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

S. Gjessing
M. Welzl
University of Oslo

June 20, 2017

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

Abstract

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

Status of this Memo

Copyright Notice

Table of Contents

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 possiblity 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

applications implementing Path MTU Discovery.

- Get max. transport frame size that may be received from the configured interface
  This is optional for a TAPS system to offer.

- 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:

- 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.

- 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 notifed 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:

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

CONNECT (flow-id dst_addr)

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

PARAMETERS:
dst_addr: the destination transport address to connect to

LISTEN (flow-id)

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

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

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

PARAMETERS:
reliability: this parameter is used to convey a choice of: fully reliable, unreliable without congestion control (which is guaranteed), unreliable, partially reliable (how to configure: TBD, probably using a time value). The latter two choices are not guaranteed and may result in full reliability.
ordered: this boolean parameter lets an application choose between ordered message delivery (true) and possibly unordered, potentially faster message delivery (false).

bundle: a boolean that expresses a preference for allowing to bundle frames (true) or not (false). No guarantees are given.
delack: a boolean that, if false, lets an application request that the peer would not delay the acknowledgement for this frame.
fragment: a boolean that expresses a preference for allowing to fragment frames (true) or not (false), at the IP level. No guarantees are given.
idempotent: a boolean that expresses whether a frame is idempotent (true) or not (false). Idempotent frames may arrive multiple times at the receiver. When data is idempotent it can be used by the receiver immediately on a connection establishment attempt. Thus, if SEND_FRAME is used before connecting, stating that a frame is idempotent facilitates transmitting it to the peer application particularly early.

CLOSE (flow-id)

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

ABORT (flow-id)

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

RECEIVE_FRAME (flow-id buffer)

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

PARAMETERS:

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

5. Conclusion

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

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

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

6. Acknowledgements

The authors would like to thank the participants of the TAPS Working Group and the NEAT research project for valuable input to this document. We especially thank Michael Tuexen for help with TAPS flow connection establishment/teardown and Gorry Fairhurst for his suggestions regarding fragmentation and packet sizes. This work has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No. 644334 (NEAT). The views expressed are solely those of the author(s).

7. IANA Considerations

XX RFC ED - PLEASE REMOVE THIS SECTION XXX

This memo includes no request to IANA.

8. Security Considerations

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

9. References

9.1. Normative References

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

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

9.2. Informative References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Appendix A. Deriving the minimal set

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

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

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

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

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

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

We assume that TAPS applications have no specific requirements that need knowledge about the network, e.g. regarding the choice of network interface or the end-to-end path. Even with these assumptions, there are certain requirements that are strictly kept by transport protocols today, and these must also be kept by a TAPS system. Some of these requirements relate to transport features that we call "Functional".

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

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

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

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

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

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

- 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:

- 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).

- 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.

- Request multiple streams
  Protocols: SCTP
  Automatable because using multi-streaming does not require application-specific knowledge.
  Implementation: see Appendix A.3.2.

- 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.

- 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.

- 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.

- 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.

- 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:

- 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).

- 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.

- 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.

- 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.

- 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.

- 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.

- 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].

- Obtain requested number of streams
  Protocols: SCTP
  Automatable because using multi-streaming does not require application-specific knowledge.
  Implementation: see Appendix A.3.2.

- Limit the number of inbound streams
  Protocols: SCTP
  Automatable because using multi-streaming does not require application-specific knowledge.
  Implementation: see Appendix A.3.2.

o Indicate (and/or obtain upon completion) an Adaptation Layer via an adaptation code point
Protocols: SCTP
Functional because it allows to send extra data for the sake of identifying an adaptation layer, which by itself is application-specific.
Implementation: via a parameter in LISTEN.SCTP.
Fall-back to TCP: not possible.

o Request to negotiate interleaving of user messages
Protocols: SCTP
Automatable because it requires using multiple streams, but requesting multiple streams in the CONNECTION.ESTABLISHMENT category is automatable.
Implementation: via a parameter in LISTEN.SCTP.

MAINTENANCE:

o Change timeout for aborting connection (using retransmit limit or time value)
Protocols: TCP, SCTP
Functional because this is closely related to potentially assumed reliable data delivery.
Implementation: via CHANGE-TIMEOUT.TCP or CHANGE-TIMEOUT.SCTP.

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

o Disable Nagle algorithm
Protocols: TCP, SCTP
Optimizing because this decision depends on knowledge about the size of future data blocks and the delay between them.
Implementation: via DISABLE-NAGLE.TCP and DISABLE-NAGLE.SCTP.

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

o Notification of Excessive Retransmissions (early warning below abortion threshold)
Protocols: TCP
Optimizing because it is an early warning to the application, informing it of an impending functional event.
Implementation: via ERROR.TCP.

- 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.

- 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.

- 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.

- 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.

- 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.

- 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.

- 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].

- Reset Stream
  Protocols: SCTP
  Automatable because using multi-streaming does not require application-specific knowledge.
  Implementation: see Appendix A.3.2.

- Notification of Stream Reset
  Protocols: STCP
  Automatable because using multi-streaming does not require application-specific knowledge.
  Implementation: see Appendix A.3.2.

- 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.

- 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.

- Add Streams
  Protocols: SCTP
  Automatable because using multi-streaming does not require application-specific knowledge.
  Implementation: see Appendix A.3.2.

- Notification of Added Stream
  Protocols: STCP
  Automatable because using multi-streaming does not require application-specific knowledge.
  Implementation: see Appendix A.3.2.

- 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.

- 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.

- 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).

- 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.

- 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.

- Configure PMTUD
  Protocols: SCTP
  Automatable because Path MTU Discovery relates to knowledge about the network, not the application.

- 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).

- 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.

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

- o Timeout event when data could not be delivered for too long
  Protocols: TCP, SCTP
  Functional because this notifies that potentially assumed reliable data delivery is no longer provided.
  Implementation: via TIMEOUT.TCP and TIMEOUT.SCTP.

A.1.2. DATA Transfer Related Transport Features

A.1.2.1. Sending Data

- o Reliably transfer data, with congestion control
  Protocols: TCP, SCTP
  Functional because this is closely tied to properties of the data that an application sends or expects to receive.
  Implementation: via SEND.TCP and SEND.SCTP.

- o Reliably transfer a message, with congestion control
  Protocols: SCTP
  Functional because this is closely tied to properties of the data that an application sends or expects to receive.
  Implementation: via SEND.SCTP and SEND.TCP. With SEND.TCP, messages will not be identifiable by the receiver. Inform the application of the result.

- o Unreliably transfer a message
  Protocols: SCTP, UDP(-Lite)
  Optimizing because only applications know about the time criticality of their communication, and reliably 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 transfering 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.

- 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.

- 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.

- 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.

- 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

- 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

- 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 distinugish between unsent and unacknowledged messages.
  Implementation: via SENDFAILURE-EVENT.SCTP and SEND_FAILURE.UDP(-Lite).
  Fall-back to TCP: do nothing: this notification is not available and will therefore not occur with TCP.

- Notification of an unsent (part of a) message
  Protocols: SCTP, UDP(-Lite)
  Automatable because the distinction between unsent and unacknowledged is network-specific.

- Notification of an unacknowledged (part of a) message
  Protocols: SCTP
  Automatable because the distinction between unsent and unacknowledged is network-specific.

- 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.

- 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

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

TERMINATION:

- Close after reliably delivering all remaining data, causing an event informing the application on the other side
- Abort without delivering remaining data, causing an event informing the application on the other side
- Abort without delivering remaining data, not causing an event informing the application on the other side
- 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

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

A.2.2.2. Receiving Data

- Receive data (with no message delineation)
- 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).

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

A.3. Step 3: Discussion

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

A.3.1. Sending Messages, Receiving Bytes

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

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

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

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

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

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

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

- 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 assocations in case of SCTP, i.e. they apply to all streams in that assocation.

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:

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

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

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

A.3.6. Security

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

Security will be discussed in a separate TAPS document (to be referenced here when it appears). The minimal set presented in the present document therefore excludes all security related transport features: "Configure authentication", "Change authentication parameters", "Obtain authentication information" and 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