

TAPS
Internet-Draft
Intended status: Informational
Expires: December 22, 2017

S. Gjessing
M. Welzl
University of Oslo
June 20, 2017

A Minimal Set of Transport Services for TAPS Systems
draft-gjessing-taps-minset-05

Abstract

This draft recommends a minimal set of IETF Transport Services offered by end systems supporting TAPS, and gives guidance on choosing among the available mechanisms and protocols. It is based on the set of transport features given in the TAPS document draft-ietf-taps-transport-services-usage-05.

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 December 22, 2017.

Copyright Notice

Copyright (c) 2017 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	3
2. Terminology	4
3. The Minimal Set of Transport Features	5
3.1. Flow Creation, Connection and Termination	5
3.2. Flow Group Configuration	6
3.3. Flow Configuration	7
3.4. Data Transfer	7
3.4.1. The Sender	7
3.4.2. The Receiver	8
4. An Abstract MinSet API	9
5. Conclusion	14
6. Acknowledgements	14
7. IANA Considerations	15
8. Security Considerations	15
9. References	15
9.1. Normative References	15
9.2. Informative References	15
Appendix A. Deriving the minimal set	17
A.1. Step 1: Categorization -- The Superset of Transport Features	17
A.1.1. CONNECTION Related Transport Features	19
A.1.2. DATA Transfer Related Transport Features	31
A.2. Step 2: Reduction -- The Reduced Set of Transport Features	35
A.2.1. CONNECTION Related Transport Features	36
A.2.2. DATA Transfer Related Transport Features	37
A.3. Step 3: Discussion	37
A.3.1. Sending Messages, Receiving Bytes	38
A.3.2. Stream Schedulers Without Streams	39
A.3.3. Early Data Transmission	40
A.3.4. Sender Running Dry	41
A.3.5. Capacity Profile	42
A.3.6. Security	42
A.3.7. Packet Size	42
Appendix B. Revision information	43
Authors' Addresses	43

1. Introduction

The task of any system that implements TAPS is to offer transport services to its applications, i.e. the applications running on top of TAPS, without binding them to a particular transport protocol. Currently, the set of transport services that most applications use is based on TCP and UDP; this limits the ability for the network stack to make use of features of other protocols. For example, if a protocol supports out-of-order message delivery but applications always assume that the network provides an ordered bytestream, then the network stack can never utilize out-of-order message delivery: doing so would break a fundamental assumption of the application.

By exposing the transport services of multiple transport protocols, a TAPS system can make it possible to use these services without having to statically bind an application to a specific transport protocol. The first step towards the design of such a system was taken by [RFC8095], which surveys a large number of transports, and [TAPS2], which identifies the specific transport features that are exposed to applications by the protocols TCP, MPTCP, UDP(-Lite) and SCTP as well as the LEDBAT congestion control mechanism. The present draft is based on these documents and follows the same terminology (also listed below).

The number of transport features of current IETF transports is large, and exposing all of them has a number of disadvantages: generally, the more functionality is exposed, the less freedom a TAPS system has to automate usage of the various functions of its available set of transport protocols. Some functions only exist in one particular protocol, and if an application would use them, this would statically tie the application to this protocol, counteracting the purpose of a TAPS system. Also, if the number of exposed features is exceedingly large, a TAPS system might become very hard to use for an application programmer. Taking [TAPS2] as a basis, this document therefore develops a minimal set of transport features, removing the ones that could be harmful to the purpose of a TAPS system but keeping the ones that must be retained for applications to benefit from useful transport functionality.

Applications use a wide variety of APIs today. The transport features in the minimal set in this document must be reflected in **all** network APIs in order for the underlying functionality to become usable everywhere. For example, it does not help an application that talks to a middleware if only the Berkeley Sockets API is extended to offer "unordered message delivery", but the middleware only offers an ordered bytestream. Both the Berkeley Sockets API and the middleware would have to expose the "unordered message delivery" transport feature (alternatively, there may be

interesting ways for certain types of middleware to use some transport features without exposing them, based on knowledge about the applications -- but this is not the general case). In most situations, in the interest of being as flexible and efficient as possible, the best choice will be for a middleware or library to expose at least all of the transport features that are recommended as a "minimal set" here.

This "minimal set" can be implemented one-sided with a fall-back to TCP: i.e., a sender-side TAPS system can talk to a non-TAPS TCP receiver, and a receiver-side TAPS system can talk to a non-TAPS TCP sender. For systems that do not have this requirement, [I-D.trammell-taps-post-sockets] describes a way to extend the functionality of the minimal set such that several of its limitations are removed.

2. Terminology

The following terms are used throughout this document, and in subsequent documents produced by TAPS that describe the composition and decomposition of transport services.

Transport Feature: a specific end-to-end feature that the transport layer provides to an application. Examples include confidentiality, reliable delivery, ordered delivery, message-versus-stream orientation, etc.

Transport Service: a set of Transport Features, without an association to any given framing protocol, which provides a complete service to an application.

Transport Protocol: an implementation that provides one or more different transport services using a specific framing and header format on the wire.

Transport Service Instance: an arrangement of transport protocols with a selected set of features and configuration parameters that implements a single transport service, e.g., a protocol stack (RTP over UDP).

Application: an entity that uses the transport layer for end-to-end delivery data across the network (this may also be an upper layer protocol or tunnel encapsulation).

Application-specific knowledge: knowledge that only applications have.

Endpoint: an entity that communicates with one or more other endpoints using a transport protocol.

Connection: shared state of two or more endpoints that persists across messages that are transmitted between these endpoints.

Socket: the combination of a destination IP address and a destination port number.

3. The Minimal Set of Transport Features

Based on the categorization, reduction and discussion in Appendix A, this section describes the minimal set of transport features that is offered by end systems supporting TAPS.

3.1. Flow Creation, Connection and Termination

A TAPS flow must be "created" before it is connected, to allow for initial configurations to be carried out. All configuration parameters in Section 3.2 and Section 3.3 can be used initially, although some of them may only take effect when the flow has been connected. Configuring a flow early helps a TAPS system make the right decisions. In particular, the "group number" can influence the TAPS system to implement a TAPS flow as a stream of a multi-streaming protocol's existing association or not.

A created flow can be queried for the maximum amount of data that an application can possibly expect to have transmitted before or during connection establishment. An application can also give the flow a message for transmission before or during connection establishment; the TAPS system will try to transmit it as early as possible. An application can facilitate sending the message particularly early by marking it as "idempotent"; in this case, the receiving application must be prepared to potentially receive multiple copies of the message.

To be compatible with multiple transports, including streams of a multi-streaming protocol (used as if they were transports themselves), the semantics of opening and closing need to be the most restrictive subset of all of them. For example, TCP's support of half-closed connections can be seen as a feature on top of the more restrictive "ABORT"; this feature cannot be supported because not all protocols used by a TAPS system (including streams of an association) support half-closed connections.

After creation, a flow can be actively connected to the other side using "Connect", or passively listen for incoming connection requests with "Listen". Note that "Connect" may or may not trigger a notification on the listening side. It is possible that the first notification on the listening side is the arrival of the first data that the active side sends (a receiver-side TAPS system could handle

this by continuing a blocking "Listen" call, immediately followed by issuing "Receive", for example). This also means that the active opening side is assumed to be the first side sending data.

A TAPS system can actively close a connection, i.e. terminate it after reliably delivering all remaining data to the peer, or it can abort it, i.e. terminate it without delivering remaining data. Unless all data transfers only used unreliable frame transmission without congestion control, closing a connection is guaranteed to cause an event to notify the peer application that the connection has been closed. Similarly, for anything but unreliable non-congestion-controlled data transfer, aborting a connection will cause an event to notify the peer application that the connection has been aborted. A timeout can be configured to abort a flow when data could not be delivered for too long; timeout-based abortion does not notify the peer application that the connection has been aborted. Because half-closed connections are not supported, when a TAPS host receives a notification that the peer is closing or aborting the flow, the other side may not be able to read outstanding data. This means that unacknowledged data residing in the TAPS system's send buffer may have to be dropped from that buffer upon arrival of a notification to close or abort the flow from the peer.

3.2. Flow Group Configuration

A flow group can be configured with a number of transport features, and there are some notifications to applications about a flow group. Here we list transport features and notifications from Appendix A.2 that sometimes automatically apply to groups of flows (e.g., when a flow is mapped to a stream of a multi-streaming protocol).

Timeout, error notifications:

- o Change timeout for aborting connection (using retransmit limit or time value)
- o Suggest timeout to the peer
- o Notification of Excessive Retransmissions (early warning below abortion threshold)
- o Notification of ICMP error message arrival

Others:

- o Choose a scheduler to operate between flows of a group
- o Obtain ECN field

The following transport features are new or changed, based on the discussion in Appendix A.3:

- o Capacity profile
This describes how an application wants to use its available capacity. Choices can be "lowest possible latency at the expense

of overhead" (which would disable any Nagle-like algorithm), "scavenger", and some more values that help determine the DSCP value for a flow (e.g. similar to table 1 in [I-D.ietf-tsvwg-rtcweb-qos]).

3.3. Flow Configuration

Here we list transport features and notifications from Appendix A.2 that only apply to a single flow.

Configure priority or weight for a scheduler

Checksums:

- o Disable checksum when sending
- o Disable checksum requirement when receiving
- o Specify checksum coverage used by the sender
- o Specify minimum checksum coverage required by receiver

3.4. Data Transfer

3.4.1. The Sender

This section discusses how to send data after flow establishment. Section 3.1 discusses the possibility to hand over a message to send before or during establishment.

Here we list per-frame properties that a sender can optionally configure if it hands over a delimited frame for sending with congestion control, taken from Appendix A.2:

- o Configurable Message Reliability
- o Choice between unordered (potentially faster) or ordered delivery of messages
- o Request not to bundle messages
- o Request not to delay the acknowledgement (SACK) of a message

Additionally, an application can hand over delimited frames for unreliable transmission without congestion control (note that such applications should perform congestion control in accordance with [RFC2914]). Then, none of the per-frame properties listed above have any effect, but it is possible to use the transport feature "Specify DF field" to allow/disallow fragmentation.

Following Appendix A.3.7, there are three transport features (two old, one new) and a notification:

- o Get max. transport frame size that may be sent without fragmentation from the configured interface
- This is optional for a TAPS system to offer. It can aid

applications implementing Path MTU Discovery.

- o Get max. transport frame size that may be received from the configured interface
This is optional for a TAPS system to offer.
- o Get maximum transport frame size
Irrespective of fragmentation, there is a size limit for the messages that can be handed over to SCTP or UDP(-Lite); because a TAPS system is independent of the transport, it must allow a TAPS application to query this value -- the maximum size of a frame in an Application-Framed-Bytestream.

There are two more sender-side notifications. These are unreliable, i.e. a TAPS system cannot be assumed to implement them, but they may occur:

- o Notification of send failures
A TAPS system may inform a sender application of a failure to send a specific frame. This was taken over unchanged from Appendix A.2.
- o Notification of draining below a low water mark
A TAPS system can notify a sender application when the TAPS system's filling level of the buffer of unsent data is below a configurable threshold in bytes. Even for TAPS systems that do implement this notification, supporting thresholds other than 0 is optional.

"Notification of draining below a low water mark" is a generic notification that tries to enable uniform access to "TCP_NOTSENT_LOWAT" as well as the "SENDER DRY" notification (as discussed in Appendix A.3.4 -- SCTP's "SENDER DRY" is a special case where the threshold (for unsent data) is 0 and there is also no more unacknowledged data in the send buffer). Note that this threshold and its notification should operate across the buffers of the whole TAPS system, i.e. also any potential buffers that the TAPS system itself may use on top of the transport's send buffer.

3.4.2. The Receiver

A receiving application obtains an Application-Framed Bytestream. Similar to TCP's receiver semantics, it is just stream of bytes. If frame boundaries were specified by the sender, a receiver-side TAPS system will still not inform the receiving application about them. Within the bytestream, frames themselves will always stay intact (partial frames are not supported - see Appendix A.3.1). Different from TCP's semantics, there is no guarantee that all frames in the

bytestream are transmitted from the sender to the receiver, and that all of them are in the same sequence in which they were handed over by the sender. If an application is aware of frame delimiters in the bytestream, and if the sender-side application has informed the TAPS system about these boundaries and about potentially relaxed requirements regarding the sequence of frames or per-frame reliability, frames within the receiver-side bytestream may be out-of-order or missing.

4. An Abstract MinSet API

Here we present an abstract API that a TAPS system can implement. This API is derived from the description in the previous section. The primitives of this API can be implemented in various ways. For example, information that is provided to an application can either be offered via a primitive that is polled, or via an asynchronous notification. The API offers specific primitives to configure such asynchronous call-backs.

CREATE (flow-group-id)
Returns: flow-id

Create a flow and associate it with an existing or new flow group number. The group number can influence the TAPS system to implement a TAPS flow as a stream of a multi-streaming protocol's existing association or not.

CONFIGURE_TIMEOUT (flow-group-id [timeout] [peer_timeout]
[retrans_notify])

This configures timeouts for all flows in a group. Configuration should generally be carried out as early as possible, ideally before flows are connected, to aid the TAPS system's decision taking.

PARAMETERS:

timeout: a timeout value for aborting connections, in seconds
peer_timeout: a timeout value to be suggested to the peer (if possible), in seconds
retrans_notify: the number of retransmissions after which the application should be notified of "Excessive Retransmissions"

CONFIGURE_CHECKSUM (flow-id [send [send_length]] [receive
[receive_length]])

This configures the usage of checksums for a flow in a group.

Configuration should generally be carried out as early as possible, ideally before the flow is connected, to aid the TAPS system's decision taking. "send" parameters concern using a checksum when sending, "receive" parameters concern requiring a checksum when receiving. There is no guarantee that any checksum limitations will indeed be enforced; all defaults are: "full coverage, checksum enabled".

PARAMETERS:

send: boolean, enable / disable usage of a checksum
send_length: if send is true, this optional parameter can provide the desired coverage of the checksum in bytes
receive: boolean, enable / disable requiring a checksum
receive_length: if receive is true, this optional parameter can provide the required minimum coverage of the checksum in bytes

CONFIGURE_URGENCY (flow-group-id [scheduler] [capacity_profile] [low_watermark])

This carries out configuration related to the urgency of sending data on flows of a group. Configuration should generally be carried out as early as possible, ideally before flows are connected, to aid the TAPS system's decision taking.

PARAMETERS:

scheduler: a number to identify the type of scheduler that should be used to operate between flows in the group (no guarantees given). Future versions of this document will be self contained, but for now we suggest the schedulers defined in [I-D.ietf-tsvwg-sctp-ndata].
capacity_profile: a number to identify how an application wants to use its available capacity. Future versions of this document will be self contained, but for now choices can be "lowest possible latency at the expense of overhead" (which would disable any Nagle-like algorithm), "scavenger", and some more values that help determine the DSCP value for a flow (e.g. similar to table 1 in [I-D.ietf-tsvwg-rtweb-qos]).
low_watermark: a buffer limit (in bytes); when the sender has less than low_watermark bytes in the buffer, the application may be notified. Notifications are not guaranteed, and supporting watermark numbers greater than 0 is not guaranteed.

CONFIGURE_PRIORITY (flow-id priority)

This configures a flow's priority or weight for a scheduler. Configuration should generally be carried out as early as possible,

ideally before flows are connected, to aid the TAPS system's decision taking.

PARAMETERS:

priority: future versions of this document will be self contained, but for now we suggest the priority as described in [I-D.ietf-tsvwg-sctp-ndata].

NOTIFICATIONS

Returns: flow-group-id notification_type

This is fired when an event occurs, notifying the application about something happening in relation to a flow group. Notification types are:

Excessive Retransmissions: the configured (or a default) number of retransmissions has been reached, yielding this early warning below an abortion threshold

ICMP Arrival (parameter: ICMP message): an ICMP packet carrying the conveyed ICMP message has arrived.

ECN Arrival (parameter: ECN value): a packet carrying the conveyed ECN value has arrived. This can be useful for applications implementing congestion control.

Timeout (parameter: s seconds): data could not be delivered for s seconds.

Close: the peer has closed the connection. The peer has no more data to send, and will not read more data. Data that is in transit or resides in the local send buffer will be discarded.

Abort: the peer has aborted the connection. The peer has no more data to send, and will not read more data. Data that is in transit or resides in the local send buffer will be discarded.

Drain: the send buffer has either drained below the configured low water mark or it has become completely empty.

Path Change (parameter: path identifier): the path has changed; the path identifier is a number that can be used to determine a previously used path is used again (e.g., the TAPS system has switched from one interface to the other and back).

Send Failure (parameter: frame identifier): this informs the application of a failure to send a specific frame. There can be a send failure without this notification happening.

QUERY_PROPERTIES (flow-group-id property_identifier)

Returns: requested property (see below)

This allows to query some properties of a flow group. Return values per property identifier are:

- o The maximum frame size that may be sent without fragmentation, in bytes
- o The maximum transport frame size that can be sent, in bytes
- o The maximum transport frame size that can be received, in bytes
- o The maximum amount of data that can possibly be sent before or during connection establishment, in bytes

CONNECT (flow-id dst_addr)

Connects a flow. This primitive may or may not trigger a notification (continuing LISTEN) on the listening side. If a send precedes this call, then data may be transmitted with this connect.

PARAMETERS:

dst_addr: the destination transport address to connect to

LISTEN (flow-id)

Blocking passive connect, listening on all interfaces. This may not be the direct result of the peer calling CONNECT - it may also be invoked upon reception of the first block of data. In this case, RECEIVE_FRAME is invoked immediately after.

SEND_FRAME (flow-id frame [reliability] [ordered] [bundle] [delack] [fragment] [idempotent])

Sends an application frame. No guarantees are given about the preservation of frame boundaries to the peer; if frame boundaries are needed, the receiving application at the peer must know about them beforehand. Note that this call can already be used before a flow is connected. All parameters refer to the frame that is being handed over.

PARAMETERS:

reliability: this parameter is used to convey a choice of: fully reliable, unreliable without congestion control (which is guaranteed), unreliable, partially reliable (how to configure: TBD, probably using a time value). The latter two choices are not guaranteed and may result in full reliability.

ordered: this boolean parameter lets an application choose between ordered message delivery (true) and possibly unordered, potentially faster message delivery (false).

bundle: a boolean that expresses a preference for allowing to bundle frames (true) or not (false). No guarantees are given.

delack: a boolean that, if false, lets an application request that the peer would not delay the acknowledgement for this frame.

fragment: a boolean that expresses a preference for allowing to fragment frames (true) or not (false), at the IP level. No guarantees are given.

idempotent: a boolean that expresses whether a frame is idempotent (true) or not (false). Idempotent frames may arrive multiple times at the receiver. When data is idempotent it can be used by the receiver immediately on a connection establishment attempt. Thus, if SEND_FRAME is used before connecting, stating that a frame is idempotent facilitates transmitting it to the peer application particularly early.

CLOSE (flow-id)

Closes the flow after all outstanding data is reliably delivered to the peer (if reliable data delivery was requested). In case reliable or partially reliable data delivery was requested earlier, the peer is notified of the CLOSE.

ABORT (flow-id)

Aborts the flow without delivering outstanding data to the peer. In case reliable or partially reliable data delivery was requested earlier, the peer is notified of the ABORT.

RECEIVE_FRAME (flow-id buffer)

This receives a block of data. This block may or may not correspond to a sender-side frame, i.e. the receiving application is not informed about frame boundaries. However, if the sending application has allowed that frames are not fully reliably transferred, or delivered out of order, then such re-ordering or unreliability may be reflected per frame in the arriving data. Frames will always stay intact - i.e. if an incomplete frame is contained at the end of the arriving data block, this frame is guaranteed to continue in the next arriving data block.

PARAMETERS:

buffer: the buffer where the received data will be stored.

5. Conclusion

By decoupling applications from transport protocols, a TAPS system provides a different abstraction level than the Berkeley sockets interface. As with high- vs. low-level programming languages, a higher abstraction level allows more freedom for automation below the interface, yet it takes some control away from the application programmer. This is the design trade-off that a TAPS system developer is facing, and this document provides guidance on the design of this abstraction level. Some transport features are currently rarely offered by APIs, yet they must be offered or they can never be used ("functional" transport features). Other transport features are offered by the APIs of the protocols covered here, but not exposing them in a TAPS API would allow for more freedom to automate protocol usage in a TAPS system.

The minimal set presented in this document is an effort to find a middle ground that can be recommended for TAPS systems to implement, on the basis of the transport features discussed in [TAPS2]. This middle ground eliminates a large number of transport features because they do not require application-specific knowledge, but rather rely on knowledge about the network or the Operating System. This leaves us with an unanswered question about how exactly a TAPS system should automate using all these transport features.

In some cases, it may be best to not entirely automate the decision making, but leave it up to a system-wide policy. For example, when multiple paths are available, a system policy could guide the decision on whether to connect via a WiFi or a cellular interface. Such high-level guidance could also be provided by application developers, e.g. via a primitive that lets applications specify such preferences. As long as this kind of information from applications is treated as advisory, it will not lead to a permanent protocol binding and does therefore not limit the flexibility of a TAPS system. Decisions to add such primitives are therefore left open to TAPS system designers.

6. Acknowledgements

The authors would like to thank the participants of the TAPS Working Group and the NEAT research project for valuable input to this document. We especially thank Michael Tuexen for help with TAPS flow connection establishment/teardown and Gorry Fairhurst for his

suggestions regarding fragmentation and packet sizes. This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 644334 (NEAT). The views expressed are solely those of the author(s).

7. IANA Considerations

XX RFC ED - PLEASE REMOVE THIS SECTION XXX

This memo includes no request to IANA.

8. Security Considerations

Authentication, confidentiality protection, and integrity protection are identified as transport features by [RFC8095]. As currently deployed in the Internet, these features are generally provided by a protocol or layer on top of the transport protocol; no current full-featured standards-track transport protocol provides all of these transport features on its own. Therefore, these transport features are not considered in this document, with the exception of native authentication capabilities of TCP and SCTP for which the security considerations in [RFC5925] and [RFC4895] apply.

9. References

9.1. Normative References

[RFC8095] Fairhurst, G., Ed., Trammell, B., Ed., and M. Kuehlewind, Ed., "Services Provided by IETF Transport Protocols and Congestion Control Mechanisms", RFC 8095, DOI 10.17487/RFC8095, March 2017, <<http://www.rfc-editor.org/info/rfc8095>>.

[TAPS2] Welzl, M., Tuexen, M., and N. Khademi, "On the Usage of Transport Features Provided by IETF Transport Protocols", draft-ietf-taps-transport-services-usage-05 (work in progress), May 2017.

9.2. Informative References

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

[I-D.ietf-tsvwg-rtcweb-qos]

Jones, P., Dhesikan, S., Jennings, C., and D. Druta, "DSCP Packet Markings for WebRTC QoS", draft-ietf-tsvwg-rtcweb-qos-18 (work in progress), August 2016.

[I-D.ietf-tsvwg-sctp-ndata]

Stewart, R., Tuexen, M., Loreto, S., and R. Seggelmann, "Stream Schedulers and User Message Interleaving for the Stream Control Transmission Protocol", draft-ietf-tsvwg-sctp-ndata-10 (work in progress), April 2017.

[I-D.trammell-taps-post-sockets]

Trammell, B., Perkins, C., Pauly, T., and M. Kuehlewind, "Post Sockets, An Abstract Programming Interface for the Transport Layer", draft-trammell-taps-post-sockets-00 (work in progress), March 2017.

[LBE-draft]

Bless, R., "A Lower Effort Per-Hop Behavior (LE PHB)", draft-tsvwg-le-phb-01 (work in progress), February 2017.

[RFC2914] Floyd, S., "Congestion Control Principles", BCP 41, RFC 2914, DOI 10.17487/RFC2914, September 2000, <<http://www.rfc-editor.org/info/rfc2914>>.

[RFC4895] Tuexen, M., Stewart, R., Lei, P., and E. Rescorla, "Authenticated Chunks for the Stream Control Transmission Protocol (SCTP)", RFC 4895, DOI 10.17487/RFC4895, August 2007, <<http://www.rfc-editor.org/info/rfc4895>>.

[RFC4987] Eddy, W., "TCP SYN Flooding Attacks and Common Mitigations", RFC 4987, DOI 10.17487/RFC4987, August 2007, <<http://www.rfc-editor.org/info/rfc4987>>.

[RFC5925] Touch, J., Mankin, A., and R. Bonica, "The TCP Authentication Option", RFC 5925, DOI 10.17487/RFC5925, June 2010, <<http://www.rfc-editor.org/info/rfc5925>>.

[RFC6458] Stewart, R., Tuexen, M., Poon, K., Lei, P., and V. Yasevich, "Sockets API Extensions for the Stream Control Transmission Protocol (SCTP)", RFC 6458, DOI 10.17487/RFC6458, December 2011, <<http://www.rfc-editor.org/info/rfc6458>>.

[RFC6525] Stewart, R., Tuexen, M., and P. Lei, "Stream Control Transmission Protocol (SCTP) Stream Reconfiguration", RFC 6525, DOI 10.17487/RFC6525, February 2012,

<<http://www.rfc-editor.org/info/rfc6525>>.

[RFC7413] Cheng, Y., Chu, J., Radhakrishnan, S., and A. Jain, "TCP Fast Open", RFC 7413, DOI 10.17487/RFC7413, December 2014, <<http://www.rfc-editor.org/info/rfc7413>>.

[WWDC2015] Lakhera, P. and S. Cheshire, "Your App and Next Generation Networks", Apple Worldwide Developers Conference 2015, San Francisco, USA, June 2015, <<https://developer.apple.com/videos/wwdc/2015/?id=719>>.

Appendix A. Deriving the minimal set

We approach the construction of a minimal set of transport features in the following way:

1. Categorization: the superset of transport features from [TAPS2] is presented, and transport features are categorized for later reduction.
2. Reduction: a shorter list of transport features is derived from the categorization in the first step. This removes all transport features that do not require application-specific knowledge or cannot be implemented with TCP.
3. Discussion: the resulting list shows a number of peculiarities that are discussed, to provide a basis for constructing the minimal set.
4. Construction: Based on the reduced set and the discussion of the transport features therein, a minimal set is constructed.

The first three steps as well as the underlying rationale for constructing the minimal set are described in this appendix. The minimal set itself is described in Section 3.

A.1. Step 1: Categorization -- The Superset of Transport Features

Following [TAPS2], we divide the transport features into two main groups as follows:

1. CONNECTION related transport features
 - ESTABLISHMENT
 - AVAILABILITY
 - MAINTENANCE
 - TERMINATION
2. DATA Transfer Related transport features
 - Sending Data
 - Receiving Data
 - Errors

We assume that TAPS applications have no specific requirements that

need knowledge about the network, e.g. regarding the choice of network interface or the end-to-end path. Even with these assumptions, there are certain requirements that are strictly kept by transport protocols today, and these must also be kept by a TAPS system. Some of these requirements relate to transport features that we call "Functional".

Functional transport features provide functionality that cannot be used without the application knowing about them, or else they violate assumptions that might cause the application to fail. For example, unordered message delivery is a functional transport feature: it cannot be used without the application knowing about it because the application's assumption could be that messages arrive in order. Failure includes any change of the application behavior that is not performance oriented, e.g. security.

"Change DSCP" and "Disable Nagle algorithm" are examples of transport features that we call "Optimizing": if a TAPS system autonomously decides to enable or disable them, an application will not fail, but a TAPS system may be able to communicate more efficiently if the application is in control of this optimizing transport feature. These transport features require application-specific knowledge (e.g., about delay/bandwidth requirements or the length of future data blocks that are to be transmitted).

The transport features of IETF transport protocols that do not require application-specific knowledge and could therefore be transparently utilized by a TAPS system are called "Automatable".

Finally, some transport features are aggregated and/or slightly changed in the TAPS API. These transport features are marked as "ADDED". The corresponding transport features are automatable, and they are listed immediately below the "ADDED" transport feature.

In this description, transport services are presented following the nomenclature "CATEGORY.[SUBCATEGORY].SERVICENAME.PROTOCOL", equivalent to "pass 2" in [TAPS2]. The PROTOCOL name "UDP(-Lite)" is used when transport features are equivalent for UDP and UDP-Lite; the PROTOCOL name "TCP" refers to both TCP and MPTCP. We also sketch how some of the TAPS transport services can be implemented. For all transport features that are categorized as "functional" or "optimizing", and for which no matching TCP primitive exists in "pass 2" of [TAPS2], a brief discussion on how to fall back to TCP is included.

We designate some transport features as "automatable" on the basis of a broader decision that affects multiple transport features:

- o Most transport features that are related to multi-streaming were designated as "automatable". This was done because the decision on whether to use multi-streaming or not does not depend on application-specific knowledge. This means that a connection that is exhibited to an application could be implemented by using a single stream of an SCTP association instead of mapping it to a complete SCTP association or TCP connection. This could be achieved by using more than one stream when an SCTP association is first established (CONNECT.SCTP parameter "outbound stream count"), maintaining an internal stream number, and using this stream number when sending data (SEND.SCTP parameter "stream number"). Closing or aborting a connection could then simply free the stream number for future use. This is discussed further in Appendix A.3.2.
- o All transport features that are related to using multiple paths or the choice of the network interface were designated as "automatable". Choosing a path or an interface does not depend on application-specific knowledge. For example, "Listen" could always listen on all available interfaces and "Connect" could use the default interface for the destination IP address.

A.1.1. CONNECTION Related Transport Features

ESTABLISHMENT:

- o Connect
Protocols: TCP, SCTP, UDP(-Lite)
Functional because the notion of a connection is often reflected in applications as an expectation to be able to communicate after a "Connect" succeeded, with a communication sequence relating to this transport feature that is defined by the application protocol.
Implementation: via CONNECT.TCP, CONNECT.SCTP or CONNECT.UDP(-Lite).
- o Specify which IP Options must always be used
Protocols: TCP, UDP(-Lite)
Automatable because IP Options relate to knowledge about the network, not the application.
- o Request multiple streams
Protocols: SCTP
Automatable because using multi-streaming does not require application-specific knowledge.
Implementation: see Appendix A.3.2.
- o Limit the number of inbound streams
Protocols: SCTP
Automatable because using multi-streaming does not require

application-specific knowledge.
Implementation: see Appendix A.3.2.

- o Specify number of attempts and/or timeout for the first establishment message
Protocols: TCP, SCTP
Functional because this is closely related to potentially assumed reliable data delivery for data that is sent before or during connection establishment.
Implementation: Using a parameter of CONNECT.TCP and CONNECT.SCTP.
- o Obtain multiple sockets
Protocols: SCTP
Automatable because the usage of multiple paths to communicate to the same end host relates to knowledge about the network, not the application.
- o Disable MPTCP
Protocols: MPTCP
Automatable because the usage of multiple paths to communicate to the same end host relates to knowledge about the network, not the application.
Implementation: via a boolean parameter in CONNECT.MPTCP.
Fall-back to TCP: Do nothing.
- o Configure authentication
Protocols: TCP, SCTP
Functional because this has a direct influence on security.
Implementation: via parameters in CONNECT.TCP and CONNECT.SCTP.
Fall-back to TCP: With TCP, this allows to configure Master Key Tuples (MKTs) to authenticate complete segments (including the TCP IPv4 pseudoheader, TCP header, and TCP data). With SCTP, this allows to specify which chunk types must always be authenticated. Authenticating only certain chunk types creates a reduced level of security that is not supported by TCP; to be compatible, this should therefore only allow to authenticate all chunk types. Key material must be provided in a way that is compatible with both [RFC4895] and [RFC5925].
- o Indicate (and/or obtain upon completion) an Adaptation Layer via an adaptation code point
Protocols: SCTP
Functional because it allows to send extra data for the sake of identifying an adaptation layer, which by itself is application-specific.
Implementation: via a parameter in CONNECT.SCTP.
Fall-back to TCP: not possible.

- o Request to negotiate interleaving of user messages
Protocols: SCTP
Automatable because it requires using multiple streams, but requesting multiple streams in the CONNECTION.ESTABLISHMENT category is automatable.
Implementation: via a parameter in CONNECT.SCTP.
- o Hand over a message to transfer (possibly multiple times) before connection establishment
Protocols: TCP
Functional because this is closely tied to properties of the data that an application sends or expects to receive.
Implementation: via a parameter in CONNECT.TCP.
- o Hand over a message to transfer during connection establishment
Protocols: SCTP
Functional because this can only work if the message is limited in size, making it closely tied to properties of the data that an application sends or expects to receive.
Implementation: via a parameter in CONNECT.SCTP.
- o Enable UDP encapsulation with a specified remote UDP port number
Protocols: SCTP
Automatable because UDP encapsulation relates to knowledge about the network, not the application.

AVAILABILITY:

- o Listen
Protocols: TCP, SCTP, UDP(-Lite)
Functional because the notion of accepting connection requests is often reflected in applications as an expectation to be able to communicate after a "Listen" succeeded, with a communication sequence relating to this transport feature that is defined by the application protocol.
ADDED. This differs from the 3 automatable transport features below in that it leaves the choice of interfaces for listening open.
Implementation: by listening on all interfaces via LISTEN.TCP (not providing a local IP address) or LISTEN.SCTP (providing SCTP port number / address pairs for all local IP addresses).
- o Listen, 1 specified local interface
Protocols: TCP, SCTP, UDP(-Lite)
Automatable because decisions about local interfaces relate to knowledge about the network and the Operating System, not the application.

- o Listen, N specified local interfaces
Protocols: SCTP
Automatable because decisions about local interfaces relate to knowledge about the network and the Operating System, not the application.
- o Listen, all local interfaces
Protocols: TCP, SCTP, UDP(-Lite)
Automatable because decisions about local interfaces relate to knowledge about the network and the Operating System, not the application.
- o Specify which IP Options must always be used
Protocols: TCP, UDP(-Lite)
Automatable because IP Options relate to knowledge about the network, not the application.
- o Disable MPTCP
Protocols: MPTCP
Automatable because the usage of multiple paths to communicate to the same end host relates to knowledge about the network, not the application.
- o Configure authentication
Protocols: TCP, SCTP
Functional because this has a direct influence on security.
Implementation: via parameters in LISTEN.TCP and LISTEN.SCTP.
Fall-back to TCP: With TCP, this allows to configure Master Key Tuples (MKTs) to authenticate complete segments (including the TCP IPv4 pseudoheader, TCP header, and TCP data). With SCTP, this allows to specify which chunk types must always be authenticated. Authenticating only certain chunk types creates a reduced level of security that is not supported by TCP; to be compatible, this should therefore only allow to authenticate all chunk types. Key material must be provided in a way that is compatible with both [RFC4895] and [RFC5925].
- o Obtain requested number of streams
Protocols: SCTP
Automatable because using multi-streaming does not require application-specific knowledge.
Implementation: see Appendix A.3.2.
- o Limit the number of inbound streams
Protocols: SCTP
Automatable because using multi-streaming does not require application-specific knowledge.
Implementation: see Appendix A.3.2.

- o Indicate (and/or obtain upon completion) an Adaptation Layer via an adaptation code point
Protocols: SCTP
Functional because it allows to send extra data for the sake of identifying an adaptation layer, which by itself is application-specific.
Implementation: via a parameter in LISTEN.SCTP.
Fall-back to TCP: not possible.
- o Request to negotiate interleaving of user messages
Protocols: SCTP
Automatable because it requires using multiple streams, but requesting multiple streams in the CONNECTION.ESTABLISHMENT category is automatable.
Implementation: via a parameter in LISTEN.SCTP.

MAINTENANCE:

- o Change timeout for aborting connection (using retransmit limit or time value)
Protocols: TCP, SCTP
Functional because this is closely related to potentially assumed reliable data delivery.
Implementation: via CHANGE-TIMEOUT.TCP or CHANGE-TIMEOUT.SCTP.
- o Suggest timeout to the peer
Protocols: TCP
Functional because this is closely related to potentially assumed reliable data delivery.
Implementation: via CHANGE-TIMEOUT.TCP.
- o Disable Nagle algorithm
Protocols: TCP, SCTP
Optimizing because this decision depends on knowledge about the size of future data blocks and the delay between them.
Implementation: via DISABLE-NAGLE.TCP and DISABLE-NAGLE.SCTP.
- o Request an immediate heartbeat, returning success/failure
Protocols: SCTP
Automatable because this informs about network-specific knowledge.
- o Notification of Excessive Retransmissions (early warning below abortion threshold)
Protocols: TCP
Optimizing because it is an early warning to the application, informing it of an impending functional event.
Implementation: via ERROR.TCP.

- o Add path
Protocols: MPTCP, SCTP
MPTCP Parameters: source-IP; source-Port; destination-IP;
destination-Port
SCTP Parameters: local IP address
Automatable because the usage of multiple paths to communicate to the same end host relates to knowledge about the network, not the application.
- o Remove path
Protocols: MPTCP, SCTP
MPTCP Parameters: source-IP; source-Port; destination-IP;
destination-Port
SCTP Parameters: local IP address
Automatable because the usage of multiple paths to communicate to the same end host relates to knowledge about the network, not the application.
- o Set primary path
Protocols: SCTP
Automatable because the usage of multiple paths to communicate to the same end host relates to knowledge about the network, not the application.
- o Suggest primary path to the peer
Protocols: SCTP
Automatable because the usage of multiple paths to communicate to the same end host relates to knowledge about the network, not the application.
- o Configure Path Switchover
Protocols: SCTP
Automatable because the usage of multiple paths to communicate to the same end host relates to knowledge about the network, not the application.
- o Obtain status (query or notification)
Protocols: SCTP, MPTCP
SCTP parameters: association connection state; destination transport address list; destination transport address reachability states; current local and peer receiver window size; current local congestion window sizes; number of unacknowledged DATA chunks; number of DATA chunks pending receipt; primary path; most recent SRTT on primary path; RTO on primary path; SRTT and RTO on other destination addresses; MTU per path; interleaving supported yes/no
MPTCP parameters: subflow-list (identified by source-IP; source-Port; destination-IP; destination-Port)
Automatable because these parameters relate to knowledge about the

network, not the application.

- o Specify DSCP field
Protocols: TCP, SCTP, UDP(-Lite)
Optimizing because choosing a suitable DSCP value requires application-specific knowledge.
Implementation: via SET_DSCP.TCP / SET_DSCP.SCTP / SET_DSCP.UDP(-Lite)
- o Notification of ICMP error message arrival
Protocols: TCP, UDP(-Lite)
Optimizing because these messages can inform about success or failure of functional transport features (e.g., host unreachable relates to "Connect")
Implementation: via ERROR.TCP or ERROR.UDP(-Lite).
- o Obtain information about interleaving support
Protocols: SCTP
Automatable because it requires using multiple streams, but requesting multiple streams in the CONNECTION.ESTABLISHMENT category is automatable.
Implementation: via a parameter in GETINTERL.SCTP.
- o Change authentication parameters
Protocols: TCP, SCTP
Functional because this has a direct influence on security.
Implementation: via SET_AUTH.TCP and SET_AUTH.SCTP.
Fall-back to TCP: With SCTP, this allows to adjust key_id, key, and hmac_id. With TCP, this allows to change the preferred outgoing MKT (current_key) and the preferred incoming MKT (rnext_key), respectively, for a segment that is sent on the connection. Key material must be provided in a way that is compatible with both [RFC4895] and [RFC5925].
- o Obtain authentication information
Protocols: SCTP
Functional because authentication decisions may have been made by the peer, and this has an influence on the necessary application-level measures to provide a certain level of security.
Implementation: via GETAUTH.SCTP.
Fall-back to TCP: With SCTP, this allows to obtain key_id and a chunk list. With TCP, this allows to obtain current_key and rnext_key from a previously received segment. Key material must be provided in a way that is compatible with both [RFC4895] and [RFC5925].

- o Reset Stream
Protocols: SCTP
Automatable because using multi-streaming does not require application-specific knowledge.
Implementation: see Appendix A.3.2.
- o Notification of Stream Reset
Protocols: STCP
Automatable because using multi-streaming does not require application-specific knowledge.
Implementation: see Appendix A.3.2.
- o Reset Association
Protocols: SCTP
Functional because it affects "Obtain a message delivery number", which is functional.
Implementation: via RESETASSOC.SCTP.
Fall-back to TCP: not possible.
- o Notification of Association Reset
Protocols: STCP
Functional because it affects "Obtain a message delivery number", which is functional.
Implementation: via RESETASSOC-EVENT.SCTP.
Fall-back to TCP: not possible.
- o Add Streams
Protocols: SCTP
Automatable because using multi-streaming does not require application-specific knowledge.
Implementation: see Appendix A.3.2.
- o Notification of Added Stream
Protocols: STCP
Automatable because using multi-streaming does not require application-specific knowledge.
Implementation: see Appendix A.3.2.
- o Choose a scheduler to operate between streams of an association
Protocols: SCTP
Optimizing because the scheduling decision requires application-specific knowledge. However, if a TAPS system would not use this, or wrongly configure it on its own, this would only affect the performance of data transfers; the outcome would still be correct within the "best effort" service model.
Implementation: using SETSTREAMSCHEDULER.SCTP.
Fall-back to TCP: do nothing.

- o Configure priority or weight for a scheduler
Protocols: SCTP
Optimizing because the priority or weight requires application-specific knowledge. However, if a TAPS system would not use this, or wrongly configure it on its own, this would only affect the performance of data transfers; the outcome would still be correct within the "best effort" service model.
Implementation: using CONFIGURESTREAMSCHEDULER.SCTP.
Fall-back to TCP: do nothing.
- o Configure send buffer size
Protocols: SCTP
Automatable because this decision relates to knowledge about the network and the Operating System, not the application (see also the discussion in Appendix A.3.4).
- o Configure receive buffer (and rwnd) size
Protocols: SCTP
Automatable because this decision relates to knowledge about the network and the Operating System, not the application.
- o Configure message fragmentation
Protocols: SCTP
Automatable because fragmentation relates to knowledge about the network and the Operating System, not the application.
Implementation: by always enabling it with CONFIG_FRAGMENTATION.SCTP and auto-setting the fragmentation size based on network or Operating System conditions.
- o Configure PMTUD
Protocols: SCTP
Automatable because Path MTU Discovery relates to knowledge about the network, not the application.
- o Configure delayed SACK timer
Protocols: SCTP
Automatable because the receiver-side decision to delay sending SACKs relates to knowledge about the network, not the application (it can be relevant for a sending application to request not to delay the SACK of a message, but this is a different transport feature).
- o Set Cookie life value
Protocols: SCTP
Functional because it relates to security (possibly weakened by keeping a cookie very long) versus the time between connection establishment attempts. Knowledge about both issues can be application-specific.

Fall-back to TCP: the closest specified TCP functionality is the cookie in TCP Fast Open; for this, [RFC7413] states that the server "can expire the cookie at any time to enhance security" and section 4.1.2 describes an example implementation where updating the key on the server side causes the cookie to expire. Alternatively, for implementations that do not support TCP Fast Open, this transport feature could also affect the validity of SYN cookies (see Section 3.6 of [RFC4987]).

- o Set maximum burst
Protocols: SCTP
Automatable because it relates to knowledge about the network, not the application.
- o Configure size where messages are broken up for partial delivery
Protocols: SCTP
Functional because this is closely tied to properties of the data that an application sends or expects to receive.
Fall-back to TCP: do nothing. Since TCP does not deliver messages, partial or not, this will have no effect on TCP.
- o Disable checksum when sending
Protocols: UDP
Functional because application-specific knowledge is necessary to decide whether it can be acceptable to lose data integrity.
Implementation: via SET_CHECKSUM_ENABLED.UDP.
Fall-back to TCP: do nothing.
- o Disable checksum requirement when receiving
Protocols: UDP
Functional because application-specific knowledge is necessary to decide whether it can be acceptable to lose data integrity.
Implementation: via SET_CHECKSUM_REQUIRED.UDP.
Fall-back to TCP: do nothing.
- o Specify checksum coverage used by the sender
Protocols: UDP-Lite
Functional because application-specific knowledge is necessary to decide for which parts of the data it can be acceptable to lose data integrity.
Implementation: via SET_CHECKSUM_COVERAGE.UDP-Lite.
Fall-back to TCP: do nothing.
- o Specify minimum checksum coverage required by receiver
Protocols: UDP-Lite
Functional because application-specific knowledge is necessary to decide for which parts of the data it can be acceptable to lose data integrity.

Implementation: via SET_MIN_CHECKSUM_COVERAGE.UDP-Lite.
Fall-back to TCP: do nothing.

- o Specify DF field
Protocols: UDP(-Lite)
Optimizing because the DF field can be used to carry out Path MTU Discovery, which can lead an application to choose message sizes that can be transmitted more efficiently.
Implementation: via MAINTENANCE.SET_DF.UDP(-Lite) and SEND_FAILURE.UDP(-Lite).
Fall-back to TCP: do nothing. With TCP the sender is not in control of transport message sizes, making this functionality irrelevant.
- o Get max. transport-message size that may be sent using a non-fragmented IP packet from the configured interface
Protocols: UDP(-Lite)
Optimizing because this can lead an application to choose message sizes that can be transmitted more efficiently.
- o Get max. transport-message size that may be received from the configured interface
Protocols: UDP(-Lite)
Optimizing because this can, for example, influence an application's memory management.
- o Specify TTL/Hop count field
Protocols: UDP(-Lite)
Automatable because a TAPS system can use a large enough system default to avoid communication failures. Allowing an application to configure it differently can produce notifications of ICMP error message arrivals that yield information which only relates to knowledge about the network, not the application.
- o Obtain TTL/Hop count field
Protocols: UDP(-Lite)
Automatable because the TTL/Hop count field relates to knowledge about the network, not the application.
- o Specify ECN field
Protocols: UDP(-Lite)
Automatable because the ECN field relates to knowledge about the network, not the application.
- o Obtain ECN field
Protocols: UDP(-Lite)
Optimizing because this information can be used by an application to better carry out congestion control (this is relevant when

choosing a data transmission transport service that does not already do congestion control).

- o Specify IP Options
Protocols: UDP(-Lite)
Automatable because IP Options relate to knowledge about the network, not the application.
- o Obtain IP Options
Protocols: UDP(-Lite)
Automatable because IP Options relate to knowledge about the network, not the application.
- o Enable and configure a "Low Extra Delay Background Transfer"
Protocols: A protocol implementing the LEDBAT congestion control mechanism
Optimizing because whether this service is appropriate or not depends on application-specific knowledge. However, wrongly using this will only affect the speed of data transfers (albeit including other transfers that may compete with the TAPS transfer in the network), so it is still correct within the "best effort" service model.
Implementation: via CONFIGURE.LEDBAT and/or SET_DSCP.TCP / SET_DSCP.SCTP / SET_DSCP.UDP(-Lite) [LBE-draft].
Fall-back to TCP: do nothing.

TERMINATION:

- o Close after reliably delivering all remaining data, causing an event informing the application on the other side
Protocols: TCP, SCTP
Functional because the notion of a connection is often reflected in applications as an expectation to have all outstanding data delivered and no longer be able to communicate after a "Close" succeeded, with a communication sequence relating to this transport feature that is defined by the application protocol.
Implementation: via CLOSE.TCP and CLOSE.SCTP.
- o Abort without delivering remaining data, causing an event informing the application on the other side
Protocols: TCP, SCTP
Functional because the notion of a connection is often reflected in applications as an expectation to potentially not have all outstanding data delivered and no longer be able to communicate after an "Abort" succeeded. On both sides of a connection, an application protocol may define a communication sequence relating to this transport feature.
Implementation: via ABORT.TCP and ABORT.SCTP.

- o Abort without delivering remaining data, not causing an event informing the application on the other side
Protocols: UDP(-Lite)
Functional because the notion of a connection is often reflected in applications as an expectation to potentially not have all outstanding data delivered and no longer be able to communicate after an "Abort" succeeded. On both sides of a connection, an application protocol may define a communication sequence relating to this transport feature.
Implementation: via ABORT.UDP(-Lite).
Fall-back to TCP: stop using the connection, wait for a timeout.
- o Timeout event when data could not be delivered for too long
Protocols: TCP, SCTP
Functional because this notifies that potentially assumed reliable data delivery is no longer provided.
Implementation: via TIMEOUT.TCP and TIMEOUT.SCTP.

A.1.2. DATA Transfer Related Transport Features

A.1.2.1. Sending Data

- o Reliably transfer data, with congestion control
Protocols: TCP, SCTP
Functional because this is closely tied to properties of the data that an application sends or expects to receive.
Implementation: via SEND.TCP and SEND.SCTP.
- o Reliably transfer a message, with congestion control
Protocols: SCTP
Functional because this is closely tied to properties of the data that an application sends or expects to receive.
Implementation: via SEND.SCTP and SEND.TCP. With SEND.TCP, messages will not be identifiable by the receiver. Inform the application of the result.
- o Unreliably transfer a message
Protocols: SCTP, UDP(-Lite)
Optimizing because only applications know about the time criticality of their communication, and reliably transferring a message is never incorrect for the receiver of a potentially unreliable data transfer, it is just slower.
ADDED. This differs from the 2 automatable transport features below in that it leaves the choice of congestion control open.
Implementation: via SEND.SCTP or SEND.UDP or SEND.TCP. With SEND.TCP, messages will not be identifiable by the receiver. Inform the application of the result.

- o Unreliably transfer a message, with congestion control
Protocols: SCTP
Automatable because congestion control relates to knowledge about the network, not the application.
- o Unreliably transfer a message, without congestion control
Protocols: UDP(-Lite)
Automatable because congestion control relates to knowledge about the network, not the application.
- o Configurable Message Reliability
Protocols: SCTP
Optimizing because only applications know about the time criticality of their communication, and reliably transferring a message is never incorrect for the receiver of a potentially unreliable data transfer, it is just slower.
Implementation: via SEND.SCTP.
Fall-back to TCP: By using SEND.TCP and ignoring this configuration: based on the assumption of the best-effort service model, unnecessarily delivering data does not violate application expectations. Moreover, it is not possible to associate the requested reliability to a "message" in TCP anyway.
- o Choice of stream
Protocols: SCTP
Automatable because it requires using multiple streams, but requesting multiple streams in the CONNECTION.ESTABLISHMENT category is automatable. Implementation: see Appendix A.3.2.
- o Choice of path (destination address)
Protocols: SCTP
Automatable because it requires using multiple sockets, but obtaining multiple sockets in the CONNECTION.ESTABLISHMENT category is automatable.
- o Choice between unordered (potentially faster) or ordered delivery of messages
Protocols: SCTP
Functional because this is closely tied to properties of the data that an application sends or expects to receive.
Implementation: via SEND.SCTP.
Fall-back to TCP: By using SEND.TCP and always sending data ordered: based on the assumption of the best-effort service model, ordered delivery may just be slower and does not violate application expectations. Moreover, it is not possible to associate the requested delivery order to a "message" in TCP anyway.

- o Request not to bundle messages
Protocols: SCTP
Optimizing because this decision depends on knowledge about the size of future data blocks and the delay between them.
Implementation: via SEND.SCTP.
Fall-back to TCP: By using SEND.TCP and DISABLE-NAGLE.TCP to disable the Nagle algorithm when the request is made and enable it again when the request is no longer made. Note that this is not fully equivalent because it relates to the time of issuing the request rather than a specific message.
- o Specifying a "payload protocol-id" (handed over as such by the receiver)
Protocols: SCTP
Functional because it allows to send extra application data with every message, for the sake of identification of data, which by itself is application-specific.
Implementation: SEND.SCTP.
Fall-back to TCP: not possible.
- o Specifying a key id to be used to authenticate a message
Protocols: SCTP
Functional because this has a direct influence on security.
Implementation: via a parameter in SEND.SCTP.
Fall-back to TCP: This could be emulated by using SET_AUTH.TCP before and after the message is sent. Note that this is not fully equivalent because it relates to the time of issuing the request rather than a specific message.
- o Request not to delay the acknowledgement (SACK) of a message
Protocols: SCTP
Optimizing because only an application knows for which message it wants to quickly be informed about success / failure of its delivery.
Fall-back to TCP: do nothing.

A.1.2.2. Receiving Data

- o Receive data (with no message delineation)
Protocols: TCP
Functional because a TAPS system must be able to send and receive data.
Implementation: via RECEIVE.TCP
- o Receive a message
Protocols: SCTP, UDP(-Lite)
Functional because this is closely tied to properties of the data

that an application sends or expects to receive.
Implementation: via RECEIVE.SCTP and RECEIVE.UDP(-Lite).
Fall-back to TCP: not possible.

- o Choice of stream to receive from
Protocols: SCTP
Automatable because it requires using multiple streams, but requesting multiple streams in the CONNECTION.ESTABLISHMENT category is automatable.
Implementation: see Appendix A.3.2.
- o Information about partial message arrival
Protocols: SCTP
Functional because this is closely tied to properties of the data that an application sends or expects to receive.
Implementation: via RECEIVE.SCTP.
Fall-back to TCP: do nothing: this information is not available with TCP.
- o Obtain a message delivery number
Protocols: SCTP
Functional because this number can let applications detect and, if desired, correct reordering. Whether messages are in the correct order or not is closely tied to properties of the data that an application sends or expects to receive.
Implementation: via RECEIVE.SCTP.
Fall-back to TCP: not possible.

A.1.2.3. Errors

This section describes sending failures that are associated with a specific call to in the "Sending Data" category (Appendix A.1.2.1).

- o Notification of send failures
Protocols: SCTP, UDP(-Lite)
Functional because this notifies that potentially assumed reliable data delivery is no longer provided.
ADDED. This differs from the 2 automatable transport features below in that it does not distinguish between unsent and unacknowledged messages.
Implementation: via SENDFAILURE-EVENT.SCTP and SEND_FAILURE.UDP(-Lite).
Fall-back to TCP: do nothing: this notification is not available and will therefore not occur with TCP.

- o Notification of an unsent (part of a) message
Protocols: SCTP, UDP(-Lite)
Automatable because the distinction between unsent and unacknowledged is network-specific.
- o Notification of an unacknowledged (part of a) message
Protocols: SCTP
Automatable because the distinction between unsent and unacknowledged is network-specific.
- o Notification that the stack has no more user data to send
Protocols: SCTP
Optimizing because reacting to this notification requires the application to be involved, and ensuring that the stack does not run dry of data (for too long) can improve performance.
Fall-back to TCP: do nothing. See also the discussion in Appendix A.3.4.
- o Notification to a receiver that a partial message delivery has been aborted
Protocols: SCTP
Functional because this is closely tied to properties of the data that an application sends or expects to receive.
Fall-back to TCP: do nothing. This notification is not available and will therefore not occur with TCP.

A.2. Step 2: Reduction -- The Reduced Set of Transport Features

By hiding automatable transport features from the application, a TAPS system can gain opportunities to automate the usage of network-related functionality. This can facilitate using the TAPS system for the application programmer and it allows for optimizations that may not be possible for an application. For instance, system-wide configurations regarding the usage of multiple interfaces can better be exploited if the choice of the interface is not entirely up to the application. Therefore, since they are not strictly necessary to expose in a TAPS system, we do not include automatable transport features in the reduced set of transport features. This leaves us with only the transport features that are either optimizing or functional.

A TAPS system should be able to fall back to TCP or UDP if alternative transport protocols are found not to work. Here we only consider falling back to TCP. For some transport features, it was identified that no fall-back to TCP is possible. This eliminates the possibility to use TCP whenever an application makes use of one of these transport features. Thus, we only keep the functional and

optimizing transport features for which a fall-back to TCP is possible in our reduced set. "Reset Association" and "Notification of Association Reset" are only functional because of their relationship to "Obtain a message delivery number", which is functional. Because "Obtain a message delivery number" does not have a fall-back to TCP, none of these three transport features are included in the reduced set.

A.2.1. CONNECTION Related Transport Features

ESTABLISHMENT:

- o Connect
- o Specify number of attempts and/or timeout for the first establishment message
- o Configure authentication
- o Hand over a message to transfer (possibly multiple times) before connection establishment
- o Hand over a message to transfer during connection establishment

AVAILABILITY:

- o Listen
- o Configure authentication

MAINTENANCE:

- o Change timeout for aborting connection (using retransmit limit or time value)
- o Suggest timeout to the peer
- o Disable Nagle algorithm
- o Notification of Excessive Retransmissions (early warning below abortion threshold)
- o Specify DSCP field
- o Notification of ICMP error message arrival
- o Change authentication parameters
- o Obtain authentication information
- o Set Cookie life value
- o Choose a scheduler to operate between streams of an association
- o Configure priority or weight for a scheduler
- o Configure size where messages are broken up for partial delivery
- o Disable checksum when sending
- o Disable checksum requirement when receiving
- o Specify checksum coverage used by the sender
- o Specify minimum checksum coverage required by receiver
- o Specify DF field
- o Get max. transport-message size that may be sent using a non-fragmented IP packet from the configured interface
- o Get max. transport-message size that may be received from the configured interface

- o Obtain ECN field
- o Enable and configure a "Low Extra Delay Background Transfer"

TERMINATION:

- o Close after reliably delivering all remaining data, causing an event informing the application on the other side
- o Abort without delivering remaining data, causing an event informing the application on the other side
- o Abort without delivering remaining data, not causing an event informing the application on the other side
- o Timeout event when data could not be delivered for too long

A.2.2. DATA Transfer Related Transport Features

A.2.2.1. Sending Data

- o Reliably transfer data, with congestion control
- o Reliably transfer a message, with congestion control
- o Unreliably transfer a message
- o Configurable Message Reliability
- o Choice between unordered (potentially faster) or ordered delivery of messages
- o Request not to bundle messages
- o Specifying a key id to be used to authenticate a message
- o Request not to delay the acknowledgement (SACK) of a message

A.2.2.2. Receiving Data

- o Receive data (with no message delineation)
- o Information about partial message arrival

A.2.2.3. Errors

This section describes sending failures that are associated with a specific call to in the "Sending Data" category (Appendix A.1.2.1).

- o Notification of send failures
- o Notification that the stack has no more user data to send
- o Notification to a receiver that a partial message delivery has been aborted

A.3. Step 3: Discussion

The reduced set in the previous section exhibits a number of peculiarities, which we will discuss in the following.

A.3.1. Sending Messages, Receiving Bytes

There are several transport features related to sending, but only a single transport feature related to receiving: "Receive data (with no message delineation)" (and, strangely, "information about partial message arrival"). Notably, the transport feature "Receive a message" is also the only non-automatable transport feature of UDP(-Lite) that had to be removed because no fall-back to TCP is possible.

To support these TCP receiver semantics, we define an "Application-Framed Bytestream" (AFra-Bytestream). AFra-Bytestreams allow senders to operate on messages while minimizing changes to the TCP socket API. In particular, nothing changes on the receiver side - data can be accepted via a normal TCP socket.

In an AFra-Bytestream, the sending application can optionally inform the transport about frame boundaries and required properties per frame (configurable order and reliability, or embedding a request not to delay the acknowledgement of a frame). Whenever the sending application specifies per-frame properties that relax the notion of reliable in-order delivery of bytes, it must assume that the receiving application is 1) able to determine frame boundaries, provided that frames are always kept intact, and 2) able to accept these relaxed per-frame properties. Any signaling of such information to the peer is up to an application-layer protocol and considered out of scope of this document.

For example, if an application requests to transfer fixed-size messages of 100 bytes with partial reliability, this needs the receiving application to be prepared to accept data in chunks of 100 bytes. If, then, some of these 100-byte messages are missing (e.g., if SCTP with Configurable Reliability is used), this is the expected application behavior. With TCP, no messages would be missing, but this is also correct for the application, and the possible retransmission delay is acceptable within the best effort service model. Still, the receiving application would separate the byte stream into 100-byte chunks.

Note that this usage of messages does not require all messages to be equal in size. Many application protocols use some form of Type-Length-Value (TLV) encoding, e.g. by defining a header including length fields; another alternative is the use of byte stuffing methods such as COBS [COBS]. If an application needs message numbers, e.g. to restore the correct sequence of messages, these must also be encoded by the application itself, as the sequence number related transport features of SCTP are no longer provided (in the interest of enabling a fall-back to TCP).

!!!NOTE: IMPLEMENTATION DETAILS BELOW WILL BE MOVED TO A SEPARATE DRAFT IN A FUTURE VERSION.!!!

For the implementation of a TAPS system, this has the following consequences:

- o Because the receiver-side transport leaves it up to the application to delineate messages, messages must always remain intact as they are handed over by the transport receiver. Data can be handed over at any time as they arrive, but the byte stream must never "skip ahead" to the beginning of the next message.
- o With SCTP, a "partial flag" informs a receiving application that a message is incomplete. Then, the next receive calls will only deliver remaining parts of the same message (i.e., no messages or partial messages will arrive on other streams until the message is complete) (see Section 8.1.20 in [RFC6458]). This can facilitate the implementation of the receiver buffer in the receiving application, but then such an application does not support message interleaving (which is required by stream schedulers). However, receiving a byte stream from multiple SCTP streams requires a per-stream receiver buffer anyway, so this potential benefit is lost and the "partial flag" (the transport feature "Information about partial message arrival") becomes unnecessary for a TAPS system. With it, the transport features "Configure size where messages are broken up for partial delivery" and "Notification to a receiver that a partial message delivery has been aborted" become unnecessary too.
- o From the above, a TAPS system should always support message interleaving because it enables the use of stream schedulers and comes at no additional implementation cost on the receiver side. Stream schedulers operate on the sender side. Hence, because a TAPS sender-side application may talk to an SCTP receiver that does not support interleaving, it cannot assume that stream schedulers will always work as expected.

A.3.2. Stream Schedulers Without Streams

We have already stated that multi-streaming does not require application-specific knowledge. Potential benefits or disadvantages of, e.g., using two streams over an SCTP association versus using two separate SCTP associations or TCP connections are related to knowledge about the network and the particular transport protocol in use, not the application. However, the transport features "Choose a scheduler to operate between streams of an association" and "Configure priority or weight for a scheduler" operate on streams. Here, streams identify communication channels between which a scheduler operates, and they can be assigned a priority. Moreover, the transport features in the MAINTENANCE category all operate on associations in case of SCTP, i.e. they apply to all streams in that

association.

With only these semantics necessary to represent, the interface to a TAPS system becomes easier if we rename connections into "TAPS flows" (the TAPS equivalent of a connection which may be a transport connection or association, but could also become a stream of an existing SCTP association, for example) and allow assigning a "Group Number" to a TAPS flow. Then, all MAINTENANCE transport features can be said to operate on flow groups, not connections, and a scheduler also operates on the flows within a group.

!!!NOTE: IMPLEMENTATION DETAILS BELOW WILL BE MOVED TO A SEPARATE DRAFT IN A FUTURE VERSION.!!!

For the implementation of a TAPS system, this has the following consequences:

- o Streams may be identified in different ways across different protocols. The only multi-streaming protocol considered in this document, SCTP, uses a stream id. The transport association below still uses a Transport Address (which includes one port number) for each communicating endpoint. To implement a TAPS system without exposed streams, an application must be given an identifier for each TAPS flow (akin to a socket), and depending on whether streams are used or not, there will be a 1:1 mapping between this identifier and local ports or not.
- o In SCTP, a fixed number of streams exists from the beginning of an association; streams are not "established", there is no handshake or any other form of signaling to create them: they can just be used. They are also not "gracefully shut down" -- at best, an "SSN Reset Request Parameter" in a "RE-CONFIG" chunk [RFC6525] can be used to inform the peer that of a "Stream Reset", as a rough equivalent of an "Abort". This has an impact on the semantics connection establishment and teardown (see Section 3.1).
- o To support stream schedulers, a receiver-side TAPS system should always support message interleaving because it comes at no additional implementation cost (because of the receiver-side stream reception discussed in Appendix A.3.1). Note, however, that Stream schedulers operate on the sender side. Hence, because a TAPS sender-side application may talk to a native TCP-based receiver-side application, it cannot assume that stream schedulers will always work as expected.

A.3.3. Early Data Transmission

There are two transport features related to transferring a message early: "Hand over a message to transfer (possibly multiple times) before connection establishment", which relates to TCP Fast Open [RFC7413], and "Hand over a message to transfer during connection

establishment", which relates to SCTP's ability to transfer data together with the COOKIE-Echo chunk. Also without TCP Fast Open, TCP can transfer data during the handshake, together with the SYN packet -- however, the receiver of this data may not hand it over to the application until the handshake has completed. This functionality is commonly available in TCP and supported in several implementations, even though the TCP specification does not explain how to provide it to applications.

A TAPS system could differentiate between the cases of transmitting data "before" (possibly multiple times) or during the handshake. Alternatively, it could also assume that data that are handed over early will be transmitted as early as possible, and "before" the handshake would only be used for data that are explicitly marked as "idempotent" (i.e., it would be acceptable to transfer it multiple times).

The amount of data that can successfully be transmitted before or during the handshake depends on various factors: the transport protocol, the use of header options, the choice of IPv4 and IPv6 and the Path MTU. A TAPS system should therefore allow a sending application to query the maximum amount of data it can possibly transmit before (or, if exposed, during) connection establishment.

A.3.4. Sender Running Dry

The transport feature "Notification that the stack has no more user data to send" relates to SCTP's "SENDER DRY" notification. Such notifications can, in principle, be used to avoid having an unnecessarily large send buffer, yet ensure that the transport sender always has data available when it has an opportunity to transmit it. This has been found to be very beneficial for some applications [WWDC2015]. However, "SENDER DRY" truly means that the entire send buffer (including both unsent and unacknowledged data) has emptied -- i.e., when it notifies the sender, it is already too late, the transport protocol already missed an opportunity to send data. Some modern TCP implementations now include the unspecified "TCP_NOTSENT_LOWAT" socket option proposed in [WWDC2015], which limits the amount of unsent data that TCP can keep in the socket buffer; this allows to specify at which buffer filling level the socket becomes writable, rather than waiting for the buffer to run empty.

SCTP allows to configure the sender-side buffer too: the automatable Transport Feature "Configure send buffer size" provides this functionality, but only for the complete buffer, which includes both unsent and unacknowledged data. SCTP does not allow to control these two sizes separately. A TAPS system should allow for uniform access

to "TCP_NOTSENT_LOWAT" as well as the "SENDER DRY" notification.

A.3.5. Capacity Profile

The transport features:

- o Disable Nagle algorithm
- o Enable and configure a "Low Extra Delay Background Transfer"
- o Specify DSCP field

all relate to a QoS-like application need such as "low latency" or "scavenger". In the interest of flexibility of a TAPS system, they could therefore be offered in a uniform, more abstract way, where a TAPS system could e.g. decide by itself how to use combinations of LEDBAT-like congestion control and certain DSCP values, and an application would only specify a general "capacity profile" (a description of how it wants to use the available capacity). A need for "lowest possible latency at the expense of overhead" could then translate into automatically disabling the Nagle algorithm.

In some cases, the Nagle algorithm is best controlled directly by the application because it is not only related to a general profile but also to knowledge about the size of future messages. For fine-grain control over Nagle-like functionality, the "Request not to bundle messages" is available.

A.3.6. Security

Both TCP and SCTP offer authentication. TCP authenticates complete segments. SCTP allows to configure which of SCTP's chunk types must always be authenticated -- if this is exposed as such, it creates an undesirable dependency on the transport protocol. For compatibility with TCP, a TAPS system should only allow to configure complete transport layer packets, including headers, IP pseudo-header (if any) and payload.

Security will be discussed in a separate TAPS document (to be referenced here when it appears). The minimal set presented in the present document therefore excludes all security related transport features: "Configure authentication", "Change authentication parameters", "Obtain authentication information" and "Set Cookie life value" as well as "Specifying a key id to be used to authenticate a message".

A.3.7. Packet Size

UDP(-Lite) has a transport feature called "Specify DF field". This yields an error message in case of sending a message that exceeds the Path MTU, which is necessary for a UDP-based application to be able to implement Path MTU Discovery (a function that UDP-based

applications must do by themselves). The "Get max. transport-message size that may be sent using a non-fragmented IP packet from the configured interface" transport feature yields an upper limit for the Path MTU (minus headers) and can therefore help to implement Path MTU Discovery more efficiently.

This also relates to the fact that the choice of path is automatable: if a TAPS system can switch a path at any time, unknown to an application, yet the application intends to do Path MTU Discovery, this could yield a very inefficient behavior. Thus, a TAPS system should probably avoid automatically switching paths, and inform the application about any unavoidable path changes, when applications request to disallow fragmentation with the "Specify DF field" feature.

Appendix B. Revision information

XXX RFC-Ed please remove this section prior to publication.

-02: implementation suggestions added, discussion section added, terminology extended, DELETED category removed, various other fixes; list of Transport Features adjusted to -01 version of [TAPS2] except that MPTCP is not included.

-03: updated to be consistent with -02 version of [TAPS2].

-04: updated to be consistent with -03 version of [TAPS2]. Reorganized document, rewrote intro and conclusion, and made a first stab at creating a real "minimal set".

-05: updated to be consistent with -05 version of [TAPS2] (minor changes). Fixed a mistake regarding Cookie Life value. Exclusion of security related transport features (to be covered in a separate document). Reorganized the document (now begins with the minset, derivation is in the appendix). First stab at an abstract API for the minset.

Authors' Addresses

Stein Gjessing
University of Oslo
PO Box 1080 Blindern
Oslo, N-0316
Norway

Phone: +47 22 85 24 44
Email: steing@ifi.uio.no

Michael Welzl
University of Oslo
PO Box 1080 Blindern
Oslo, N-0316
Norway

Phone: +47 22 85 24 20
Email: michawe@ifi.uio.no

TAPS
Internet-Draft
Intended status: Informational
Expires: April 29, 2018

M. Welzl
University of Oslo
M. Tuexen
Muenster Univ. of Appl. Sciences
N. Khademi
University of Oslo
October 26, 2017

On the Usage of Transport Features Provided by IETF Transport Protocols
draft-ietf-taps-transports-usage-09

Abstract

This document describes how the transport protocols Transmission Control Protocol (TCP), MultiPath TCP (MPTCP), Stream Control Transmission Protocol (SCTP), User Datagram Protocol (UDP) and Lightweight User Datagram Protocol (UDP-Lite) expose services to applications and how an application can configure and use the features that make up these services. It also discusses the service provided by the Low Extra Delay Background Transport (LEDBAT) congestion control mechanism. The description results in a set of transport abstractions that can be exported in a TAPS API.

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 April 29, 2018.

Copyright Notice

Copyright (c) 2017 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. Terminology	3
2. Introduction	3
3. Pass 1	5
3.1. Primitives Provided by TCP	5
3.1.1. Excluded Primitives or Parameters	9
3.2. Primitives Provided by MPTCP	10
3.3. Primitives Provided by SCTP	11
3.3.1. Excluded Primitives or Parameters	18
3.4. Primitives Provided by UDP and UDP-Lite	18
3.5. The service of LEDBAT	18
4. Pass 2	19
4.1. CONNECTION Related Primitives	20
4.2. DATA Transfer Related Primitives	38
5. Pass 3	41
5.1. CONNECTION Related Transport Features	41
5.2. DATA Transfer Related Transport Features	47
5.2.1. Sending Data	47
5.2.2. Receiving Data	48
5.2.3. Errors	49
6. Acknowledgements	49
7. IANA Considerations	49
8. Security Considerations	50
9. References	50
9.1. Normative References	50
9.2. Informative References	52
Appendix A. Overview of RFCs used as input for pass 1	53
Appendix B. How this document was developed	54
Appendix C. Revision information	55
Authors' Addresses	57

1. Terminology

Transport Feature: a specific end-to-end feature that the transport layer provides to an application. Examples include confidentiality, reliable delivery, ordered delivery, message-versus-stream orientation, etc.

Transport Service: a set of Transport Features, without an association to any given framing protocol, which provides a complete service to an application.

Transport Protocol: an implementation that provides one or more transport services using a specific framing and header format on the wire.

Transport Protocol Component: an implementation of a Transport Feature within a protocol.

Transport Service Instance: an arrangement of transport protocols with a selected set of features and configuration parameters that implements a single transport service, e.g., a protocol stack (RTP over UDP).

Application: an entity that uses the transport layer for end-to-end delivery of data across the network (this may also be an upper layer protocol or tunnel encapsulation).

Endpoint: an entity that communicates with one or more other endpoints using a transport protocol.

Connection: shared state of two or more endpoints that persists across messages that are transmitted between these endpoints.

Primitive: a function call that is used to locally communicate between an application and a transport endpoint. A primitive is related to one or more Transport Features.

Event: a primitive that is invoked by a transport endpoint.

Parameter: a value passed between an application and a transport protocol by a primitive.

Socket: the combination of a destination IP address and a destination port number.

Transport Address: the combination of an IP address, transport protocol and the port number used by the transport protocol.

2. Introduction

This specification describes an abstract interface for applications to make use of Transport Services, such that applications are no longer directly tied to a specific protocol. Breaking this strict connection can reduce the effort for an application programmer, yet attain greater transport flexibility by pushing complexity into an underlying Transport Services (TAPS) system.

This design process has started with a survey of the services provided by IETF transport protocols and congestion control

mechanisms [RFC8095]. The present document and [FJ16] complement this survey with an in-depth look at the defined interactions between applications and the following unicast transport protocols: Transmission Control Protocol (TCP), MultiPath TCP (MPTCP), Stream Control Transmission Protocol (SCTP), User Datagram Protocol (UDP), Lightweight User Datagram Protocol (UDP-Lite). We also define a primitive to enable/disable and configure the Low Extra Delay Background Transport (LEDBAT) unicast congestion control mechanism. For UDP and UDP-Lite, the first step of the protocol analysis -- a discussion of relevant RFC text -- is documented in [FJ16].

This snapshot in time analysis of the IETF transport protocols is published as an RFC to document the authors' and working group's analysis, generating a set of transport abstractions that can be exported in a TAPS API. It provides the basis for the minimal set of transport services that end systems supporting TAPS should implement [I-D.draft-gjessing-taps-minset].

The list of primitives, events and transport features in this document is strictly based on the parts of protocol specifications that describe what the protocol provides to an application using it and how the application interacts with it. Transport protocols provide communication between processes that operate on network endpoints, which means that they allow for multiplexing of communication between the same IP addresses, and this multiplexing is achieved using port numbers. Port multiplexing is therefore assumed to be always provided and not discussed in this document.

Parts of a protocol that are explicitly stated as optional to implement are not covered. Interactions between the application and a transport protocol that are not directly related to the operation of the protocol are also not covered. For example, there are various ways for an application to use socket options to indicate its interest in receiving certain notifications [RFC6458]. However, for the purpose of identifying primitives, events and transport features, the ability to enable or disable the reception of notifications is irrelevant. Similarly, "one-to-many style sockets" [RFC6458] just affect the application programming style, not how the underlying protocol operates, and they are therefore not discussed here. The same is true for the ability to obtain the unchanged value of a parameter that an application has previously set (e.g., via "get" in get/set operations [RFC6458]).

The document presents a three-pass process to arrive at a list of transport features. In the first pass, the relevant RFC text is discussed per protocol. In the second pass, this discussion is used to derive a list of primitives and events that are uniformly categorized across protocols. Here, an attempt is made to present or

-- where text describing primitives or events does not yet exist -- construct primitives or events in a slightly generalized form to highlight similarities. This is, for example, achieved by renaming primitives or events of protocols or by avoiding a strict 1:1-mapping between the primitives or events in the protocol specification and primitives or events in the list. Finally, the third pass presents transport features based on pass 2, identifying which protocols implement them.

In the list resulting from the second pass, some transport features are missing because they are implicit in some protocols, and they only become explicit when we consider the superset of all transport features offered by all protocols. For example, TCP always carries out congestion control; we have to consider it together with a protocol like UDP (which does not have congestion control) before we can consider congestion control as a transport feature. The complete list of transport features across all protocols is therefore only available after pass 3.

Some protocols are connection-oriented. Connection-oriented protocols often use an initial call to a specific primitive to open a connection before communication can progress, and require communication to be explicitly terminated by issuing another call to a primitive (usually called "close"). A "connection" is the common state that some transport primitives refer to, e.g., to adjust general configuration settings. Connection establishment, maintenance and termination are therefore used to categorize transport primitives of connection-oriented transport protocols in pass 2 and pass 3. For this purpose, UDP is assumed to be used with "connected" sockets, i.e. sockets that are bound to a specific pair of addresses and ports [FJ16].

3. Pass 1

This first iteration summarizes the relevant text parts of the RFCs describing the protocols, focusing on what each transport protocol provides to the application and how it is used (abstract API descriptions, where they are available). When presenting primitives, events and parameters, the use of lower- and upper-case characters is made uniform for the sake of readability.

3.1. Primitives Provided by TCP

The initial TCP specification [RFC0793] states: "The Transmission Control Protocol (TCP) is intended for use as a highly reliable host-to-host protocol between hosts in packet-switched computer communication networks, and in interconnected systems of such

networks". Section 3.8 in this specification [RFC0793] further specifies the interaction with the application by listing several transport primitives. It is also assumed that an Operating System provides a means for TCP to asynchronously signal the application; the primitives representing such signals are called 'events' in this section. This section describes the relevant primitives.

Open: this is either active or passive, to initiate a connection or listen for incoming connections. All other primitives are associated with a specific connection, which is assumed to first have been opened. An active open call contains a socket. A passive open call with a socket waits for a particular connection; alternatively, a passive open call can leave the socket unspecified to accept any incoming connection. A fully specified passive call can later be made active by calling 'Send'. Optionally, a timeout can be specified, after which TCP will abort the connection if data has not been successfully delivered to the destination (else a default timeout value is used). A procedure for aborting the connection is used to avoid excessive retransmissions, and an application is able to control the threshold used to determine the condition for aborting; this threshold may be measured in time units or as a count of retransmission [RFC1122]. This indicates that the timeout could also be specified as a count of retransmission.

Also optional, for multihomed hosts, the local IP address can be provided [RFC1122]. If it is not provided, a default choice will be made in case of active open calls. A passive open call will await incoming connection requests to all local addresses and then maintain usage of the local IP address where the incoming connection request has arrived. Finally, the 'options' parameter allows the application to specify IP options such as source route, record route, or timestamp [RFC1122]. It is not stated on which segments of a connection these options should be applied, but probably all segments, as this is also stated in a specification given for the usage of source route (section 4.2.3.8 of [RFC1122]). Source route is the only non-optional IP option in this parameter, allowing an application to specify a source route when it actively opens a TCP connection.

Master Key Tuples (MKTs) for authentication can optionally be configured when calling open (section 7.1 of [RFC5925]). When authentication is in use, complete TCP segments are authenticated, including the TCP IPv4 pseudoheader, TCP header, and TCP data.

TCP Fast Open (TFO) [RFC7413] allows applications to immediately hand over a message from the active open to the passive open side of a TCP connection together with the first message establishment

packet (the SYN). This can be useful for applications that are sensitive to TCP's connection setup delay. [RFC7413] states that "TCP implementations MUST NOT use TFO by default, but only use TFO if requested explicitly by the application on a per-service-port basis". The size of the message sent with TFO cannot be more than TCP's maximum segment size (minus options used in the SYN). For the active open side, it is recommended to change or replace the connect() call in order to support a user data buffer argument [RFC7413]. For the passive open side, the application needs to enable the reception of Fast Open requests, e.g. via a new TCP_FASTOPEN setsockopt() socket option before listen(). The receiving application must be prepared to accept duplicates of the TFO message, as the first data written to a socket can be delivered more than once to the application on the remote host.

Send: this is the primitive that an application uses to give the local TCP transport endpoint a number of bytes that TCP should reliably send to the other side of the connection. The 'urgent' flag, if set, states that the data handed over by this send call is urgent and this urgency should be indicated to the receiving process in case the receiving application has not yet consumed all non-urgent data preceding it. An optional timeout parameter can be provided that updates the connection's timeout (see 'open'). Additionally, optional parameters allow to indicate the preferred outgoing MKT (current_key) and/or the preferred incoming MKT (rnext_key) of a connection (section 7.1 of [RFC5925]).

Receive: This primitive allocates a receiving buffer for a provided number of bytes. It returns the number of received bytes provided in the buffer when these bytes have been received and written into the buffer by TCP. The application is informed of urgent data via an 'urgent' flag: if it is on, there is urgent data. If it is off, there is no urgent data or this call to 'receive' has returned all the urgent data. The application is also informed about the current_key and rnext_key information carried in a recently received segment via an optional parameter (section 7.1 of [RFC5925]).

Close: This primitive closes one side of a connection. It is semantically equivalent to "I have no more data to send" but does not mean "I will not receive any more", as the other side may still have data to send. This call reliably delivers any data that has already been given to TCP (and if that fails, 'close' becomes 'abort').

Abort: This primitive causes all pending 'send' and 'receive' calls to be aborted. A TCP "RESET" message is sent to the TCP endpoint on the other side of the connection [RFC0793].

Close Event: TCP uses this primitive to inform an application that the application on the other side has called the 'close' primitive, so the local application can also issue a 'close' and terminate the connection gracefully. See [RFC0793], Section 3.5.

Abort Event: When TCP aborts a connection upon receiving a "RESET" from the peer, it "advises the user and goes to the CLOSED state." See [RFC0793], Section 3.4.

User Timeout Event: This event is executed when the user timeout expires (see 'open') (section 3.9 of [RFC0793]). All queues are flushed and the application is informed that the connection had to be aborted due to user timeout.

Error_Report event: This event informs the application of "soft errors" that can be safely ignored [RFC5461], including the arrival of an ICMP error message or excessive retransmissions (reaching a threshold below the threshold where the connection is aborted). See section 4.2.4.1 of [RFC1122].

Type-of-Service: Section 4.2.4.2 of the requirements for Internet hosts [RFC1122] states that "the application layer MUST be able to specify the Type-of-Service (TOS) for segments that are sent on a connection". The application should be able to change the TOS during the connection lifetime, and the TOS value should be passed to the IP layer unchanged. Since then the TOS field has been redefined. The Differentiated Services (DiffServ) model [RFC2475] [RFC3260] replaces this field in the IP Header, assigning the six most significant bits to carry the Differentiated Services Code Point (DSCP) field [RFC2474].

Nagle: The Nagle algorithm delays sending data for some time to increase the likelihood of sending a full-sized segment (section 4.2.3.4 of [RFC1122]). An application can disable the Nagle algorithm for an individual connection.

User Timeout Option: The User Timeout Option (UTO) [RFC5482] allows one end of a TCP connection to advertise its current user timeout value so that the other end of the TCP connection can adapt its own user timeout accordingly. In addition to the configurable value of the User Timeout (see 'send'), there are three per-connection state variables that an application can adjust to control the operation of the User Timeout Option (UTO): 'adv_uto' is the value of the UTO advertised to the remote TCP peer

(default: system-wide default user timeout); 'enabled' (default false) is a boolean-type flag that controls whether the UTO option is enabled for a connection. This applies to both sending and receiving. 'changeable' is a boolean-type flag (default true) that controls whether the user timeout may be changed based on a UTO option received from the other end of the connection. 'changeable' becomes false when an application explicitly sets the user timeout (see 'send').

Set / Get Authentication Parameters: The preferred outgoing MKT (current_key) and/or the preferred incoming MKT (rnext_key) of a connection can be configured. Information about current_key and rnext_key carried in a recently received segment can be retrieved (section 7.1 of [RFC5925]).

3.1.1. Excluded Primitives or Parameters

The 'open' primitive can be handed optional Precedence or security/compartment information [RFC0793], but this was not included here because it is mostly irrelevant today [RFC7414].

The 'Status' primitive was not included because the initial TCP specification describes this primitive as "implementation dependent" and states that it "could be excluded without adverse effect" [RFC0793]. Moreover, while a data block containing specific information is described, it is also stated that not all of this information may always be available. While [RFC5925] states that 'Status' "SHOULD be augmented to allow the MKTs of a current or pending connection to be read (for confirmation)", the same information is also available via 'Receive', which, following [RFC5925], "MUST be augmented" with that functionality. The 'Send' primitive includes an optional 'push' flag which, if set, requires data to be promptly transmitted to the receiver without delay [RFC0793]; the 'Receive' primitive described in can (under some conditions) yield the status of the 'push' flag. Because "push" functionality is optional to implement for both the 'send' and 'receive' primitives [RFC1122], this functionality is not included here. The requirements for Internet hosts [RFC1122] also introduce keep-alives to TCP, but these are optional to implement and hence not considered here. The same document also describes that "some TCP implementations have included a FLUSH call", indicating that this call is also optional to implement. It is therefore not considered here.

3.2. Primitives Provided by MPTCP

Multipath TCP (MPTCP) is an extension to TCP that allows the use of multiple paths for a single data-stream. It achieves this by creating different so-called TCP subflows for each of the interfaces and scheduling the traffic across these TCP subflows. The service provided by MPTCP is described as follows in [RFC6182]: "Multipath TCP MUST follow the same service model as TCP [RFC0793]: in-order, reliable, and byte-oriented delivery. Furthermore, a Multipath TCP connection SHOULD provide the application with no worse throughput or resilience than it would expect from running a single TCP connection over any one of its available paths."

Further, there are some constraints on the API exposed by MPTCP, stated in [RFC6182]: "A multipath-capable equivalent of TCP MUST retain some level of backward compatibility with existing TCP APIs, so that existing applications can use the newer merely by upgrading the operating systems of the end hosts." As such, the primitives provided by MPTCP are equivalent to the ones provided by TCP. Nevertheless, the MPTCP RFCs [RFC6824] and [RFC6897] clarify some parts of TCP's primitives with respect to MPTCP and add some extensions for better control on MPTCP's subflows. Hereafter is a list of the clarifications and extensions the above cited RFCs provide to TCP's primitives.

Open: "An application should be able to request to turn on or turn off the usage of MPTCP" [RFC6897]. This functionality can be provided through a socket-option called 'tcp_multipath_enable'. Further, MPTCP must be disabled in case the application is binding to a specific address [RFC6897].

Send/Receive: The sending and receiving of data does not require any changes to the application when MPTCP is being used [RFC6824]. The MPTCP-layer will "take one input data stream from an application, and split it into one or more subflows, with sufficient control information to allow it to be reassembled and delivered reliably and in order to the recipient application." The use of the Urgent Pointer is special in MPTCP [RFC6824], which states: "a TCP subflow MUST NOT use the Urgent Pointer to interrupt an existing mapping."

Address and Subflow Management: MPTCP uses different addresses and allows a host to announce these addresses as part of the protocol. The MPTCP API Considerations RFC [RFC6897] says "An application should be able to restrict MPTCP to binding to a given set of addresses" and thus allows applications to limit the set of addresses that are being used by MPTCP. Further, "An application should be able to obtain information on the pairs of addresses

used by the MPTCP subflows".

3.3. Primitives Provided by SCTP

TCP has a number of limitations that SCTP removes (section 1.1 of [RFC4960]). The following three removed limitations directly translate into transport features that are visible to an application using SCTP: 1) it allows for preservation of message delimiters; 2) it does not provide in-order or reliable delivery unless the application wants that; 3) multi-homing is supported. In SCTP, connections are called "associations" and they can be between not only two (as in TCP) but multiple addresses at each endpoint.

Section 10 of the SCTP base protocol specification [RFC4960] specifies the interaction with the application (which SCTP calls the "Upper Layer Protocol" (ULP)). It is assumed that the Operating System provides a means for SCTP to asynchronously signal the application; the primitives representing such signals are called 'events' in this section. Here, we describe the relevant primitives. In addition to the abstract API described in the section 10 of the SCTP base protocol specification [RFC4960], an extension to the socket API is described in [RFC6458]. This covers the functionality of the base protocol [RFC4960] and some of its extensions [RFC3758], [RFC4895], [RFC5061]. For other protocol extensions [RFC6525], [RFC6951], [RFC7053], [RFC7496], [RFC7829], [I-D.ietf-tsvwg-sctp-ndata], the corresponding extensions of the socket API are specified in these protocol specifications. The functionality exposed to the ULP through all these APIs is considered here.

The abstract API contains a 'SetProtocolParameters' primitive that allows to adjust elements of a parameter list [RFC4960]; it is stated that SCTP implementations "may allow ULP to customize some of these protocol parameters", indicating that none of the elements of this parameter list are mandatory to make ULP-configurable. Thus, we only consider the parameters in the abstract API that are also covered in one of the other RFCs listed above, which leads us to exclude the parameters RTO.Alpha, RTO.Beta and HB.Max.Burst. For clarity, we also replace 'SetProtocolParameters' itself with primitives that adjust parameters or groups of parameters that fit together.

Initialize: Initialize creates a local SCTP instance that it binds to a set of local addresses (and, if provided, a local port number) [RFC4960]. Initialize needs to be called only once per set of local addresses. A number of per-association initialization parameters can be used when an association is created, but before it is connected (via the primitive 'Associate'

below): the maximum number of inbound streams the application is prepared to support, the maximum number of attempts to be made when sending the INIT (the first message of association establishment), and the maximum retransmission timeout (RTO) value to use when attempting an INIT [RFC6458]. At this point, before connecting, an application can also enable UDP encapsulation by configuring the remote UDP encapsulation port number [RFC6951].

Associate: This creates an association (the SCTP equivalent of a connection) that connects the local SCTP instance and a remote SCTP instance. To identify the remote endpoint, it can be given one or multiple (using "connectx") sockets (section 9.9 of [RFC6458]). Most primitives are associated with a specific association, which is assumed to first have been created. Associate can return a list of destination transport addresses so that multiple paths can later be used. One of the returned sockets will be selected by the local endpoint as default primary path for sending SCTP packets to this peer, but this choice can be changed by the application using the list of destination addresses. Associate is also given the number of outgoing streams to request and optionally returns the number of negotiated outgoing streams. An optional parameter of 32 bits, the adaptation layer indication, can be provided [RFC5061]. If authenticated chunks are used, the chunk types required to be sent authenticated by the peer can be provided [RFC4895]. A 'SCTP_Cant_Str_Assoc' notification is used to inform the application of a failure to create an association [RFC6458]. An application could use sendto() or sendmsg() to implicitly setup an association, thereby handing over a message that SCTP might send during the association setup phase [RFC6458]. Note that this mechanism is different from TCP's TFO mechanism: the message would arrive only once, after at least one RTT, as it is sent together with the third message exchanged during association setup, the COOKIE-ECHO chunk).

Send: This sends a message of a certain length in bytes over an association. A number can be provided to later refer to the correct message when reporting an error, and a stream id is provided to specify the stream to be used inside an association (we consider this as a mandatory parameter here for simplicity: if not provided, the stream id defaults to 0). A condition to abandon the message can be specified (for example limiting the number of retransmissions or the lifetime of the user message). This allows to control the partial reliability extension [RFC3758], [RFC7496]. An optional maximum life time can specify the time after which the message should be discarded rather than sent. A choice (advisory, i.e. not guaranteed) of the preferred path can be made by providing a socket, and the message can be

delivered out-of-order if the unordered flag is set. An advisory flag indicates that the peer should not delay the acknowledgement of the user message provided [RFC7053]. Another advisory flag indicates whether the application prefers to avoid bundling user data with other outbound DATA chunks (i.e., in the same packet). A payload protocol-id can be provided to pass a value that indicates the type of payload protocol data to the peer. If authenticated chunks are used, the key identifier for authenticating DATA chunks can be provided [RFC4895].

Receive: Messages are received from an association, and optionally a stream within the association, with their size returned. The application is notified of the availability of data via a 'Data Arrive' notification. If the sender has included a payload protocol-id, this value is also returned. If the received message is only a partial delivery of a whole message, a partial flag will indicate so, in which case the stream id and a stream sequence number are provided to the application.

Shutdown: This primitive gracefully closes an association, reliably delivering any data that has already been handed over to SCTP. A parameter lets the application control whether further receive or send operations or both are disabled when the call is issued. A return code informs about success or failure of this procedure.

Abort: This ungracefully closes an association, by discarding any locally queued data and informing the peer that the association was aborted. Optionally, an abort reason to be passed to the peer may be provided by the application. A return code informs about success or failure of this procedure.

Change Heartbeat / Request Heartbeat: This allows the application to enable/disable heartbeats and optionally specify a heartbeat frequency as well as requesting a single heartbeat to be carried out upon a function call, with a notification about success or failure of transmitting the HEARTBEAT chunk to the destination.

Configure Max. Retransmissions of an Association: The parameter `Association.Max.Retrans` [RFC4960] (called "sasoc_maxrxt" in the SCTP socket API extensions [RFC6458]), allows to configure the number of unsuccessful retransmissions after which an entire association is considered as failed; this should invoke a 'Communication Lost' notification.

Set Primary: This allows to set a new primary default path for an association by providing a socket. Optionally, a default source address to be used in IP datagrams can be provided.

Change Local Address / Set Peer Primary: This allows an endpoint to add/remove local addresses to/from an association. In addition, the peer can be given a hint which address to use as the primary address [RFC5061].

Configure Path Switchover: The abstract API contains a primitive called 'Set Failure Threshold' [RFC4960]. This configures the parameter "Path.Max.Retrans", which determines after how many retransmissions a particular transport address is considered as unreachable. If there are more transport addresses available in an association, reaching this limit will invoke a path switchover. An extension called "SCTP-PF" adds a concept of "Potentially Failed" (PF) paths to this method [RFC7829]. When a path is in PF state, SCTP will not entirely give up sending on that path, but it will preferably send data on other active paths if such paths are available. Entering the PF state is done upon exceeding a configured maximum number of retransmissions. Thus, for all paths where this mechanism is used, there are two configurable error thresholds: one to decide that a path is in PF state, and one to decide that the transport address is unreachable.

Set / Get Authentication Parameters: This allows an endpoint to add/remove key material to/from an association. In addition, the chunk types being authenticated can be queried [RFC4895].

Add / Reset Streams, Reset Association: This allows an endpoint to add streams to an existing association or or to reset them individually. Additionally, the association can be reset [RFC6525].

Status: The 'Status' primitive returns a data block with information about a specified association, containing: association connection state; destination transport address list; destination transport address reachability states; current local and peer receiver window sizes; current local congestion window sizes; number of unacknowledged DATA chunks; number of DATA chunks pending receipt; primary path; most recent SRTT on primary path; RTO on primary path; SRTT and RTO on other destination addresses [RFC4960] and MTU per path [RFC6458].

Enable / Disable Interleaving: This allows to enable or disable the negotiation of user message interleaving support for future associations. For existing associations it is possible to query whether user message interleaving support was negotiated or not on a particular association [I-D.ietf-tsvwg-sctp-ndata].

Set Stream Scheduler: This allows to select a stream scheduler per association, with a choice of: First Come First Serve, Round Robin, Round Robin per Packet, Priority Based, Fair Bandwidth, Weighted Fair Queuing [I-D.ietf-tsvwg-sctp-ndata].

Configure Stream Scheduler: This allows to change a parameter per stream for the schedulers: a priority value for the Priority Based scheduler and a weight for the Weighted Fair Queuing scheduler.

Enable / Disable NoDelay: This turns on/off any Nagle-like algorithm for an association [RFC6458].

Configure Send Buffer Size: This controls the amount of data SCTP may have waiting in internal buffers to be sent or retransmitted [RFC6458].

Configure Receive Buffer Size: This sets the receive buffer size in octets, thereby controlling the receiver window for an association [RFC6458].

Configure Message Fragmentation: If a user message causes an SCTP packet to exceed the maximum fragmentation size (which can be provided by the application, and is otherwise the PMTU size), then the message will be fragmented by SCTP. Disabling message fragmentation will produce an error instead of fragmenting the message [RFC6458].

Configure Path MTU Discovery: Path MTU Discovery can be enabled or disabled per peer address of an association (section 8.1.12 of [RFC6458]). When it is enabled, the current Path MTU value can be obtained. When it is disabled, the Path MTU to be used can be controlled by the application.

Configure Delayed SACK Timer: The time before sending a SACK can be adjusted; delaying SACKs can be disabled; the number of packets that must be received before a SACK is sent without waiting for the delay timer to expire can be configured [RFC6458].

Set Cookie Life Value: The Cookie life value can be adjusted (section 8.1.2 of [RFC6458]). "Valid.Cookie.Life" is also one of the parameters that is potentially adjustable with 'SetProtocolParameters' [RFC4960].

Set Maximum Burst: The maximum burst of packets that can be emitted by a particular association (default 4, and values above 4 are optional to implement) can be adjusted (section 8.1.2 of [RFC6458]). "Max.Burst" is also one of the parameters that is potentially adjustable with 'SetProtocolParameters' [RFC4960].

Configure RTO Calculation: The abstract API contains the following adjustable parameters: RTO.Initial; RTO.Min; RTO.Max; RTO.Alpha; RTO.Beta. Only the initial, minimum and maximum RTO are also described as configurable in the SCTP sockets API extensions [RFC6458].

Set DSCP Value: The DSCP value can be set per peer address of an association (section 8.1.12 of [RFC6458]).

Set IPv6 Flow Label: The flow label field can be set per peer address of an association (section 8.1.12 of [RFC6458]).

Set Partial Delivery Point: This allows to specify the size of a message where partial delivery will be invoked. Setting this to a lower value will cause partial deliveries to happen more often [RFC6458].

Communication Up Notification: When a lost communication to an endpoint is restored or when SCTP becomes ready to send or receive user messages, this notification informs the application process about the affected association, the type of event that has occurred, the complete set of sockets of the peer, the maximum number of allowed streams and the inbound stream count (the number of streams the peer endpoint has requested). If interleaving is supported by both endpoints, this information is also included in this notification.

Restart Notification: When SCTP has detected that the peer has restarted, this notification is passed to the upper layer [RFC6458].

Data Arrive Notification: When a message is ready to be retrieved via the 'Receive' primitive, the application is informed by this notification.

Send Failure Notification / Receive Unsent Message / Receive Unacknowledged Message: When a message cannot be delivered via an association, the sender can be informed about it and learn whether the message has just not been acknowledged or (e.g. in case of lifetime expiry) if it has not even been sent. This can also inform the sender that a part of the message has been successfully delivered.

Network Status Change Notification: This informs the application about a socket becoming active/inactive [RFC4960] or "Potentially Failed" [RFC7829].

Communication Lost Notification: When SCTP loses communication to an endpoint (e.g. via Heartbeats or excessive retransmission) or detects an abort, this notification informs the application process of the affected association and the type of event (failure OR termination in response to a shutdown or abort request).

Shutdown Complete Notification: When SCTP completes the shutdown procedures, this notification is passed to the upper layer, informing it about the affected association.

Authentication Notification: When SCTP wants to notify the upper layer regarding the key management related to authenticated chunks [RFC4895], this notification is passed to the upper layer.

Adaptation Layer Indication Notification: When SCTP completes the association setup and the peer provided an adaptation layer indication, this is passed to the upper layer [RFC5061], [RFC6458].

Stream Reset Notification: When SCTP completes the procedure for resetting streams [RFC6525], this notification is passed to the upper layer, informing it about the result.

Association Reset Notification: When SCTP completes the association reset procedure [RFC6525], this notification is passed to the upper layer, informing it about the result.

Stream Change Notification: When SCTP completes the procedure used to increase the number of streams [RFC6525], this notification is passed to the upper layer, informing it about the result.

Sender Dry Notification: When SCTP has no more user data to send or retransmit on a particular association, this notification is passed to the upper layer [RFC6458].

Partial Delivery Aborted Notification: When a receiver has begun to receive parts of a user message but the delivery of this message is then aborted, this notification is passed to the upper layer (section 6.1.7 of [RFC6458]).

3.3.1. Excluded Primitives or Parameters

The 'Receive' primitive can return certain additional information, but this is optional to implement and therefore not considered. With a 'Communication Lost' notification, some more information may optionally be passed to the application (e.g., identification to retrieve unsent and unacknowledged data). SCTP "can invoke" a 'Communication Error' notification and "may send" a 'Restart' notification, making these two notifications optional to implement. The list provided under 'Status' includes "etc", indicating that more information could be provided. The primitive 'Get SRTT Report' returns information that is included in the information that 'Status' provides and is therefore not discussed. The 'Destroy SCTP Instance' API function was excluded: it erases the SCTP instance that was created by 'Initialize', but is not a Primitive as defined in this document because it does not relate to a transport feature. The 'Shutdown' event informs an application that the peer has sent a SHUTDOWN, and hence no further data should be sent on this socket (section 6.1 of [RFC6458]). However, if an application would try to send data on the socket, it would get an error message anyway; thus, this event is classified as "just affecting the application programming style, not how the underlying protocol operates" and not included here.

3.4. Primitives Provided by UDP and UDP-Lite

The set of pass 1 primitives for UDP and UDP-Lite is documented in [FJ16].

3.5. The service of LEDBAT

The service of the Low Extra Delay Background Transport (LEDBAT) congestion control mechanism is described as follows: "LEDBAT is designed for use by background bulk-transfer applications to be no more aggressive than standard TCP congestion control (as specified in RFC 5681) and to yield in the presence of competing flows, thus limiting interference with the network performance of competing flows" [RFC6817].

LEDBAT does not have any primitives, as LEDBAT is not a transport protocol. According to its specification [RFC6817], "LEDBAT can be used as part of a transport protocol or as part of an application, as

long as the data transmission mechanisms are capable of carrying timestamps and acknowledging data frequently. LEDBAT can be used with TCP, Stream Control Transmission Protocol (SCTP), and Datagram Congestion Control Protocol (DCCP), with appropriate extensions where necessary; and it can be used with proprietary application protocols, such as those built on top of UDP for peer-to-peer (P2P) applications." At the time of writing, the appropriate extensions for TCP, SCTP or DCCP do not exist.

A number of configurable parameters exist in the LEDBAT specification: TARGET, which is the queuing delay target at which LEDBAT tries to operate, must be set to 100ms or less. 'allowed_increase' (should be 1, must be greater than 0) limits the speed at which LEDBAT increases its rate. 'gain', which, according to [RFC6817], "MUST be set to 1 or less" to avoid a faster ramp-up than TCP Reno, determines how quickly the sender responds to changes in queueing delay. Implementations may divide 'gain' into two parameters, one for increase and a possibly larger one for decrease. We call these parameters 'Gain_Inc' and 'Gain_Dec' here. 'Base_History' is the size of the list of measured base delays, and, according to [RFC6817], "SHOULD be 10". This list can be filtered using a 'Filter' function which is not prescribed [RFC6817], yielding a list of size 'Current_Filter'. The initial and minimum congestion windows, 'Init_CWND' and 'Min_CWND', should both be 2.

Regarding which of these parameters should be under control of an application, the possible range goes from exposing nothing on the one hand, to considering everything that is not prescribed with a "MUST" in the specification as a parameter on the other hand. Function implementations are not provided as a parameter to any of the transport protocols discussed here, and hence we do not regard the 'Filter' function as a parameter. However, to avoid unnecessarily limiting future implementations, we consider all other parameters above as tunable parameters that should be exposed.

4. Pass 2

This pass categorizes the primitives from pass 1 based on whether they relate to a connection or to data transmission. Primitives are presented following the nomenclature "CATEGORY.[SUBCATEGORY].PRIMITIVE_NAME.PROTOCOL". The CATEGORY can be CONNECTION or DATA. Within the CONNECTION category, ESTABLISHMENT, AVAILABILITY, MAINTENANCE and TERMINATION subcategories can be considered. The DATA category does not have any SUBCATEGORY. The PROTOCOL name "UDP(-Lite)" is used when primitives are equivalent for UDP and UDP-Lite; the PROTOCOL name "TCP" refers to both TCP and MPTCP. We present "connection" as a general protocol-independent

concept and use it to refer to, e.g., TCP connections (identifiable by a unique pair of IP addresses and TCP port numbers), SCTP associations (identifiable by multiple IP address and port number pairs), as well UDP and UDP-Lite connections (identifiable by a unique socket pair).

Some minor details are omitted for the sake of generalization -- e.g., SCTP's 'Close' [RFC4960] returns success or failure, and lets the application control whether further receive or send operations or both are disabled [RFC6458]. This is not described in the same way for TCP [RFC0793], but these details play no significant role for the primitives provided by either TCP or SCTP (for the sake of being generic, it could be assumed that both receive and send operations are disabled in both cases).

The TCP 'Send' and 'Receive' primitives include usage of an 'urgent' parameter. This parameter controls a mechanism that is required to implement the "synch signal" used by telnet [RFC0854], but [RFC6093] states that "new applications SHOULD NOT employ the TCP urgent mechanism". Because pass 2 is meant as a basis for the creation of future systems, the "urgent" mechanism is excluded. This also concerns the notification 'Urgent Pointer Advance' in the 'Error_Report' (section 4.2.4.1 of [RFC1122]).

Since LEDBAT is a congestion control mechanism and not a protocol, it is not currently defined when to enable / disable or configure the mechanism. For instance, it could be a one-time choice upon connection establishment or when listening for incoming connections, in which case it should be categorized under CONNECTION. ESTABLISHMENT or CONNECTION.AVAILABILITY, respectively. To avoid unnecessarily limiting future implementations, it was decided to place it under CONNECTION.MAINTENANCE, with all parameters that are described in the specification [RFC6817] made configurable.

4.1. CONNECTION Related Primitives

ESTABLISHMENT:

Active creation of a connection from one transport endpoint to one or more transport endpoints.

Interfaces to UDP and UDP-Lite allow both connection-oriented and connection-less usage of the API [RFC8085].

o CONNECT.TCP:

Pass 1 primitive / event: 'Open' (active) or 'Open' (passive) with socket, followed by 'Send'

Parameters: 1 local IP address (optional); 1 destination transport address (for active open; else the socket and the local IP address of the succeeding incoming connection request will be maintained); timeout (optional); options (optional); MKT configuration (optional); user message (optional)

Comments: If the local IP address is not provided, a default choice will automatically be made. The timeout can also be a retransmission count. The options are IP options to be used on all segments of the connection. At least the Source Route option is mandatory for TCP to provide. 'MKT configuration' refers to the ability to configure Master Key Tuples (MKTs) for authentication. The user message may be transmitted to the peer application immediately upon reception of the TCP SYN packet. To benefit from the lower latency this provides as part of the experimental TFO mechanism, its length must be at most the TCP's maximum segment size (minus TCP options used in the SYN). The message may also be delivered more than once to the application on the remote host.

o CONNECT.SCTP:

Pass 1 primitive / event: 'Initialize', followed by 'Enable / Disable Interleaving' (optional), followed by 'Associate'

Parameters: list of local SCTP port number / IP address pairs ('Initialize'); one or several sockets (identifying the peer); outbound stream count; maximum allowed inbound stream count; adaptation layer indication (optional); chunk types required to be authenticated (optional); request interleaving on/off; maximum number of INIT attempts (optional); maximum init. RTO for INIT (optional); user message (optional); remote UDP port number (optional)

Returns: socket list or failure

Comments: 'Initialize' needs to be called only once per list of local SCTP port number / IP address pairs. One socket will automatically be chosen; it can later be changed in MAINTENANCE. The user message may be transmitted to the peer application immediately upon reception of the packet containing the COOKIE-ECHO chunk. To benefit from the lower latency this provides, its length must be limited such that it fits into the packet containing the COOKIE-ECHO chunk. If a remote UDP port number is provided, SCTP packets will be encapsulated in UDP.

- o CONNECT.MPTCP:

This is similar to CONNECT.TCP except for one additional boolean parameter that allows to enable or disable MPTCP for a particular connection or socket (default: enabled).

- o CONNECT.UDP(-Lite):

Pass 1 primitive / event: 'Connect' followed by 'Send'.

Parameters: 1 local IP address (default (ANY), or specified); 1 destination transport address; 1 local port (default (OS chooses), or specified); 1 destination port (default (OS chooses), or specified).

Comments: Associates a transport address creating a UDP(-Lite) socket connection. This can be called again with a new transport address to create a new connection. The CONNECT function allows an application to receive errors from messages sent to a transport address.

AVAILABILITY:

Preparing to receive incoming connection requests.

- o LISTEN.TCP:

Pass 1 primitive / event: 'Open' (passive)

Parameters: 1 local IP address (optional); 1 socket (optional); timeout (optional); buffer to receive a user message (optional); MKT configuration (optional)

Comments: if the socket and/or local IP address is provided, this waits for incoming connections from only and/or to only the provided address. Else this waits for incoming connections without this / these constraint(s). ESTABLISHMENT can later be performed with 'Send'. If a buffer is provided to receive a user message, a user message can be received from a TFO-enabled sender before TCP's connection handshake is completed. This message may arrive multiple times. 'MKT configuration' refers to the ability to configure Master Key Tuples (MKTs) for authentication.

- o LISTEN.SCTP:

Pass 1 primitive / event: 'Initialize', followed by 'Communication Up' or 'Restart' notification and possibly 'Adaptation Layer' notification

Parameters: list of local SCTP port number / IP address pairs (initialize)

Returns: socket list; outbound stream count; inbound stream count; adaptation layer indication; chunks required to be authenticated; interleaving supported on both sides yes/no

Comments: 'Initialize' needs to be called only once per list of local SCTP port number / IP address pairs. 'Communication Up' can also follow a 'Communication Lost' notification, indicating that the lost communication is restored. If the peer has provided an adaptation layer indication, an 'Adaptation Layer' notification is issued.

- o LISTEN.MPTCP:

This is similar to LISTEN.TCP except for one additional boolean parameter that allows to enable or disable MPTCP for a particular connection or socket (default: enabled).

- o LISTEN.UDP(-Lite):

Pass 1 primitive / event: 'Receive'.

Parameters: 1 local IP address (default (ANY), or specified); 1 destination transport address; local port (default (OS chooses), or specified); destination port (default (OS chooses), or specified).

Comments: The 'Receive' function registers the application to listen for incoming UDP(-Lite) datagrams at an endpoint.

MAINTENANCE:

Adjustments made to an open connection, or notifications about it. These are out-of-band messages to the protocol that can be issued at any time, at least after a connection has been established and before it has been terminated (with one exception: CHANGE_TIMEOUT.TCP can

only be issued for an open connection when DATA.SEND.TCP is called). In some cases, these primitives can also be immediately issued during ESTABLISHMENT or AVAILABILITY, without waiting for the connection to be opened (e.g. CHANGE_TIMEOUT.TCP can be done using TCP's 'Open' primitive). For UDP and UDP-Lite, these functions may establish a setting per connection, but may also be changed per datagram message.

o CHANGE_TIMEOUT.TCP:

Pass 1 primitive / event: 'Open' or 'Send' combined with unspecified control of per-connection state variables

Parameters: timeout value (optional); adv_uto (optional); boolean uto_enabled (optional, default false); boolean changeable (optional, default true)

Comments: when sending data, an application can adjust the connection's timeout value (time after which the connection will be aborted if data could not be delivered). If 'uto_enabled' is true, the 'timeout value' (or, if provided, the value 'adv_uto') will be advertised for the TCP on the other side of the connection to adapt its own user timeout accordingly. 'uto_enabled' controls whether the UTO option is enabled for a connection. This applies to both sending and receiving. 'changeable' controls whether the user timeout may be changed based on a UTO option received from the other end of the connection; it becomes false when 'timeout value' is used.

o CHANGE_TIMEOUT.SCTP:

Pass 1 primitive / event: 'Change HeartBeat' combined with 'Configure Max. Retransmissions of an Association'

Parameters: 'Change Heartbeat': heartbeat frequency; 'Configure Max. Retransmissions of an Association': association.max.retrans

Comments: 'Change Heartbeat' can enable / disable heartbeats in SCTP as well as change their frequency. The parameter 'association.max.retrans' defines after how many unsuccessful transmissions of any packets (including heartbeats) the association will be terminated; thus these two primitives / parameters together can yield a similar behavior for SCTP associations as CHANGE_TIMEOUT.TCP does for TCP connections.

- o `DISABLE_NAGLE.TCP`:

Pass 1 primitive / event: not specified

Parameters: one boolean value

Comments: the Nagle algorithm delays data transmission to increase the chance to send a full-sized segment. An application must be able to disable this algorithm for a connection.

- o `DISABLE_NAGLE.SCTP`:

Pass 1 primitive / event: 'Enable / Disable NoDelay'

Parameters: one boolean value

Comments: Nagle-like algorithms delay data transmission to increase the chance to send a full-sized packet.

- o `REQUEST_HEARTBEAT.SCTP`:

Pass 1 primitive / event: 'Request HeartBeat'

Parameters: socket

Returns: success or failure

Comments: requests an immediate heartbeat on a path, returning success or failure.

- o `ADD_PATH.MPTCP`:

Pass 1 primitive / event: not specified

Parameters: local IP address and optionally the local port number

Comments: the application specifies the local IP address and port number that must be used for a new subflow.

- o `ADD_PATH.SCTP`:

Pass 1 primitive / event: 'Change Local Address / Set Peer Primary'

Parameters: local IP address

- o REM_PATH.MPTCP:

Pass 1 primitive / event: not specified

Parameters: local IP address; local port number; remote IP address; remote port number

Comments: the application removes the subflow specified by the IP/port-pair. The MPTCP implementation must trigger a removal of the subflow that belongs to this IP/port-pair.

- o REM_PATH.SCTP:

Pass 1 primitive / event: 'Change Local Address / Set Peer Primary'

Parameters: local IP address

- o SET_PRIMARY.SCTP:

Pass 1 primitive / event: 'Set Primary'

Parameters: socket

Returns: result of attempting this operation

Comments: update the current primary address to be used, based on the set of available sockets of the association.

- o SET_PEER_PRIMARY.SCTP:

Pass 1 primitive / event: 'Change Local Address / Set Peer Primary'

Parameters: local IP address

Comments: this is only advisory for the peer.

- o CONFIG_SWITCHOVER.SCTP:

Pass 1 primitive / event: 'Configure Path Switchover'

Parameters: primary max retrans (no. of retransmissions after which a path is considered inactive); PF max retrans (no. of retransmissions after which a path is considered to be "Potentially Failed", and others will be preferably used) (optional)

o STATUS.SCTP:

Pass 1 primitive / event: 'Status', 'Enable / Disable Interleaving' and 'Network Status Change' notification.

Returns: data block with information about a specified association, containing: association connection state; destination transport address list; destination transport address reachability states; current local and peer receiver window sizes; current local congestion window sizes; number of unacknowledged DATA chunks; number of DATA chunks pending receipt; primary path; most recent SRTT on primary path; RTO on primary path; SRTT and RTO on other destination addresses; MTU per path; interleaving supported yes/no.

Comments: The 'Network Status Change' notification informs the application about a socket becoming active/inactive; this only affects the programming style, as the same information is also available via 'Status'.

o STATUS.MPTCP:

Pass 1 primitive / event: not specified

Returns: list of pairs of tuples of IP address and TCP port number of each subflow. The first of the pair is the local IP and port number, while the second is the remote IP and port number.

o SET_DSCP.TCP:

Pass 1 primitive / event: not specified

Parameters: DSCP value

Comments: this allows an application to change the DSCP value for outgoing segments.

- o SET_DSCP.SCTP:

Pass 1 primitive / event: 'Set DSCP value'

Parameters: DSCP value

Comments: this allows an application to change the DSCP value for outgoing packets on a path.

- o SET_DSCP.UDP(-Lite):

Pass 1 primitive / event: 'Set_DSCP'

Parameter: DSCP value

Comments: This allows an application to change the DSCP value for outgoing UDP(-Lite) datagrams. [RFC7657] and [RFC8085] provide current guidance on using this value with UDP.

- o ERROR.TCP:

Pass 1 primitive / event: 'Error_Report'

Returns: reason (encoding not specified); subreason (encoding not specified)

Comments: soft errors that can be ignored without harm by many applications; an application should be able to disable these notifications. The reported conditions include at least: ICMP error message arrived; Excessive Retransmissions.

- o ERROR.UDP(-Lite):

Pass 1 primitive / event: 'Error_Report'

Returns: Error report

Comments: This returns soft errors that may be ignored without harm by many applications; An application must connect to be able receive these notifications.

- o SET_AUTH.TCP:

Pass 1 primitive / event: not specified

Parameters: current_key; rnext_key

Comments: current_key and rnext_key are the preferred outgoing MKT and the preferred incoming MKT, respectively, for a segment that is sent on the connection.

- o SET_AUTH.SCTP:

Pass 1 primitive / event: 'Set / Get Authentication Parameters'

Parameters: key_id; key; hmac_id

- o GET_AUTH.TCP:

Pass 1 primitive / event: not specified

Parameters: current_key; rnext_key

Comments: current_key and rnext_key are the preferred outgoing MKT and the preferred incoming MKT, respectively, that were carried on a recently received segment.

- o GET_AUTH.SCTP:

Pass 1 primitive / event: 'Set / Get Authentication Parameters'

Parameters: key_id; chunk_list

- o RESET_STREAM.SCTP:

Pass 1 primitive / event: 'Add / Reset Streams, Reset Association'

Parameters: sid; direction

- o RESET_STREAM-EVENT.SCTP:

Pass 1 primitive / event: 'Stream Reset' notification

Parameters: information about the result of RESET_STREAM.SCTP.

Comments: This is issued when the procedure for resetting streams has completed.

- o RESET_ASSOC.SCTP:

Pass 1 primitive / event: 'Add / Reset Streams, Reset Association'

Parameters: information related to the extension, defined in [RFC3260].

- o RESET_ASSOC-EVENT.SCTP:

Pass 1 primitive / event: 'Association Reset' notification

Parameters: information about the result of RESET_ASSOC.SCTP.

Comments: this is issued when the procedure for resetting an association has completed.

- o ADD_STREAM.SCTP:

Pass 1 primitive / event: 'Add / Reset Streams, Reset Association'

Parameters: number of outgoing and incoming streams to be added

- o ADD_STREAM-EVENT.SCTP:

Pass 1 primitive / event: 'Stream Change' notification

Parameters: information about the result of ADD_STREAM.SCTP.

Comments: this is issued when the procedure for adding a stream has completed.

- o SET_STREAM_SCHEDULER.SCTP:

Pass 1 primitive / event: 'Set Stream Scheduler'

Parameters: scheduler identifier

Comments: choice of First Come First Serve, Round Robin, Round Robin per Packet, Priority Based, Fair Bandwidth, Weighted Fair Queuing.

- o CONFIGURE_STREAM_SCHEDULER.SCTP:

Pass 1 primitive / event: 'Configure Stream Scheduler'

Parameters: priority

Comments: the priority value only applies when Priority Based or Weighted Fair Queuing scheduling is chosen with SET_STREAM_SCHEDULER.SCTP. The meaning of the parameter differs between these two schedulers but in both cases it realizes some form of prioritization regarding how bandwidth is divided among streams.

- o SET_FLOWLABEL.SCTP:

Pass 1 primitive / event: 'Set IPv6 Flow Label'

Parameters: flow label

Comments: this allows an application to change the IPv6 header's flow label field for outgoing packets on a path.

- o AUTHENTICATION_NOTIFICATION-EVENT.SCTP:

Pass 1 primitive / event: 'Authentication' notification

Returns: information regarding key management.

- o CONFIG_SEND_BUFFER.SCTP:

Pass 1 primitive / event: 'Configure Send Buffer Size'

Parameters: size value in octets

- o CONFIG_RECEIVE_BUFFER.SCTP:

Pass 1 primitive / event: 'Configure Receive Buffer Size'

Parameters: size value in octets

Comments: this controls the receiver window.

- o CONFIG_FRAGMENTATION.SCTP:

Pass 1 primitive / event: 'Configure Message Fragmentation'

Parameters: one boolean value (enable/disable); maximum fragmentation size (optional; default: PMTU)

Comments: if fragmentation is enabled, messages exceeding the maximum fragmentation size will be fragmented. If fragmentation is disabled, trying to send a message that exceeds the maximum fragmentation size will produce an error.

- o CONFIG_PMTUD.SCTP:

Pass 1 primitive / event: 'Configure Path MTU Discovery'

Parameters: one boolean value (PMTUD on/off); PMTU value (optional)

Returns: PMTU value

Comments: this returns a meaningful PMTU value when PMTUD is enabled (the boolean is true), and the PMTU value can be set if PMTUD is disabled (the boolean is false)

- o CONFIG_DELAYED_SACK.SCTP:

Pass 1 primitive / event: 'Configure Delayed SACK Timer'

Parameters: one boolean value (delayed SACK on/off); timer value (optional); number of packets to wait for (default 2)

Comments: if delayed SACK is enabled, SCTP will send a SACK upon either receiving the provided number of packets or when the timer expires, whatever occurs first.

- o CONFIG_RTO.SCTP:

Pass 1 primitive / event: 'Configure RTO Calculation'

Parameters: init (optional); min (optional); max (optional)

Comments: this adjusts the initial, minimum and maximum RTO values.

- o SET_COOKIE_LIFE.SCTP:
Pass 1 primitive / event: 'Set Cookie Life Value'
Parameters: cookie life value
- o SET_MAX_BURST.SCTP:
Pass 1 primitive / event: 'Set Maximum Burst'
Parameters: max burst value
Comments: not all implementations allow values above the default of 4.
- o SET_PARTIAL_DELIVERY_POINT.SCTP:
Pass 1 primitive / event: 'Set Partial Delivery Point'
Parameters: partial delivery point (integer)
Comments: this parameter must be smaller or equal to the socket receive buffer size.
- o SET_CHECKSUM_ENABLED.UDP:
Pass 1 primitive / event: 'Checksum_Enabled'.
Parameters: 0 when zero checksum is used at sender, 1 for checksum at sender (default)
- o SET_CHECKSUM_REQUIRED.UDP:
Pass 1 primitive / event: 'Require_Checksum'.
Parameter: 0 to allow zero checksum, 1 when a non-zero checksum is required (default) at receiver
- o SET_CHECKSUM_COVERAGE.UDP-Lite:
Pass 1 primitive / event: 'Set_Checksum_Coverage'
Parameters: coverage length at sender (default maximum coverage)

- o SET_MIN_CHECKSUM_COVERAGE.UDP-Lite:
Pass 1 primitive / event: 'Set_Min_Coverage'.
Parameter: coverage length at receiver (default minimum coverage)
- o SET_DF.UDP(-Lite):
Pass 1 primitive event: 'Set_DF'.
Parameter: 0 when DF is not set (default) in the IPv4 header, 1 when DF is set
- o GET_MMS_S.UDP(-Lite):
Pass 1 primitive event: 'Get_MM_S'.
Comments: this retrieves the maximum transport-message size that may be sent using a non-fragmented IP packet from the configured interface.
- o GET_MMS_R.UDP(-Lite):
Pass 1 primitive event: 'Get_MMS_R'.
Comments: this retrieves the maximum transport-message size that may be received from the configured interface.
- o SET_TTL.UDP(-Lite) (IPV6_UNICAST_HOPS):
Pass 1 primitive / event: 'Set_TTL' and 'Set_IPV6_Unicast_Hops'
Parameters: IPv4 TTL value or IPv6 Hop Count value
Comments: this allows an application to change the IPv4 TTL of IPv6 Hop count value for outgoing UDP(-Lite) datagrams.
- o GET_TTL.UDP(-Lite) (IPV6_UNICAST_HOPS):
Pass 1 primitive / event: 'Get_TTL' and 'Get_IPV6_Unicast_Hops'
Returns: IPv4 TTL value or IPv6 Hop Count value

Comments: this allows an application to read the the IPv4 TTL of IPv6 Hop count value from a received UDP(-Lite) datagram.

o SET_ECN.UDP(-Lite):

Pass 1 primitive / event: 'Set_ECN'

Parameters: ECN value

Comments: this allows a UDP(-Lite) application to set the ECN codepoint field for outgoing UDP(-Lite) datagrams. Defaults to sending '00'.

o GET_ECN.UDP(-Lite):

Pass 1 primitive / event: 'Get_ECN'

Parameters: ECN value

Comments: this allows a UDP(-Lite) application to read the ECN codepoint field from a received UDP(-Lite) datagram.

o SET_IP_OPTIONS.UDP(-Lite):

Pass 1 primitive / event: 'Set_IP_Options'

Parameters: options

Comments: this allows a UDP(-Lite) application to set IP Options for outgoing UDP(-Lite) datagrams. These options can at least be the Source Route, Record Route, and Time Stamp option.

o GET_IP_OPTIONS.UDP(-Lite):

Pass 1 primitive / event: 'Get_IP_Options'

Returns: options

Comments: this allows a UDP(-Lite) application to receive any IP options that are contained in a received UDP(-Lite) datagram.

- o CONFIGURE.LEDBAT:

Pass 1 primitive / event: N/A

Parameters: enable (boolean); target; allowed_increase; gain_inc;
gain_dec; base_history; current_filter; init_cwnd; min_cwnd
Comments: 'enable' is a newly invented parameter that enables or
disables the whole LEDBAT service.

TERMINATION:

Gracefully or forcefully closing a connection, or being informed
about this event happening.

- o CLOSE.TCP:

Pass 1 primitive / event: 'Close'

Comments: this terminates the sending side of a connection after
reliably delivering all remaining data.

- o CLOSE.SCTP:

Pass 1 primitive / event: 'Shutdown'

Comments: this terminates a connection after reliably delivering
all remaining data.

- o ABORT.TCP:

Pass 1 primitive / event: 'Abort'

Comments: this terminates a connection without delivering
remaining data and sends an error message to the other side.

- o ABORT.SCTP:

Pass 1 primitive / event: 'Abort'

Parameters: abort reason to be given to the peer (optional)

Comments: this terminates a connection without delivering
remaining data and sends an error message to the other side.

- o ABORT.UDP(-Lite):

Pass 1 primitive event: 'Close'

Comments: this terminates a connection without delivering remaining data. No further UDP(-Lite) datagrams are sent/received for this transport service instance.

- o TIMEOUT.TCP:

Pass 1 primitive / event: 'User Timeout' event

Comments: the application is informed that the connection is aborted. This event is executed on expiration of the timeout set in CONNECTION.ESTABLISHMENT.CONNECT.TCP (possibly adjusted in CONNECTION.MAINTENANCE.CHANGE_TIMEOUT.TCP).

- o TIMEOUT.SCTP:

Pass 1 primitive / event: 'Communication Lost' event

Comments: the application is informed that the connection is aborted. this event is executed on expiration of the timeout that should be enabled by default (see the beginning of section 8.3 in [RFC4960]) and was possibly adjusted in CONNECTION.MAINTENANCE.CHANGE_TIMEOUT.SCTP.

- o ABORT-EVENT.TCP:

Pass 1 primitive / event: not specified.

- o ABORT-EVENT.SCTP:

Pass 1 primitive / event: 'Communication Lost' event

Returns: abort reason from the peer (if available)

Comments: the application is informed that the other side has aborted the connection using CONNECTION.TERMINATION.ABORT.SCTP.

- o CLOSE-EVENT.TCP:

Pass 1 primitive / event: not specified.

- o CLOSE-EVENT.SCTP:

Pass 1 primitive / event: 'Shutdown Complete' event

Comments: the application is informed that
CONNECTION.TERMINATION.CLOSE.SCTP was successfully completed.

4.2. DATA Transfer Related Primitives

All primitives in this section refer to an existing connection, i.e. a connection that was either established or made available for receiving data (although this is optional for the primitives of UDP(-Lite)). In addition to the listed parameters, all sending primitives contain a reference to a data block and all receiving primitives contain a reference to available buffer space for the data. Note that CONNECT.TCP and LISTEN.TCP in the ESTABLISHMENT and AVAILABILITY category also allow to transfer data (an optional user message) before the connection is fully established.

- o SEND.TCP:

Pass 1 primitive / event: 'Send'

Parameters: timeout (optional); current_key (optional); rnext_key (optional)

Comments: this gives TCP a data block for reliable transmission to the TCP on the other side of the connection. The timeout can be configured with this call (see also CONNECTION.MAINTENANCE.CHANGE_TIMEOUT.TCP). 'current_key' and 'rnext_key' are authentication parameters that can be configured with this call (see also CONNECTION.MAINTENANCE.SET_AUTH.TCP).

- o SEND.SCTP:

Pass 1 primitive / event: 'Send'

Parameters: stream number; context (optional); socket (optional); unordered flag (optional); no-bundle flag (optional); payload protocol-id (optional); pr-policy (optional) pr-value (optional); sack-immediately flag (optional); key-id (optional)

Comments: this gives SCTP a data block for transmission to the SCTP on the other side of the connection (SCTP association). The 'stream number' denotes the stream to be used. The 'context'

number can later be used to refer to the correct message when an error is reported. The 'socket' can be used to state which path should be preferred, if there are multiple paths available (see also CONNECTION.MAINTENANCE.SETPRIMARY.SCTP). The data block can be delivered out-of-order if the 'unordered flag' is set. The 'no-bundle flag' can be set to indicate a preference to avoid bundling. The 'payload protocol-id' is a number that will, if provided, be handed over to the receiving application. Using pr-policy and pr-value the level of reliability can be controlled. The 'sack-immediately' flag can be used to indicate that the peer should not delay the sending of a SACK corresponding to the provided user message. If specified, the provided key-id is used for authenticating the user message.

- o SEND.UDP(-Lite):

Pass 1 primitive / event: 'Send'

Parameters: IP Address and Port Number of the destination endpoint (optional if connected)

Comments: this provides a message for unreliable transmission using UDP(-Lite) to the specified transport address. IP address and Port may be omitted for connected UDP(-Lite) sockets. All CONNECTION.MAINTENANCE.SET_*.UDP(-Lite) primitives apply per message sent.

- o RECEIVE.TCP:

Pass 1 primitive / event: 'Receive'.

Parameters: current_key (optional); rnext_key (optional)

Comments: 'current_key' and 'rnext_key' are authentication parameters that can be read with this call (see also CONNECTION.MAINTENANCE.GET_AUTH.TCP).

- o RECEIVE.SCTP:

Pass 1 primitive / event: 'Data Arrive' notification, followed by 'Receive'

Parameters: stream number (optional)

Returns: stream sequence number (optional); partial flag

(optional)

Comments: if the 'stream number' is provided, the call to receive only receives data on one particular stream. If a partial message arrives, this is indicated by the 'partial flag', and then the 'stream sequence number' must be provided such that an application can restore the correct order of data blocks that comprise an entire message.

o RECEIVE.UDP(-Lite):

Pass 1 primitive / event: 'Receive',

Parameters: buffer for received datagram

Comments: all CONNECTION.MAINTENANCE.GET_*.UDP(-Lite) primitives apply per message received.

o SENDFAILURE-EVENT.SCTP:

Pass 1 primitive / event: 'Send Failure' notification, optionally followed by 'Receive Unsent Message' or 'Receive Unacknowledged Message'

Returns: cause code; context; unsent or unacknowledged message (optional)

Comments: 'cause code' indicates the reason of the failure, and 'context' is the context number if such a number has been provided in DATA.SEND.SCTP, for later use with 'Receive Unsent Message' or 'Receive Unacknowledged Message', respectively. These primitives can be used to retrieve the unsent or unacknowledged message (or part of the message, in case a part was delivered) if desired.

o SEND_FAILURE.UDP(-Lite):

Pass 1 primitive / event: 'Send'

Comments: this may be used to probe for the effective PMTU when using in combination with the 'MAINTENANCE.SET_DF' primitive.

o SENDER_DRY-EVENT.SCTP:

Pass 1 primitive / event: 'Sender Dry' notification

Comments: this informs the application that the stack has no more user data to send.

- o PARTIAL_DELIVERY_ABORTED-EVENT.SCTP:

Pass 1 primitive / event: 'Partial Delivery Aborted' notification

Comments: this informs the receiver of a partial message that the further delivery of the message has been aborted.

5. Pass 3

This section presents the superset of all transport features in all protocols that were discussed in the preceding sections, based on the list of primitives in pass 2 but also on text in pass 1 to include transport features that can be configured in one protocol and are static properties in another (congestion control, for example). Again, some minor details are omitted for the sake of generalization -- e.g., TCP may provide various different IP options, but only source route is mandatory to implement, and this detail is not visible in the Pass 3 transport feature "Specify IP Options". As before, "UDP(-Lite)" represents both UDP and UDP-Lite, and TCP refers to both TCP and MPTCP.

5.1. CONNECTION Related Transport Features

ESTABLISHMENT:

Active creation of a connection from one transport endpoint to one or more transport endpoints.

- o Connect
Protocols: TCP, SCTP, UDP(-Lite)
- o Specify which IP Options must always be used
Protocols: TCP, UDP(-Lite)
- o Request multiple streams
Protocols: SCTP
- o Limit the number of inbound streams
Protocols: SCTP

- o Specify number of attempts and/or timeout for the first establishment message
Protocols: TCP, SCTP
- o Obtain multiple sockets
Protocols: SCTP
- o Disable MPTCP
Protocols: MPTCP
- o Configure authentication
Protocols: TCP, SCTP
Comments: with TCP, this allows to configure Master Key Tuples (MKTs). In SCTP, this allows to specify which chunk types must always be authenticated. DATA, ACK etc. are different 'chunks' in SCTP; one or more chunks may be included in a single packet.
- o Indicate an Adaptation Layer (via an adaptation code point)
Protocols: SCTP
- o Request to negotiate interleaving of user messages
Protocols: SCTP
- o Hand over a message to reliably transfer (possibly multiple times) before connection establishment
Protocols: TCP
- o Hand over a message to reliably transfer during connection establishment
Protocols: SCTP
- o Enable UDP encapsulation with a specified remote UDP port number
Protocols: SCTP

AVAILABILITY:

Preparing to receive incoming connection requests.

- o Listen, 1 specified local interface
Protocols: TCP, SCTP, UDP(-Lite)
- o Listen, N specified local interfaces
Protocols: SCTP
- o Listen, all local interfaces
Protocols: TCP, SCTP, UDP(-Lite)

- o Obtain requested number of streams
Protocols: SCTP
- o Limit the number of inbound streams
Protocols: SCTP
- o Specify which IP Options must always be used
Protocols: TCP, UDP(-Lite)
- o Disable MPTCP
Protocols: MPTCP
- o Configure authentication
Protocols: TCP, SCTP
Comments: with TCP, this allows to configure Master Key Tuples (MKTs). In SCTP, this allows to specify which chunk types must always be authenticated. DATA, ACK etc. are different 'chunks' in SCTP; one or more chunks may be included in a single packet.
- o Indicate an Adaptation Layer (via an adaptation code point)
Protocols: SCTP

MAINTENANCE:

Adjustments made to an open connection, or notifications about it.

- o Change timeout for aborting connection (using retransmit limit or time value)
Protocols: TCP, SCTP
- o Suggest timeout to the peer
Protocols: TCP
- o Disable Nagle algorithm
Protocols: TCP, SCTP
- o Request an immediate heartbeat, returning success/failure
Protocols: SCTP
- o Notification of Excessive Retransmissions (early warning below abortion threshold)
Protocols: TCP
- o Add path
Protocols: MPTCP, SCTP
MPTCP Parameters: source-IP; source-Port; destination-IP; destination-Port
SCTP Parameters: local IP address

- o Remove path
Protocols: MPTCP, SCTP
MPTCP Parameters: source-IP; source-Port; destination-IP;
destination-Port
SCTP Parameters: local IP address
- o Set primary path
Protocols: SCTP
- o Suggest primary path to the peer
Protocols: SCTP
- o Configure Path Switchover
Protocols: SCTP
- o Obtain status (query or notification)
Protocols: SCTP, MPTCP
SCTP parameters: association connection state; destination
transport address list; destination transport address reachability
states; current local and peer receiver window sizes; current
local congestion window sizes; number of unacknowledged DATA
chunks; number of DATA chunks pending receipt; primary path; most
recent SRTT on primary path; RTO on primary path; SRTT and RTO on
other destination addresses; MTU per path; interleaving supported
yes/no
MPTCP parameters: subflow-list (identified by source-IP; source-
Port; destination-IP; destination-Port)
- o Specify DSCP field
Protocols: TCP, SCTP, UDP(-Lite)
- o Notification of ICMP error message arrival
Protocols: TCP, UDP(-Lite)
- o Change authentication parameters
Protocols: TCP, SCTP
- o Obtain authentication information
Protocols: TCP, SCTP
- o Reset Stream
Protocols: SCTP
- o Notification of Stream Reset
Protocols: STCP

- o Reset Association
Protocols: SCTP
- o Notification of Association Reset
Protocols: STCP
- o Add Streams
Protocols: SCTP
- o Notification of Added Stream
Protocols: STCP
- o Choose a scheduler to operate between streams of an association
Protocols: SCTP
- o Configure priority or weight for a scheduler
Protocols: SCTP
- o Specify IPv6 flow label field
Protocols: SCTP
- o Configure send buffer size
Protocols: SCTP
- o Configure receive buffer (and rwnd) size
Protocols: SCTP
- o Configure message fragmentation
Protocols: SCTP
- o Configure PMTUD
Protocols: SCTP
- o Configure delayed SACK timer
Protocols: SCTP
- o Set Cookie life value
Protocols: SCTP
- o Set maximum burst
Protocols: SCTP
- o Configure size where messages are broken up for partial delivery
Protocols: SCTP
- o Disable checksum when sending
Protocols: UDP

- o Disable checksum requirement when receiving
Protocols: UDP
- o Specify checksum coverage used by the sender
Protocols: UDP-Lite
- o Specify minimum checksum coverage required by receiver
Protocols: UDP-Lite
- o Specify DF field
Protocols: UDP(-Lite)
- o Get max. transport-message size that may be sent using a non-fragmented IP packet from the configured interface
Protocols: UDP(-Lite)
- o Get max. transport-message size that may be received from the configured interface
Protocols: UDP(-Lite)
- o Specify TTL/Hop count field
Protocols: UDP(-Lite)
- o Obtain TTL/Hop count field
Protocols: UDP(-Lite)
- o Specify ECN field
Protocols: UDP(-Lite)
- o Obtain ECN field
Protocols: UDP(-Lite)
- o Specify IP Options
Protocols: UDP(-Lite)
- o Obtain IP Options
Protocols: UDP(-Lite)
- o Enable and configure "Low Extra Delay Background Transfer"
Protocols: A protocol implementing the LEDBAT congestion control mechanism

TERMINATION:

Gracefully or forcefully closing a connection, or being informed about this event happening.

- o Close after reliably delivering all remaining data, causing an event informing the application on the other side
Protocols: TCP, SCTP
Comments: a TCP endpoint locally only closes the connection for sending; it may still receive data afterwards.
- o Abort without delivering remaining data, causing an event informing the application on the other side
Protocols: TCP, SCTP
Comments: in SCTP a reason can optionally be given by the application on the aborting side, which can then be received by the application on the other side.
- o Abort without delivering remaining data, not causing an event informing the application on the other side
Protocols: UDP(-Lite)
- o Timeout event when data could not be delivered for too long
Protocols: TCP, SCTP
Comments: the timeout is configured with CONNECTION.MAINTENANCE "Change timeout for aborting connection (using retransmit limit or time value)".

5.2. DATA Transfer Related Transport Features

All transport features in this section refer to an existing connection, i.e. a connection that was either established or made available for receiving data. Note that TCP allows to transfer data (a single optional user message, possibly arriving multiple times) before the connection is fully established. Reliable data transfer entails delay -- e.g. for the sender to wait until it can transmit data, or due to retransmission in case of packet loss.

5.2.1. Sending Data

All transport features in this section are provided by DATA.SEND from pass 2. DATA.SEND is given a data block from the application, which we here call a "message" if the beginning and end of the data block can be identified at the receiver, and "data" otherwise.

- o Reliably transfer data, with congestion control
Protocols: TCP
- o Reliably transfer a message, with congestion control
Protocols: SCTP

- o Unreliably transfer a message, with congestion control
Protocols: SCTP
- o Unreliably transfer a message, without congestion control
Protocols: UDP(-Lite)
- o Configurable Message Reliability
Protocols: SCTP
- o Choice of stream
Protocols: SCTP
- o Choice of path (destination address)
Protocols: SCTP
- o Ordered message delivery (potentially slower than unordered)
Protocols: SCTP
- o Unordered message delivery (potentially faster than ordered)
Protocols: SCTP, UDP(-Lite)
- o Request not to bundle messages
Protocols: SCTP
- o Specifying a "payload protocol-id" (handed over as such by the receiver)
Protocols: SCTP
- o Specifying a key id to be used to authenticate a message
Protocols: SCTP
- o Request not to delay the acknowledgement (SACK) of a message
Protocols: SCTP

5.2.2. Receiving Data

All transport features in this section are provided by DATA.RECEIVE from pass 2. DATA.RECEIVE fills a buffer provided by the application, with what we here call a "message" if the beginning and end of the data block can be identified at the receiver, and "data" otherwise.

- o Receive data (with no message delimiting)
Protocols: TCP

- o Receive a message
Protocols: SCTP, UDP(-Lite)
- o Choice of stream to receive from
Protocols: SCTP
- o Information about partial message arrival
Protocols: SCTP
Comments: in SCTP, partial messages are combined with a stream sequence number so that the application can restore the correct order of data blocks an entire message consists of.

5.2.3. Errors

This section describes sending failures that are associated with a specific call to DATA.SEND from pass 2.

- o Notification of an unsent (part of a) message
Protocols: SCTP, UDP(-Lite)
- o Notification of an unacknowledged (part of a) message
Protocols: SCTP
- o Notification that the stack has no more user data to send
Protocols: SCTP
- o Notification to a receiver that a partial message delivery has been aborted
Protocols: SCTP

6. Acknowledgements

The authors would like to thank (in alphabetical order) Bob Briscoe, Spencer Dawkins, Aaron Falk, David Hayes, Karen Nielsen, Tommy Pauly, Joe Touch and Brian Trammell for providing valuable feedback on this document. We especially thank Christoph Paasch for providing input related to Multipath TCP, and Gorry Fairhurst and Tom Jones for providing input related to UDP(-Lite). This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 644334 (NEAT).

7. IANA Considerations

This memo includes no request to IANA.

8. Security Considerations

Authentication, confidentiality protection, and integrity protection are identified as transport features [RFC8095]. These transport features are generally provided by a protocol or layer on top of the transport protocol; none of the transport protocols considered in this document provides these transport features on its own. Therefore, these transport features are not considered in this document, with the exception of native authentication capabilities of TCP and SCTP for which the security considerations in [RFC5925] and [RFC4895] apply.

Security considerations for the use of UDP and UDP-Lite are provided in the referenced RFCs. Security guidance for application usage is provided in the UDP-Guidelines [RFC8085].

9. References

9.1. Normative References

- [FJ16] Fairhurst, G. and T. Jones, "Features of the User Datagram Protocol (UDP) and Lightweight UDP (UDP-Lite) Transport Protocols", draft-ietf-taps-transport-usage-udp-04 (work in progress), July 2017.
- [I-D.ietf-tsvwg-sctp-ndata] Stewart, R., Tuexen, M., Loreto, S., and R. Seggelmann, "Stream Schedulers and User Message Interleaving for the Stream Control Transmission Protocol", draft-ietf-tsvwg-sctp-ndata-08 (work in progress), October 2016.
- [RFC0793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, DOI 10.17487/RFC0793, September 1981, <<https://www.rfc-editor.org/info/rfc793>>.
- [RFC1122] Braden, R., Ed., "Requirements for Internet Hosts - Communication Layers", STD 3, RFC 1122, DOI 10.17487/RFC1122, October 1989, <<https://www.rfc-editor.org/info/rfc1122>>.
- [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, <<https://www.rfc-editor.org/info/rfc3758>>.

- [RFC4895] Tuexen, M., Stewart, R., Lei, P., and E. Rescorla, "Authenticated Chunks for the Stream Control Transmission Protocol (SCTP)", RFC 4895, DOI 10.17487/RFC4895, August 2007, <<https://www.rfc-editor.org/info/rfc4895>>.
- [RFC4960] Stewart, R., Ed., "Stream Control Transmission Protocol", RFC 4960, DOI 10.17487/RFC4960, September 2007, <<https://www.rfc-editor.org/info/rfc4960>>.
- [RFC5061] Stewart, R., Xie, Q., Tuexen, M., Maruyama, S., and M. Kozuka, "Stream Control Transmission Protocol (SCTP) Dynamic Address Reconfiguration", RFC 5061, DOI 10.17487/RFC5061, September 2007, <<https://www.rfc-editor.org/info/rfc5061>>.
- [RFC5482] Eggert, L. and F. Gont, "TCP User Timeout Option", RFC 5482, DOI 10.17487/RFC5482, March 2009, <<https://www.rfc-editor.org/info/rfc5482>>.
- [RFC5925] Touch, J., Mankin, A., and R. Bonica, "The TCP Authentication Option", RFC 5925, DOI 10.17487/RFC5925, June 2010, <<https://www.rfc-editor.org/info/rfc5925>>.
- [RFC6182] Ford, A., Raiciu, C., Handley, M., Barre, S., and J. Iyengar, "Architectural Guidelines for Multipath TCP Development", RFC 6182, DOI 10.17487/RFC6182, March 2011, <<https://www.rfc-editor.org/info/rfc6182>>.
- [RFC6458] Stewart, R., Tuexen, M., Poon, K., Lei, P., and V. Yasevich, "Sockets API Extensions for the Stream Control Transmission Protocol (SCTP)", RFC 6458, DOI 10.17487/RFC6458, December 2011, <<https://www.rfc-editor.org/info/rfc6458>>.
- [RFC6525] Stewart, R., Tuexen, M., and P. Lei, "Stream Control Transmission Protocol (SCTP) Stream Reconfiguration", RFC 6525, DOI 10.17487/RFC6525, February 2012, <<https://www.rfc-editor.org/info/rfc6525>>.
- [RFC6817] Shalunov, S., Hazel, G., Iyengar, J., and M. Kuehlewind, "Low Extra Delay Background Transport (LEDBAT)", RFC 6817, DOI 10.17487/RFC6817, December 2012, <<https://www.rfc-editor.org/info/rfc6817>>.
- [RFC6824] Ford, A., Raiciu, C., Handley, M., and O. Bonaventure, "TCP Extensions for Multipath Operation with Multiple Addresses", RFC 6824, DOI 10.17487/RFC6824, January 2013, <<https://www.rfc-editor.org/info/rfc6824>>.

- [RFC6897] Scharf, M. and A. Ford, "Multipath TCP (MPTCP) Application Interface Considerations", RFC 6897, DOI 10.17487/RFC6897, March 2013, <<https://www.rfc-editor.org/info/rfc6897>>.
- [RFC6951] Tuexen, M. and R. Stewart, "UDP Encapsulation of Stream Control Transmission Protocol (SCTP) Packets for End-Host to End-Host Communication", RFC 6951, DOI 10.17487/RFC6951, May 2013, <<https://www.rfc-editor.org/info/rfc6951>>.
- [RFC7053] Tuexen, M., Ruengeler, I., and R. Stewart, "SACK-IMMEDIATELY Extension for the Stream Control Transmission Protocol", RFC 7053, DOI 10.17487/RFC7053, November 2013, <<https://www.rfc-editor.org/info/rfc7053>>.
- [RFC7413] Cheng, Y., Chu, J., Radhakrishnan, S., and A. Jain, "TCP Fast Open", RFC 7413, DOI 10.17487/RFC7413, December 2014, <<https://www.rfc-editor.org/info/rfc7413>>.
- [RFC7496] Tuexen, M., Seggelmann, R., Stewart, R., and S. Loreto, "Additional Policies for the Partially Reliable Stream Control Transmission Protocol Extension", RFC 7496, DOI 10.17487/RFC7496, April 2015, <<https://www.rfc-editor.org/info/rfc7496>>.
- [RFC7829] Nishida, Y., Natarajan, P., Caro, A., Amer, P., and K. Nielsen, "SCTP-PF: A Quick Failover Algorithm for the Stream Control Transmission Protocol", RFC 7829, DOI 10.17487/RFC7829, April 2016, <<https://www.rfc-editor.org/info/rfc7829>>.
- [RFC8085] Eggert, L., Fairhurst, G., and G. Shepherd, "UDP Usage Guidelines", BCP 145, RFC 8085, DOI 10.17487/RFC8085, March 2017, <<https://www.rfc-editor.org/info/rfc8085>>.

9.2. Informative References

- [I-D.draft-gjessing-taps-minset] Gjessing, S. and M. Welzl, "A Minimal Set of Transport Services for TAPS Systems", draft-gjessing-taps-minset-05 (work in progress), June 2017.
- [RFC0854] Postel, J. and J. Reynolds, "Telnet Protocol Specification", STD 8, RFC 854, DOI 10.17487/RFC0854, May 1983, <<https://www.rfc-editor.org/info/rfc854>>.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/

- RFC2119, March 1997,
<<https://www.rfc-editor.org/info/rfc2119>>.
- [RFC2474] Nichols, K., Blake, S., Baker, F., and D. Black,
"Definition of the Differentiated Services Field (DS
Field) in the IPv4 and IPv6 Headers", RFC 2474,
DOI 10.17487/RFC2474, December 1998,
<<https://www.rfc-editor.org/info/rfc2474>>.
- [RFC2475] Blake, S., Black, D., Carlson, M., Davies, E., Wang, Z.,
and W. Weiss, "An Architecture for Differentiated
Services", RFC 2475, DOI 10.17487/RFC2475, December 1998,
<<https://www.rfc-editor.org/info/rfc2475>>.
- [RFC3260] Grossman, D., "New Terminology and Clarifications for
Diffserv", RFC 3260, DOI 10.17487/RFC3260, April 2002,
<<https://www.rfc-editor.org/info/rfc3260>>.
- [RFC5461] Gont, F., "TCP's Reaction to Soft Errors", RFC 5461,
DOI 10.17487/RFC5461, February 2009,
<<https://www.rfc-editor.org/info/rfc5461>>.
- [RFC6093] Gont, F. and A. Yourtchenko, "On the Implementation of the
TCP Urgent Mechanism", RFC 6093, DOI 10.17487/RFC6093,
January 2011, <<https://www.rfc-editor.org/info/rfc6093>>.
- [RFC7414] Duke, M., Braden, R., Eddy, W., Blanton, E., and A.
Zimmermann, "A Roadmap for Transmission Control Protocol
(TCP) Specification Documents", RFC 7414, DOI 10.17487/
RFC7414, February 2015,
<<https://www.rfc-editor.org/info/rfc7414>>.
- [RFC7657] Black, D., Ed. and P. Jones, "Differentiated Services
(Diffserv) and Real-Time Communication", RFC 7657,
DOI 10.17487/RFC7657, November 2015,
<<https://www.rfc-editor.org/info/rfc7657>>.
- [RFC8095] Fairhurst, G., Ed., Trammell, B., Ed., and M. Kuehlewind,
Ed., "Services Provided by IETF Transport Protocols and
Congestion Control Mechanisms", RFC 8095, DOI 10.17487/
RFC8095, March 2017,
<<https://www.rfc-editor.org/info/rfc8095>>.

Appendix A. Overview of RFCs used as input for pass 1

TCP: [RFC0793], [RFC1122], [RFC5482], [RFC5925], [RFC7413]
MPTCP: [RFC6182], [RFC6824], [RFC6897]
SCTP: RFCs without a socket API specification: [RFC3758], [RFC4895],
[RFC4960], [RFC5061].
RFCs that include a socket API specification: [RFC6458],
[RFC6525], [RFC6951], [RFC7053], [RFC7496] [RFC7829].
UDP(-Lite): See [FJ16]
LEDBAT: [RFC6817].

Appendix B. How this document was developed

This section gives an overview of the method that was used to develop this document. It was given to contributors for guidance, and it can be helpful for future updates or extensions.

This document is only concerned with transport features that are explicitly exposed to applications via primitives. It also strictly follows RFC text: if a transport feature is truly relevant for an application, the RFCs should say so, and they should describe how to use and configure it. Thus, the approach followed for developing this document was to identify the right RFCs, then analyze and process their text.

Primitives that "MAY" be implemented by a transport protocol were excluded. To be included, the minimum requirement level for a primitive to be implemented by a protocol was "SHOULD". Where [RFC2119]-style requirements levels are not used, primitives were excluded when they are described in conjunction with statements like, e.g.: "some implementations also provide" or "an implementation may also". Excluded primitives or parameters were briefly described in a dedicated subsection.

Pass 1: This began by identifying text that talks about primitives. An API specification, abstract or not, obviously describes primitives -- but we are not *only* interested in API specifications. The text describing the 'send' primitive in the API specified in [RFC0793], for instance, does not say that data transfer is reliable. TCP's reliability is clear, however, from this text in Section 1 of [RFC0793]: "The Transmission Control Protocol (TCP) is intended for use as a highly reliable host-to-host protocol between hosts in packet-switched computer communication networks, and in interconnected systems of such networks."

Some text for pass 1 subsections was developed copy+pasting all the relevant text parts from the relevant RFCs, then adjusting terminology to match the terminology in Section 1 and adjusting (shortening!) phrasing to match the general style of the document.

An effort was made to formulate everything as a primitive description such that the primitive descriptions became as complete as possible (e.g., the "SEND.TCP" primitive in pass 2 is explicitly described as reliably transferring data); text that is relevant for the primitives presented in this pass but still does not fit directly under any primitive was used in a subsection's introduction.

Pass 2: The main goal of this pass is unification of primitives. As input, only text from pass 1 was used (no exterior sources). The list in pass 2 is not arranged by protocol ("first protocol X, here are all the primitives; then protocol Y, here are all the primitives, ..") but by primitive ("primitive A, implemented this way in protocol X, this way in protocol Y, ..."). It was a goal to obtain as many similar pass 2 primitives as possible. For instance, this was sometimes achieved by not always maintaining a 1:1 mapping between pass 1 and pass 2 primitives, renaming primitives etc. For every new primitive, the already existing primitives were considered to try to make them as coherent as possible.

For each primitive, the following style was used:

```
o PRIMITIVENAME.PROTOCOL:
  Pass 1 primitive / event:
  Parameters:
  Returns:
  Comments:
```

The entries "Parameters", "Returns" and "Comments" were skipped when a primitive had no parameters, no described return value or no comments seemed necessary, respectively. Optional parameters are followed by "(optional)". When a default value is known, this was also provided.

Pass 3: the main point of this pass is to identify transport features that are the result of static properties of protocols, for which all protocols have to be listed together; this is then the final list of all available transport features. This list was primarily based on text from pass 2, with additional input from pass 1 (but no external sources).

Appendix C. Revision information

XXX RFC-Ed please remove this section prior to publication.

-00 (from draft-welzl-taps-transports): this now covers TCP based on all TCP RFCs (this means: if you know of something in any TCP RFC that you think should be addressed, please speak up!) as well as

SCTP, exclusively based on [RFC4960]. We decided to also incorporate [RFC6458] for SCTP, but this hasn't happened yet. Terminology made in line with [RFC8095]. Addressed comments by Karen Nielsen and Gorrry Fairhurst; various other fixes. Appendices (TCP overview and how-to-contribute) added.

-01: this now also covers MPTCP based on [RFC6182], [RFC6824] and [RFC6897].

-02: included UDP, UDP-Lite, and all extensions of SCTPs. This includes fixing the [RFC6458] omission from -00.

-03: wrote security considerations. The "how to contribute" section was updated to reflect how the document *was* created, not how it *should be* created; it also no longer wrongly says that Experimental RFCs are excluded. Included LEDBAT. Changed abstract and intro to reflect which protocols/mechanisms are covered (TCP, MPTCP, SCTP, UDP, UDP-Lite, LEDBAT) instead of talking about "transport protocols". Interleaving and stream scheduling added (draft-ietf-tsvwg-sctp-ndata). TFO added. "Set protocol parameters" in SCTP replaced with per-parameter (or parameter group) primitives. More primitives added, mostly previously overlooked ones from [RFC6458]. Updated terminology (s/transport service feature/transport feature) in line with an update of [RFC8095]. Made sequence of transport features / primitives more logical. Combined MPTCP's add/rem subflow with SCTP's add/remove local address.

-04: changed UDP's close into an ABORT (to better fit with the primitives of TCP and SCTP), and incorporated the corresponding transport feature in step 3 (this addresses a comment from Gorrry Fairhurst). Added TCP Authentication (RFC 5925, section 7.1). Changed TFO from looking like a primitive in pass 1 to be a part of 'open'. Changed description of SCTP authentication in pass 3 to encompass both TCP and SCTP. Added citations of [RFC8095] and minset [I-D.draft-gjessing-taps-minset] to the intro, to give the context of this document.

-05: minor fix to TCP authentication (comment from Joe Touch), several fixes from Gorrry Fairhurst and Tom Jones. Language fixes; updated to align with latest taps-transport-usage-udp ID.

-06: addressed WGLC comments from Aaron Falk and Tommy Pauly.

-07: addressed AD review comments from Spencer Dawkins.

-08: removed "delivery number" which was based on an error in RFC 4960: <https://tools.ietf.org/html/draft-ietf-tsvwg-rfc4960-errata-02#section-3.34>.

-09: for consistency with the draft-ietf-taps-minset-00, adjusted the following transport features in "pass 3": "Choice between unordered (potentially faster) or ordered delivery of messages" divided into two transport features (one for unordered, one for ordered); the word "reliably" was added to the transport features "Hand over a message to reliably transfer (possibly multiple times) before connection establishment" and "Hand over a message to reliably transfer during connection establishment". Fixed RFC2119-style language into explicit citations (comment by Eric Rescorla and others). Addressed editorial comments by Mirja Kuehlewind, Ben Campbell, Benoit Claise and the Gen-ART reviewer Roni Even, except for moving terminology section after the intro because the terminology is already used in the intro text.

Authors' Addresses

Michael Welzl
University of Oslo
PO Box 1080 Blindern
Oslo, N-0316
Norway

Email: michawe@ifi.uio.no

Michael Tuexen
Muenster University of Applied Sciences
Stegerwaldstrasse 39
Steinfurt 48565
Germany

Email: tuexen@fh-muenster.de

Naeem Khademi
University of Oslo
PO Box 1080 Blindern
Oslo, N-0316
Norway

Email: naeemk@ifi.uio.no

Internet Engineering Task Force
Internet-Draft
Intended status: Informational
Expires: March 23, 2018

G. Fairhurst
T. Jones
University of Aberdeen
September 19, 2017

Features of the User Datagram Protocol (UDP) and Lightweight UDP (UDP-
Lite) Transport Protocols
draft-ietf-taps-transports-usage-udp-07

Abstract

This is an informational document that describes the transport protocol interface primitives provided by the User Datagram Protocol (UDP) and the Lightweight User Datagram Protocol (UDP-Lite) transport protocols. It identifies the datagram services exposed to applications and how an application can configure and use the features offered by the Internet datagram transport service. RFCxxxx documents the usage of transport features provided by IETF transport protocols, describing the way UDP, UDP-Lite and other transport protocols expose their services to applications and how an application can configure and use the features that make up these services. This document provides input to and context for that document, as well as offering a road map to documentation that may be of help to users of the UDP and UDP-Lite protocols.

XXX RFC-Ed Note - please replace RFCxxxx with the published RFC number for I-D.ietf-taps-transports-usage, when these documents are both published XXX.

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 <https://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 March 23, 2018.

Copyright Notice

Copyright (c) 2017 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 (<https://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. Terminology	4
3. UDP and UDP-Lite Primitives	4
3.1. Primitives Provided by UDP	4
3.1.1. Excluded Primitives	11
3.2. Primitives Provided by UDP-Lite	11
4. Acknowledgements	12
5. IANA Considerations	12
6. Security Considerations	12
7. References	13
7.1. Normative References	13
7.2. Informative References	14
Appendix A. Multicast Primitives	16
Appendix B. Revision Notes	19
Authors' Addresses	22

1. Introduction

This document presents defined interactions between transport protocols and applications in the form of 'primitives' (function calls) for the User Datagram Protocol (UDP) [RFC0768] and the Lightweight User Datagram Protocol (UDP-Lite) [RFC3828]. In this usage, the word application refers to any program built on the datagram interface, and including tunnels and other upper layer protocols that use UDP and UDP-Lite.

UDP is widely implemented and deployed. It is used for a wide range of applications. A special class of applications can derive benefit from having partially damaged payloads delivered, rather than discarded, when using paths that include error-prone links. Applications that can tolerate payload corruption can choose to use

UDP-Lite instead of UDP and use the application programming interface (API) to control checksum protection. Conversely, UDP applications could choose to use UDP-Lite, but this is currently less widely deployed and users could encounter paths that do not support UDP-Lite. These topics are discussed more in section 3.4 of the UDP Usage Guidelines [RFC8085].

The IEEE standard API for TCP/IP applications is the "socket" interface [POSIX]. An application can use the `recv()` and `send()` POSIX functions as well as the `recvfrom()` and `sendto()` and `recvmsg()` and `sendmsg()` functions. The UDP and UDP-Lite sockets API differs from that for TCP in several key ways. (Examples of usage of this API are provided in [STEVENS].) In UDP and UDP-Lite, each datagram is a self-contained message of a specified length, and options at the transport layer can be used to set properties for all subsequent datagrams sent using a socket or changed for each datagram. For datagrams, this can require the application to use the API to set IP-level information (the IP Time To Live (TTL), Differentiated Services (DiffServ) Code Point, IP fragmentation, etc) for the datagrams it sends and receives. In contrast, when using TCP and other connection-oriented transports, the IP-level information normally either remains the same for the duration of a connection or is controlled by the transport protocol rather than the application.

Socket options are used in the socket API to provide additional functions. For example, the `IP_RECVTTL` socket option is used by some UDP multicast applications to return the IP TTL field from IP header of a received datagram.

Some platforms also offer applications the ability to directly assemble and transmit IP packets through "raw sockets" or similar facilities. The raw socket API is a second, more cumbersome, method to send UDP datagrams. The use of this API is discussed in the RFC series in the UDP Guidelines [RFC8085].

The list of transport service features and primitives in this document is strictly based on the parts of protocol specifications in RFC-series that relate to what the transport protocol provides to an application that uses it and how the application interacts with the transport protocol. Primitives can be invoked by an application or a transport protocol; the latter type is called an "event".

The description in Section 3 follows the methodology defined by the IETF TAPS working group in [I-D.ietf-taps-transports-usage]. Specifically, this document provides the first pass of this process, which discusses the relevant RFC text describing primitives for each protocol. [I-D.ietf-taps-transports-usage] uses this input to document the usage of transport features provided by IETF transport

protocols, describing the way UDP, UDP-Lite and other transport protocols expose their services to applications and how an application can configure and use the features that make up these services.

The presented road map to documentation of the transport interface may also help developers working with UDP and UDP-Lite.

2. Terminology

This document provides details for the Pass 1 analysis of UDP and UDP-Lite that is used in "Usage of Transport Features Provided by IETF Transport Protocols" [I-D.ietf-taps-transports-usage]. It uses common terminology defined in that document and also quotes RFCs that use the terminology of RFC 2119 [RFC2119].

3. UDP and UDP-Lite Primitives

The User Datagram Protocol (UDP) [RFC0768][RFC8200] and UDP-Lite protocols [RFC3828] are IETF standards track transport protocols. These protocols provide unidirectional, datagram services, supporting transmit and receive operations that preserve message boundaries.

This section summarises the relevant text parts of the RFCs describing the UDP and UDP-Lite protocols, focusing on what the transport protocols provide to the application and how the transport is used (based on abstract API descriptions, where they are available). It describes how UDP is used with IPv4 or IPv6 to send unicast or anycast datagrams and the use to send broadcast datagrams for IPv4. A set of network-layer primitives required to use UDP or UDP-Lite with IP multicast (for IPv4 and IPv6) have been specified in the RFC series. Appendix A describes where to find documentation for network-layer primitives required to use UDP or UDP-Lite with IP multicast (for IPv4 and IPv6).

3.1. Primitives Provided by UDP

The User Datagram Protocol (UDP) [RFC0768] States: "This User Datagram Protocol (UDP) is defined to make available a datagram mode of packet-switched computer communication in the environment of an interconnected set of computer networks." It "provides a procedure for application programs to send messages to other programs with a minimum of protocol mechanism (...)".

The User Interface section of RFC768 states that the user interface to an application should allow "the creation of new receive ports, receive operations on the receive ports that return the data octets and an indication of source port and source address, and an operation

that allows a datagram to be sent, specifying the data, source and destination ports and addresses to be sent".

UDP has been defined for IPv6 [RFC8200], together with API extensions for a Basic Socket Interface Extensions for IPv6 [RFC3493]. [RFC6935] and [RFC6936] define an update to the UDP transport originally specified in RFC2460. This enables use of a zero UDP checksum mode with a tunnel protocol, providing that the method satisfies the requirements in the corresponding applicability statement [RFC6936].

UDP offers only a basic transport interface. UDP datagrams may be directly sent and received, without exchanging messages between the endpoints to setup a connection (i.e., no handshake is performed by the transport protocol prior to communication). Using the sockets API, applications can receive packets from more than one IP source address on a single UDP socket. Common support allows specification of the local IP address, destination IP address, local port and destination port values. Any or all of these can be indicated, with defaults supplied by the local system when these are not specified. The local endpoint is set using the BIND call and set on the remote endpoint using the CONNECT call. The CLOSE function has local significance only. It does not impact the status of the remote endpoint.

Neither UDP nor UDP-Lite provide congestion control, retransmission, nor do they provide mechanisms for application-level packetisation that would avoid IP fragmentation and other transport functions. This means that applications using UDP need to provide additional functions on top of the UDP transport API [RFC8085]. Some transport functions require parameters to be passed through the API to control the network layer (IPv4 or IPv6). These additional primitives could be considered a part of the network layer (e.g., control of the setting of the Don't Fragment (DF) flag on a transmitted IPv4 datagram), but are nonetheless essential to allow a user of the UDP API to implement functions that are normally associated with the transport layer (such as probing for the path maximum transmission size). This document includes such primitives.

Guidance on the use of the services provided by UDP is provided in the UDP Guidelines [RFC8085]. This also states "many operating systems also allow a UDP socket to be connected, i.e., to bind a UDP socket to a specific pair of addresses and ports. This is similar to the corresponding TCP sockets API functionality. However, for UDP, this is only a local operation that serves to simplify the local send/receive functions and to filter the traffic for the specified addresses and ports. Binding a UDP socket does not establish a connection - UDP does not notify the remote endpoint when a local UDP

socket is bound. Binding a socket also allows configuring options that affect the UDP or IP layers, for example, use of the UDP checksum or the IP Timestamp Option. On some stacks, a bound socket also allows an application to be notified when Internet Control Message (ICMP) error messages are received for its transmissions [RFC1122]."

The POSIX Base Specifications [POSIX] define an API that offers mechanisms for an application to receive asynchronous data events at the socket layer. Calls such as "poll", "select" or "queue" allow an application to be notified when data has arrived at a socket or when a socket has flushed its buffers.

A callback-driven API to the network interface can be structured on top of these calls. Implicit connection setup allows an application to delegate connection life management to the transport API. The transport API uses protocol primitives to offer the automated service to the application via the sockets API. By combining UDP primitives (CONNECT.UDP, SEND.UDP), a higher level API could offer a similar service.

The following datagram primitives are specified:

CONNECT: The CONNECT primitive allows the association of source and destination port sets to a socket to enable creation of a 'connection' for UDP traffic. This UDP connection allows an application to be notified of errors received from the network stack and provides a shorthand access to the send and receive primitives. Since UDP is itself connectionless, no datagrams are sent because this primitive is executed. A further connect call can be used to change the association.

The roles of a client and a server are often not appropriate for UDP, where connections can be peer-to-peer. The listening functions are performed using one of the forms of the CONNECT primitive:

1. bind(): A bind operation sets the local port, either implicitly, triggered by a "sendto" operation on an unbound unconnected socket using an ephemeral port. Or by an explicit "bind" to use a configured or well-known port.
2. bind(); connect(): A bind operation that is followed by a CONNECT primitive. The bind operation establishes the use of a known local port for datagrams, rather than using an ephemeral port. The connect operation specifies a known

address port combination to be used by default for future datagrams. This form is used either after receiving a datagram from an endpoint that causes the creation of a connection, or can be triggered by third party configuration or a protocol trigger (such as reception of a UDP Service Description Protocol, SDP [RFC4566], record).

SEND: The SEND primitive hands over a provided number of bytes that UDP should send to the other side of a UDP connection in a UDP datagram. The primitive can be used by an application to directly send datagrams to an endpoint defined by an address/port pair. If a connection has been created, then the address/port pair is inferred from the current connection for the socket. Connecting a socket allows network errors to be returned to the application as a notification on the send primitive. Messages passed to the send primitive that cannot be sent atomically in an IP packet will not be sent by the network layer, generating an error.

RECEIVE: The RECEIVE primitive allocates a receiving buffer to accommodate a received datagram. The primitive returns the number of bytes provided from a received UDP datagram. Section 4.1.3.5 of the requirements of Internet hosts [RFC1122] states "When a UDP datagram is received, its specific-destination address MUST be passed up to the application layer."

CHECKSUM_ENABLED: The optional CHECKSUM_ENABLED primitive controls whether a sender enables the UDP checksum when sending datagrams ([RFC0768] and [RFC6935] [RFC6936] [RFC8085]). When unset, this overrides the default UDP behaviour, disabling the checksum on sending. Section 4.1.3.4 of the requirements for Internet hosts [RFC1122] states "An application MAY optionally be able to control whether a UDP checksum will be generated, but it MUST default to checksumming on."

REQUIRE_CHECKSUM: The optional REQUIRE_CHECKSUM primitive determines whether UDP datagrams received with a zero checksum are permitted or discarded, UDP defaults to requiring checksums. Section 4.1.3.4 of the requirements for Internet hosts [RFC1122] states "An application MAY optionally be able to control whether UDP datagrams without checksums should be discarded or passed to the application." Section 3.1 of the specification for UDP-Lite [RFC3828] requires that the checksum field is non-zero, and hence the UDP-Lite API must discard all datagrams received with a zero checksum.

SET_IP_OPTIONS: The SET_IP_OPTIONS primitive requests the network-layer to send a datagram with the specified IP options. Section 4.1.3.2 of the requirements for Internet hosts [RFC1122]

states that an "application MUST be able to specify IP options to be sent in its UDP datagrams, and UDP MUST pass these options to the IP layer."

GET_IP_OPTIONS: The GET_IP_OPTIONS primitive retrieves the IP options of a datagram received at the network-layer. Section 4.1.3.2 of the requirements for Internet hosts[RFC1122] states that a UDP receiver "MUST pass any IP option that it receives from the IP layer transparently to the application layer".

SET_DF: The SET_DF primitive allows the network-layer to fragment packets using the Fragment Offset in IPv4 [RFC6864] and a host to use Fragment Headers in IPv6 [RFC8200]. The SET_DF primitive sets the Don't Fragment (DF) flag in the IPv4 packet header that carries a UDP datagram, which allows routers to fragment IPv4 packets. Although some specific applications rely on fragmentation support, in general, a UDP application should implement a method that avoids IP fragmentation (section 4 of [RFC8085]). NOTE: In many other IETF transports (e.g., TCP, SCTP) the transport provides the support needed to use DF. However, when using UDP, the application is responsible for the techniques needed to discover the effective Path Maximum Transmission Unit (PMTU) allowed on the network path, coordinating with the network layer. Classical PMTU Discovery (PMTUD) [RFC1191] relies upon the network path returning ICMP Fragmentation Needed or ICMPv6 Packet Too Big messages to the sender. When these ICMP messages are not delivered (or filtered) a sender is unable to learn the actual path MTU, and UDP Datagrams larger than the PMTU will be "black holed". To avoid this, an application can instead implement Packetization Layer Path MTU Discovery (PLPMTUD) [RFC4821] that does not rely upon network support for ICMPv6 messages and is therefore considered more robust than standard PMTUD, as recommended in [RFC8085] and [RFC8201].

GET_MMS_S: The GET_MMS_S primitive retrieves a network-layer value that indicates the maximum message size (MMS) that may be sent at the transport layer using a non-fragmented IP packet from the configured interface. This value is specified in section 6.1 of [RFC1191] and section 5.1 of [RFC8201]. It is calculated from Effective Maximum Transmit Unit for Sending (EMTU_S), and the link MTU for the given source IP address. This takes into account the size of the IP header plus space reserved by the IP layer for additional headers (if any). UDP applications should use this value as part of a method to avoid sending UDP datagrams that would result in IP packets that exceed the effective PMTU allowed across the network path. The effective PMTU (specified in Section 1 of [RFC1191]) is equivalent to the EMTU_S (specified in

[RFC1122]). The specification of PLPMTUD [RFC4821] states: "If PLPMTUD updates the MTU for a particular path, all Packetization Layer sessions that share the path representation (as described in Section 5.2) SHOULD be notified to make use of the new MTU and make the required congestion control adjustments".

GET_MMS_R: The GET_MMS_R primitive retrieves a network-layer value that indicates the maximum message size (MMS) that may be received at the transport layer from the configured interface. This value is specified in section 3.1 of [RFC1191]. It is calculated from Effective Maximum Transmit Unit for Receiving (EMTU_R), and the link MTU for the given source IP address, and takes into account the size of the IP header plus space reserved by the IP layer for additional headers (if any).

SET_TTL: The SET_TTL primitive sets the hop limit (TTL field) in the network-layer that is used in the IPv4 header of a packet that carries an UDP datagram. This is used to limit the scope of unicast datagrams. Section 3.2.2.4 of the requirements for Internet hosts [RFC1122] states an "incoming Time Exceeded message MUST be passed to the transport layer".

GET_TTL: The GET_TTL primitive retrieves the value of the TTL field in an IP packet received at the network layer. An application using the Generalized TTL Security Mechanism (GTSM) [RFC5082] can use this information to trust datagrams with a TTL value within the expected range, as described in Section 3 of RFC5082.

SET_MIN_TTL: The SET_MIN_TTL primitive restricts Datagrams delivered to the application to those received with an IP TTL value greater than or equal to passed parameter. This primitive can be used to implement applications such as Generalized TTL Security Mechanism (GTSM) [RFC5082] to as described in Section 3 of RFC5082, but this RFC does not specify this method.

SET_IPV6_UNICAST_HOPS: The SET_IPV6_UNICAST_HOPS primitive sets the network-layer hop limit field in an IPv6 packet header [RFC8200] carrying a UDP datagram. For IPv6 unicast datagrams, this is functionally equivalent to the SET_TTL IPv4 function.

GET_IPV6_UNICAST_HOPS: The GET_IPV6_UNICAST_HOPS primitive is a network-layer function that reads the hop count in the IPv6 header [RFC8200] information of a received UDP datagram. This is specified in section 6.3 of RFC3542. For IPv6 unicast datagrams, this is functionally equivalent to the GET_TTL IPv4 function.

SET_DSCP: The SET_DSCP primitive is a network-layer function that sets the DSCP, (or the legacy Type of Service, ToS) value

[RFC2474] to be used in the field of an IP header of a packet that carries a UDP datagram. Section 2.4 of the requirements for Internet hosts [RFC1123] states that "Applications MUST select appropriate ToS values when they invoke transport layer services, and these values MUST be configurable.". The application should be able to change the ToS during the connection lifetime, and the ToS value should be passed to the IP layer unchanged. Section 4.1.4 of [RFC1122] also states that on reception the "UDP MAY pass the received ToS value up to the application layer". The DiffServ model [RFC2475] [RFC3260] replaces this field in the IP Header assigning the six most significant bits to carry the DSCP field [RFC2474]. Preserving the intention of the host requirements [RFC1122] to allow the application to specify the "Type of Service", this should be interpreted to mean that an API should allow the application to set the DSCP. Section 3.1.6 of the UDP Guidelines [RFC8085] describes the way UDP applications should use this field. Normally a UDP socket will assign a single DSCP value to all datagrams in a flow, but a sender is allowed to use different DSCP values for datagrams within the same flow in certain cases [RFC8085]. There are guidelines for WebRTC that illustrate this use [RFC7657].

SET_ECN: The SET_ECN primitive is a network-layer function that sets the Explicit Congestion Notification (ECN) field in the IP Header of a UDP datagram. The ECN field defaults to a value of 00. When the use of the ToS field was redefined by DiffServ [RFC3260], 2 bits of the field were assigned to support ECN [RFC3168]. Section 3.1.5 of the UDP Guidelines [RFC8085] describes the way UDP applications should use this field. NOTE: In many other IETF transports (e.g., TCP) the transport provides the support needed to use ECN, when using UDP, the application or higher layer protocol is itself responsible for the techniques needed to use ECN.

GET_ECN: The GET_ECN primitive is a network-layer function that returns the value of the ECN field in the IP Header of a received UDP datagram. Section 3.1.5 of the UDP Guidelines [RFC8085] states that a UDP receiver "MUST check the ECN field at the receiver for each UDP datagram that it receives on this port", requiring the UDP receiver API to pass to pass the received ECN field up to the application layer to enable appropriate congestion feedback.

ERROR_REPORT The ERROR_REPORT event informs an application of "soft errors", including the arrival of an ICMP or ICMPv6 error message. Section 4.1.4 of the host requirements [RFC1122] states "UDP MUST pass to the application layer all ICMP error messages that it receives from the IP layer." For example, this event is required

to implement ICMP-based Path MTU Discovery [RFC1191] [RFC8201]. UDP applications must perform a CONNECT to receive ICMP errors.

CLOSE: The close primitive closes a connection. No further datagrams can be sent or received. Since UDP is itself connectionless, no datagrams are sent when this primitive is executed.

3.1.1. Excluded Primitives

Section 3.4 of the host requirements [RFC1122] also describes "GET_MAXSIZES, GET_SRCADDR (Section 3.3.4.3) and ADVISE_DELIVPROB:". These mechanisms are no longer used. It also specifies use of the Source Quench ICMP message, which has since been deprecated [RFC6633].

The IPV6_V6ONLY function is a network-layer primitive that applies to all transport services, defined in Section 5.3 of the basic socket interface for IPv6 [RFC3493]. This restricts the use of information from the name resolver to only allow communication of AF_INET6 sockets to use IPv6 only. This is not considered part of the transport service.

3.2. Primitives Provided by UDP-Lite

The Lightweight User Datagram Protocol (UDP-Lite) [RFC3828] provides similar services to UDP. It changed the semantics of the UDP "payload length" field to that of a "checksum coverage length" field. UDP-Lite requires the pseudo-header checksum to be computed at the sender and checked at a receiver. Apart from the length and coverage changes, UDP-Lite is semantically identical to UDP.

The sending interface of UDP-Lite differs from that of UDP by the addition of a single (socket) option that communicates the checksum coverage length. This specifies the intended checksum coverage, with the remaining unprotected part of the payload called the "error-insensitive part".

The receiving interface of UDP-Lite differs from that of UDP by the addition of a single (socket) option that specifies the minimum acceptable checksum coverage. The UDP-Lite Management Information Base (MIB) [RFC5097] further defines the checksum coverage method. Guidance on the use of services provided by UDP-Lite is provided in the UDP Guidelines [RFC8085].

UDP-Lite requires use of the UDP or UDP-Lite checksum, and hence it is not permitted to use the "DISABLE_CHECKSUM:" function to disable use of a checksum, nor is it possible to disable receiver checksum

processing using the "REQUIRE_CHECKSUM:" function . All other primitives and functions for UDP are permitted.

In addition, the following are defined:

SET_CHECKSUM_COVERAGE: The SET_CHECKSUM_COVERAGE primitive sets the coverage area for a sent datagram. UDP-Lite traffic uses this primitive to set the coverage length provided by the UDP checksum. Section 3.3 of the UDP-Lite MIB [RFC5097] states that "Applications that wish to define the payload as partially insensitive to bit errors ... Should do this by an explicit system call on the sender side." The default is to provide the same coverage as for UDP.

SET_MIN_COVERAGE The SET_MIN_COVERAGE primitive sets the minimum acceptable coverage protection for received datagrams. UDP-Lite traffic uses this primitive to set the coverage length that is checked on receive. (Section 1.1 of the UDP-Lite MIB [RFC5097] describes the corresponding MIB entry as `udpliteEndpointMinCoverage`.) Section 3.3 of the UDP-Lite specification [RFC3828] states that "applications that wish to receive payloads that were only partially covered by a checksum should inform the receiving system by an explicit system call". The default is to require only minimal coverage of the datagram payload.

4. Acknowledgements

This work was partially funded by the European Union's Horizon 2020 research and innovation programme under grant agreement No. 644334 (NEAT). Thanks to all who have commented or contributed, including Joe Touch, Ted Hardie, Aaron Falk, Tommy Pauly, and Francis Dupont.

5. IANA Considerations

This memo includes no request to IANA.

The authors request the section to be removed during conversion into an RFC by the RFC Editor.

6. Security Considerations

Security considerations for the use of UDP and UDP-Lite are provided in the referenced RFCs. Security guidance for application usage is provided in the UDP-Guidelines [RFC8085].

7. References

7.1. Normative References

- [I-D.ietf-taps-transports-usage]
Welzl, M., Tuexen, M., and N. Khademi, "On the Usage of Transport Features Provided by IETF Transport Protocols", draft-ietf-taps-transports-usage-08 (work in progress), August 2017.
- [RFC0768] Postel, J., "User Datagram Protocol", STD 6, RFC 768, DOI 10.17487/RFC0768, August 1980, <<https://www.rfc-editor.org/info/rfc768>>.
- [RFC1112] Deering, S., "Host extensions for IP multicasting", STD 5, RFC 1112, DOI 10.17487/RFC1112, August 1989, <<https://www.rfc-editor.org/info/rfc1112>>.
- [RFC1122] Braden, R., Ed., "Requirements for Internet Hosts - Communication Layers", STD 3, RFC 1122, DOI 10.17487/RFC1122, October 1989, <<https://www.rfc-editor.org/info/rfc1122>>.
- [RFC1123] Braden, R., Ed., "Requirements for Internet Hosts - Application and Support", STD 3, RFC 1123, DOI 10.17487/RFC1123, October 1989, <<https://www.rfc-editor.org/info/rfc1123>>.
- [RFC1191] Mogul, J. and S. Deering, "Path MTU discovery", RFC 1191, DOI 10.17487/RFC1191, November 1990, <<https://www.rfc-editor.org/info/rfc1191>>.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.
- [RFC3168] Ramakrishnan, K., Floyd, S., and D. Black, "The Addition of Explicit Congestion Notification (ECN) to IP", RFC 3168, DOI 10.17487/RFC3168, September 2001, <<https://www.rfc-editor.org/info/rfc3168>>.
- [RFC3493] Gilligan, R., Thomson, S., Bound, J., McCann, J., and W. Stevens, "Basic Socket Interface Extensions for IPv6", RFC 3493, DOI 10.17487/RFC3493, February 2003, <<https://www.rfc-editor.org/info/rfc3493>>.

- [RFC3828] Larzon, L-A., Degermark, M., Pink, S., Jonsson, L-E., Ed., and G. Fairhurst, Ed., "The Lightweight User Datagram Protocol (UDP-Lite)", RFC 3828, DOI 10.17487/RFC3828, July 2004, <<https://www.rfc-editor.org/info/rfc3828>>.
- [RFC6864] Touch, J., "Updated Specification of the IPv4 ID Field", RFC 6864, DOI 10.17487/RFC6864, February 2013, <<https://www.rfc-editor.org/info/rfc6864>>.
- [RFC6935] Eubanks, M., Chimento, P., and M. Westerlund, "IPv6 and UDP Checksums for Tunneled Packets", RFC 6935, DOI 10.17487/RFC6935, April 2013, <<https://www.rfc-editor.org/info/rfc6935>>.
- [RFC6936] Fairhurst, G. and M. Westerlund, "Applicability Statement for the Use of IPv6 UDP Datagrams with Zero Checksums", RFC 6936, DOI 10.17487/RFC6936, April 2013, <<https://www.rfc-editor.org/info/rfc6936>>.
- [RFC8085] Eggert, L., Fairhurst, G., and G. Shepherd, "UDP Usage Guidelines", BCP 145, RFC 8085, DOI 10.17487/RFC8085, March 2017, <<https://www.rfc-editor.org/info/rfc8085>>.
- [RFC8200] Deering, S. and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification", STD 86, RFC 8200, DOI 10.17487/RFC8200, July 2017, <<https://www.rfc-editor.org/info/rfc8200>>.
- [RFC8201] McCann, J., Deering, S., Mogul, J., and R. Hinden, Ed., "Path MTU Discovery for IP version 6", STD 87, RFC 8201, DOI 10.17487/RFC8201, July 2017, <<https://www.rfc-editor.org/info/rfc8201>>.

7.2. Informative References

- [POSIX] "IEEE Std. 1003.1-2001, , "Standard for Information Technology - Portable Operating System Interface (POSIX)", Open Group Technical Standard: Base Specifications Issue 6, ISO/IEC 9945:2002", December 2001.
- [RFC2474] Nichols, K., Blake, S., Baker, F., and D. Black, "Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers", RFC 2474, DOI 10.17487/RFC2474, December 1998, <<https://www.rfc-editor.org/info/rfc2474>>.

- [RFC2475] Blake, S., Black, D., Carlson, M., Davies, E., Wang, Z., and W. Weiss, "An Architecture for Differentiated Services", RFC 2475, DOI 10.17487/RFC2475, December 1998, <<https://www.rfc-editor.org/info/rfc2475>>.
- [RFC3260] Grossman, D., "New Terminology and Clarifications for Diffserv", RFC 3260, DOI 10.17487/RFC3260, April 2002, <<https://www.rfc-editor.org/info/rfc3260>>.
- [RFC3376] Cain, B., Deering, S., Kouvelas, I., Fenner, B., and A. Thyagarajan, "Internet Group Management Protocol, Version 3", RFC 3376, DOI 10.17487/RFC3376, October 2002, <<https://www.rfc-editor.org/info/rfc3376>>.
- [RFC3678] Thaler, D., Fenner, B., and B. Quinn, "Socket Interface Extensions for Multicast Source Filters", RFC 3678, DOI 10.17487/RFC3678, January 2004, <<https://www.rfc-editor.org/info/rfc3678>>.
- [RFC3810] Vida, R., Ed. and L. Costa, Ed., "Multicast Listener Discovery Version 2 (MLDv2) for IPv6", RFC 3810, DOI 10.17487/RFC3810, June 2004, <<https://www.rfc-editor.org/info/rfc3810>>.
- [RFC4566] Handley, M., Jacobson, V., and C. Perkins, "SDP: Session Description Protocol", RFC 4566, DOI 10.17487/RFC4566, July 2006, <<https://www.rfc-editor.org/info/rfc4566>>.
- [RFC4604] Holbrook, H., Cain, B., and B. Haberman, "Using Internet Group Management Protocol Version 3 (IGMPv3) and Multicast Listener Discovery Protocol Version 2 (MLDv2) for Source-Specific Multicast", RFC 4604, DOI 10.17487/RFC4604, August 2006, <<https://www.rfc-editor.org/info/rfc4604>>.
- [RFC4821] Mathis, M. and J. Heffner, "Packetization Layer Path MTU Discovery", RFC 4821, DOI 10.17487/RFC4821, March 2007, <<https://www.rfc-editor.org/info/rfc4821>>.
- [RFC5082] Gill, V., Heasley, J., Meyer, D., Savola, P., Ed., and C. Pignataro, "The Generalized TTL Security Mechanism (GTSM)", RFC 5082, DOI 10.17487/RFC5082, October 2007, <<https://www.rfc-editor.org/info/rfc5082>>.
- [RFC5097] Renker, G. and G. Fairhurst, "MIB for the UDP-Lite protocol", RFC 5097, DOI 10.17487/RFC5097, January 2008, <<https://www.rfc-editor.org/info/rfc5097>>.

- [RFC5790] Liu, H., Cao, W., and H. Asaeda, "Lightweight Internet Group Management Protocol Version 3 (IGMPv3) and Multicast Listener Discovery Version 2 (MLDv2) Protocols", RFC 5790, DOI 10.17487/RFC5790, February 2010, <<https://www.rfc-editor.org/info/rfc5790>>.
- [RFC6633] Gont, F., "Deprecation of ICMP Source Quench Messages", RFC 6633, DOI 10.17487/RFC6633, May 2012, <<https://www.rfc-editor.org/info/rfc6633>>.
- [RFC7657] Black, D., Ed. and P. Jones, "Differentiated Services (Diffserv) and Real-Time Communication", RFC 7657, DOI 10.17487/RFC7657, November 2015, <<https://www.rfc-editor.org/info/rfc7657>>.
- [STEVENS] "Stevens, W., Fenner, B., and A. Rudoff, "UNIX Network Programming, The sockets Networking API", Addison-Wesley.", 2004.

Appendix A. Multicast Primitives

This appendix describes primitives that are used when UDP and UDP-Lite support IPv4/IPv6 Multicast. Multicast services are not considered by the IETF TAPS WG, but the currently specified primitives are included for completeness in this appendix. Guidance on the use of UDP and UDP-Lite for multicast services is provided in the UDP Guidelines[RFC8085].

IP multicast may be supported using the Any Source Multicast (ASM) model or by the Source-Specific Multicast (SSM) model. The latter requires use of a Multicast Source Filter (MSF) when specifying an IP multicast group destination address.

Use of multicast requires additional primitives at the transport API that need to be called to coordinate operation of the IPv4 and IPv6 network layer protocols. For example, to receive datagrams sent to a group, an endpoint must first become a member of a multicast group at the network layer. Local multicast reception is signalled for IPv4 by the Internet Group Management Protocol (IGMP) [RFC3376] [RFC4604]. IPv6 uses the equivalent Multicast Listener Discovery (MLD) protocol [RFC3810] [RFC5790], carried over ICMPv6. A lightweight version of these protocols has also been specified [RFC5790].

The following are defined:

JoinGroup: Section 7.1 of the Host Extensions for IP Multicasting [RFC1112] provides a function that allows receiving traffic from an IP multicast group.

- JoinLocalGroup:** Section 7.2 of the Host Extensions for IP Multicasting [RFC1112] provides a function that allows receiving traffic from a local IP multicast group.
- LeaveHostGroup:** Section 7.1 of the Host Extensions for IP Multicasting [RFC1112] provides a function that allows leaving an IP multicast group.
- LeaveLocalGroup:** Section 7.2 of the Host Extensions for IP Multicasting [RFC1112] provides a function that allows leaving a local IP multicast group.
- IPV6_MULTICAST_IF:** Section 5.2 of the basic socket extensions for IPv6 [RFC3493] states that this sets the interface that will be used for outgoing multicast packets.
- IP_MULTICAST_TTL:** This sets the time to live field *t* to use for outgoing IPv4 multicast packets. This is used to limit scope of multicast datagrams. Methods such as The Generalized TTL Security Mechanism (GTSM) [RFC5082], set this value to ensure link-local transmission. GTSM also requires the UDP receiver API to pass the received value of this field to the application.
- IPV6_MULTICAST_HOPS:** Section 5.2 of the basic socket extensions for IPv6 [RFC3493] states that sets the hop count to use for outgoing multicast IPv6 packets. (This is equivalent to **IP_MULTICAST_TTL** used for IPv4 multicast).
- IPV6_MULTICAST_LOOP:** Section 5.2 of the basic socket extensions for IPv6 [RFC3493] states that this sets whether a copy of a datagram is looped back by the IP layer for local delivery when the datagram is sent to a group to which the sending host itself belongs).
- IPV6_JOIN_GROUP:** Section 5.2 of the basic socket extensions for IPv6 [RFC3493] provides a function that allows an endpoint to join an IPv6 multicast group.
- SIOCGIPMSFILTER:** Section 8.1 of the socket interface for MSF [RFC3678] provides a function that allows reading the multicast source filters.
- SIOCSIPMSFILTER:** Section 8.1 of the socket interface for MSF [RFC3678] provides a function that allows setting/modifying the multicast source filters.

IPV6_LEAVE_GROUP: Section 5.2 of the basic socket extensions for IPv6 [RFC3493] provides a function that allows leaving an IPv6 multicast group.

Section 4.1.1 of the Socket Interface Extensions for MSF [RFC3678] updates the multicast interface to add support for MSF for IPv4 and IPv6 required by IGMPv3. Three sets of API functionality are defined:

1. IPv4 Basic (Delta-based) API. "Each function call specifies a single source address which should be added to or removed from the existing filter for a given multicast group address on which to listen."
2. IPv4 Advanced (Full-state) API. "This API allows an application to define a complete source-filter comprised of zero or more source addresses, and replace the previous filter with a new one."
3. Protocol-Independent Basic MSF (Delta-based) API.
4. Protocol-Independent Advanced MSF (Full-state) API.

It specifies the following primitives:

IP_ADD_MEMBERSHIP: This is used to join an ASM group.

IP_BLOCK_SOURCE: This MSF can block data from a given multicast source to a given ASM or SSM group.

IP_UNBLOCK_SOURCE: This updates an MSF to undo a previous call to IP_UNBLOCK_SOURCE for an ASM or SSM group.

IP_DROP_MEMBERSHIP: This is used to leave an ASM or SSM group. (In SSM, this drops all sources that have been joined for a particular group and interface. The operations are the same as if the socket had been closed.)

Section 4.1.2 of the socket interface for MSF [RFC3678] updates the interface to add IPv4 MSF support to IGMPv3 using ASM:

IP_ADD_SOURCE_MEMBERSHIP: This is used to join an SSM group.

IP_DROP_SOURCE_MEMBERSHIP: This is used to leave an SSM group.

Section 4.1.2 of the socket interface for MSF [RFC3678] defines the Advanced (Full-state) API:

`setipv4sourcefilter` This is used to join an IPv4 multicast group, or to enable multicast from a specified source.

`getipv4sourcefilter`: This is used to leave an IPv4 multicast group, or to filter multicast from a specified source.

Section 5.1 of the socket interface for MSF [RFC3678] specifies Protocol-Independent Multicast API functions:

`MCAST_JOIN_GROUP` This is used to join an ASM group.

`MCAST_JOIN_SOURCE_GROUP` This is used to join an SSM group.

`MCAST_BLOCK_SOURCE`: This is used to block a source in an ASM group.

`MCAST_UNBLOCK_SOURCE`: This removes a previous MSF set by `MCAST_BLOCK_SOURCE`.

`MCAST_LEAVE_GROUP`: This leaves an ASM or SSM group.

`MCAST_LEAVE_SOURCE_GROUP`: This leaves a SSM group.

Section 5.2 of the socket interface for MSF [RFC3678] specifies the Protocol-Independent Advanced MSF (Full-state) API applicable for both IPv4 and IPv6:

`setsourcefilter` This is used to join an IPv4 or IPv6 multicast group, or to enable multicast from a specified source.

`getsourcefilter`: This is used to leave an IPv4 or IPv6 multicast group, or to filter multicast from a specified source.

The Lightweight IGMPv3 (LW_IGMPv3) and MLDv2 protocol [RFC5790] updates this interface (in Section 7.2 of RFC5790).

Appendix B. Revision Notes

Note to RFC-Editor: please remove this entire section prior to publication.

Individual draft -00:

- o This is the first version. Comments and corrections are welcome directly to the authors or via the IETF TAPS working group mailing list.

Individual draft -01:

- o Includes ability of a UDP receiver to disallow zero checksum datagrams.
- o Fixes to references and some connect on UDP usage.

Individual draft -02:

- o Fixes to address issues noted by WG.
- o Completed Multicast section to specify modern APIs.
- o Noted comments on API usage for UDP.
- o Feedback from various reviewers.

Individual draft -03:

- o Removes pass 2 and 3 of the TAPS analysis from this revision. These are expected to be incorporated into a combined draft of the TAPS WG.
- o Fixed Typos.

TAPS WG draft -00:

- o Expected to progress with draft-ietf-taps-transport-usage of the TAPS WG.

TAPS WG draft -01:

- o No intentional changes were made to the specification of primitives, this update is editorial
- o Reorganised text to eliminate the appendices.
- o Editorial changes were made to complete the document for a WGLSeC.
- o Rephrasing to eliminate using references as nouns, and to make text more consistent.
- o One appendix was incorporated.
- o This appendix was moved to the end (for later deletion by the RFC-Ed).

TAPS WG draft -02:

- o Updated to align with latest taps-transport-usage ID.

- o Revised to clarify MTU usage and track work in IPv6 PMTU
- o Usage of DF clarified.

o

TAPS WG draft -03

- o edit to MMS entries.

TAPS WG draft -04

- o Typos noted by Tommy Pauly 4/6/2017 and corrected here.
- o Checked and corrected parenthesis and use of period.
- o Document Shepherd review 7/2017.
- o Fixed citations and abbreviations.

TAPS WG draft -05

- o AD review 8/2017.
- o Updates to reflect published RFCs and refer to PMTUD for IPv6.
- o Aligned to latest TAPS transport usage ID.

TAPS WG draft -06

- o Fix to text for get TTL and IPv6 Hop Count

TAPS WG draft -07

- o Edit after secdir review - text on how a sender knows how to request UDP-Lite - added a para;
- o Abstract query about citing TAPS-transport;
- o Secdir editorial/format fixes have been applied.
- o Moved the note about "LISTEN:" to the text on "CONNECT:", Mirja suggested clarity that there is no LISTEN primitive for UDP.
- o Ben Campbell: Clarified the socket options were common examples used by multicast sockets.

- o Ben Campbell: Clarified that RFC 2119 is being cited, and not used to create new terms.
- o Ben Campbell: Added a direct copy of the text in RFC 768 describing the User Interface.
- o Francis Dupont: Many technical corrections.

Authors' Addresses

Godred Fairhurst
University of Aberdeen
School of Engineering
Fraser Noble Building
Fraser Noble Building Aberdeen AB24 3UE
UK

Email: gorry@erg.abdn.ac.uk

Tom Jones
University of Aberdeen
School of Engineering
Fraser Noble Building
Aberdeen AB24 3UE
UK

Email: tom@erg.abdn.ac.uk

TAPS Working Group
Internet-Draft
Intended status: Informational
Expires: April 30, 2017

B. Trammell
ETH Zurich
C. Perkins
University of Glasgow
T. Pauly
Apple Inc.
M. Kuehlewind
ETH Zurich
October 27, 2016

Post Sockets, An Abstract Programming Interface for the Transport Layer
draft-trammell-post-sockets-00

Abstract

This document describes Post Sockets, an asynchronous abstract programming interface for the atomic transmission of objects in an explicitly multipath environment. Post replaces connections with long-lived associations between endpoints, with the possibility to cache cryptographic state in order to reduce amortized connection latency. We present this abstract interface as an illustration of what is possible with present developments in transport protocols when freed from the strictures of the current sockets API.

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 April 30, 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. Abstractions and Terminology	5
2.1. Association	5
2.2. Listener	5
2.3. Remote	6
2.4. Local	6
2.5. Path	6
2.6. Object	7
2.7. Stream	9
3. Abstract Programming Interface	9
3.1. Active Association Creation	10
3.2. Listener and Passive Association Creation	11
3.3. Sending Objects	12
3.4. Receiving Objects	12
3.5. Creating and Destroying Streams	13
3.6. Events	13
3.7. Paths and Path Properties	14
3.8. Address Resolution	14
4. Acknowledgments	15
5. Informative References	15
Authors' Addresses	16

1. Introduction

The BSD Unix Sockets API's SOCK_STREAM abstraction, by bringing network sockets into the UNIX programming model, allowing anyone who knew how to write programs that dealt with sequential-access files to also write network applications, was a revolution in simplicity. It would not be an overstatement to say that this simple API is the reason the Internet won the protocol wars of the 1980s. SOCK_STREAM is tied to the Transmission Control Protocol (TCP), specified in 1981 [RFC0793]. TCP has scaled remarkably well over the past three and a half decades, but its total ubiquity has hidden an uncomfortable fact: the network is not really a file, and stream abstractions are too simplistic for many modern application programming models.

In the meantime, the nature of Internet access is evolving. Many end-user devices are connected to the Internet via multiple interfaces, which suggests it is time to promote the "path" by which a host is connected to a first-order object; we call this "path primacy".

Implicit multipath communication is available for these multihomed nodes in the present Internet architecture with the Multipath TCP extension (MPTCP) [RFC6824]. Since many multihomed nodes are connected to the Internet through access paths with widely different properties with respect to bandwidth, latency and cost, adding explicit path control to MPTCP's API would be useful in many situations. Path primacy for cooperation with path elements is also useful in single-homed architectures, such as the mechanism proposed by the Path Layer UDP Substrate (PLUS) effort (see [I-D.trammell-plus-statefulness] and [I-D.trammell-plus-abstract-mech]).

Another trend straining the traditional layering of the transport stack associated with the SOCK_STREAM interface is the widespread interest in ubiquitous deployment of encryption to guarantee confidentiality, authenticity, and integrity, in the face of pervasive surveillance [RFC7258]. Layering the most widely deployed encryption technology, Transport Layer Security (TLS), strictly atop TCP (i.e., via a TLS library such as OpenSSL that uses the sockets API) requires the encryption-layer handshake to happen after the transport-layer handshake, which increases connection setup latency on the order of one or two round-trip times, an unacceptable delay for many applications. Integrating cryptographic state setup and maintenance into the path abstraction naturally complements efforts in new protocols (e.g. QUIC [I-D.hamilton-quic-transport-protocol]) to mitigate this strict layering.

From these three starting points - more flexible abstraction, path primacy, and encryption by default - we define the Post-Socket Application Programming Interface (API), described in detail in this work. Post is designed to be language, transport protocol, and architecture independent, allowing applications to be written to a common abstract interface, easily ported among different platforms, and used even in environments where transport protocol selection may be done dynamically, as proposed in the IETF's Transport Services working group (see <https://datatracker.ietf.org/wg/taps/charter>).

Post replaces the traditional SOCK_STREAM abstraction with an Object abstraction, which can be seen as a generalization of the Stream Control Transmission Protocol's [RFC4960] SOCK_SEQPACKET service. Objects can be small (e.g. messages in message-oriented protocols) or large (e.g. an HTTP response containing header and body). It

replaces the notions of a socket address and connected socket with an Association with a remote endpoint via set of Paths. Implementation and wire format for transport protocol(s) implementing the Post API are explicitly out of scope for this work; these abstractions need not map directly to implementation-level concepts, and indeed with various amounts of shimming and glue could be implemented with varying success atop any sufficiently flexible transport protocol.

For compatibility with situations where only strictly stream-oriented transport protocols are available, applications with data streams that cannot be easily split into Objects at the sender, and and for easy porting of the great deal of existing stream-oriented application code to Post, Post also provides a SOCK_STREAM compatible abstraction, unimaginatively named Stream.

The key features of Post as compared with the existing sockets API are:

- o Explicit Object orientation, with framing and atomicity guarantees for Object transmission.
- o Asynchronous reception, allowing all receiver-side interactions to be event-driven.
- o Explicit support for multipath transport protocols and network architectures.
- o Long-lived Associations, whose lifetimes may not be bound to underlying transport connections. This allows associations to cache state and cryptographic key material to enable fast (0-rtt) resumption of communication.

This work is the synthesis of many years of Internet transport protocol research and development. It is heavily inspired by concepts from the Stream Control Transmission Protocol (SCTP) [RFC4960], TCP Minion [I-D.iyengar-minion-protocol], MinimaLT[MinimaLT], and various bulk object transports.

We present Post Sockets as an illustration of what is possible with present developments in transport protocols when freed from the strictures of the current sockets API. While much of the work for building parts of the protocols needed to implement Post are already ongoing in other IETF working groups (e.g. TAPS, MPTCP, QUIC, TLS), we argue that an abstract programming interface unifying access all these efforts is necessary to fully exploit their potential.

2. Abstractions and Terminology

gratuitously colorful SVG goes [here](#); see slide six of

<https://www.ietf.org/proceedings/96/slides/slides-96-taps-2.pdf>

in the meantime

Figure 1: Abstractions and relationships in Post Sockets

Post is based on a small set of abstractions, the relationships among which are shown in Figure Figure 1 and detailed in this section.

2.1. Association

An Association is a container for all the state necessary for a local endpoint to communicate with a remote endpoint in an explicitly multipath environment. It contains a set of Paths, certificate(s) for identifying the remote endpoint, certificate(s) and key(s) for identifying the local endpoint to the remote endpoint, and any cached cryptographic state for the communication to the remote endpoint. An Association may have one or more Streams active at any given time. Objects are sent to Associations, as well.

Note that, in contrast to current SOCK_STREAM sockets, Associations are meant to be relatively long-lived. The lifetime of an Association is not bound to the lifetime of any transport-layer connection between the two endpoints; connections may be opened or closed as necessary to support the Streams and Object transmissions required by the application, and the application need not be bothered with the underlying connectivity state unless this is important to the application's semantics.

Paths may be dynamically added or removed from an association, as well, as connectivity between the endpoints changes. Cryptographic identifiers and state for endpoints may also be added and removed as necessary due to certificate lifetimes, key rollover, and revocation.

2.2. Listener

In many applications, there is a distinction between the active opener (or connection initiator, often a client), and the passive opener (often a server). A Listener represents an endpoint's willingness to start Associations in this passive opener/server role. It is, in essence, a one-sided, Path-less Association from which fully-formed Associations can be created.

Listeners work very much like sockets on which the `listen(2)` call has been called in the `SOCK_STREAM` API.

2.3. Remote

A Remote represents all the information required to establish and maintain a connection with the far end of an Association: network-layer address, transport-layer port, information about public keys or certificate authorities used to identify the remote on connection establishment, etc. Each Association is associated with a single Remote, either explicitly by the application (when created by active open) or by the Listener (when created by passive open). The resolution of Remotes from higher-layer information (URIs, hostnames) is architecture-dependent.

2.4. Local

A Local represents all the information about the local endpoint necessary to establish an Association or a Listener: interface and port designators, as well as certificates and associated private keys.

2.5. Path

A Path represents a local and remote endpoint address, an optional set of intermediary path elements between the local and remote endpoint addresses, and a set of properties associated with the path.

The set of available properties is a function of the underlying network-layer protocols used to expose the properties to the endpoint. However, the following core properties are generally useful for applications and transport layer protocols to choose among paths for specific Objects:

- o Maximum Transmission Unit (MTU): the maximum size of an Object's payload (subtracting transport, network, and link layer overhead) which will likely fit into a single frame. Derived from signals sent by path elements, where available, and/or path MTU discovery processes run by the transport layer.
- o Latency Expectation: expected one-way delay along the Path. Generally provided by inline measurements performed by the transport layer, as opposed to signaled by path elements.
- o Loss Probability Expectation: expected probability of a loss of any given single frame along the Path. Generally provided by inline measurements performed by the transport layer, as opposed to signaled by path elements.

- o Available Data Rate Expectation: expected maximum data rate along the Path. May be derived from passive measurements by the transport layer, or from signals from path elements.
- o Reserved Data Rate: Committed, reserved data rate for the given Association along the Path. Requires a bandwidth reservation service in the underlying transport and network layer protocol.
- o Path Element Membership: Identifiers for some or all nodes along the path, depending on the capabilities of the underlying network layer protocol to provide this.

Path properties are generally read-only. MTU is a property of the underlying link-layer technology on each link in the path; latency, loss, and rate expectations are dynamic properties of the network configuration and network traffic conditions; path element membership is a function of network topology. In an explicitly multipath architecture, application and transport layer requirements are met by having multiple paths with different properties to select from. Post can also provide signaling to the path, but this signaling is derived from information provided to the Object abstraction, below.

Note that information about the path and signaling to path elements could be provided by a facility such as PLUS
[I-D.trammell-plus-abstract-mech].

2.6. Object

Post provides two ways to send data over an Association. We start with the Object abstraction, as a fundamental insight behind the interface is that most applications fundamentally deal in object transport.

An Object is an atomic unit of communication between applications; or in other words, an ordered collection of bytes $B_0..B_m$, such that every byte B_n depends on every other byte in the Object. An object that cannot be delivered in its entirety within the constraints of the network connectivity and the requirements of the application is not delivered at all.

Objects can represent both relatively small structures, such as messages in application-layer protocols built around datagram or message exchange, as well as relatively large structures, such files of arbitrary size in a filesystem. Objects larger than the MTU on the Path on which they are sent will be segmented into multiple frames. Multiple objects that will fit into a single frame may be concatenated into one frame. There is no preference for transmitting the multiple frames for a given Object in any particular order, or by

default, that objects will be delivered in the order sent by the application. This implies that both the sending and receiving endpoint, whether in the application layer or the transport layer, must guarantee storage for the full size of an object.

Three object properties allow applications fine control ordering and reliability requirements in line with application semantics. An Object may have a "lifetime" - a wallclock duration before which the object must be available to the application layer at the remote end. If a lifetime cannot be met, the object is discarded as soon as possible; therefore, Objects with lifetimes are implicitly sent non-reliably, and lifetimes are used to prioritize Object delivery. Lifetimes may be signaled to path elements by the underlying transport, so that path elements that realize a lifetime cannot be met can discard frames containing the object instead of forwarding them.

Second, Objects may have a "niceness" - a category in an unbounded hierarchy most naturally represented as a non-negative integer. By default, Objects are in niceness class 0, or highest priority. Niceness class 1 Objects will yield to niceness class 0 objects, class 2 to class 1, and so on. Niceness may be translated to a priority signal for exposure to path elements (e.g. DSCP codepoint) to allow prioritization along the path as well as at the sender and receiver. This inversion of normal schemes for expressing priority has a convenient property: priority increases as both niceness and deadline decrease.

An object may have both a niceness and a lifetime - objects with higher niceness classes will yield to lower classes if resource constraints mean only one can meet the lifetime.

Third, an Object may have "antecedents" - other Objects on which it depends, which must be delivered before it (the "successor") is delivered. The sending transport uses deadlines, niceness, and antecedents, along with information about the properties of the Paths available, to determine when to send which object down which Path.

When an application has hard semantic requirements that all the frames of a given object be sent down a given Path or Paths, these hard constraints can also be expressed by the application.

After calling the send function, the application can register event handlers to be informed of the transmission status of the object; the object can either be acknowledged (i.e., it has been received in full by the remote endpoint) or expired (its lifetime ran out before it was acknowledged).

2.7. Stream

The Stream abstraction is provided for two reasons. First, since it is the most like the existing SOCK_STREAM interface, it is the simplest abstraction to be used by applications ported to Post to take advantages of Path primacy. Second, some environments have connectivity so impaired (by local network operation policy and/or accidental middlebox interference) that only stream-based transport protocols are available, and applications should have the option to use streams directly in these situations.

A Stream is a sequence of bytes $B_0 \dots B_m$ such that the reception (and delivery to the receiving application of) B_n always depends on B_{n-1} . This property is inherited from the BSD UNIX file abstraction, which in turn inherited it from the physical limitations of sequential access media (stacks of punch cards, paper and/or magnetic tape).

A Stream is bound to an Association. Writing a byte to the stream will cause it to be received by the remote, in order, or will cause an error condition and termination of the stream if the byte cannot be delivered. Due to the strong sequential dependence on a stream, streams must always be reliable and ordered. If frames containing Stream data are lost, these must be retransmitted or reconstructed using an error correction technique. If frames containing Stream data arrive out of order, the remote end must buffer them until the unordered frames are received and reassembled.

As with Objects, Streams may have a niceness for prioritization. When mixing Stream and Object data on the same Path in an association, the niceness classes for Streams and Objects are interleaved; e.g. niceness 2 Stream frames will yield to niceness 1 Object frames.

The underlying transport protocol may make whatever use of the Paths and known properties of those Paths it sees fit when transporting a Stream.

3. Abstract Programming Interface

We now turn to the design of an abstract programming interface to provide a simple interface to Post's abstractions, constrained by the following design principles:

- o Flexibility is paramount. So is simplicity. Applications must be given as many controls and as much information as they may need, but they must be able to ignore controls and information irrelevant to their operation. This implies that the "default"

interface must be no more complicated than BSD sockets, and must do something reasonable.

- o A new API cannot be bound to a single transport protocol and expect wide deployment. As the API is transport-independent and may support runtime transport selection, it must impose the minimum possible set of constraints on its underlying transports, though some API features may require underlying transport features to work optimally. It must be possible to implement Post over vanilla TCP in the present Internet architecture.
- o Reception is an inherently asynchronous activity. While the API is designed to be as platform-independent as possible, one key insight it is based on is that an object receiver's behavior in a packet-switched network is inherently asynchronous, driven by the receipt of packets, and that this asynchronicity must be reflected in the API. The actual implementation of receive and event callbacks will need to be aligned to the method a given platform provides for asynchronous I/O.

The API we define consists of three classes (listener, association, and stream), four entry points (listen(), associate(), send(), and open_stream()) and a set of callbacks for handling events at each endpoint. The details are given in the subsections below.

3.1. Active Association Creation

Associations can be created two ways: actively by a connection initiator, and passively by a Listener that accepts a connection. Connection initiation uses the associate() entry point:

```
association = associate(local, remote, receive_handler)
```

where:

- o local: a resolved Local (see Section 3.8) describing the local identity and interface(s) to use
- o remote: a resolved Remote (see Section 3.8) to associate with
- o receive_handler: a callback to be invoked when new objects are received; see Section 3.4

The returned association has the following additional properties:

- o paths: a set of Paths that the Association can currently use to transport Objects. When the underlying transport connection is closed, this set will be empty. For explicitly multipath

architectures and transports, this set may change dynamically during the lifetime of an association, even while it remains connected.

Since the existence of an association does not necessarily imply current connection state at both ends of the Association, these objects are durable, and can be cached, migrated, and restored, as long as the mappings to their event handlers are stable. An attempt to send an object or open a stream on a dormant, previously actively-opened association will cause the underlying transport connection state to be resumed.

3.2. Listener and Passive Association Creation

In order to accept new Association requests from clients, a server must create a Listener object, using the `listen()` entry point:

```
listener = listen(local, accept_handler)
```

where:

- o `local`: resolved Local (see Section 3.8) describing the local identity and interface(s) to use for Associations created by this listener.
- o `accept_handler`: callback to be invoked each time an association is requested by a remote, to finalize setting the association up. Platforms may provide a default here for supporting synchronous association request handling via an object queue.

The `accept_handler` has the following prototype:

```
accepted = accept_handler(listener, local, remote)
```

where:

- o `local`: a resolved Local on which the association request was received.
- o `remote`: a resolved Remote from which the association request was received.
- o `accepted`: flag, true if the handler decided to accept the request, false otherwise.

The `accept_handler()` calls the `accept()` entry point to finally create the association:


```
association = accept(listener, local, remote, receive_handler)
```

3.3. Sending Objects

Objects are sent using the `send()` entry point:

```
send(association, bytes, [lifetime], [niceness], [oid],  
[antecedent_oids], [paths])
```

where:

- o `association`: the association to send the object on
- o `bytes`: sequence of bytes making up the object. For platforms without bounded byte arrays, this may be implemented as a pointer and a length.
- o `lifetime`: lifetime of the object in milliseconds. This parameter is optional and defaults to infinity (for fully reliable object transport).
- o `niceness`: the object's niceness class. This parameter is optional and defaults to zero (for lowest niceness / highest priority)
- o `oid`: opaque identifier for an object, assigned by the application. Used to refer to this object as a subsequently sent object's antecedent, or in an ack or expired handler (see Section 3.6). Optional, defaults to null.
- o `antecedent_oids`: set of object identifiers on which this object depends and which must be sent before this object. Optional, defaults to empty, meaning this object has no antecedent constraints.
- o `paths`: set of paths, as a subset of those available to the association, to explicitly use for this object. Optional, defaults to empty, meaning all paths are acceptable.

Calls to `send` are non-blocking; a synchronous send which blocks on remote acknowledgment or expiry of an object can be implemented by a call to `send()` followed by a wait on the ack or expired events (see Section 3.6).

3.4. Receiving Objects

An application receives objects via its `receive_handler` callback, registered at association creation time. This callback has the following prototype:

```
receive_handler(association, bytes)
```

where: - association: the association the object was received from.
- bytes: the sequence of bytes making up the object.

For ease of porting synchronous datagram applications, implementations may make a default receive handler available, which allows messages to be synchronously polled from a per-association object queue. If this default is available, the entry point for the polling call is:

```
bytes = receive_next(association)
```

3.5. Creating and Destroying Streams

A stream may be created on an association via the `open_stream()` entry point:

```
stream = open_stream(association, [sid])
```

where:

- o association: the association to open the stream on
- o sid: opaque identifier for a stream. For transport protocols which do not support multiple streaming, this argument has no effect.

A stream with a given sid must be opened by both sides before it can be used.

The stream object returned should act like a file descriptor or bidirectional I/O object, according to the conventions of the platform implementing Post.

3.6. Events

Message reception is a specific case of an event that can occur on an association. Other events are also available, and the application can register event handlers for each of these. Event handlers are registered via the `handle()` entry point:

```
handle(association, event, handler) or
```

```
handle(oid, event, handler)
```

where

- o association: the association to register a handler on, or
- o oid: the object identifier to register a handler on
- o event: an identifier of the event to register a handler on
- o handler: a callback to be invoked when the event occurs, or null if the event should be ignored.

The following events are supported; every event handler takes the association on which it is registered as well as any additional arguments listed:

- o receive (bytes): an object has been received
- o path_up (path): a path is newly available
- o path_down (path): a path is no longer available
- o dormant: no more paths are available, the association is now dormant, and the connection will need to be resumed if further objects are to be sent
- o ack (oid): an object was successfully received by the remote
- o expired (oid): an object expired before being sent to the remote

Handlers for the ack and expired events can be registered on an association (in which case they are called for all objects sent on the association) or on an oid (in which case they are only called for the oid).

3.7. Paths and Path Properties

As defined in Section 2.5, the properties of a path include both the addresses of elements along the path as well as measurement-derived latency and capacity characteristics. The path_up and path_down events provide access to information about the paths available via the path argument to the event handler. This argument encapsulates these properties in a platform and transport-specific way, depending on the availability of information about the path.

3.8. Address Resolution

Address resolution turns the name of a Remote into a resolved Remote object, which encapsulates all the information needed to connect (address, certificate parameters, cached cryptographic state, etc.); and an interface identifier on a local system to information needed

to connect. Remote and local resolvers have the following entry points:

```
remote = resolve(endpoint_name, configuration)
```

```
local = resolve_local(endpoint_name, configuration)
```

where:

- o `endpoint_name`: a name identifying the remote or local endpoint, including port
- o `configuration`: a platform-specific configuration object for configuring certificates, name resolution contexts, cached cryptographic state, etc.

4. Acknowledgments

Many thanks to Laurent Chuat and Jason Lee at the Network Security Group at ETH Zurich for contributions to the initial design of Post Sockets.

This work is partially supported by the European Commission under Horizon 2020 grant agreement no. 688421 Measurement and Architecture for a Middleboxed Internet (MAMI), and by the Swiss State Secretariat for Education, Research, and Innovation under contract no. 15.0268. This support does not imply endorsement.

5. Informative References

[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.iyengar-minion-protocol]

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

[I-D.trammell-plus-abstract-mech]

Trammell, B., "Abstract Mechanisms for a Cooperative Path Layer under Endpoint Control", draft-trammell-plus-abstract-mech-00 (work in progress), September 2016.

[I-D.trammell-plus-statefulness]

Kuehlewind, M., Trammell, B., and J. Hildebrand,
"Transport-Independent Path Layer State Management",
draft-trammell-plus-statefulness-00 (work in progress),
October 2016.

[MinimalT]

Petullo, W., Zhang, X., Solworth, J., Bernstein, D., and
T. Lange, "MinimalT, Minimal-latency Networking Through
Better Security", May 2013.

[RFC0793] Postel, J., "Transmission Control Protocol", STD 7,
RFC 793, DOI 10.17487/RFC0793, September 1981,
<<http://www.rfc-editor.org/info/rfc793>>.

[RFC4960] Stewart, R., Ed., "Stream Control Transmission Protocol",
RFC 4960, DOI 10.17487/RFC4960, September 2007,
<<http://www.rfc-editor.org/info/rfc4960>>.

[RFC6824] Ford, A., Raiciu, C., Handley, M., and O. Bonaventure,
"TCP Extensions for Multipath Operation with Multiple
Addresses", RFC 6824, DOI 10.17487/RFC6824, January 2013,
<<http://www.rfc-editor.org/info/rfc6824>>.

[RFC7258] Farrell, S. and H. Tschofenig, "Pervasive Monitoring Is an
Attack", BCP 188, RFC 7258, DOI 10.17487/RFC7258, May
2014, <<http://www.rfc-editor.org/info/rfc7258>>.

Authors' Addresses

Brian Trammell
ETH Zurich
Gloriastrasse 35
8092 Zurich
Switzerland

Email: ietf@trammell.ch

Colin Perkins
University of Glasgow
School of Computing Science
Glasgow G12 8QQ
United Kingdom

Email: csp@cperkins.net

Tommy Pauly
Apple Inc.
1 Infinite Loop
Cupertino, California 95014
United States of America

Email: tpauly@apple.com

Mirja Kuehlewind
ETH Zurich
Gloriastrasse 35
8092 Zurich
Switzerland

Email: mirja.kuehlewind@tik.ee.ethz.ch