Network Working Group	J. Manner	
Internet-Draft	N. Varis	
Intended status: Experimental	TKK	
Expires: March 18, 2010	September 14, 2009	

TOC

Generic UDP Tunnelling (GUT)
draft-manner-tsvwg-gut-00.txt

Status of this Memo

Copyright Notice

Abstract

Deploying new transport protocols on the Internet is a well-known problem, as NATs and firewall drop packets with new protocol types. Tunnelling over UDP is one way to make IP packets hide the actual payload and enable end-to-end delivery. This draft proposes a simple UDP tunnelling encapsulation and end-host operation to enable new IP

payloads, e.g., new transport protocols, to be deployed on the Internet.

Table of Contents

---

1. Terminology

TOC

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

---

2. Introduction

TOC

New transport layer technology, such as SCTP [RFC4960] (Stewart, R., “Stream Control Transmission Protocol,” September 2007.) and DCCP [RFC4340] (Kohler, E., Handley, M., and S. Floyd, “Datagram Congestion Control Protocol (DCCP),” March 2006.), have well-known problems with deployment on the Internet. Firewalls drop IP packets with unknown (too new) transport protocol types, and NAT boxes do not know how to translate these protocols.
Tunnelling over UDP has often been mentioned as a means to traverse middleboxes. Mostly the solutions are ad-hoc and protocol-specific. In order to make deployment of UDP tunnelling at least somewhat

consistent, this draft proposes a simple mechanism to realise the goal. The benefit is that with a generic solution we avoid the need to define tunneling specifications for each transport protocol.
IP-in-IP encapsulation is also one potential solution. However, if the communicating hosts are behind NATs, they have private source addresses within the inner IP headers, which will break any communication. Moreover, if NATs and firewalls probe deeper into the packet, they will encounter potentially an unknown transport protocol and drop the packet.
The basic idea of GUT is to encapsulate the original transport protocol and its payload (in general the whole IP payload) within a UDP packet destined to the well-known port GUT_P. Between the outer UDP header and the inner transport header, we have a magic number and original protocol type. The purpose of the magic number is to enable the recipient to distinguish between GUT traffic and sporadic packets sent to the GUT_P port, e.g., due to port scanning, and to reconstruct the original IP packet with the correct IP protocol type. The protocol does not require back-and-forth signalling, it just encapsulates the original transport protocol and its payload - to any middlebox on the way this looks like a normal UDP flow to port GUT_P.
If the inner transport protocol has a handshake or any back-and-forth messaging, these are run automatically within the UDP-tunnel created by GUT: GUT is meant to be fully transparent to the inner transport protocol. Note that GUT can also tunnel protocol types which do not have any port informations, such as RSVP or ICMP. The GUT encapsulation is agnostic to the IP protocol version being used (IPv4 or IPv6).

---

3. Basic operation

TOC

The basic idea of the protocol is to encapsulate the transport protocol header and possible payload within a UDP header and send the packet to a well-known UDP port GUT_P. The receiver will get the UDP packets, check the magic number, and if it matches the expected well-known value, reconstruct the original IP packet, and forward it for further processing within the OS stack. Figure 1 (GUT encapsulation) shows the encapsulation.

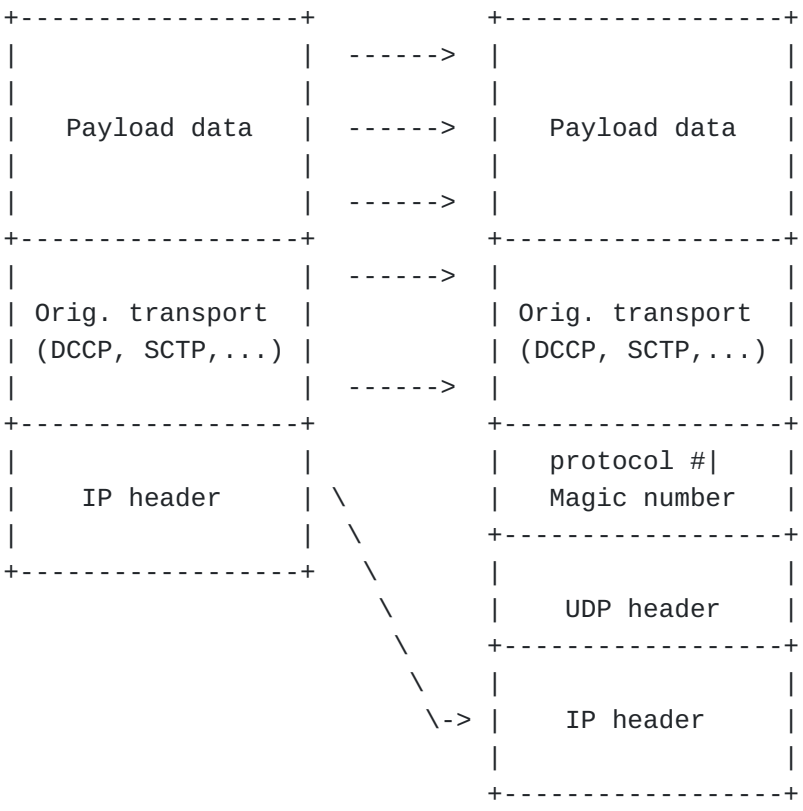

Figure 1: GUT encapsulation

The magic number MAGIC_N is a 32-bit value allocated by IANA. After the magic number we have 24-bits reserved for future use, and the original encapsulated 8-bit protocol number. All in all, this header is thus 64 bits.
The 24-bit reserved field is currently unused, but we may need to use some of it for fragmentation and/or for use with IP options, as discussed below.

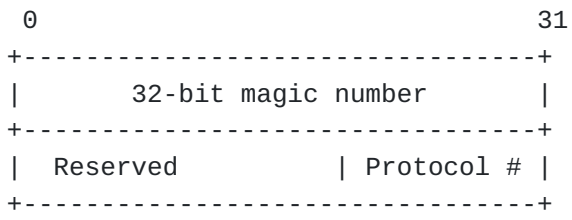

Figure 2: GUT header

Discussion:

*Basically, we could drop the GUT header and just encapsulate the original IP payload into a UDP datagram. However, this results in two challenges at the receiver: (1) we need to do pattern matching or some data analysis to figure out what the original IP payload (e.g. transport protocol) was, and (2) we end up building IP packets from all the traffic arriving at the GUT_P UDP port.

3.1. Sender operation

TOC

A GUT sender operates basically as any data sender. It receives data (from transport protocol Y going to port X) and sends it out to the GUT_P port over UDP. The source port MAY be chosen freely, although if the encapsulated protocol had a notion of port numners, the sender MAY choose the same source port. The IP header indicates a UDP transport, the GUT header is the first bytes of the UDP payload and gives the inner protocol number. The IP header length obviously gives the length of the whole GUT packet including the encapsulated transport protocol packet.
The current value of GUT_P is 4887 (rule of thumb 1-800-GUTP)
Discussion:

*Fragmentation issues. GUT adds 16 octets of headers (UDP+GUT) which may cause fragmentation to happen. We could do fragmentation at the IP layer or within GUT by using the bits in the GUT header to indicate the offset. We have bits unused in the GUT header and could use them to implement fragmentation within GUT; the question is, is IP fragmentation a problem with firewall and NAT traversal?

*IP options are a bit problematic. We could hide the IP options within the GUT encapsulation, thus they would be forwarded unoticed within the network, between the sender and receiver. The could also copy or move them to the outer header and make them visible in the network when the encapsulated packet is routed. However, this may result in unwanted behavior. For example, if we have a RAO option in the original IP packet, and we keep this visible in the GUT-encapsulated datagram, any node on the path that wants to check the IP payload after the RAO option will encounter a UDP header and a GUT header, which the node most probably will not recognize. The IP options to be left visible between the two GUT nodes must be decided case-by-case.

3.2. Receiver operation

TOC

Receiving GUT encapsulated traffic is done through normal transport player receive mechanisms. GUT must be able to receive packets with two distinct destination ports, GUT_P and the original source port. The former is when the node receiving a packet is the flow destination, i.e., it will receive packets to the GUT_P port as indicated above. The latter case happens for a 2-way flow and the node is the flow source, i.e., it will receive upstream packets to the initial source port it chose when sending the very first packet of the flow.
When the host receives packets to port GUT_P, i.e., it is the destination of the flow, it MUST store the source IP, encapsulated protocol number and any port numbers. This state information is needed to send back packets belonging to the same flow - it is not strictly needed, e.g., if the flow is unidirectional, but since GUT may not know this, storing the state is needed.
On receiving a packet to the GUT_P UDP port, the GUT process MUST first check the magic number. If this matches, the host can continue processing, otherwise, it MUST discard the packet silently.
After decapsulation of the 64-bit GUT header, the GUT processing reconstructs the original IP packet by using the included protocol number, and injects the resulting packet into the host stack for further processing. The packet may now be subject to host firewall rules. If there are no listening sockets for the encapsulated protocol Y, the host packet processing takes care of this event. So essentially, GUT operates as a transparent encapsulation (well, sort of, we still receive packets for the GUT_P port which obviously is not "transparent").
Since the encapsulated payload may have had a different IP header at the source, and thus a different transport header checksum, on building the new IP packet, the checksum field of the original header (if any) must be recomputed. The IP header may differ (original vs. received),

for example, because the sender was behind a NAT, or the receiver was behind NAT with port forwarding enabled.
Discussion:

*Fragmentation and reassembly. This will be determined once we fix whether framentation would be an IP layer or GUT function.

*Handling IP options: TBD once the final solution is determined.

3.3. Example with one NAT in between

TOC

The following figure describes how various protocol fields are mapped on a two-way signaling session. The example shows a DCCP-transfer going from A to B. The figure presents the content of IP packets as they are sent out from a component on the path. Note that if the encapsulated protocol does not have port numbers, the GUT processing is even simpler.

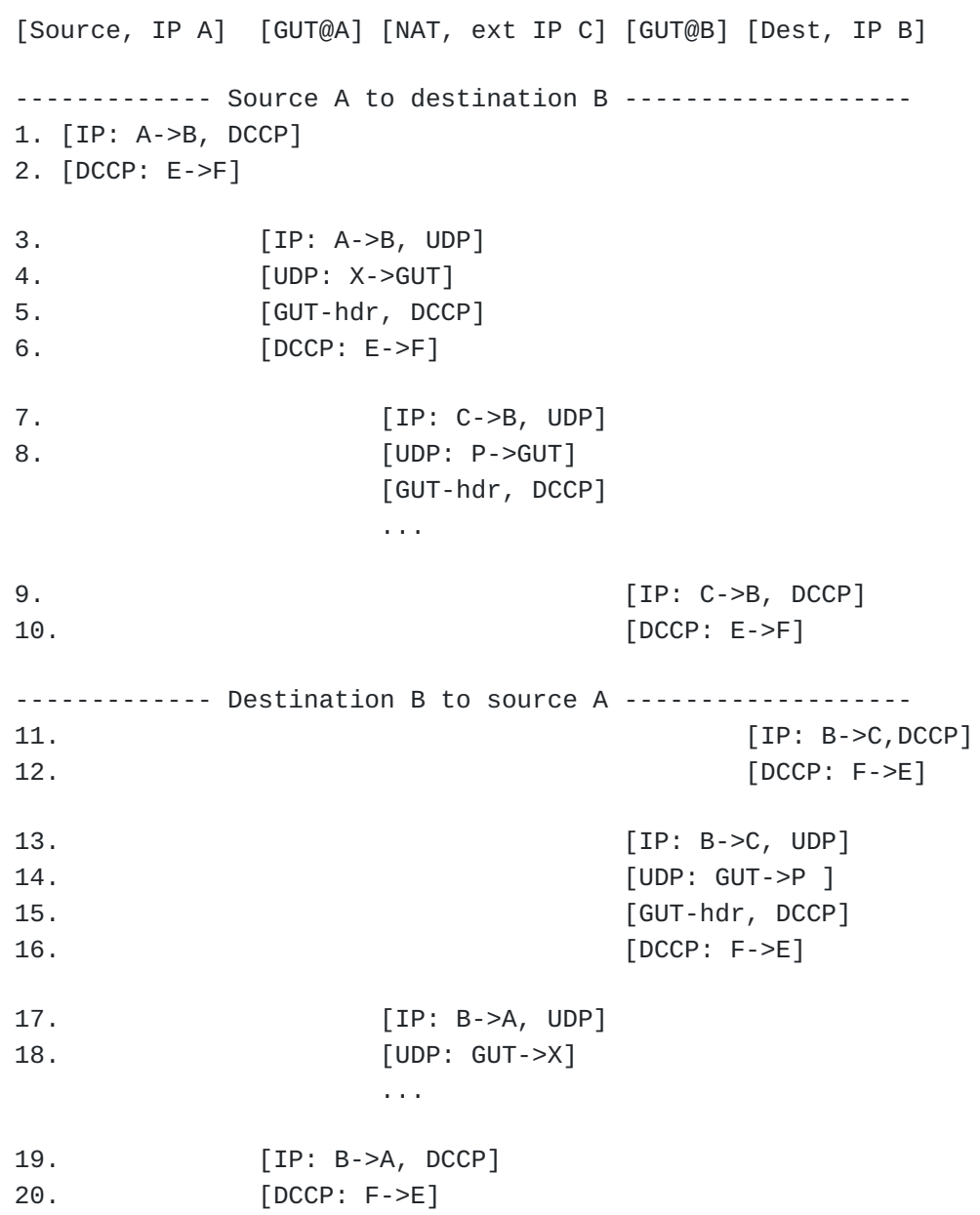

Figure 3: GUT encapsulation example

A few details from the figure above:

*Line 4: the GUT process takes GUT_P as the destination port, and chooses a source port, either randomly or a fixed port called "X" in the figure.

*Line 8: the NAT may choose a new source port P, instead of X, and rewrite the UDP header.

*Line 10: before sending the packet out, the GUT process takes note of the source IP and port numbers, and the encapsulated protocol.

*Line 11-12: the tunneled protocol has not seen the GUT encapsulation, thus, it will use the encapsulated port numbers in the reverse traffic.

*- Lines 13-16: the GUT process has earlier stored state about the flow, knows now that the packet is for an existing stream, and can direct the flow to the right destination port "P", instead of sending it to GUT_P, as if the packet belonged to a new stream.

---

4. Deployment Considerations

TOC

The basic goal of GUT is to look like generic UDP messaging to any middlebox on the path. If the inner transport protocol has support for congestion control, GUT encapsulated packets that are lost will trigger the inner transport to react.
As GUT only encapsulates the original transport header, any ECN [RFC3168] (Ramakrishnan, K., Floyd, S., and D. Black, “The Addition of Explicit Congestion Notification (ECN) to IP,” September 2001.) marking are kept. Specifically, if the inner transport protocol has support for ECN, and the receiver wants to send congestion notifications to the sender, this information is encoded into the inner transport header and carried intact all the way back to the sender. The GUT end-points have to note ECN and operate as follows:

1. Sender: If the outgoing transport protocol wants to indicate it supports ECN, this information MUST be kept intact in GUT processing.

2. Receiver: If the IP header has the ECN CE codepoint, this MUST be propagated to the inner transport protocol stack. (If the receiver wants to send ECN congestion notifications back to the sender, it uses its own mechanism to do that, inside GUT.)

In general, GUT does not carry any ECN information by itself, it works as a transparent layer between the inner transport protocol and the IP layer. How the codepoint information is propagated by and through GUT is an implementation issue.
As GUT-encapsulated traffic looks like an ordinary stream of UDP packets, existing NAT traversal protocols and techniques work out of the box. For example, a receiving GUT-daemon can, when needed, maintain the GUT_P open at the NAT using any suitable NAT-traversal protocol.

GUT was originally designed to be used for host-to-host communication. Yet, nothing actually prohibits to have a network node that takes the IP packets coming from a host, and tunnels them through GUT. Similarly, a network node on the receiving side of the connection can decapsulate the packets before they actually hit the receiving end-host, so essentially making a GUT-proxy service.
There is yet one issue to consider, namely when to encapsulate a transport protocol in GUT, and when not. This can be done automatically, e.g., when replies to a transport protocol Y's connection initiation are not received. Using GUT can also be a configuration parameter, say, e.g., the host always encapsulates DCCP packets into GUT; this operation is fully transparent to the inner transport protocol.

---

5. Encapsulation of other protocols

TOC

GUT is originally designed to counter the problems of deploying relatively new transport protocols on existing Internet. Yet, GUT can also be used to encapsulate any other protocol, e.g., RSVP or HIP. Note that some protocols may not involve port numbers, e.g., RSVP. In such cases, GUT is free to choose a random port for the sender's port number; the receiver's port is always GUT_P.
TBA: more discussion on other encapsulation?

---

6. Security Considerations

TOC

Using GUT opens up a trivial DoS attack: the host can be bombarded with UDP packets to GUT_P with a valid magic number. The host can diminish this case by closing the GUT_P listening socket (and NAT binding) when there are no listening sockets open that require GUT; GUT is only active when an application is running, expecting to receive data.
The use of GUT must not bypass the host's internal firewall rules, i.e., if a packet it received through GUT, after GUT processing, the packet MUST be forward through the firewall rule chain as if it came directly from the network. GUT must operate transparently to most of the host software.
GUT itself does not employ any security functions for content protection. Yet, one could use any one-way mechanism, or purely rely on the security functions of the inner payload. If security measures are used on GUT, it should be a one-way scheme, which does not rely on back-and-forth signalling; we don't want to force two-way signaling within GUT, this may or may not happen due to the inner protocol being tunneled.

GUT enables hosts to payloads through firewalls that would otherwise we dropped. Thus, it enables by-passing firewall rules, which the network admin may not appreciate. However, it would be trivial to block also GUT, by disabling traffic to port GUT_P. Obviously one could run GUT over any UDP port, and thus force a strict firewall to look for the magic number in the UDP payload. However, how to block GUT properly and completely is out of scope of this specification.

---

7. IANA Considerations

TOC

This document requests IANA to allocate two values:
1. A new UDP port number GUT_P as referred to in the document.
2. A 56-bit "magic number" to be used in filtering actual GUT packets.
TBA: a discussion on what the value of this magic number should be. We probably should not just take any random value but choose it such that there would be a very small probability that it is something often used in a UDP-based transport protocol. Choosing a good value may involve some statistical analysis of current UDP traffic.

---

8. Summary

TOC

Essentially this draft proposes to define a generic mechanism for tunneling any IP payload over a UDP tunnel. The concrete steps to be specified are:
1. Allocate a well-known port number for end-hosts to send UDP-encapsulated traffic to. This is important because the sender would need to know what port a receiver has open for GUT traffic. Also, firewall administrators may want to choose if they allow UDP tunneling to happen.
2. Define the encapsulation and decapsulation procedure so that the receiver knows how to rebuild the original IP packet.
3. Define the fragmentation and handling of IP options in a unified way.
The benefits are:

1. Existing IP protocols, with or without port information, work without changes. Yet, if they employ IP options, we need to make this possible.

2. Deployment can be done on the end-host or a network proxy.

3. No changes are required for existing NAT and firewall devices.

9. References

TOC

9.1. Normative References

TOC

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

9.2. Informative References

TOC

[RFC3168]	Ramakrishnan, K., Floyd, S., and D. Black, "The Addition of Explicit Congestion Notification (ECN) to IP," RFC 3168, September 2001 (TXT).
[RFC4340]	Kohler, E., Handley, M., and S. Floyd, "Datagram Congestion Control Protocol (DCCP)," RFC 4340, March 2006 (TXT).
[RFC4960]	Stewart, R., "Stream Control Transmission Protocol," RFC 4960, September 2007 (TXT).

Authors' Addresses

TOC

	Jukka Manner
	Helsinki University of Technology (TKK)
	P.O. Box 3000
	Espoo FIN-02015 TKK
	Finland
Phone:	+358 9 451 2481
Email:	jukka.manner@tkk.fi
URI:	http://www.netlab.tkk.fi/~jmanner/
	Nuutti Varis
	Helsinki University of Technology (TKK)
	P.O. Box 3000
	Espoo FIN-02015 TKK
	Finland
Email:	nvaris@cc.hut.fi