TAPS Working Group
Internet-Draft
Intended status: Informational
Expires: May 4, 2017

S. McQuistin
C. Perkins
University of Glasgow
M. Fayed
University of Stirling

October 31, 2016

Transport Services for Low-Latency Real-Time Applications
draft-mcquistin-taps-low-latency-services-00

Abstract

This document describes the set of transport services required by low-latency, real-time applications. These services are derived from the needs of the applications, rather than from the current capabilities of the transport layer. An example API, based on the Berkeley Sockets API, is also provided, alongside examples of how the target applications would use the API to access the transport services described.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on May 4, 2017.

Copyright Notice

Copyright (c) 2016 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect

to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction 2
2. Low-Latency Transport Services 3
3. Abstract API 5
3.1. Socket Setup & Teardown 5
3.2. Socket Options 6
3.3. Connection Handling 6
3.4. Timing 7
3.5. Messaging 7
3.6. Utilities 8
3.7. Design Assumptions 8
4. Example Usage 9
4.1. HTTP/1.1 9
4.2. HTTP/2 9
4.3. Real-Time Multimedia 9
5. IANA Considerations 10
6. Security Considerations 10
7. Informative References 11
Authors' Addresses 12

1. Introduction

The goal of the TAPS working group is to break down the existing transport layer into a set of transport services, decoupling these from the transport protocols that provide them. As a result, the standardisation of transport services becomes feasible and important. The first phase of this process has been to inspect the existing transport protocols, and determine the services they provide. The next phase, enabled both by the use of transport services and their separation from transport protocols, is to define novel transport services based on the needs of applications. In essence, these services are derived from applications by thinking about what the transport layer would provide to applications if the constraints of the existing transport protocols were not in place.

This document considers the transport services required by low-latency applications. This is an important class of applications, not only because it represents a significant portion of Internet traffic, but because it is arguably poorly served by the existing transport layer: the available transport protocols bundle services together in such a way that no one protocol provides all the services required.

After detailing the transport services required by these applications, a sample API that provides these services is outlined. This API is then used to demonstrate how applications might make use of the transport services described.

This document does not consider how the transport services described map to the transport protocols that might provide them.

2. Low-Latency Transport Services

The decoupling of transport services from transport protocols allows for the development of novel transport services, based on the needs of applications. This section describes the transport services required by low-latency applications.

Timing

Timing is the most essential characteristic of the data generated by low-latency applications. Data has a lifetime associated with it: a deadline, relative to the time the data was produced, by which it must be delivered. Once this lifetime has been exceeded, the data is no longer useful to the receiving application. Data lifetimes depend on the latency bounds of the application. Interactive applications, such as telephony, video conferencing, or telepresence, require a low end-to-end latency, ranging from tens to a few hundred milliseconds. Non-interactive applications can accomodate higher latency, with bounds in the order of seconds.

There are a number of factors used in predicting if data will arrive within its lifetime: the time between being sent by the application and being sent on the wire, the one-way network delay, and the length of time the data will be buffered at the receiver before being used (the play-out delay). Multimedia applications, which comprise a significant portion of the target application area, have a play-out buffer to reduce the impact of jitter. Estimates for both RTT and play-out delay must be available to the sender.

Partial Reliability

The combination of a lossy, best-effort network (e.g., IP) layer and support for timing and lifetimes results in the need for a partially reliable service. Given the limitations of forward error correct techniques, there is some probability that packet loss can only be recovered from by retransmitting the lost packet. This introduces potentially unbounded delay, given that retransmissions themselves may be lost. Therefore, timing and fully reliable transport services cannot be provided

together -- the reliable delivery of data cannot be guaranteed within a given lifetime.

This implies a partial reliability service, where data is delivered reliably only while it is likely to be useful to the receiving application. Once a message has exceeded its lifetime, attempts to transmit it will be abandoned.

Dependencies
: Partial reliability means that not all data that is sent will be received successfully. This means that a dependency management transport service is required. Data must not be sent if it relies upon earlier data that has not been successfully delivered.

Messaging
: Given that not all data will arrive successfully, it is important to maximise the utility of the data that does arrive. Application-level framing [CCR20] allows the application to package data into units that are useful (assuming delivery of their dependencies) to the receiver. The combination of a messaging service (to maintain application data unit boundaries) and a dependency services provides greater utility to applications than a stream-oriented service on lossy networks. To minimise latency, messages are delivered to the application in the order they arrive. Reordering messages into transmission order introduces latency when applied across a lossy IP network; messages may be buffered waiting for earlier messages to be retransmitted. Application-layer protocols, such as RTP, introduce sequencing, allowing applications to reorder messages if required. Introducing order at this layer makes the resultant latency explicit to the application.

Multistreaming
: Messaging enables a multistreaming service. Many applications are comprised of multiple streams, each with their own characteristics with regards to the other services listed. For example, a typical multimedia application has at least two flows for audio and video, each with different properties (e.g., loss tolerance, message sizes). A multistreaming service allows these streams to be configured separately.

Multipath
: Multiple paths often exist between hosts; a multipath service is required to allow applications to benefit from this. This is an extension of the multistreaming service: different streams should be mapped to the most suitable path, given the configuration of the stream, and the network conditions of the

path. Messaging is required to make optimal use of multiple paths with different loss and delay characteristics.

Congestion Control
: Congestion control is an essential service. A given algorithm is not suitable for all traffic types, and so one is not prescribed. The service should require the use of a suitable congestion control algorithm, and enforce this using a circuit breaker.

Connections (optional)
: Maintaining per-connection metadata at the endpoints is helpful for the implementation of many congestion control algorithms. Further, connection setup and teardown messages can also benefit in-network services, including NAT traversal and firewall pinhole management. As a result, it is often desirable to have support for a connection-oriented service.

This set of transport services demonstrates the need for a top-down approach. Timing is a crucial characteristic of low-latency applications, from which the other services follow.

3. Abstract API

This section describes an abstract API that supports the transport services described in Section 2. This allows the usage of these services to be demonstrated in Section 4.

It should be noted that the main contribution of this document is the set of transport services specified in Section 2. An abstract API is described here to illustrate how these services might be provided, based on Berkeley Sockets for familiarity. Other APIs, not constrained by the limitations of Berkeley Sockets, would be more appropriate.

3.1. Socket Setup & Teardown

Hosts setup and tear-down sockets using the socket() and close() functions, as in the standard Berkeley sockets API.

The function signatures are:

```
int socket(int address_family,
           int socket_type);

int close(int sd);
```

socket() returns a socket descriptor (sd) on success, while close() returns 0 on success, and -1 on failure.

3.2. Socket Options

Socket options can be set and read using the setsockopt() and getsockopt() functions respectively. This mirrors the standard Berkeley sockets API.

The function signatures are:

```
int getsockopt(int sd,
               int level,
               int option,
               void *value,
               socklen_t *len);

int setsockopt(int sd,
               int level,
               int option,
               const void *value,
               socklen_t len);
```

Both return 0 on success, and -1 on failure.

3.3. Connection Handling

The connection primitives are the same as those of TCP sockets. Servers bind() to a particular address and port, then listen() for and accept() incoming connections. Clients connect() to a server.

The function signatures are:

```
int bind(int sd,
         struct sockaddr *addr,
         socklen_t addrlen);

int listen(int sd);

int accept(int sd,
           struct sockaddr *addr,
           socklen_t *addrlen);

int connect(int sd,
            const struct sockaddr *addr,
            socklen_t addrlen);
```

All return 0 on success, and -1 on failure.

3.4. Timing

Once a connection has been establised, the receiver then indicates its media play-out delay, in milliseconds, via the set_po_delay() function. This specifies how long the application will buffer data for before it is rendered to the user. The play-out delay is fed back to the sender, for use in estimating message liveness.

The function signature is:

```
int set_po_delay(int delay);
```

The function returns 0 on success, and -1 on failure.

3.5. Messaging

Message-oriented data transmission is exposed by the send_message() and recv_message() functions. These expose a partially reliable message delivery service to the application, framing data such that either the complete message is delivered, or lost in its entirety.

The function signatures are:

```
int send_message(int sd,
                 const void *buf,
                 size_t len,
                 uint16_t *seq_num,
                 int lifetime,
                 uint16_t depends_on,
                 uint8_t substream);

int recv_message(int sd,
                 void *buf,
                 size_t len,
                 uint16_t *seq_num,
                 uint8_t *substream);
```

send_message() returns the number of bytes sent and the sequence number of the message, while recv_message returns the sub-stream identifier and length of the message, along with the received message data.

It is instructive to compare the partially reliable send and receive functions to their Berkeley sockets API counterparts. The send_message() call takes three additional parameters, providing support for the transport services described in Section 2:

- Lifetime: combined with an estimate of the round-trip time, the time that the message has spent in the sending buffer, and the play-out delay, to estimate message liveness

- Dependency message sequence number: used to determine if this message depends on another that was not sent successfully

- Sub-stream identifier: to provide the multistreaming service

send_message() returns the sequence number of the sent message, allowing it to be used as the dependency of future messages. The sequence number will increase by 1 for each message sent (and wrap around within the 16-bit field). recv_message() returns the sequence number of the received message, allowing the application to identify gaps in the sequence space (indicating reordering or loss).

[Discussion question: would it be useful for recv_message() to expose the message arrival time?]

3.6. Utilities

Two utility functions are needed to support the other services. To allow applications to size messages to increase their utility, get_pmtu() provides the path MTU. get_rtt_est() provides an estimate of the round-trip time, to enable applications to calculate an appropriate play-out delay value.

The function signatures are:

```
int get_pmtu(int sd);

int get_rtt_est(int sd);
```

get_pmtu() returns the path MTU in bytes. get_rtt_est() returns the current round-trip time estimate in milliseconds.

Endpoints need to send data regularly to obtain an accurate RTT estimate. Where an endpoint would not otherwise transmit data, the messages sent by set_po_delay() will allow an RTT estimate to be calculated.

3.7. Design Assumptions

The API specified here makes a number of assumptions about how the services described in Section 2 should be provided. For example, the way in which sequence numbers are implemented limits messages to expressing their dependence on messages that have been sent

previously. A different API may be implement a different set of constraints on the dependency management service.

[Discussion questions: does this mean that the services aren't properly defined? How much flexibility should be given when designing an API?]

4. Example Usage

In this section, examples of how applications might use the API described in Section 3 are given. This is important, not only for demonstrating the usability of the API, but for validating the selection of transport services described.

4.1. HTTP/1.1

tbd

[Not typically described as "low latency", but is used by many applications that would benefit from lower latency. Mapping to a message-oriented transport service is not obvious.]

4.2. HTTP/2

tbd

[Mapping to message-oriented transport is a lot more obvious.]

4.3. Real-Time Multimedia

Real-time multimedia applications typically make use of RTP [RFC3550] as their application-layer protocol. This adds a header to each message, with timing and sequencing metadata. RTP is compatible with the partially reliable, unordered message delivery model described, making its mapping to the API straightforward.

Sender:

- The sender opens a socket(), and bind()s to a particular address and port. It then listen()s for and accept()s incoming connections.

- Messages are sent using send_message(). The application specifies the lifetime of the message; this is the maximum delay each message can tolerate. For example, a VOIP application would set this to 150ms. The sequence number of message this message depends on is set, and the sub-stream specified. get_pmtu() can be used to help determine the message sizes.

- At the end of the connection, the sender close()s the socket.

Receiver:

- The receiver opens a socket(), and connect()s to a server.
- Once the connection has been established, the receiver sets its play-out delay, in milliseconds, using set_po_delay(). The utility function get_rtt_est() can be used in calculating this value. The play-out delay can be changed over time.
- Messages are received using recv_message(). The application receives the sequence number and sub-stream identifier alongside the message data.
- At the end of the connection, the receiver close()s the socket.

5. IANA Considerations

This memo includes no request to IANA.

6. Security Considerations

The transport services described do not themselves introduce any security considerations beyond those of TCP or UDP. No additional metadata needs to be exposed on the wire to provide the transport services described. The transport services result in a partially reliable, message-oriented delivery model. As a result, Datagram TLS [RFC6347] is required for security and encryption.

[Discussion question: should DTLS/security be listed as an essential service?]

There are a number of options for deploying the transport services described in this document. They could be deployed within a new protocol, but this will likely limit deployability. Alternatively, for greater deployability, existing protocols could be modified. Partially Reliable SCTP [RFC3758] and WebRTC's data channels [I-D.ietf-rtcweb-data-channel] provide a similar set of services over SCTP. For maximum deployability, the services should be provided over TCP. Modifying TCP's in-order byte stream abstraction to provide an out-of-order message-oriented delivery model is challenging. The designs of TCP Hollywood [IFIP2016], Minion [I-D.iyengar-minion-protocol], and QUIC [I-D.hamilton-quic-transport-protocol] show ways of addressing these challenges.

7. Informative References

[CCR20] Clark, D. and D. Tennenhouse, "Architectural Considerations for a New Generation of Protocols", ACM SIGCOMM Computer Communications Review 20 (4) 200-208, DOI 10.1145/99508.99553, September 1990, <http://dx.doi.org/10.1145/99508.99553>.

[I-D.hamilton-quic-transport-protocol] Hamilton, R., Iyengar, J., Swett, I., and A. Wilk, "QUIC: A UDP-Based Multiplexed and Secure Transport", draft-hamilton-quic-transport-protocol-00 (work in progress), July 2016.

[I-D.ietf-rtcweb-data-channel] Jesup, R., Loreto, S., and M. Tuexen, "WebRTC Data Channels", draft-ietf-rtcweb-data-channel-13 (work in progress), January 2015.

[I-D.iyengar-minion-protocol] Jana, J., Cheshire, S., and J. Graessley, "Minion - Wire Protocol", draft-iyengar-minion-protocol-02 (work in progress), October 2013.

[IFIP2016] McQuistin, S., Perkins, C., and M. Fayed, "TCP Hollywood: An Unordered, Time-Lined, TCP for Networked Multimedia Applications", IFIP Networking 2016, DOI 10.1109/IFIPNetworking.2016.7497221, May 2016, <http://dx.doi.org/10.1109/IFIPNetworking.2016.7497221>.

[RFC3550] Schulzrinne, H., Casner, S., Frederick, R., and V. Jacobson, "RTP: A Transport Protocol for Real-Time Applications", STD 64, RFC 3550, DOI 10.17487/RFC3550, July 2003, <http://www.rfc-editor.org/info/rfc3550>.

[RFC3758] Stewart, R., Ramalho, M., Xie, Q., Tuexen, M., and P. Conrad, "Stream Control Transmission Protocol (SCTP) Partial Reliability Extension", RFC 3758, DOI 10.17487/RFC3758, May 2004, <http://www.rfc-editor.org/info/rfc3758>.

[RFC6347] Rescorla, E. and N. Modadugu, "Datagram Transport Layer Security Version 1.2", RFC 6347, DOI 10.17487/RFC6347, January 2012, <http://www.rfc-editor.org/info/rfc6347>.

Authors' Addresses

Stephen McQuistin
University of Glasgow
School of Computing Science
Glasgow G12 8QQ
UK

Email: sm@smcquistin.uk

Colin Perkins
University of Glasgow
School of Computing Science
Glasgow G12 8QQ
UK

Email: csp@csperkins.org

Marwan Fayed
University of Stirling
Department of Computing Science & Maths
Stirling FK9 4LA
UK

Email: mmf@cs.stir.ac.uk