Internet Area WG
Internet-Draft
Intended status: Standard track
Expires April 28, 2020

T. Herbert
Quantonium
L. Yong
Independent
O. Zia
Microsoft
October 26, 2019

Generic UDP Encapsulation
draft-ietf-intarea-gue-09

Status of this Memo

Copyright Notice

Abstract

This specification describes Generic UDP Encapsulation (GUE), which is a scheme for using UDP to encapsulate packets of different 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

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 a virtual networking identifier, security data for validating or authenticating the GUE header, congestion control data, etc.

This document does not define any specific GUE extensions. [GUEEXTEN] specifies a set of initial extensions.

1.1. Applicability

GUE is a network encapsulation protocol that encapsulates packets for various IP protocols. Potential use cases include network tunneling, multi-tenant network virtualization, tunneling for mobility, and transport layer encapsulation. GUE is intended for deploying overlay networks in public or private data center environments, as well as providing a general tunneling mechanism usable in the Internet.

GUE is a UDP based encapsulation protocol transported over existing IPv4 and IPv6 networks. Hence, as a UDP based protocol, GUE adheres to the UDP usage guidelines as specified in [RFC8085]. Applicability of these guidelines are dependent on the underlay IP network and the nature of GUE payload protocol (for example TCP/IP or IP/Ethernet). GUE may also be used to create IP tunnels, hence the guidelines in [IPTUN] are applicable.

[RFC8085] outlines two applicability scenarios for UDP applications: (1) general Internet and (2) a traffic-managed controlled environment (TMCE). The requirements of [RFC8085] pertaining to deployment of a UDP encapsulation protocol in these environments are applicable. Section 5 provides the specifics for satisfying requirements of [RFC8085]. It is the responsibility of the operator deploying GUE to ensure that the necessary operational requirements are met for the environment in which GUE is being deployed.

GUE has much of the same applicability and benefits as GRE-in-UDP [RFC8086] that are afforded by UDP encapsulation protocols. GUE

offers the possibility of good performance for load-balancing encapsulated IP traffic in transit networks using existing Equal-Cost Multipath (ECMP) mechanisms that use a hash of the five-tuple of source IP address, destination IP address, UDP/TCP source port, UDP/TCP destination port, and protocol number. Encapsulating packets in UDP enables use of the UDP source port to provide entropy to ECMP hashing. A material difference between GUE and GRE-in-UDP is that the payload of GUE is always an IP protocol whereas the payload in GRE-in-UDP may be a non-IP protocol; this distinction is pertinent in the discussion of congestion considerations (section 5.9) since IP protocols are generally assumed to be congestion controlled.

In addition, GUE enables extending the use of atypical IP protocols (those other than TCP and UDP) across networks that might otherwise filter packets carrying those protocols. GUE may also be used with connection oriented UDP semantics in order to facilitate traversal through stateful firewalls and stateful NAT.

Additional motivation for the GUE protocol is provided in section 6.

1.2. Terminology and acronyms

GUE	Generic UDP Encapsulation
GUE Header	A variable length protocol header that is composed of a primary four byte header and zero or more four byte words of optional header data
GUE packet	A UDP/IP packet that contains a GUE header and GUE payload within the UDP payload
GUE variant	A version of the GUE protocol or an alternate form of a version
Encapsulator	A network node that encapsulates packets in GUE
Decapsulator	A network node that decapsulates and processes packets encapsulated in GUE
Data message	An encapsulated packet in a GUE payload that is addressed to the protocol stack for an associated protocol
Control message	A formatted message in the GUE payload that is implicitly addressed to the decapsulator to monitor or control the state or behavior of a tunnel
Flags	A set of bit flags in the primary GUE header

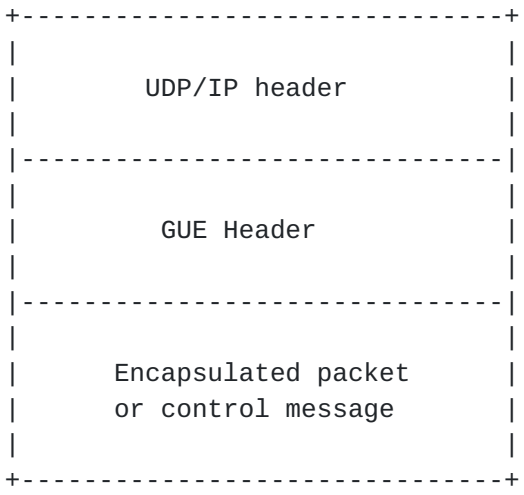
Extension field	An optional field in a GUE header whose presence is indicated by corresponding flag(s)
C-bit	A single bit flag in the primary GUE header that indicates whether the GUE packet contains a control message or data message
Hlen	A field in the primary GUE header that gives the length of the GUE header
Proto/ctype	A field in the GUE header that holds either the IP protocol number for a data message or a type for a control message
Outer IP header	Refers to the outer most IP header or packet when encapsulating a packet over IP
Inner IP header	Refers to an encapsulated IP header when an IP packet is encapsulated
Outer packet	Refers to an encapsulating packet
Inner packet	Refers to a packet that is encapsulated
TMCE	A traffic-managed controlled environment, i.e., an IP network that is traffic-engineered and/or otherwise managed

1.3. 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].

2. Base packet format

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 such as an OAM (Operations, Administration, and Management) message. A GUE packet has the general format:



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

2.1. GUE variant

The first two bits of the GUE header contain the GUE protocol variant number. The variant number can indicate the version of the GUE protocol as well as alternate forms of a version.

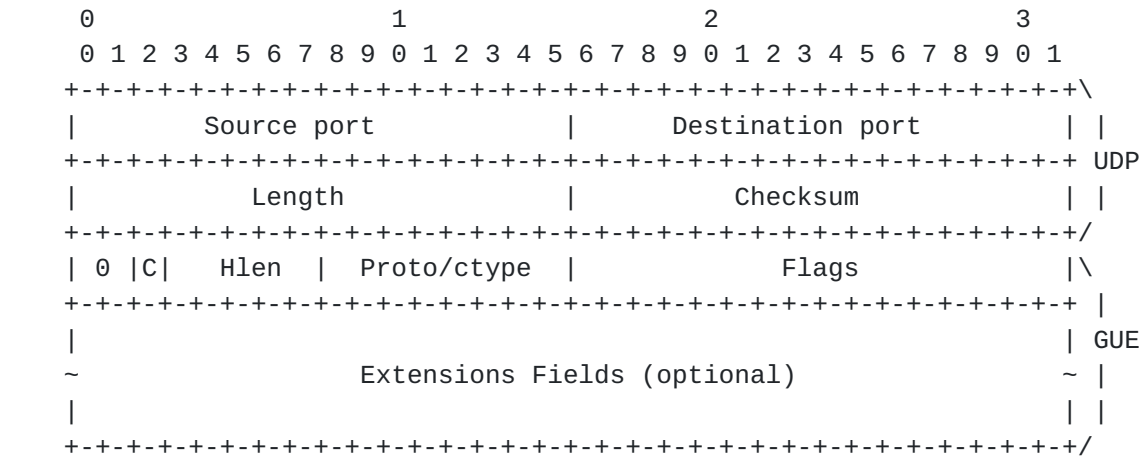
Variants 0 and 1 are described in this specification; variants 2 and 3 are reserved.

3. Variant 0

Variant 0 indicates version 0 of GUE. This variant defines a generic extensible format to encapsulate packets by Internet protocol number.

3.1. Header format

The header format for variant 0 of GUE in UDP is:



The contents of the UDP header are:

- Source port: If connection semantics (section 5.6.1) are applied to an encapsulation, this is set to the local source port for the connection. When connection semantics are not applied, the source port is either set to a flow entropy value, as described in section 5.11, or is set to the GUE assigned port number, 6080.

- Destination port: If connection semantics (section 5.6.1) are applied to an encapsulation, this is set to the destination port for the tuple. If connection semantics are not applied then the destination port is set to the GUE assigned port number, 6080.

- Length: Canonical length of the UDP packet (length of UDP header and payload).

- Checksum: Standard UDP checksum (handling is described in section 5.8).

The GUE header consists of:

- Variant: 0 indicates GUE protocol version 0 with a header.

- C: C-bit: When set indicates a control message. When not set indicates a data message.

- Hlen: Length in 32-bit words of the GUE header, including optional extension fields but not the first four bytes of the header. Computed as (header_len - 4) / 4, where header_len is the total header length in bytes. All GUE headers are a multiple of four bytes in length. Maximum header length is 128 bytes.

- Proto/ctype: When the C-bit is set, this field contains a control message type for the payload (section 3.2.2). When the C-bit is not set, the field holds the Internet protocol number for the encapsulated packet in the payload (section 3.2.1). The control message or encapsulated packet begins at the offset provided by Hlen.

- Flags: Header flags that may be allocated for various purposes and may indicate the presence of extension fields. Undefined header flag bits MUST be set to zero on transmission.

- Extension Fields: Optional fields whose presence is indicated by corresponding flags.

3.2. Proto/ctype field

The proto/ctype fields either contains an Internet protocol number (when the C-bit is not set) or GUE control message type (when the C-bit is set).

3.2.1. Proto field

When the C-bit is not set, the proto/ctype field MUST contain an IANA Internet Protocol Number [IANA-PN]. The protocol number is interpreted relative to the IP protocol that encapsulates the UDP packet (i.e. protocol of the outer IP header). The protocol number serves as an indication of the type of the next protocol header which is contained in the GUE payload at the offset indicated in Hlen.

IP protocol number 59 ("No next header") can be set to indicate that the GUE payload does not begin with the header of an IP protocol. This would be the case, for instance, if the GUE payload were a fragment when performing GUE level fragmentation. The interpretation of the payload is performed through other means such as flags and extension fields, and nodes MUST NOT parse packets based on the IP protocol number in this case.

3.2.2. Ctype field

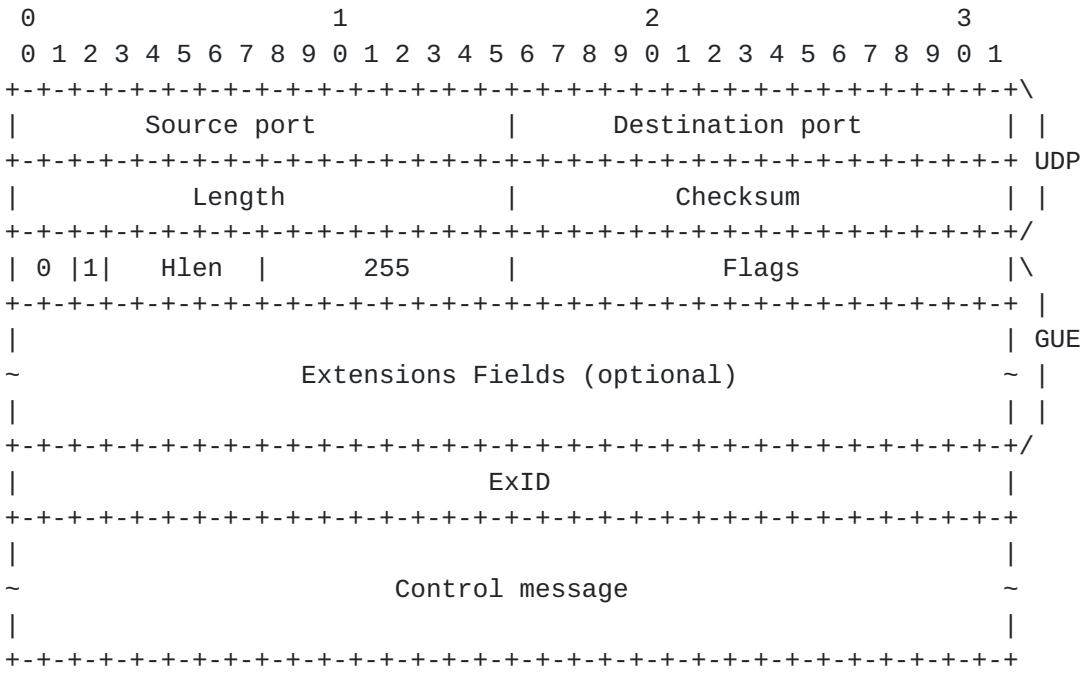
When the C-bit is set, the proto/ctype field MUST be set to a valid control message type. Control messages will be defined in an IANA registry. Type 0 and type 255 are specified in this document, type 1

through 254 are reserved and may be defined in standards.

Type 0 indicates that the GUE payload is a control message, or part of a control message that cannot be correctly parsed or interpreted without additional context. This might be the case when the payload is a fragment of a control message, where only the reassembled packet can be interpreted as a control message.

Type 255 is reserved for experimentation. When this control type is set the first four bytes of the GUE payload (control message) are an experiment identifier (ExId). The ExID is used to differentiate experiments (similar to the experimental identifier defined for TCP options in [RFC6994]). A control message of type 255 MUST include an ExID.

The format of a GUE control message with the experimental control message type is:



Note that the ExID is not part of the GUE header, it is in the payload. In particular, the ExID is not accounted for in the GUE Hlen.

ExIDs are selected at design time, when the protocol designer first implements or specifies the experimental control message. An ExID is thirty-two bits. The value is stored in the header in network-standard (big-endian) byte order.

ExIDs are registered with IANA using "first come, first served"

(FCFS) priority. ExIDs MUST be unique.

3.3. Flags and extension fields

Flags and associated extension fields are the primary mechanism of extensibility in GUE. As mentioned in section 3.1, GUE header flags indicate the presence of optional extension fields in the GUE header. [GUEEXTEN] defines an initial set of GUE extensions.

3.3.1. Requirements

There are sixteen flag bits in the GUE header. Flags may indicate presence of extension fields. The size of an extension field indicated by a flag MUST be fixed in the specification of the flag.

Flags can be grouped together to allow different lengths for an extension field. For example, if two flag bits are grouped, a field can possibly be three different lengths-- that is bit value of 00 indicates no field present; 01, 10, and 11 indicate three possible lengths for the field. Regardless of how flag bits are grouped, the lengths and offsets of extension fields corresponding to a set of flags MUST be well defined and deterministic.

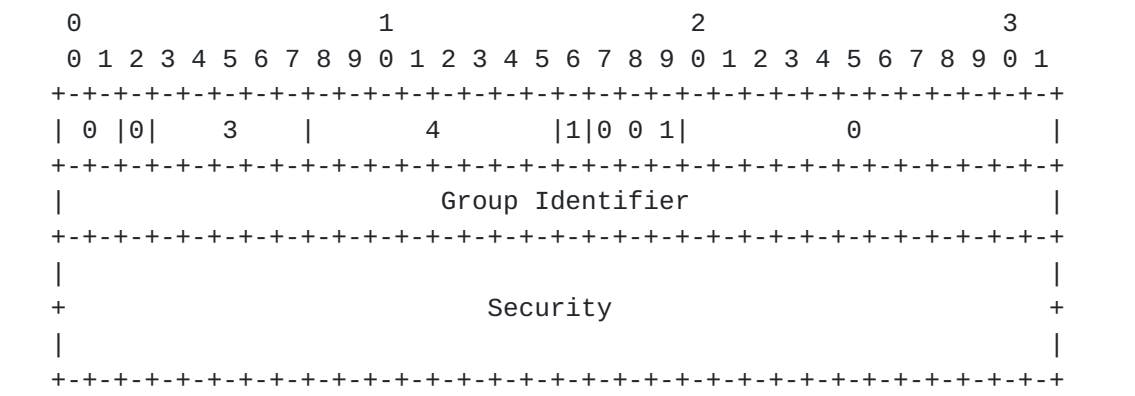
Extension fields are placed in order of the flags. New flags are to be allocated from high to low order bit contiguously without holes. Flags allow random access, for instance to inspect the field corresponding to the Nth flag bit, an implementation only considers the previous N-1 flags to determine the offset. Flags after the Nth flag are not pertinent in calculating the offset of the field for the Nth flag. Random access of flags and fields permits processing of optional extensions in an order that is independent of their position in the packet.

Flags (or grouped flags) are idempotent such that new flags MUST NOT cause reinterpretation of old flags. Also, new flags MUST 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).

The set of available flags can be extended in the future by defining a "flag extensions bit" that refers to a field containing a new set of flags.

3.3.2. Example GUE header with extension fields

An example GUE header for a data message encapsulating an IPv4 packet and containing the Group Identifier and Security extension fields (both defined in [GUEEXTEN]) is shown below:



In the above example, the first flag bit is set which indicates that the Group Identifier extension is present which is a 32 bit field. The second through fourth bits of the flags are grouped flags that indicate the presence of a Security field with seven possible sizes. In this example 001 indicates a sixty-four bit security field.

3.4. Surplus space

The length of a GUE header, as indicated in the GUE Hlen field, may exceed the space consumed by optional extensions in a packet. The space between the end of the last optional field and the end of the header is termed the "surplus space".

Surplus space is reserved per this specification and uses may be defined in future specifications. If a node receives a GUE packet with non-zero length of surplus space then it MUST NOT attempt to interpret the data in the surplus space. For purposes of transforms across the header, such as optional integrity check over the header, the surplus space is considered to be part of the GUE header and would be included in computation.

3.5. Message types

There are two message types in GUE variant 0: control messages and data messages.

3.5.1. Control messages

Control messages carry formatted data that are implicitly addressed to the decapsulator to monitor or control the state or behavior of a tunnel (OAM). For instance, an echo request and corresponding echo reply message can be defined to test for liveness.

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; this ensures that control messages can be created that follow the same path through the network as data messages.

3.5.2. Data messages

Data messages carry encapsulated packets that are addressed to the protocol stack for the associated protocol. Data messages are a primary means of encapsulation and can be used to create tunnels for overlay networks.

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

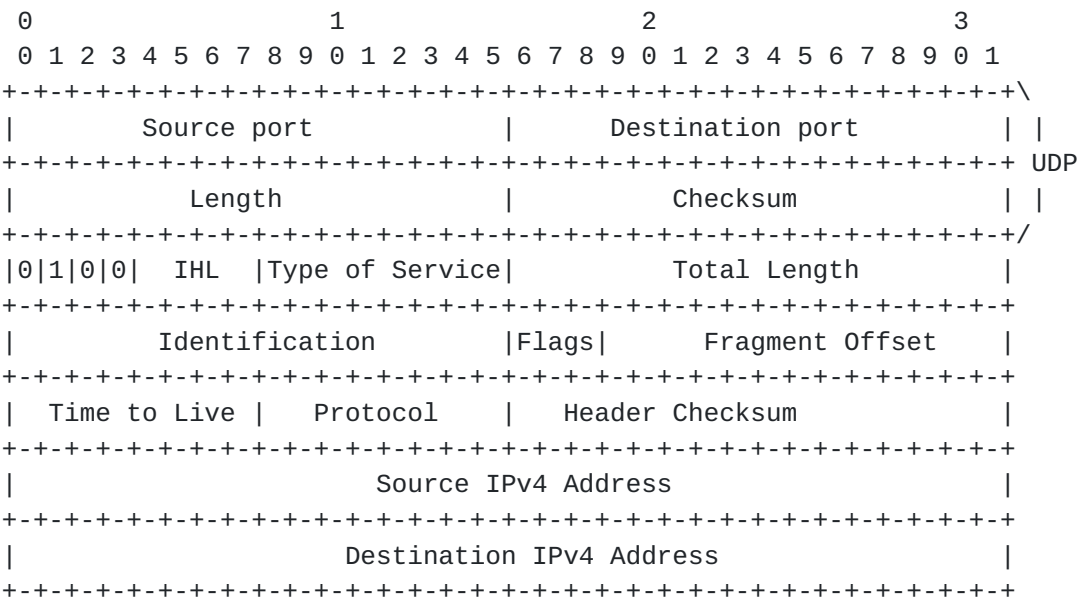
4. Variant 1

Variant 1 of GUE allows direct encapsulation of IPv4 and IPv6 in UDP. In this variant there is no GUE header, a UDP packet carries an IP packet. The first two bits of the UDP payload are the GUE variant field and coincide with the first two bits of the version number in the IP header. The first two version bits of IPv4 and IPv6 are 01, so we use GUE variant 1 for direct IP encapsulation which makes the two bits of GUE variant to also be 01.

This technique is effectively a means to compress out the GUE version 0 header when encapsulating IPv4 or IPv6 packets and there are no flags or extension fields. This method is compatible to use on the same port number as packets with the GUE header (GUE variant 0 packets). This technique saves encapsulation overhead on costly links for the common use of IP encapsulation, and also obviates the need to allocate a separate UDP port number for IP-over-UDP encapsulation.

4.1. Direct encapsulation of IPv4

The format for encapsulating IPv4 directly in UDP is:

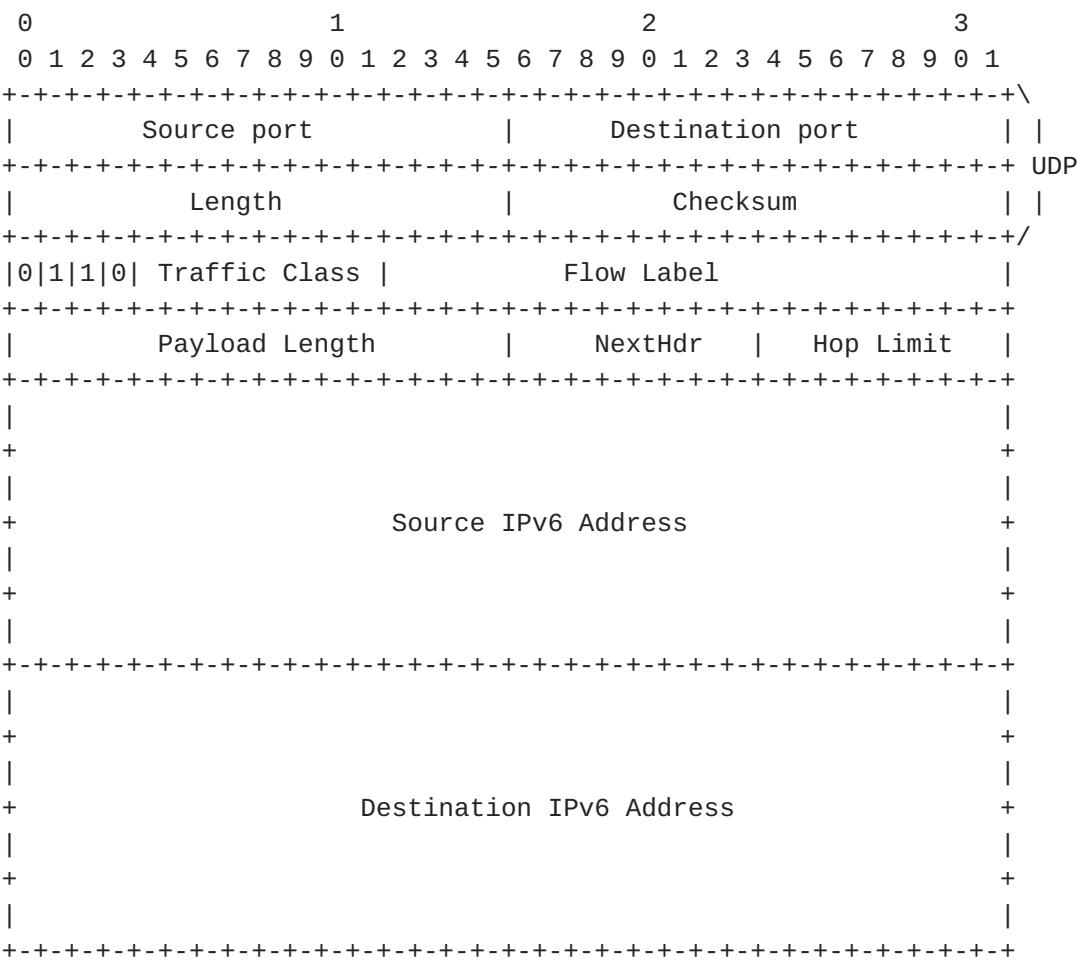


The UDP fields are set in a similar manner as described in section 3.1.

Note that the 0100 value in the first four bits of the UDP payload expresses both the GUE variant as 1 (bits 01) and IP version as 4 (bits 0100).

4.2. Direct encapsulation of IPv6

The format for encapsulating IPv6 directly in UDP is demonstrated below:

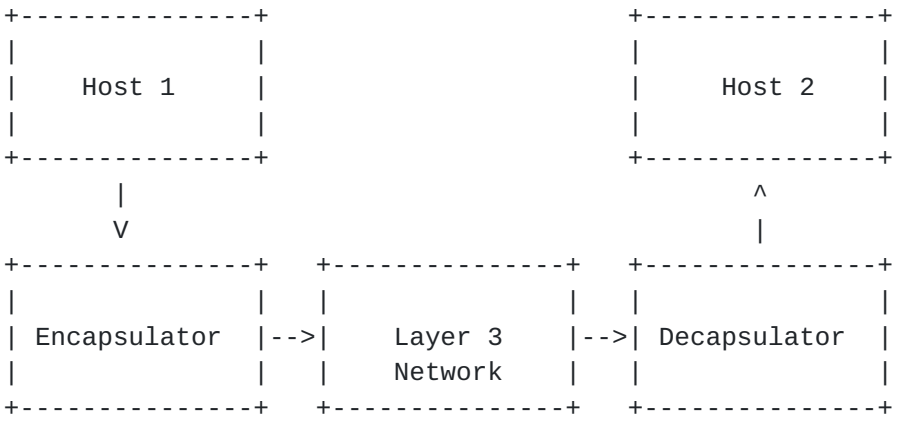


The UDP fields are set in a similar manner as described in section 3.1.

Note that the 0110 value in the first four bits of the the UDP payload expresses both the GUE variant as 1 (bits 01) and IP version as 6 (bits 0110).

5. Operation

The figure below illustrates the use of GUE encapsulation between two hosts. Host 1 is sending packets to Host 2. An encapsulator performs encapsulation of packets from Host 1. These encapsulated packets traverse the network as UDP packets. At the decapsulator, packets are decapsulated and sent on to Host 2. Packet flow in the reverse direction need not be symmetric; for example, the reverse path might not use GUE or any other form of encapsulation.



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

5.1. 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 hosts.

5.2. Transport layer encapsulation

When encapsulating layer 4 packets, the encapsulator and decapsulator should be co-resident with the hosts. 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 encapsulated transport protocol. Note that the transport layer ports in the encapsulated packet are independent of the UDP ports in the outer packet.

5.3. Encapsulator operation

Encapsulators create GUE data messages, set the fields of the UDP header, set flags and optional extension fields in the GUE header, and forward packets to a decapsulator.

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

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 one protocol over another. For instance, if an IP packet is being encapsulated in GUE then diffserv interaction [RFC2983] and ECN propagation for tunnels [RFC6040] SHOULD be followed.

5.4. Decapsulator operation

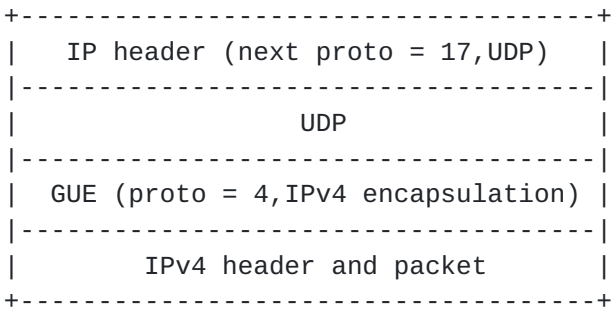
A decapsulator performs decapsulation of GUE packets. A decapsulator is addressed by the outer destination IP address and UDP destination port of a GUE packet. The decapsulator validates packets, including fields of the GUE header.

If a decapsulator receives a GUE packet with an unsupported variant, unknown flag, bad header length (too small for included extension fields), unknown control message type, bad protocol number, an unsupported payload type, or an otherwise malformed header, it MUST drop the packet. Such events MAY be logged subject to configuration and rate limiting of logging messages. Note that set flags in a GUE header 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.

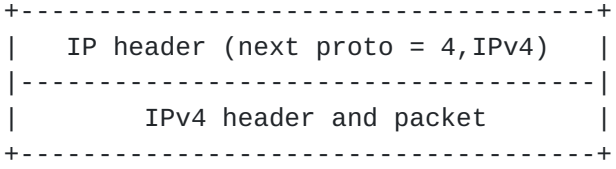
5.4.1. Processing a received data message

If a valid data message is received, the UDP header and GUE header are (logically) removed from the packet. The outer IP header remains intact and the next protocol in the IP header is set to the protocol from the proto field in the GUE header. The resulting packet is then resubmitted into the protocol stack to process the packet as though it was received with the protocol indicated in the GUE header.

As an example, consider that a data message is received where GUE encapsulates an IPv4 packet using GUE variant 0. In this case proto field in the GUE header is set to 4 for IPv4 encapsulation:



The receiver removes the UDP and GUE headers and sets the next protocol field in the IP packet to 4, which is derived from the GUE proto field. The resultant packet would have the format:



This packet is then resubmitted into the protocol stack to be processed as an IPv4 encapsulated packet.

5.4.2. Processing a received control message

If a valid control message is received, the packet MUST be processed as a control message. The specific processing to be performed depends on the value in the ctype field of the GUE header.

If an experimental control message is received (ctype is 255) then the ExID MUST be processed. The ExID is used to identify the particular experimental control message.

If a receiver does not recognize a control message type, or an experimental identifier in an experimental control message, then the packet MUST be dropped and and error message MAY be logged. If a GUE control message is received with control type 255 and the length of the GUE payload is less than four, the size of the ExId, then the packet MUST be dropped and an error message MAY be logged.

5.5. Middlebox inspection

A middlebox MAY inspect a GUE header. A middlebox MUST NOT modify a GUE header or UDP payload.

To inspect a GUE header, a middlebox needs to identify GUE packets. The obvious method is to match the destination UDP port number to be the GUE port number (i.e. 6080). Per [RFC7605], transport port

numbers only have meaning at the endpoints of communications, so inferring the type of a UDP payload based on port number may be incorrect. Middleboxes MUST NOT take any action that would have harmful side effects if a UDP packet were misinterpreted as being a GUE packet. In particular, a middlebox MUST NOT modify a UDP payload based on inferring the payload type from the port number lest the middlebox could cause silent data corruption.

A middlebox MAY interpret some flags and extension fields of the GUE header for classification purposes, but is not required to understand any of the flags or extension fields in GUE packets. A middlebox MUST NOT drop a GUE packet merely because there are flags unknown to it. Similarly, a middlebox MUST NOT arbitrarily filter packets based on GUE flags or extension fields that are present or not present. The header length in the GUE header allows a middlebox to inspect the payload packet without needing to parse the flags or extension fields.

5.6. 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 router does not normally need to parse a GUE header, and none of the flags or extension fields in the GUE header are expected to affect routing. In cases where the outer five-tuple does not provide sufficient entropy for flow classification, for instance UDP ports are fixed to provide connection semantics (section 5.6.1), then the encapsulated packet MAY be parsed to determine flow entropy.

A router MUST NOT modify a GUE header or payload when forwarding a packet. It MAY encapsulate a GUE packet in another GUE packet, for instance to implement a network tunnel (i.e. by encapsulating an IP packet with a GUE payload in another IP packet as a GUE payload). In this case, the router takes the role of an encapsulator, and the corresponding decapsulator is the logical endpoint of the tunnel. When encapsulating a GUE packet within another GUE packet, there are no provisions to automatically copy flags or fields to the outer GUE header. Each layer of encapsulation is considered independent.

5.6.1. Connection semantics

A middlebox might infer bidirectional connection semantics for a UDP flow. For instance, a stateful firewall might 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 should be configured to

assume connected semantics defined by the UDP five tuple and the use of GUE encapsulation needs to be symmetric between both endpoints. The source port set in the UDP header MUST be the destination port the peer would set for replies. In this case, the UDP source port for a tunnel would be a fixed value and not set to be flow entropy.

The selection of whether to make the UDP source port fixed or set to a flow entropy value for each packet sent SHOULD be configurable for a tunnel. The default MUST be to set the flow entropy value in the UDP source port.

5.6.2. NAT

IP address and port translation can be performed on the UDP/IP headers adhering to the requirements for NAT (Network Address Translation) with UDP [RFC4787]. In the case of stateful NAT, connection semantics MUST be applied to a GUE tunnel as described in section 5.6.1. GUE endpoints MAY also invoke STUN [RFC5389] or ICE [RFC5245] to manage NAT port mappings for encapsulations.

5.7. MTU and fragmentation

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 UDP source port. An operator SHOULD set MTU to account for encapsulation overhead and reduce the likelihood of fragmentation.

Alternative to IP fragmentation, the GUE fragmentation extension can be used. GUE fragmentation is described in [GUEEXTEN].

5.8. UDP Checksum Handling

5.8.1. UDP Checksum with IPv4

For UDP in IPv4, when a non-zero UDP checksum is used, the UDP checksum MUST be processed as specified in [RFC0768] and [RFC1122] for both transmit and receive. The IPv4 header includes a checksum that protects against misdelivery 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. Disabling the use of checksums is a deployment consideration that should take into account the risk and effects of packet corruption.

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 by 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.

5.8.2. UDP Checksum with IPv6

For UDP in IPv6, the UDP checksum MUST be processed as specified in [RFC0768] and [RFC2460] for both transmit and receive.

When UDP is used over IPv6, the UDP checksum is relied upon to protect both the IPv6 and UDP headers from corruption. As such, by default a GUE encapsulator MUST use UDP checksums.

[GUEEXTEN] specifies a GUE checksum option that includes a pseudo header containing the IP addresses. An encapsulator MAY use zero-UDP checksums if it uses the GUE checksum. A non-zero UDP checksum and the GUE checksum SHOULD NOT be used simultaneously in a packet since that would be redundant.

When deployed in a TMCE, a GUE encapsulator MAY be configured to use UDP zero-checksum mode and no GUE checksum if the traffic-managed controlled environment or a set of closely cooperating traffic-managed controlled environments (such as by network operators who have agreed to work together in order to jointly provide specific services) meet at least one of the following conditions:

a. It is known (perhaps through knowledge of equipment types and lower-layer checks) that packet corruption is exceptionally unlikely and where the operator is willing to take the risk of undetected packet corruption.

b. It is judged through observational measurements (perhaps of historic or current traffic flows that use a non-zero checksum) that the level of packet corruption is tolerably low and where the operator is willing to take the risk of undetected packet corruption.

c. Carrying applications that are tolerant of misdelivered or corrupted packets (perhaps through higher-layer checksum,

validation, and retransmission or transmission redundancy) where the operator is willing to rely on the applications using GUE to survive any corrupt packets.

The following requirements apply to encapsulators deployed in a TMCE environment that use UDP zero-checksum mode:

a. Use of the UDP checksum with IPv6 MUST be the default configuration for all communications.

b. The GUE implementation MUST comply with all requirements specified in Section 4 of [RFC6936] and with requirement 1 specified in Section 5 of [RFC6936].

c. A decapsulator SHOULD only allow the use of UDP zero-checksum mode for IPv6 on a single received UDP Destination Port, regardless of the encapsulator. The motivation for this requirement is possible corruption of the UDP Destination Port, which may cause packet delivery to the wrong UDP port. If that other UDP port requires the UDP checksum, the misdelivered packet will be discarded.

d. It is RECOMMENDED that the UDP zero-checksum mode for IPv6 is only enabled for certain selected source addresses. The decapsulator MUST check that the source and destination IPv6 addresses in a received packets are permitted by configuration to use UDP zero-checksum mode and discard any packet for which this check fails.

e. The tunnel encapsulator SHOULD use different IPv6 addresses for each GUE communication (tunnel or transport flow) that uses UDP zero-checksum mode, regardless of the decapsulator, in order to strengthen the decapsulator's check of the IPv6 source address (i.e., the same IPv6 source address SHOULD NOT be used with more than one IPv6 destination address, independent of whether that destination address is a unicast or multicast address). When this is not possible, it is RECOMMENDED to use each source IPv6 address for as few GUE communications that use UDP zero-checksum mode as is feasible.

f. When any middlebox exists on the path of GUE communication, it is RECOMMENDED to use the default mode, i.e., use UDP checksum, to reduce the chance that the encapsulated packets will be dropped.

g. Any middlebox that allows the UDP zero-checksum mode for IPv6 MUST comply with requirements 1 and 8-10 in Section 5 of [RFC6936].

h. Measures SHOULD be taken to prevent IPv6 traffic with zero UDP checksums from "escaping" to the general Internet; see Section 5.9 for examples of such measures.

i. IPv6 traffic with zero UDP checksums MUST be actively monitored for errors by the network operator. For example, the operator may monitor Ethernet-layer packet error rates.

j. If a packet with a non-zero checksum is received, the checksum MUST be verified before accepting the packet. This is regardless of whether the tunnel encapsulator and decapsulator have been configured with UDP zero-checksum mode.

The above requirements do not change either the requirements specified in [RFC8200] as modified by [RFC6935] or the requirements specified in [RFC6936].

The requirement to check the source IPv6 address in addition to the destination IPv6 address and the strong recommendation against reuse of source IPv6 addresses among GUE communications collectively provide some mitigation for the absence of UDP checksum coverage of the IPv6 header. A traffic-managed controlled environment that satisfies at least one of three conditions listed at the beginning of this section provides additional assurance.

GUE packets are suitable for transmission over lower layers in the traffic-managed controlled environments that are allowed by the exceptions stated above, and the rate of corruption of the inner IP packet on such networks is not expected to increase by comparison to traffic that is not encapsulated in UDP. For these reasons, GUE does not provide an additional integrity check except when GUE checksum [GUEEXTEN] is used when UDP zero-checksum mode is used with IPv6, and this design is in accordance with requirements 2, 3, and 5 specified in Section 5 of [RFC6936].

Generic UDP Encapsulation does not accumulate incorrect transport-layer state as a consequence of GUE header corruption. A corrupt GUE packet may result in either packet discard or packet forwarding without accumulation of GUE state. Active monitoring of GUE traffic for errors is REQUIRED, as the occurrence of errors will result in some accumulation of error information outside the protocol for operational and management purposes. This design is in accordance with requirement 4 specified in Section 5 of [RFC6936].

The remaining requirements specified in Section 5 of [RFC6936] are not applicable to GUE. Requirements 6 and 7 do not apply because GUE does not include a control feedback mechanism. Requirements 8-10 are middlebox requirements that do not apply to GUE tunnel endpoints.

(See Section 5.5 for further middlebox discussion.)

In summary, a TMCE GUE tunnel is allowed to use UDP zero- checksum mode for IPv6 when the conditions and requirements stated above are met. Otherwise, the UDP checksum needs to be used for IPv6 as specified in [RFC768] and [RFC8200]. Use of GUE checksum is RECOMMENDED when the UDP checksum is not used.

5.9. Congestion Considerations

This section describes congestion considerations for GUE tunnels (Layer 2 and Layer 3 encapsulation) and transport layer encapsulation (Layer 4 protocol over GUE).

5.9.1. GUE tunnels

Section 3.1.9 of [RFC8085] discusses the congestion considerations for design and use of UDP tunnels; this is important because other flows could share the path with one or more UDP tunnels, necessitating congestion control [RFC2914] to avoid destructive interference.

Congestion has potential impacts both on the rest of the network containing a UDP tunnel and on the traffic flows using the UDP tunnels. These impacts depend upon what sort of traffic is carried over the tunnel, as well as the path of the tunnel. The GUE protocol does not provide any congestion control and GUE UDP packets are regular UDP packets. Therefore, a GUE tunnel MUST NOT be deployed to carry non-congestion-controlled traffic over the Internet [RFC8085].

Within a TMCE network, GUE tunnels are appropriate for carrying traffic that is not known to be congestion controlled. For example, a GUE tunnel may be used to carry Multiprotocol Label Switching (MPLS) traffic such as pseudowires or VPNs where specific bandwidth guarantees are provided to each pseudowire or VPN. In such cases, operators of TMCE networks avoid congestion by careful provisioning of their networks, rate-limiting of user data traffic, and traffic engineering according to path capacity.

When a GUE tunnel carries traffic that is not known to be congestion controlled in a TMCE network, the tunnel MUST be deployed entirely within that network, and measures SHOULD be taken to prevent the GUE traffic from "escaping" the network to the general Internet. Examples of such measures are:

- physical or logical isolation of the links carrying GUE from the general Internet,

- deployment of packet filters that block the UDP ports assigned for GUE, and
- imposition of restrictions on GUE traffic by software tools used to set up GUE tunnels between specific end systems (as might be used within a single data center) or by tunnel ingress nodes for tunnels that don't terminate at end systems.

5.9.2 Transport layer encapsulation

If GUE encapsulates a transport layer protocol, such as TCP, it is expected that the transport layer or application layer properly implements congestion control or avoidance. In the case that UDP is encapsulated, the application is expected to provide congestion control as specified in [RFC8085].

5.10. Multicast

GUE packets can be multicast to decapsulators using a multicast destination address in the outer IP header. Each receiving host will decapsulate the packet independently following normal decapsulator operations. The receiving decapsulators need to agree on the same set of GUE parameters and properties; how such an agreement is reached is outside the scope of this document.

GUE allows encapsulation of unicast, broadcast, or multicast traffic. Flow entropy (the value in the UDP source port) can be generated from the header of encapsulated unicast or broadcast/multicast packets at an encapsulator. The mapping mechanism between the encapsulated multicast traffic and the multicast capability in the IP network is transparent and independent of the encapsulation and is otherwise outside the scope of this document.

5.11. Flow entropy for ECMP

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 of the outer headers.

5.11.1. Flow classification

When a packet is encapsulated with GUE and connection semantics are not applied, the source port in the outer UDP packet is set to a flow entropy value that corresponds to 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.

Examples of deriving flow entropy for encapsulation are:

- If the encapsulated packet is a layer 4 packet, TCP/IPv4 for instance, the flow entropy could be based on the canonical five-tuple hash of the inner packet.

- If the encapsulated packet is an AH transport mode packet with TCP as next header, the flow entropy could be a hash over a three-tuple: TCP protocol and TCP ports of the encapsulated packet.

- If a node is encrypting a packet using ESP tunnel mode and GUE encapsulation, the flow entropy could be based on the contents of the clear-text packet. For instance, a canonical five-tuple hash for a TCP/IP packet could be used.

[RFC6438] discusses methods to compute and set flow entropy value for IPv6 flow labels, such methods can also be used to create flow entropy values for GUE.

5.11.2. Flow entropy properties

The flow entropy is the value set in the UDP source port of a GUE packet. Flow entropy in the UDP source port SHOULD adhere to the following properties:

- The value set in the source port is within the ephemeral port range (49152 to 65535 [RFC6335]). Since the high order two bits of the port are set to one, this provides fourteen bits of entropy for the value.

- The flow entropy has a uniform distribution across encapsulated flows.

- An encapsulator MAY occasionally change the flow entropy used for an inner flow per its discretion (for security, route selection, etc). To avoid thrashing or flapping the value, the flow entropy used for a flow SHOULD NOT change more than once every thirty seconds (or a configurable value).

- Decapsulators, or any networking devices, SHOULD NOT attempt to interpret flow entropy as anything more than an opaque value. Neither should they attempt to reproduce the hash calculation used by an encapasulator in creating a flow entropy value. They MAY use the value to match further receive packets for steering decisions, but MUST NOT assume that the hash uniquely or permanently identifies a flow.

- Input to the flow entropy calculation is not restricted to ports and addresses; input could include the 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.

- The assignment function for flow entropy SHOULD be randomly seeded to mitigate denial of service attacks. The seed SHOULD be changed periodically.

5.12. Negotiation of acceptable flags and extension fields

An encapsulator and decapsulator need to achieve agreement about GUE parameters that will be used in communications. Parameters include supported GUE variants, flags and extension fields that can be used, security algorithms and keys, supported protocols and control messages, etc. This document proposes different general methods to accomplish this, however the details of implementing these are considered out of scope.

General methods for this are:

- Configuration. The parameters used for a tunnel are configured at each endpoint.

- Negotiation. A tunnel negotiation can be performed. This could be accomplished in-band of GUE using control messages.

- Via a control plane. Parameters for communicating with a tunnel endpoint can be set in a control plane protocol (such as that needed for network virtualization).

- Via security negotiation. Use of security typically implies a key exchange between endpoints. Other GUE parameters may be conveyed as part of that process.

6. Motivation for GUE

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

6.1. Benefits of GUE

* GUE is a generic encapsulation protocol. GUE can encapsulate protocols that are represented by an IP protocol number. This includes layer 2, layer 3, and layer 4 protocols.

* GUE is an extensible encapsulation protocol. Standardized optional data such as security, virtual networking identifiers,

fragmentation are defined.

* For extensibility, GUE uses flag fields as opposed to TLVs as some other encapsulation protocols do. Flag fields are strictly ordered, allow random access, and are efficient in use of header space.

* GUE allows sending of control messages such as OAM using the same GUE header format (for routing purposes) as normal data messages.

* GUE maximizes deliverability of non-UDP and non-TCP protocols.

* GUE provides a means for exposing per flow entropy for ECMP for IP atypical protocols such as SCTP, DCCP, ESP, etc.

6.2. Comparison of GUE to other encapsulations

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 [RFC7637] 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 [RFC7348], LISP [RFC6830] which encapsulates layer 3 packets, MPLS/UDP [RFC7510], GENEVE [GENEVE], and GRE-in-UDP Encapsulation [RFC8086].

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 for flow entropy as input to ECMP are defined in section 5.11.

o GUE permits encapsulation of arbitrary IP protocols, which includes layer 2, 3, and 4 protocols.

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).

- GUE is extensible. New flags and extension fields can be defined.

- 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.

- GUE includes both data messages (encapsulation of packets) and control messages (such as OAM).

- The flags-field model facilitates efficient implementation of extensibility in hardware. For instance, a TCAM can be used to parse a known set of N flags where the number of entries in the TCAM is 2^N. By comparison, the number of TCAM entries needed to parse a set of N arbitrarily ordered TLVs is approximately e*N!.

- GUE includes a variant that encapsulates IPv4 and IPv6 packets directly within UDP.

7. Security Considerations

There are two important considerations of security with respect to GUE.

- Authentication and integrity of the GUE header.
- Authentication, integrity, and confidentiality of the GUE payload.

GUE security is provided by extensions for security defined in [GUEEXTEN]. These extensions include methods to authenticate the GUE header and encrypt the GUE payload.

The GUE header can be authenticated using a security extension for an HMAC (Hashed Message Authentication Code). Securing the GUE payload can be accomplished by use of the GUE Payload Transform extension. This extension allows the use of DTLS (Datagram Transport Layer Security) to encrypt and authenticate the GUE payload.

A hash function for computing flow entropy (section 5.11) SHOULD be randomly seeded to mitigate some possible denial service attacks.

8. IANA Considerations

8.1. UDP source port

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

```
Service Name: gue
Transport Protocol(s): UDP
Assignee: Tom Herbert <tom@herbertland.com>
Contact: Tom Herbert <tom@herbertland.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
```

8.2. GUE variant number

IANA is requested to set up a registry for the GUE variant number. The GUE variant number is two bits containing four possible values. This document defines variants 0 and 1. New values are assigned in accordance with RFC Required policy [RFC5226].

Variant number	Description	Reference
0	GUE Version 0 with header	This document
1	GUE Version 0 with direct IP encapsulation	This document
2..3	Unassigned	

8.3. Control types

IANA is requested to set up a registry for the GUE control types. Control types are 8 bit values. New values for control types 1-127 are assigned in accordance with RFC Required policy [RFC5226].

Control type	Description	Reference
0	Control payload needs more context for interpretation	This document
1..254	Unassigned	
255	Experimental	This document

8.4 Control Type Experimental Identifiers

IANA is requested to create a "GUE Control Type Experimental Identifiers (GUE Control ExIDs)" registry. The registry records 32-bit ExIDs, as well as a reference (description, document pointer, assignee name, and e-mail contact) for each entry.

Entries are assigned on a First Come, First Served (FCFS) basis [RFC5226]. The registry operates FCFS on the entire ExID (in network-

standard order).

IANA will advise applicants of duplicate entries to select an alternate value, as per typical FCFS processing.

IANA will record known duplicate uses to assist the community in both debugging assigned uses as well as correcting unauthorized duplicate uses.

IANA should impose no requirements on making a registration other than indicating the desired codepoint and providing a point of contact. A short description or acronym for the use is desired but should not be required.

Initial assignments are:

ExI D	Description	Reference
1..x0fffffff	Unassigned	

9. Acknowledgements

The authors would like to thank David Liu, Erik Nordmark, Fred Templin, Adrian Farrel, Bob Briscoe, Murray Kucherawy, Mirja Kuhlewind, David Black, Joe Touch, and Greg Mirsky for valuable input on this draft. Special thanks to Fred Templin who is serving as document shepherd.

10. References

10.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC0768] Postel, J., "User Datagram Protocol", STD 6, RFC 768, DOI 10.17487/RFC0768, August 1980, <http://www.rfc-editor.org/info/rfc768>.

[RFC8085] Eggert, L., Fairhurst, G., and G. Shepherd, "UDP Usage Guidelines", BCP 145, RFC 8085, DOI 10.17487/RFC8085, March 2017, <https://www.rfc-editor.org/info/rfc8085>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC2983] Black, D., "Differentiated Services and Tunnels", RFC 2983, DOI 10.17487/RFC2983, October 2000, <http://www.rfc-editor.org/info/rfc2983>.

[RFC6040] Briscoe, B., "Tunnelling of Explicit Congestion Notification", RFC 6040, DOI 10.17487/RFC6040, November 2010, <http://www.rfc-editor.org/info/rfc6040>.

[RFC6935] Eubanks, M., Chimento, P., and M. Westerlund, "IPv6 and UDP Checksums for Tunneled Packets", RFC 6935, DOI 10.17487/RFC6935, April 2013, <http://www.rfc-editor.org/info/rfc6935>.

[RFC6936] Fairhurst, G. and M. Westerlund, "Applicability Statement for the Use of IPv6 UDP Datagrams with Zero Checksums", RFC 6936, DOI 10.17487/RFC6936, April 2013, <http://www.rfc-editor.org/info/rfc6936>.

[RFC1122] Braden, R., Ed., "Requirements for Internet Hosts - Communication Layers", STD 3, RFC 1122, DOI 10.17487/RFC1122, October 1989, <http://www.rfc-editor.org/info/rfc1122>.

[RFC4459] Savola, P., "MTU and Fragmentation Issues with In-the-Network Tunneling", RFC 4459, DOI 10.17487/RFC4459, April

2006, <http://www.rfc-editor.org/info/rfc4459>.

[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, DOI 10.17487/RFC6335, August 2011, <https://www.rfc-editor.org/info/rfc6335>.

[RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", RFC 5226, DOI 10.17487/RFC5226, May 2008, <https://www.rfc-editor.org/info/rfc5226>.

10.2. Informative References

[RFC6994] Touch, J., "Shared Use of Experimental TCP Options", RFC 6994, DOI 10.17487/RFC6994, August 2013, <https://www.rfc-editor.org/info/rfc6994>.

[RFC8086] Yong, L., Ed., Crabbe, E., Xu, X., and T. Herbert, "GRE-in-UDP Encapsulation", RFC 8086, DOI 10.17487/RFC8086, March 2017, <http://www.rfc-editor.org/info/rfc8086>.

[RFC7605] Touch, J., "Recommendations on Using Assigned Transport Port Numbers", BCP 165, RFC 7605, DOI 10.17487/RFC7605, August 2015, <https://www.rfc-editor.org/info/rfc7605>.

[RFC4787] Audet, F., Ed., and C. Jennings, "Network Address Translation (NAT) Behavioral Requirements for Unicast UDP", BCP 127, RFC 4787, DOI 10.17487/RFC4787, January 2007, <http://www.rfc-editor.org/info/rfc4787>.

[RFC5389] Rosenberg, J., Mahy, R., Matthews, P., and D. Wing, "Session Traversal Utilities for NAT (STUN)", RFC 5389, DOI 10.17487/RFC5389, October 2008, <http://www.rfc-editor.org/info/rfc5389>.

[RFC5245] Rosenberg, J., "Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols", RFC 5245, DOI 10.17487/RFC5245, April 2010, <http://www.rfc-editor.org/info/rfc5245>.

[RFC8084] Fairhurst, G., "Network Transport Circuit Breakers", BCP 208, RFC 8084, DOI 10.17487/RFC8084, March 2017, <https://www.rfc-editor.org/info/rfc8084>.

[RFC6438] Carpenter, B. and S. Amante, "Using the IPv6 Flow Label for Equal Cost Multipath Routing and Link Aggregation in Tunnels", RFC 6438, DOI 10.17487/RFC6438, November 2011, <http://www.rfc-editor.org/info/rfc6438>.

[RFC3378] Housley, R. and S. Hollenbeck, "EtherIP: Tunneling Ethernet Frames in IP Datagrams", RFC 3378, DOI 10.17487/RFC3378, September 2002, <http://www.rfc-editor.org/info/rfc3378>.

[RFC2784] Farinacci, D., Li, T., Hanks, S., Meyer, D., and P. Traina, "Generic Routing Encapsulation (GRE)", RFC 2784, DOI 10.17487/RFC2784, March 2000, <http://www.rfc-editor.org/info/rfc2784>.

[RFC4023] Worster, T., Rekhter, Y., and E. Rosen, Ed., "Encapsulating MPLS in IP or Generic Routing Encapsulation (GRE)", RFC 4023, DOI 10.17487/RFC4023, March 2005, <http://www.rfc-editor.org/info/rfc4023>.

[RFC2661] Townsley, W., Valencia, A., Rubens, A., Pall, G., Zorn, G., and B. Palter, "Layer Two Tunneling Protocol "L2TP"", RFC 2661, DOI 10.17487/RFC2661, August 1999, <http://www.rfc-editor.org/info/rfc2661>.

[RFC7637] Garg, P., Ed., and Y. Wang, Ed., "NVGRE: Network Virtualization Using Generic Routing Encapsulation", RFC 7637, DOI 10.17487/RFC7637, September 2015, <https://www.rfc-editor.org/info/rfc7637>.

[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>.

[RFC2003] Perkins, C., "IP Encapsulation within IP", RFC 2003, DOI 10.17487/RFC2003, October 1996, <http://www.rfc-editor.org/info/rfc2003>.

[RFC2473] Conta, A. and S. Deering, "Generic Packet Tunneling in IPv6 Specification", RFC 2473, DOI 10.17487/RFC2473, December 1998, <https://www.rfc-editor.org/info/rfc2473>.

[RFC3948] Huttunen, A., Swander, B., Volpe, V., DiBurro, L., and M. Stenberg, "UDP Encapsulation of IPsec ESP Packets", RFC 3948, DOI 10.17487/RFC3948, January 2005, <http://www.rfc-

editor.org/info/rfc3948>.

[RFC6830] Farinacci, D., Fuller, V., Meyer, D., and D. Lewis, "The Locator/ID Separation Protocol (LISP)", RFC 6830, DOI 10.17487/RFC6830, January 2013, <http://www.rfc-editor.org/info/rfc6830>.

[RFC7510] Xu, X., Sheth, N., Yong, L., Callon, R., and D. Black, "Encapsulating MPLS in UDP", RFC 7510, DOI 10.17487/RFC7510, April 2015, <http://www.rfc-editor.org/info/rfc7510>.

[GUEEXTEN] Herbert, T., Yong, L., and Templin, F., "Extensions for Generic UDP Encapsulation", draft-ietf-intarea-gue-extensions-06

[IPTUN] Touch, J. and Townsley, M., "IP Tunnels in the Internet Architecture", draft-ietf-intarea-tunnels-10

[IANA-PN] IANA, "Protocol Numbers", <https://www.iana.org/assignments/protocol-numbers>.

[TCPUDP] Chesire, S., Graessley, J., and McGuire, R., "Encapsulation of TCP and other Transport Protocols over UDP", draft-cheshire-tcp-over-udp-00

[GENEVE] Gross, J., Ed., Ganga, I. Ed., and Sridhar, T., "Geneve: Generic Network Virtualization Encapsulation", draft-ietf-nvo3-geneve-10

[UDPENCAP] Herbert, T., "UDP Encapsulation in Linux", <http://people.netfilter.org/pablo/netdev0.1/papers/UDP-Encapsulation-in-Linux.pdf>

[MULTIQ] Herbert, T. and de Bruijn, W., "Scaling in the Linux Networking Stack", <https://www.kernel.org/doc/Documentation/networking/scaling.txt>

[CSUMOFF] Cree, E., "Checksum Offloads in the Linux Networking Stack", <https://www.kernel.org/doc/Documentation/networking/checksum-offloads.txt>

[SEGOFF] Duyck, A., "Segmentation Offloads in the Linux Networking Stack", <https://www.kernel.org/doc/Documentation/networking/segmentation-offloads.txt>

Appendix A: NIC processing for GUE

This appendix is informational and does not constitute a normative part of this document.

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. An overview of UDP based encapsulation and acceleration is in [UDPENCAP]

A.1. Receive multi-queue

Contemporary NICs support multiple receive descriptor queues (multi-queue) [MUTLIQ]. Multi-queue enables load balancing of network processing for a NIC across multiple CPUs. On packet reception, a NIC selects an appropriate queue for host processing. Receive Side Scaling (RSS) 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 is compatible with multi-queue NICs that support five-tuple hash calculation for UDP/IP packets as input to RSS. The flow entropy in the UDP 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 is section 5.7, fragmentation of GUE packets is mostly avoided by fragmenting packets before entering a tunnel, GUE fragmentation, path MTU discovery in higher layer protocols, or operator adjusting MTUs. Other UDP traffic might not implement such procedures to avoid fragmentation, so enabling UDP RSS support in the NIC might be a considered tradeoff during configuration.

A.2. Checksum offload

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

A.2.1. Transmit checksum offload

NICs can provide a protocol agnostic method to offload the 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 a 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.

If an encapsulator is co-resident with a host, then checksum offload may be performed using remote checksum offload (RCO)[GUEEXTEN]. 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.

Another alternative when an encapsulator is co-resident with a host is to perform Local Checksum Offload (LCO) [CSUMOFF]. In this method, the inner transport layer checksum is offloaded and the outer UDP checksum can be deduced based on the fact that the portion of the packet covered by the inner transport checksum will sum to zero or at least the bitwise "not" of the inner pseudo header.

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 checksums, 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 bitwise "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) [SEGOFF] 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 of 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 extension fields do not contain values that need to be updated on a per segment basis. For example, extension 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) [SEGOFF] is a NIC feature where received 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: Implementation considerations

This appendix is informational and does not constitute a normative part of this document.

B.1. Priveleged ports

Using the source port to contain a flow entropy 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.

B.2. Setting flow entropy as a route selector

An encapsulator generating flow entropy in the UDP source port could 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 flow entropy. For instance, a host can store a flow hash in its protocol control block (PCB) for an inner flow, and might alter the value upon detecting that packets are traversing a lossy path. Changing the flow entropy for a flow SHOULD be subject to hysteresis (at most once every thirty seconds) to limit the number of out of order packets.

B.3. Hardware protocol implementation considerations

Low level data path protocols, such as GUE, are often supported in high speed network device hardware. Variable length header (VLH) protocols like GUE are sometimes 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 the 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, the needs and requirements of a protocol may change which may manifest themselves as new parameterizations to be supported in the fast path. To allow this extensibility, a device practicing constrained flexibility should allow fast path parameterizations to be programmable.

Authors' Addresses

Tom Herbert
Quantonium
4701 Patrick Henry
Santa Clara, CA 95054
US

Email: tom@herbertland.com

Lucy Yong
Independent
Austin, TX
US

Email: lucy_yong@yahoo.com

Osama Zia
Microsoft
1 Microsoft Way
Redmond, WA 98029
US

Email: osamaz@microsoft.com