Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: April 24, 2014

J. Iyengar
Franklin and Marshall College
S. Cheshire
J. Graessley
Apple

October 21, 2013

Minion - Wire Protocol
draft-iyengar-minion-protocol-02

Abstract

Minion uses TCP-format packets on-the-wire, for compatibility with existing NATs, Firewalls, and similar middleboxes, but provides a richer set of facilities to the application, as described in the Minion Service Model document. This document specifies the details of the on-the-wire protocol used to provide those services.

Status of this Memo

Copyright Notice

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].

This document uses terminology like "kernel" and "user-level", as those terms pertain to many of today's Unix-like operating systems. Equivalent concepts apply to software that is built using a different architectural model than may not include such an obvious kernel/user split.

2. Introduction

Minion uses TCP-format packets on-the-wire, to provide full compatibility with existing NATs, Firewalls, and similar middleboxes, but provides a richer set of facilities to the application, described in the Minion Service Model and Conceptual API document [minserv]. This document specifies the details of the on-the-wire protocol used to provide those services. Before reading this protocol specification document, familiarity with the Minion Service Model [minserv] is strongly recommended. That information is not repeated here.

Minion runs over a standard TCP connection. Therefore, IP addresses and TCP ports are used just as they are with TCP [RFC0793].

Minion is also designed to be able to use a modified TCP connection which supports out-of-order delivery, giving better low-latency performance on lossy networks, for use by the kinds of application that today would use UDP [RFC0768] to achieve low-latency delivery. The goal of providing low-latency delivery -- and consequently the need to be able to handle a data stream that may have gaps -- is reflected in various aspects of the Minion protocol design, such as the use of DTLS instead of TLS, and the use of Consistent Overhead Byte Stuffing [COBS] for reliably extracting messages from an incomplete data stream. Minion is able to take advantage of out-of-order delivery where the network stack offers that, but Minion does not require it. Minion still works correctly when the performance benefits of out-of-order delivery are not available.

Minion supports messages of arbitrary size. Large messages are broken into chunks a little under 16 kilobytes each (the DTLS maximum record size, minus a few bytes for Minion header). At the receiving end the Minion chunks are reassembled into Minion messages and delivered to the client application. Small messages are sent in a single Minion chunk.

Normally messages are sent by the client as a single atomic unit, and delivered to the receiving client as a single atomic unit. For messages too large to fit conveniently in memory, the message may be built incrementally by the sender, and delivered to the receiving client incrementally, a chunk at a time.

When a Minion message is complete, or has at least one maximum Minion chunk size of data accumulated, then if it is eligible to be sent according to the message ordering facilities offered by the Minion Service Model [minserv] (Sender Ordering, Receiver Ordering, and Chaining) a Minion chunk is generated.

Each Minion chunk contains a Minion chunk header followed by the client's message data, as described in Section 3 "Minion Chunk Format".

Each Minion chunk is encrypted using DTLS [RFC6347].

Each encrypted DTLS payload is then framed using RECOBS, as described in Section 4 "Recursively Embeddable COBS", so that it begins with a 00 byte and ends with an FF byte.

The framed, encrypted chunk is then enqueued for transmission.

If the kernel networking code supports multiple priorities, then the framed, encrypted chunk is placed in the transmission queue for the stated priority level. Any time the TCP congestion window and/or receive window rules allow more data to be sent, data is drawn from the highest-priority non-empty transmit buffer, assigned the next block of unused TCP sequence numbers, formed into a TCP segment, and transmitted on the wire. This just-in-time TCP sequencing mechanism has the effect of causing higher-priority data to be inserted right at the front of the conceptual combined transmit buffer, at the earliest possible byte boundary, unconstrained by message or chunk boundaries in the lower-priority messages. This is possible because the RECOBS framing is robust to pre-emption at any arbitrary byte boundary.

Note that, when priorities are supported, chunks above the lowest priority MUST be delivered to the kernel in such a way that they are sent completely before the kernel resumes sending the lower-priority traffic. The RECOBS framing supports interrupting a lower priority stream with a higher-priority chunk, but not alternating back and forth between two priority levels. Once a higher-priority chunk interrupts lower-priority traffic, the higher-priority chunk must be completed before the lower-priority traffic resumes. Typically this is easily achieved by delivering the chunk to the kernel atomically in a single write call.

2.1. Comparison of TCP and UDP NAT Traversal

When connecting to a server with a globally routable address, TCP is generally preferable to UDP. TCP includes the SYN and FIN bits which tell a NAT gateway when a connection starts and ends. In particular, the FIN bit tells the NAT gateway when it can discard state related to that mapping. UDP has no defined connection start/end indicators, which means that unused UDP mappings are much more likely to accumulate, which means that NAT gateways tend to be more aggressive about timing out UDP mappings [Study], which means that clients using UDP need to be more aggressive about sending keepalive traffic, which is bad both for network efficiency and for battery life. Port Control Protocol (PCP) [RFC6887] offers some future hope of alleviating this problem by allowing clients to explicitly negotiate for longer mapping lifetimes, but PCP is not yet widely deployed. In the meantime, if use of UDP increases, NAT gateways are likely to be accumulating mappings even more rapidly, with no way to differentiate which are still required and which may be safely discarded, with the result that UDP mappings may have to be discarded even more aggressively. While a discarded UDP mapping can be recreated by another outgoing UDP packet, in the time between when the UDP mapping is discarded and then recreated, the client is cut off an unable to receive inbound communication from server or peer at the other end. Therefore, we believe that it is preferable to use TCP where possible.

However, when connecting to a peer which is itself also behind a NAT gateway, in the absence of PCP support [RFC6887], techniques like Interactive Connectivity Establishment (ICE) [RFC5245] are used, and research has shown that there are cases where ICE works for UDP but not for TCP [RFC5128].

To accomodate both usage scenarios, Minion is generally used with standard TCP format packets, but for peer-to-peer scenarios where TCP ICE is found not to work, Minion can be used encapsulated inside UDP [TCPoUDP] instead.

3. Minion Chunk Format

A Minion Chunk begins with an eight-byte header, followed by the client's message data:

```
    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
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |C|    Code     |Pri|     This Minion Chunk ID                  |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   | Reserved      |RCP|     Referenced Minion Chunk ID            |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   :                                                               :
   :                      Minion Chunk Data                        :
   :                                                               :
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 1: Minion Chunk Format

If the Complete ('C') bit is zero, this message is incomplete; the receiver should expect to receive additional continuation chunks for this message. If the Complete bit is one, this message is complete; there will be no subsequent continuation chunks for this message.

The seven-bit chunk code identifies what type of chunk this is, as described below.

The two-bit priority field indicates the priority level for this message, with 0 being the highest priority and 3 being the default (lowest-level) priority.

Every Minion chunk has a Chunk ID. This is a 22-bit value assigned from a monotonically increasing 22-bit cyclic counter. This means that Chunk IDs are reused every 2^22 chunks. At any given moment in time though, only a small portion of the 22-bit ID space is actively in use, so Chunk IDs are not ambiguous. Each of the four priority levels has its own 22-bit Chunk ID space, i.e., Priority 1 Chunk 7 and Priority 2 Chunk 7 are different chunks. Also, the Chunk ID spaces in opposite directions on a connection are separate. Each sender is responsible for selecting the Chunk IDs for the chunks it sends.

In some cases it is useful to refer to messages by ID, and the terms "Message ID" and "Chunk ID" are sometimes used interchangeably. For a message that is sent using a single chunk, the Message ID is the same as the Chunk ID. For a message that is sent using multiple chunks, the Message ID is the Chunk ID of the *final* chunk of the message. One implication of this is that a message's ID is undefined

until the message is complete.

Because Chunk IDs are eventually reused, issues of ID lifetime must be carefully considered in the Minion protocol design. For example, since a remote peer could, in principle, wait an arbitrary long length of time before replying to a message, the Message ID of a request that is awaiting a response MUST NOT be reused until the response has been received, and the client has disposed of the request message. Otherwise, a reply could be ambiguous, if there were two outstanding request messages both using the same Message ID at the same time. Likewise, the last Chunk ID of an incomplete message MUST NOT be reused until some subsequent chunk has been added to that message, referencing the previous Chunk ID.

The Reserved field MUST be set to zero on transmission, and MUST be ignored on reception.

For chunk types that need to refer to some other chunk, the Referenced Minion Chunk Priority (RCP) and Referenced Minion Chunk ID fields identify the referenced chunk. Note that some chunk types refer to chunks going in the same direction (e.g., a continuation chunk) and some chunk types refer to chunks going in the reverse direction (e.g., a reply chunk). For chunk types that do not to refer to any other chunk, these two fields MUST be set to zero on transmission, and MUST be ignored on reception.

The Minion Chunk payload data follows the Minion Chunk Header.

There is no explicit length field in the Minion Chunk Header, because the chunk length is determined implicitly in the RECOBS decoding step.

3.1. Minion Chunk Codes

The seven-bit chunk code identifies what kind of chunk this is. There are 128 chunk codes available. The following eight chunk codes are currently defined:

00 Continuation. This is a continuation of a previously incomplete message. The Referenced Minion Chunk ID identifies what previous chunk this is adding to. (If the Complete bit is one then this chunk is the final chunk and completes the message; no further chunks for this message will be arriving.)

01 Cancellation. This is a cancellation of a previously incomplete message. The Referenced Minion Chunk ID identifies what previous chunk this is cancelling. In this case Complete bit is unused; the Complete bit MUST be set to zero on transmission, and MUST be ignored on reception.

02 Unordered Message. This chunk begins a new unordered message. The Referenced Minion Chunk ID is unused, and MUST be set to zero on transmission, and MUST be ignored on reception.

03 Sender Ordered Message. This chunk begins a new Sender Ordered message. If received out-of-order, it should nonetheless be delivered immediately to the receiving client. The Referenced Minion Chunk ID is used to deduce the Sender Ordering that should be applied if the receiving client generates a reply to this message. If the received message identifed by the Referenced Minion Chunk ID generated a reply A', then a reply to this message should have an automatic Sender Ordering dependency that it follow message A'.

04 Receiver Ordered Message. This chunk begins a new Receiver Ordered message. This message is subject to Receiver Ordering; it MUST NOT be delivered to the receiving client until the message indicated by the Referenced Minion Chunk ID field has been delivered. If the receiving client generates a reply to this message, then the reply should have an automatic Receiver Ordering dependency that it follow the reply to the message indicated by the Referenced Minion Chunk ID field.

05 Superseding Messages. This chunk begins a new message that supersedes a preceding message. The Referenced Minion Chunk ID identifies the preceding message. If the preceding message has not yet been received, then when it does arrive, it MUST be silently discarded.

06 Chained Message. This chunk begins a new message that chains on after a preceding message. The Referenced Minion Chunk ID identifies the preceding message. This message MUST NOT be delivered to the receiving client until the previous message of the chain as been delivered to the receiving client, and this message MUST be delivered to the receiving client in a manner that indicates to the client that it is related to the previous message.

07 Reply/Acknowledge. This chunk begins a new message which is an explicit reply to a previously received message. The Referenced Minion Chunk ID identifies the received message to which this is a reply. A reply may be empty, in which case it serves as a simple acknowledgement that the request was received and accepted, or it may contain data. It is anticipated that future Minion protocol development will create additional Minion chunk codes to negotiate future protocol features. For these capability negotiation messages, an empty reply referencing the request serves as an acknowledgement that the requested protocol feature is supported.

08 Reject. A Minion Reject code indicates that the referenced received message had an error or was not accepted for some other reason. A Reject Message may be empty, or may contain data giving information concerning the reason for the rejection. It is possible to reject an incomplete message that is still arriving, by sending a Reject referencing the most recent Chunk ID for that partial message. The sender will respond by sending a Cancellation for that message, confirming that no further chunks will be sent. When used for Minion protocol capability negotiation, a Reject message referencing the request indicates that the requested protocol feature is not supported.

09 End Minion. It is anticipated that there will be existing application protocols that initially add Minion as an optional feature, which they use only when the remote peer indicates it also has Minion support, and otherwise they will communicate using the existing protocol without the Minion features. Such application protocols typically will first connect using their existing protocol, and then negotiate an "upgrade" to Minion framing. For symmetry, it would be good if such an "upgrade" were not an irreversible one-way path. We would like to offer the ability for applications to connect over raw TCP, switch to Minion for some message exchanges, and then drop back to raw TCP for some subsequent communication. This Minion chunk code exists to signal, "This is the final Minion-format message you will receive in this particular Minion session; after this you're on your own."

4. Recursively Embeddable COBS

Consistent Overhead Byte Stuffing [COBS] allows complete messages to be reliably located within an incomplete data stream that may contain gaps.

COBS works by transforming the payload data to eliminate all occurrences of zero bytes. This is like PPP byte stuffing, but more efficient; COBS has a worst-case data size overhead below 0.5%. Having created a zero-free payload, the payloads can then be concatenated into a single byte stream, separated by single zero bytes, and the zero bytes unambiguously mark the boundaries between payloads, because we know the payloads themselves no longer contain any zero bytes. At the receiving end the transformation is reversed to recreate the original payload data.

The transformation process [COBS] is, in effect, a simple run length encoding. An extremely simplified summary of the original 1997 COBS encoding is as follows:

- o If the payload begins with three nonzero bytes followed by a zero, then the output is the byte value 4 (the run length) followed by the three nonzero bytes, and the subsequent zero is skipped.
- o If that is followed by fifty nonzero bytes followed by a zero, then the output is the byte value 51 (the run length) followed by the fifty nonzero bytes, and the subsequent zero is skipped.
- o This process is repeated until the entire payload has been replaced by its zero-free equivalent.

Recursively Embeddable COBS (RECOBS) is a derivative of the original 1997 COBS encoding. RECOBS code bytes have the following meanings:

00	New payload begins
01	Represents a single zero byte
02	Two bytes: a single nonzero byte, followed by a single zero byte
03	Three bytes: two nonzero bytes, followed by a single zero byte
n	Represents n bytes: n-1 nonzero bytes, followed by a zero byte
FD	253 bytes: 252 nonzero bytes, followed by a single zero byte
FE	253 bytes: 253 nonzero bytes, with *no* following zero byte
FF	Payload ends

This has the effect that, after encoding, every payload has unambiguous bookends; every payload begins with a single 00, and ends with a single FF. Using this encoding, recursive embedding becomes possible. At *any* point in the encoded byte stream it is now possible to interrupt the byte stream, insert a new RECOBS-encoded

payload, and then resume the previous byte stream.

At the receiving end, the decoder is part-way through decoding a payload when the interruption occurs. The decoder sees a 00, which is not legal in RECOBS-encoded data, so the decoder knows a new payload is beginning. Because the decoder has not yet seen the FF end-marker for the previous payload, it knows that payload is incomplete, so it saves its decoding state for later resumption. The decoder then proceeds to decode the embedded payload. When the decoder sees the FF end-marker for the embedded payload, it delivers that fully decoded payload to the waiting client, and then resumes its decoding of the previously interrupted payload.

In principle this recursive embedding could be nested arbitrarily deeply, limited only by the amount of storage the decoder has available for partially-received payloads and their associated decoding state.

In practice, Minion limits RECOBS embedding to four levels (the base level plus three levels of nested interruption) to establish a defined upper bound on the amount of storage required by a decoder.

5. Flow Control

TCP [RFC0793] implements flow control in the form of the advertised receive window. This is to prevent a faster sender from overwhelming a slower receiver. Minion requires similar protection to prevent a slower receiver running out of memory trying to buffer messages arriving faster than it can handle them.

For a pure user-level library implementation of Minion, this is achieved by having the library set an upper bound on the amount of memory it will use for storing received messages that have not yet been handled by the client. Once this limit is met, the library ceases reading TCP data from the kernel, which causes the TCP receive window to fill up, which causes the sender to stop sending. Once the client consumes some messages, the library then reads more data from the kernel, the TCP receive window opens up, and the sender is permitted to send more data.

However, this means that there is some duplication of buffering -- the TCP receive window in the kernel and additional buffering in the user-level library. For this reason a kernel extension is proposed where a client (the Minion library in this case) can read data from the connection *without* raising the TCP receive window. In a sense it is reading the data "secretly", without admitting to the sender at the other end that it has been read. Those bytes, even though read into user space, are still counted against the TCP receive window. Later, after the client application has actually consumed the message, another kernel call is made to acknowledge consumption of those bytes, and the TCP receive window is raised.

This mechanism integrates message-level flow control with TCP's byte-level flow control, rather than having two independent flow control mechanisms happening concurrently at different levels, in ways that might interact badly with each other.

Note that the Minion protocol design will have to consider possible deadlock situations. For example, suppose one Minion host is refusing to consume any more Minion Chunks because it wishes to send a Reject message for them, but it cannot, because the peer's receive window is closed. Suppose also that the reason the peer's receive window is closed is because the peer also is sitting on a pile of unwanted Minion Chunks that it refuses to consume until it can send a Reject message for them. Possible deadlocks such as these need to be considered, and mechanisms to avoid them created.

6. Retransmission Policy

One of the main arguments that is often presented to justify why a particular application protocol is built on UDP instead of TCP is that, "UDP is better for 'real time' applications." The supporting reasoning for this is often that, "TCP insists on continuing to retransmit data long after the client doesn't need any more." In truth the real problem is not retransmission; it is that the conventional TCP APIs don't allow received data to be delivered out of order. Suppose a TCP sender has 50 packets in flight at any given time (e.g., the bandwidth x delay product is 75 kB) then the loss of a single packet causes all 49 following packets to stall at the receiver because the API doesn't allow for them to be delivered to the client until the missing packet has been received.

Minion solves this problem by allowing data to be delivered as it arrives, even if there are gaps. But the argument still remains that even after removing the ordering requirement at the receiver, it may still be a waste of bandwidth to retransmit data that will arrive too late to be useful. And indeed, it is possible with TCP to fraudulently acknowledge segments that were in fact not received, and this will cause the sender to not retransmit those segments.

However, we chose not to use fraudulent acknowledgements to suppress retransmissions, because certain NATs, Firewalls and other middleboxes may block traffic if they observe implausible protocol actions which they find suspicious. One of the important goals of Minion is 100% compatibility with today's existing Internet devices, not 99% compatibility.

We expect packet loss to be about 1% (at most a few percent) in a functioning network, and the cost of retransmitting those lost packets, even in the extreme case where *all* the retransmissions turn out to be unnecessary, is an overhead of about 1%. We argue that an overhead of about 1% is an acceptable price to pay in exchange for 100% compatibility with existing NATs, Firewalls and other middleboxes.

7. Optional Kernel Extensions

While Minion can be implemented entirely as a user-level library built on top of existing standard networking APIs like BSD sockets, it can also benefit from some optional kernel extensions:

Send Priorities

Normal TCP APIs transmit data strictly in the order is is given to the kernel. The addition of priority support allows a sendmsg() call to be used in conjunction with cmsg ancillary data to indicate the priority level of the data. For normal applications this capability would be of little use because it would most likely result in corruption of the data stream, but it is useful with Minion because the RECOBS encoding is robust against message insertion at arbitrary byte boundaries. An alternative way to achieve a similar effect is, instead of buffering data in the kernel, to keep the data in the user-space library for as long as possible. When the TCP congestion window and/or receive window rules allow more data to be sent, the kernel generates some kind of upcall (e.g., a kevent notification) to the user-space library informing it of the ability to transmit, and the user-space library responds by selecting which particular block of data to hand to the kernel next.

Just-In-Time Data Generation

Through operational experience, we have learned (not that this was any great surprise) that excessive buffering in the kernel leads to poor behaviors. For example, two messages at the same priority level are not interleaved effectively if the first message is swallowed whole by the kernel, and held in kernel buffers, before the second message is even created. When that happens, the result is that the first message is sent in its entirety, followed by the second message in its entirety, with no interleaving.

To prevent this unintended serialization, we need to avoid irrevocably handing off data to the kernel prematurely. We want to give the kernel enough data to keep the pipeline full (an amount equal to the connection's Bandwidth Delay Product) but no more.

To this end, rather than having the kernel indicate that a socket is writable any time the kernel has space available to buffer more data, we'd like the kernel to indicate that a socket is writable only when TCP (according its protocol rules, such as receive window, congestion window, and Nagle's Algorithm) would be willing to send data, but has no data available to send. When this situation occurs, the socket becomes writable, and the client (the user-level Minion library) is able to perform a just-in-time

determination of what data ought to be sent next.

This just-in-time data generation could be achieved in the BSD sockets API by adding a new socket option. When using this new socket option, a socket will only be writable when TCP is actively waiting for new data. If the context-switching latency or software overhead is such that it takes the user-level code a little too long to generate data strictly on demand, then a middle ground can be achieved by modifying the new socket option such that a socket will only be writable when the socket has less data buffered than it expects to need imminently. For example, a TCP connection in slow start expects it will need four TCP segments when the next ack arrives. When used this way, if an incoming ACK allows TCP to send out four segments then those four segments are already buffered and ready in the kernel, and the socket then becomes writable again to allow the user-level code to generate the next four segments, so that they will be ready and waiting the next time TCP is able to transmit additional segments.

We are currently experimenting with just-in-time data generation. If it proves to be as effective as we hope, it might even work well enough to provide effective priority support too, eliminating the need for the "Send Priorities" kernel extension.

Immediate Receive

Normal TCP APIs deliver data only in TCP sequence number order. The addition of support for new cmsg ancillary data in the recvmsg() call allows the user-space library to request *any* available data, not only in-order data. The cmsg ancillary data returned from the recvmsg() call indicates to the user-space library where in the TCP sequence space this particular block of data lies. A setsockopt() option (or equivalent) is also required to put the socket into this "Immediate Receive" mode, to inform the kernel that the client will accept out-of-order data on this socket, and therefore the client should be notified (via select(), kevent(), etc.), not only when there is in-order data available to be read, but also when there is out-of-order data available to be read.

Integrated Receive Window

Normal TCP APIs raise the receive window any time data is read out of the kernel into user space. The addition of new cmsg ancillary data in the recvmsg() call allows the user-space library to request that the kernel return received data *without* reflecting this in its receive window calculation. After the client application has consumed the message data from the user-space Minion library, the Minion library makes a subsequent recvmsg() call with appropriate cmsg ancillary data to inform the kernel how

many bytes to add back into its receive window. In essence, the receive window boundary is stretched outside the kernel to account for data held by *both* the kernel *and* the user-space Minion library.

These optional kernel extensions are a key part of what makes Minion compelling. Minion can be adopted today by any application, using Minion as a purely user-space library. Such an application performs as well as any application can when it is built on top of standard TCP. However, unlike an application built on top of standard TCP, Minion offers the promise of future kernel support for even better performance. Any given application with its own application-specific protocol is unlikely to receive special kernel support to make just that one application work better. But when many applications all use the Minion protocol, it then becomes reasonable to add kernel support to improve all of those applications.

8. TCP Deviations

When implemented entirely as a user-level library, Minion naturally adheres to the TCP specifications (insofar as the underlying operating system adheres to the TCP specifications) because Minion is merely using the operating system's networking APIs.

When optional kernel extensions are in use, they may allow Minion to deviate from classical TCP protocol rules. One such instance of this deviation has already been identified. The TCP protocol rules allow a sender to send a FIN to end a connection, and then follow it with additional data bytes (with higher TCP sequence numbers, so that they fall later in the data stream) which the receiver is expected to discard because it recognizes that they fall after the FIN in the data stream. When out-of-order delivery is enabled, it's possible that if the TCP segment containing the FIN is lost or delayed, then subsequent TCP segments containing data bytes could be incorrectly delivered to the client application, when the TCP protocol rules dictate that they should have been discarded. The ability to send data following the FIN that the receiver is expected to discard is incompatible with out-of-order delivery. Note that this is referring to data that follows the FIN in TCP sequence number space, not data that follows the FIN in transmission order. If, after the FIN has been sent, previously transmitted data is lost and needs to be retransmitted, then this does not cause any problems; the bytes in such retransmitted TCP segments fall *before* the FIN in TCP sequence number space, not after. As a result of this observation, TCP's protocol rules, when used with Minion traffic, are effectively modified as follows:

- o A client using Minion MUST NOT send new data on a connection after that connection has been closed (i.e. a FIN indication has been sequenced and sent).

In reality we do not expect this to be a major burden to TCP implementations. We are not aware of TCP implementations that send data after a connection is closed and then rely on the receiver to discard that data.

9. IANA Considerations

No IANA actions are required by this document.

10. Security Considerations

We take security seriously. As this work develops, this section will contain details of any known security issues and possible mitigations.

11. Acknowledgements

Many thanks to Bryan Ford, Padma Bhooma and Anumita Biswas for their contributions to the development of Minion.

Thanks to Joe Touch for pointing out that Minion restricts TCP's ability to send data, after a connection is closed, that will then be ignored by the receiver.

12. References

12.1. Normative References

[COBS] Cheshire, S. and M. Baker, "Consistent Overhead Byte Stuffing", September 1997, <http://stuartcheshire.org/papers/COBSforToN.pdf>.

[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.

[RFC6347] Rescorla, E. and N. Modadugu, "Datagram Transport Layer Security Version 1.2", RFC 6347, January 2012.

[minserv] Iyengar, J., "Minion - Service Model and Conceptual API", draft-iyengar-minion-concept-02 (work in progress), October 2013.

12.2. Informative References

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

[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.

[RFC6887] Wing, D., Cheshire, S., Boucadair, M., Penno, R., and P. Selkirk, "Port Control Protocol (PCP)", RFC 6887, April 2013.

[Study] Hatonen, S., Nyrhinen, A., Eggert, L., Strowes, S., Sarolahti, P., and M. Kojo, "An Experimental Study of Home Gateway Characteristics", September 1997, <http://conferences.sigcomm.org/imc/2010/papers/p260.pdf>.

[TCPoUDP] Cheshire, S., Graessley, J., and S. Cheshire, "Encapsulation of TCP and other Transport Protocols over UDP", draft-cheshire-tcp-over-udp-00 (work in progress), June 2013.

Authors' Addresses

Janardhan Iyengar
Franklin and Marshall College
Mathematics and Computer Science
PO Box 3003
Lancaster, Pennsylvania 17604-3003
USA

Phone: +1 717 358 4774
Email: janardhan.iyengar@fandm.edu

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