Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: January 1, 2014

S. Cheshire
J. Graessley
R. McGuire
Apple

June 30, 2013

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

Abstract

Encapsulation of TCP and other transport protocols over UDP enables use of UDP-based NAT traversal techniques with other transport protocols.

1. Conventions and Terminology Used in this Document

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in "Key words for use in RFCs to Indicate Requirement Levels" [RFC2119].

2. Introduction

To establish direct communication between two devices that are both behind NAT gateways, Interactive Connectivity Establishment (ICE) [RFC5245] is used to create the necessary mappings in both NAT gateways. While, in principle, ICE should work for both TCP and UDP, recent work has shown that in practice success rates are higher using UDP (about 80% for UDP, compared to 60% for TCP) [RFC5128].

However, many applications want flow control, congestion control, reliability, and other properties provided by TCP. Hence it would be desirable to encapsulate TCP over UDP, to provide the transport protocol capabilities provided by TCP, combined with the NAT-traversal capability available with UDP.

Using ICE [RFC5245] entails sending and receiving STUN [RFC5389] packets. Therefore it is necessary for the encapsulation format to support STUN packets and encapsulated TCP packets sharing the same UDP port.

This document defines a suitable encapsulation of TCP (and other transport protocols) over UDP.

We anticipate in-kernel implementations of TCP-over-UDP, making use of the kernel's existing mature TCP code, but user-level implementations of TCP-over-UDP are also possible, using a high-quality user-space TCP implementation that provides the necessary congestion control and other desirable aspects of TCP. This allows applications to use TCP-over-UDP on operating systems that don't provide TCP-over-UDP.

The performance and congestion control properties of TCP-over-UDP are exactly the same as traditional TCP. TCP-over-UDP is traditional TCP using UDP/IP as the datagram transport, instead of just raw IP as the datagram transport. Existing TCP facilities such as window scaling, timestamps, selective ack, and TCP header options are supported, as they are with native TCP. In fact, TCP options are expected to work more reliably with TCP-over-UDP, because middleboxes will be less able to easily interfere with such options, modifying them, stripping them, or dropping packets containing TCP options, as they often do

today with native TCP packets. In particular, Multipath TCP-over-UDP is expected to work more reliably than native Multipath TCP [RFC6824], because middleboxes that interfere with use of those TCP options will be less able to do that when the packets are encapsulated inside UDP.

Any protocol than can be run over native TCP, including TLS, can be run over TCP-over-UDP.

NAT gateways typically use shorter timeouts for UDP port mappings than they do for TCP port mappings. This means that long-lived TCP-over-UDP connections will need to send more frequent keepalive packets than native TCP connections. For this reason, native TCP connections are still preferable for long-lived mostly-idle connections. For these connections, TCP-over-UDP should be used only when native TCP fails.

3. Conceptual API

While the protocol specified in this document could be implemented in a variety of ways, it is helpful to describe one possible API model to illustrate the intended functionality. In this illustrative API, the client application first creates an "attachable" UDP socket, and then creates an "attached" TCP socket which shares its UDP port. All TCP packets sent and received by the "attached" TCP socket are encapsulated inside UDP packets.

Note that the TCP socket conceptually has no associated source port of its own. The UDP port numbers provide all the necessary traffic demultiplexing, and fully identify the software endpoint to which a given UDP packet is directed. No further demultiplexing at the TCP level is required. Equivalently, the TCP source port could be thought of as being "UDP port X". Note that TCP using "UDP port X" as its source port is not that same as a native TCP connection using "TCP port X" as its source port. For example, a host with a TCP-over-UDP socket listening for TCP-over-UDP connections to UDP port 80 will often also have a native TCP socket listening for native TCP connections to TCP port 80.

4. Packet Format

The most-significant four bits of the first octet of the UDP payload determine whether the payload is:

- 0x0-0x3: A raw UDP payload (typically a STUN packet)
- 0x5-0xF: An encapsulated TCP packet
- 0x4: Some other transport protocol (e.g., SCTP, DCCP, or even UDP)

These three packet varieties are described in more detail below.

4.1. Raw UDP

When the client makes an API call to transmit a UDP payload on an "attachable" UDP socket, where the most-significant four bits of the first octet of the payload are in the range 0x0-0x3 (as is the case for a STUN [RFC5389] packet, where the most-significant two bits are always zero) the entire UDP payload is sent-as is, with no modification.

Upon reception of a UDP packet where the most-significant four bits of the first octet are in the range 0x0-0x3, the entire payload is delivered to the application's UDP socket without modification.

This allows a client application to exchange STUN packets with an unmodified STUN server that knows nothing about this new encapsulation.

4.2. Encapsulated TCP

When the client makes an API call to transmit TCP data on an "attached" TCP socket, encapsulated TCP packets are generated and sent.

For clarity of explanation, this section describes the process of generating these packets in terms of (i) first generating a standard TCP packet in the conventional way, and then (ii) performing a rewriting step to transform it into a TCP-over-UDP packet just prior to transmission. Upon reception, the inverse rewrite is performed to transform it back into a conventional TCP packet, which is then handed to the TCP stack for the usual TCP processing. In this model the only required change to an existing in-kernal TCP implementation is that its per-connection data structures need to include an additional one-bit flag signifying whether this is a native TCP connection or a TCP-over-UDP connection. This is necessary to allow TCP port X and TCP-over-UDP port X to coexist simultaneously.

It is likely that, for better efficiency, implementers may choose to modify their TCP code to generate TCP-over-UDP packets directly, rather than first generating a standard TCP header and then rewriting it. Nonetheless, for clarity, the description which follows assumes that a standard TCP packet has been generated, and describes how such a packet would be transformed into a TCP-over-UDP packet.

In the IP header, the IP protocol field is changed from 0x06 (TCP) to 0x11 (UDP).

The TCP header [RFC0793] is then rewritten as described below to transform it into a legal UDP header [RFC0768]. A 20-octet (or more) TCP header is formatted as shown below:

```
    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        |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                        Sequence Number                        |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                    Acknowledgment Number                      |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |  Data |           |U|A|P|R|S|F|                               |
   | Offset| Reserved  |R|C|S|S|Y|I|            Window             |
   |       |           |G|K|H|T|N|N|                               |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |           Checksum            |         Urgent Pointer        |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                      (Optional) Options                       |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 1: TCP Header Format

This header is rewritten into the encapsulated TCP-over-UDP format shown below:

```
    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            |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |  Data |           | |A|P|R|S|F|                               |
   | Offset| Reserved  |0|C|S|S|Y|I|            Window             |
   |       |           | |K|H|T|N|N|                               |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                        Sequence Number                        |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                    Acknowledgment Number                      |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                      (Optional) Options                       |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 2: Encapsulated TCP-over-UDP Header Format

The specified TCP source port is replaced by the UDP socket's source port. If the implementation generates the TCP header using the UDP port number, then this is a no-op.

The specified destination port is preserved. Note that for the packet to be interpreted correctly upon reception, the receiving peer must (obviously) implement TCP-over-UDP and have it enabled for the receiving UDP socket.

The length is the customary UDP length field, indicating the number of octets from the start of this header to the end of the payload. It can be computed from the Total Length and Internet Header Length fields in the IP header.

The Checksum is the customary UDP Checksum. Note that the checksum does not have to be recomputed by brute-force; it can be derived using a simple calculation involving the original TCP Checksum and the fields modified in the course of this header rewrite.

The header up to this point is now a standard UDP header.

The remainder of the TCP header is re-ordered so that the "Data Offset" line comes next. Since the minimum legal value for Data Offset is 5, this yields a UDP payload where the most-significant four bits of the first octet are necessarily in the range 0x5-0xF.

The Sequence Number and Acknowledgment Number appear next.

The TCP Checksum is omitted, since it is redundant. The UDP header has its own checksum.

The TCP Urgent Pointer field is omitted. TCP-over-UDP does not support urgent data. The TCP URG flag MUST NOT be set.

This in-place rewrite converts the 20-octet (or more) TCP header into a 20-octet (or more) TCP-over-UDP header. Since the header size is the same, the TCP MSS is unchanged.

Upon reception of a UDP packet where the most-significant four bits of the first octet are in the range 0x5-0xF, on a UDP port with TCP-over-UDP enabled, the code performs the inverse of the transformation described above, and then hands the resulting TCP packet to the existing TCP implementation for further processing.

4.3. Encapsulated UDP and Other Transport Protocols

When the client makes an API call to transmit a UDP payload where the most-significant four bits of the first octet are not in the range 0x0-0x3, an explicit UDP-in-UDP encapsulation is used. A four-octet header is inserted before the UDP payload:

```
    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
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |     0x40      | proto = 0x11  |     0x00      |     0x00      |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 3: Encapsulated UDP-over-UDP Header Format

Upon reception of a UDP packet where the most-significant four bits of the first octet have the value 0x4, on a UDP port with TCP-over-UDP enabled, this signifies an encapsulated transport protocol (other than TCP). The value in the second octet indicates the encapsulated protocol.

The details of how a given transport protocol is encapsulated over UDP are defined on a per-protocol basis. In particular, the complete transport protocol SHOULD NOT be included in its entirety, since some of the fields are redundant or unnecessary (as illustrated above for TCP). For protocols that use 16-bit port numbers, these port number fields SHOULD be omitted from the encapsulated header, since the necessary demultiplexing function is performed by the UDP header's port number fields.

In the case of UDP, none of the UDP header fields are replicated in the encapsulated content, since the outer UDP header contains all the necessary information to infer the effective inner UDP header contents (i.e. the source and destination ports are the same, the length field of the effective inner UDP header is four octets less than the outer UDP header's length field, and the checksum is recomputed). Upon reception of such a packet, the four-octet encapsulation header is stripped off, and the remaining payload delivered to the application. For UDP packets where the most-significant four bits of the first octet are not in the range 0x0-0x3, this results in an effective MTU reduction of four octets. This is not expected to cause any significant problems. The primary use of TCP-over-UDP is expected to be for STUN and TCP sharing a UDP port.

5. IANA Considerations

No IANA actions are required by this document.

6. Security Considerations

No new security risks occur as a result of using this protocol.

7. References

7.1. Normative References

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

[RFC0793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, September 1981.

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

7.2. Informative References

[RFC5128] Srisuresh, P., Ford, B., and D. Kegel, "State of Peer-to-Peer (P2P) Communication across Network Address Translators (NATs)", RFC 5128, March 2008.

[RFC5245] Rosenberg, J., "Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols", RFC 5245, April 2010.

[RFC5389] Rosenberg, J., Mahy, R., Matthews, P., and D. Wing, "Session Traversal Utilities for NAT (STUN)", RFC 5389, October 2008.

[RFC6824] Ford, A., Raiciu, C., Handley, M., and O. Bonaventure, "TCP Extensions for Multipath Operation with Multiple Addresses", RFC 6824, January 2013.

Authors' Addresses

Stuart Cheshire
Apple Inc.
1 Infinite Loop
Cupertino, California 95014
USA

Phone: +1 408 974 3207
Email: cheshire@apple.com

Josh Graessley
Apple Inc.
1 Infinite Loop
Cupertino, California 95014
USA

Phone: +1 408 974 5710
Email: jgraessley@apple.com

Rory McGuire
Apple Inc.
1 Infinite Loop
Cupertino, California 95014
USA

Phone: +1 408 862 3633
Email: rlpm@apple.com