intarea Working Group
Internet-Draft
Intended status: Standards Track
Expires: June 28, 2015

Y. Cui
L. Li
C. Liu
J. Wu
Tsinghua University
F. Baker
Cisco Systems

December 25, 2014

DHCPv6 Options for Discovery of 464XLAT IPv6 Prefixes
draft-cui-intarea-464xlat-prefix-dhcp-00

Abstract

464XLAT provides limited IPv4 connectivity across an IPv6-only network using translation technology. The customer-side translator (CLAT) performs stateless 1:1 mapping of an IPv4 destination address into a provider-side translator (PLAT) IPv6 prefix, which subsequently translates it back into IPv4. Different PLATs will likely have different IPv6 prefixes, to attract traffic to the correct PLAT. Thus, an automatic PLAT-side prefix discovery method is necessary for CLATs.

This document defines a DHCPv6-based method to inform a CLAT of a PLAT’s IPv6 prefix and the IPv4 prefixes it serves.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

464XLAT [RFC6877] describes an IPv4-over-IPv6 solution as one technique for IPv4 service extension and encouragement of IPv6 deployment. The 464XLAT architecture uses IPv4/IPv6 translation, described in [RFC6144], and standardized in [RFC6052], [RFC6145], and [RFC6146]. It encourages the IPv6 transition by making IPv4 service reachable across IPv6-only networks and providing IPv6 and IPv4 connectivity to single-stack IPv4 or IPv6 servers and peers. In the 464XLAT architecture, the CLAT must determine which of potentially several PLAT-side translation IPv6 prefix to use in order to send a packet to the PLAT with connectivity to its destination.

[RFC7050] describes a mechanism to learn the PLAT-side IPv6 prefix for protocol translation by DNS64 [RFC6147]. Although it supports multiple PLAT-side prefix by responding with multiple AAAA records to a DNS64 query, it does not support mapping IPv4 prefixes to IPv6

prefix, which would be required, for example, if one PLAT has connectivity to the general Internet following a default route, another has connectivity to a BGP peer, and a third has connectivity to a network using private addressing [RFC1918]. Therefore, in the scenario with multiple PLATs, [RFC7050] does not directly support destination-based IPv4 routing among PLATs; instead, the DNS64 database must contain equivalent information. It also requires the additional deployment of DNS64 service in customer-side networks, which is not required in 464XLAT deployment.

This document proposes a method for PLAT-side IPv6 prefix discovery based on DHCPv6, which is widely deployed and supported in customer networks. It defines two new dhcpv6 options for use by a CLAT to discover the PLAT-side translation IPv6 prefix(es). Also, the proposed mechanism can deal with the scenario with multiple independent DNS64 databases supporting separate PLATs.

2. 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. New DHCPv6 Option

3.1. PLAT Prefix List Option Format

The PLAT Prefix List Option is a container for PLAT Prefix Option(s). A PLAT Prefix List Option MAY contain multiple PLAT Prefix Options.

The format of the PLAT Prefix List Option 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
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   OPTION_PLAT_PREFIX_LIST     |       option-length           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                           PLAT_PREFIX-options                 +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

- o option-code: OPTION_PLAT_PREFIX_LIST (TBA1)

- o option-length: length of PLAT_PREFIX-options, specified in octets.

- o PLAT_PREFIX-options: one or more OPTION_PLAT_PREFIX options.

3.2. PLAT Prefix Option Format

The PLAT Prefix Option is encapsulated in the PLAT Prefix List Option. This option allows the mapping of destination IPv4 address ranges (contained in the IPv4 Prefix List) to a PLAT IPv6 prefix. If there is more than one such prefix, each prefix comes in its own option, with its associated IPv4 prefix list. In this way, the CLAT can select the PLAT with the corresponding destination IPv4 address.

The format of the PLAT Prefix Option 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
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   OPTION_PLAT_PREFIX          |         option-length         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| platv6-prelen |                                               |
+-+-+-+-+-+-+-+-+          platv6-prefix                        |
|                       (variable length)                       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
.                           (optional)                          .
.              IPv4 Prefix List (variable length)               .
.                       (see Figure 3)                          .
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

 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
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| IPv4-prelen   |   IPv4 Prefix (32 bits)                       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    (cont.)    | IPv4-prelen   | IPv4 Prefix (32 bits)         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    IPv4 Prefix (cont)         |                ...            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                            ...                                |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

- o option-code: OPTION_PLAT_PREFIX (TBA2)

- o option-length: 1 + length of platv6-prefix + length of IPv4 Prefix List, specified in octets.

- o platv6-prelen: length of platv6-prefix.

- o platv6-prefix: The PLAT IPv6 prefix that the CLAT used for IPv6 address synthesis.

- o IPv4 Prefix List: This is an optional field. The format of the

IPv4 Prefix List is shown in Figure 3. It is a list of zero or more IPv4 Prefixes. Each entry is formed by IPv4-prelen and IPv4 Prefix. The total length of the field is 5*number of IPv4 prefixes.

- o IPv4-prelen: the length of the IPv4 Prefix.

- o IPv4 Prefix: the destination-based IPv4 Prefix. The length is 4 octets.

4. Client Behavior

The client requests the OPTION_PLAT_PREFIX_LIST option using the Option Request option (ORO) in every Solicit, Request, Renew, Rebind, and Information-request message. If the DHCPv6 server includes the OPTION_PLAT_PREFIX_LIST option in its response, the CLAT may use the contained platv6-prefix to translate the destination IPv4 address into the destination IPv6 address.

When receiving the OPTION_PLAT_PREFIX option with IPv4 Prefix List, the CLAT MUST record the received IPv6 prefix and the corresponding IPv4 prefixes in IPv4 Prefix List. When receiving the OPTION_PLAT_PREFIX option without IPv4 Prefix List, the CLAT MUST treat the IPv6 prefix and the default IPv4 prefix 0.0.0.0/0 as one of the records.

If the CLAT loses contact with the DHCPv6 server, the CLAT SHOULD clear the prefix(es) it learned from the DHCPv6 server.

When translating the destination IPv4 address into the destination IPv6 address, CLAT MUST search an IPv4 routing database using the longest-match-first rule and select the IPv6 prefix offering that IPv4 prefix.

5. Message Flow Illustration

The figure below shows an example of message flow for a Client learning IPv6 prefixes using DHCPv6.

In this example, two IPv6 prefixes are provided by the DHCPv6 server. The first IPv6 prefix is 2001:db8:122:300::/56, the corresponding IPv4 prefixes are 192.0.2.0/24 and 198.51.100.0/24. The second IPv6 prefix is 2001:db8:122::/48, the corresponding IPv4 prefix is 192.0.2.128/25.

When the CLAT receives the packet with destination IPv4 address 192.0.2.1, according to the rule of longest prefix match, the PLAT with IPv6 prefix 2001:db8:122::/48 is chosen. In the same way, the

PLAT with IPv6 prefix 2001:db8:122::/48 is chosen.

```
+----------+                                     +-----------------+
|   CLAT   |                                     |  DHCPv6 server  |
+----------+                                     +-----------------+
     |            DHCPv6 query for IPv6 prefix            |
     |--------------------------------------------------->|
     |         ORO with OPTION_V6_PLATPREFIX_LIST         |
     |                                                    |
     |       DHCPv6 response with:                        |
     |         PLATPREFIX{                                |
     |           platv6-pre = 2001:db8:122:300::/56       |
     |           platv4-pre = 192.0.2.0/24                |
     |           platv4-pre = 198.51.100.0/24}            |
     |         PLATPREFIX{                                |
     |           platv6-pre = 2001:db8:122::/48           |
     |           platv4-pre = 192.0.2.128/25}             |
     |<---------------------------------------------------|
     |                                                    |
     |
     |                     +-----------------+   +-----------------+
     |                     |     PLAT 1      |   |     PLAT 2      |
     |                     +-----------------+   +-----------------+
     |                         platv6-pre =          platv6-pre =
     |                    2001:db8:122:300::/56   2001:db8:122::/48
     |                         platv4-pre =          platv4-pre =
     |                        192.0.2.0/24       192.0.2.128/25
     |                        198.51.100.0/24             |
     |                              |                     |
     |       Dest IPv4 addr:        |                     |
     |        192.0.2.1             |                     |
     |       Dest IPv6 addr:        |                     |
     |  2001:db8:122:300::c000:201  |                     |
     |----------------------------->|                     |
     |                              |                     |
     |                                                    |
     |     Dest IPv4 addr: 192.0.2.193                    |
     |     Dest IPv6 addr: 2001:db8:122::c000:2c1         |
     |--------------------------------------------------->|
```

6. Security Considerations

Considerations for security in this type of environment are primarily around the operation of the DHCPv6 protocol and the databases it uses.

In the DHCPv6 server, should the database be compromised, it will deliver incorrect data to its CLAT clients. In the CLAT, should its

database be compromised by attack or polluted by an incorrect DHCPv6 server database, it will route data incorrectly. In both cases, the security of the systems and their databases in an operational matter, not managed by protocol.

However, the operation of the DHCPv6 protocol itself is also required to be correct - the server and its clients must recognize valid requests and reject invalid ones. Therefore, DHCPv6 exchanges MUST be secured as described in [RFC3315].

7. IANA Considerations

We request that IANA allocate two DHCPv6 option codes for use by OPTION_V6_PLATPREFIX_LIST and OPTION_V6_PLATPREFIX from the "Option Codes" table

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.

[RFC3315] Droms, R., Bound, J., Volz, B., Lemon, T., Perkins, C., and M. Carney, "Dynamic Host Configuration Protocol for IPv6 (DHCPv6)", RFC 3315, July 2003.

[RFC6877] Mawatari, M., Kawashima, M., and C. Byrne, "464XLAT: Combination of Stateful and Stateless Translation", RFC 6877, April 2013.

8.2. Informative References

[RFC1918] Rekhter, Y., Moskowitz, R., Karrenberg, D., Groot, G., and E. Lear, "Address Allocation for Private Internets", BCP 5, RFC 1918, February 1996.

[RFC6052] Bao, C., Huitema, C., Bagnulo, M., Boucadair, M., and X. Li, "IPv6 Addressing of IPv4/IPv6 Translators", RFC 6052, October 2010.

[RFC6144] Baker, F., Li, X., Bao, C., and K. Yin, "Framework for IPv4/IPv6 Translation", RFC 6144, April 2011.

[RFC6145] Li, X., Bao, C., and F. Baker, "IP/ICMP Translation Algorithm", RFC 6145, April 2011.

[RFC6146] Bagnulo, M., Matthews, P., and I. van Beijnum, "Stateful NAT64: Network Address and Protocol Translation from IPv6 Clients to IPv4 Servers", RFC 6146, April 2011.

[RFC6147] Bagnulo, M., Sullivan, A., Matthews, P., and I. van Beijnum, "DNS64: DNS Extensions for Network Address Translation from IPv6 Clients to IPv4 Servers", RFC 6147, April 2011.

[RFC7050] Savolainen, T., Korhonen, J., and D. Wing, "Discovery of the IPv6 Prefix Used for IPv6 Address Synthesis", RFC 7050, November 2013.

Authors' Addresses

Yong Cui
Tsinghua University
Beijing 100084
P.R.China

Phone: +86-10-6260-3059
Email: yong@csnet1.cs.tsinghua.edu.cn

Lishan Li
Tsinghua University
Beijing 100084
P.R.China

Phone: +86-15201441862
Email: lilishan9248@126.com

Cong Liu
Tsinghua University
Beijing 100084
P.R.China

Phone: +86-10-6278-5822
Email: gnocuil@gmail.com

Jianping Wu
Tsinghua University
Beijing 100084
P.R.China

Phone: +86-10-6278-5983
Email: jianping@cernet.edu.cn

Fred Baker
Cisco Systems
Santa Barbara, CA 93117
United States

Email: fred@cisco.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/1id-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

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 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

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:

```
+-------------------------------+
|                               |
|        UDP/IP header          |
|                               |
|-------------------------------|
|                               |
|         GUE Header            |
|                               |
|-------------------------------|
|                               |
|      Encapsulated packet      |
|      or control message       |
|                               |
+-------------------------------+
```

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 (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:

```
 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
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|        Source port            |      Destination port         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           Length              |          Checksum             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0x0|C|   Hlen  |  Proto/ctype  |             Flags             |E|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
~                       Fields (optional)                       ~
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                   Extension flags (optional)                  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
~                     Extension fields (optional)               ~
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
~                     Private data (optional)                   ~
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

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 sixteens 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:

```
Private_length = (Hlen * 4) - Length(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. Sever 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.

```
+---------------+                       +---------------+
|               |                       |               |
|   Server 1    |                       |    Server 2   |
|               |                       |               |
+---------------+                       +---------------+
       |                                        ^
       V                                        |
+---------------+   +---------------+   +---------------+
|               |   |               |   |               |
| Encapsulator  |-->|    Layer 3    |-->| Decapsulator  |
|               |   |    Network    |   |               |
+---------------+   +---------------+   +---------------+
```

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, IPIP, 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 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 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:

```
+-------------------------------+
|                               |
|        UDP/IP header          |
|                               |
|-------------------------------|
|                               |
|         GUE Header            |
|                               |
|-------------------------------|
|                               |
|     ESP/AH/private security   |
|                               |
|-------------------------------|
|                               |
|       Encapsulated packet     |
|                               |
+-------------------------------+
```

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 an 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

Intarea Working Group
Internet-Draft
Updates: 2784 (if approved)
Intended status: Standards Track
Expires: August 10, 2015

C. Pignataro
Cisco Systems
R. Bonica
Juniper Networks
S. Krishnan
Ericsson

February 06, 2015

IPv6 Support for Generic Routing Encapsulation (GRE)
draft-ietf-intarea-gre-ipv6-02

Abstract

Generic Routing Encapsulation (GRE) can be used to carry any network-layer payload protocol over any network-layer delivery protocol. GRE procedures are specified for IPv4, used as either the payload or delivery protocol. However, GRE procedures are not specified for IPv6.

This document specifies GRE procedures for IPv6, used as either the payload or delivery protocol. It updates the GRE specification, RFC 2784.

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].

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

Generic Routing Encapsulation (GRE) [RFC2784] [RFC2890] can be used to carry any network-layer payload protocol over any network-layer delivery protocol. GRE procedures are specified for IPv4 [RFC0791], used as either the payload or delivery protocol. However, GRE procedures are not specified for IPv6 [RFC2460].

This document specifies GRE procedures for IPv6, used as either the payload or delivery protocol. It updates RFC 2784 [RFC2784]. Like RFC 2784, this specification describes GRE how has been implemented by several vendors.

1.1. Terminology

The following terms are specific to GRE and are taken from [RFC2784]:

- o GRE delivery header - an IPv4 or IPv6 header whose source address represents the GRE ingress node and whose destination address represents the GRE egress node. The GRE delivery header encapsulates a GRE header.

- o GRE header - the GRE protocol header. The GRE header is encapsulated in the GRE delivery header and encapsulates GRE payload.

- o GRE payload - a network layer packet that is encapsulated by the GRE header.

The following terms are specific MTU discovery:

- o path MTU (PMTU) - the minimum MTU of all the links in a path between a source node and a destination node. If the source and destination node are connected through equal cost multipath (ECMP), the PMTU is equal to the minimum link MTU of all links contributing to the multipath.

- o Path MTU Discovery (PMTUD) - A procedure for dynamically discovering the PMTU between two nodes on the Internet. PMTUD procedures for IPv6 are defined in [RFC1981].

2. GRE Header Fields

This document does not change the GRE header format or any behaviors specified by [RFC2784] or [RFC2890].

2.1. Checksum Present

When the delivery protocol is IPv6, the GRE ingress router SHOULD set the Checksum Present field to zero. GRE egress routers MUST accept either a value of zero or one in this field. If the GRE egress router receives a value of one, it MUST use that information to calculate the GRE header length. However, the GRE ingress router is not required to use the checksum to verify packet integrity.

2.2. Protocol Type

The Protocol Type field contains the protocol type of the payload packet. Protocol Types are defined in [ETYPES]. An implementation receiving a packet containing a Protocol Type which is not listed in [ETYPES] SHOULD discard the packet.

3. IPv6 as a GRE Payload

When the GRE payload is IPv6, the Protocol Type field in the GRE header MUST be set to 0x86DD.

3.1. MTU Considerations

The GRE ingress router maintains an estimate of the GRE MTU (GMTU). The GMTU is equal to the PMTU associated with the path between the GRE ingress and the GRE egress, minus the GRE overhead. The GRE overhead is the combined length of the GRE and IP delivery headers.

The GRE ingress router obtains a PMTU estimate using any of the following:

- o System defaults
- o Configuration
- o PMTUD

When the GRE ingress receives an IPv6 payload packet whose length is less than or equal to the GMTU, it can encapsulate and forward the packet without fragmentation of any kind. In this case, the GRE ingress router MUST NOT fragment the payload or delivery packets.

When the GRE ingress receives an IPv6 payload packet whose length is greater than the GMTU, and the GMTU is greater than or equal to 1280 octets, the GRE ingress router MUST:

- o discard the IPv6 payload packet
- o send an ICMPv6 Packet Too Big (PTB) [RFC4443] message to the IPv6 payload packet source. The MTU field in the ICMPv6 PTB message is set to the GMTU.

The GRE ingress router MUST support a configuration option that determines how the GRE ingress behaves when it receives an IPv6 payload packet whose length is greater than the GMTU, and the GMTU is less than 1280 octets. In its default configuration, the GRE ingress router MUST:

- o discard the IPv6 packet
- o send an ICMPv6 Packet Too Big (PTB) [RFC4443] message to the IPv6 packet source. The MTU field in the ICMPv6 PTB message is set to the GMTU.

However, in an alternative configuration, the GRE ingress MAY:

- o encapsulate the entire IPv6 packet in a single GRE header and IP delivery header

- o fragment the delivery header, so that it can be reassembled by the GRE egress

4. IPv6 as a GRE Delivery Protocol

When the GRE delivery protocol is IPv6, the GRE header can immediately follow the GRE delivery header. Alternatively, IPv6 extension headers MAY be inserted between the GRE delivery header and the GRE header.

If the GRE header immediately follows the GRE delivery header, the Next Header field in the IPv6 header of the GRE delivery packet MUST be set to 47. If extension headers are inserted between the GRE delivery header and the GRE header, the Next Header field in the last IPv6 extension header MUST be set to 47.

4.1. MTU Considerations

"IPv6 requires that every link in the Internet have an MTU of 1280 octets or greater. On any link that cannot convey a 1280-octet packet in one piece, link-specific fragmentation and reassembly must be provided at a layer below IPv6" [RFC2460].

IP adjacencies formed by GRE over IPv6 share this requirement. The IP adjacency MUST have an MTU of 1280 octets or greater. This requirement is fulfilled if all permissible paths between the GRE ingress and GRE egress have PMTU greater than the 1280 plus the GRE overhead.

In case all permissible routes between the GRE ingress and GRE egress do not have PMTU greater than 1280 plus the GRE overhead, implementations MUST be capable of fragmenting and reassembling the GRE delivery header, as described in Section 3.1.

5. IANA Considerations

This document makes no request of IANA.

6. Security Considerations

This document adds no additional security risks to GRE, beyond what is specified in [RFC2784]. It also does not provide any additional security for GRE.

7. Acknowledgements

The authors would like to thank Fred Baker, Dino Farinacci, Tom Herbert, Fred Templin, Joe Touch and Andrew Yourtchenko for their thorough review and useful comments.

8. Normative References

[ETYPES] IANA, "ETHER TYPES", 2014, <http://www.iana.org/assignments/ieee-802-numbers/ieee-802-numbers.xhtml#ieee-802-numbers-1>.

[RFC0791] Postel, J., "Internet Protocol", STD 5, RFC 791, September 1981.

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

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

[RFC2460] Deering, S. and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification", RFC 2460, December 1998.

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

[RFC2890] Dommety, G., "Key and Sequence Number Extensions to GRE", RFC 2890, September 2000.

[RFC4443] Conta, A., Deering, S., and M. Gupta, "Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification", RFC 4443, March 2006.

Authors' Addresses

Carlos Pignataro
Cisco Systems
7200-12 Kit Creek Road
Research Triangle Park, North Carolina 27709
USA

Email: cpignata@cisco.com

Ron Bonica
Juniper Networks
2251 Corporate Park Drive
Herndon, Virginia
USA

Email: rbonica@juniper.net

Suresh Krishnan
Ericsson
8400 Decarie Blvd.
Town of Mount Royal, QC
Canada

Phone: +1 514 345 7900 x42871
Email: suresh.krishnan@ericsson.com

Network Working Group
Internet-Draft
Intended status: Informational
Expires: August 31, 2015

F. Templin, Ed.
Boeing Research & Technology
February 27, 2015

IPv6 Path MTU Interactions With Link Adaptation
draft-templin-6man-linkadapt-02.txt

Abstract

IPv6 intentionally deprecates fragmentation by routers in the network. Instead, links with restricting Maximum Transmission Units (MTUs) must either drop each too-large packet and return an ICMPv6 Packet Too Big (PTB) message or perform link-specific fragmentation and reassembly (also known as "link adaptation") at a layer below IPv6. This latter category of links is often performance-challenged to accommodate steady-state link adaptation. This document therefore proposes an update to the base IPv6 specification to better accommodate links that require link-specific adaptation.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

IPv6 intentionally deprecates fragmentation by routers in the network. Instead, links with restricting Maximum Transmission Units (MTUs) must either drop each too-large packet and return an ICMPv6 Packet Too Big (PTB) message or perform link-specific fragmentation and reassembly (also known as "link adaptation") at a layer below IPv6. This latter category of links is often performance-challenged to accommodate steady-state link adaptation. This document therefore proposes an update to the base IPv6 specification to better accommodate links that require link-specific adaptation.

2. Problem Statement

The current "Internet cell size" is effectively 1500 bytes, i.e., the minimum MTU configured by the vast majority of links in the Internet. IPv6 constrains this even further by specifying a minimum link MTU of 1280 bytes [RFC2460]. However, due to operational issues with Path MTU Discovery (PMTUD) [RFC1981] these sizes can often only be accommodated when links with smaller link-layer segment sizes are configured to perform link adaptation.

Unfortunately, link adaptation can present a significant burden to the link endpoints, i.e., especially when the link supports high data rates and/or is located nearer the "middle" of the network instead of nearer the "edge". An alternative therefore is to ask the originating IPv6 node to either reduce the size of the packets it sends or perform host-based fragmentation, in which case reassembly would be performed by the final destination.

In addition to the above considerations, it is becoming more and more evident that PMTUD uncertainties can be encountered even when there are no links in the path that must perform link adaptation. This is due to the fact that the PTB messages required for PMTUD can be lost due to network filters that block ICMPv6 messages [RFC2923][WAND][SIGCOMM]. Originating IPv6 node are therefore advised to take precautions to avoid path MTU related failure modes.

This document updates the IPv6 protocol specification [RFC2460] to better accommodate paths with various MTUs as described in the following sections.

3. Link Adaptation Signaling and Accommodation

Section 5 of [RFC2460] states:

"IPv6 requires that every link in the Internet have an MTU of 1280 octets or greater. On any link that cannot convey a 1280-octet packet in one piece, link-specific fragmentation and reassembly must be provided at a layer below IPv6."

and:

"A node must be able to accept a fragmented packet that, after reassembly, is as large as 1500 octets.".

This document does not propose to change these requirements, but notes that link adaptation can be burdensome for some links to the point that it would be highly desirable to signal the MTU limitation to the IPv6 communication endpoints. In order to accommodate this, when the router at the link ingress performs link adaptation on a packet it should also send an ICMPv6 PTB message back to the original source (subject to rate limiting) with a Next-Hop MTU set to the link adaptation threshold and with Code field set to 1 [RFC4443]. (Note that these PTB messages are advisory in nature and do not necessarily indicate packet loss.)

As a result, the originating IPv6 node may receive this "new kind" of PTB message and should modify its behavior accordingly. This is accomplished by adding a new final paragraph to Section 5 of [RFC2460] as follows:

"In response to an IPv6 packet that is sent to a destination located beyond an IPv6 link that must perform link adaptation, the originating IPv6 node may receive an ICMP Packet Too Big message with Code=1. In that case, the IPv6 node can either reduce the size of subsequent packet it sends or perform IPv6 fragmentation on packets no larger than 1500 bytes by breaking the packet into N roughly

equal-length pieces (where N is minimized and the length of each piece is smaller than the Next-Hop MTU). These fragments will be reassembled by the destination."

3.1. Accommodating Legacy Nodes

Legacy IPv6 nodes observe the current final paragraph of Section 5 of [RFC2460]:

"In response to an IPv6 packet that is sent to an IPv4 destination (i.e., a packet that undergoes translation from IPv6 to IPv4), the originating IPv6 node may receive an ICMP Packet Too Big message reporting a Next-Hop MTU less than 1280. In that case, the IPv6 node is not required to reduce the size of subsequent packets to less than 1280, but must include a Fragment header in those packets so that the IPv6-to-IPv4 translating router can obtain a suitable Identification value to use in resulting IPv4 fragments. Note that this means the payload may have to be reduced to 1232 octets (1280 minus 40 for the IPv6 header and 8 for the Fragment header), and smaller still if additional extension headers are used."

For such legacy nodes, the receipt of a PTB message with a Next-Hop MTU less than 1280 will result in the above behavior regardless of the value in the Code field. As a result, a link ingress node that returns this new kind of PTB message may receive future packets containing a Fragment header with the More Fragments (MF) bit and Offset field set to 0. The link ingress node should process these packets as an indication that the originating IPv6 node is a legacy node, and should not send further PTB messages. Instead, the link ingress node should use the fragment header supplied by the source to fragment the original packet to a size that would avoid link adaptation. These fragments are then reassembled by the final destination.

4. IANA Considerations

There are no IANA considerations for this document.

5. Security Considerations

The security considerations for [RFC2460] apply also to this document.

6. Acknowledgments

This method was inspired through discussion on the IETF v6ops and NANOG mailing lists in the May through July 2012 timeframe. Further

discussion occurred on the Intarea list in the February 2015 timeframe.

7. References

7.1. Normative References

[RFC2460] Deering, S. and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification", RFC 2460, December 1998.

[RFC4443] Conta, A., Deering, S., and M. Gupta, "Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification", RFC 4443, March 2006.

7.2. Informative References

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

[RFC2923] Lahey, K., "TCP Problems with Path MTU Discovery", RFC 2923, September 2000.

[RFC4821] Mathis, M. and J. Heffner, "Packetization Layer Path MTU Discovery", RFC 4821, March 2007.

[SIGCOMM] Luckie, M. and B. Stasiewicz, "Measuring Path MTU Discovery Behavior", November 2010.

[WAND] Luckie, M., Cho, K., and B. Owens, "Inferring and Debugging Path MTU Discovery Failures", October 2005.

Author’s Address

Fred L. Templin (editor)
Boeing Research & Technology
P.O. Box 3707
Seattle, WA 98124
USA

Email: fltemplin@acm.org