and D. Black, "Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers", RFC 2474, December 1998.
- [RFC2475] Blake, S., Black, D., Carlson, M., Davies, E., Wang, Z., and W. Weiss, "An Architecture for Differentiated Services", RFC 2475, December 1998.
- [RFC2597] Heinanen, J., Baker, F., Weiss, W., and J. Wroclawski, "Assured Forwarding PHB Group", RFC 2597, June 1999.
- [RFC3246] Davie, B., Charny, A., Bennet, J., Benson, K., Le Boudec, J., Courtney, W., Davari, S., Firoiu, V., and D. Stiliadis, "An Expedited Forwarding PHB (Per-Hop Behavior)", RFC 3246, March 2002.

- [RFC4594] Babiarz, J., Chan, K., and F. Baker, "Configuration Guidelines for DiffServ Service Classes", RFC 4594, August 2006.
- [RFC5865] Baker, F., Polk, J., and M. Dolly, "A Differentiated Services Code Point (DSCP) for Capacity-Admitted Traffic", RFC 5865, May 2010.
- [I-D.ietf-nvo3-arch]
Black, D., Hudson, J., Kreeger, L., Lasserre, M., and T. Narten, "An Architecture for Overlay Networks (NVO3)", draft-ietf-nvo3-arch-02 (work in progress), October 2014.
- [IEEE802.1D]
IEEE, "Virtual Bridged Local Area Networks", IEEE Std 802.1D-2005, May 2006.

11.2. Informative References

- [RFC7348] Mahalingam, M., Dutt, D., Duda, K., Agarwal, P., Kreeger, L., Sridhar, T., Bursell, M., and C. Wright, "Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks", RFC 7348, August 2014.
- [I-D.geib-tsvwg-diffserv-intercon]
Geib, R. and D. Black, "DiffServ interconnection classes and practice", draft-geib-tsvwg-diffserv-intercon-08 (work in progress), November 2014.
- [I-D.sarikaya-nvo3-proxy-vxlan]
Sarikaya, B. and F. Xia, "Virtual eXtensible Local Area Network over IEEE 802.1Qbg", draft-sarikaya-nvo3-proxy-vxlan-00 (work in progress), October 2014.

Authors' Addresses

Frank Xia
Huawei Technologies Co., Ltd.
101 Software Avenue, Yuhua District
Nanjing, Jiangsu 210012, China

Phone: ++86-25-56625443
Email: xiayangsong@huawei.com

Behcet Sarikaya
Huawei Technologies Co., Ltd.
5340 Legacy Dr. Building 3
Plano, TX 75024

Phone: +1 972-509-5599
Email: sarikaya@ieee.org

Shi Fan
China Telecom
Room 708, No.118, Xizhimennei Street
Beijing , P.R. China 100035

Phone: +86-10-58552140
Email: shifan@ctbri.com.cn

INTERNET-DRAFT
Intended Status: Standards Track

B. Liu, Ed.
Huawei
R. Chen
ZTE
F. Qin
China Mobile
R. Rahman
Cisco

Expires: March 9, 2020

September 6, 2019

Base YANG Data Model for NVO3 Protocols
draft-zhang-nvo3-yang-cfg-07.txt

Abstract

This document describes the base YANG data model that can be used by operators to configure and manage Network Virtualization Overlay protocols. The model is focused on the common configuration requirement of various encapsulation options, such as VXLAN, NVGRE, GENEVE and VXLAN-GPE. Using this model as a starting point, incremental work can be done to satisfy the requirement of a specific encapsulation.

Status of this Memo

This Internet-Draft is submitted to IETF in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

The list of current Internet-Drafts can be accessed at
<http://www.ietf.org/lid-abstracts.html>

The list of Internet-Draft Shadow Directories can be accessed at
<http://www.ietf.org/shadow.html>

Copyright and License Notice

Copyright (c) 2019 IETF Trust and the persons identified as the

document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	3
2. Acronyms and Terminology	3
2.1. Acronyms	3
2.2. Terminology	3
3. The YANG Data Model for NVO3	3
3.1 Mapping to the NVO3 architecture	4
3.2. The Configuration Parameters	4
3.2.1. NVE as an interface	4
3.2.2. Virtual Network Instance	5
3.2.3. BUM Mode	5
3.3. Statistics	5
3.3. Model Structure	5
3.4. YANG Module	8
4. Security Considerations	24
5. IANA Considerations	24
6. Contributors	24
7. Acknowledgements	25
8. References	25
8.1. Normative References	25
8.2. Informative References	26
Author's Addresses	27

1. Introduction

Network Virtualization Overlays (NVO3), such as VXLAN, NVGRE, GENEVE and VXLAN-GPE, enable network virtualization for data center networks environment that assumes an IP-based underlay.

YANG [RFC6020] is a data definition language that was introduced to define the contents of a conceptual data store that allows networked devices to be managed using NETCONF [RFC6241]. This document specifies a YANG data model that can be used to configure and manage NVO3 protocols. The model covers the configuration of NVO3 instances as well as their operation states, which are the basic common requirements of the different tunnel encapsulations. Thus it is called "the base model for NVO3" in this document.

As the Network Virtualization Overlay evolves, newly defined tunnel encapsulation may require extra configuration. For example, GENEVE may require configuration of TLVs at the NVE. The base module can be augmented to accommodate these new solutions.

2. Acronyms and Terminology

2.1. Acronyms

NVO3: Network Virtualization Overlays
VNI: Virtual Network Instance
BUM: Broadcast, Unknown Unicast, Multicast traffic

2.2. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

Familiarity with [RFC7348], [RFC7364], [RFC7365] and [RFC8014] is assumed in this document.

3. The YANG Data Model for NVO3

The NVO3 base YANG model defined in this document is used to configure the NVEs. It is divided into three containers. The first container contains the configuration of the virtual network instances, e.g. the VNI, the NVE that the instance is mounted, the peer NVEs which can be determined dynamically via a control plane or given statically, and the statistical states of the instance. The other two containers are separately the statistical states of the

peer NVEs and the tunnels.

3.1 Mapping to the NVO3 architecture

The NVO3 base YANG model is defined according to the NVO3 architecture [RFC8014]. As shown in Figure 3.1, the reference model of the NVE defined in [RFC8014], multiple instances can be mounted under a NVE. The key of the instance is VNI. The source NVE of the instance is the NVE configured by the base YANG. An instance can have several peer NVEs. A NVO3 tunnel can be determined by the VNI, the source NVE and the peer NVE. The tunnel can be built statically by manually indicate the addresses of the peer NVEs, or dynamically via a control plane, e.g. EVPN [RFC8365]. An enabler is defined in the NVO3 base YANG to choose from these two modes.

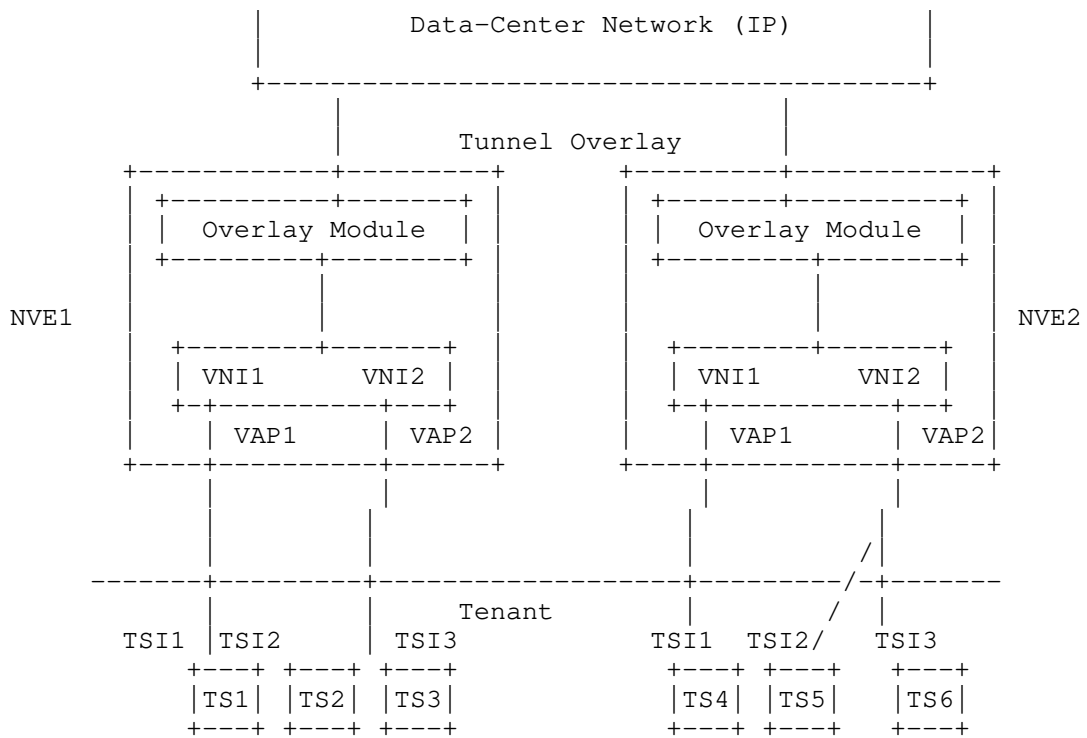


Figure 3.1. NVE Reference model in RFC 8014

3.2. The Configuration Parameters

3.2.1. NVE as an interface

A NVE in the NVO3 base YANG is defined via augmenting the IETF

interface YANG. If anycast gateway is enabled, the source VTEP address is the address of the anycast gateway, and a bypass address is used to uniquely identify the NVE. Otherwise, the source VTEP address is the NVE interface's own IP address.

3.2.2. Virtual Network Instance

A Virtual Network Instance ('VNI') is a specific VN instance on an NVE [RFC7365]. At each NVE, a Tenant System is connect to VNIs through Virtual Access Points (VAP). VAPs can be physical ports or virtual ports identified by the bridge domain Identifier ('bdId'). The mapping between VNI and bdId is managed by the operator.

As defined in [draft-ietf-bess-evpn-inter-subnet-forwarding], a tenant can have multiple bridge domains, and each domain has its own VNI. Thus these VNIs are used as L2VPN. Besides, a dedicated VNI can be used for routing between the bridge domains, i.e. used as L3VPN. The mapping relationship between VNI and L2VPN (respectively, L3VPN) is given by augmenting the IETF YANG of L2VPN (respectively L3VPN).

3.2.3. BUM Mode

An NVE SHOULD support either ingress replication, or multicast proxy, or point to multipoint tunnels on a per-VNI basis. It is possible that both modes be used simultaneously in one NVO3 network by different NVEs.

If ingress replication is used, the receiver addresses are listed in 'peers'. If multicast proxy [RFC8293] is used, the proxy's address is given in "flood-proxy". If the choice is point to multipoint tunnels, the multicast address is given as 'multiAddr'.

3.3. Statistics

Operators can determine whether a NVE should gather statistic values on a per-VNI basis. An enabler is contained in the 'static' list as 'statistic-enable' leaf. If the gathering for a VNI is enabled, the statistical information about the local NVEs, the remote NVEs, the flows and the MAC addresses will be collected by the NVEs in this VNI.

3.3. Model Structure

```

module: ietf-nvo3
  +--rw nvo3
  |   +--rw vni-instances
  |   |   +--rw vni-instance* [vni-id]
  |   |   |   +--rw vni-id                uint32
  |   |   |   +--rw vni-mode              enumeration

```

```

+--rw source-nve                if:interface-ref
+--rw protocol-bgp?             boolean
+--ro status?                   vni-status-type
+--rw static-ipv4-peers
|   +--rw static-peer* [peer-ip]
|   |   +--rw peer-ip            inet:ipv4-address-no-zone
|   |   +--rw out-vni-id?        uint32
+--rw static-ipv6-peers
|   +--rw static-ipv6-peer* [peer-ip]
|   |   +--rw peer-ip            inet:ipv6-address-no-zone
+--rw flood-proxys
|   +--rw flood-proxy* [peer-ip]
|   |   +--rw peer-ip            inet:ipv4-address-no-zone
+--rw mcast-groups
|   +--rw mcast-group* [mcast-ip]
|   |   +--rw mcast-ip           inet:ipv4-address-no-zone
+--rw statistic
|   +--rw statistic-enable?      boolean
|   +--ro statistic-info
|   |   +--ro rx-bits-per-sec?    uint64
|   |   +--ro rx-pkt-per-sec?     uint64
|   |   +--ro tx-bits-per-sec?    uint64
|   |   +--ro tx-pkt-per-sec?     uint64
|   |   +--ro rx-pkts?            uint64
|   |   +--ro rx-bytes?           uint64
|   |   +--ro tx-pkts?            uint64
|   |   +--ro tx-bytes?           uint64
|   |   +--ro rx-unicast-pkts?    uint64
|   |   +--ro rx-multicast-pkts?  uint64
|   |   +--ro rx-broadcast-pkts?  uint64
|   |   +--ro drop-unicast-pkts?  uint64
|   |   +--ro drop-multicast-pkts? uint64
|   |   +--ro drop-broadcast-pkts? uint64
|   |   +--ro tx-unicast-pkts?    uint64
|   |   +--ro tx-multicast-pkts?  uint64
|   |   +--ro tx-broadcast-pkts?  uint64
+--ro vni-peer-infos
|   +--ro peers
|   |   +--ro peer* [vni-id source-ip peer-ip]
|   |   |   +--ro vni-id          uint32
|   |   |   +--ro source-ip       inet:ip-address-no-zone
|   |   |   +--ro peer-ip         inet:ip-address-no-zone
|   |   |   +--ro tunnel-type?    peer-type
|   |   |   +--ro out-vni-id?     uint32
+--ro tunnel-infos
|   +--ro tunnel-info* [tunnel-id]
|   |   +--ro tunnel-id           uint32
|   |   +--ro source-ip?          inet:ip-address-no-zone

```

```

    +--ro peer-ip?          inet:ip-address-no-zone
    +--ro status?           tunnel-status
    +--ro type?             tunnel-type
    +--ro up-time?          string
    +--ro vrf-name?         -> /ni:network-instances/network-instance/name

augment /if:interfaces/if:interface:
  +--rw nvo3-nve
    +--rw nvo3-config
      +--rw source-vtep-ip?    inet:ipv4-address-no-zone
      +--rw source-vtep-ipv6?  inet:ipv6-address-no-zone
      +--rw bypass-vtep-ip?    inet:ipv4-address-no-zone
      +--rw statistics
        +--rw statistic* [vni-id mode peer-ip direction]
          +--rw vni-id        uint32
          +--rw mode          vni-type
          +--rw peer-ip       inet:ipv4-address-no-zone
          +--rw direction     direction-type
          +--ro info
            +--ro rx-pkts?      uint64
            +--ro rx-bytes?     uint64
            +--ro tx-pkts?      uint64
            +--ro tx-bytes?     uint64
            +--ro rx-unicast-pkts? uint64
            +--ro rx-multicast-pkts? uint64
            +--ro rx-broadcast-pkts? uint64
            +--ro tx-unicast-pkts? uint64
            +--ro tx-multicast-pkts? uint64
            +--ro tx-broadcast-pkts? uint64
            +--ro drop-unicast-pkts? uint64
            +--ro drop-multicast-pkts? uint64
            +--ro drop-broadcast-pkts? uint64
            +--ro rx-bits-per-sec? uint64
            +--ro rx-pkt-per-sec? uint64
            +--ro tx-bits-per-sec? uint64
            +--ro tx-pkt-per-sec? uint64
      +--rw nvo3-gateway
        +--rw nvo3-gateway
          +--rw vxlan-anycast-gateway? boolean
    augment /ni:network-instances/ni:network-instance/ni:ni-type/l3vpn:l3vpn/l3
    vpn:l3vpn:
      +--rw vni-lists
        +--rw vni* [vni-id]
          +--rw vni-id      uint32
    augment /ni:network-instances/ni:network-instance/ni:ni-type/l2vpn:l2vpn:
    +--rw vni-lists
      +--rw vni* [vni-id]
        +--rw vni-id      uint32
        +--rw split-horizon-mode? vni-bind-type

```



```

        +---rw split-group?          string

    rpcs:
      +---x reset-vni-instance-statistic
      |   +---w input
      |   |   +---w vni-id          uint32
      +---x reset-vni-peer-statistic
      |   +---w input
      |   |   +---w vni-id          uint32
      |   |   +---w mode            vni-type
      |   |   +---w peer-ip         inet:ipv4-address-no-zone
      |   |   +---w direction       direction-type

```

Figure 3.2. The tree structure of YANG module for NVO3 configuration

3.4. YANG Module

```

<CODE BEGINS> file "ietf-nvo3-base@2019-07-01.yang"
module ietf-nvo3 {
  yang-version 1.1;
  namespace "urn:ietf:params:xml:ns:yang:ietf-nvo3";
  prefix "nvo3";

  import ietf-network-instance {
    prefix "ni";
  }

  import ietf-interfaces {
    prefix "if";
  }

  import ietf-inet-types {
    prefix "inet";
  }

  import ietf-l2vpn {
    prefix "l2vpn";
  }

  import ietf-bgp-l3vpn {
    prefix "l3vpn";
  }

  organization "ietf";
  contact "ietf";
  description "Yang model for NVO3";

  revision 2019-04-01 {

```

```
    description
      "Init revision";
    reference
      "";
  }

  typedef vni-status-type {
    type enumeration {
      enum "up" {
        description
          "Vni status up.";
      }
      enum "down" {
        description
          "Vni status down.";
      }
    }
    description
      "Vni status";
  }

  typedef vni-type {
    type enumeration {
      enum "l2" {
        description
          "layer 2 mode";
      }
      enum "l3" {
        description
          "layer 3 mode";
      }
    }
    description
      "vni type";
  }

  typedef peer-type {
    type enumeration {
      enum "static" {
        description
          "Static.";
      }
      enum "dynamic" {
        description
          "Dynamic.";
      }
    }
    description
```

```
        "Peer type";
    }

typedef tunnel-status {
    type enumeration {
        enum "up" {
            description
                "The tunnel is up.";
        }
        enum "down" {
            description
                "The tunnel is down.";
        }
    }
    description
        "Tunnel status";
}

typedef tunnel-type {
    type enumeration {
        enum "dynamic" {
            description
                "The tunnel is dynamic.";
        }
        enum "static" {
            description
                "The tunnel is static.";
        }
        enum "invalid" {
            description
                "The tunnel is invalid.";
        }
    }
    description
        "Tunnel type";
}

typedef direction-type {
    type enumeration {
        enum "inbound" {
            description
                "Inbound.";
        }
        enum "outbound" {
            description
                "Outbound.";
        }
        enum "bidirection" {
            description
```

```
        "Bidirection.";
    }
}
description
    "Bound direction";
}

typedef vni-bind-type {
    type enumeration {
        enum "hub-mode" {
            description
                "Hub mode.";
        }
        enum "spoke-mode" {
            description
                "Spoke mode.";
        }
    }
    description
        "bdBindVniType";
}

container nvo3 {
    description
        "Management of NVO3.";

    container vni-instances {
        description
            "The confiuration and information table of the VNI.";
        list vni-instance {
            key "vni-id";
            must "(/if:interfaces/if:interface[if:name = current()/source_nve]/if
:type = 'Nve')";
            description
                "The confiuration and information of the VNI.";
            leaf vni-id {
                type uint32 {
                    range "1..16777215";
                }
                description
                    "The id of VNI.";
            }
            leaf vni-mode {
                type enumeration {
                    enum "Local" {
                        description
                            "Local mode";
                    }
                    enum "Global" {
```

```
        description
            "Global mode";
        }
    }
    description
        "The mode of the VNI instance.";
}
leaf source-nve {
    type if:interface-ref;
    mandatory true;
    description
        "The name of the nve interface .";
}
leaf protocol-bgp {
    type boolean;
    default "false";
    description
        "Whether use bgp as vxlan's protocol.";
}
leaf status {
    type vni-status-type;
    config false;
    description
        "The status of the VNI.";
}
container static-ipv4-peers {
    description
        "The remote NVE address table in a same VNI.";
    list static-peer {
        key "peer-ip";
        description
            "The remote NVE address in a same VNI.";
        leaf peer-ip {
            type inet:ipv4-address-no-zone;
            description
                "The address of the NVE.";
        }
        leaf out-vni-id {
            type uint32 {
                range "1..16777215";
            }
            description
                "The ID of the out VNI. Do not support separate deletion.";
        }
    }
}
container static-ipv6-peers {
    description
```

```
    "The remote NVE ipv6 address table in a same VNI.";
  list static-ipv6-peer {
    key "peer-ip";
    description
      "The remote NVE ipv6 address in a same VNI.";
    leaf peer-ip {
      type inet:ipv6-address-no-zone;
      description
        "The ipv6 address of the NVE.";
    }
  }
}
container flood-proxys {
  description
    "The flood proxys for this VNI";
  list flood-proxy {
    key "peer-ip";
    leaf peer-ip {
      type inet:ipv4-address-no-zone;
      description
        "peer ip address";
    }
    description
      "List of the flood proxys";
  }
}
container mcast-groups {
  description
    "The mcast address table.";
  list mcast-group {
    key "mcast-ip";
    description
      "The mcast address.";
    leaf mcast-ip {
      type inet:ipv4-address-no-zone;
      description
        "The mcast address of NVO3.";
    }
  }
}
container statistic {
  description
    "The VNI member in a same NVE.";
  leaf statistic-enable {
    type boolean;
    default "false";
    description
      "To determine whether to enable the statistics for a VNI.";
```

```
}
container statistic-info {
  config false;
  description
    "The vni instance traffic statistics information.";
  leaf rx-bits-per-sec {
    type uint64;
    config false;
    description
      "Number of bits received per second.";
  }
  leaf rx-pkt-per-sec {
    type uint64;
    config false;
    description
      "Number of packets received per second.";
  }
  leaf tx-bits-per-sec {
    type uint64;
    config false;
    description
      "Number of bits sent per second.";
  }
  leaf tx-pkt-per-sec {
    type uint64;
    config false;
    description
      "Number of packets sent per second.";
  }
  leaf rx-pkts {
    type uint64;
    config false;
    description
      "Total number of received packets.";
  }
  leaf rx-bytes {
    type uint64;
    config false;
    description
      "Total number of received bytes.";
  }
  leaf tx-pkts {
    type uint64;
    config false;
    description
      "Total number of sent packets.";
  }
  leaf tx-bytes {
```

```
        type uint64;
        config false;
        description
            "Total number of sent bytes.";
    }
    leaf rx-unicast-pkts {
        type uint64;
        config false;
        description
            "Number of received unicast packets.";
    }
    leaf rx-multicast-pkts {
        type uint64;
        config false;
        description
            "Number of received multicast packets.";
    }
    leaf rx-broadcast-pkts {
        type uint64;
        config false;
        description
            "Number of received broadcast packets.";
    }
    leaf drop-unicast-pkts {
        type uint64;
        config false;
        description
            "Number of discarded unicast packets.";
    }
    leaf drop-multicast-pkts {
        type uint64;
        config false;
        description
            "Number of discarded multicast packets.";
    }
    leaf drop-broadcast-pkts {
        type uint64;
        config false;
        description
            "Number of discarded broadcast packets.";
    }
    leaf tx-unicast-pkts {
        type uint64;
        config false;
        description
            "Number of sent unicast packets.";
    }
    leaf tx-multicast-pkts {
```



```

        type uint64;
        config false;
        description
            "Number of sent multicast packets.";
    }
    leaf tx-broadcast-pkts {
        type uint64;
        config false;
        description
            "Number of sent broadcast packets.";
    }
}

}

}

}

container vni-peer-infos {
    config false;
    description
        "The information table of vni members.";
    container peers {
        config false;
        description
            "The remote nve address in a same VNI.";
        list peer {
            key "vni-id source-ip peer-ip";
            config false;
            description
                "The remote nve address list in a same VNI.";
            leaf vni-id {
                type uint32 {
                    range "1..16777215";
                }
                config false;
                description
                    "The ID of VNI.";
            }
            leaf source-ip {
                type inet:ip-address-no-zone;
                config false;
                description
                    "The source address of the NVE interface.";
            }
            leaf peer-ip {
                type inet:ip-address-no-zone;
                config false;
                description

```

```
        "The remote NVE address.";
    }
    leaf tunnel-type {
        type peer-type;
        config false;
        description
            "Tunnel type.";
    }
    leaf out-vni-id {
        type uint32 {
            range "1..16777215";
        }
        config false;
        description
            "The ID of the out VNI.";
    }
}
}
}

container tunnel-infos {
    config false;
    description
        "VxLAN tunnel information.";
    list tunnel-info {
        key "tunnel-id";
        config false;
        description
            "VxLAN tunnel information list.";
        leaf tunnel-id {
            type uint32 {
                range "1..4294967295";
            }
            config false;
            description
                "The ID of Vxlan tunnel.";
        }
        leaf source-ip {
            type inet:ip-address-no-zone;
            config false;
            description
                "Local NVE interface address.";
        }
        leaf peer-ip {
            type inet:ip-address-no-zone;
            config false;
            description
                "Remote NVE interface address.";
```

```
    }
    leaf status {
        type tunnel-status;
        config false;
        description
            "Tunnel status.";
    }
    leaf type {
        type tunnel-type;
        config false;
        description
            "Tunnel type.";
    }
    leaf up-time {
        type string {
            length "1..10";
        }
        config false;
        description
            "Vxlan tunnel up time.";
    }
    leaf vrf-name {
        type leafref {
            path "/ni:network-instances/ni:network-instance/ni:name";
        }
        default "_public_";
        config false;
        description
            "The name of VPN instance.";
    }
}

augment "/if:interfaces/if:interface" {
    description
        "Augment the interface, NVE as an interface.";
    container nvo3-nve {
        when "if:interfaces/if:interface/if:type = 'Nve'";
        description
            "Network virtualization edge.";
        leaf source-vtep-ip {
            type inet:ipv4-address-no-zone;
            description
                "The source address of the NVE interface.";
        }
        leaf source-vtep-ipv6 {
            type inet:ipv6-address-no-zone;
            description

```

```
        "The source ipv6 address of the NVE interface.";
    }
    leaf bypass-vtep-ip {
        type inet:ipv4-address-no-zone;
        description
            "The source address of bypass VXLAN tunnel.";
    }
    container statistics {
        description
            "VXLAN Tunnel Traffic Statistical Configuration Table.";
        list statistic {
            key "vni-id mode peer-ip direction";
            description
                "VXLAN Tunnel Traffic Statistics Configuration.";
            leaf vni-id {
                type uint32 {
                    range "1..16777215";
                }
                description
                    "ID of the VNI.";
            }
            leaf mode {
                type vni-type;
                description
                    "The type of the NVE interface.";
            }
            leaf peer-ip {
                type inet:ipv4-address-no-zone;
                description
                    "IP address of the remote VTEP.";
            }
            leaf direction {
                type direction-type;
                must "(./mode='l3' and ./bound!='bidirection')";
                description
                    "Traffic statistics type about the VXLAN tunnel.";
            }
        }
        container info {
            config false;
            description
                "Traffic statistics about the peer.";
            leaf rx-pkts {
                type uint64;
                config false;
                description
                    "Total number of received packets.";
            }
            leaf rx-bytes {
```

```
        type uint64;
        config false;
        description
            "Total number of received bytes.";
    }
    leaf tx-pkts {
        type uint64;
        config false;
        description
            "Total number of sent packets.";
    }
    leaf tx-bytes {
        type uint64;
        config false;
        description
            "Total number of sent bytes.";
    }
    leaf rx-unicast-pkts {
        type uint64;
        config false;
        description
            "Number of received unicast packets.";
    }
    leaf rx-multicast-pkts {
        type uint64;
        config false;
        description
            "Number of received multicast packets.";
    }
    leaf rx-broadcast-pkts {
        type uint64;
        config false;
        description
            "Number of received broadcast packets.";
    }
    leaf tx-unicast-pkts {
        type uint64;
        config false;
        description
            "Number of sent unicast packets.";
    }
    leaf tx-multicast-pkts {
        type uint64;
        config false;
        description
            "Number of sent multicast packets.";
    }
    leaf tx-broadcast-pkts {
```

```
        type uint64;
        config false;
        description
            "Number of sent broadcast packets.";
    }
    leaf drop-unicast-pkts {
        type uint64;
        config false;
        description
            "Number of discarded unicast packets.";
    }
    leaf drop-multicast-pkts {
        type uint64;
        config false;
        description
            "Number of discarded multicast packets.";
    }
    leaf drop-broadcast-pkts {
        type uint64;
        config false;
        description
            "Number of discarded broadcast packets.";
    }
    leaf rx-bits-per-sec {
        type uint64;
        config false;
        description
            "Number of bits received per second.";
    }
    leaf rx-pkt-per-sec {
        type uint64;
        config false;
        description
            "Number of packets received per second.";
    }
    leaf tx-bits-per-sec {
        type uint64;
        config false;
        description
            "Number of bits sent per second.";
    }
    leaf tx-pkt-per-sec {
        type uint64;
        config false;
        description
            "Number of packets sent per second.";
    }
}
```

```
    }
  }

}

container nvo3-gateway {
  when "if:interfaces/if:interface/if:type = 'Vbdif'";
  description
    "Enable anycast gateway.";
  leaf vxlan-anycast-gateway {
    type boolean;
    default "false";
    description
      "Enable vxlan anycast gateway.";
  }
}

}

augment "/ni:network-instances/ni:network-instance/ni:ni-type" +
  "/l3vpn:l3vpn/l3vpn:l3vpn" {
  description "Augment for l3vpn instance";
  container vni-lists {
    description "Vni list for l3vpn";
    list vni {
      key "vni-id";
      description
        "Vni for current l3vpn instance";
      leaf vni-id {
        type uint32 {
          range "1..16777215";
        }
        description
          "The id of VNI.";
      }
    }
  }
}

}

augment "/ni:network-instances/ni:network-instance/ni:ni-type" +
  "/l2vpn:l2vpn" {
  description "Augment for l2vpn instance";
  container vni-lists {
    description "Vni list for l2vpn";
    list vni {
      key "vni-id";
      description
        "Vni for current l2vpn instance";
      leaf vni-id {
        type uint32 {
```

```
        range "1..16777215";
    }
    description
        "The id of VNI.";
}
leaf split-horizon-mode {
    type vni-bind-type;
    default "hub-mode";
    description
        "Split horizon mode.";
}
leaf split-group {
    type string {
        length "1..31";
    }
    description
        "Split group name.";
}
}
}
}

rpc reset-vni-instance-statistic {
    description
        "Clear traffic statistics about the VNI.";
    input {
        leaf vni-id {
            type uint32 {
                range "1..16777215";
            }
            mandatory true;
            description
                "ID of the VNI.";
        }
    }
}

rpc reset-vni-peer-statistic {
    description
        "Clear traffic statistics about the VXLAN tunnel.";
    input {
        leaf vni-id {
            type uint32 {
                range "1..16777215";
            }
            mandatory true;
            description
                "ID of the VNI.";
        }
    }
}
```



```
    leaf mode {
        type vni-type;
        mandatory true;
        description
            "The type of vni memeber statistic.";
    }
    leaf peer-ip {
        type inet:ipv4-address-no-zone;
        mandatory true;
        description
            "IP address of the remote NVE interface.";
    }
    leaf direction{
        type direction-type;
        must "(./mode='mode-l3' and ./bound!='bidirection')";
        mandatory true;
        description
            "Traffic statistics type about the VXLAN tunnel.";
    }
}
}
```

<CODE ENDS>

4. Security Considerations

This document raises no new security issues.

5. IANA Considerations

The namespace URI defined in Section 3.3 need be registered in the IETF XML registry [RFC3688].

This document need to register the 'ietf-nvo3-base' YANG module in the YANG Module Names registry [RFC6020].

6. Contributors

Haibo Wang
Huawei
Email: rainsword.wang@huawei.com

Yuan Gao
Huawei
Email: sean.gao@huawei.com

Gang Yan

Huawei
Email: yangang@huawei.com

Mingui Zhang
Huawei
Email: zhangmingui@huawei.com

Yubao (Bob) Wang
ZTE Corporation
Email: yubao.wang2008@hotmail.com

Ruixue Wang
China Mobile
Email: wangruixue@chinamobile.com

Sijun Weng
China Mobile
Email: wengsijun@chinamobile.com

7. Acknowledgements

Authors would like to thank the comments and suggestions from Tao Han, Weilian Jiang.

8. References

8.1. Normative References

- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.
- [RFC7364] T. Narten, E. Gray, et al, "Problem Statement: Overlays for Network Virtualization", draft-ietf-nvo3-overlay-problem-statement, working in progress.
- [RFC7365] Marc Lasserre, Florin Balus, et al, "Framework for DC Network Virtualization", draft-ietf-nvo3-framework, working in progress.
- [RFC7348] Mahalingam, M., Dutt, D., Duda, K., Agarwal, P., Kreeger, L., Sridhar, T., Bursell, M., and C. Wright, "Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks", RFC 7348, August 2014.
- [I-D.ietf-nvo3-geneve] Gross, J., Ganga, I., and T. Sridhar, "Geneve: Generic Network Virtualization Encapsulation", draft-ietf-

nvo3-geneve-10 (work in progress), March 2019.

- [RFC3688] Mealling, M., "The IETF XML Registry", BCP 81, RFC 3688, January 2004.
- [RFC6020] Bjorklund, M., Ed., "YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF)", RFC 6020, October 2010.
- [RFC6241] Enns, R., Ed., Bjorklund, M., Ed., Schoenwaelder, J., Ed., and A. Bierman, Ed., "Network Configuration Protocol (NETCONF)", RFC 6241, June 2011.
- [RFC8014] D. Black, J. Hudson, L. Kreeger, M. Lasserre, T. Narten, An Architecture for Data-Center Network Virtualization over Layer 3 (NVO3), RFC8014, December 2016.

8.2. Informative References

- [RFC7637] M. Sridharan, A. Greenberg, et al, "NVGRE: Network Virtualization using Generic Routing Encapsulation", RFC7637, September 2015.
- [I-D.ietf-nvo3-vxlan-gpe] Maino, F., Kreeger, L., and U. Elzur, "Generic Protocol Extension for VXLAN", draft-ietf-nvo3-vxlan-gpe-06 (work in progress), April 2018.
- [I-D.draft-ietf-bess-evpn-inter-subnet-forwarding] A. Sajassi, S. Salam, S. Thoria, J. Drake, J. Rabadan, "Integrated Routing and Bridging in EVPN", draft-ietf-bess-evpn-inter-subnet-forwarding-08, March 4, 2019.
- [RFC8293] A. Ghanwani, L. Dunbar, V. Bannai, M. McBride, R. Krishnan, "A Framework for Multicast in Network Virtualization over Layer 3", RFC8293, January 2018.

Author's Addresses

Bing Liu
Huawei Technologies
No. 156 Beiqing Rd. Haidian District,
Beijing 100095
P.R. China

Email: remy.liubing@huawei.com

Ran Chen
ZTE Corporation

Email: chen.ran@zte.com.cn

Fengwei Qin
China Mobile
32 Xuanwumen West Ave, Xicheng District
Beijing, Beijing 100053
China

Email: qinfengwei@chinamobile.com

Reshad Rahman
Cisco Systems

Email: rrahman@cisco.com