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-transports-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:
- Disable checksum when sending
- Disable checksum requirement when receiving
- Specify checksum coverage used by the sender
- 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:
- Configurable Message Reliability
- Choice between unordered (potentially faster) or ordered delivery of messages
- Request not to bundle messages
- 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:
- 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
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-rtcweb-qos]).
- low_watermark: a buffer limit (in bytes); when the sender has less then 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:
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


9.2. Informative References


[I-D.ietf-tsvwg-rtcweb-qos]
Jones, P., Dhesikan, S., Jennings, C., and D. Druta, "DSCP Packet Markings for WebRTC QoS",

[I-D.ietf-tsvwg-sctp-ndata]
Stewart, R., Tuexen, M., Loreto, S., and R. Seggelmann,
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,

[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,

[RFC4987]  Eddy, W., "TCP SYN Flooding Attacks and Common Mitigations",
RFC 4987, DOI 10.17487/RFC4987, August 2007,

RFC 5925, DOI 10.17487/RFC5925, June 2010,

[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,

RFC 6525, DOI 10.17487/RFC6525, February 2012,
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:
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.

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.

- 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.

- 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.

- 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.

- 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].

- 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.
- 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.

- 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:
- 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.

- Suggest timeout to the peer
  Protocols: TCP
  Functional because this is closely related to potentially assumed reliable data delivery.
  Implementation: via CHANGE-TIMEOUT.TCP.

- 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.

- Request an immediate heartbeat, returning success/failure
  Protocols: SCTP
  Automatable because this informs about network-specific knowledge.

- 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.
network, not the application.

- 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)

- 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).

- 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.

- 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].

- 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]).

- Set maximum burst
  Protocols: SCTP
  Automatable because it relates to knowledge about the network, not the application.

- 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.

- 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.

- 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.

- 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.

- 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.

- 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.

- 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.

- 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.

- 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.

- Obtain TTL/Hop count field
  Protocols: UDP(-Lite)
  Automatable because the TTL/Hop count field relates to knowledge
  about the network, not the application.

- Specify ECN field
  Protocols: UDP(-Lite)
  Automatable because the ECN field relates to knowledge about the
  network, not the application.

- 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).

- Specify IP Options
  Protocols: UDP(-Lite)
  Automatable because IP Options relate to knowledge about the network, not the application.

- Obtain IP Options
  Protocols: UDP(-Lite)
  Automatable because IP Options relate to knowledge about the network, not the application.

- 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:**

- 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.

- 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.
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.

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

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.

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.

Unreliably transfer a message

Protocols: SCTP, UDP(-Lite)

Optimizing because only applications know about the time criticality of their communication, and reliably transfering 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.
- Unreliably transfer a message, with congestion control
  Protocols: SCTP
  Automatable because congestion control relates to knowledge about
  the network, not the application.

- Unreliably transfer a message, without congestion control
  Protocols: UDP(-Lite)
  Automatable because congestion control relates to knowledge about
  the network, not the application.

- 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.

- 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.

- 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.

- 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.

- 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.

- 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.

- 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).

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

AVAILABILITY:
- Listen
- Configure authentication

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

- 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.

- 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.

- 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:

- 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.

- 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).

- 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:
- Disable Nagle algorithm
- Enable and configure a "Low Extra Delay Background Transfer"
- 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 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
Services provided by IETF transport protocols and congestion control mechanisms
draft-ietf-taps-transports-06

Abstract

This document describes services provided by existing IETF protocols and congestion control mechanisms. It is designed to help application and network stack programmers and to inform the work of the IETF TAPS Working Group.

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 14, 2015.

Copyright Notice

Copyright (c) 2015 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. Existing Transport Protocols ............................ 4
   3.1. Transport Control Protocol (TCP) .................... 4
       3.1.1. Protocol Description ........................... 5
       3.1.2. Interface description ........................... 6
       3.1.3. Transport Protocol Components ................. 6
   3.2. Multipath TCP (MPTCP) .............................. 7
       3.2.1. Protocol Description ........................... 8
       3.2.2. Interface Description ........................... 8
       3.2.3. Transport Protocol Components ................. 8
   3.3. Stream Control Transmission Protocol (SCTP) ......... 9
       3.3.1. Protocol Description ........................... 9
       3.3.2. Interface Description ........................... 11
       3.3.3. Transport Protocol Components ................. 13
   3.4. User Datagram Protocol (UDP) ....................... 13
       3.4.1. Protocol Description ........................... 14
       3.4.2. Interface Description ........................... 14
       3.4.3. Transport Protocol Components ................. 16
   3.5. Lightweight User Datagram Protocol (UDP-Lite) ....... 15
       3.5.1. Protocol Description ........................... 15
       3.5.2. Interface Description ........................... 16
       3.5.3. Transport Protocol Components ................. 16
   3.6. Datagram Congestion Control Protocol (DCCP) ....... 17
       3.6.1. Protocol Description ........................... 17
       3.6.2. Interface Description ........................... 19
       3.6.3. Transport Protocol Components ................. 19
   3.7. Realtime Transport Protocol (RTP) .................. 19
   3.8. NACK-Oriented Reliable Multicast (NORM) ............ 20
       3.8.1. Protocol Description ........................... 20
       3.8.2. Interface Description ........................... 21
       3.8.3. Transport Protocol Components ................. 21
   3.9. Transport Layer Security (TLS) and Datagram TLS (DTLS) as a pseudotransport ....................... 22
       3.9.1. Protocol Description ........................... 23
       3.9.2. Interface Description ........................... 24
       3.9.3. Transport Protocol Components ................. 24
   3.10. Hypertext Transport Protocol (HTTP) over TCP as a pseudotransport .................................... 25
       3.10.1. Protocol Description ........................... 25
       3.10.2. Interface Description ........................... 26
       3.10.3. Transport Protocol Components ................. 27
   3.11. WebSockets ....................................... 27
1. Introduction

Most Internet applications make use of the Transport Services provided by TCP (a reliable, in-order stream protocol) or UDP (an unreliable datagram protocol). We use the term "Transport Service" to mean the end-to-end service provided to an application by the transport layer. That service can only be provided correctly if information about the intended usage is supplied from the application. The application may determine this information at design time, compile time, or run time, and may include guidance on whether a feature is required, a preference by the application, or something in between. Examples of features of Transport Services are reliable delivery, ordered delivery, content privacy to in-path devices, integrity protection, and minimal latency.

The IETF has defined a wide variety of transport protocols beyond TCP and UDP, including SCTP, DCCP, MP-TCP, and UDP-Lite. Transport services may be provided directly by these transport protocols, or layered on top of them using protocols such as WebSockets (which runs over TCP), RTP (over TCP or UDP) or WebRTC data channels (which run over SCTP over DTLS over UDP or TCP). Services built on top of UDP or UDP-Lite typically also need to specify additional mechanisms, including a congestion control mechanism (such as a windowed congestion control, TFRC or LEDBAT congestion control mechanism). This extends the set of available Transport Services beyond those provided to applications by TCP and UDP.

Transport protocols can also be differentiated by the features of the services they provide: for instance, SCTP offers a message-based service providing full or partial reliability and allowing to minimize the head of line blocking due to the support of unordered and unordered message delivery within multiple streams, UDP-Lite provides partial integrity protection, and LEDBAT can provide low-priority "scavenger" communication.
2. Terminology

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

[EDITOR’S NOTE: we may want to add definitions for the different kinds of interfaces that are important here.]

Transport Service Feature: a specific end-to-end feature that a transport service provides to its clients. Examples include confidentiality, reliable delivery, ordered delivery, message-versus-stream orientation, etc.

Transport Service: a set of transport service 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 Protocol Component: an implementation of a transport service 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 data across the network (this may also be an upper layer protocol or tunnel encapsulation).

3. Existing Transport Protocols

This section provides a list of known IETF transport protocol and transport protocol frameworks.

[EDITOR’S NOTE: Contributions to the subsections below are welcome]

3.1. Transport Control Protocol (TCP)

TCP is an IETF standards track transport protocol. [RFC0793] introduces TCP as follows: "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." Since its introduction,
TCP has become the default connection-oriented, stream-based transport protocol in the Internet. It is widely implemented by endpoints and widely used by common applications.

3.1.1. Protocol Description

TCP is a connection-oriented protocol, providing a three way handshake to allow a client and server to set up a connection, and mechanisms for orderly completion and immediate teardown of a connection. TCP is defined by a family of RFCs [RFC4614].

TCP provides multiplexing to multiple sockets on each host using port numbers. An active TCP session is identified by its four-tuple of local and remote IP addresses and local port and remote port numbers. The destination port during connection setup has a different role as it is often used to indicate the requested service.

TCP partitions a continuous stream of bytes into segments, sized to fit in IP packets. ICMP-based PathMTU discovery [RFC1191][RFC1981] as well as Packetization Layer Path MTU Discovery (PMTUD) [RFC4821] are supported.

Each byte in the stream is identified by a sequence number. The sequence number is used to order segments on receipt, to identify segments in acknowledgments, and to detect unacknowledged segments for retransmission. This is the basis of TCP’s reliable, ordered delivery of data in a stream. TCP Selective Acknowledgment [RFC2018] extends this mechanism by making it possible to identify missing segments more precisely, reducing spurious retransmission.

Receiver flow control is provided by a sliding window: limiting the amount of unacknowledged data that can be outstanding at a given time. The window scale option [RFC7323] allows a receiver to use windows greater than 64KB.

All TCP senders provide Congestion Control: This uses a separate window, where each time congestion is detected, this congestion window is reduced. A receiver detects congestion using one of three mechanisms: A retransmission timer, detection of loss (interpreted as a congestion signal), or Explicit Congestion Notification (ECN) [RFC3168] to provide early signaling (see [I-D.ietf-aqm-ecn-benefits])

A TCP protocol instance can be extended [RFC4614] and tuned. Some features are sender-side only, requiring no negotiation with the receiver; some are receiver-side only, some are explicitly negotiated during connection setup.
By default, TCP segment partitioning uses Nagle’s algorithm [RFC0896] to buffer data at the sender into large segments, potentially incurring sender-side buffering delay; this algorithm can be disabled by the sender to transmit more immediately, e.g. to enable smoother interactive sessions.

[EDITOR’S NOTE: add URGENT and PUSH flag (note [RFC6093] says SHOULD NOT use due to the range of TCP implementations that process TCP urgent indications differently.) ]

A checksum provides an Integrity Check and is mandatory across the entire packet. The TCP checksum does not support partial corruption protection as in DCCP/UDP-Lite). This check protects from misdelivery of data corrupted data, but is relatively weak, and applications that require end to end integrity of data are recommended to include a stronger integrity check of their payload data.

A TCP service is unicast.

3.1.2. Interface description

A User/TCP Interface is defined in [RFC0793] providing six user commands: Open, Send, Receive, Close, Status. This interface does not describe configuration of TCP options or parameters beside use of the PUSH and URGENT flags.

In API implementations derived from the BSD Sockets API, TCP sockets are created using the "SOCK_STREAM" socket type.

The features used by a protocol instance may be set and tuned via this API.

(see more on the API goes here)

3.1.3. Transport Protocol Components

The transport protocol components provided by TCP (new version) are:

[EDITOR’S NOTE: discussion of how to map this to features and TAPS: what does the higher layer need to decide? what can the transport layer decide based on global settings? what must the transport layer decide based on network characteristics?]

- Connection-oriented bidirectional communication using three-way handshake connection setup with feature negotiation and an explicit distinction between passive and active open: This implies both unicast addressing and a guarantee of return routability.
Single stream-oriented transmission: The stream abstraction atop the datagram service provided by IP is implemented by dividing the stream into segments.

Limited control over segment transmission scheduling (Nagle’s algorithm): This allows for delay minimization in interactive applications by preventing the transport to add additional delays (by deactivating Nagle’s algorithm).

Port multiplexing, with application-to-port mapping during connection setup: Note that in the presence of network address and port translation (NAPT), TCP ports are in effect part of the endpoint address for forwarding purposes.

Full reliability using (S)ACK- and RTO-based loss detection and retransmissions: Loss is sensed using duplicated ACKs ("fast retransmit"), which places a lower bound on the delay inherent in this approach to reliability. The retransmission timeout determines the upper bound on the delay (expect if also exponential back-off is performed). The use of selective acknowledgements further reduces the latency for retransmissions if multiple packets are lost during one congestion event.

Error detection based on a checksum covering the network and transport headers as well as payload: Packets that are detected as corrupted are dropped, relying on the reliability mechanism to retransmit them.

Window-based flow control, with receiver-side window management and signaling of available window: Scaling the flow control window beyond 64kB requires the use of an optional feature, which has performance implications in environments where this option is not supported; this can be the case either if the receiver does not implement window scaling or if a network node on the path strips the window scaling option.

Window-based congestion control reacting to loss, delay, retransmission timeout, or an explicit congestion signal (ECN): Most commonly used is a loss signal from the reliability component’s retransmission mechanism. TCP reacts to a congestion signal by reducing the size of the congestion window; retransmission timeout is generally handled with a larger reaction than other signals.

3.2. Multipath TCP (MPTCP)

Multipath TCP [RFC6824] is an extension for TCP to support multi-homing. It is designed to be as transparent as possible to middle-
boxes. It does so by establishing regular TCP flows between a pair of source/destination endpoints, and multiplexing the application’s stream over these flows.

### 3.2.1. Protocol Description

MPTCP uses TCP options for its control plane. They are used to signal multipath capabilities, as well as to negotiate data sequence numbers, and advertise other available IP addresses and establish new sessions between pairs of endpoints.

### 3.2.2. Interface Description

By default, MPTCP exposes the same interface as TCP to the application. [RFC6897] however describes a richer API for MPTCP-aware applications.

This Basic API describes how an application can - enable or disable MPTCP; - bind a socket to one or more selected local endpoints; - query local and remote endpoint addresses; - get a unique connection identifier (similar to an address-port pair for TCP).

The document also recommend the use of extensions defined for SCTP [RFC6458] (see next section) to deal with multihoming.

[AUTHOR’S NOTE: research work, and some implementation, also suggest that the scheduling algorithm, as well as the path manager, are configurable options that should be exposed to higher layer. Should this be discussed here?]

### 3.2.3. Transport Protocol Components

[AUTHOR’S NOTE: shouldn’t it be "service feature"?]

As an extension to TCP, MPTCP provides mostly the same components. By establishing multiple sessions between available endpoints, it can additionally provide soft failover solutions should one of the paths become unusable. In addition, by multiplexing one byte stream over separate paths, it can achieve a higher throughput than TCP in certain situations (note however that coupled congestion control [RFC6356] might limit this benefit to maintain fairness to other flows at the bottleneck). When aggregating capacity over multiple paths, and depending on the way packets are scheduled on each TCP subflow, an additional delay and higher jitter might be observed before in-order delivery of data to the applications.

The transport protocol components provided by MPTCP in addition to TCP therefore are:
o congestion control with load balancing over multiple connections

o endpoint multiplexing of a single byte stream (higher throughput)

o resilience to network failure and/or handovers

[AUTHOR’S NOTE: it is unclear whether MPTCP has to provide data bundling.]  [AUTHOR’S NOTE: AF multiplexing? sub-flows can be started over IPv4 or IPv6 for the same session]

3.3. Stream Control Transmission Protocol (SCTP)

SCTP is a message oriented standards track transport protocol and the base protocol is specified in [RFC4960]. It supports multi-homing to handle path failures. An SCTP association has multiple unidirectional streams in each direction and provides in-sequence delivery of user messages only within each stream. This allows to minimize head of line blocking. SCTP is extensible and the currently defined extensions include mechanisms for dynamic re-configurations of streams [RFC6525] and IP-addresses [RFC5061]. Furthermore, the extension specified in [RFC3758] introduces the concept of partial reliability for user messages.

SCTP was originally developed for transporting telephony signalling messages and is deployed in telephony signalling networks, especially in mobile telephony networks. Additionally, it is used in the WebRTC framework for data channels and is therefore deployed in all WEB-browsers supporting WebRTC.

3.3.1. Protocol Description

SCTP is a connection oriented protocol using a four way handshake to establish an SCTP association and a three way message exchange to gracefully shut it down. It uses the same port number concept as DCCP, TCP, UDP, and UDP-Lite do and only supports unicast.

SCTP uses the 32-bit CRC32c for protecting SCTP packets against bit errors. This is stronger than the 16-bit checksums used by TCP or UDP. However, a partial checksum coverage as provided by DCCP or UDP-Lite is not supported.

SCTP has been designed with extensibility in mind. Each SCTP packet starts with a single common header containing the port numbers, a verification tag and the CRC32c checksum. This common header is followed by a sequence of chunks. Each chunk consists of a type field, flags, a length field and a value. [RFC4960] defines how a receiver processes chunks with an unknown chunk type. The support of extensions can be negotiated during the SCTP handshake.
SCTP provides a message-oriented service. Multiple small user messages can be bundled into a single SCTP packet to improve the efficiency. For example, this bundling may be done by delaying user messages at the sender side similar to the Nagle algorithm used by TCP. User messages which would result in IP packets larger than the MTU will be fragmented at the sender side and reassembled at the receiver side. There is no protocol limit on the user message size. ICMP-based path MTU discovery as specified for IPv4 in [RFC1191] and for IPv6 in [RFC1981] as well as packetization layer path MTU discovery as specified in [RFC4821] with probe packets using the padding chunks defined in [RFC4820] are supported. [RFC4960] specifies a TCP friendly congestion control to protect the network against overload. SCTP also uses a sliding window flow control to protect receivers against overflow.

Each SCTP association has between 1 and 65536 uni-directional streams in each direction. The number of streams can be different in each direction. Every user-message is sent on a particular stream. User messages can be sent un-ordered or ordered upon request by the upper layer. Un-ordered messages can be delivered as soon as they are completely received. Only all ordered messages sent on the same stream are delivered at the receiver in the same order as sent by the sender. For user messages not requiring fragmentation, this minimises head of line blocking. The base protocol defined in [RFC4960] doesn’t allow interleaving of user-messages, which results in sending a large message on one stream can block the sending of user messages on other streams. [I-D.ietf-tsvwg-sctp-ndata] overcomes this limitation. Furthermore, [I-D.ietf-tsvwg-sctp-ndata] specifies multiple algorithms for the sender side selection of which streams to send data from supporting a variety of scheduling algorithms including priority based ones. The stream re-configuration extension defined in [RFC6525] allows to reset streams during the lifetime of an association and to increase the number of streams, if the number of streams negotiated in the SCTP handshake is not sufficient.

According to [RFC4960], each user message sent is either delivered to the receiver or, in case of excessive retransmissions, the association is terminated in a non-graceful way, similar to the TCP behaviour. In addition to this reliable transfer, the partial reliability extension defined in [RFC3758] allows the sender to abandon user messages. The application can specify the policy for abandoning user messages. Examples for these policies include:

- Limiting the time a user message is dealt with by the sender.
Limiting the number of retransmissions for each fragment of a user message. If the number of retransmissions is limited to 0, one gets a service similar to UDP.

Abandoning messages of lower priority in case of a send buffer shortage.

SCTP supports multi-homing. Each SCTP end-point uses a list of IP-addresses and a single port number. These addresses can be any mixture of IPv4 and IPv6 addresses. These addresses are negotiated during the handshake and the address re-configuration extension specified in [RFC5061] in combination with [RFC4895] can be used to change these addresses in an authenticated way during the livetime of an SCTP association. This allows for transport layer mobility. Multiple addresses are used for improved resilience. If a remote address becomes unreachable, the traffic is switched over to a reachable one, if one exists. Each SCTP end-point supervises continuously the reachability of all peer addresses using a heartbeat mechanism.

For securing user messages, the use of TLS over SCTP has been specified in [RFC3436]. However, this solution does not support all services provided by SCTP (for example un-ordered delivery or partial reliability), and therefore the use of DTLS over SCTP has been specified in [RFC6083] to overcome these limitations. When using DTLS over SCTP, the application can use almost all services provided by SCTP.

[I-D.ietf-tsvwg-natsupp] defines a methods for end-hosts and middleboxes to provide for NAT support for SCTP over IPv4. For legacy NAT traversal, [RFC6951] defines the UDP encapsulation of SCTP-packets. Alternatively, SCTP packets can be encapsulated in DTLS packets as specified in [I-D.ietf-tsvwg-sctp-dtls-encaps]. The latter encapsulation is used with in the WebRTC context.

Having a well defined API is also a feature provided by SCTP as described in the next subsection.

3.3.2. Interface Description

[RFC4960] defines an abstract API for the base protocol. An extension to the BSD Sockets API is defined in [RFC6458] and covers:

- the base protocol defined in [RFC4960].
- the SCTP Partial Reliability extension defined in [RFC3758].
- the SCTP Authentication extension defined in [RFC4895].
the SCTP Dynamic Address Reconfiguration extension defined in [RFC5061].

For the following SCTP protocol extensions the BSD Sockets API extension is defined in the document specifying the protocol extensions:

- the SCTP SACK-IMMEDIATELY extension defined in [RFC7053].
- the SCTP Stream Reconfiguration extension defined in [RFC6525].
- the UDP Encapsulation of SCTP packets extension defined in [RFC6951].
- the additional PR-SCTP policies defined in [I-D.ietf-tsvwg-sctp-prpolicies].

Future documents describing SCTP protocol extensions are expected to describe the corresponding BSD Sockets API extension in a "Socket API Considerations" section.

The SCTP socket API supports two kinds of sockets:

- one-to-one style sockets (by using the socket type "SOCK_STREAM").
- one-to-many style socket (by using the socket type "SOCK_SEQPACKET").

One-to-one style sockets are similar to TCP sockets, there is a 1:1 relationship between the sockets and the SCTP associations (except for listening sockets). One-to-many style SCTP sockets are similar to unconnected UDP sockets as there is a 1:n relationship between the sockets and the SCTP associations.

The SCTP stack can provide information to the applications about state changes of the individual paths and the association whenever they occur. These events are delivered similar to user messages but are specifically marked as notifications.

A couple of new functions have been introduced to support the use of multiple local and remote addresses. Additional SCTP-specific send and receive calls have been defined to allow dealing with the SCTP specific information without using ancillary data in the form of additional cmsgs, which are also defined. These functions provide support for detecting partial delivery of user messages and notifications.
The SCTP socket API allows a fine-grained control of the protocol behaviour through an extensive set of socket options.

The SCTP kernel implementations of FreeBSD, Linux and Solaris follow mostly the specified extension to the BSD Sockets API for the base protocol and the corresponding supported protocol extensions.

3.3.3. Transport Protocol Components

The transport protocol components provided by SCTP are:

- unicast
- connection setup with feature negotiation and application-to-port mapping
- port multiplexing
- reliable or partially reliable delivery
- ordered and unordered delivery within a stream
- support for multiple concurrent streams
- support for stream scheduling prioritization
- flow control
- message-oriented delivery
- congestion control
- user message bundling
- user message fragmentation and reassembly
- strong error detection (CRC32C)
- transport layer multihoming for resilience
- transport layer mobility

[EDITOR’S NOTE: update this list.]

3.4. User Datagram Protocol (UDP)

The User Datagram Protocol (UDP) [RFC0768] [RFC2460] is an IETF standards track transport protocol. It provides a uni-directional,
datagram protocol which preserves message boundaries. It provides none of the following transport features: error correction, congestion control, or flow control. It can be used to send broadcast datagrams (IPv4) or multicast datagrams (IPv4 and IPv6), in addition to unicast (and anycast) datagrams. IETF guidance on the use of UDP is provided in [RFC5405]. UDP is widely implemented and widely used by common applications, especially DNS.

3.4.1. Protocol Description

UDP is a connection-less protocol which maintains message boundaries, with no connection setup or feature negotiation. The protocol uses independent messages, ordinarily called datagrams. The lack of error control and flow control implies messages may be damaged, re-ordered, lost, or duplicated in transit. A receiving application unable to run sufficiently fast or frequently may miss messages. The lack of congestion handling implies UDP traffic may cause the loss of messages from other protocols (e.g., TCP) when sharing the same network paths. UDP traffic can also cause the loss of other UDP traffic in the same or other flows for the same reasons.

Messages with bit errors are ordinarily detected by an invalid end-to-end checksum and are discarded before being delivered to an application. There are some exceptions to this general rule, however. UDP-Lite (see [RFC3828], and below) provides the ability for portions of the message contents to be exempt from checksum coverage. It is also possible to create UDP datagrams with no checksum, and while this is generally discouraged [RFC1122] [RFC5405], certain special cases permit its use [RFC6935]. The checksum support considerations for omitting the checksum are defined in [RFC6936]. Note that due to the relatively weak form of checksum used by UDP, applications that require end to end integrity of data are recommended to include a stronger integrity check of their payload data.

On transmission, UDP encapsulates each datagram into an IP packet, which may in turn be fragmented by IP. Applications concerned with fragmentation or that have other requirements such as receiver flow control, congestion control, PathMTU discovery/PLPMTUd, support for ECN, etc need to be provided by protocols other than UDP [RFC5405].

3.4.2. Interface Description

[RFC0768] describes basic requirements for an API for UDP. Guidance on use of common APIs is provided in [RFC5405].

A UDP endpoint consists of a tuple of (IP address, port number). Demultiplexing using multiple abstract endpoints (sockets) on the
same IP address are supported. The same socket may be used by a single server to interact with multiple clients (note: this behavior differs from TCP, which uses a pair of tuples to identify a connection). Multiple server instances (processes) binding the same socket can cooperate to service multiple clients—the socket implementation arranges to not duplicate the same received unicast message to multiple server processes.

Many operating systems also allow a UDP socket to be "connected", i.e., to bind a UDP socket to a specific (remote) UDP endpoint. Unlike TCP’s connect primitive, 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 [RFC5405].

3.4.3. Transport Protocol Components

The transport protocol components provided by UDP are:

- unidirectional
- port multiplexing
- 2-tuple endpoints
- IPv4 broadcast, multicast and anycast
- IPv6 multicast and anycast
- IPv6 jumbograms
- message-oriented delivery
- error detection (checksum)
- checksum optional

3.5. Lightweight User Datagram Protocol (UDP-Lite)

The Lightweight User Datagram Protocol (UDP-Lite) [RFC3828] is an IETF standards track transport protocol. UDP-Lite provides a bidirectional set of logical unicast or multicast message streams over a datagram protocol. IETF guidance on the use of UDP-Lite is provided in [RFC5405].

3.5.1. Protocol Description
UDP-Lite is a connection-less datagram protocol, with no connection setup or feature negotiation. The protocol use messages, rather than a byte-stream. Each stream of messages is independently managed, therefore retransmission does not hold back data sent using other logical streams.

It provides multiplexing to multiple sockets on each host using port numbers. An active UDP-Lite session is identified by its four-tuple of local and remote IP addresses and local port and remote port numbers.

UDP-Lite fragments packets into IP packets, constrained by the maximum size of IP packet.

UDP-Lite changes the semantics of the UDP "payload length" field to that of a "checksum coverage length" field. Otherwise, UDP-Lite is semantically identical to UDP. Applications using UDP-Lite therefore can not make assumptions regarding the correctness of the data received in the insensitive part of the UDP-Lite payload.

As for UDP, mechanisms for receiver flow control, congestion control, PMTU or PLPMTU discovery, support for ECN, etc need to be provided by upper layer protocols [RFC5405].

Examples of use include a class of applications that can derive benefit from having partially-damaged payloads delivered, rather than discarded. One use is to support error tolerate payload corruption when used over paths that include error-prone links, another application is when header integrity checks are required, but payload integrity is provided by some other mechanism (e.g. [RFC6936]).

A UDP-Lite service may support IPv4 broadcast, multicast, anycast and unicast.

3.5.2. Interface Description

There is no current API specified in the RFC Series, but guidance on use of common APIs is provided in [RFC5405].

The interface of UDP-Lite differs from that of UDP by the addition of a single (socket) option that communicates a checksum coverage length value: at the sender, this specifies the intended checksum coverage, with the remaining unprotected part of the payload called the "error-insensitive part". The checksum coverage may also be made visible to the application via the UDP-Lite MIB module [RFC5097].

3.5.3. Transport Protocol Components
The transport protocol components provided by UDP-Lite are:
- unicast
- IPv4 broadcast, multicast and anycast
- port multiplexing
- non-reliable, non-ordered delivery
- message-oriented delivery
- partial integrity protection

3.6. Datagram Congestion Control Protocol (DCCP)

Datagram Congestion Control Protocol (DCCP) [RFC4340] is an IETF standards track bidirectional transport protocol that provides unicast connections of congestion-controlled unreliable messages.

[EDITOR’S NOTE: Gorry Fairhurst signed up as a contributor for this section.]

The DCCP Problem Statement describes the goals that DCCP sought to address [RFC4336]. It is suitable for applications that transfer fairly large amounts of data and that can benefit from control over the trade off between timeliness and reliability [RFC4336].

It offers low overhead, and many characteristics common to UDP, but can avoid "Re-inventing the wheel" each time a new multimedia application emerges. Specifically it includes core functions (feature negotiation, path state management, RTT calculation, PMTUD, etc): This allows applications to use a compatible method defining how they send packets and where suitable to choose common algorithms to manage their functions. Examples of suitable applications include interactive applications, streaming media or on-line games [RFC4336].

3.6.1. Protocol Description

DCCP is a connection-oriented datagram protocol, providing a three way handshake to allow a client and server to set up a connection, and mechanisms for orderly completion and immediate teardown of a connection. The protocol is defined by a family of RFCs.

It provides multiplexing to multiple sockets on each host using port numbers. An active DCCP session is identified by its four-tuple of local and remote IP addresses and local port and remote port numbers. At connection setup, DCCP also exchanges the the service code
[RFC5595] mechanism to allow transport instantiations to indicate the service treatment that is expected from the network.

The protocol segments data into messages, typically sized to fit in IP packets, but which may be fragmented providing they are less than the A DCCP interface MAY allow applications to request fragmentation for packets larger than PMTU, but not larger than the maximum packet size allowed by the current congestion control mechanism (CCMPS) [RFC4340].

Each message is identified by a sequence number. The sequence number is used to identify segments in acknowledgments, to detect unacknowledged segments, to measure RTT, etc. The protocol may support ordered or unordered delivery of data, and does not itself provide retransmission. There is a Data Checksum option, which contains a strong CRC, lets endpoints detect application data corruption. It also supports reduced checksum coverage, a partial integrity mechanisms similar to UDP-1tite.

Receiver flow control is supported: limiting the amount of unacknowledged data that can be outstanding at a given time.

A DCCP protocol instance can be extended [RFC4340] and tuned. Some features are sender-side only, requiring no negotiation with the receiver; some are receiver-side only, some are explicitly negotiated during connection setup.

DCCP supports negotiation of the congestion control profile, to provide Plug and Play congestion control mechanisms. Examples of specified profiles include [RFC4341] [RFC4342] [RFC5662]. All IETF-defined methods provide Congestion Control.

DCCP use a Connect packet to start a session, and permits half-connections that allow each client to choose features it wishes to support. Simultaneous open [RFC5596], as in TCP, can enable interoperability in the presence of middleboxes. The Connect packet includes a Service Code field [RFC5595] designed to allow middle boxes and endpoints to identify the characteristics required by a session. A lightweight UDP-based encapsulation (DCCP-UDP) has been defined [RFC6773] that permits DCCP to be used over paths where it is not natively supported. Support in NATs is defined in [RFC4340] and [RFC5595].

Upper layer protocols specified on top of DCCP include: DTLS [RFC5595], RTP [RFC5672], ICE/SDP [RFC6773].

A DCCP service is unicast.
A common packet format has allowed tools to evolve that can read and interpret DCCP packets (e.g. Wireshark).

3.6.2. Interface Description

API characteristics include:
- Datagram transmission.
- Notification of the current maximum packet size.
- Send and reception of zero-length payloads.
- Set the Slow Receiver flow control at a receiver.
- Detect a Slow receiver at the sender.

There is no current API specified in the RFC Series.

3.6.3. Transport Protocol Components

The transport protocol components provided by DCCP are:

- unicast
- connection setup with feature negotiation and application-to-port mapping
- Service Codes
- port multiplexing
- non-reliable, ordered delivery
- flow control (slow receiver function)
- drop notification
- timestamps
- message-oriented delivery
- partial integrity protection

3.7. Realtime Transport Protocol (RTP)

RTP provides an end-to-end network transport service, suitable for applications transmitting real-time data, such as audio, video or data, over multicast or unicast network services, including TCP, UDP, UDP-Lite, DCCP.

[EDITOR’S NOTE: Varun Singh signed up as contributor for this section. Given the complexity of RTP, suggest to have an abbreviated section here contrasting RTP with other transports, and focusing on those features that are RTP-unique.]
3.8. NACK-Oriented Reliable Multicast (NORM)

NORM is an IETF standards track protocol specified in [RFC5740]. The protocol was designed to support reliable bulk data dissemination to receiver groups using IP Multicast but also provides for point-to-point unicast operation. Its support for bulk data dissemination includes discrete file or computer memory-based "objects" as well as byte- and message-streaming. NORM is designed to incorporate packet erasure coding as an inherent part of its selective ARQ in response to receiver negative acknowledgements. The packet erasure coding can also be proactively applied for forward protection from packet loss. NORM transmissions are governed by TCP-friendly congestion control. NORM's reliability, congestion control, and flow control mechanism are distinct components and can be separately controlled to meet different application needs.

3.8.1. Protocol Description

[EDITOR'S NOTE: needs to be more clear about the application of FEC and packet erasure coding; expand ARQ.]

The NORM protocol is encapsulated in UDP datagrams and thus provides multiplexing for multiple sockets on hosts using port numbers. For purposes of loosely coordinated IP Multicast, NORM is not strictly connection-oriented although per-sender state is maintained by receivers for protocol operation. [RFC5740] does not specify a handshake protocol for connection establishment and separate session initiation can be used to coordinate port numbers. However, in-band "client-server" style connection establishment can be accomplished with the NORM congestion control signaling messages using port binding techniques like those for TCP client-server connections.

NORM supports bulk "objects" such as file or in-memory content but also can treat a stream of data as a logical bulk object for purposes of packet erasure coding. In the case of stream transport, NORM can support either byte streams or message streams where application-defined message boundary information is carried in the NORM protocol messages. This allows the receiver(s) to join/re-join and recover message boundaries mid-stream as needed. Application content is carried and identified by the NORM protocol with encoding symbol identifiers depending upon the Forward Error Correction (FEC) Scheme [RFC3452] configured. NORM uses NACK-based selective ARQ to reliably deliver the application content to the receiver(s). NORM proactively measures round-trip timing information to scale ARQ timers appropriately and to support congestion control. For multicast operation, timer-based feedback suppression is used to achieve group size scaling with low feedback traffic levels. The feedback suppression is not applied for unicast operation.
NORM uses rate-based congestion control based upon the TCP-Friendly Rate Control (TFRC) [RFC4324] principles that are also used in DCCP [RFC4340]. NORM uses control messages to measure RTT and collect congestion event (e.g., loss event, ECN event, etc) information from the receiver(s) to support dynamic rate control adjustment. The TCP-Friendly Multicast Congestion Control (TFMCC) [RFC4654] used provides some extra features to support multicast but is functionally equivalent to TFRC in the unicast case.

NORM’s reliability mechanism is decoupled from congestion control. This allows alternative arrangements of transport services to be invoked. For example, fixed-rate reliable delivery can be supported or unreliable (but optionally "better than best effort" via packet erasure coding) delivery with rate-control per TFRC can be achieved. Additionally, alternative congestion control techniques may be applied. For example, TFRC rate control with congestion event detection based on ECN for links with high packet loss (e.g., wireless) has been implemented and demonstrated with NORM.

While NORM is NACK-based for reliability transfer, it also supports a positive acknowledgment (ACK) mechanism that can be used for receiver flow control. Again, since this mechanism is decoupled from the reliability and congestion control, applications that have different needs in this aspect can use the protocol differently. One example is the use of NORM for quasi-reliable delivery where timely delivery of newer content may be favored over completely reliable delivery of older content within buffering and RTT constraints.

3.8.2. Interface Description

The NORM specification does not describe a specific application programming interface (API) to control protocol operation. A freely-available, open source reference implementation of NORM is available at https://www.nrl.navy.mil/itd/ncs/products/norm, and a documented API is provided for this implementation. While a sockets-like API is not currently documented, the existing API supports the necessary functions for that to be implemented.

3.8.3. Transport Protocol Components

The transport protocol components provided by NORM are:

- unicast
- multicast
- port multiplexing (UDP ports)
- reliable delivery
- unordered delivery of in-memory data or file bulk content objects
- error detection (UDP checksum)
- segmentation
- stream-oriented delivery in a single stream
- object-oriented delivery of discrete data or file items
- data bundling (Nagle’s algorithm)
- flow control (timer-based and/or ack-based)
- congestion control
- packet erasure coding (both proactively and as part of ARQ)

3.9. Transport Layer Security (TLS) and Datagram TLS (DTLS) as a pseudotransport

Transport Layer Security (TLS) and Datagram TLS (DTLS) are IETF protocols that provide several security-related features to applications. TLS is designed to run on top of a reliable streaming transport protocol (usually TCP), while DTLS is designed to run on top of a best-effort datagram protocol (usually UDP). At the time of writing, the current version of TLS is 1.2; it is defined in [RFC5246]. DTLS provides nearly identical functionality to applications; it is defined in [RFC6347] and its current version is also 1.2. The TLS protocol evolved from the Secure Sockets Layer (SSL) protocols developed in the mid 90s to support protection of HTTP traffic.

While older versions of TLS and DTLS are still in use, they provide weaker security guarantees. [RFC7457] outlines important attacks on TLS and DTLS. [RFC7525] is a Best Current Practices (BCP) document that describes secure configurations for TLS and DTLS to counter these attacks. The recommendations are applicable for the vast majority of use cases.

[NOTE: The Logjam authors (weakdh.org) give (inconclusive) evidence that one of the recommendations of [RFC7525], namely the use of DHE-1024 as a fallback, may not be sufficient in all cases to counter an attacker with the resources of a nation-state. It is unclear at this time if the RFC is going to be updated as a result, or whether there will be an RFC7525bis.]
3.9.1. Protocol Description

Both TLS and DTLS provide the same security features and can thus be discussed together. The features they provide are:

- Confidentiality
- Data integrity
- Peer authentication (optional)
- Perfect forward secrecy (optional)

The authentication of the peer entity can be omitted; a common web use case is where the server is authenticated and the client is not. TLS also provides a completely anonymous operation mode in which neither peer’s identity is authenticated. It is important to note that TLS itself does not specify how a peering entity’s identity should be interpreted. For example, in the common use case of authentication by means of an X.509 certificate, it is the application’s decision whether the certificate of the peering entity is acceptable for authorization decisions. Perfect forward secrecy, if enabled and supported by the selected algorithms, ensures that traffic encrypted and captured during a session at time t₀ cannot be later decrypted at time t₁ (t₁ > t₀), even if the long-term secrets of the communicating peers are later compromised.

As DTLS is generally used over an unreliable datagram transport such as TCP, applications will need to tolerate loss, re-ordered, or duplicated datagrams. Like TLS, DTLS conveys application data in a sequence of independent records. However, because records are mapped to unreliable datagrams, there are several features unique to DTLS that are not applicable to TLS:

- Record replay detection (optional)
- Record size negotiation (estimates of PMTU and record size expansion factor)
- Conveyance of IP don’t fragment (DF) bit settings by application
- An anti-DoS stateless cookie mechanism (optional)

Generally, DTLS follows the TLS design as closely as possible. To operate over datagrams, DTLS includes a sequence number and limited forms of retransmission and fragmentation for its internal operations. The sequence number may be used for detecting replayed information, according to the windowing procedure described in...
Section 4.1.2.6 of [RFC6347]. Note also that DTLS bans the use of stream ciphers, which are essentially incompatible when operating on independent encrypted records.

3.9.2. Interface Description

TLS is commonly invoked using an API provided by packages such as OpenSSL, wolfSSL, or GnuTLS. Using such APIs entails the manipulation of several important abstractions, which fall into the following categories: long-term keys and algorithms, session state, and communications/connections. There may also be special APIs required to deal with time and/or random numbers, both of which are needed by a variety of encryption algorithms and protocols.

Considerable care is required in the use of TLS APIs in order to create a secure application. The programmer should have at least a basic understanding of encryption and digital signature algorithms and their strengths, public key infrastructure (including X.509 certificates and certificate revocation), and the sockets API. See [RFC7525] and [RFC7457], as mentioned above.

As an example, in the case of OpenSSL, the primary abstractions are the library itself and method (protocol), session, context, cipher and connection. After initializing the library and setting the method, a cipher suite is chosen and used to configure a context object. Session objects may then be minted according to the parameters present in a context object and associated with individual connections. Depending on how precisely the programmer wishes to select different algorithmic or protocol options, various levels of details may be required.

3.9.3. Transport Protocol Components

Both TLS and DTLS employ a layered architecture. The lower layer is commonly called the record protocol. It is responsible for fragmenting messages, applying message authentication codes (MACs), encrypting data, and invoking transmission from the underlying transport protocol. DTLS augments the TLS record protocol with sequence numbers used for ordering and replay detection.

Several protocols are layered on top of the record protocol. These include the handshake, alert, and change cipher spec protocols. There is also the data protocol, used to carry application traffic. The handshake protocol is used to establish cryptographic and compression parameters when a connection is first set up. In DTLS, this protocol also has a basic fragmentation and retransmission capability and a cookie-like mechanism to resist DoS attacks. (TLS compression is not recommended at present). The alert protocol is
used to inform the peer of various conditions, most of which are terminal for the connection. The change cipher spec protocol is used to synchronize changes in cryptographic parameters for each peer.

3.10. Hypertext Transport Protocol (HTTP) over TCP as a pseudotransport

Hypertext Transfer Protocol (HTTP) is an application-level protocol widely used on the Internet. Version 1.1 of the protocol is specified in [RFC7230] [RFC7231] [RFC7232] [RFC7233] [RFC7234] [RFC7235], and version 2 in [RFC7540]. Furthermore, HTTP is used as a substrate for other application-layer protocols. There are various reasons for this practice listed in [RFC3205]; these include being a well-known and well-understood protocol, reusability of existing servers and client libraries, easy use of existing security mechanisms such as HTTP digest authentication [RFC2617] and TLS [RFC5246], the ability of HTTP to traverse firewalls which makes it work with a lot of infrastructure, and cases where a application server often needs to support HTTP anyway.

Depending on application’s needs, the use of HTTP as a substrate protocol may add complexity and overhead in comparison to a special-purpose protocol (e.g. HTTP headers, suitability of the HTTP security model etc.). [RFC3205] address this issues and provides some guidelines and concerns about the use of HTTP standard port 80 and 443, the use of HTTP URL scheme and interaction with existing firewalls, proxies and NATs.

Though not strictly bound to TCP, HTTP is almost exclusively run over TCP, and therefore inherits its properties when used in this way.

3.10.1. Protocol Description

Hypertext Transfer Protocol (HTTP) is a request/response protocol. A client sends a request containing a request method, URI and protocol version followed by a MIME-like message (see [RFC7231] for the differences between an HTTP object and a MIME message), containing information about the client and request modifiers. The message can contain a message body carrying application data as well. The server responds with a status or error code followed by a MIME-like message containing information about the server and information about carried data and it can include a message body. It is possible to specify a data format for the message body using MIME media types [RFC2045]. Furthermore, the protocol has numerous additional features; features relevant to pseudotransport are described below.

Content negotiation, specified in [RFC7231], is a mechanism provided by HTTP for selecting a representation on a requested resource. The client and server negotiate acceptable data formats, charsets, data
encoding (e.g. data can be transferred compressed, gzip), etc. HTTP can accommodate exchange of messages as well as data streaming (using chunked transfer encoding [RFC7230]). It is also possible to request a part of a resource using range requests specified in [RFC7233]. The protocol provides powerful cache control signalling defined in [RFC7234].

HTTP 1.1’s and HTTP 2.0’s persistent connections can be use to perform multiple request-response transactions during the life-time of a single HTTP connection. Moreover, HTTP 2.0 connections can multiplex many request/response pairs in parallel on a single connection. This reduces connection establishment overhead and the effect of TCP slow-start on each transaction, important for HTTP’s primary use case.

It is possible to combine HTTP with security mechanisms, like TLS (denoted by HTTPS), which adds protocol properties provided by such a mechanism (e.g. authentication, encryption, etc.). TLS’s Application-Layer Protocol Negotiation (ALPN) extension [RFC7301] can be used for HTTP version negotiation within TLS handshake which eliminates addition round-trip. Arbitrary cookie strings, included as part of the MIME headers, are often used as bearer tokens in HTTP.

Application layer protocols using HTTP as substrate may use existing method and data formats, or specify new methods and data formats. Furthermore some protocols may not fit a request/response paradigm and instead rely on HTTP to send messages (e.g. [RFC6546]). Because HTTP is working in many restricted infrastructures, it is also used to tunnel other application-layer protocols.

3.10.2. Interface Description

There are many HTTP libraries available exposing different APIs. The APIs provide a way to specify a request by providing a URI, a method, request modifiers and optionally a request body. For the response, callbacks can be registered that will be invoked when the response is received. If TLS is used, API expose a registration of callbacks in case a server requests client authentication and when certificate verification is needed.

World Wide Web Consortium (W3C) standardized the XMLHttpRequest API [XHR], an API that can be use for sending HTTP/HTTPS requests and receiving server responses. Besides XML data format, request and response data format can also be JSON, HTML and plain text. Specifically JavaScript and XMLHttpRequest are a ubiquitous programming model for websites, and more general applications, where native code is less attractive.
Representational State Transfer (REST) [REST] is another example how applications can use HTTP as transport protocol. REST is an architecture style for building application on the Internet. It uses HTTP as a communication protocol.

3.10.3. Transport Protocol Components

The transport protocol components provided by HTTP, when used as a pseudotransport, are:

- unicast
- reliable delivery
- ordered delivery
- message and stream-oriented
- object range request
- message content type negotiation
- congestion control

HTTPS (HTTP over TLS) additionally provides the following components:

- authentication (of one or both ends of a connection)
- confidentiality
- integrity protection

3.11. WebSockets

[RFC6455]

[EDITOR’S NOTE: Salvatore Loreto will contribute text for this section.]

3.11.1. Protocol Description

3.11.2. Interface Description
3.11.3. Transport Protocol Components

4. Transport Service Features

[EDITOR’S NOTE: This section is still work-in-progress. This list is probably not complete and/or too detailed.]

The transport protocol components analyzed in this document which can be used as a basis for defining common transport service features, normalized and separated into categories, are as follows:

- Control Functions
  - Addressing
    - unicast
    - broadcast (IPv4 only)
    - multicast
    - anycast
    - something on ports and NAT
  - Multihoming support
    - multihoming for resilience
    - multihoming for mobility
      - specify handover latency?
    - multihoming for load-balancing
      - specify interleaving delay?
  - Multiplexing
    - application to port mapping
    - single vs. multiple streaming
- Delivery
  - reliability
    - reliable delivery
+ partially reliable delivery
  - packet erasure coding
+ unreliable delivery
  - drop notification
  - Integrity protection
    o checksum for error detection
    o partial checksum protection
    o checksum optional
* ordering
  + ordered delivery
  + unordered delivery
    - unordered delivery of in-memory data
* type/framing
  + stream-oriented delivery
  + message-oriented delivery
  + object-oriented delivery of discrete data or file items
    - object content type negotiation
  + range-based partial object transmission
  + file bulk content objects

  o Transmission control
* rate control
  + timer-based
  + ACK-based
* congestion control
* flow control
* segmentation
* data/message bundling (Nagle’s algorithm)
* stream scheduling prioritization

- Security
  * authentication of one end of a connection
  * authentication of both ends of a connection
  * confidentiality
  * cryptographic integrity protection

The next revision of this document will define transport service features based upon this list.

[EDITOR’S NOTE: this section will draw from the candidate features provided by protocol components in the previous section - please discuss on taps@ietf.org list]

4.1. Complete Protocol Feature Matrix

[EDITOR’S NOTE: Dave Thaler has signed up as a contributor for this section. Michael Welzl also has a beginning of a matrix which could be useful here.]

[EDITOR’S NOTE: The below is a strawman proposal below by Gorry Fairhurst for initial discussion]

The table below summarises protocol mechanisms that have been standardised. It does not make an assessment on whether specific implementations are fully compliant to these specifications.

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>UDP</th>
<th>UDP-L</th>
<th>DCCP</th>
<th>SCTP</th>
<th>TCP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unicast</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Mcast/IPv4Bcast</td>
<td>Yes(2)</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Port Mux</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Mode</td>
<td>Dgram</td>
<td>Dgram</td>
<td>Dgram</td>
<td>Dgram</td>
<td>Stream</td>
</tr>
<tr>
<td>Connected</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Data bundling</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Feature Nego</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Options</td>
<td>No</td>
<td>No</td>
<td>Support</td>
<td>Support</td>
<td>Support</td>
</tr>
<tr>
<td>Data priority</td>
<td>*</td>
<td>*</td>
<td>Yes</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Data bundling</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Reliability</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>Select</td>
<td>Full</td>
</tr>
<tr>
<td>Ordered deliv</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Stream</td>
<td>Yes</td>
</tr>
<tr>
<td>Corruption Tol.</td>
<td>No</td>
<td>Support</td>
<td>Support</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Flow Control</td>
<td>No</td>
<td>No</td>
<td>Support</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>PMTU/PLPMTU</td>
<td>(1)</td>
<td>(1)</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Cong Control</td>
<td>(1)</td>
<td>(1)</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>ECN Support</td>
<td>(1)</td>
<td>(1)</td>
<td>Yes</td>
<td>TBD</td>
<td>Yes</td>
</tr>
<tr>
<td>NAT support</td>
<td>Limited</td>
<td>Limited</td>
<td>Support</td>
<td>TBD</td>
<td>Support</td>
</tr>
<tr>
<td>Security</td>
<td>DTLS</td>
<td>DTLS</td>
<td>DTLS</td>
<td>TLS, AO</td>
<td></td>
</tr>
<tr>
<td>UDP encaps</td>
<td>N/A</td>
<td>None</td>
<td>Yes</td>
<td>Yes</td>
<td>None</td>
</tr>
<tr>
<td>RTP support</td>
<td>Support</td>
<td>Support</td>
<td>Support</td>
<td>?</td>
<td>Support</td>
</tr>
</tbody>
</table>

Note (1): this feature requires support in an upper layer protocol.

Note (2): this feature requires support in an upper layer protocol when used with IPv6.

5. IANA Considerations

This document has no considerations for IANA.

6. Security Considerations
This document surveys existing transport protocols and protocols providing transport-like services. Confidentiality, integrity, and authenticity are among the features provided by those services. This document does not specify any new components or mechanisms for providing these features. Each RFC listed in this document discusses the security considerations of the specification it contains.

7. Contributors

[Editor’s Note: turn this into a real contributors section with addresses once we figure out how to trick the toolchain into doing so]

- Section 3.2 on MPTCP was contributed by Simone Ferlin-Oliviera (ferlin@simula.no) and Olivier Mehani (olivier.mehani@nicta.com.au)
- Section 3.4 on UDP was contributed by Kevin Fall (kfall@kfall.com)
- Section 3.3 on SCTP was contributed by Michael Tuexen (tuexen@fh-muenster.de)
- Section 3.8 on NORM was contributed by Brian Adamson (brian.adamson@nrl.navy.mil)
- Section 3.9 on MPTCP was contributed by Ralph Holz (ralph.holz@nicta.com.au) and Olivier Mehani (olivier.mehani@nicta.com.au)
- Section 3.10 on HTTP was contributed by Dragana Damjanovic (ddamjanovic@mozilla.com)

8. Acknowledgments

Thanks to Karen Nielsen, Joe Touch, and Michael Welzl for the comments, feedback, and discussion. This work is partially supported by the European Commission under grant agreement FP7-ICT-318627 mPlane; support does not imply endorsement.

[EDITOR’S NOTE: add H2020-NEAT ack].

9. References
9.1. Normative References


9.2. Informative References


Authors’ Addresses

Godred Fairhurst (editor)
University of Aberdeen
School of Engineering, Fraser Noble Building
Aberdeen AB24 3UE

Email: gorry@erg.abdn.ac.uk

Brian Trammell (editor)
ETH Zurich
Gloriasterasse 35
8092 Zurich
Switzerland

Email: ietf@trammell.ch

Mirja Kuehlewind (editor)
ETH Zurich
Gloriasterasse 35
8092 Zurich
Switzerland

Email: mirja.kuehlewind@tik.ee.ethz.ch
Services provided by IETF transport protocols and congestion control mechanisms
draft-ietf-taps-transports-14

Abstract

This document describes, surveys, and classifies the protocol mechanisms provided by existing IETF protocols, as background for determining a common set of transport services. It examines the Transmission Control Protocol (TCP), Multipath TCP, the Stream Control Transmission Protocol (SCTP), the User Datagram Protocol (UDP), UDP-Lite, the Datagram Congestion Control Protocol (DCCP), the Internet Control Message Protocol (ICMP), the Realtime Transport Protocol (RTP), File Delivery over Unidirectional Transport/Asynchronous Layered Coding Reliable Multicast (FLUTE/ALC), and NACK-Oriented Reliable Multicast (NORM), Transport Layer Security (TLS), Datagram TLS (DTLS), and the Hypertext Transport Protocol (HTTP), when HTTP is used as a pseudotransport. This survey provides background for the definition of transport services within the TAPS working group.

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 June 9, 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 .................................................. 3
   1.1. Overview of Transport Features .......................... 4
2. Terminology ................................................... 5
3. Existing Transport Protocols ................................. 6
   3.1. Transport Control Protocol (TCP) ......................... 6
       3.1.1. Protocol Description ................................ 6
       3.1.2. Interface description .............................. 8
       3.1.3. Transport Features ............................... 8
   3.2. Multipath TCP (MPTCP) .................................. 9
       3.2.1. Protocol Description ................................ 9
       3.2.2. Interface Description .............................. 10
       3.2.3. Transport features ............................... 10
   3.3. User Datagram Protocol (UDP) ............................ 11
       3.3.1. Protocol Description ................................ 11
       3.3.2. Interface Description .............................. 12
       3.3.3. Transport Features ............................... 12
   3.4. Lightweight User Datagram Protocol (UDP-Lite) ......... 13
       3.4.1. Protocol Description ................................ 13
       3.4.2. Interface Description .............................. 13
       3.4.3. Transport Features ............................... 14
   3.5. Stream Control Transmission Protocol (SCTP) ........... 14
       3.5.1. Protocol Description ................................ 14
       3.5.2. Interface Description .............................. 16
       3.5.3. Transport Features ............................... 19
   3.6. Datagram Congestion Control Protocol (DCCP) .......... 20
       3.6.1. Protocol Description ................................ 20
       3.6.2. Interface Description .............................. 21
       3.6.3. Transport Features ............................... 22
   3.7. Transport Layer Security (TLS) and Datagram TLS (DTLS) as a pseudotransport .................................... 22
       3.7.1. Protocol Description ................................ 23
1. Introduction

Internet applications make use of the Services provided by a Transport protocol, such as TCP (a reliable, in-order stream protocol) or UDP (an unreliable datagram protocol). We use the term "Transport Service" to mean the end-to-end service provided to an application by the transport layer. That service can only be provided correctly if information about the intended usage is supplied from the application. The application may determine this information at design time, compile time, or run time, and may include guidance on whether a feature is required, a preference by the application, or something in between. Examples of features of Transport Services are reliable delivery, ordered delivery, content privacy to in-path devices, and integrity protection.
The IETF has defined a wide variety of transport protocols beyond TCP and UDP, including SCTP, DCCP, MPTCP, and UDP-Lite. Transport services may be provided directly by these transport protocols, or layered on top of them using protocols such as WebSockets (which runs over TCP), RTP (over TCP or UDP) or WebRTC data channels (which run over SCTP over DTLS over UDP or TCP). Services built on top of UDP or UDP-Lite typically also need to specify additional mechanisms, including a congestion control mechanism (such as NewReno [RFC6582], TFRC [RFC5348] or LEDBAT [RFC6817]). This extends the set of available Transport Services beyond those provided to applications by TCP and UDP.

The transport protocols described in this document provide a basis for the definition of transport services provided by common protocols, as background for the TAPS working group. The protocols listed here were chosen to help expose as many potential transport services as possible, and are not meant to be a comprehensive survey or classification of all transport protocols.

1.1. Overview of Transport Features

Transport protocols can be differentiated by the features of the services they provide.

Some of these provided features are closely related to basic control function that a protocol needs to work over a network path, such as addressing. The number of participants in a given association also determines its applicability: if a connection is between endpoints (unicast), to one of multiple endpoints (anycast), or simultaneously to multiple endpoints (multicast). Unicast protocols usually support bidirectional communication, while multicast is generally unidirectional. Another feature is whether a transport requires a control exchange across the network at setup (e.g., TCP), or whether it is connection-less (e.g., UDP).

For packet delivery itself, reliability and integrity protection, ordering, and framing are basic features. However, these features are implemented with different levels of assurance in different protocols. As an example, a transport service may provide full reliability, providing detection of loss and retransmission (e.g., TCP). SCTP offers a message-based service that can provide full or partial reliability, and allows the protocol to minimize the head of line blocking due to the support of ordered and unordered message delivery within multiple streams. UDP-Lite and DCCP can provide partial integrity protection to enable corruption tolerance.

Usually a protocol has been designed to support one specific type of delivery/framing: data either needs to be divided into transmission
units based on network packets (datagram service), a data stream is segmented and re-combined across multiple packets (stream service), or whole objects such as files are handled accordingly. This decision strongly influences the interface that is provided to the upper layer.

In addition, transport protocols offer a certain support for transmission control. For example, a transport service can provide flow control to allow a receiver to regulate the transmission rate of a sender. Further a transport service can provide congestion control (see Section 4). As an example TCP and SCTP provide congestion control for use in the Internet, whereas UDP leaves this function to the upper layer protocol that uses UDP.

Security features are often provided independent of the transport protocol, via Transport Layer Security (TLS, see Section 3.7) or by the application layer protocol itself. The security properties TLS provides to the application (such as confidentiality, integrity, and authenticity) are also features of the transport layer, even though they are often presently implemented in a separate protocol.

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 Service 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).
3. Existing Transport Protocols

This section provides a list of known IETF transport protocols and transport protocol frameworks. It does not make an assessment about whether specific implementations of protocols are fully compliant to current IETF specifications.

3.1. Transport Control Protocol (TCP)

TCP is an IETF standards track transport protocol. [RFC0793] introduces TCP as follows: "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." Since its introduction, TCP has become the default connection-oriented, stream-based transport protocol in the Internet. It is widely implemented by endpoints and widely used by common applications.

3.1.1. Protocol Description

TCP is a connection-oriented protocol, providing a three way handshake to allow a client and server to set up a connection and negotiate features, and mechanisms for orderly completion and immediate teardown of a connection. TCP is defined by a family of RFCs [RFC7414].

TCP provides multiplexing to multiple sockets on each host using port numbers. A similar approach is adopted by other IETF-defined transports. An active TCP session is identified by its four-tuple of local and remote IP addresses and local port and remote port numbers. The destination port during connection setup is often used to indicate the requested service.

TCP partitions a continuous stream of bytes into segments, sized to fit in IP packets based on a negotiated maximum segment size and further constrained by the effective Maximum Transmission Unit (MTU) from Path MTU Discovery (PMTUD). ICMP-based Path MTU discovery [RFC1191][RFC1981] as well as Packetization Layer Path MTU Discovery (PMTUD) [RFC4821] have been defined by the IETF.

Each byte in the stream is identified by a sequence number. The sequence number is used to order segments on receipt, to identify segments in acknowledgments, and to detect unacknowledged segments for retransmission. This is the basis of the reliable, ordered delivery of data in a TCP stream. TCP Selective Acknowledgment (SACK) [RFC2018] extends this mechanism by making it possible to provide earlier identification of which segments are missing,
allowing faster retransmission. SACK-based methods (e.g. Duplicate Selective ACK) can also result in less spurious retransmission.

Receiver flow control is provided by a sliding window: limiting the amount of unacknowledged data that can be outstanding at a given time. The window scale option [RFC7323] allows a receiver to use windows greater than 64KB.

All TCP senders provide congestion control, such as described in [RFC5681]. TCP uses a sequence number with a sliding receiver window for flow control. The TCP congestion control mechanism also utilises this TCP sequence number to manage a separate congestion window [RFC5681]. The sending window at a given point in time is the minimum of the receiver window and the congestion window. The congestion window is increased in the absence of congestion and, respectively, decreased if congestion is detected. Often loss is implicitly handled as a congestion indication which is detected in TCP (also as input for retransmission handling) based on two mechanisms: A retransmission timer with exponential back-off or the reception of three acknowledgments for the same segment, so called duplicated ACKs (Fast retransmit). In addition, Explicit Congestion Notification (ECN) [RFC3168] can be used in TCP, if supported by both endpoints, that allows a network node to signal congestion without inducing loss. Alternatively, a delay-based congestion control scheme can be used in TCP that reacts to changes in delay as an early indication of congestion as also further described in Section 4. Examples for different kind of congestion control schemes are given in Section 4.

TCP protocol instances can be extended [RFC7414] and tuned. Some features are sender-side only, requiring no negotiation with the receiver; some are receiver-side only, some are explicitly negotiated during connection setup.

TCP may buffer data, e.g., to optimize processing or capacity usage. TCP can therefore provides mechanisms to control this, including an optional "PUSH" function [RFC0793] that explicitly requests the transport service not to delay data. By default, TCP segment partitioning uses Nagle's algorithm [RFC0896] to buffer data at the sender into large segments, potentially incurring sender-side buffering delay; this algorithm can be disabled by the sender to transmit more immediately, e.g., to reduce latency for interactive sessions.

TCP provides an "urgent data" function for limited out-of-order delivery of the data. This function is deprecated [RFC6093].
A TCP Reset (RST) control message may be used to force a TCP endpoint to close a session [RFC0793], aborting the connection.

A mandatory checksum provides a basic integrity check against misdelivery and data corruption over the entire packet. Applications that require end to end integrity of data are recommended to include a stronger integrity check of their payload data. The TCP checksum [RFC1071] [RFC2460] does not support partial payload protection (as in DCCP/UDP-Lite).

TCP supports only unicast connections.

3.1.2. Interface description

A User/TCP Interface is defined in [RFC0793] providing six user commands: Open, Send, Receive, Close, Status. This interface does not describe configuration of TCP options or parameters beside use of the PUSH and URGENT flags.

[RFC1122] describes extensions of the TCP/application layer interface for:

- reporting soft errors such as reception of ICMP error messages, extensive retransmission or urgent pointer advance,
- providing a possibility to specify the Differentiated Services Code Point (DSCP) [RFC3260] (formerly, the Type-of-Service, TOS) for segments,
- providing a flush call to empty the TCP send queue, and
- multihoming support.

In API implementations derived from the BSD Sockets API, TCP sockets are created using the "SOCK_STREAM" socket type as described in the IEEE Portable Operating System Interface (POSIX) Base Specifications [POSIX]. The features used by a protocol instance may be set and tuned via this API. There are currently no documents in the RFC Series that describe this interface.

3.1.3. Transport Features

The transport features provided by TCP are:

- connection-oriented transport with feature negotiation and application-to-port mapping (implemented using SYN segments and the TCP option field to negotiate features),
o unicast transport (though anycast TCP is implemented, at risk of instability due to rerouting),

o port multiplexing,

o uni- or bidirectional communication,

o stream-oriented delivery in a single stream,

o fully reliable delivery (implemented using ACKs sent from the receiver to confirm delivery),

o error detection (implemented using a segment checksum to verify delivery to the correct endpoint and integrity of the data and options),

o segmentation,

o data bundling (optional; uses Nagle’s algorithm to coalesce data sent within the same RTT into full-sized segments),

o flow control (implemented using a window-based mechanism where the receiver advertises the window that it is willing to buffer),

o congestion control (usually implemented using a window-based mechanism and four algorithms for different phases of the transmission: slow start, congestion avoidance, fast retransmit, and fast recovery [RFC5681]).

3.2. Multipath TCP (MPTCP)

Multipath TCP [RFC6824] is an extension for TCP to support multi-homing for resilience, mobility and load-balancing. It is designed to be as indistinguishable to middleboxes from non-multipath TCP as possible. It does so by establishing regular TCP flows between a pair of source/destination endpoints, and multiplexing the application’s stream over these flows. Sub-flows can be started over IPv4 or IPv6 for the same session.

3.2.1. Protocol Description

MPTCP uses TCP options for its control plane. They are used to signal multipath capabilities, as well as to negotiating data sequence numbers, and advertise other available IP addresses and establish new sessions between pairs of endpoints.

By multiplexing one byte stream over separate paths, MPTCP can achieve a higher throughput than TCP in certain situations. However,
if coupled congestion control [RFC6356] is used, it might limit this benefit to maintain fairness to other flows at the bottleneck. When aggregating capacity over multiple paths, and depending on the way packets are scheduled on each TCP subflow, additional delay and higher jitter might be observed before in-order delivery of data to the applications.

3.2.2. Interface Description

By default, MPTCP exposes the same interface as TCP to the application. [RFC6897] however describes a richer API for MPTCP-aware applications.

This Basic API describes how an application can:

- enable or disable MPTCP.
- bind a socket to one or more selected local endpoints.
- query local and remote endpoint addresses.
- get a unique connection identifier (similar to an address-port pair for TCP).

The document also recommends the use of extensions defined for SCTP [RFC6458] (see next section) to support multihoming for resilience and mobility.

3.2.3. Transport features

As an extension to TCP, MPTCP provides mostly the same features. By establishing multiple sessions between available endpoints, it can additionally provide soft failover solutions in the case that one of the paths become unusable.

The transport features provided by MPTCP in addition to TCP therefore are:

- multihoming for load-balancing, with endpoint multiplexing of a single byte stream, using either coupled congestion control or for throughput maximization,
- address family multiplexing (using IPv4 and IPv6 for the same session),
- resilience to network failure and/or handover.
3.3. User Datagram Protocol (UDP)

The User Datagram Protocol (UDP) [RFC0768] [RFC2460] is an IETF standards track transport protocol. It provides a unidirectional datagram protocol that preserves message boundaries. It provides no error correction, congestion control, or flow control. It can be used to send broadcast datagrams (IPv4) or multicast datagrams (IPv4 and IPv6), in addition to unicast and anycast datagrams. IETF guidance on the use of UDP is provided in [I-D.ietf-tsvwg-rfc5405bis]. UDP is widely implemented and widely used by common applications, including DNS.

3.3.1. Protocol Description

UDP is a connection-less protocol that maintains message boundaries, with no connection setup or feature negotiation. The protocol uses independent messages, ordinarily called datagrams. It provides detection of payload errors and misdelivery of packets to an unintended endpoint, either of which result in discard of received datagrams, with no indication to the user of the service.

It is possible to create IPv4 UDP datagrams with no checksum, and while this is generally discouraged [RFC1122] [I-D.ietf-tsvwg-rfc5405bis], certain special cases permit this use. These datagrams rely on the IPv4 header checksum to protect from misdelivery to an unintended endpoint. IPv6 does not permit UDP datagrams with no checksum, although in certain cases [RFC6936] this rule may be relaxed [RFC6935].

UDP does not provide reliability and does not provide retransmission. Messages may be re-ordered, lost, or duplicated in transit. Note that due to the relatively weak form of checksum used by UDP, applications that require end to end integrity of data are recommended to include a stronger integrity check of their payload data.

Because UDP provides no flow control, a receiving application that is unable to run sufficiently fast, or frequently, may miss messages. The lack of congestion handling implies UDP traffic may experience loss when using an overloaded path, and may cause the loss of messages from other protocols (e.g., TCP) when sharing the same network path.

On transmission, UDP encapsulates each datagram into a single IP packet or several IP packet fragments. This allows a datagram to be larger than the effective path MTU. Fragments are reassembled before delivery to the UDP receiver, making this transparent to the user of
the transport service. When the jumbograms are supported, larger messages may be sent without performing fragmentation.

UDP on its own does not provide support for segmentation, receiver flow control, congestion control, PathMTU discovery/PLPMTUD, or ECN. Applications that require these features need to provide them on their own, or by using a protocol over UDP that provides them [I-D.ietf-tsvwg-rfc5405bis].

3.3.2. Interface Description

[RFC0768] describes basic requirements for an API for UDP. Guidance on use of common APIs is provided in [I-D.ietf-tsvwg-rfc5405bis].

A UDP endpoint consists of a tuple of (IP address, port number). De-multiplexing using multiple abstract endpoints (sockets) on the same IP address is supported. The same socket may be used by a single server to interact with multiple clients (note: this behavior differs from TCP, which uses a pair of tuples to identify a connection). Multiple server instances (processes) that bind to the same socket can cooperate to service multiple clients. The socket implementation arranges to not duplicate the same received unicast message to multiple server processes.

Many operating systems also allow a UDP socket to be "connected", i.e., to bind a UDP socket to a specific (remote) UDP endpoint. Unlike TCP’s connect primitive, 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 [I-D.ietf-tsvwg-rfc5405bis].

3.3.3. Transport Features

The transport features provided by UDP are:

- unicast, multicast, anycast, or IPv4 broadcast transport,
- port multiplexing (where a receiving port can be configured to receive datagrams from multiple senders),
- message-oriented delivery,
- uni- or bidirectional communication where the transmissions in each direction are independent,
- non-reliable delivery,
- unordered delivery,
3.4. Lightweight User Datagram Protocol (UDP-Lite)

The Lightweight User Datagram Protocol (UDP-Lite) [RFC3828] is an IETF standards track transport protocol. It provides a unidirectional, datagram protocol that preserves message boundaries. IETF guidance on the use of UDP-Lite is provided in [I-D.ietf-tsvwg-rfc5405bis]. A UDP-Lite service may support IPv4 broadcast, multicast, anycast and unicast, and IPv6 multicast, anycast and unicast.

Examples of use include a class of applications that can derive benefit from having partially-damaged payloads delivered, rather than discarded. One use is to provider header integrity checks but allow delivery of corrupted payloads to error-tolerant applications, or when payload integrity is provided by some other mechanism (see [RFC6936]).

3.4.1. Protocol Description

Like UDP, UDP-Lite is a connection-less datagram protocol, with no connection setup or feature negotiation. It changes the semantics of the UDP "payload length" field to that of a "checksum coverage length" field, and is identified by a different IP protocol/next-header value. The "checksum coverage length" field specifies the intended checksum coverage, with the remaining unprotected part of the payload called the "error-insensitive part". Applications using UDP-Lite therefore cannot make assumptions regarding the correctness of the data received in the insensitive part of the UDP-Lite payload.

Otherwise, UDP-Lite is semantically identical to UDP. In the same way as for UDP, mechanisms for receiver flow control, congestion control, PMTU or PLPMTU discovery, support for ECN, etc. needs to be provided by upper layer protocols [I-D.ietf-tsvwg-rfc5405bis].

3.4.2. Interface Description

There is no API currently specified in the RFC Series, but guidance on use of common APIs is provided in [I-D.ietf-tsvwg-rfc5405bis].

The interface of UDP-Lite differs from that of UDP by the addition of a single (socket) option that communicates a checksum coverage length value. The checksum coverage may also be made visible to the application via the UDP-Lite MIB module [RFC5097].
3.4.3. Transport Features

The transport features provided by UDP-Lite are:

- unicast, multicast, anycast, or IPv4 broadcast transport (as for UDP),
- port multiplexing (as for UDP),
- message-oriented delivery (as for UDP),
- Uni- or bidirectional communication where the transmissions in each direction are independent (as for UDP),
- non-reliable delivery (as for UDP),
- non-ordered delivery (as for UDP),
- partial or full payload error detection (where the checksum coverage field indicates the size of the payload data covered by the checksum).

3.5. Stream Control Transmission Protocol (SCTP)

SCTP is a message-oriented IETF standards track transport protocol. The base protocol is specified in [RFC4960]. It supports multi-homing and path failover to provide resilience to path failures. An SCTP association has multiple streams in each direction, providing in-sequence delivery of user messages within each stream. This allows it to minimize head of line blocking. SCTP supports multiple stream scheduling schemes controlling stream multiplexing, including priority and fair weighting schemes.

SCTP was originally developed for transporting telephony signaling messages and is deployed in telephony signaling networks, especially in mobile telephony networks. It can also be used for other services, for example, in the WebRTC framework for data channels.

3.5.1. Protocol Description

SCTP is a connection-oriented protocol using a four way handshake to establish an SCTP association, and a three way message exchange to gracefully shut it down. It uses the same port number concept as DCCP, TCP, UDP, and UDP-Lite. SCTP only supports unicast.

SCTP uses the 32-bit CRC32c for protecting SCTP packets against bit errors and misdelivery of packets to an unintended endpoint. This is stronger than the 16-bit checksums used by TCP or UDP. However,
partial payload checksum coverage as provided by DCCP or UDP-Lite is not supported.

SCTP has been designed with extensibility in mind. A common header is followed by a sequence of chunks. [RFC4960] defines how a receiver processes chunks with an unknown chunk type. The support of extensions can be negotiated during the SCTP handshake. Currently defined extensions include mechanisms for dynamic re-configuration of streams [RFC6525] and IP addresses [RFC5061]. Furthermore, the extension specified in [RFC3758] introduces the concept of partial reliability for user messages.

SCTP provides a message-oriented service. Multiple small user messages can be bundled into a single SCTP packet to improve efficiency. For example, this bundling may be done by delaying user messages at the sender, similar to Nagle’s algorithm used by TCP. User messages which would result in IP packets larger than the MTU will be fragmented at the sender and reassembled at the receiver. There is no protocol limit on the user message size. For MTU discovery the same mechanism than for TCP can be used [RFC1981][RFC4821], as well as utilizing probe packets with padding chunks, as defined in [RFC4820].

[RFC4960] specifies TCP-friendly congestion control to protect the network against overload. SCTP also uses sliding window flow control to protect receivers against overflow. Similar to TCP, SCTP also supports delaying acknowledgments. [RFC7053] provides a way for the sender of user messages to request the immediate sending of the corresponding acknowledgments.

Each SCTP association has between 1 and 65536 uni-directional streams in each direction. The number of streams can be different in each direction. Every user message is sent on a particular stream. User messages can be sent un-ordered, or ordered upon request by the upper layer. Un-ordered messages can be delivered as soon as they are completely received. For user messages not requiring fragmentation, this minimizes head of line blocking. On the other hand, ordered messages sent on the same stream are delivered at the receiver in the same order as sent by the sender.

The base protocol defined in [RFC4960] does not allow interleaving of user messages. Large messages on one stream can therefore block the sending of user messages on other streams. [I-D.ietf-tsvwg-sctp-ndata] overcomes this limitation. This draft also specifies multiple algorithms for the sender side selection of which streams to send data from, supporting a variety of scheduling algorithms including priority based methods. The stream re-configuration extension defined in [RFC6525] allows streams to be
reset during the lifetime of an association and to increase the number of streams, if the number of streams negotiated in the SCTP handshake becomes insufficient.

Each user message sent is either delivered to the receiver or, in case of excessive retransmissions, the association is terminated in a non-graceful way [RFC4960], similar to TCP behavior. In addition to this reliable transfer, the partial reliability extension [RFC3758] allows a sender to abandon user messages. The application can specify the policy for abandoning user messages.

SCTP supports multi-homing. Each SCTP endpoint uses a list of IP-addresses and a single port number. These addresses can be any mixture of IPv4 and IPv6 addresses. These addresses are negotiated during the handshake and the address re-configuration extension specified in [RFC5061] in combination with [RFC4895] can be used to change these addresses in an authenticated way during the lifetime of an SCTP association. This allows for transport layer mobility. Multiple addresses are used for improved resilience. If a remote address becomes unreachable, the traffic is switched over to a reachable one, if one exists.

For securing user messages, the use of TLS over SCTP has been specified in [RFC3436]. However, this solution does not support all services provided by SCTP, such as un-ordered delivery or partial reliability. Therefore, the use of DTLS over SCTP has been specified in [RFC6083] to overcome these limitations. When using DTLS over SCTP, the application can use almost all services provided by SCTP.

[I-D.ietf-tsvwg-natsupp] defines methods for endpoints and middleboxes to provide NAT traversal for SCTP over IPv4. For legacy NAT traversal, [RFC6951] defines the UDP encapsulation of SCTP-packets. Alternatively, SCTP packets can be encapsulated in DTLS packets as specified in [I-D.ietf-tsvwg-sctp-dtls-encaps]. The latter encapsulation is used within the WebRTC [I-D.ietf-rtcweb-transports] context.

An SCTP ABORT chunk may be used to force a SCTP endpoint to close a session [RFC4960], aborting the connection.

SCTP has a well-defined API, described in the next subsection.

3.5.2. Interface Description

[RFC4960] defines an abstract API for the base protocol. This API describes the following functions callable by the upper layer of SCTP: Initialize, Associate, Send, Receive, Receive Unsent Message, Receive Unacknowledged Message, Shutdown, Abort, SetPrimary, Status,
Change Heartbeat, Request Heartbeat, Get SRTT Report, Set Failure Threshold, Set Protocol Parameters, and Destroy. The following notifications are provided by the SCTP stack to the upper layer: COMMUNICATION UP, DATA ARRIVE, SHUTDOWN COMPLETE, COMMUNICATION LOST, COMMUNICATION ERROR, RESTART, SEND FAILURE, NETWORK STATUS CHANGE.

An extension to the BSD Sockets API is defined in [RFC6458] and covers:

- the base protocol defined in [RFC4960]. The API allows control over local addresses and port numbers and the primary path. Furthermore the application has fine control about parameters like retransmission thresholds, the path supervision parameters, the delayed acknowledgment timeout, and the fragmentation point. The API provides a mechanism to allow the SCTP stack to notify the application about events if the application has requested them. These notifications provide information about status changes of the association and each of the peer addresses. In case of send failures, including drop of messages sent unreliably, the application can also be notified and user messages can be returned to the application. When sending user messages, the stream id, a payload protocol identifier, an indication whether ordered delivery is requested or not. These parameters can also be provided on message reception. Additionally a context can be provided when sending, which can be used in case of send failures. The sending of arbitrary large user messages is supported.

- the SCTP Partial Reliability extension defined in [RFC3758] to specify for a user message the PR-SCTP policy and the policy specific parameter. Examples of these policies defined in [RFC3758] and [RFC7496] are:

  - Limiting the time a user message is dealt with by the sender.
  - Limiting the number of retransmissions for each fragment of a user message. If the number of retransmissions is limited to 0, one gets a service similar to UDP.
  - Abandoning messages of lower priority in case of a send buffer shortage.

- the SCTP Authentication extension defined in [RFC4895] allowing to manage the shared keys, the HMAC to use, set the chunk types which are only accepted in an authenticated way, and get the list of chunks which are accepted by the local and remote end point in an authenticated way.
the SCTP Dynamic Address Reconfiguration extension defined in [RFC5061]. It allows to manually add and delete local addresses for SCTP associations and the enabling of automatic address addition and deletion. Furthermore the peer can be given a hint for choosing its primary path.

A BSD Sockets API extension has been defined in the documents that specify the following SCTP protocol extensions:

- the SCTP Stream Reconfiguration extension defined in [RFC6525]. The API allows to trigger the reset operation for incoming and outgoing streams and the whole association. It provides also a way to notify the association about the corresponding events. Furthermore the application can increase the number of streams.

- the UDP Encapsulation of SCTP packets extension defined in [RFC6951]. The API allows the management of the remote UDP encapsulation port.

- the SCTP SACK-IMMEDIATELY extension defined in [RFC7053]. The API allows the sender of a user message to request the receiver to send the corresponding acknowledgment immediately.

- the additional PR-SCTP policies defined in [RFC7496]. The API allows to enable/disable the PR-SCTP extension, choose the PR-SCTP policies defined in the document and provide statistical information about abandoned messages.

Future documents describing SCTP protocol extensions are expected to describe the corresponding BSD Sockets API extension in a "Socket API Considerations" section.

The SCTP socket API supports two kinds of sockets:

- one-to-one style sockets (by using the socket type "SOCK_STREAM").

- one-to-many style socket (by using the socket type "SOCK_SEQPACKET").

One-to-one style sockets are similar to TCP sockets, there is a 1:1 relationship between the sockets and the SCTP associations (except for listening sockets). One-to-many style SCTP sockets are similar to unconnected UDP sockets, where there is a 1:n relationship between the sockets and the SCTP associations.

The SCTP stack can provide information to the applications about state changes of the individual paths and the association whenever
they occur. These events are delivered similar to user messages but are specifically marked as notifications.

New functions have been introduced to support the use of multiple local and remote addresses. Additional SCTP-specific send and receive calls have been defined to permit SCTP-specific information to be sent without using ancillary data in the form of additional cmsgs. These functions provide support for detecting partial delivery of user messages and notifications.

The SCTP socket API allows a fine-grained control of the protocol behavior through an extensive set of socket options.

The SCTP kernel implementations of FreeBSD, Linux and Solaris follow mostly the specified extension to the BSD Sockets API for the base protocol and the corresponding supported protocol extensions.

3.5.3. Transport Features

The transport features provided by SCTP are:

- connection-oriented transport with feature negotiation and application-to-port mapping,
- unicast transport,
- port multiplexing,
- uni- or bidirectional communication,
- message-oriented delivery with durable message framing supporting multiple concurrent streams,
- fully reliable, partially reliable, or unreliable delivery (based on user specified policy to handle abandoned user messages) with drop notification,
- ordered and unordered delivery within a stream,
- support for stream scheduling prioritization,
- segmentation,
- user message bundling,
- flow control using a window-based mechanism,
- congestion control using methods similar to TCP,
3.6. Datagram Congestion Control Protocol (DCCP)

Datagram Congestion Control Protocol (DCCP) [RFC4340] is an IETF standards track bidirectional transport protocol that provides unicast connections of congestion-controlled messages without providing reliability.

The DCCP Problem Statement describes the goals that DCCP sought to address [RFC4336]: It is suitable for applications that transfer fairly large amounts of data and that can benefit from control over the trade off between timeliness and reliability [RFC4336].

DCCP offers low overhead, and many characteristics common to UDP, but can avoid "re-inventing the wheel" each time a new multimedia application emerges. Specifically it includes core transport functions (feature negotiation, path state management, RTT calculation, PMTUD, etc.): DCCP applications select how they send packets and, where suitable, choose common algorithms to manage their functions. Examples of applications that can benefit from such transport services include interactive applications, streaming media, or on-line games [RFC4336].

3.6.1. Protocol Description

DCCP is a connection-oriented datagram protocol, providing a three-way handshake to allow a client and server to set up a connection, and mechanisms for orderly completion and immediate teardown of a connection.

A DCCP protocol instance can be extended [RFC4340] and tuned using additional features. Some features are sender-side only, requiring no negotiation with the receiver; some are receiver-side only; and some are explicitly negotiated during connection setup.

DCCP uses a Connect packet to initiate a session, and permits each endpoint to choose the features it wishes to support. Simultaneous open [RFC5596], as in TCP, can enable interoperability in the presence of middleboxes. The Connect packet includes a Service Code [RFC5595] that identifies the application or protocol using DCCP, providing middleboxes with information about the intended use of a connection.

The DCCP service is unicast-only.
It provides multiplexing to multiple sockets at each endpoint using port numbers. An active DCCP session is identified by its four-tuple of local and remote IP addresses and local port and remote port numbers.

The protocol segments data into messages, typically sized to fit in IP packets, but which may be fragmented providing they are smaller than the maximum packet size. A DCCP interface allows applications to request fragmentation for packets larger than PMTU, but not larger than the maximum packet size allowed by the current congestion control mechanism (CCMPS) [RFC4340].

Each message is identified by a sequence number. The sequence number is used to identify segments in acknowledgments, to detect unacknowledged segments, to measure RTT, etc. The protocol may support unordered delivery of data, and does not itself provide retransmission. DCCP supports reduced checksum coverage, a partial payload protection mechanism similar to UDP-Lite. There is also a Data Checksum option, which when enabled, contains a strong CRC, to enable endpoints to detect application data corruption.

Receiver flow control is supported, which limits the amount of unacknowledged data that can be outstanding at a given time.

A DCCP Reset packet may be used to force a DCCP endpoint to close a session [RFC4340], aborting the connection.

DCCP supports negotiation of the congestion control profile between endpoints, to provide plug-and-play congestion control mechanisms. Examples of specified profiles include "TCP-like" [RFC4341], "TCP-friendly" [RFC4342], and "TCP-friendly for small packets" [RFC5622]. Additional mechanisms are recorded in an IANA registry.

A lightweight UDP-based encapsulation (DCCP-UDP) has been defined [RFC6773] that permits DCCP to be used over paths where DCCP is not natively supported. Support for DCCP in NAPT/NATs is defined in [RFC4340] and [RFC5595]. Upper layer protocols specified on top of DCCP include DTLS [RFC5595], RTP [RFC5672], ICE/SDP [RFC6773].

3.6.2. Interface Description

Functions expected for a DCCP API include: Open, Close and Management of the progress a DCCP connection. The Open function provides feature negotiation, selection of an appropriate CCID for congestion control and other parameters associated with the DCCP connection. A function allows an application to send DCCP datagrams, including setting the required checksum coverage, and any required options. (DCCP permits sending datagrams with a zero-length payload.)
function allows reception of data, including indicating if the data was used or dropped. Functions can also make the status of a connection visible to an application, including detection of the maximum packet size and the ability to perform flow control by detecting a slow receiver at the sender.

There is no API currently specified in the RFC Series.

3.6.3. Transport Features

The transport features provided by DCCP are:

- unicast transport,
- connection-oriented communication with feature negotiation and application-to-port mapping,
- signaling of application class for middlebox support (implemented using Service Codes),
- port multiplexing,
- uni-or bidirectional communication,
- message-oriented delivery,
- unreliable delivery with drop notification,
- unordered delivery,
- flow control (implemented using the slow receiver function)
- partial and full payload error detection (with optional strong integrity check).

3.7. Transport Layer Security (TLS) and Datagram TLS (DTLS) as a pseudotransport

Transport Layer Security (TLS) [RFC5246] and Datagram TLS (DTLS) [RFC6347] are IETF protocols that provide several security-related features to applications. TLS is designed to run on top of a reliable streaming transport protocol (usually TCP), while DTLS is designed to run on top of a best-effort datagram protocol (UDP or DCCP [RFC5238]). At the time of writing, the current version of TLS is 1.2, defined in [RFC5246]; work on TLS version 1.3 [I-D.ietf-tls-tls13] nearing completion. DTLS provides nearly identical functionality to applications; it is defined in [RFC6347] and its current version is also 1.2. The TLS protocol evolved from
the Secure Sockets Layer (SSL) [RFC6101] protocols developed in the mid-1990s to support protection of HTTP traffic.

While older versions of TLS and DTLS are still in use, they provide weaker security guarantees. [RFC7457] outlines important attacks on TLS and DTLS. [RFC7525] is a Best Current Practices (BCP) document that describes secure configurations for TLS and DTLS to counter these attacks. The recommendations are applicable for the vast majority of use cases.

3.7.1. Protocol Description

Both TLS and DTLS provide the same security features and can thus be discussed together. The features they provide are:

- Confidentiality
- Data integrity
- Peer authentication (optional)
- Perfect forward secrecy (optional)

The authentication of the peer entity can be omitted; a common web use case is where the server is authenticated and the client is not. TLS also provides a completely anonymous operation mode in which neither peer’s identity is authenticated. It is important to note that TLS itself does not specify how a peering entity’s identity should be interpreted. For example, in the common use case of authentication by means of an X.509 certificate, it is the application’s decision whether the certificate of the peering entity is acceptable for authorization decisions.

Perfect forward secrecy, if enabled and supported by the selected algorithms, ensures that traffic encrypted and captured during a session at time $t_0$ cannot be later decrypted at time $t_1$ ($t_1 > t_0$), even if the long-term secrets of the communicating peers are later compromised.

As DTLS is generally used over an unreliable datagram transport such as UDP, applications will need to tolerate lost, re-ordered, or duplicated datagrams. Like TLS, DTLS conveys application data in a sequence of independent records. However, because records are mapped to unreliable datagrams, there are several features unique to DTLS that are not applicable to TLS:

- Record replay detection (optional).
3.7.2. Interface Description

TLS is commonly invoked using an API provided by packages such as OpenSSL, wolfSSL, or GnuTLS. Using such APIs entails the manipulation of several important abstractions, which fall into the following categories: long-term keys and algorithms, session state, and communications/connections.

Considerable care is required in the use of TLS APIs to ensure creation of a secure application. The programmer should have at least a basic understanding of encryption and digital signature algorithms and their strengths, public key infrastructure (including X.509 certificates and certificate revocation), and the sockets API. See [RFC7525] and [RFC7457], as mentioned above.

As an example, in the case of OpenSSL, the primary abstractions are the library itself and method (protocol), session, context, cipher and connection. After initializing the library and setting the method, a cipher suite is chosen and used to configure a context object. Session objects may then be minted according to the parameters present in a context object and associated with individual connections. Depending on how precisely the programmer wishes to select different algorithmic or protocol options, various levels of details may be required.

3.7.3. Transport Features

Both TLS and DTLS employ a layered architecture. The lower layer is commonly called the record protocol. It is responsible for:

- message fragmentation,
authentication and integrity via message authentication codes (MAC),

data encryption,

scheduling transmission using the underlying transport protocol.

DTLS augments the TLS record protocol with:

ordering and replay protection, implemented using sequence numbers.

Several protocols are layered on top of the record protocol. These include the handshake, alert, and change cipher spec protocols. There is also the data protocol, used to carry application traffic. The handshake protocol is used to establish cryptographic and compression parameters when a connection is first set up. In DTLS, this protocol also has a basic fragmentation and retransmission capability and a cookie-like mechanism to resist DoS attacks. (TLS compression is not recommended at present). The alert protocol is used to inform the peer of various conditions, most of which are terminal for the connection. The change cipher spec protocol is used to synchronize changes in cryptographic parameters for each peer.

The data protocol, when used with an appropriate cipher, provides:

authentication of one end or both ends of a connection,

confidentiality,

cryptographic integrity protection.

Both TLS and DTLS are unicast-only.

3.8. Realtime Transport Protocol (RTP)

RTP provides an end-to-end network transport service, suitable for applications transmitting real-time data, such as audio, video or data, over multicast or unicast transport services, including TCP, UDP, UDP-Lite, DCCP, TLS and DTLS.

3.8.1. Protocol Description

The RTP standard [RFC3550] defines a pair of protocols, RTP and the RTP control protocol, RTCP. The transport does not provide connection setup, instead relying on out-of-band techniques or associated control protocols to setup, negotiate parameters or tear down a session.
An RTP sender encapsulates audio/video data into RTP packets to transport media streams. The RFC-series specifies RTP payload formats that allow packets to carry a wide range of media, and specifies a wide range of multiplexing, error control and other support mechanisms.

If a frame of media data is large, it will be fragmented into several RTP packets. Likewise, several small frames may be bundled into a single RTP packet.

An RTP receiver collects RTP packets from the network, validates them for correctness, and sends them to the media decoder input-queue. Missing packet detection is performed by the channel decoder. The play-out buffer is ordered by time stamp and is used to reorder packets. Damaged frames may be repaired before the media payloads are decompressed to display or store the data. Some uses of RTP are able to exploit the partial payload protection features offered by DCCP and UDP-Lite.

RTCP is a control protocol that works alongside an RTP flow. Both the RTP sender and receiver will send RTCP report packets. This is used to periodically send control information and report performance. Based on received RTCP feedback, an RTP sender can adjust the transmission, e.g., perform rate adaptation at the application layer in the case of congestion.

An RTCP receiver report (RTCP RR) is returned to the sender periodically to report key parameters (e.g., the fraction of packets lost in the last reporting interval, the cumulative number of packets lost, the highest sequence number received, and the inter-arrival jitter). The RTCP RR packets also contain timing information that allows the sender to estimate the network round trip time (RTT) to the receivers.

The interval between reports sent from each receiver tends to be on the order of a few seconds on average, although this varies with the session rate, and sub-second reporting intervals are possible for high rate sessions. The interval is randomized to avoid synchronization of reports from multiple receivers.

3.8.2. Interface Description

There is no standard application programming interface defined for RTP or RTCP. Implementations are typically tightly integrated with a particular application, and closely follow the principles of application level framing and integrated layer processing [ClarkArch] in media processing [RFC2736], error recovery and concealment, rate adaptation, and security [RFC7202]. Accordingly, RTP implementations
tend to be targeted at particular application domains (e.g., voice-over-IP, IPTV, or video conferencing), with a feature set optimized for that domain, rather than being general purpose implementations of the protocol.

3.8.3. Transport Features

The transport features provided by RTP are:

- unicast, multicast or IPv4 broadcast (provided by lower layer protocol),
- port multiplexing (provided by lower layer protocol),
- uni- or bidirectional communication (provided by lower layer protocol),
- message-oriented delivery with support for media types and other extensions,
- reliable delivery when using erasure coding or unreliable delivery with drop notification (if supported by lower layer protocol),
- connection setup with feature negotiation (using associated protocols) and application-to-port mapping (provided by lower layer protocol),
- segmentation,
- performance metric reporting (using associated protocols).

3.9. Hypertext Transport Protocol (HTTP) over TCP as a pseudotransport

The Hypertext Transfer Protocol (HTTP) is an application-level protocol widely used on the Internet. It provides object-oriented delivery of discrete data or files. Version 1.1 of the protocol is specified in [RFC7230] [RFC7231] [RFC7232] [RFC7233] [RFC7234] [RFC7235], and version 2 in [RFC7540]. HTTP is usually transported over TCP using port 80 and 443, although it can be used with other transports. When used over TCP it inherits TCP’s properties.

Application layer protocols may use HTTP as a substrate with an existing method and data formats, or specify new methods and data formats. There are various reasons for this practice listed in [RFC3205]; these include being a well-known and well-understood protocol, reusability of existing servers and client libraries, easy use of existing security mechanisms such as HTTP digest authentication [RFC2617] and TLS [RFC5246], the ability of HTTP to
traverse firewalls makes it work over many types of infrastructure, and in cases where an application server often needs to support HTTP anyway.

Depending on application need, the use of HTTP as a substrate protocol may add complexity and overhead in comparison to a special-purpose protocol (e.g., HTTP headers, suitability of the HTTP security model, etc.). [RFC3205] addresses this issue and provides some guidelines and identifies concerns about the use of HTTP standard port 80 and 443, the use of HTTP URL scheme and interaction with existing firewalls, proxies and NATs.

Representational State Transfer (REST) [REST] is another example of how applications can use HTTP as transport protocol. REST is an architecture style that may be used to build applications using HTTP as a communication protocol.

3.9.1. Protocol Description

Hypertext Transfer Protocol (HTTP) is a request/response protocol. A client sends a request containing a request method, URI and protocol version followed message whose design is inspired by MIME (see [RFC7231] for the differences between an HTTP object and a MIME message), containing information about the client and request modifiers. The message can also contain a message body carrying application data. The server responds with a status or error code followed by a message containing information about the server and information about the data. This may include a message body. It is possible to specify a data format for the message body using MIME media types [RFC2045]. The protocol has additional features, some relevant to pseudo-transport are described below.

Content negotiation, specified in [RFC7231], is a mechanism provided by HTTP to allow selection of a representation for a requested resource. The client and server negotiate acceptable data formats, character sets, data encoding (e.g., data can be transferred compressed using gzip). HTTP can accommodate exchange of messages as well as data streaming (using chunked transfer encoding [RFC7230]). It is also possible to request a part of a resource using an object range request [RFC7233]. The protocol provides powerful cache control signaling defined in [RFC7234].

The persistent connections of HTTP 1.1 and HTTP 2.0 allow multiple request-response transactions (streams) during the life-time of a single HTTP connection. This reduces overhead during connection establishment and mitigates transport layer slow-start that would have otherwise been incurred for each transaction. HTTP 2.0 connections can multiplex many request/response pairs in parallel on
a single transport connection. Both are important to reduce latency for HTTP’s primary use case.

HTTP can be combined with security mechanisms, such as TLS (denoted by HTTPS). This adds protocol properties provided by such a mechanism (e.g., authentication, encryption). The TLS Application-Layer Protocol Negotiation (ALPN) extension [RFC7301] can be used to negotiate the HTTP version within the TLS handshake, eliminating the latency incurred by additional round-trip exchanges. Arbitrary cookie strings, included as part of the request headers, are often used as bearer tokens in HTTP.

3.9.2. Interface Description

There are many HTTP libraries available exposing different APIs. The APIs provide a way to specify a request by providing a URI, a method, request modifiers and optionally a request body. For the response, callbacks can be registered that will be invoked when the response is received. If HTTPS is used, the API exposes a registration of callbacks for a server that requests client authentication and when certificate verification is needed.

The World Wide Web Consortium (W3C) has standardized the XMLHttpRequest API [XHR]. This API can be used for sending HTTP/HTTPS requests and receiving server responses. Besides the XML data format, the request and response data format can also be JSON, HTML, and plain text. JavaScript and XMLHttpRequest are ubiquitous programming models for websites, and more general applications, where native code is less attractive.

3.9.3. Transport features

The transport features provided by HTTP, when used as a pseudo-transport, are:

- unicast transport (provided by the lower layer protocol, usually TCP),
- uni- or bidirectional communication,
- transfer of objects in multiple streams with object content type negotiation, supporting partial transmission of object ranges,
- ordered delivery (provided by the lower layer protocol, usually TCP),
- fully reliable delivery (provided by the lower layer protocol, usually TCP),
flow control (provided by the lower layer protocol, usually TCP).
congestion control (provided by the lower layer protocol, usually TCP).
HTTPS (HTTP over TLS) additionally provides the following features (as provided by TLS):
authentication (of one or both ends of a connection),
confidentiality,
integrity protection.

3.10. File Delivery over Unidirectional Transport/Asynchronous Layered Coding Reliable Multicast (FLUTE/ALC)

FLUTE/ALC is an IETF standards track protocol specified in [RFC6726] and [RFC5775]. It provides object-oriented delivery of discrete data or files. Asynchronous Layer Coding (ALC) provides an underlying reliable transport service and FLUTE a file-oriented specialization of the ALC service (e.g., to carry associated metadata). The [RFC6726] and [RFC5775] protocols are non-backward-compatible updates of the [RFC3926] and [RFC3450] experimental protocols; these experimental protocols are currently largely deployed in the 3GPP Multimedia Broadcast and Multicast Services (MBMS) (see [MBMS], section 7) and similar contexts (e.g., the Japanese ISDB-Tmm standard).

The FLUTE/ALC protocol has been designed to support massively scalable reliable bulk data dissemination to receiver groups of arbitrary size using IP Multicast over any type of delivery network, including unidirectional networks (e.g., broadcast wireless channels). However, the FLUTE/ALC protocol also supports point-to-point unicast transmissions.

FLUTE/ALC bulk data dissemination has been designed for discrete file or memory-based "objects". Although FLUTE/ALC is not well adapted to byte- and message-streaming, there is an exception: FLUTE/ALC is used to carry 3GPP Dynamic Adaptive Streaming over HTTP (DASH) when scalability is a requirement (see [MBMS], section 5.6).

FLUTE/ALC’s reliability, delivery mode, congestion control, and flow/rate control mechanisms can be separately controlled to meet different application needs. Section 4.1 of [I-D.ietf-tsvwg-rfc5405bis] describes multicast congestion control requirements for UDP.
3.10.1. Protocol Description

The FLUTE/ALC protocol works on top of UDP (though it could work on top of any datagram delivery transport protocol), without requiring any connectivity from receivers to the sender. Purely unidirectional networks are therefore supported by FLUTE/ALC. This guarantees scalability to an unlimited number of receivers in a session, since the sender behaves exactly the same regardless of the number of receivers.

FLUTE/ALC supports the transfer of bulk objects such as file or in-memory content, using either a push or an on-demand mode. In push mode, content is sent once to the receivers, while in on-demand mode, content is sent continuously during periods of time that can greatly exceed the average time required to download the session objects (see [RFC5651], section 4.2).

This enables receivers to join a session asynchronously, at their own discretion, receive the content and leave the session. In this case, data content is typically sent continuously, in loops (also known as "carousels"). FLUTE/ALC also supports the transfer of an object stream, with loose real-time constraints. This is particularly useful to carry 3GPP DASH when scalability is a requirement and unicast transmissions over HTTP cannot be used ([MBMS], section 5.6). In this case, packets are sent in sequence using push mode. FLUTE/ALC is not well adapted to byte- and message-streaming and other solutions could be preferred (e.g., FECFRAME [RFC6363] with real-time flows).

The FLUTE file delivery instantiation of ALC provides a metadata delivery service. Each object of the FLUTE/ALC session is described in a dedicated entry of a File Delivery Table (FDT), using an XML format (see [RFC6726], section 3.2). This metadata can include, but is not restricted to, a URI attribute (to identify and locate the object), a media type attribute, a size attribute, an encoding attribute, or a message digest attribute. Since the set of objects sent within a session can be dynamic, with new objects being added and old ones removed, several instances of the FDT can be sent and a mechanism is provided to identify a new FDT Instance.

Error detection and verification of the protocol control information relies on the on the underlying transport (e.g., UDP checksum).

To provide robustness against packet loss and improve the efficiency of the on-demand mode, FLUTE/ALC relies on packet erasure coding (AL-FEC). AL-FEC encoding is proactive (since there is no feedback and therefore no (N)ACK-based retransmission) and ALC packets containing repair data are sent along with ALC packets containing source data.
Several FEC Schemes have been standardized; FLUTE/ALC does not mandate the use of any particular one. Several strategies concerning the transmission order of ALC source and repair packets are possible, in particular in on-demand mode where it can deeply impact the service provided (e.g., to favor the recovery of objects in sequence, or at the other extreme, to favor the recovery of all objects in parallel), and FLUTE/ALC does not mandate nor recommend the use of any particular one.

A FLUTE/ALC session is composed of one or more channels, associated to different destination unicast and/or multicast IP addresses. ALC packets are sent in those channels at a certain transmission rate, with a rate that often differs depending on the channel. FLUTE/ALC does not mandate nor recommend any strategy to select which ALC packet to send on which channel. FLUTE/ALC can use a multiple rate congestion control building block (e.g., WEBRC) to provide congestion control that is feedback free, where receivers adjust their reception rates individually by joining and leaving channels associated with the session. To that purpose, the ALC header provides a specific field to carry congestion control specific information. However, FLUTE/ALC does not mandate the use of a particular congestion control mechanism although WEBRC is mandatory to support for the Internet ([RFC6726], section 1.1.4). FLUTE/ALC is often used over a network path with pre-provisioned capacity [I-D.ietf-tsvwg-rfc5405bis] where there are no flows competing for capacity. In this case, a sender-based rate control mechanism and a single channel is sufficient.

[RFC6584] provides per-packet authentication, integrity, and anti-replay protection in the context of the ALC and NORM protocols. Several mechanisms are proposed that seamlessly integrate into these protocols using the ALC and NORM header extension mechanisms.

3.10.2. Interface Description

The FLUTE/ALC specification does not describe a specific application programming interface (API) to control protocol operation. Although open source and commercial implementations have specified APIs, there is no IETF-specified API for FLUTE/ALC.

3.10.3. Transport Features

The transport features provided by FLUTE/ALC are:

- unicast, multicast, anycast or IPv4 broadcast transmission,

- object-oriented delivery of discrete data or files and associated metadata,
- fully reliable or partially reliable delivery (of file or in-memory objects), using proactive packet erasure coding (AL-FEC) to recover from packet erasures,
- ordered or unordered delivery (of file or in-memory objects),
- error detection (based on the UDP checksum),
- per-packet authentication,
- per-packet integrity,
- per-packet replay protection,
- congestion control for layered flows (e.g., with WEBRC).

3.11. NACK-Oriented Reliable Multicast (NORM)

NORM is an IETF standards track protocol specified in [RFC5740]. It provides object-oriented delivery of discrete data or files.

The protocol was designed to support reliable bulk data dissemination to receiver groups using IP Multicast but also provides for point-to-point unicast operation. Support for bulk data dissemination includes discrete file or computer memory-based "objects" as well as byte- and message-streaming.

NORM can incorporate packet erasure coding as a part of its selective ARQ in response to negative acknowledgments from the receiver. The packet erasure coding can also be proactively applied for forward protection from packet loss. NORM transmissions are governed by TCP-friendly multicast congestion control (TFMCC, [RFC4654]). The reliability, congestion control and flow control mechanisms can be separately controlled to meet different application needs.

3.11.1. Protocol Description

The NORM protocol is encapsulated in UDP datagrams and thus provides multiplexing for multiple sockets on hosts using port numbers. For loosely coordinated IP Multicast, NORM is not strictly connection-oriented although per-sender state is maintained by receivers for protocol operation. [RFC5740] does not specify a handshake protocol for connection establishment. Separate session initiation can be used to coordinate port numbers. However, in-band "client-server" style connection establishment can be accomplished with the NORM congestion control signaling messages using port binding techniques like those for TCP client-server connections.
NORM supports bulk "objects" such as file or in-memory content but also can treat a stream of data as a logical bulk object for purposes of packet erasure coding. In the case of stream transport, NORM can support either byte streams or message streams where application-defined message boundary information is carried in the NORM protocol messages. This allows the receiver(s) to join/re-join and recover message boundaries mid-stream as needed. Application content is carried and identified by the NORM protocol with encoding symbol identifiers depending upon the Forward Error Correction (FEC) Scheme [RFC3452] configured. NORM uses NACK-based selective ARQ to reliably deliver the application content to the receiver(s). NORM proactively measures round-trip timing information to scale ARQ timers appropriately and to support congestion control. For multicast operation, timer-based feedback suppression is used to achieve group size scaling with low feedback traffic levels. The feedback suppression is not applied for unicast operation.

NORM uses rate-based congestion control based upon the TCP-Friendly Rate Control (TFRC) [RFC4324] principles that are also used in DCCP [RFC4340]. NORM uses control messages to measure RTT and collect congestion event information (e.g., reflecting a loss event or ECN event) from the receiver(s) to support dynamic adjustment of the rate. The TCP-Friendly Multicast Congestion Control (TFMCC) [RFC4654] provides additional features to support multicast, but is functionally equivalent to TFRC for unicast.

Error detection and verification of the protocol control information relies on the underlying transport (e.g., UDP checksum).

The reliability mechanism is decoupled from congestion control. This allows invocation of alternative arrangements of transport services. For example, to support, fixed-rate reliable delivery or unreliable delivery (that may optionally be "better than best effort" via packet erasure coding) using TFRC. Alternative congestion control techniques may be applied. For example, TFRC rate control with congestion event detection based on ECN.

While NORM provides NACK-based reliability, it also supports a positive acknowledgment (ACK) mechanism that can be used for receiver flow control. This mechanism is decoupled from the reliability and congestion control, supporting applications with different needs. One example is use of NORM for quasi-reliable delivery, where timely delivery of newer content may be favored over completely reliable delivery of older content within buffering and RTT constraints.
3.11.2. Interface Description

The NORM specification does not describe a specific application programming interface (API) to control protocol operation. A freely-available, open source reference implementation of NORM is available at https://www.nrl.navy.mil/itd/ncs/products/norm, and a documented API is provided for this implementation. While a sockets-like API is not currently documented, the existing API supports the necessary functions for that to be implemented.

3.11.3. Transport Features

The transport features provided by NORM are:

- unicast or multicast transport,
- unidirectional communication,
- stream-oriented delivery in a single stream or object-oriented delivery of in-memory data or file bulk content objects,
- fully reliable (NACK-based) or partially reliable (using erasure coding both proactively and as part of ARQ) delivery,
- unordered delivery,
- error detection (relies on UDP checksum),
- segmentation,
- data bundling (using Nagle’s algorithm),
- flow control (timer-based and/or ack-based),
- congestion control (also supporting fixed rate reliable or unreliable delivery).

3.12. Internet Control Message Protocol (ICMP)

The Internet Control Message Protocol (ICMP) [RFC0792] for IPv4 and ICMP for IPv6 [RFC4443] are IETF standards track protocols. It is a connection-less unidirectional protocol that delivers individual messages, without error correction, congestion control, or flow control. Messages may be sent as unicast, IPv4 broadcast or multicast datagrams (IPv4 and IPv6), in addition to anycast datagrams.
While ICMP is not typically described as a transport protocol, it does position itself over the network layer, and the operation of other transport protocols can be closely linked to the functions provided by ICMP.

Transport Protocols and upper layer protocols can use received ICMP messages to help them take appropriate decisions when network or endpoint errors are reported. For example, to implement, ICMP-based Path MTU discovery [RFC1191][RFC1981] or assist in Packetization Layer Path MTU Discovery (PMTUD) [RFC4821]. Such reactions to received messages need to protect from off-path data injection [I-D.ietf-tsvwg-rfc5405bis], to avoid an application receiving packets created by an unauthorized third party. An application therefore needs to ensure that all messages are appropriately validated, by checking the payload of the messages to ensure these are received in response to actually transmitted traffic (e.g., a reported error condition that corresponds to a UDP datagram or TCP segment was actually sent by the application). This requires context [RFC6056], such as local state about communication instances to each destination (e.g., in the TCP, DCCP, or SCTP protocols). This state is not always maintained by UDP-based applications [I-D.ietf-tsvwg-rfc5405bis].

3.12.1. Protocol Description

ICMP is a connection-less unidirectional protocol, it delivers independent messages, called datagrams. Each message is required to carry a checksum as an integrity check and to protect from mis-delivery to an unintended endpoint.

ICMP messages typically relay diagnostic information from an endpoint [RFC1122] or network device [RFC1812] addressed to the sender of a flow. This usually contains the network protocol header of a packet that encountered a reported issue. Some formats of messages can also carry other payload data. Each message carries an integrity check calculated in the same way as for UDP, this checksum is not optional.

The RFC series defines additional IPv6 message formats to support a range of uses. In the case of IPv6 the protocol incorporates neighbor discovery [RFC2461] [RFC3971] (provided by ARP for IPv4) and the Multicast Listener Discovery (MLD) [RFC2710] group management functions (provided by IGMP for IPv4).

Reliable transmission can not be assumed. A receiving application that is unable to run sufficiently fast, or frequently, may miss messages since there is no flow or congestion control. In addition some network devices rate-limit ICMP messages.
3.12.2. Interface Description

ICMP processing is integrated in many connection-oriented transports, but like other functions needs to be provided by an upper-layer protocol when using UDP and UDP-Lite.

On some stacks, a bound socket also allows a UDP application to be notified when ICMP error messages are received for its transmissions [I-D.ietf-tsvwg-rfc5405bis].

Any response to ICMP error messages ought to be robust to temporary routing failures (sometimes called "soft errors"), e.g., transient ICMP "unreachable" messages ought to not normally cause a communication abort [RFC5461] [I-D.ietf-tsvwg-rfc5405bis].

3.12.3. Transport Features

ICMP does not provide any transport service directly to applications. Used together with other transport protocols, it provides transmission of control, error, and measurement data between endpoints, or from devices along the path to one endpoint.

4. Congestion Control

Congestion control is critical to the stable operation of the Internet. A variety of mechanisms are used to provide the congestion control needed by many Internet transport protocols. Congestion is detected based on sensing of network conditions, whether through explicit or implicit feedback. The congestion control mechanisms that can be applied by different transport protocols are largely orthogonal to the choice of transport protocol. This section provides an overview of the congestion control mechanisms available to the protocols described in Section 3.

Many protocols use a separate window to determine the maximum sending rate that is allowed by the congestion control. The used congestion control mechanism will increase the congestion window if feedback is received that indicates that the currently used network path is not congested, and will reduce the window otherwise. Window-based mechanisms often increase their window slowing over multiple RTTs, while decreasing strongly when the first indication of congestion is received. One example is an Additive Increase Multiplicative Decrease (AIMD) scheme, where the window is increased by a certain number of packets/bytes for each data segment that has been successfully transmitted, while the window decreases multiplicatively on the occurrence of a congestion event. This can lead to a rather unstable, oscillating sending rate, but will resolve a congestion situation quickly. TCP New Reno [RFC5681] which is one of the
initial proposed schemes for TCP as well as TCP Cubic
[I-D.ietf-tcpm-cubic] which is the default mechanism for TCP in Linux
are two examples for window-based AIMD schemes. This approach is
also used by DCCP CCID-2 for datagram congestion control.

Some classes of applications prefer to use a transport service that
allows sending at a more stable rate, that is slowly varied in
response to congestion. Rate-based methods offer this type of
congestion control and have been defined based on the loss ratio and
observed round trip time, such as TFRC [RFC5348] and TFRC-SP
[RFC4828]. These methods utilize a throughput equation to determine
the maximum acceptable rate. Such methods are used with DCCP CCID-3
[RFC4342] and CCID-4 [RFC5622], WEBRC [RFC3738], and other
applications.

Another class of applications prefer a transport service that yields
to other (higher-priority) traffic, such as interactive
transmissions. While most traffic in the Internet uses loss-based
congestion control and therefore tends to fill the network buffers
(to a certain level if Active Queue Management (AQM) is used), low-
priority congestion control methods often react to changes in delay
as an earlier indication of congestion. This approach tends to
induce less loss than a loss-based method but does generally not
compete well with loss-based traffic across shared bottleneck links.
Therefore, methods such as LEDBAT [RFC6824], are deployed in the
Internet for scavenger traffic that aim to only utilize otherwise
unused capacity.

5. Transport Features

The transport protocol features described in this document can be
used as a basis for defining common transport features, listed below
with the protocols supporting them:

- Control Functions
  - Addressing
    - unicast (TCP, MPTCP, UDP, UDP-Lite, SCTP, DCCP, TLS, RTP,
    HTTP, ICMP)
    - multicast (UDP, UDP-Lite, RTP, ICMP, FLUTE/ALC, NORM). Note
      that, as TLS and DTLS are unicast-only, there is no widely
      deployed mechanism for supporting the features in the
      Security section below when using multicast addressing.
    - IPv4 broadcast (UDP, UDP-Lite, ICMP)
+ anycast (UDP, UDP-Lite). Connection-oriented protocols such as TCP and DCCP have also been deployed using anycast addressing, with the risk that routing changes may cause connection failure.

* Association type
  + connection-oriented (TCP, MPTCP, DCCP, SCTP, TLS, RTP, HTTP, NORM)
  + connectionless (UDP, UDP-Lite, FLUTE/ALC)

* Multihoming support
  + resilience and mobility (MPTCP, SCTP)
  + load-balancing (MPTCP)
  + address family multiplexing (MPTCP, SCTP)

* Middlebox cooperation
  + application-class signaling to middleboxes (DCCP)
  + error condition signaling from middleboxes and routers to endpoints (ICMP)

* Signaling
  + control information and error signaling (ICMP)
  + application performance reporting (RTP)

  Delivery

* Reliability
  + fully reliable delivery (TCP, MPTCP, SCTP, TLS, HTTP, FLUTE/ALC, NORM)
  + partially reliable delivery (SCTP, NORM)
    - using packet erasure coding (RTP, FLUTE/ALC, NORM)
    - with specified policy for dropped messages (SCTP)
  + unreliable delivery (SCTP, UDP, UDP-Lite, DCCP, RTP)
- with drop notification to sender (SCTP, DCCP, RTP)
+ error detection
- checksum for error detection (TCP, MPTCP, UDP, UDP-Lite, SCTP, DCCP, TLS, DTLS, FLUTE/ALC, NORM, ICMP)
- partial payload checksum protection (UDP-Lite, DCCP). Some uses of RTP can exploit partial payload checksum protection feature to provide a corruption tolerant transport service.
- checksum optional (UDP). Possible with IPv4 and in certain cases with IPv6.

* Ordering
+ ordered delivery (TCP, MPTCP, SCTP, TLS, RTP, HTTP, FLUTE)
+ unordered delivery permitted (UDP, UDP-Lite, SCTP, DCCP, RTP, NORM)

* Type/framing
+ stream-oriented delivery (TCP, MPTCP, SCTP, TLS, HTTP)
  - with multiple streams per association (SCTP, HTTP2)
+ message-oriented delivery (UDP, UDP-Lite, SCTP, DCCP, DTLS, RTP)
+ object-oriented delivery of discrete data or files and associated metadata (HTTP, FLUTE/ALC, NORM)
  - with partial delivery of object ranges (HTTP)

* Directionality
+ unidirectional (UDP, UDP-Lite, DCCP, RTP, FLUTE/ALC, NORM)
+ bidirectional (TCP, MPTCP, SCTP, TLS, HTTP)

o Transmission control

* flow control (TCP, MPTCP, SCTP, DCCP, TLS, RTP, HTTP)
* congestion control (TCP, MPTCP, SCTP, DCCP, RTP, FLUTE/ALC, NORM). Congestion control can also be provided by the transport supporting an upper layer transport (e.g., TLS, RTP, HTTP).

* segmentation (TCP, MPTCP, SCTP, TLS, RTP, HTTP, FLUTE/ALC, NORM)

* data/message bundling (TCP, MPTCP, SCTP, TLS, HTTP)

* stream scheduling prioritization (SCTP, HTTP2)

* endpoint multiplexing (MPTCP)

o Security

* authentication of one end of a connection (TLS, DTLS, FLUTE/ALC)

* authentication of both ends of a connection (TLS, DTLS)

* confidentiality (TLS, DTLS)

* cryptographic integrity protection (TLS, DTLS)

* replay protection (TLS, DTLS, FLUTE/ALC)

6. IANA Considerations

This document has no considerations for IANA.

7. Security Considerations

This document surveys existing transport protocols and protocols providing transport-like services. Confidentiality, integrity, and authenticity are among the features provided by those services. This document does not specify any new features or mechanisms for providing these features. Each RFC referenced by this document discusses the security considerations of the specification it contains.

8. Contributors

In addition to the editors, this document is the work of Brian Adamson, Dragana Damjanovic, Kevin Fall, Simone Ferlin-Oliviera, Ralph Holz, Olivier Mehani, Karen Nielsen, Colin Perkins, Vincent Roca, and Michael Tuexen.
Section 3.2 on MPTCP was contributed by Simone Ferlin-Oliviera (ferlin@simula.no) and Olivier Mehani (olivier.mehani@nicta.com.au)

Section 3.3 on UDP was contributed by Kevin Fall (kfall@kfall.com)

Section 3.5 on SCTP was contributed by Michael Tuexen (tuexen@fh-muenster.de) and Karen Nielsen (karen.nielsen@tieto.com)

Section 3.7 on TLS and DTLS was contributed by Ralph Holz (ralph.holz@nicta.com.au) and Olivier Mehani (olivier.mehani@nicta.com.au)

Section 3.8 on RTP contains contributions from Colin Perkins (csp@cisperkins.org)

Section 3.9 on HTTP was contributed by Dragana Damjanovic (ddamjanovic@mozilla.com)

Section 3.10 on FLUTE/ALC was contributed by Vincent Roca (vincent.roca@inria.fr)

Section 3.11 on NORM was contributed by Brian Adamson (brian.adamson@nrl.navy.mil)

9. Acknowledgments

Thanks to Joe Touch, Michael Welzl, Spencer Dawkins, and the TAPS Working Group for the comments, feedback, and discussion. This work is supported by the European Commission under grant agreement No. 318627 mPlane and from the Horizon 2020 research and innovation program under grant agreements No. 644334 (NEAT) and No. 688421 (MAMI). This support does not imply endorsement.

10. Informative References


Authors’ Addresses

Godred Fairhurst (editor)
University of Aberdeen
School of Engineering, Fraser Noble Building
Aberdeen AB24 3UE

Email: gorry@erg.abdn.ac.uk

Brian Trammell (editor)
ETH Zurich
Gloriastrasse 35
8092 Zurich
Switzerland

Email: ietf@trammell.ch

Mirja Kuehlewind (editor)
ETH Zurich
Gloriastrasse 35
8092 Zurich
Switzerland

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