Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: January 01, 2014

J. Iyengar
Franklin and Marshall College
S. Cheshire
J. Graessley
Apple
June 30, 2013

Minion - Service Model and Conceptual API
draft-iyengar-minion-concept-00

Abstract

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. Minion's richer facilities include a message-oriented API rather than TCP's unstructured byte-stream service model, multiplexing of multiple messages (or message streams) on a single connection, interleaving of multiplexed messages (to eliminate head-of-line blocking), message cancellation, request/reply support, ordered and unordered messages, chained messages, multiple priority levels with byte-granularity preemption, and DTLS Security.

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

Back in 1983 application developers had the choice of UDP [RFC0768] or TCP [RFC0793]. UDP preserves message boundaries, but provides no reliability, ordering, flow control, or congestion control, and only supports small messages (typically UDP packets larger than 1468 bytes result in undesirable IP fragmentation). TCP provides these important facilities, and doesn't impose any message size limit -- but only because it doesn't have any concept of messages, and doesn't claim to preserve message boundaries. Consequently, whichever base protocol the application developer chose, they were left building part of the transport-layer solution themselves.

Thirty years later, in 2013, little has changed. Application developers on mainstream platforms like Android and iOS still have the same two choices -- UDP and TCP.

Attempts to provide richer application facilities have failed to achieve widespread adoption. Protocols like SCTP are not supported by mainstream NAT gateways. Consequently, mainstream apps for platforms like Android and iOS don't use SCTP, because it would severely limit their real-world deployment. Consequently, operating systems like Android and iOS don't have in-kernel native implementations of SCTP, because there's little developer demand for a protocol they can't use. Consequently, there's little incentive for NAT gateway vendors to do the work to add support for a protocol that's neither supported in the popular operating systems nor used by mainstream applications.

Like SCTP, Minion goes beyond UDP and TCP by providing richer application facilities, making it possible to create applications that work better and more reliably (and can be brought to market quicker and easier) than is possible when each application has to re-create those facilities from scratch every time.

However, unlike SCTP, Minion provides facilities that can be used by an application developer immediately, without having to wait for OS support or NAT gateway support. OS support and NAT gateway support can come later, and provide additional incremental improvements. This incremental deployment path -- which begins first with the application developer who can choose to use Minion and immediately reap the benefits of that decision -- is an important property of Minion, and removes one of the major obstacles that hindered SCTP adoption.

When used without kernel support, Minion acts like a typical TCP-based application protocol, and as such, performs as well as any other TCP-based application protocol. However, unlike most application-specific protocols, Minion also offers the potential of kernel support giving better low-latency message performance and better prioritization. A general application protocol is unlikely to receive special-case kernel support tailored to support that one specific application, but as a general-purpose transport protocol built to support a wide range of applications, special kernel support for Minion is feasible.

Minion preserves the important properties of TCP, like congestion control, while adding a range of richer application facilities:

Message Oriented
: Rather than an unstructured byte stream, Minion supports messages. TCP provides an unstructured byte stream, but virtually every application needs to send and receive semantic messages, which means that virtually every application needs to build its own message framing mechanism on top of TCP. In contrast, Minion respects and preserves semantic boundaries. If an application writes a 27-byte message followed by a 53-byte message, then a 27-byte message and a 53-byte message are delivered to the receiving client, not a single combined block of 80 bytes.

Arbitrary Size Messages
: While most Minion messages are expected to be small, Minion itself imposes no upper limit on message size. For example, a 6 gigabyte movie download could be sent as a single Minion message. Messages do not have to fit in memory. A very large message can be generated incrementally by the sender, and will be delivered incrementally to the receiver.

Multiplexing
: Multiple messages can be sent, in both directions, on a single Minion connection. Unlike protocols like HTTP/1.0 where each request used a separate connection, many Minion messages can share a connection.

Interleaving
: When multiple (possibly large) messages are being sent concurrently on a single Minion connection, the connection bandwidth is shared round-robin between the messages. This avoids head-of-line blocking, where messages are blocked waiting for a large message to complete.

Cancellation
: Messages do not have to be sent to completion. If either the sender or the receiver determines that a message is no longer needed, then that single message can be cancelled without having to tear down the entire Minion connection.

Request/Reply Support
: Many application protocols are request/reply-oriented. Minion facilitates this by allowing an outgoing message to be explicitly identified as a reply to a previously-received message, which causes the reply message to be delivered automatically to the appropriate message handler at the receiving end.

  Replies do not have to be delivered in the same order that the requests were received. When multiple (possibly large) replies are in flight at the same time, the interleaving and bandwidth-sharing described above applies, as it does for all Minion messages.

  Replies can themselves generate further replies, resulting in an unbounded back-and-forth of ping-pong messages, each going to the appropriate reply handler on the receiving side.

Unordered Messages
: 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. A simple kernel extension (in the form of a new socket option) removes this limitation, and allows out-of-order data to be delivered to the client. This avoids the problem where a single lost TCP segment causes all the following TCP segments to be delayed.

Note that this kernel extension is not *required* for a client to use Minion; it it an optional extension that provides better performance for real-time applications in situations where there is packet loss or reordering. For many applications it is an irrelevant benefit and they can operate perfectly well without it. For a few applications it is a significant benefit, and it allows Minion to provide the low-latency performance that often drives developers to use UDP.

Receiver Ordering

Although sometimes it can be desirable to receive messages out of order as they arrive, often it is not. In many cases the application cannot usefully use (certain) messages out of order, and delivering them potentially out of order would burden the application with the task of sorting the messages back into the correct order before processing them. In such cases, it is more convenient for the application to have Minion deliver messages to it in the right order. For this reason Minion supports ordered messages as well as unordered messages. Unordered messages and ordered messages are supported simultaneously on a single Minion connection.

Other transport protocols support the notion of multiple message streams sharing a single connection. Minion takes this idea and generalizes it to the more expressive notion of Receiver Ordering Message Dependencies. Receiver Ordering Message Dependencies indicate that a dependent message must not be delivered before the message it depends upon.

Traditional message streams can be created in Minion by using a sequence of Receiver Ordering Message Dependencies: If message B is specified to follow message A, and message C is specified to follow message B, and so on, then messages A,B,C... form an ordered "stream". Similarly, if at the same time message Q is specified to follow message P, and message R is specified to follow message Q, and so on, then messages P,Q,R... form another independent ordered "stream" of their own.

In addition to such disjoint ordered streams (A,B,C... and P,Q,R...), Receiver Ordering Message Dependencies also allow richer relationships to be expressed. For example, in H.264

video, P-frames reference I-frames, but P-frames do not reference other P-frames. If a single P-frame is lost or delayed, it is not necessary to delay all subsequent P-frames. Each P-frame has a time it is due to be displayed, and when that time arrives the frame should be displayed if possible, even if (or especially if) preceding P-frames did not arrive in time. However, there is no benefit in delivering a P-frame to the application before the I-frame it depends upon.

To give another example, a web browser client may need to retrieve many resources to display a page, but it cannot display *any* of the page until it has received the style sheet. Consequently it would be beneficial if the web browser client could request all of the resources it needs, but for each one, indicate that it depends on the style sheet resource (or upon some other resource which depends by transitive closure on the sheet resource). This dependency information tells the sender that it should not devote *any* bytes of available bandwidth to delivering other resources until after it has completed sending the all-important style sheet.

Sender Ordering

Even in cases where the receiver does not have a strict ordering requirement, it may still be useful to cause data packets to be sent in a favourable order. For example, with a group of H.264 video P-frames, the first frame of the group is likely to be needed for playback sooner than the last frame of the group. Therefore, delivering them all concurrently by sharing bandwidth between them may cause the first frame to be delivered too late to be played. In this case the sender uses Sender Ordering to indicate that a particular message should follow another message on the wire.

Sender Ordering is more lightweight than Receiver Ordering; it is used solely to control the transmission order, and is not communicated to the receiver. If a message is lost or delayed in transit then following messages are still delivered to the application immediately, except when an explicit Receiver Ordering Message Dependency indicates that they should not be.

Chained Messages

Minion is intended to be used to deliver messages containing a single logical semantic unit. Although Minion can "stream" a message of unbounded size to the receiver, Minion is not generally intended to be used to batch multiple logical semantic units into single large message, which is then "streamed" to the receiver, which then parses the incoming "streamed" message as it arrives for the logical semantic units contained within it. Part of the

purpose of Minion is to take the burden of message framing off the application writer; treating a single unbounded "streaming" Minion message like a TCP connection places the burden of parsing firmly back in the hands of the application developer.

In the event that a logical message contains multiple related parts, like a header with an associated body, Minion can facilitate this structuring through the use of Chained Messages.

Chained Messages are substantially similar to Receiver Ordering Message Dependencies, except that in addition to controlling the order of data transmission on the wire, and the order of message delivery to the client, the chained message relationship is also exposed to the client application at the receiving end. Instead of being delivered to the Minion connection's main message handler function, the way most messages are, Chained Messages are delivered instead to the Chained Message handler function of the previous message in the chain.

The Chained Message mechanism allows an application to provide a main message handler function that receives and processes the "header" portion of each two-part message, and that main message handler function in turn provides a different message handler function that receives and processes the subsequent "body" portion.

As with other messages, each component in a message chain can optionally generate an explicit reply, which is delivered to the reply handler for the originating message.

If any message in a chain is cancelled by the sender or the receiver, then all subsequent messages in that chain are implicitly cancelled.

The sender of a chain of messages may wait at each step for a reply confirming that the previous message was acceptable before sending the next message of the chain, or it may send the entire chain and let the receiver cancel the message chain if an error occurs.

Priority Levels

While Receiver Ordering, Sender Ordering, and Message Chaining allow relationships between messages to be adequately expressed where they are known in advance, sometimes there are urgent messages that need to be sent at short notice that are not known in advance. For example, consider a music application which is streaming out audio data with a generous playback buffer, and then the user performs a user-interface operation to change the volume

level. We would like this volume change to be performed as promptly as possible, regardless of how much audio data is queued up in the transmit buffer. For this reason Minion supports four priority levels. A higher-priority message can preempt a lower-priority message at any arbitrary byte boundary in the lower-priority data stream. (This byte-granularity preemption is made possible by the Minion wire protocol [minprot]).

Minion provides strict priorities, meaning that no lower-priority data at all is sent as long as there is higher-priority data waiting. This means that a sustained flow of higher-priority data can starve lower-priority data indefinitely. For this reason, Minion priorities are intended to support small amounts of high-priority data intermixed with larger amounts of lower-priority data. If the amount of high-priority data exceeds the current throughput of the Minion connection then all the available throughput will be consumed attempting to meet the high-priority data demand, and no lower-priority data will be sent. If this outcome is undesirable to the application, it should ensure that it does not generate sustained high-priority data at a rate exceeding the network throughput for a prolonged period of time. Minion does not attempt to provide proportional or weighted bandwidth allocation between different priority levels.

The Minion model is that if message B has a Receiver Ordering or Sender Ordering dependency upon message A, then Minion should not expend any available throughput delivering any part of message B until after message A has been entirely sent. Similarly, if message C is higher priority than messages A and B, then Minion should not expend any available throughput delivering any part of messages A or B until after message C has been entirely sent.

DTLS Security
: Minion includes security support. Because of the potential for out-of-order message reception, Minion uses DTLS (which includes an explicit record number) instead of TLS (which assumes strictly-ordered delivery over TCP).

3. Conceptual API

While different implementations in different languages may provide APIs that differ in details and programming model, the common conceptual framework of Minion APIs is as follows:

- Outbound Connections
  - Create new Minion Connection to remote peer, with handler function or object to receive incoming messages.

    * Close Minion Connection when it is no longer needed.

- Inbound Connections

    * Listen on a port for incoming connections.

    * Upon receipt of incoming connection request, a new Minion Connection object (substantially similar to the Outbound Minion Connection object above) is generated, and delivered to the application.

    * Set handler function or object to handle incoming messages received on an inbound connection.

    * Stop listening for incoming connections.

- Sending Messages

    * Create new Outbound Minion Message, associated with an existing connection (either outbound or inbound), specifying the priority level for the message.

    * Optionally, indicate Sender Ordering for this message by reference to some previously-created Minion message.

    * Optionally, indicate Receiver Ordering for this message, or Chaining for this message, or that this message is a reply to some previously received Minion message. Note that these three options are mutually exclusive. An outgoing message can be identified as a response to a received message, or a subsequent member of a multi-part message chain, or a message with a Receiver Ordering Message Dependency, but not more than one of these three things.

    * Optionally, provide a reply handler function or object to receive replies to this message.

    * Provide (possibly incomplete) data for the message.

    * Optionally, add further units of data to the message.

    * Indicate when message is complete. This tells the Minion implementation layer that it should now send the message. Alternatively, a message can also be cancelled if it is no longer needed.

    * Dispose of the message when it is no longer needed.

- Receiving Messages
  - Upon receipt of a message, a handler function or object is handed a new inbound message:
  - If the message is a chained continuation message, and a specific handler exists for that chain, then that specific handler is invoked.
  - Else, if the message is a reply, and a specific handler exists for the originating message, then that specific handler is invoked.
  - Else, the Minion Connection's generic message handler is invoked.
  - Read data from the message. For large messages, this may not be the entire message. After one or more reads, a return code (or similar) indicates to the application when the message is complete (or alternatively, that is is incomplete, and will not be completed, because it has been cancelled by the sender).
  - The application may decide to reject a message before it has been entirely received, by canceling it.
  - The message handler may generate outbound messages in response to the received message, including outbound explicit reply messages, outbound chained messages, and simple outbound standalone messages.
  - Dispose of the received message when it is no longer needed.

4. Client Isolation

Minion allows multiple messages to share the available throughput of a single connection. The sources of those multiple messages (if not the same application) are assumed to be mutually trusting. Minion does not attempt to prevent one message source on a connection from consuming an unfair share of the bandwidth, nor does Minion attempt to guard against a client that fails to read its messages, causing the receive window to close, thereby preventing any messages from being received.

In the event that some proxy or similar technology allows multiple mutually untrusting clients to share a single Minion connection, that application-layer code that is allowing the single Minion connection to be shared is responsible for policing the traffic so that the single Minion connection is shared reasonably.

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. Acknowledgements

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

8. References

8.1. 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.

8.2. Informative References

[minprot] Iyengar, J., "Minion - Wire Protocol", draft-iyengar-minion-protocol-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