TAPS
Internet-Draft
Intended status: Informational
Expires: September 14, 2017

S. Gjessing
M. Welzl
University of Oslo
March 13, 2017

A Minimal Set of Transport Services for TAPS Systems
draft-gjessing-taps-minset-04

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-03.

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 September 14, 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. Step 1: Categorization -- The Superset of Transport Features 5
3.1. CONNECTION Related Transport Features 7
3.2. DATA Transfer Related Transport Features 18
3.2.1. Sending Data 18
3.2.2. Receiving Data 20
3.2.3. Errors 21
4. Step 2: Reduction -- The Reduced Set of Transport Features . . 22
4.1. CONNECTION Related Transport Features 23
4.2. DATA Transfer Related Transport Features 24
4.2.1. Sending Data 24
4.2.2. Receiving Data 24
4.2.3. Errors 24
5. Step 3: Discussion 24
5.1. Sending Messages, Receiving Bytes 24
5.2. Stream Schedulers Without Streams 26
5.3. Early Data Transmission 27
5.4. Sender Running Dry 27
5.5. Capacity Profile 28
5.6. Security 28
5.7. Packet Size 29
6. Step 4: Construction -- the Minimal Set of Transport Features 29
6.1. Flow Creation, Connection and Termination 30
6.2. Flow Group Configuration 31
6.3. Flow Configuration 32
6.4. Data Transfer 32
6.4.1. The Sender 32
6.4.2. The Receiver 34
7. Conclusion 34
8. Acknowledgements 35
9. IANA Considerations 35
10. Security Considerations 36
11. References 36
11.1. Normative References 36
11.2. Informative References 36
Appendix A. Revision information 37
Authors' Addresses 38

1. Introduction

An application has an intended usage and demands for transport services, and the task of any system that implements TAPS is to offer these 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 point of this document is to identify transport features that 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". Instead, the middleware would have to expose the "unordered message delivery" transport feature to its applications (alternatively, there may be

interesting ways for certain types of middleware to try to use some of the transport features that we describe here without exposing them to applications, 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 all of the transport features that are recommended as a "minimal set" here. As an example considering only TCP and UDP, a middleware or library that only exposes TCP's reliable bytestream cannot make use of UDP (unless it implements extra functionality on top of UDP) -- doing so could break a fundamental assumption that applications make about the data they send and receive.

This document approaches 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.

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. 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

Because QoS is out of scope of TAPS, this document assumes a "best effort" service model [RFC5290], [RFC7305]. Applications using a TAPS system can therefore not make any assumptions about e.g. the time it will take to send a message. We also 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 Section 5.2.

- 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.

3.1. 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
  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 Section 5.2.

- Limit the number of inbound streams
  Protocols: SCTP
  Automatable because using multi-streaming does not require application-specific knowledge.
  Implementation: see Section 5.2.

- 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.

- Specify which chunk types must always be authenticated
  Protocols: SCTP
  Functional because this has a direct influence on security.
  Implementation: via a parameter in CONNECT.SCTP.
  Fall-back to TCP: TBD: this relates to the TCP Authentication Option in Section 7.1 of [RFC5925], which is not currently covered by [TAPS2].

- 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, UDP(-Lite)
  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
  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.

- Specify which chunk types must always be authenticated
  Protocols: SCTP
  Functional because this has a direct influence on security.
  Implementation: via a parameter in CONNECT.SCTP.
  Fall-back to TCP: TBD: this relates to the TCP Authentication Option in Section 7.1 of [RFC5925], which is not currently covered by [TAPS2].

- Obtain requested number of streams
  Protocols: SCTP
  Automatable because using multi-streaming does not require application-specific knowledge.
  Implementation: see Section 5.2.

- Limit the number of inbound streams
  Protocols: SCTP
  Automatable because using multi-streaming does not require application-specific knowledge.
  Implementation: see Section 5.2.

- 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.

- 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: SCTP
  Functional because this has a direct influence on security.
  Implementation: via SETAUTH.SCTP.
  Fall-back to TCP: TBD: this relates to the TCP Authentication Option in Section 7.1 of [RFC5925], which is not currently covered by [TAPS2].

- 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: TBD: this relates to the TCP Authentication Option in Section 7.1 of [RFC5925], which is not currently covered by [TAPS2].

- Reset Stream
  Protocols: SCTP
  Automatable because using multi-streaming does not require application-specific knowledge.
  Implementation: see Section 5.2.

- Notification of Stream Reset
  Protocols: STCP
  Automatable because using multi-streaming does not require application-specific knowledge.
  Implementation: see Section 5.2.

- 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 Section 5.2.

- Notification of Added Stream
  Protocols: STCP
  Automatable because using multi-streaming does not require application-specific knowledge.
  Implementation: see Section 5.2.

- 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 Section 5.4).

- 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 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; however, this is different from this transport feature because SCTP's cookie life value is set on the client side, not the server side. The TCP client has no control of this value. Thus, the recommended fall-back implementation is to do nothing.

- 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 CHECKSUM.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 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.

- 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)
  Automatable because the ECN field relates to knowledge about the network, not the application.

- 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, with a communication sequence relating to this transport feature that is defined by the application protocol.
Implementation: via ABORT.TCP and ABORT.SCTP.

- 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.

3.2. DATA Transfer Related Transport Features

3.2.1. 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 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 Section 5.2.

- 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.

- 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: TBD: this relates to the TCP Authentication Option in Section 7.1 of [RFC5925], which is not currently covered by [TAPS2].

- 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.

3.2.2. 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 Section 5.2.

- 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.

3.2.3. Errors

This section describes sending failures that are associated with a specific call to in the "Sending Data" category (Section 3.2.1).

- 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 Section 5.4.

- 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.

4. 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.

4.1. CONNECTION Related Transport Features

ESTABLISHMENT:
- Connect
- Specify number of attempts and/or timeout for the first establishment message
- Specify which chunk types must always be authenticated
- Hand over a message to transfer (possibly multiple times) before connection establishment
- Hand over a message to transfer during connection establishment

AVAILABILITY:
- Listen
- Specify which chunk types must always be authenticated

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
- Choose a scheduler to operate between streams of an association
- Configure priority or weight for a scheduler
- Set Cookie life value
- 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
- 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

- Timeout event when data could not be delivered for too long

4.2. DATA Transfer Related Transport Features

4.2.1. 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

4.2.2. Receiving Data

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

4.2.3. Errors

This section describes sending failures that are associated with a specific call to in the "Sending Data" category (Section 3.2.1).

- 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

5. Step 3: Discussion

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

5.1. 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. It is also represents the only way that UDP(-Lite) applications can receive data today.

For the transport to operate on messages, it only needs be informed about them as they are handed over by a sending application; on the receiver side, receiving a message only differs from receiving a bytestream in that the application is told where messages begin and end in the former case but not in the latter. The receiving application can still operate on these messages as long as it does not rely on the transport layer to inform it about message boundaries.

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 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.

5.2. 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.

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 6.1).
- 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 Section 5.1). 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.

5.3. 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, but the TCP specification does not specify how to provide it to applications.

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 during connection establishment, respectively.

5.4. 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 buffer 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 has means 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.

5.5. 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.

5.6. Security

Both TCP and SCTP offer authentication. 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. Generally, to an application it is relevant whether the transport protocol authenticates its own control data, the user data, or both, and a TAPS system should therefore allow to

configure and query these three cases.

TBD -- more to come in the next version. This relates to the TCP Authentication Option in Section 7.1 of [RFC5925], which is not currently covered.

Set Cookie life value -- TBD in the next version: SCTP is client-side, TCP is server-side.

5.7. 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). This is the only transport feature related to packet sizes. UDP applications typically make use of IP-layer functionality to obtain the size of the link MTU; it would therefore seem that offering such functionality to TAPS applications could be useful, albeit in a transport protocol independent way.

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 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.

6. Step 4: Construction -- the Minimal Set of Transport Features

Based on the categorization, reduction and discussion in the previous sections, this section presents the minimal set of transport features that is offered by end systems supporting TAPS. They are described in an abstract fashion, i.e. they can be implemented in various different 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.

Future versions of this document will probably describe the transport features in this section in greater detail; for now, we only specify how they differ from the transport features they are based upon. We carry out an additional simplification: CONNECTION.ESTABLISHMENT "Specify number of attempts and/or timeout for the first establishment message" and CONNECTION.MAINTENANCE "Change timeout for

aborting connection (using retransmit limit or time value)" are essentially the same, just applied upon connection establishment or during the lifetime of a connection. The same is the case for CONNECTION.ESTABLISHMENT "Specify which chunk types must always be authenticated" and CONNECTION.MAINTENANCE "Change authentication parameters". We therefore state that connections (called TAPS flows) must be instantiated before connecting them, and allow configurations to be carried out before connecting (in cases where this is not allowed by the transport protocol, a TAPS system will have to internall delay this configuration until the flow has been connected).

6.1. 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 6.2 and Section 6.3 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 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, and specify which case is preferred (before / during). In case of transmission before establishment, 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 flow can be actively closed, i.e. terminated after reliably delivering all remaining data, or aborted, i.e. terminated without delivering remaining data. A timeout can be configured to abort a flow when data could not be delivered for too long. 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. In case of SCTP streams, "Stream Reset" (a "SSN Reset Request Parameter" in a "RE-CONFIG" chunk [RFC6525]) can be used to notify a peer of an intention to close a flow.

6.2. 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 that are taken from Section 4 unchanged, with the exception that some of them can also be applied initially (before a flow is connected).

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

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

Others:
- Choose a scheduler to operate between flows of a group

The following transport features are new or changed, based on the discussion in Section 5:
- Capacity profile
  This describes how an application wants to use its available capacity. Choices can be "lowest possible latency at the expense of overhead", "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]). (details TBD)

- Authentication
  TBD in the next version: Different from SCTP's original transport features, this will only allow to configure authenticating the whole transport, all control information, or user data (not to distinguish between various SCTP chunks, to avoid this protocol dependency). It will also have to be made in line with TCP Authentication [RFC5925]. For SCTP, this functionality will be based on the transport features "Change authentication parameters", "Obtain authentication information" and the initially available "Specify which chunk types must always be authenticated". Note that SCTP also allows per-message configuration via "Specifying a key id to be used to authenticate a message", which may affect Section 6.4.

- Set Cookie life value
  TBD in the next version (not yet sure how to handle the client vs. server semantics of SCTP and TCP, respectively)

6.3. Flow Configuration

A flow can be assigned a priority or weight for a scheduler.

6.4. Data Transfer

6.4.1. The Sender

This section discusses how to send data after flow establishment. Section 6.1 discusses the possiblity to hand over a message to send before or during establishment.

For compatibility with TCP receiver semantics, we define an "Application-Framed Bytestream". This is a bytestream where the sending application optionally informs 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.

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 Section 4:

- 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.

AUTHOR'S NOTE: do folks agree with this design? It ties fragmentation to UDP only, because we called SCTP's "Configure message fragmentation" transport feature "automatable". It is indeed questionable whether applications need control over fragmentation when they work with SCTP -- doing so creates a complication for app writers that may not be necessary, especially when messages can be interleaved.

Following Section 5.7, there are two new transport features and a notification:

- Query maximum unfragmented frame size
  This is optional for a TAPS system to offer, and if it is offered, it informs the sender about the maximum expected size of a data frame that it can send without fragmentation. This can aid applications implementing Path MTU Discovery.

- Query 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.

- Notify the application of a path change
  If an application has disallowed fragmentation via the "Specify DF field" transport feature, this notification may optionally tell it that a path has changed (with a means to identify the path, so that the application can e.g. tell two flipping paths apart from completely diverse path changes). This informs the application that it may have to repeat Path MTU Discovery, and it can have relevance for application-level congestion control. For MPTCP and SCTP, a TAPS system can implement this functionality using the "Obtain status (query or notification)" transport feature.

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 Section 4.

- 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 Section 5.4 -- SCTP's "SENDER DRY" is a special case where the threshold is 0). 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.

6.4.2. 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 TAPS system will still not inform the receiving application about them, but frames themselves will always stay intact (partial frames are not supported - see Section 5.1). Different from TCP's semantics, there is no guarantee that all bytes in the bytestream are received, 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.

7. 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 on the basis that 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.

The answers are different for every case. 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.

8. 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).

9. IANA Considerations

XX RFC ED - PLEASE REMOVE THIS SECTION XXX

This memo includes no request to IANA.

10. 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.

11. References

11.1. 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-03 (work in progress), March 2017.

11.2. 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.

[LBE-draft] Bless, R., "A Lower Effort Per-Hop Behavior (LE PHB)", draft-tsvwg-le-phb-00 (work in progress), October 2016.

[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>.

[RFC5290] Floyd, S. and M. Allman, "Comments on the Usefulness of Simple Best-Effort Traffic", RFC 5290, DOI 10.17487/RFC5290, July 2008, <http://www.rfc-editor.org/info/rfc5290>.

[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>.

[RFC7305] Lear, E., Ed., "Report from the IAB Workshop on Internet Technology Adoption and Transition (ITAT)", RFC 7305, DOI 10.17487/RFC7305, July 2014, <http://www.rfc-editor.org/info/rfc7305>.

[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. 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".

Authors' Addresses

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

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

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

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

TAPS
Internet-Draft
Intended status: Informational
Expires: 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

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

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

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

- 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.

- 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.

- 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 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:
- 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

TAPS
Internet-Draft
Intended status: Informational
Expires: September 14, 2017

K-J. Grinnemo
A. Brunstrom
P. Hurtig
Karlstad University
N. Khademi
University of Oslo
Z. Bozakov
Dell EMC Research Europe

March 13, 2017

Happy Eyeballs for Transport Selection
draft-grinnemo-taps-he-02

Abstract

Ideally, network applications should be able to select an appropriate transport solution from among available transport solutions. However, at present, there is no agreed-upon way to do this. In fact, there is not even an agreed-upon way for a source end host to determine if there is support for a particular transport along a network path. This draft addresses these issues, by proposing a Happy Eyeballs framework. The proposed Happy Eyeballs framework enables the selection of a transport solution that according to application requirements, pre-set policies, and estimated network conditions is the most appropriate one. Additionally, the proposed framework makes it possible for an application to find out whether a particular transport is supported along a network connection towards a specific destination or not.

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 September 14, 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. Definitions 2
2. Introduction 2
3. Problem Statement 3
4. The Happy Eyeballs Framework 3
5. Design and Implementation Considerations 5
5.1. Candidate List Generation 5
5.2. Caching 7
5.3. Concurrent Connection Attempts 7
6. Example Happy Eyeballs Scenario 7
7. IANA Considerations 8
8. Security Considerations 8
9. Acknowledgements 8
10. References 8
10.1. Normative References 8
10.2. Informative References 9
Authors' Addresses 9

1. Definitions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

2. Introduction

Information services on the Internet come in varying forms, such as web browsing, email, and on-demand multimedia. The main motivation behind the design of next-generation computer and communications networks is to provide a universal and easy access to these various types of information services on a single multi-service Internet. This means that all forms of communications, e.g., video, voice, data

and control signaling, along with all types of services -- from plain text web pages to multimedia applications -- are bonded in a single-service platform through Internet technology. To enable the next-generation networks, the TAPS Working Group suggests a decoupling between the transport service provided to an application, and the transport stack providing this transport service: An application requests an appropriate transport service on the basis of its transport requirements, and the available transport stack that best meets these requirements is selected. In case the most preferred transport stack is not supported along the network path to the destination, or is not supported by the end host, a less-preferred transport stack is selected instead. As a way to realize the selection of transport stacks, this document suggests a generalization of the transport Happy Eyeballs (HE) mechanism proposed in Wing et al. [RFC6555] which addresses the selection of complete transport solutions, and which lends itself to arbitrary transport selection criterias.

The HE mechanism was introduced as a means to facilitate IPv6 adoption. Dual-stack client applications should be encouraged to try setting up connections over IPv6 first, and fall back to using IPv4 if IPv6 connection attempts fail. However, serializing tests for IPv6 and IPv4 connectivity can result in large connection latencies. HE for IPv6 minimizes the cost in delay by parallelizing attempts over IPv6 and IPv4. HE has also been proposed as an efficient way to find out the optimal combination of IPv4/IPv6 and TCP/SCTP to use to connect to a server [I-D.wing-tsvwg-happy-eyeballs-sctp]. The HE framework suggested in this document could be seen as a natural continuation of this proposal.

3. Problem Statement

Currently, there is no agreed-upon way for a source end host to select an appropriate transport service for a given application. In fact, there is no common way for a source end-host to find out if a transport stack is supported along a network path between itself and a destination end host. As a consequence, it has become increasingly difficult to introduce new transport stacks, and several applications, including many web applications, run over TCP although there are other transport protocols that better meet the requirements of these applications.

4. The Happy Eyeballs Framework

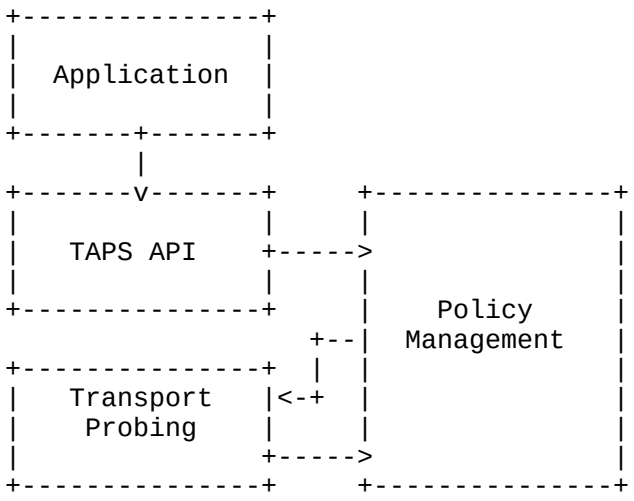


Figure 1: The Happy Eyeballs Framework.

The generalized HE mechanism proposed in this draft is carried out within the framework depicted in Figure 1. It comprises the following steps:

1. The Policy Management component takes as input application requirements from the TAPS API, stored information about previous connection attempts (e.g., whether previous connection attempts succeeded or not), and network conditions and configurations. On the basis of this input, the Policy Management component creates a list of candidate transport solutions, L, sorted in decreasing priority order. To be compliant with RFC 6555 [RFC6555], the Policy Management component SHOULD, in those cases there are no policies telling otherwise, give priority to IPv6 over IPv4.
2. It is the responsibility of the Transport Probing component to select the most appropriate transport solution. This is done by initiating connection attempts for each transport solution on L. To minimize the number of connection attempts that are initiated, the Transport Probing component SHOULD cache the outcome of connection attempts in a repository kept by the Policy Management component. The Policy Management component SHOULD in turn only include those transport solutions on L that have not been previously attempted, have valid successful connection-attempt cache entries, or have previously been attempted but whose cached connection-attempt entries have expired. Cached connection-attempt results SHOULD be valid for a configurable amount of time after which they SHOULD expire and have to be repeated. The transport solutions on L are initiated in priority order. The

difference in priority between two consecutive candidates, C1 and C2, is translated according to some criteria to a delay, D. D then governs the delay between the initiation of the connection attempts C1 and C2.

3. After the initiation of the connection attempts, the Transport Probing component waits for the first or winning connection to be established, which becomes the selected transport solution. For the Transport Probing component to be able to efficiently use the connection-attempt cache, already-initiated, non-winning transport solutions SHOULD NOT be terminated as soon as a winning connection has been established. Instead, they SHOULD themselves be given a fair chance to establish connections. In that way, the connection-attempt cache will be provided with a fairly accurate knowledge of which transport solutions work and does not work against frequently visited transport endpoints. Moreover, it MAY be beneficial to let those transport solutions which have a higher priority than the winning transport solution, live a predetermined amount of time after their establishment, since this enables the reuse of already established connections in later application requests.

5. Design and Implementation Considerations

This section discusses implementation issues that should be considered when a HE mechanism is designed and implemented on the basis of the HE framework proposed in this document.

5.1. Candidate List Generation

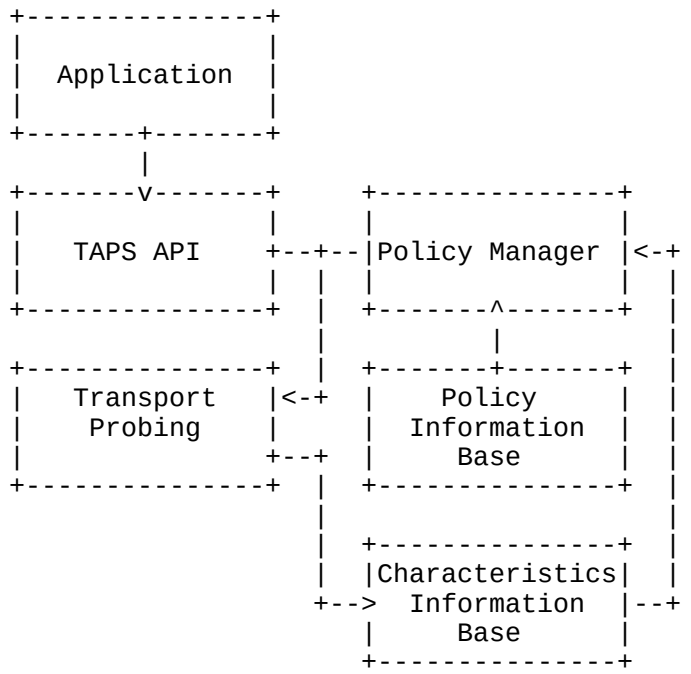


Figure 2: The NEAT Happy Eyeballs Framework.

There are several ways in which the list of candidate transport solutions, L, could be created by the Policy Management component. For example, L could be a list of all available transport solutions in an order that except for giving priority to IPv6 is arbitrary.

The NEAT System is developed as part of the EU Horizon 2020 project, "A New, Evolutive API and Transport-Layer Architecture for the Internet" (NEAT) [NEAT-Webb], and aims to provide a flexible and evolvable transport system that aligns with the charter of the TAPS Working Group. In the NEAT System [NEAT-Git], the HE framework is realized as shown in Figure 2. As follows, the Policy Management component comprises three components in the NEAT HE framework: a Policy Manager (PM), a Policy Information Base (PIB), and a Characteristics Information Base (CIB). PIB is a repository that stores a collection of policies that map application requests to transport solutions, i.e., map application requests to appropriately configured transport protocols, and CIB is a repository that stores information about previous connection attempts, available network interfaces, supported transport protocols etc. The PM takes as input application requirements from the TAPS API, and information from PIB and CIB. On the basis of this input, the PM creates L.

5.2. Caching

As pointed out in RFC 6555 [RFC6555], a HE algorithm should not waste networking resources by routinely making simultaneous connection attempts. To this end, the HE algorithm should cache the outcome of previous connection attempts to the same peer. The impact and efficiency of the HE algorithm has been evaluated in [Papastergiou16]. The paper suggests that caching significantly reduces the CPU load imposed by a HE mechanism.

5.3. Concurrent Connection Attempts

As mentioned in Section 4, it is the responsibility of the Transport Probing component to choose the most appropriate transport solution on the list of candidate transport solutions, L. Often this implies that several transport solutions need to be tried out, something which should not be carried out sequentially, but concurrently or partly overlapping depending on the transport-solution priorities. The way this is done is implementation dependent and varies between platforms. The NEAT library [NEAT-Git], which implements the HE framework herein, is built around the libuv asynchronous I/O library [LIBUV] and uses an event-based concurrency model to realize the concurrent initialization of connection attempts. The rationale behind using an event-based concurrency model is at least twofold: The first is that correctly managing concurrency in multi-threaded applications can be challenging with, for example, missing locks or deadlocks. The second is that multi-threading typically offers little or no control over what is scheduled at a given momemt in time. Given the complexity of building a general-purpose scheduler that works well in all cases, sometimes the OS will schedule work in a manner that is less than optimal. Proponents of threads argue that threads are a natural extension of sequential programming in that it maps work to be executed with individual threads. Threads are also a well-known and understood parts of OSes, and are mandatory for exploiting true CPU concurrency.

6. Example Happy Eyeballs Scenario

Consider a scenario in which an IPv4-only client using the NEAT System wishes to setup a connection to a server. Assume both the client and server support SCTP and TCP. The PM is queried about feasible transport solutions to connect to the server. This results in the PM retrieving information about network connections against this server from the CIB, e.g., supported transport protocols and the outcome of previous connection attempts. In our scenario, the PM learns from the CIB that the server supports SCTP and TCP, and, for the sake of this example, let us assume that the PM is also informed that previous connection attempts against this server, using both

SCTP and TCP, were successful. Next, the PM retrieves applicable policies from the PIB, and combines these policies with the previously retrieved CIB information. We assume in this example that the SCTP transport solution has a higher priority than the TCP solution. As a next step, the PM puts together the feasible candidate transport solutions in a list with SCTP over IPv4 placed at the head of the list followed by TCP over IPv4, and supplies this list to the Transport Probing component. The Transport Probing component traverses the candidate list, and initiates a connection attempt with SCTP against the server followed after a short while (governed by the difference in priorities between the SCTP and TCP transport solutions) by a connection attempt with TCP against the server. In our example, assume both connection attempts are successful, however, the SCTP connection attempt completes before the TCP attempt. The Transport Probing component caches in the CIB the SCTP connection attempt as successful, and returns the SCTP connection as the winning connection. When the TCP connection is established some time later, the Transport Probing component caches that connection attempt as successful as well.

7. IANA Considerations

XX RFC ED - PLEASE REMOVE THIS SECTION XXX

This memo includes no request to IANA.

8. Security Considerations

Security will be considered in future versions of this document.

9. Acknowledgements

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).

10. References

10.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://www.rfc-editor.org/info/rfc2119>.

[RFC6555] Wing, D. and A. Yourtchenko, "Happy Eyeballs: Success with Dual-Stack Hosts", RFC 6555, DOI 10.17487/RFC6555, April 2012, <http://www.rfc-editor.org/info/rfc6555>.

10.2. Informative References

[I-D.wing-tsvwg-happy-eyeballs-sctp]
Wing, D. and P. Natarajan, "Happy Eyeballs: Trending Towards Success with SCTP", draft-wing-tsvwg-happy-eyeballs-sctp-02 (work in progress), October 2010.

[LIBUV] libuv -- Asynchronous I/O Made Simple, "http://libuv.org", March 2017.

[NEAT-Git]
A New, Evolutive API and Transport-Layer Architecture for the Internet (NEAT), "https://github.com/NEAT-project/neat", March 2017.

[NEAT-Webb]
NEAT -- A New, Evolutive API and Transport-Layer Architecture for the Internet, "https://www.neat-project.org", March 2017.

[Papastergiou16]
Papastergiou, G., Grinnemo, K-J., Brunstrom, A., Ros, D., Tuexen, M., Khademi, N., and P. Hurtig, "On the Cost of Using Happy Eyeballs for Transport Protocol Selection", July 2016.

Authors' Addresses

Karl-Johan Grinnemo
Karlstad University
Universitetsgatan 2
Karlstad 651 88
Sweden

Phone: +46 54 700 24 40
Email: karl-johan.grinnemo@kau.se

Anna Brunstrom
Karlstad University
Universitetsgatan 2
Karlstad 651 88
Sweden

Phone: +46 54 700 17 95
Email: anna.brunstrom@kau.se

Per Hurtig
Karlstad University
Universitetsgatan 2
Karlstad 651 88
Sweden

Phone: +46 54 700 23 35
Email: per.hurtig@kau.se

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

Email: naeemk@ifi.uio.no

Zdravko Bozakov
Dell EMC Research Europe
Ovens, Co.
Cork
Ireland

Phone: +353 21 4945733
Email: Zdravko.Bozakov@dell.com

TAPS
Internet-Draft
Intended status: Experimental
Expires: December 3, 2017

K-J. Grinnemo
A. Brunstrom
P. Hurtig
Karlstad University
N. Khademi
University of Oslo
Z. Bozakov
Dell EMC Research Europe

June 2017

Happy Eyeballs for Transport Selection
draft-grinnemo-taps-he-03

Abstract

Ideally, network applications should be able to select an appropriate transport solution from among available transport solutions. However, at present, there is no agreed-upon way to do this. In fact, there is not even an agreed-upon way for a source end host to determine if there is support for a particular transport along a network path. This draft addresses these issues, by proposing a Happy Eyeballs framework. The proposed Happy Eyeballs framework enables the selection of a transport solution that according to application requirements, pre-set policies, and estimated network conditions is the most appropriate one. Additionally, the proposed framework makes it possible for an application to find out whether a particular transport is supported along a network connection towards a specific destination or not.

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 3, 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. Definitions 2
2. Introduction 2
3. Problem Statement 3
4. The Happy Eyeballs Framework 4
5. Design and Implementation Considerations 5
5.1. Candidate List Generation 5
5.2. Caching 7
5.3. Concurrent Connection Attempts 7
6. Example Happy Eyeballs Scenario 7
7. IANA Considerations 8
8. Security Considerations 8
9. Acknowledgements 8
10. References 8
10.1. Normative References 9
10.2. Informative References 9
Authors' Addresses 9

1. Definitions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

2. Introduction

Information services on the Internet come in varying forms, such as web browsing, email, and on-demand multimedia. The main motivation behind the design of next-generation computer and communications networks is to provide a universal and easy access to these various types of information services on a single multi-service Internet. This means that all forms of communications, e.g., video, voice, data

and control signaling, along with all types of services -- from plain text web pages to multimedia applications -- are bonded in a single-service platform through Internet technology. To enable the next-generation networks, the TAPS Working Group suggests a decoupling between the transport service provided to an application, and the transport stack providing this transport service: An application requests an appropriate transport service on the basis of its transport requirements, and the available transport stack that best meets these requirements is selected. In case the most preferred transport stack is not supported along the network path to the destination, or is not supported by the end host, a less-preferred transport stack is selected instead. As a way to realize the selection of transport stacks, this document suggests a generalization of the Happy Eyeballs (HE) mechanism proposed in Wing et al. [RFC6555] which addresses the selection of complete transport solutions, and which lends itself to arbitrary transport selection criterias. The proposed HE mechanism targets connection-oriented transport solutions, and connectionless transport solutions provided they offer some reasonable way to determine their successful use between endpoints.

The HE mechanism was introduced as a means to promote the use of dual network stacks. Dual-stack client applications should be encouraged to try setting up connections over IPv6 first, and fall back to using IPv4 if IPv6 connection attempts fail. However, serializing tests for IPv6 and IPv4 connectivity can result in large connection latencies. HE for IPv6 minimizes the cost in delay by parallelizing attempts over IPv6 and IPv4. HE has also been proposed as an efficient way to find out the optimal combination of IPv4/IPv6 and TCP/SCTP to use to connect to a server [I-D.wing-tsvwg-happy-eyeballs-sctp]. The HE framework suggested in this document could be seen as a natural continuation of this proposal.

3. Problem Statement

Currently, there is no agreed-upon way for a source end host to select an appropriate transport service for a given application. In fact, there is no common way for a source end-host to find out if a transport stack is supported along a network path between itself and a destination end host. As a consequence, it has become increasingly difficult to introduce new transport stacks, and several applications, including many web applications, run over TCP although there are other transport protocols that better meet the requirements of these applications.

4. The Happy Eyeballs Framework

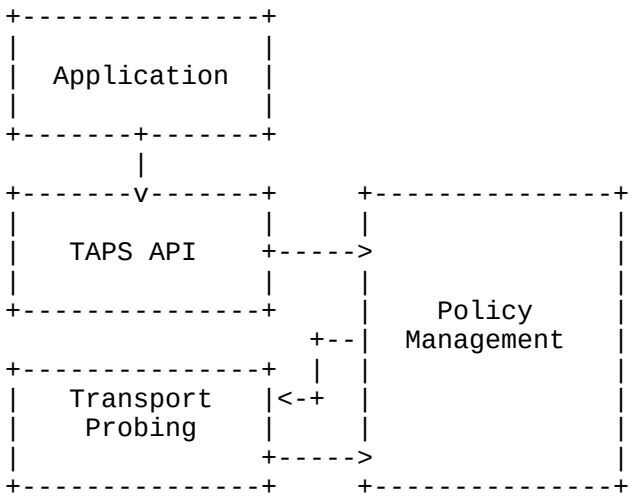


Figure 1: The Happy Eyeballs Framework.

The generalized HE mechanism proposed in this draft is carried out within the framework depicted in Figure 1. It comprises the following steps:

1. The Policy Management component takes as input application requirements from the TAPS API, stored information about previous connection attempts (e.g., whether previous connection attempts succeeded or not), and network conditions and configurations. On the basis of this input and the policies configured in the system, the Policy Management component creates a list of candidate transport solutions, L, sorted in decreasing priority order. To be compliant with RFC 6555 [RFC6555], the Policy Management component SHOULD, in those cases there are no policies telling otherwise, following the host's address preference, something which usually means giving preference to IPv6 over IPv4.
2. It is the responsibility of the Transport Probing component to select the most appropriate transport solution. This is done by initiating connection attempts for each transport solution on L. To minimize the number of connection attempts that are initiated, the Transport Probing component SHOULD cache the outcome of connection attempts in a repository kept by the Policy Management component. The Policy Management component SHOULD in turn only include those transport solutions on L that have not been previously attempted, have valid successful connection-attempt

cache entries, or have previously been attempted but whose cached connection-attempt entries have expired. Cached connection-attempt results SHOULD be valid for a configurable amount of time after which they SHOULD expire and have to be repeated. The transport solutions on L are initiated in priority order. The difference in priority between two consecutive candidates, C1 and C2, is translated according to some criteria to a delay, D. D then governs the delay between the initiation of the connection attempts C1 and C2.

3. After the initiation of the connection attempts, the Transport Probing component waits for the first or winning connection to be established, which becomes the selected transport solution. For the Transport Probing component to be able to efficiently use the connection-attempt cache, already-initiated, non-winning connection attempts SHOULD be given a fair chance to complete. In that way, the connection-attempt cache will be provided with a fairly accurate knowledge of which transport solutions work and does not work against frequently visited transport endpoints. Moreover, it MAY be beneficial to let those transport solutions which have a higher priority than the winning transport solution, live a predetermined amount of time after their establishment, since this enables the reuse of already established connections in later application requests.

5. Design and Implementation Considerations

This section discusses implementation issues that should be considered when a HE mechanism is designed and implemented on the basis of the HE framework proposed in this document.

5.1. Candidate List Generation

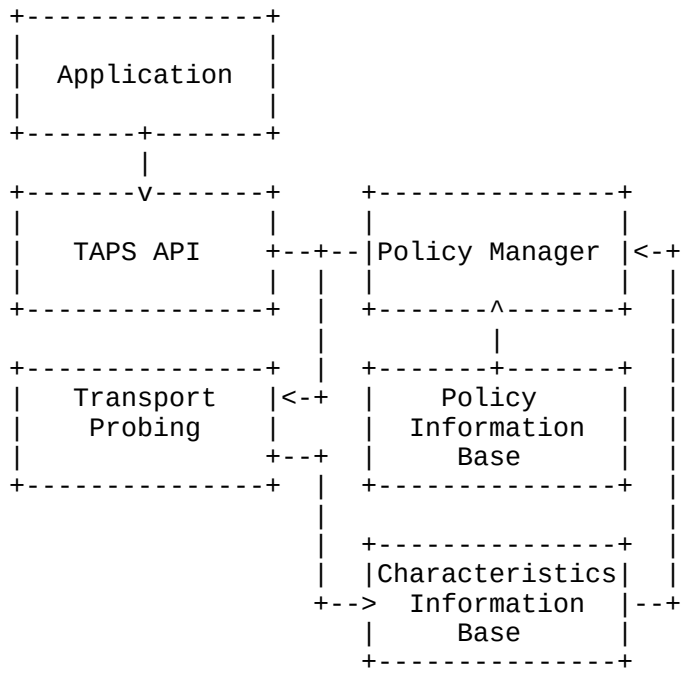


Figure 2: Principle Design of the NEAT Happy Eyeballs Framework.

There are several ways in which the list of candidate transport solutions, L, could be created by the Policy Management component. For example, L could be a list of all available transport solutions in an order that, except for following the host's address preference, is arbitrary; another, more sophisticated, way of creating the list of candidate transport solutions is the one employed by the NEAT System.

The NEAT System is developed as part of the EU Horizon 2020 project, "A New, Evolutive API and Transport-Layer Architecture for the Internet" (NEAT) [NEAT-Webb], and aims to provide a flexible and evolvable transport system that aligns with the charter of the TAPS Working Group. In the NEAT System [NEAT-Git], the HE framework is realized as shown in Figure 2. As follows, the Policy Management component comprises three components in the NEAT HE framework: a Policy Manager (PM), a Policy Information Base (PIB), and a Characteristics Information Base (CIB). PIB is a repository that stores a collection of policies that map application requests to transport solutions, i.e., map application requests to appropriately configured transport protocols, and CIB is a repository that stores information about previous connection attempts, available network

interfaces, supported transport protocols etc. The PM takes as input application requirements from the TAPS API, and information from PIB and CIB. On the basis of this input, the PM creates L.

5.2. Caching

As pointed out in RFC 6555 [RFC6555], a HE algorithm should not waste networking resources by routinely making simultaneous connection attempts. To this end, the HE algorithm should cache the outcome of previous connection attempts to the same peer. The cache lifetime is considered system dependent and should be set on a case-by-case basis. The impact and efficiency of the HE algorithm have been evaluated in [Papastergiou16]. The paper suggests that caching significantly reduces the CPU load imposed by a HE mechanism. It also indicates that the internal-memory footprint of a HE mechanism is essentially the same as for single-flow establishments.

5.3. Concurrent Connection Attempts

As mentioned in Section 4, it is the responsibility of the Transport Probing component to choose the most appropriate transport solution on the list of candidate transport solutions, L. Often this implies that several transport solutions need to be tried out, something which should not be carried out sequentially, but concurrently or partly overlapping depending on the transport-solution priorities. The way this is done is implementation dependent and varies between platforms. The NEAT library [NEAT-Git], which implements the HE framework herein, is built around the libuv asynchronous I/O library [LIBUV] and uses an event-based concurrency model to realize the concurrent initialization of connection attempts. The rationale behind using an event-based concurrency model is at least twofold: The first is that correctly managing concurrency in multi-threaded applications can be challenging with, for example, missing locks or deadlocks. The second is that multi-threading typically offers little or no control over what is scheduled at a given momemt in time. Given the complexity of building a general-purpose scheduler that works well in all cases, sometimes the OS will schedule work in a manner that is less than optimal. Those in favor of threads argue that threads are a natural extension of sequential programming in that it maps work to be executed with individual threads. Threads are also a well-known and understood parts of OSes, and are mandatory for exploiting true CPU concurrency.

6. Example Happy Eyeballs Scenario

Consider a scenario in which an IPv6-enabled client using the NEAT System wishes to setup a connection to a server. Assume both the client and server support SCTP and TCP. The Policy Management is

queried about feasible transport solutions to connect to the server. In the NEAT System, this results in PM retrieving information about network connections against this server from the CIB, e.g., supported transport protocols and the outcome of previous connection attempts. In our scenario, the PM learns from the CIB that the server supports SCTP and TCP, and, for the sake of this example, let us assume that the PM is also informed that previous connection attempts against this server, using both SCTP and TCP, were successful. Next, the PM retrieves applicable policies from the PIB, and combines these policies with the previously retrieved CIB information. We assume in this example that the SCTP transport solution has a higher priority than the TCP solution. As a next step, the PM puts together the feasible candidate transport solutions in a list with SCTP over IPv6 placed at the head of the list followed by TCP over IPv6, and supplies this list to the Transport Probing component. The Transport Probing component traverses the candidate list, and initiates a connection attempt with SCTP against the server followed after a short while (governed by the difference in priorities between the SCTP and TCP transport solutions) by a connection attempt with TCP against the server. In our example, assume both connection attempts are successful, however, the SCTP connection attempt completes before the TCP attempt. The Transport Probing component caches in the CIB the SCTP connection attempt as successful, and returns the SCTP connection as the winning connection. When the TCP connection is established some time later, the Transport Probing component caches that connection attempt as successful as well.

7. IANA Considerations

XX RFC ED - PLEASE REMOVE THIS SECTION XXX

This memo includes no request to IANA.

8. Security Considerations

Security will be considered in future versions of this document.

9. Acknowledgements

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).

10. References

10.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://www.rfc-editor.org/info/rfc2119>.

[RFC6555] Wing, D. and A. Yourtchenko, "Happy Eyeballs: Success with Dual-Stack Hosts", RFC 6555, DOI 10.17487/RFC6555, April 2012, <http://www.rfc-editor.org/info/rfc6555>.

10.2. Informative References

[I-D.wing-tsvwg-happy-eyeballs-sctp]
Wing, D. and P. Natarajan, "Happy Eyeballs: Trending Towards Success with SCTP", draft-wing-tsvwg-happy-eyeballs-sctp-02 (work in progress), October 2010.

[LIBUV] libuv -- Asynchronous I/O Made Simple, "http://libuv.org", March 2017.

[NEAT-Git]
A New, Evolutive API and Transport-Layer Architecture for the Internet (NEAT), "https://github.com/NEAT-project/neat", March 2017.

[NEAT-Webb]
NEAT -- A New, Evolutive API and Transport-Layer Architecture for the Internet, "https://www.neat-project.org", March 2017.

[Papastergiou16]
Papastergiou, G., Grinnemo, K-J., Brunstrom, A., Ros, D., Tuexen, M., Khademi, N., and P. Hurtig, "On the Cost of Using Happy Eyeballs for Transport Protocol Selection", July 2016.

Authors' Addresses

Karl-Johan Grinnemo
Karlstad University
Universitetsgatan 2
Karlstad 651 88
Sweden

Phone: +46 54 700 24 40
Email: karl-johan.grinnemo@kau.se

Anna Brunstrom
Karlstad University
Universitetsgatan 2
Karlstad 651 88
Sweden

Phone: +46 54 700 17 95
Email: anna.brunstrom@kau.se

Per Hurtig
Karlstad University
Universitetsgatan 2
Karlstad 651 88
Sweden

Phone: +46 54 700 23 35
Email: per.hurtig@kau.se

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

Email: naeemk@ifi.uio.no

Zdravko Bozakov
Dell EMC Research Europe
Ovens, Co.
Cork
Ireland

Phone: +353 21 4945733
Email: Zdravko.Bozakov@dell.com

TAPS
Internet-Draft
Intended status: Informational
Expires: September 9, 2017

M. Welzl
University of Oslo
M. Tuexen
Muenster Univ. of Appl. Sciences
N. Khademi
University of Oslo
March 8, 2017

On the Usage of Transport Features Provided by IETF Transport Protocols
draft-ietf-taps-transports-usage-03

Abstract

This document describes how TCP, MPTCP, SCTP, UDP and UDP-Lite expose services to applications and how an application can configure and use the transport features that make up these services. It also discusses the service provided by the LEDBAT congestion control mechanism.

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 September 9, 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. Terminology 3
2. Introduction 3
3. Pass 1 5
3.1. Primitives Provided by TCP 5
3.1.1. Excluded Primitives or Parameters 8
3.2. Primitives Provided by MPTCP 9
3.3. Primitives Provided by SCTP 10
3.3.1. Excluded Primitives or Parameters 17
3.4. Primitives Provided by UDP and UDP-Lite 17
3.5. The service of LEDBAT 17
4. Pass 2 18
4.1. CONNECTION Related Primitives 19
4.2. DATA Transfer Related Primitives 30
5. Pass 3 33
5.1. CONNECTION Related Transport Features 33
5.2. DATA Transfer Related Transport Features 39
5.2.1. Sending Data 39
5.2.2. Receiving Data 40
5.2.3. Errors 40
6. Acknowledgements 41
7. IANA Considerations 41
8. Security Considerations 41
9. References 41
9.1. Normative References 41
9.2. Informative References 44
Appendix A. Overview of RFCs used as input for pass 1 45
Appendix B. How this document was developed 45
Appendix C. Revision information 47
Authors' Addresses 47

1. Terminology

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

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

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

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

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

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

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

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

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

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

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

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

2. Introduction

This document presents defined interactions between applications and the transport protocols TCP, MPTCP, SCTP, UDP and UDP-Lite as well as the LEDBAT congestion control mechanism in the form of primitives and Transport Features. Primitives can be invoked by an application or a transport protocol; the latter type is called an "event". The list of primitives and Transport Features in this document is strictly based on the parts of protocol specifications that describe what the protocol provides to an application using it and how the application interacts with it.

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

The document presents a three-pass process to arrive at a list of Transport Features. In the first pass, the relevant RFC text is discussed per protocol. In the second pass, this discussion is used to derive a list of primitives that are uniformly categorized across protocols. Here, an attempt is made to present or -- where text describing primitives does not yet exist -- construct primitives in a slightly generalized form to highlight similarities. This is, for example, achieved by renaming primitives of protocols or by avoiding a strict 1:1-mapping between the primitives in the protocol specification and primitives in the list. Finally, the third pass presents Transport Features based on pass 2, identifying which protocols implement them.

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

This document discusses unicast transport protocols and a unicast congestion control mechanism. Transport protocols provide communication between processes that operate on network endpoints, which means that they allow for multiplexing of communication between the same IP addresses, and normally this multiplexing is achieved using port numbers. Port multiplexing is therefore assumed to be always provided and not discussed in this document.

Some protocols are connection-oriented. Connection-oriented protocols often use an initial call to a specific transport primitive

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

3. Pass 1

This first iteration summarizes the relevant text parts of the RFCs describing the protocols, focusing on what each transport protocol provides to the application and how it is used (abstract API descriptions, where they are available).

3.1. Primitives Provided by TCP

[RFC0793] states: "The Transmission Control Protocol (TCP) is intended for use as a highly reliable host-to-host protocol between hosts in packet-switched computer communication networks, and in interconnected systems of such networks". Section 3.8 in [RFC0793] further specifies the interaction with the application by listing several transport primitives. It is also assumed that an Operating System provides a means for TCP to asynchronously signal the application; the primitives representing such signals are called ’events’ in this section. This section describes the relevant primitives.

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

This indicates that the timeout could also be specified as a count of retransmission.

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

send: this is the primitive that an application uses to give the local TCP transport endpoint a number of bytes that TCP should reliably send to the other side of the connection. The URGENT flag, if set, states that the data handed over by this send call is urgent and this urgency should be indicated to the receiving process in case the receiving application has not yet consumed all non-urgent data preceding it. An optional timeout parameter can be provided that updates the connection's timeout (see 'open').

receive: This primitive allocates a receiving buffer for a provided number of bytes. It returns the number of received bytes provided in the buffer when these bytes have been received and written into the buffer by TCP. The application is informed of urgent data via an URGENT flag: if it is on, there is urgent data. If it is off, there is no urgent data or this call to 'receive' has returned all the urgent data.

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

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

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

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

USER TIMEOUT event: This event, described in Section 3.9 of [RFC0793], is executed when the user timeout expires (see 'open'). All queues are flushed and the application is informed that the connection had to be aborted due to user timeout.

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

Type-of-Service: Section 4.2.4.2 of [RFC1122] states that the application layer MUST be able to specify the Type-of-Service (TOS) for segments that are sent on a connection. The application should be able to change the TOS during the connection lifetime, and the TOS value should be passed to the IP layer unchanged. Since then the TOS field has been redefined. A part of the field has been assigned to ECN [RFC3168] and the six most significant bits have been assigned to carry the DiffServ CodePoint, DSField [RFC3260]. Staying with the intention behind the application's ability to specify the "Type of Service", this should probably be interpreted to mean the value in the DSField, which is the Differentiated Services Codepoint (DSCP).

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

User Timeout Option: The User Timeout Option (UTO) [RFC5482] allows one end of a TCP connection to advertise its current user timeout value so that the other end of the TCP connection can adapt its own user timeout accordingly. In addition to the configurable value of the User Timeout (see 'send'), [RFC5482] introduces three per-connection state variables that an application can adjust to control the operation of the User Timeout Option (UTO): ADV_UTO is the value of the UTO advertised to the remote TCP peer (default: system-wide default user timeout); ENABLED (default false) is a boolean-type flag that controls whether the UTO option is enabled

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

Fast Open: TCP Fast Open (TFO) [RFC7413] allows to immediately hand over a message from the active open to the passive open side of a TCP connection together with the first message establishment packet (the SYN). This can be useful for applications that are sensitive to TCP's connection setup delay. TCP implementations MUST NOT use TFO by default, but only use TFO if requested explicitly by the application on a per-service-port basis. To benefit from TFO, the first application data unit (e.g., an HTTP request) needs to be no more than TCP's maximum segment size (minus options used in the SYN). For the active open side, [RFC7413] recommends changing or replacing the connect() call in order to support a user data buffer argument. For the passive open side, the application needs to enable the reception of Fast Open requests, e.g. via a new TCP_FASTOPEN setsockopt() socket option before listen(). The receiving application must be prepared to accept duplicates of the TFO message, as the first data written to a socket can be delivered more than once to the application on the remote host.

3.1.1. Excluded Primitives or Parameters

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

The 'status' primitive was not included because [RFC0793] describes this primitive as "implementation dependent" and states that it "could be excluded without adverse effect". Moreover, while a data block containing specific information is described, it is also stated that not all of this information may always be available. The 'send' primitive described in [RFC0793] includes an optional PUSH flag which, if set, requires data to be promptly transmitted to the receiver without delay; the 'receive' primitive described in [RFC0793] can (under some conditions) yield the status of the PUSH flag. Because PUSH functionality is made optional to implement for both the 'send' and 'receive' primitives in [RFC1122], this functionality is not included here. [RFC1122] also introduces keep-alives to TCP, but these are optional to implement and hence not considered here. [RFC1122] describes that "some TCP implementations

have included a FLUSH call", indicating that this call is also optional to implement. It is therefore not considered here.

3.2. Primitives Provided by MPTCP

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

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

open: [RFC6897] states "An application should be able to request to turn on or turn off the usage of MPTCP.". The RFC states that this functionality can be provided through a socket-option called TCP_MULTIPATH_ENABLE. Further, [RFC6897] says that MPTCP must be disabled in case the application is binding to a specific address.

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

address and subflow management: MPTCP uses different addresses and allows a host to announce these addresses as part of the protocol. [RFC6897] says "An application should be able to restrict MPTCP to binding to a given set of addresses." and thus allows applications to limit the set of addresses that are being used by MPTCP.

Further, "An application should be able to obtain information on the pairs of addresses used by the MPTCP subflows.".

3.3. Primitives Provided by SCTP

Section 1.1 of [RFC4960] lists limitations of TCP that SCTP removes. Three of the four mentioned limitations directly translate into Transport Features that are visible to an application using SCTP: 1) it allows for preservation of message delineations; 2) these messages, while reliably transferred, do not require to be in order unless the application wants it; 3) multi-homing is supported. In SCTP, connections are called "associations" and they can be between not only two (as in TCP) but multiple addresses at each endpoint.

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

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

Initialize: Initialize, described in [RFC4960], creates a local SCTP instance that it binds to a set of local addresses (and, if provided, port number). Initialize needs to be called only once per set of local addresses. [RFC6458] also describes a number of per-association initialization parameters that can be used when an

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

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

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

rather than sent. A choice (advisory, i.e. not guaranteed) of the preferred path can be made by providing a socket, and the message can be delivered out-of-order if the unordered flag is set. An advisory flag indicates that the peer should not delay the acknowledgement of the user message provided by making use of the I-bit specified in [RFC7053]. Another advisory flag indicates whether the application prefers to avoid bundling user data with other outbound DATA chunks (i.e., in the same packet). A payload protocol-id can be provided to pass a value that indicates the type of payload protocol data to the peer. If the extension specified in [RFC4895] is used, the key identifier used for authenticating the DATA chunks can be provided.

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

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

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

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

Configure Max. Retransmissions of an Association: The parameter Association.Max.Retrans in [RFC4960], called sasoc_maxrxt in [RFC6458], allows to configure the number of unsuccessful retransmissions after which an entire association is considered as failed (which should invoke a COMMUNICATION LOST notification).

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Set Cookie life value: The Cookie life value can be adjusted as explained in Section 8.1.2 of [RFC6458]. "Valid.Cookie.Life" is also one of the parameters listed as potentially adjustable with SETPROTOCOLPARAMETERS in [RFC4960].

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

Configure RTO calculation: [RFC4960] lists the following adjustable parameters: RTO.Initial; RTO.Min; RTO.Max; RTO.Alpha; RTO.Beta. Only the initial, minimum and maximum RTO are also described as configurable [RFC6458].

Set DSCP value: Section 8.1.12 of [RFC6458] explains how to set the DSCP value per peer address of an association.

Set IPv6 flow label: Section 8.1.12 of [RFC6458] explains how to set the flow label field per peer address of an association.

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

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

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

DATA ARRIVE notification: When a message is ready to be retrieved via the Receive primitive, the application is informed by this notification.

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

NETWORK STATUS CHANGE notification: The NETWORK STATUS CHANGE notification informs the application about a socket becoming active/inactive [RFC4960] or "Potentially Failed" [RFC7829].

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

SHUTDOWN COMPLETE notification: When SCTP completes the shutdown procedures, this notification is passed to the upper layer, informing it about the affected assocation.

AUTHENTICATION notification: When SCTP wants to notify the upper layer regarding the key management related to the extension defined in [RFC4895], this notification is passed to the upper layer.

ADAPTATION LAYER INDICATION notification: When SCTP completes the association setup and the peer provided an adaptation layer indication, this is passed to the upper layer. This extension is defined in [RFC5061] and [RFC6458].

STREAM RESET notification: When SCTP completes the procedure for resetting streams as specified in [RFC6525], this notification is passed to the upper layer, informing it about the result.

ASSOCIATION RESET notification: When SCTP completes the association reset procedure as specified in [RFC6525], this notification is passed to the upper layer, informing it about the result.

STREAM CHANGE notification: When SCTP completes the procedure used to increase the number of streams as specified in [RFC6525], this notification is passed to the upper layer, informing it about the result.

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

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

3.3.1. Excluded Primitives or Parameters

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

3.4. Primitives Provided by UDP and UDP-Lite

The primitives provided by UDP and UDP-Lite are described in [FJ16].

3.5. The service of LEDBAT

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

LEDBAT does not have any primitives, as LEDBAT is not a transport protocol. [RFC6817] states: "LEDBAT can be used as part of a transport protocol or as part of an application, as long as the data transmission mechanisms are capable of carrying timestamps and acknowledging data frequently. LEDBAT can be used with TCP, Stream Control Transmission Protocol (SCTP), and Datagram Congestion Control Protocol (DCCP), with appropriate extensions where necessary; and it can be used with proprietary application protocols, such as those built on top of UDP for peer-to- peer (P2P) applications." At the time of writing, the appropriate extensions for TCP, SCTP or DCCP do not exist.

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

Regarding which of these parameters should be under control of an application, the possible range goes from exposing nothing on the one hand, to considering everything that is not fully prescribed with a MUST in [RFC6817] as a parameter on the other hand. Function implementations are not provided as a parameter to any of the transport protocols discussed here, and hence we do not regard the FILTER() function as a parameter. However, to avoid unnecessarily limiting future implementations, we consider all other parameters above as tunable parameters that a TAPS system should expose.

4. Pass 2

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

UDP and UDP-Lite; the PROTOCOL name "TCP" refers to both TCP and MPTCP. We present "connection" as a general protocol-independent concept and use it to refer to, e.g., TCP connections (identifiable by a unique pair of IP addresses and TCP port numbers), SCTP associations (identifiable by multiple IP address and port number pairs), as well UDP and UDP-Lite connections (identifiable by a unique socket pair).

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

The TCP 'send' and 'receive' primitives include usage of an "URGENT" mechanism. This mechanism is required to implement the "synch signal" used by telnet [RFC0854], but SHOULD NOT be used by new applications [RFC6093]. Because pass 2 is meant as a basis for the creation of TAPS systems, the "URGENT" mechanism is excluded. This also concerns the notification "Urgent pointer advance" in the ERROR_REPORT described in Section 4.2.4.1 of [RFC1122].

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

4.1. CONNECTION Related Primitives

ESTABLISHMENT:
Active creation of a connection from one transport endpoint to one or more transport endpoints.
Interfaces to UDP and UDP-Lite allow both connection-oriented and connection-less usage of the API . [RFC8085]

- CONNECT.TCP:
  Pass 1 primitive / event: 'open' (active) or 'open' (passive) with socket, followed by 'send'
  Parameters: 1 local IP address (optional); 1 destination transport address (for active open; else the socket and the local IP address

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

- CONNECT.SCTP:
  Pass 1 primitive / event: 'initialize', followed by 'enable / disable interleaving' (optional), followed by 'associate'
  Parameters: list of local SCTP port number / IP address pairs (initialize); one or several sockets (identifying the peer); outbound stream count; maximum allowed inbound stream count; adaptation layer indication (optional); chunk types required to be authenticated (optional); request interleaving on/off; maximum number of INIT attemps (optional); maximum init. RTO for INIT (optional); user message (optional); remote UDP port number (optional)
  Returns: socket list or failure
  Comments: 'initialize' needs to be called only once per list of local SCTP port number / IP address pairs. One socket will automatically be chosen; it can later be changed in MAINTENANCE. The user message may be transmitted to the peer application immediately upon reception of the packet containing the COOKIE-ECHO chunk. To benefit from the lower latency this provides, its length must be limited such that it fits into the packet containing the COOKIE-ECHO chunk. If a remote UDP port number is provided, SCTP packets will be encapsulated in UDP.

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

- CONNECT.UDP(-Lite):
  Pass 1 primitive / event: 'connect' followed by 'send'.
  Parameters: 1 local IP address (default (ANY), or specified); 1 destination transport address; 1 local port (default (OS chooses), or specified); 1 destination port (default (OS chooses), or specified).
  Comments: Associates a transport address creating a UDP(-Lite)

socket connection. This can be called again with a new transport address to create a new connection. The CONNECT function allows an application to receive errors from messages sent to a transport address.

AVAILABILITY:
Preparing to receive incoming connection requests.

- LISTEN.TCP:
  Pass 1 primitive / event: 'open' (passive)
  Parameters: 1 local IP address (optional); 1 socket (optional); timeout (optional); buffer to receive a user message (optional)
  Comments: if the socket and/or local IP address is provided, this waits for incoming connections from only and/or to only the provided address. Else this waits for incoming connections without this / these constraint(s). ESTABLISHMENT can later be performed with 'send'. If a buffer is provided to receive a user message, a user message can be received from a TFO-enabled sender before TCP's connection handshake is completed. This message may arrive multiple times.

- LISTEN.SCTP:
  Pass 1 primitive / event: 'initialize', followed by 'COMMUNICATION UP' or 'RESTART' notification and possibly 'ADAPTATION LAYER' notification
  Parameters: list of local SCTP port number / IP address pairs (initialize)
  Returns: socket list; outbound stream count; inbound stream count; adaptation layer indication; chunks required to be authenticated; interleaving supported on both sides yes/no
  Comments: initialize needs to be called only once per list of local SCTP port number / IP address pairs. COMMUNICATION UP can also follow a COMMUNICATION LOST notification, indicating that the lost communication is restored. If the peer has provided an adaptation layer indication, an 'ADAPTATION LAYER' notification is issued.

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

- LISTEN.UDP(-Lite):
  Pass 1 primitive / event: 'receive'.
  Parameters: 1 local IP address (default (ANY), or specified); 1 destination transport address; local port (default (OS chooses), or specified); destination port (default (OS chooses), or

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

MAINTENANCE:
Adjustments made to an open connection, or notifications about it. These are out-of-band messages to the protocol that can be issued at any time, at least after a connection has been established and before it has been terminated (with one exception: CHANGE_TIMEOUT.TCP can only be issued for an open connection when DATA.SEND.TCP is called). In some cases, these primitives can also be immediately issued during ESTABLISHMENT or AVAILABILITY, without waiting for the connection to be opened (e.g. CHANGE_TIMEOUT.TCP can be done using TCP's 'open' primitive). For UDP and UDP-Lite, these functions may establish a setting per connection, but may also be changed per datagram message.

- CHANGE_TIMEOUT.TCP:
  Pass 1 primitive / event: 'open' or 'send' combined with unspecified control of per-connection state variables
  Parameters: timeout value (optional); ADV_UTO (optional); boolean UTO_ENABLED (optional, default false); boolean CHANGEABLE (optional, default true)
  Comments: when sending data, an application can adjust the connection's timeout value (time after which the connection will be aborted if data could not be delivered). If UTO_ENABLED is true, the user timeout value (or, if provided, the value ADV_UTO) will be advertised for the TCP on the other side of the connection to adapt its own user timeout accordingly. UTO_ENABLED controls whether the UTO option is enabled for a connection. This applies to both sending and receiving. CHANGEABLE controls whether the user timeout may be changed based on a UTO option received from the other end of the connection; it becomes false when 'timeout value' is used.

- CHANGE_TIMEOUT.SCTP:
  Pass 1 primitive / event: 'Change HeartBeat' combined with 'Configure Max. Retransmissions of an Association'
  Parameters: 'Change HeartBeat': heartbeat frequency; 'Configure Max. Retransmissions of an Association': Association.Max.Retrans
  Comments: Change Heartbeat can enable / disable heartbeats in SCTP as well as change their frequency. The parameter Association.Max.Retrans defines after how many unsuccessful transmissions of any packets (including heartbeats) the association will be terminated; thus these two primitives / parameters together can yield a similar behavior for SCTP associations as CHANGE_TIMEOUT.TCP does for TCP connections.

- DISABLE_NAGLE.TCP:
  Pass 1 primitive / event: not specified
  Parameters: one boolean value
  Comments: the Nagle algorithm delays data transmission to increase the chance to send a full-sized segment. An application must be able to disable this algorithm for a connection.

- DISABLE_NAGLE.SCTP:
  Pass 1 primitive / event: 'Enable/disable NODELAY'
  Parameters: one boolean value
  Comments: Nagle-like algorithms delay data transmission to increase the chance to send a full-sized packet.

- REQUEST_HEARTBEAT.SCTP:
  Pass 1 primitive / event: 'Request HeartBeat'
  Parameters: socket
  Returns: success or failure
  Comments: requests an immediate heartbeat on a path, returning success or failure.

- ADD_PATH.MPTCP:
  Pass 1 primitive / event: not specified
  Parameters: local IP address and optionally the local port number
  Comments: the application specifies the local IP address and port number that must be used for a new subflow.

- ADD_PATH.SCTP:
  Pass 1 primitive / event: Change Local Address / Set Peer Primary
  Parameters: local IP address

- REM_PATH.MPTCP:
  Pass 1 primitive / event: not specified
  Parameters: local IP address, local port number, remote IP address, remote port number
  Comments: the application removes the subflow specified by the IP/port-pair. The MPTCP implementation must trigger a removal of the subflow that belongs to this IP/port-pair.

- REM_PATH.SCTP:
  Pass 1 primitive / event: 'Change Local Address / Set Peer Primary'
  Parameters: local IP address

- SET_PRIMARY.SCTP:
  Pass 1 primitive / event: 'Set Primary'
  Parameters: socket
  Returns: result of attempting this operation
  Comments: update the current primary address to be used, based on

the set of available sockets of the association.

- SET_PEER_PRIMARY.SCTP:
  Pass 1 primitive / event: 'Change Local Address / Set Peer Primary'
  Parameters: local IP address
  Comments: this is only advisory for the peer.

- CONFIG_SWITCHOVER.SCTP:
  Pass 1 primitive / event: 'Configure Path Switchover'
  Parameters: primary max retrans (no. of retransmissions after which a path is considered inactive), PF max retrans (no. of retransmissions after which a path is considered to be "Potentially Failed", and others will be preferably used) (optional)

- STATUS.SCTP:
  Pass 1 primitive / event: 'Status', 'Enable / Disable Interleaving' and 'NETWORK STATUS CHANGE notification'.
  Returns: data block with information about a specified association, containing: association connection state; destination transport address list; destination transport address reachability states; current local and peer receiver window sizes; current local congestion window sizes; number of unacknowledged DATA chunks; number of DATA chunks pending receipt; primary path; most recent SRTT on primary path; RTO on primary path; SRTT and RTO on other destination addresses; MTU per path; interleaving supported yes/no.
  Comments: The NETWORK STATUS CHANGE notification informs the application about a socket becoming active/inactive; this only affects the programming style, as the same information is also available via 'Status'.

- STATUS.MPTCP:
  Pass 1 primitive / event: not specified
  Returns: list of pairs of tuples of IP address and TCP port number of each subflow. The first of the pair is the local IP and port number, while the second is the remote IP and port number.

- SET_DSCP.TCP:
  Pass 1 primitive / event: not specified
  Parameters: DSCP value
  Comments: this allows an application to change the DSCP value for outgoing segments. For TCP this was originally specified for the TOS field [RFC1122], which is here interpreted to refer to the DSField [RFC3260].

- SET_DSCP.SCTP:
  Pass 1 primitive / event: 'Set DSCP value'
  Parameters: DSCP value
  Comments: this allows an application to change the DSCP value for outgoing packets on a path.

- SET_DSCP.UDP(-Lite):
  Pass 1 primitive / event: 'SET_DSCP'
  Parameter: DSCP value
  Comments: This allows an application to change the DSCP value for outgoing UDP(-Lite) datagrams. [RFC7657] and [RFC8085] provide current guidance on using this value with UDP.

- ERROR.TCP:
  Pass 1 primitive / event: 'ERROR_REPORT'
  Returns: reason (encoding not specified); subreason (encoding not specified)
  Comments: soft errors that can be ignored without harm by many applications; an application should be able to disable these notifications. The reported conditions include at least: ICMP error message arrived; Excessive Retransmissions.

- ERROR.UDP(-Lite):
  Pass 1 primitive / event: 'ERROR_REPORT'.
  Returns: Error report
  Comments: This returns soft errors that may be ignored without harm by many applications; An application must connect to be able receive these notifications.

- SET_AUTH.SCTP:
  Pass 1 primitive / event: 'Set / Get Authentication Parameters'
  Parameters: key_id, key, hmac_id

- GET_AUTH.SCTP:
  Pass 1 primitive / event: 'Set / Get Authentication Parameters'
  Parameters: key_id, chunk_list

- RESET_STREAM.SCTP:
  Pass 1 primitive / event: 'Add / Reset Streams, Reset Association'
  Parameters: sid, direction

- RESET_STREAM-EVENT.SCTP:
  Pass 1 primitive / event: 'STREAM RESET notification'
  Parameters: information about the result of RESET_STREAM.SCTP.
  Comments: This is issued when the procedure for resetting streams has completed.

- RESET_ASSOC.SCTP:
  Pass 1 primitive / event: 'Add / Reset Streams, Reset Association'
  Parameters: information related to the extension defined in [RFC3260].

- RESET_ASSOC-EVENT.SCTP:
  Pass 1 primitive / event: 'ASSOCIATION RESET notification'
  Parameters: information about the result of RESET_ASSOC.SCTP.
  Comments: This is issued when the procedure for resetting an association has completed.

- ADD_STREAM.SCTP:
  Pass 1 primitive / event: 'Add / Reset Streams, Reset Association'
  Parameters: number if outgoing and incoming streams to be added

- ADD_STREAM-EVENT.SCTP:
  Pass 1 primitive / event: 'STREAM CHANGE notification'
  Parameters: information about the result of ADD_STREAM.SCTP.
  Comments: This is issued when the procedure for adding a stream has completed.

- SET_STREAM_SCHEDULER.SCTP:
  Pass 1 primitive / event: 'Set Stream Scheduler'
  Parameters: scheduler identifier
  Comments: choice of First Come First Serve, Round Robin, Round Robin per Packet, Priority Based, Fair Bandwidth, Weighted Fair Queuing.

- CONFIGURE_STREAM_SCHEDULER.SCTP:
  Pass 1 primitive / event: 'Configure Stream Scheduler'
  Parameters: priority
  Comments: the priority value only applies when Priority Based or Weighted Fair Queuing scheduling is chosen with SET_STREAM_SCHEDULER.SCTP. The meaning of the parameter differs between these two schedulers but in both cases it realizes some form of prioritization regarding how bandwidth is divided among streams.

- SET_FLOWLABEL.SCTP:
  Pass 1 primitive / event: 'Set IPv6 flow label'
  Parameters: flow label
  Comments: this allows an application to change the IPv6 header's flow label field for outgoing packets on a path.

- AUTHENTICATION_NOTIFICATION-EVENT.SCTP:
  Pass 1 primitive / event: 'AUTHENTICATION notification'
  Returns: information regarding key management.

- CONFIG_SEND_BUFFER.SCTP:
  Pass 1 primitive / event: 'Configure send buffer size'
  Parameters: size value in octets

- CONFIG_RECEIVE_BUFFER.SCTP:
  Pass 1 primitive / event: 'Configure receive buffer size'
  Parameters: size value in octets
  Comments: this controls the receiver window.

- CONFIG_FRAGMENTATION.SCTP:
  Pass 1 primitive / event: 'Configure message fragmentation'
  Parameters: one boolean value (enable/disable), maximum fragmentation size (optional; default: PMTU)
  Comments: if fragmentation is enabled, messages exceeding the maximum fragmentation size will be fragmented. If fragmentation is disabled, trying to send a message that exceeds the maximum fragmentation size will produce an error.

- CONFIG_PMTUD.SCTP:
  Pass 1 primitive / event: 'Configure Path MTU Discovery'
  Parameters: one boolean value (PMTUD on/off), PMTU value (optional)
  Returns: PMTU value
  Comments: This returns a meaningful PMTU value when PMTUD is enabled (the boolean is true), and the PMTU value can be set if PMTUD is disabled (the boolean is false)

- CONFIG_DELAYED_SACK.SCTP:
  Pass 1 primitive / event: 'Configure delayed SACK timer'
  Parameters: one boolean value (delayed SACK on/off), timer value (optional), number of packets to wait for (default 2)
  Comments: If delayed SACK is enabled, SCTP will send a SACK upon either receiving the provided number of packets or when the timer expires, whatever occurs first.

- CONFIG_RTO.SCTP:
  Pass 1 primitive / event: 'Configure RTO calculation'
  Parameters: init (optional), min (optional), max (optional)
  Comments: This adjusts the initial, minimum and maximum RTO values.

- SET_COOKIE_LIFE.SCTP:
  Pass 1 primitive / event: 'Set Cookie life value'
  Parameters: cookie life value

- SET_MAX_BURST.SCTP:
  Pass 1 primitive / event: 'Set maximum burst'
  Parameters: max burst value

Comments: not all implementations allow values above the default of 4.

- SET_PARTIAL_DELIVERY_POINT.SCTP:
  Pass 1 primitive / event: 'Set Partial Delivery Point'
  Parameters: partial delivery point (integer)
  Comments: this parameter must be smaller or equal to the socket receive buffer size.

- CHECKSUM.UDP:
  Pass 1 primitive / event: 'DISABLE_CHECKSUM'.
  Parameters: 0 when no checksum is used at sender, 1 for checksum at sender (default)

- CHECKSUM_REQUIRED.UDP:
  Pass 1 primitive / event: 'REQUIRE_CHECKSUM'.
  Parameter: 0 when checksum is required at receiver, 1 to allow zero checksum at receiver (default)

- SET_CHECKSUM_COVERAGE.UDP-Lite:
  Pass 1 primitive / event: 'SET_CHECKSUM_COVERAGE'
  Parameters: Coverage length at sender (default maximum coverage)

- SET_MIN_CHECKSUM_COVERAGE.UDP-Lite:
  Pass 1 primitive / event: 'SET_MIN_COVERAGE'.
  Parameter: Coverage length at receiver (default minimum coverage)

- SET_DF.UDP(-Lite):
  Pass 1 primitive event: 'SET_DF'.
  Parameter: 0 when DF is not set (default), 1 when DF is set

- SET_TTL.UDP(-Lite) (IPV6_UNICAST_HOPS):
  Pass 1 primitive / event: 'SET_TTL' and 'SET_IPV6_UNICAST_HOPS'
  Parameters: IPv4 TTL value or IPv6 Hop Count value
  Comments: This allows an application to change the IPv4 TTL of IPv6 Hop count value for outgoing UDP(-Lite) datagrams.

- GET_TTL.UDP(-Lite) (IPV6_UNICAST_HOPS):
  Pass 1 primitive / event: 'GET_TTL' and 'GET_IPV6_UNICAST_HOPS'
  Returns: IPv4 TTL value or IPv6 Hop Count value
  Comments: This allows an application to read the the IPv4 TTL of IPv6 Hop count value from a received UDP(-Lite) datagram.

- SET_ECN.UDP(-Lite):
  Pass 1 primitive / event: 'SET_ECN'
  Parameters: ECN value
  Comments: This allows a UDP(-Lite) application to set the ECN codepoint field for outgoing UDP(-Lite) datagrams.

- GET_ECN.UDP(-Lite):
  Pass 1 primitive / event: 'GET_ECN'
  Parameters: ECN value
  Comments: This allows a UDP(-Lite) application to read the ECN codepoint field from a received UDP(-Lite) datagram.

- SET_IP_OPTIONS.UDP(-Lite):
  Pass 1 primitive / event: 'SET_IP_OPTIONS'
  Parameters: options
  Comments: This allows a UDP(-Lite) application to set IP Options for outgoing UDP(-Lite) datagrams. These options can at least be the Source Route, Record Route, and Time Stamp option.

- GET_IP_OPTIONS.UDP(-Lite):
  Pass 1 primitive / event: 'GET_IP_OPTIONS'
  Returns: options
  Comments: This allows a UDP(-Lite) application to receive any IP options that are contained in a received UDP(-Lite) datagram.

- CONFIGURE.LEDBAT:
  Pass 1 primitive / event: N/A
  Parameters: enable (boolean), TARGET, ALLOWED_INCREASE, GAIN_INC, GAIN_DEC, BASE_HISTORY, CURRENT_FILTER, INIT_CWND, MIN_CWND
  Comments: enable is a newly invented parameter that enables or disables the whole LEDBAT service.

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

- CLOSE.TCP:
  Pass 1 primitive / event: 'close'
  Comments: this terminates the sending side of a connection after reliably delivering all remaining data.

- CLOSE.SCTP:
  Pass 1 primitive / event: 'Shutdown'
  Comments: this terminates a connection after reliably delivering all remaining data.

- CLOSE.UDP(-Lite):
  Pass 1 primitive event: 'CLOSE'
  Comments: No further UDP(-Lite) datagrams are sent/received on this connection.

- ABORT.TCP:
  Pass 1 primitive / event: 'abort'
  Comments: this terminates a connection without delivering remaining data and sends an error message to the other side.

- ABORT.SCTP:
  Pass 1 primitive / event: 'abort'
  Parameters: abort reason to be given to the peer (optional)
  Comments: this terminates a connection without delivering remaining data and sends an error message to the other side.

- TIMEOUT.TCP:
  Pass 1 primitive / event: 'USER TIMEOUT' event
  Comments: the application is informed that the connection is aborted. This event is executed on expiration of the timeout set in CONNECTION.ESTABLISHMENT.CONNECT.TCP (possibly adjusted in CONNECTION.MAINTENANCE.CHANGE_TIMEOUT.TCP).

- TIMEOUT.SCTP:
  Pass 1 primitive / event: 'COMMUNICATION LOST' event
  Comments: the application is informed that the connection is aborted. this event is executed on expiration of the timeout that should be enabled by default (see beginning of section 8.3 in [RFC4960]) and was possibly adjusted in CONNECTION.MAINTENANCE.CHANGE_TIMEOOUT.SCTP.

- ABORT-EVENT.TCP:
  Pass 1 primitive / event: not specified.

- ABORT-EVENT.SCTP:
  Pass 1 primitive / event: 'COMMUNICATION LOST' event
  Returns: abort reason from the peer (if available)
  Comments: the application is informed that the other side has aborted the connection using CONNECTION.TERMINATION.ABORT.SCTP.

- CLOSE-EVENT.TCP:
  Pass 1 primitive / event: not specified.

- CLOSE-EVENT.SCTP:
  Pass 1 primitive / event: 'SHUTDOWN COMPLETE' event
  Comments: the application is informed that CONNECTION.TERMINATION.CLOSE.SCTP was successfully completed.

4.2. DATA Transfer Related Primitives

All primitives in this section refer to an existing connection, i.e. a connection that was either established or made available for

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

- SEND.TCP:
  Pass 1 primitive / event: 'send'
  Parameters: timeout (optional)
  Comments: this gives TCP a data block for reliable transmission to the TCP on the other side of the connection. The timeout can be configured with this call whenever data are sent (see also CONNECTION.MAINTENANCE.CHANGE-TIMEOUT.TCP).

- SEND.SCTP:
  Pass 1 primitive / event: 'Send'
  Parameters: stream number; context (optional); socket (optional); unordered flag (optional); no-bundle flag (optional); payload protocol-id (optional); pr-policy (optional) pr-value (optional); sack-immediately flag (optional); key-id (optional)
  Comments: this gives SCTP a data block for transmission to the SCTP on the other side of the connection (SCTP association). The 'stream number' denotes the stream to be used. The 'context' number can later be used to refer to the correct message when an error is reported. The 'socket' can be used to state which path should be preferred, if there are multiple paths available (see also CONNECTION.MAINTENANCE.SETPRIMARY.SCTP). The data block can be delivered out-of-order if the 'unordered flag' is set. The 'no-bundle flag' can be set to indicate a preference to avoid bundling. The 'payload protocol-id' is a number that will, if provided, be handed over to the receiving application. Using pr-policy and pr-value the level of reliability can be controlled. The 'sack-immediately' flag can be used to indicate that the peer should not delay the sending of a SACK corresponding to the provided user message. If specified, the provided key-id is used for authenticating the user message.

- SEND.UDP(-Lite):
  Pass 1 primitive / event: 'SEND'
  Parameters: IP Address and Port Number of the destination endpoint (optional if connected).
  Comments: This provides a message for unreliable transmission using UDP(-Lite) to the specified transport address. IP address and Port may be omitted for connected UDP(-Lite) sockets. All CONNECTION.MAINTENANCE.SET_*.UDP(-Lite) primitives apply per message sent.

- RECEIVE.TCP:
  Pass 1 primitive / event: 'receive'.

- RECEIVE.SCTP:
  Pass 1 primitive / event: 'DATA ARRIVE' notification, followed by 'Receive'
  Parameters: stream number (optional)
  Returns: stream sequence number (optional), partial flag (optional)
  Comments: if the 'stream number' is provided, the call to receive only receives data on one particular stream. If a partial message arrives, this is indicated by the 'partial flag', and then the 'stream sequence number' must be provided such that an application can restore the correct order of data blocks that comprise an entire message. Additionally, a delivery number lets the application detect reordering.

- RECEIVE.UDP(-Lite):
  Pass 1 primitive / event: 'RECEIVE',
  Parameters: Buffer for received datagram.
  Comments: All CONNECTION.MAINTENANCE.GET_*.UDP(-Lite) primitives apply per message received.

- SENDFAILURE-EVENT.SCTP:
  Pass 1 primitive / event: 'SEND FAILURE' notification, optionally followed by 'Receive Unsent Message' or 'Receive Unacknowledged Message'
  Returns: cause code; context; unsent or unacknowledged message (optional)
  Comments: 'cause code' indicates the reason of the failure, and 'context' is the context number if such a number has been provided in DATA.SEND.SCTP, for later use with 'Receive Unsent Message' or 'Receive Unacknowledged Message', respectively. These primitives can be used to retrieve the unsent or unacknowledged message (or part of the message, in case a part was delivered) if desired.

- SEND_FAILURE.UDP(-Lite):
  Pass 1 primitive / event: 'SEND'
  Comments: This may be used to probe for the effective PMTU when using in combination with the 'MAINTENANCE.SET_DF' primitive.

- SENDER_DRY-EVENT.SCTP:
  Pass 1 primitive / event: 'SENDER DRY' notification
  Comments: This informs the application that the stack has no more user data to send.

- PARTIAL_DELIVERY_ABORTED-EVENT.SCTP:
  Pass 1 primitive / event: 'PARTIAL DELIVERY ABORTED' notification
  Comments: This informs the receiver of a partial message that the further delivery of the message has been aborted.

5. Pass 3

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

5.1. CONNECTION Related Transport Features

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

- Connect
  Protocols: TCP, SCTP, UDP(-Lite)

- Specify which IP Options must always be used
  Protocols: TCP

- Request multiple streams
  Protocols: SCTP

- Limit the number of inbound streams
  Protocols: SCTP

- Specify number of attempts and/or timeout for the first establishment message
  Protocols: TCP, SCTP

- Obtain multiple sockets
  Protocols: SCTP

- Disable MPTCP
  Protocols: MPTCP

- Specify which chunk types must always be authenticated
  Protocols: SCTP
  Comments: DATA, ACK etc. are different 'chunks' in SCTP; one or more chunks may be included in a single packet.

- Indicate an Adaptation Layer (via an adaptation code point)
  Protocols: SCTP

- Request to negotiate interleaving of user messages
  Protocols: SCTP

- Hand over a message to transfer (possibly multiple times) before connection establishment
  Protocols: TCP

- Hand over a message to transfer during connection establishment
  Protocols: SCTP

- Enable UDP encapsulation with a specified remote UDP port number
  Protocols: SCTP

AVAILABILITY:
Preparing to receive incoming connection requests.

- Listen, 1 specified local interface
  Protocols: TCP, SCTP, UDP(-Lite)

- Listen, N specified local interfaces
  Protocols: SCTP, UDP(-Lite)

- Listen, all local interfaces
  Protocols: TCP, SCTP, UDP(-Lite)

- Obtain requested number of streams
  Protocols: SCTP

- Limit the number of inbound streams
  Protocols: SCTP

- Specify which IP Options must always be used
  Protocols: TCP

- Disable MPTCP
  Protocols: MPTCP

- Specify which chunk types must always be authenticated
  Protocols: SCTP
  Comments: DATA, ACK etc. are different 'chunks' in SCTP; one or more chunks may be included in a single packet.

- Indicate an Adaptation Layer (via an adaptation code point)
  Protocols: SCTP

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

- Change timeout for aborting connection (using retransmit limit or time value)
  Protocols: TCP, SCTP

- Suggest timeout to the peer
  Protocols: TCP

- Disable Nagle algorithm
  Protocols: TCP, SCTP

- Request an immediate heartbeat, returning success/failure
  Protocols: SCTP

- Notification of Excessive Retransmissions (early warning below abortion threshold)
  Protocols: TCP

- Add path
  Protocols: MPTCP, SCTP
  MPTCP Parameters: source-IP; source-Port; destination-IP; destination-Port
  SCTP Parameters: local IP address

- Remove path
  Protocols: MPTCP, SCTP
  MPTCP Parameters: source-IP; source-Port; destination-IP; destination-Port
  SCTP Parameters: local IP address

- Set primary path
  Protocols: SCTP

- Suggest primary path to the peer
  Protocols: SCTP

- Configure Path Switchover
  Protocols: SCTP

- Obtain status (query or notification)
  Protocols: SCTP, MPTCP
  SCTP parameters: association connection state; destination transport address list; destination transport address reachability states; current local and peer receiver window sizes; current local congestion window sizes; number of unacknowledged DATA chunks; number of DATA chunks pending receipt; primary path; most recent SRTT on primary path; RTO on primary path; SRTT and RTO on other destination addresses; MTU per path; interleaving supported yes/no
  MPTCP parameters: subflow-list (identified by source-IP; source-Port; destination-IP; destination-Port)

- Specify DSCP field
  Protocols: TCP, SCTP, UDP(-Lite)

- Notification of ICMP error message arrival
  Protocols: TCP, UDP(-Lite)

- Change authentication parameters
  Protocols: SCTP

- Obtain authentication information
  Protocols: SCTP

- Reset Stream
  Protocols: SCTP

- Notification of Stream Reset
  Protocols: STCP

- Reset Association
  Protocols: SCTP

- Notification of Association Reset
  Protocols: STCP

- Add Streams
  Protocols: SCTP

- Notification of Added Stream
  Protocols: STCP

- Choose a scheduler to operate between streams of an association
  Protocols: SCTP

- Configure priority or weight for a scheduler
  Protocols: SCTP

- Specify IPv6 flow label field
  Protocols: SCTP

- Configure send buffer size
  Protocols: SCTP

- Configure receive buffer (and rwnd) size
  Protocols: SCTP

- Configure message fragmentation
  Protocols: SCTP

- Configure PMTUD
  Protocols: SCTP

- Configure delayed SACK timer
  Protocols: SCTP

- Set Cookie life value
  Protocols: SCTP

- Set maximum burst
  Protocols: SCTP

- Configure size where messages are broken up for partial delivery
  Protocols: SCTP

- Disable checksum when sending
  Protocols: UDP

- Disable checksum requirement when receiving
  Protocols: UDP

- Specify checksum coverage used by the sender
  Protocols: UDP-Lite

- Specify minimum checksum coverage required by receiver
  Protocols: UDP-Lite

- Specify DF field
  Protocols: UDP(-Lite)

- Specify TTL/Hop count field
  Protocols: UDP(-Lite)

- Obtain TTL/Hop count field
  Protocols: UDP(-Lite)

- Specify ECN field
  Protocols: UDP(-Lite)

- Obtain ECN field
  Protocols: UDP(-Lite)

- Specify IP Options
  Protocols: UDP(-Lite)

- Obtain IP Options
  Protocols: UDP(-Lite)

- Enable and configure "Low Extra Delay Background Transfer"
  Protocols: A protocol implementing the LEDBAT congestion control mechanism

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

- Close after reliably delivering all remaining data, causing an event informing the application on the other side
  Protocols: TCP, SCTP
  Comments: A TCP endpoint locally only closes the connection for sending; it may still receive data afterwards.

- Abort without delivering remaining data, causing an event informing the application on the other side
  Protocols: TCP, SCTP
  Comments: In SCTP a reason can optionally be given by the application on the aborting side, which can then be received by the application on the other side.

- Timeout event when data could not be delivered for too long
  Protocols: TCP, SCTP
  Comments: the timeout is configured with CONNECTION.MAINTENANCE "Change timeout for aborting connection (using retransmit limit or time value)".

5.2. DATA Transfer Related Transport Features

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

5.2.1. Sending Data

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

- Reliably transfer data, with congestion control
  Protocols: TCP

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

- Unreliably transfer a message, with congestion control
  Protocols: SCTP

- Unreliably transfer a message, without congestion control
  Protocols: UDP(-Lite)

- Configurable Message Reliability
  Protocols: SCTP

- Choice of stream
  Protocols: SCTP

- Choice of path (destination address)
  Protocols: SCTP

- Choice between unordered (potentially faster) or ordered delivery of messages
  Protocols: SCTP

- Request not to bundle messages
  Protocols: SCTP

- Specifying a "payload protocol-id" (handed over as such by the receiver)
  Protocols: SCTP

- Specifying a key id to be used to authenticate a message
  Protocols: SCTP

- Request not to delay the acknowledgement (SACK) of a message
  Protocols: SCTP

5.2.2. Receiving Data

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

- Receive data (with no message delineation)
  Protocols: TCP

- Receive a message
  Protocols: SCTP, UDP(-Lite)

- Choice of stream to receive from
  Protocols: SCTP

- Information about partial message arrival
  Protocols: SCTP
  Comments: In SCTP, partial messages are combined with a stream sequence number so that the application can restore the correct order of data blocks an entire message consists of.

- Obtain a message delivery number
  Protocols: SCTP
  Comments: This number can let applications detect and, if desired, correct reordering.

5.2.3. Errors

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

- Notification of an unsent (part of a) message
  Protocols: SCTP, UDP(-Lite)

- Notification of an unacknowledged (part of a) message
  Protocols: SCTP

- Notification that the stack has no more user data to send
  Protocols: SCTP

- Notification to a receiver that a partial message delivery has been aborted
  Protocols: SCTP

6. Acknowledgements

The authors would like to thank (in alphabetical order) Bob Briscoe, Gorry Fairhurst, David Hayes, Tom Jones, Karen Nielsen, Joe Touch and Brian Trammell for providing valuable feedback on this document. We especially thank Christoph Paasch for providing input related to Multipath TCP. 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 these features on its own. Therefore, these features are not considered in this document, with the exception of native authentication capabilities of SCTP for which the security considerations in [RFC4895] apply.

9. References

9.1. Normative References

[FJ16] Fairhurst, G. and T. Jones, "Features of the User Datagram Protocol (UDP) and Lightweight UDP (UDP-Lite) Transport Protocols", draft-ietf-taps-transports-usage-udp-00 (work in progress), November 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-08 (work in progress), October 2016.

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

[RFC1122] Braden, R., Ed., "Requirements for Internet Hosts - Communication Layers", STD 3, RFC 1122, DOI 10.17487/RFC1122, October 1989, <http://www.rfc-editor.org/info/rfc1122>.

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

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

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

[RFC5061] Stewart, R., Xie, Q., Tuexen, M., Maruyama, S., and M. Kozuka, "Stream Control Transmission Protocol (SCTP) Dynamic Address Reconfiguration", RFC 5061, DOI 10.17487/RFC5061, September 2007, <http://www.rfc-editor.org/info/rfc5061>.

[RFC5482] Eggert, L. and F. Gont, "TCP User Timeout Option", RFC 5482, DOI 10.17487/RFC5482, March 2009, <http://www.rfc-editor.org/info/rfc5482>.

[RFC6182] Ford, A., Raiciu, C., Handley, M., Barre, S., and J. Iyengar, "Architectural Guidelines for Multipath TCP Development", RFC 6182, DOI 10.17487/RFC6182, March 2011, <http://www.rfc-editor.org/info/rfc6182>.

[RFC6458] Stewart, R., Tuexen, M., Poon, K., Lei, P., and V. Yasevich, "Sockets API Extensions for the Stream Control

Transmission Protocol (SCTP)", RFC 6458, DOI 10.17487/RFC6458, December 2011, <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>.

[RFC6817] Shalunov, S., Hazel, G., Iyengar, J., and M. Kuehlewind, "Low Extra Delay Background Transport (LEDBAT)", RFC 6817, DOI 10.17487/RFC6817, December 2012, <http://www.rfc-editor.org/info/rfc6817>.

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

[RFC6897] Scharf, M. and A. Ford, "Multipath TCP (MPTCP) Application Interface Considerations", RFC 6897, DOI 10.17487/RFC6897, March 2013, <http://www.rfc-editor.org/info/rfc6897>.

[RFC6951] Tuexen, M. and R. Stewart, "UDP Encapsulation of Stream Control Transmission Protocol (SCTP) Packets for End-Host to End-Host Communication", RFC 6951, DOI 10.17487/RFC6951, May 2013, <http://www.rfc-editor.org/info/rfc6951>.

[RFC7053] Tuexen, M., Ruengeler, I., and R. Stewart, "SACK-IMMEDIATELY Extension for the Stream Control Transmission Protocol", RFC 7053, DOI 10.17487/RFC7053, November 2013, <http://www.rfc-editor.org/info/rfc7053>.

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

[RFC7496] Tuexen, M., Seggelmann, R., Stewart, R., and S. Loreto, "Additional Policies for the Partially Reliable Stream Control Transmission Protocol Extension", RFC 7496, DOI 10.17487/RFC7496, April 2015, <http://www.rfc-editor.org/info/rfc7496>.

[RFC7829] Nishida, Y., Natarajan, P., Caro, A., Amer, P., and K. Nielsen, "SCTP-PF: A Quick Failover Algorithm for the Stream Control Transmission Protocol", RFC 7829, DOI 10.17487/RFC7829, April 2016,

<http://www.rfc-editor.org/info/rfc7829>.

[RFC8085] Eggert, L., Fairhurst, G., and G. Shepherd, "UDP Usage Guidelines", BCP 145, RFC 8085, DOI 10.17487/RFC8085, March 2017, <http://www.rfc-editor.org/info/rfc8085>.

9.2. Informative References

[RFC0854] Postel, J. and J. Reynolds, "Telnet Protocol Specification", STD 8, RFC 854, DOI 10.17487/RFC0854, May 1983, <http://www.rfc-editor.org/info/rfc854>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://www.rfc-editor.org/info/rfc2119>.

[RFC3168] Ramakrishnan, K., Floyd, S., and D. Black, "The Addition of Explicit Congestion Notification (ECN) to IP", RFC 3168, DOI 10.17487/RFC3168, September 2001, <http://www.rfc-editor.org/info/rfc3168>.

[RFC3260] Grossman, D., "New Terminology and Clarifications for Diffserv", RFC 3260, DOI 10.17487/RFC3260, April 2002, <http://www.rfc-editor.org/info/rfc3260>.

[RFC5461] Gont, F., "TCP’s Reaction to Soft Errors", RFC 5461, DOI 10.17487/RFC5461, February 2009, <http://www.rfc-editor.org/info/rfc5461>.

[RFC6093] Gont, F. and A. Yourtchenko, "On the Implementation of the TCP Urgent Mechanism", RFC 6093, DOI 10.17487/RFC6093, January 2011, <http://www.rfc-editor.org/info/rfc6093>.

[RFC7414] Duke, M., Braden, R., Eddy, W., Blanton, E., and A. Zimmermann, "A Roadmap for Transmission Control Protocol (TCP) Specification Documents", RFC 7414, DOI 10.17487/RFC7414, February 2015, <http://www.rfc-editor.org/info/rfc7414>.

[RFC7657] Black, D., Ed. and P. Jones, "Differentiated Services (Diffserv) and Real-Time Communication", RFC 7657, DOI 10.17487/RFC7657, November 2015, <http://www.rfc-editor.org/info/rfc7657>.

[RFC8095] Fairhurst, G., Ed., Trammell, B., Ed., and M. Kuehlewind, Ed., "Services Provided by IETF Transport Protocols and Congestion Control Mechanisms", RFC 8095, DOI 10.17487/

RFC8095, March 2017,
<http://www.rfc-editor.org/info/rfc8095>.

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

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

Appendix B. How this document was developed

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

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

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

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

interconnected systems of such networks."

Some text for pass 1 subsections was developed copy+pasting all the relevant text parts from the relevant RFCs, then adjusting terminology to match the terminology in Section 1 and adjusting (shortening!) phrasing to match the general style of the document. An effort was made to formulate everything as a primitive description such that the primitive descriptions became as complete as possible (e.g., the "SEND.TCP" primitive in pass 2 is explicitly described as reliably transferring data); text that is relevant for the primitives presented in this pass but still does not fit directly under any primitive was used in a subsection’s introduction.

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

For each primitive, the following style was used:

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

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

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

Appendix C. Revision information

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

-00 (from draft-welzl-taps-transports): this now covers TCP based on all TCP RFCs (this means: if you know of something in any TCP RFC that you think should be addressed, please speak up!) as well as SCTP, exclusively based on [RFC4960]. We decided to also incorporate [RFC6458] for SCTP, but this hasn't happened yet. Terminology made in line with [RFC8095]. Addressed comments by Karen Nielsen and Gorry Fairhurst; various other fixes. Appendices (TCP overview and how-to-contribute) added.

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

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

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

Authors' Addresses

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

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

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

Email: tuexen@fh-muenster.de

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

Email: naeemk@ifi.uio.no

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

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

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

Abstract

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

Status of this Memo

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

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

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

This Internet-Draft will expire on April 29, 2018.

Copyright Notice

Copyright (c) 2017 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust’s Legal Provisions Relating to IETF Documents

(http://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

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

1. Terminology

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

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

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

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

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

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

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

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

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

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

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

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

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

2. Introduction

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

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

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

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

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

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

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

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

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

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

3. Pass 1

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

3.1. Primitives Provided by TCP

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3.1.1. Excluded Primitives or Parameters

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

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

3.2. Primitives Provided by MPTCP

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

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

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

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

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

used by the MPTCP subflows".

3.3. Primitives Provided by SCTP

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3.3.1. Excluded Primitives or Parameters

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

3.4. Primitives Provided by UDP and UDP-Lite

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

3.5. The service of LEDBAT

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

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

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

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

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

4. Pass 2

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

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

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

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

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

4.1. CONNECTION Related Primitives

ESTABLISHMENT:

Active creation of a connection from one transport endpoint to one or more transport endpoints.
Interfaces to UDP and UDP-Lite allow both connection-oriented and connection-less usage of the API [RFC8085].

- CONNECT.TCP:

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

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

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

- CONNECT.SCTP:

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

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

  Returns: socket list or failure

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

- CONNECT.MPTCP:

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

- CONNECT.UDP(-Lite):

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

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

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

AVAILABILITY:

Preparing to receive incoming connection requests.

- LISTEN.TCP:

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

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

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

- LISTEN.SCTP:

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

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

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

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

- LISTEN.MPTCP:

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

- LISTEN.UDP(-Lite):

  Pass 1 primitive / event: 'Receive'.

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

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

MAINTENANCE:

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

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

- o CHANGE_TIMEOUT.TCP:

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

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

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

- o CHANGE_TIMEOUT.SCTP:

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

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

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

- DISABLE_NAGLE.TCP:

  Pass 1 primitive / event: not specified

  Parameters: one boolean value

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

- DISABLE_NAGLE.SCTP:

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

  Parameters: one boolean value

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

- REQUEST_HEARTBEAT.SCTP:

  Pass 1 primitive / event: 'Request HeartBeat'

  Parameters: socket

  Returns: success or failure

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

- ADD_PATH.MPTCP:

  Pass 1 primitive / event: not specified

  Parameters: local IP address and optionally the local port number

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

- ADD_PATH.SCTP:

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

Parameters: local IP address

- REM_PATH.MPTCP:

  Pass 1 primitive / event: not specified

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

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

- REM_PATH.SCTP:

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

  Parameters: local IP address

- SET_PRIMARY.SCTP:

  Pass 1 primitive / event: 'Set Primary'

  Parameters: socket

  Returns: result of attempting this operation

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

- SET_PEER_PRIMARY.SCTP:

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

  Parameters: local IP address

  Comments: this is only advisory for the peer.

- CONFIG_SWITCHOVER.SCTP:

  Pass 1 primitive / event: 'Configure Path Switchover'

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

- STATUS.SCTP:

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

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

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

- STATUS.MPTCP:

  Pass 1 primitive / event: not specified

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

- SET_DSCP.TCP:

  Pass 1 primitive / event: not specified

  Parameters: DSCP value

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

- SET_DSCP.SCTP:

  Pass 1 primitive / event: ’Set DSCP value’

  Parameters: DSCP value

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

- SET_DSCP.UDP(-Lite):

  Pass 1 primitive / event: ’Set_DSCP’

  Parameter: DSCP value

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

- ERROR.TCP:

  Pass 1 primitive / event: ’Error_Report’

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

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

- ERROR.UDP(-Lite):

  Pass 1 primitive / event: ’Error_Report’

  Returns: Error report

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

- SET_AUTH.TCP:

  Pass 1 primitive / event: not specified

Parameters: current_key; rnext_key

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

- SET_AUTH.SCTP:

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

  Parameters: key_id; key; hmac_id

- GET_AUTH.TCP:

  Pass 1 primitive / event: not specified

  Parameters: current_key; rnext_key

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

- GET_AUTH.SCTP:

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

  Parameters: key_id; chunk_list

- RESET_STREAM.SCTP:

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

  Parameters: sid; direction

- RESET_STREAM-EVENT.SCTP:

  Pass 1 primitive / event: 'Stream Reset' notification

  Parameters: information about the result of RESET_STREAM.SCTP.

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

- RESET_ASSOC.SCTP:

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

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

- RESET_ASSOC-EVENT.SCTP:

  Pass 1 primitive / event: ’Association Reset’ notification

  Parameters: information about the result of RESET_ASSOC.SCTP.

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

- ADD_STREAM.SCTP:

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

  Parameters: number if outgoing and incoming streams to be added

- ADD_STREAM-EVENT.SCTP:

  Pass 1 primitive / event: ’Stream Change’ notification

  Parameters: information about the result of ADD_STREAM.SCTP.

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

- SET_STREAM_SCHEDULER.SCTP:

  Pass 1 primitive / event: ’Set Stream Scheduler’

  Parameters: scheduler identifier

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

- CONFIGURE_STREAM_SCHEDULER.SCTP:

  Pass 1 primitive / event: ’Configure Stream Scheduler’

  Parameters: priority

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

- SET_FLOWLABEL.SCTP:

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

  Parameters: flow label

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

- AUTHENTICATION_NOTIFICATION-EVENT.SCTP:

  Pass 1 primitive / event: ’Authentication’ notification

  Returns: information regarding key management.

- CONFIG_SEND_BUFFER.SCTP:

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

  Parameters: size value in octets

- CONFIG_RECEIVE_BUFFER.SCTP:

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

  Parameters: size value in octets

  Comments: this controls the receiver window.

- CONFIG_FRAGMENTATION.SCTP:

  Pass 1 primitive / event: ’Configure Message Fragmentation’

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

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

- CONFIG_PMTUD.SCTP:

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

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

  Returns: PMTU value

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

- CONFIG_DELAYED_SACK.SCTP:

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

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

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

- CONFIG_RTO.SCTP:

  Pass 1 primitive / event: ’Configure RTO Calculation’

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

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

- SET_COOKIE_LIFE.SCTP:

  Pass 1 primitive / event: ’Set Cookie Life Value’

  Parameters: cookie life value

- SET_MAX_BURST.SCTP:

  Pass 1 primitive / event: ’Set Maximum Burst’

  Parameters: max burst value

  Comments: not all implementations allow values above the default of 4.

- SET_PARTIAL_DELIVERY_POINT.SCTP:

  Pass 1 primitive / event: ’Set Partial Delivery Point’

  Parameters: partial delivery point (integer)

  Comments: this parameter must be smaller or equal to the socket receive buffer size.

- SET_CHECKSUM_ENABLED.UDP:

  Pass 1 primitive / event: ’Checksum_Enabled’.

  Parameters: 0 when zero checksum is used at sender, 1 for checksum at sender (default)

- SET_CHECKSUM_REQUIRED.UDP:

  Pass 1 primitive / event: ’Require_Checksum’.

  Parameter: 0 to allow zero checksum, 1 when a non-zero checksum is required (default) at receiver

- SET_CHECKSUM_COVERAGE.UDP-Lite:

  Pass 1 primitive / event: ’Set_Checksum_Coverage’

  Parameters: coverage length at sender (default maximum coverage)

- SET_MIN_CHECKSUM_COVERAGE.UDP-Lite:

  Pass 1 primitive / event: 'Set_Min_Coverage'.

  Parameter: coverage length at receiver (default minimum coverage)

- SET_DF.UDP(-Lite):

  Pass 1 primitive event: 'Set_DF'.

  Parameter: 0 when DF is not set (default) in the IPv4 header, 1 when DF is set

- GET_MMS_S.UDP(-Lite):

  Pass 1 primitive event: 'Get_MM_S'.

  Comments: this retrieves the maximum transport-message size that may be sent using a non-fragmented IP packet from the configured interface.

- GET_MMS_R.UDP(-Lite):

  Pass 1 primitive event: 'Get_MMS_R'.

  Comments: this retrieves the maximum transport-message size that may be received from the configured interface.

- SET_TTL.UDP(-Lite) (IPV6_UNICAST_HOPS):

  Pass 1 primitive / event: 'Set_TTL' and 'Set_IPV6_Unicast_Hops'

  Parameters: IPv4 TTL value or IPv6 Hop Count value

  Comments: this allows an application to change the IPv4 TTL of IPv6 Hop count value for outgoing UDP(-Lite) datagrams.

- GET_TTL.UDP(-Lite) (IPV6_UNICAST_HOPS):

  Pass 1 primitive / event: 'Get_TTL' and 'Get_IPV6_Unicast_Hops'

  Returns: IPv4 TTL value or IPv6 Hop Count value

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

- SET_ECN.UDP(-Lite):

  Pass 1 primitive / event: 'Set_ECN'

  Parameters: ECN value

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

- GET_ECN.UDP(-Lite):

  Pass 1 primitive / event: 'Get_ECN'

  Parameters: ECN value

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

- SET_IP_OPTIONS.UDP(-Lite):

  Pass 1 primitive / event: 'Set_IP_Options'

  Parameters: options

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

- GET_IP_OPTIONS.UDP(-Lite):

  Pass 1 primitive / event: 'Get_IP_Options'

  Returns: options

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

- CONFIGURE.LEDBAT:

  Pass 1 primitive / event: N/A

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

TERMINATION:

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

- CLOSE.TCP:

  Pass 1 primitive / event: 'Close'

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

- CLOSE.SCTP:

  Pass 1 primitive / event: 'Shutdown'

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

- ABORT.TCP:

  Pass 1 primitive / event: 'Abort'

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

- ABORT.SCTP:

  Pass 1 primitive / event: 'Abort'

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

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

- ABORT.UDP(-Lite):

  Pass 1 primitive event: ’Close’

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

- TIMEOUT.TCP:

  Pass 1 primitive / event: ’User Timeout’ event

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

- TIMEOUT.SCTP:

  Pass 1 primitive / event: ’Communication Lost’ event

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

- ABORT-EVENT.TCP:

  Pass 1 primitive / event: not specified.

- ABORT-EVENT.SCTP:

  Pass 1 primitive / event: ’Communication Lost’ event

  Returns: abort reason from the peer (if available)

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

- CLOSE-EVENT.TCP:

  Pass 1 primitive / event: not specified.

- CLOSE-EVENT.SCTP:

  Pass 1 primitive / event: ’Shutdown Complete’ event

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

4.2. DATA Transfer Related Primitives

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

- SEND.TCP:

  Pass 1 primitive / event: ’Send’

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

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

- SEND.SCTP:

  Pass 1 primitive / event: ’Send’

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

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

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

- SEND.UDP(-Lite):

  Pass 1 primitive / event: 'Send'

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

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

- RECEIVE.TCP:

  Pass 1 primitive / event: 'Receive'.

  Parameters: current_key (optional); rnext_key (optional)

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

- RECEIVE.SCTP:

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

  Parameters: stream number (optional)

  Returns: stream sequence number (optional); partial flag

(optional)

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

- RECEIVE.UDP(-Lite):

  Pass 1 primitive / event: 'Receive',

  Parameters: buffer for received datagram

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

- SENDFAILURE-EVENT.SCTP:

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

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

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

- SEND_FAILURE.UDP(-Lite):

  Pass 1 primitive / event: 'Send'

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

- SENDER_DRY-EVENT.SCTP:

  Pass 1 primitive / event: 'Sender Dry' notification

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

- PARTIAL_DELIVERY_ABORTED-EVENT.SCTP:

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

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

5. Pass 3

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

5.1. CONNECTION Related Transport Features

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

- Connect
  Protocols: TCP, SCTP, UDP(-Lite)

- Specify which IP Options must always be used
  Protocols: TCP, UDP(-Lite)

- Request multiple streams
  Protocols: SCTP

- Limit the number of inbound streams
  Protocols: SCTP

- Specify number of attempts and/or timeout for the first establishment message
  Protocols: TCP, SCTP

- Obtain multiple sockets
  Protocols: SCTP

- Disable MPTCP
  Protocols: MPTCP

- Configure authentication
  Protocols: TCP, SCTP
  Comments: with TCP, this allows to configure Master Key Tuples (MKTs). In SCTP, this allows to specify which chunk types must always be authenticated. DATA, ACK etc. are different 'chunks' in SCTP; one or more chunks may be included in a single packet.

- Indicate an Adaptation Layer (via an adaptation code point)
  Protocols: SCTP

- Request to negotiate interleaving of user messages
  Protocols: SCTP

- Hand over a message to reliably transfer (possibly multiple times) before connection establishment
  Protocols: TCP

- Hand over a message to reliably transfer during connection establishment
  Protocols: SCTP

- Enable UDP encapsulation with a specified remote UDP port number
  Protocols: SCTP

AVAILABILITY:
Preparing to receive incoming connection requests.

- Listen, 1 specified local interface
  Protocols: TCP, SCTP, UDP(-Lite)

- Listen, N specified local interfaces
  Protocols: SCTP

- Listen, all local interfaces
  Protocols: TCP, SCTP, UDP(-Lite)

- Obtain requested number of streams
  Protocols: SCTP

- Limit the number of inbound streams
  Protocols: SCTP

- Specify which IP Options must always be used
  Protocols: TCP, UDP(-Lite)

- Disable MPTCP
  Protocols: MPTCP

- Configure authentication
  Protocols: TCP, SCTP
  Comments: with TCP, this allows to configure Master Key Tuples (MKTs). In SCTP, this allows to specify which chunk types must always be authenticated. DATA, ACK etc. are different 'chunks' in SCTP; one or more chunks may be included in a single packet.

- Indicate an Adaptation Layer (via an adaptation code point)
  Protocols: SCTP

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

- Change timeout for aborting connection (using retransmit limit or time value)
  Protocols: TCP, SCTP

- Suggest timeout to the peer
  Protocols: TCP

- Disable Nagle algorithm
  Protocols: TCP, SCTP

- Request an immediate heartbeat, returning success/failure
  Protocols: SCTP

- Notification of Excessive Retransmissions (early warning below abortion threshold)
  Protocols: TCP

- Add path
  Protocols: MPTCP, SCTP
  MPTCP Parameters: source-IP; source-Port; destination-IP; destination-Port
  SCTP Parameters: local IP address

- Remove path
  Protocols: MPTCP, SCTP
  MPTCP Parameters: source-IP; source-Port; destination-IP; destination-Port
  SCTP Parameters: local IP address

- Set primary path
  Protocols: SCTP

- Suggest primary path to the peer
  Protocols: SCTP

- Configure Path Switchover
  Protocols: SCTP

- Obtain status (query or notification)
  Protocols: SCTP, MPTCP
  SCTP parameters: association connection state; destination transport address list; destination transport address reachability states; current local and peer receiver window sizes; current local congestion window sizes; number of unacknowledged DATA chunks; number of DATA chunks pending receipt; primary path; most recent SRTT on primary path; RTO on primary path; SRTT and RTO on other destination addresses; MTU per path; interleaving supported yes/no
  MPTCP parameters: subflow-list (identified by source-IP; source-Port; destination-IP; destination-Port)

- Specify DSCP field
  Protocols: TCP, SCTP, UDP(-Lite)

- Notification of ICMP error message arrival
  Protocols: TCP, UDP(-Lite)

- Change authentication parameters
  Protocols: TCP, SCTP

- Obtain authentication information
  Protocols: TCP, SCTP

- Reset Stream
  Protocols: SCTP

- Notification of Stream Reset
  Protocols: STCP

- Reset Association
  Protocols: SCTP

- Notification of Association Reset
  Protocols: STCP

- Add Streams
  Protocols: SCTP

- Notification of Added Stream
  Protocols: STCP

- Choose a scheduler to operate between streams of an association
  Protocols: SCTP

- Configure priority or weight for a scheduler
  Protocols: SCTP

- Specify IPv6 flow label field
  Protocols: SCTP

- Configure send buffer size
  Protocols: SCTP

- Configure receive buffer (and rwnd) size
  Protocols: SCTP

- Configure message fragmentation
  Protocols: SCTP

- Configure PMTUD
  Protocols: SCTP

- Configure delayed SACK timer
  Protocols: SCTP

- Set Cookie life value
  Protocols: SCTP

- Set maximum burst
  Protocols: SCTP

- Configure size where messages are broken up for partial delivery
  Protocols: SCTP

- Disable checksum when sending
  Protocols: UDP

- Disable checksum requirement when receiving
  Protocols: UDP

- Specify checksum coverage used by the sender
  Protocols: UDP-Lite

- Specify minimum checksum coverage required by receiver
  Protocols: UDP-Lite

- Specify DF field
  Protocols: UDP(-Lite)

- Get max. transport-message size that may be sent using a non-fragmented IP packet from the configured interface
  Protocols: UDP(-Lite)

- Get max. transport-message size that may be received from the configured interface
  Protocols: UDP(-Lite)

- Specify TTL/Hop count field
  Protocols: UDP(-Lite)

- Obtain TTL/Hop count field
  Protocols: UDP(-Lite)

- Specify ECN field
  Protocols: UDP(-Lite)

- Obtain ECN field
  Protocols: UDP(-Lite)

- Specify IP Options
  Protocols: UDP(-Lite)

- Obtain IP Options
  Protocols: UDP(-Lite)

- Enable and configure "Low Extra Delay Background Transfer"
  Protocols: A protocol implementing the LEDBAT congestion control mechanism

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

- Close after reliably delivering all remaining data, causing an event informing the application on the other side
  Protocols: TCP, SCTP
  Comments: a TCP endpoint locally only closes the connection for sending; it may still receive data afterwards.

- Abort without delivering remaining data, causing an event informing the application on the other side
  Protocols: TCP, SCTP
  Comments: in SCTP a reason can optionally be given by the application on the aborting side, which can then be received by the application on the other side.

- Abort without delivering remaining data, not causing an event informing the application on the other side
  Protocols: UDP(-Lite)

- Timeout event when data could not be delivered for too long
  Protocols: TCP, SCTP
  Comments: the timeout is configured with CONNECTION.MAINTENANCE "Change timeout for aborting connection (using retransmit limit or time value)".

5.2. DATA Transfer Related Transport Features

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

5.2.1. Sending Data

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

- Reliably transfer data, with congestion control
  Protocols: TCP

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

- Unreliably transfer a message, with congestion control
  Protocols: SCTP

- Unreliably transfer a message, without congestion control
  Protocols: UDP(-Lite)

- Configurable Message Reliability
  Protocols: SCTP

- Choice of stream
  Protocols: SCTP

- Choice of path (destination address)
  Protocols: SCTP

- Ordered message delivery (potentially slower than unordered)
  Protocols: SCTP

- Unordered message delivery (potentially faster than ordered)
  Protocols: SCTP, UDP(-Lite)

- Request not to bundle messages
  Protocols: SCTP

- Specifying a "payload protocol-id" (handed over as such by the receiver)
  Protocols: SCTP

- Specifying a key id to be used to authenticate a message
  Protocols: SCTP

- Request not to delay the acknowledgement (SACK) of a message
  Protocols: SCTP

5.2.2. Receiving Data

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

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

- Receive a message
  Protocols: SCTP, UDP(-Lite)

- Choice of stream to receive from
  Protocols: SCTP

- Information about partial message arrival
  Protocols: SCTP
  Comments: in SCTP, partial messages are combined with a stream sequence number so that the application can restore the correct order of data blocks an entire message consists of.

5.2.3. Errors

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

- Notification of an unsent (part of a) message
  Protocols: SCTP, UDP(-Lite)

- Notification of an unacknowledged (part of a) message
  Protocols: SCTP

- Notification that the stack has no more user data to send
  Protocols: SCTP

- Notification to a receiver that a partial message delivery has been aborted
  Protocols: SCTP

6. Acknowledgements

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

7. IANA Considerations

This memo includes no request to IANA.

8. Security Considerations

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

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

9. References

9.1. Normative References

[FJ16] Fairhurst, G. and T. Jones, "Features of the User Datagram Protocol (UDP) and Lightweight UDP (UDP-Lite) Transport Protocols", draft-ietf-taps-transports-usage-udp-04 (work in progress), July 2017.

[I-D.ietf-tsvwg-sctp-ndata] Stewart, R., Tuexen, M., Loreto, S., and R. Seggelmann, "Stream Schedulers and User Message Interleaving for the Stream Control Transmission Protocol", draft-ietf-tsvwg-sctp-ndata-08 (work in progress), October 2016.

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

[RFC1122] Braden, R., Ed., "Requirements for Internet Hosts - Communication Layers", STD 3, RFC 1122, DOI 10.17487/RFC1122, October 1989, <https://www.rfc-editor.org/info/rfc1122>.

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

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

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

[RFC5061] Stewart, R., Xie, Q., Tuexen, M., Maruyama, S., and M. Kozuka, "Stream Control Transmission Protocol (SCTP) Dynamic Address Reconfiguration", RFC 5061, DOI 10.17487/RFC5061, September 2007, <https://www.rfc-editor.org/info/rfc5061>.

[RFC5482] Eggert, L. and F. Gont, "TCP User Timeout Option", RFC 5482, DOI 10.17487/RFC5482, March 2009, <https://www.rfc-editor.org/info/rfc5482>.

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

[RFC6182] Ford, A., Raiciu, C., Handley, M., Barre, S., and J. Iyengar, "Architectural Guidelines for Multipath TCP Development", RFC 6182, DOI 10.17487/RFC6182, March 2011, <https://www.rfc-editor.org/info/rfc6182>.

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

[RFC6525] Stewart, R., Tuexen, M., and P. Lei, "Stream Control Transmission Protocol (SCTP) Stream Reconfiguration", RFC 6525, DOI 10.17487/RFC6525, February 2012, <https://www.rfc-editor.org/info/rfc6525>.

[RFC6817] Shalunov, S., Hazel, G., Iyengar, J., and M. Kuehlewind, "Low Extra Delay Background Transport (LEDBAT)", RFC 6817, DOI 10.17487/RFC6817, December 2012, <https://www.rfc-editor.org/info/rfc6817>.

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

[RFC6897] Scharf, M. and A. Ford, "Multipath TCP (MPTCP) Application Interface Considerations", RFC 6897, DOI 10.17487/RFC6897, March 2013, <https://www.rfc-editor.org/info/rfc6897>.

[RFC6951] Tuexen, M. and R. Stewart, "UDP Encapsulation of Stream Control Transmission Protocol (SCTP) Packets for End-Host to End-Host Communication", RFC 6951, DOI 10.17487/RFC6951, May 2013, <https://www.rfc-editor.org/info/rfc6951>.

[RFC7053] Tuexen, M., Ruengeler, I., and R. Stewart, "SACK-IMMEDIATELY Extension for the Stream Control Transmission Protocol", RFC 7053, DOI 10.17487/RFC7053, November 2013, <https://www.rfc-editor.org/info/rfc7053>.

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

[RFC7496] Tuexen, M., Seggelmann, R., Stewart, R., and S. Loreto, "Additional Policies for the Partially Reliable Stream Control Transmission Protocol Extension", RFC 7496, DOI 10.17487/RFC7496, April 2015, <https://www.rfc-editor.org/info/rfc7496>.

[RFC7829] Nishida, Y., Natarajan, P., Caro, A., Amer, P., and K. Nielsen, "SCTP-PF: A Quick Failover Algorithm for the Stream Control Transmission Protocol", RFC 7829, DOI 10.17487/RFC7829, April 2016, <https://www.rfc-editor.org/info/rfc7829>.

[RFC8085] Eggert, L., Fairhurst, G., and G. Shepherd, "UDP Usage Guidelines", BCP 145, RFC 8085, DOI 10.17487/RFC8085, March 2017, <https://www.rfc-editor.org/info/rfc8085>.

9.2. Informative References

[I-D.draft-gjessing-taps-minset] Gjessing, S. and M. Welzl, "A Minimal Set of Transport Services for TAPS Systems", draft-gjessing-taps-minset-05 (work in progress), June 2017.

[RFC0854] Postel, J. and J. Reynolds, "Telnet Protocol Specification", STD 8, RFC 854, DOI 10.17487/RFC0854, May 1983, <https://www.rfc-editor.org/info/rfc854>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/

RFC2119, March 1997,
<https://www.rfc-editor.org/info/rfc2119>.

[RFC2474] Nichols, K., Blake, S., Baker, F., and D. Black, "Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers", RFC 2474, DOI 10.17487/RFC2474, December 1998, <https://www.rfc-editor.org/info/rfc2474>.

[RFC2475] Blake, S., Black, D., Carlson, M., Davies, E., Wang, Z., and W. Weiss, "An Architecture for Differentiated Services", RFC 2475, DOI 10.17487/RFC2475, December 1998, <https://www.rfc-editor.org/info/rfc2475>.

[RFC3260] Grossman, D., "New Terminology and Clarifications for Diffserv", RFC 3260, DOI 10.17487/RFC3260, April 2002, <https://www.rfc-editor.org/info/rfc3260>.

[RFC5461] Gont, F., "TCP's Reaction to Soft Errors", RFC 5461, DOI 10.17487/RFC5461, February 2009, <https://www.rfc-editor.org/info/rfc5461>.

[RFC6093] Gont, F. and A. Yourtchenko, "On the Implementation of the TCP Urgent Mechanism", RFC 6093, DOI 10.17487/RFC6093, January 2011, <https://www.rfc-editor.org/info/rfc6093>.

[RFC7414] Duke, M., Braden, R., Eddy, W., Blanton, E., and A. Zimmermann, "A Roadmap for Transmission Control Protocol (TCP) Specification Documents", RFC 7414, DOI 10.17487/RFC7414, February 2015, <https://www.rfc-editor.org/info/rfc7414>.

[RFC7657] Black, D., Ed. and P. Jones, "Differentiated Services (Diffserv) and Real-Time Communication", RFC 7657, DOI 10.17487/RFC7657, November 2015, <https://www.rfc-editor.org/info/rfc7657>.

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

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

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

Appendix B. How this document was developed

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

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

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

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

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

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

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

For each primitive, the following style was used:

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

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

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

Appendix C. Revision information

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

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

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

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

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

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

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

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

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

-07: addressed AD review comments from Spencer Dawkins.

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

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

Authors' Addresses

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

Email: michawe@ifi.uio.no

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

Email: tuexen@fh-muenster.de

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

Email: naeemk@ifi.uio.no

TAPS Working Group
Internet-Draft
Intended status: Informational
Expires: September 9, 2017

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

March 08, 2017

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

Abstract

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

Status of This Memo

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

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

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

This Internet-Draft will expire on September 9, 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. Abstractions and Terminology 5
2.1. Message Carrier 6
2.1.1. Listener 7
2.1.2. Source 8
2.1.3. Sink 8
2.1.4. Responder 8
2.1.5. Stream 8
2.2. Message 8
2.2.1. Lifetime and Partial Reliability 9
2.2.2. Priority 10
2.2.3. Dependence 10
2.2.4. Idempotence 10
2.2.5. Immediacy 10
2.2.6. Additional Events 10
2.3. Association 11
2.4. Remote 11
2.5. Local 12
2.6. Transient 12
2.7. Path 12
2.8. Policy Context 13
3. Abstract Programming Interface 14
3.1. Example Connection Patterns 15
3.1.1. Client-Server 15
3.1.2. Client-Server with Happy Eyeballs and 0-RTT establishment 16
3.1.3. Peer to Peer with Network Address Translation 17
3.1.4. Multicast Receiver 17
3.2. Implementation Considerations 17
3.2.1. Message Framing and Deframing 18
3.2.2. Message Size Limitations 18
3.2.3. Backpressure 18
4. Acknowledgments 19
5. References 19
5.1. Normative References 19
5.2. Informative References 19

Appendix A. API sketch in Golang 21
Authors' Addresses 25

1. Introduction

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

In the meantime, the nature of Internet access, and the variety of Internet transport protocols, is evolving. The challenges that new protocols and access paradigms present to the sockets API and to programming models based on them inspire the design elements of a new approach

Many end-user devices are connected to the Internet via multiple interfaces, which suggests it is time to promote the paths by which two endpoints are connected to each other to a first-order object. While implicit multipath communication is available for these multihomed nodes in the present Internet architecture with the Multipath TCP extension (MPTCP) [RFC6824], MPTCP was specifically designed to hide multipath communication from the application for purposes of compatibility. Since many multihomed nodes are connected to the Internet through access paths with widely different properties with respect to bandwidth, latency and cost, adding explicit path control to MPTCP's API would be useful in many situations. Applications also need control over cooperation with path elements via mechanisms such as that proposed by the Path Layer UDP Substrate (PLUS) effort (see [I-D.trammell-plus-statefulness] and [I-D.trammell-plus-abstract-mech]).

Another trend straining the traditional layering of the transport stack associated with the SOCK_STREAM interface is the widespread interest in ubiquitous deployment of encryption to guarantee confidentiality, authenticity, and integrity, in the face of pervasive surveillance [RFC7258]. Layering the most widely deployed encryption technology, Transport Layer Security (TLS), strictly atop TCP (i.e., via a TLS library such as OpenSSL that uses the sockets API) requires the encryption-layer handshake to happen after the transport-layer handshake, which increases connection setup latency

on the order of one or two round-trip times, an unacceptable delay for many applications. Integrating cryptographic state setup and maintenance into the path abstraction naturally complements efforts in new protocols (e.g. QUIC [I-D.ietf-quic-transport]) to mitigate this strict layering.

To meet these challenges, we present the Post-Socket Application Programming Interface (API), described in detail in this work. Post is designed to be language, transport protocol, and architecture independent, allowing applications to be written to a common abstract interface, easily ported among different platforms, and used even in environments where transport protocol selection may be done dynamically, as proposed in the IETF's Transport Services working group.

Post replaces the traditional SOCK_STREAM abstraction with an Message abstraction, which can be seen as a generalization of the Stream Control Transmission Protocol's [RFC4960] SOCK_SEQPACKET service. Messages are sent and received on Carriers, which logically group Messages for transmission and reception. For backward compatibility, these Carriers can also be opened as Streams, presenting a file-like interface to the network as with SOCK_STREAM.

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

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

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

to explicitly take care of connection establishment mechanics such as connection racing [RFC6555] and peer-to-peer rendezvous [RFC5245].

- Transport protocol stack independence, allowing applications to be written in terms of the semantics best for the application's own design, separate from the protocol(s) used on the wire to achieve them. This enables applications written to a single API to make use of transport protocols in terms of the features they provide, as in [I-D.ietf-taps-transports].

This work is the synthesis of many years of Internet transport protocol research and development. It is inspired by concepts from the Stream Control Transmission Protocol (SCTP) [RFC4960], TCP Minion [I-D.iyengar-minion-protocol], and MinimaLT[MinimaLT], among other transport protocol modernization efforts. We present Post Sockets as an illustration of what is possible with present developments in transport protocols when freed from the strictures of the current sockets API. While much of the work for building parts of the protocols needed to implement Post are already ongoing in other IETF working groups (e.g. MPTCP, QUIC, TLS), we argue that an abstract programming interface unifying access all these efforts is necessary to fully exploit their potential.

2. Abstractions and Terminology

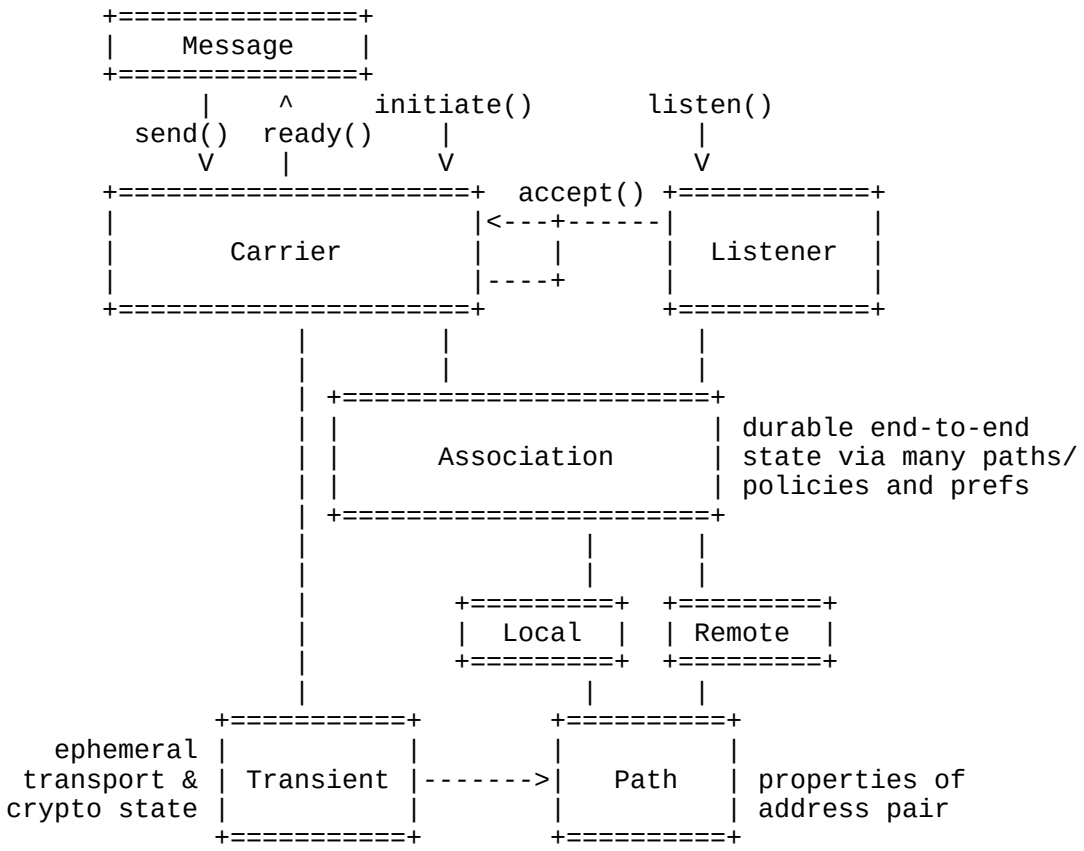


Figure 1: Abstractions and relationships in Post Sockets

Post is based on a small set of abstractions, centered around a Message Carrier as the entry point for an application to the networking API. The relationships among them are shown in Figure Figure 1 and detailed in this section.

2.1. Message Carrier

A Message Carrier (or simply Carrier) is a transport protocol stack-independent interface for sending and receiving messages between an application and a remote endpoint; it is roughly analogous to a socket in the present sockets API.

Sending a Message over a Carrier is driven by the application, while receipt is driven by the arrival of the last packet that allows the Message to be assembled, decrypted, and passed to the application.

Receipt is therefore asynchronous; given the different models for asynchronous I/O and concurrency supported by different platforms, it may be implemented in any number of ways. The abstract API provides only for a way for the application to register how it wants to handle incoming messages.

All the Messages sent to a Message Carrier will be received on the corresponding Message Carrier at the remote endpoint, though not necessarily reliably or in order, depending on Message properties and the underlying transport protocol stack.

A Message Carrier that is backed by current transport protocol stack state (such as a TCP connection; see Section 2.6) is said to be "active": messages can be sent and received over it. A Message Carrier can also be "dormant": there is long-term state associated with it (via the underlying Association; see Section 2.3), and it may be able to reactivated, but messages cannot be sent and received immediately.

If supported by the underlying transport protocol stack, a Message Carrier may be forked: creating a new Message Carrier associated with a new Message Carrier at the same remote endpoint. The semantics of the usage of multiple Message Carriers based on the same Association are application-specific. When a Message Carrier is forked, its corresponding Message Carrier at the remote endpoint receives a fork request, which it must accept in order to fully establish the new carrier. Multiple message carriers between endpoints are implemented differently by different transport protocol stacks, either using multiple separate transport-layer connections, or using multiple streams of multistreaming transport protocols.

To exchange messages with a given remote endpoint, an application may initiate a Message Carrier given its remote (see Section 2.4 and local (see Section 2.5) identities; this is an equivalent to an active open. There are five special cases of Message Carriers, as well, supporting different initiation and interaction patterns, defined in the subsections below.

2.1.1. Listener

A Listener is a special case of Message Carrier which only responds to requests to create a new Carrier from a remote endpoint, analogous to a server or listening socket in the present sockets API. Instead of being bound to a specific remote endpoint, it is bound only to a local identity; however, its interface for accepting fork requests is identical to that for fully fledged Message Carriers.

2.1.2. Source

A Source is a special case of Message Carrier over which messages can only be sent, intended for unidirectional applications such as multicast transmitters. Sources cannot be forked, and need not accept forks.

2.1.3. Sink

A Sink is a special case of Message Carrier over which messages can only be received, intended for unidirectional applications such as multicast receivers. Sinks cannot be forked, and need not accept forks.

2.1.4. Responder

A Responder is a special case of Message Carrier which may receive messages from many remote sources, for cases in which an application will only ever send Messages in reply back to the source from which a Message was received. This is a common implementation pattern for servers in client-server applications. A Responder’s receiver gets a Message, as well as a Source to send replies to. Responders cannot be forked, and need not accept forks.

2.1.5. Stream

A Message Carrier may be irreversibly morphed into a Stream, in order to provide a strictly ordered, reliable service as with SOCK_STREAM. Morphing a Message Carrier into a Stream should return a "file-like object" as appropriate for the platform implementing the API. Typically, both ends of a communication using a stream service will morph their respective Message Carriers independently before sending any Messages.

Writing a byte to a Stream will cause it to be received by the remote, in order, or will cause an error condition and termination of the stream if the byte cannot be delivered. Due to the strong sequential dependence on a stream, streams must always be reliable and ordered. A Message Carrier may only be morphed to a Stream if it uses transport protocol stack that provides reliable, ordered service, and only before it is used to send a Message.

2.2. Message

A Message is an atomic unit of communication between applications. A Message that cannot be delivered in its entirety within the constraints of the network connectivity and the requirements of the application is not delivered at all.

Messages can represent both relatively small structures, such as requests in a request/response protocol such as HTTP; as well as relatively large structures, such as files of arbitrary size in a filesystem.

In the general case, there is no mapping between a Message and packets sent by the underlying protocol stack on the wire: the transport protocol may freely segment messages and/or combine messages into packets. However, a message may be marked as immediate, which will cause it to be sent in a single packet, if it will fit.

This implies that both the sending and receiving endpoint, whether in the application layer or the transport layer, must guarantee storage for the full size of an Message.

Messages are sent over and received from Message Carriers (see Section 2.1).

On sending, Messages have properties that allow the application to specify its requirements with respect to reliability, ordering, priority, idempotence, and immediacy; these are described in detail below. Messages may also have arbitrary properties which provide additional information to the underlying transport protocol stack on how they should be handled, in a protocol-specific way. These stacks may also deliver or set properties on received messages, but in the general case a received messages contains only a sequence of ordered bytes.

2.2.1. Lifetime and Partial Reliability

A Message may have a "lifetime" - a wallclock duration before which the Message must be available to the application layer at the remote end. If a lifetime cannot be met, the Message is discarded as soon as possible. Messages without lifetimes are sent reliably if supported by the transport protocol stack. Lifetimes are also used to prioritize Message delivery.

There is no guarantee that a Message will not be delivered after the end of its lifetime; for example, a Message delivered over a strictly reliable transport will be delivered regardless of its lifetime. Depending on the transport protocol stack used to transmit the message, these lifetimes may also be signaled to path elements by the underlying transport, so that path elements that realize a lifetime cannot be met can discard frames containing the Messages instead of forwarding them.

2.2.2. Priority

Messages have a "niceness" - a priority among other messages sent over the same Message Carrier in an unbounded hierarchy most naturally represented as a non-negative integer. By default, Messages are in niceness class 0, or highest priority. Niceness class 1 Messages will yield to niceness class 0 Messages sent over the same Carrier, class 2 to class 1, and so on. Niceness may be translated to a priority signal for exposure to path elements (e.g. DSCP codepoint) to allow prioritization along the path as well as at the sender and receiver. This inversion of normal schemes for expressing priority has a convenient property: priority increases as both niceness and lifetime decrease. A Message may have both a niceness and a lifetime - Messages with higher niceness classes will yield to lower classes if resource constraints mean only one can meet the lifetime.

2.2.3. Dependence

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

2.2.4. Idempotence

A sending application may mark a Message as "idempotent" to signal to the underlying transport protocol stack that its application semantics make it safe to send in situations that may cause it to be received more than once (i.e., for 0-RTT session resumption as in TCP Fast Open, TLS 1.3, and QUIC).

2.2.5. Immediacy

A sending application may mark a Message as "immediate" to signal to the underlying transport protocol stack that its application semantics require it to be placed in a single packet, on its own, instead of waiting to be combined with other messages or parts thereof (i.e., for media transports and interactive sessions with small messages).

2.2.6. Additional Events

Senders may also be asynchronously notified of three events on Messages they have sent: that the Message has been transmitted, that the Message has been acknowledged by the receiver, or that the

Message has expired before transmission/acknowledgment. Not all transport protocol stacks will support all of these events.

2.3. Association

An Association contains the long-term state necessary to support communications between a Local (see Section 2.5) and a Remote (see Section 2.4) endpoint, such as cryptographic session resumption parameters or rendezvous information; information about the policies constraining the selection of transport protocols and local interfaces to create Transients (see Section 2.6) to carry Messages; and information about the paths through the network available available between them (see Section 2.7).

All Message Carriers are bound to an Association. New Message Carriers will reuse an Association if they can be carried from the same Local to the same Remote over the same Paths; this re-use of an Association may implies the creation of a new Transient.

2.4. Remote

A Remote represents information required to establish and maintain a connection with the far end of an Association: name(s), address(es), and transport protocol parameters that can be used to establish a Transient; transport protocols to use; information about public keys or certificate authorities used to identify the remote on connection establishment; and so on. Each Association is associated with a single Remote, either explicitly by the application (when created by the initiation of a Message Carrier) or a Listener (when created by forking a Message Carrier on passive open).

A Remote may be resolved, which results in zero or more Remotes with more specific information. For example, an application may want to establish a connection to a website identified by a URL https://www.example.com. This URL would be wrapped in a Remote and passed to a call to initiate a Message Carrier. The first pass resolution might parse the URL, decomposing it into a name, a transport port, and a transport protocol to try connecting with. A second pass resolution would then look up network-layer addresses associated with that name through DNS, and store any certificates available from DANE. Once a Remote has been resolved to the point that a transport protocol stack can use it to create a Transient, it is considered fully resolved.

2.5. Local

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

2.6. Transient

A Transient represents a binding between a Message Carrier and the instance of the transport protocol stack that implements it. As an Association contains long-term state for communications between two endpoints, a Transient contains ephemeral state for a single transport protocol over a single Path at a given point in time.

A Message Carrier may be served by multiple Transients at once, e.g. when implementing multipath communication such that the separate paths are exposed to the API by the underlying transport protocol stack. Each Transient serves only one Message Carrier, although multiple Transients may share the same underlying protocol stack; e.g. when multiplexing Carriers over streams in a multistreaming protocol.

Transients are generally not exposed by the API to the application, though they may be accessible for debugging and logging purposes.

2.7. Path

A Path represents information about a single path through the network used by an Association, in terms of source and destination network and transport layer addresses within an addressing context, and the provisioning domain [RFC7556] of the local interface. This information may be learned through a resolution, discovery, or rendezvous process (e.g. DNS, ICE), by measurements taken by the transport protocol stack, or by some other path information discovery mechanism. It is used by the transport protocol stack to maintain and/or (re-)establish communications for the Association.

The set of available properties is a function of the transport protocol stacks in use by an association. However, the following core properties are generally useful for applications and transport layer protocols to choose among paths for specific Messages:

- Maximum Transmission Unit (MTU): the maximum size of an Message's payload (subtracting transport, network, and link layer overhead) which will likely fit into a single frame. Derived from signals sent by path elements, where available, and/or path MTU discovery processes run by the transport layer.

- Latency Expectation: expected one-way delay along the Path. Generally provided by inline measurements performed by the transport layer, as opposed to signaled by path elements.

- Loss Probability Expectation: expected probability of a loss of any given single frame along the Path. Generally provided by inline measurements performed by the transport layer, as opposed to signaled by path elements.

- Available Data Rate Expectation: expected maximum data rate along the Path. May be derived from passive measurements by the transport layer, or from signals from path elements.

- Reserved Data Rate: Committed, reserved data rate for the given Association along the Path. Requires a bandwidth reservation service in the underlying transport protocol stack.

- Path Element Membership: Identifiers for some or all nodes along the path, depending on the capabilities of the underlying network layer protocol to provide this.

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

2.8. Policy Context

A Local and a Remote is not necessarily enough to establish a Message Carrier between two endpoints. For instance, an application may require or prefer certain transport features (see [I-D.ietf-taps-transports]) in the transport protocol stacks used by the Transients underlying the Carrier; it may also prefer Paths over one interface to those over another (e.g. WiFi access over LTE when roaming on a foreign LTE network, due to cost). These policies are expressed in a Policy Context bound to an Association. Multiple policy contexts may be active at once; e.g. a system Policy Context expressing administrative preferences about interface and protocol selection, an application Policy Context expressing transport feature information. The expression of policy contexts and the resolution of conflicts among Policy Contexts is currently implementation-specific;

note that these are equivalent to the Policy API in the NEAT architeture [NEAT].

3. Abstract Programming Interface

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

- Flexibility is paramount. So is simplicity. Applications must be given as many controls and as much information as they may need, but they must be able to ignore controls and information irrelevant to their operation. This implies that the "default" interface must be no more complicated than BSD sockets, and must do something reasonable.

- Reception is an inherently asynchronous activity. While the API is designed to be as platform-independent as possible, one key insight it is based on is that an Message receiver's behavior in a packet-switched network is inherently asynchronous, driven by the receipt of packets, and that this asynchronicity must be reflected in the API. The actual implementation of receive and event handling will need to be aligned to the method a given platform provides for asynchronous I/O.

- A new API cannot be bound to a single transport protocol and expect wide deployment. As the API is transport-independent and may support runtime transport selection, it must impose the minimum possible set of constraints on its underlying transports, though some API features may require underlying transport features to work optimally. It must be possible to implement Post over vanilla TCP in the present Internet architecture.

The API we design from these principles is centered around a Carrier, which can be created actively via initiate() or passively via a listen(); the latter creates a Listener from which new Carriers can be accept()ed. Messages may be created explicitly and passed to this Carrier, or implicitly through a simplified interface which uses default message properties (reliable transport without priority or deadline, which guarantees ordered delivery over a single Carrier when the underlying transport protocol stack supports it).

The current state of API development is illustrated as a set of interfaces and function prototypes in the Go programming language in Appendix A; future revisions of this document will give more a more abstract specification of the API as development completes.

3.1. Example Connection Patterns

Here, we illustrate the usage of the API outlined in Appendix A for common connection patterns. Note that error handling is ignored in these illustrations for ease of reading.

3.1.1. Client-Server

Here's an example client-server application. The server echoes messages. The client sends a message and prints what it receives.

The client in Figure 2 connects, sends a message, and sets up a receiver to print messages received in response. The carrier is inactive after the Initiate() call; the Send() call blocks until the carrier can be activated.

```
// connect to a server given a remote
func sayHello() {

    carrier := Initiate(local, remote)

    carrier.Send([]byte("Hello!"))
    carrier.Ready(func (msg InMessage) {
        fmt.Println(string([]byte(msg))
        return false
    })
    carrier.Close()
}
```

Figure 2: Example client

The server in Figure 3 creates a Listener, which accepts Carriers and passes them to a server. The server echos the content of each message it receives.

```
// run a server for a specific carrier, echo all its messages
func runMyServerOn(carrier Carrier) {
    carrier.Ready(func (msg InMessage) {
        carrier.Send(msg)
    })
}

// accept connections forever, spawn servers for them
func acceptConnections() {
    listener := Listen(local)
    listener.Accept(func(carrier Carrier) bool {
        go runMyServerOn(carrier)
        return true
    })
}
```

Figure 3: Example server

The Responder allows the server to be significantly simplified, as shown in Figure 4.

```
func echo(msg InMessage, reply Sink) {
    reply.Send(msg)
}

Respond(local, echo)
```

Figure 4: Example responder

3.1.2. Client-Server with Happy Eyeballs and 0-RTT establishment

The fundamental design of a client need not change at all for happy eyeballs [RFC6555] (selection of multiple potential protocol stacks through connection racing); this is handled by the Post Sockets implementation automatically. If this connection racing is to use 0-RTT data (i.e., as provided by TCP Fast Open [RFC7413], the client must mark the outgoing message as idempotent.

```
// connect to a server given a remote
func sayHelloQuickly() {

    carrier := Initiate(local, remote)

    carrier.SendMsg(OutMessage{Content: []byte("Hello!"), Idempotent: true}, nil
, nil, nil)
    carrier.Ready(func (msg InMessage) {
        fmt.Println(string([]byte(msg)))
        return false
    })
    carrier.Close()
}
```

3.1.3. Peer to Peer with Network Address Translation

In the client-server examples shown above, the Remote given to the Initiate call refers to the name and port of the server to connect to. This need not be the case, however; a Remote may also refer to an identity and a rendezvous point for rendezvous as in ICE [RFC5245]. Here, each peer does its own Initiate call simultaneously, and the result on each side is a Carrier attached to an appropriate Association.

3.1.4. Multicast Receiver

A multicast receiver is implemented using a Sink attached to a Local encapsulating a multicast address on which to receive multicast datagrams. The following example prints messages received on the multicast address forever.

```
func receiveMulticast() {
    sink = NewSink(local)
    sink.Ready(func (msg InMessage) {
        fmt.Println(string([]byte(msg)))
        return true
    })
}
```

3.2. Implementation Considerations

Here we discuss an incomplete list of API implementation considerations that have arisen with experimentation with the prototype in Appendix A.

3.2.1. Message Framing and Deframing

An obvious goal of Post Sockets is interoperability with non-Post Sockets endpoints: a Post Sockets endpoint using a given protocol stack must be able to communicate with another endpoint using the same protocol stack, but not using Post Sockets. This implies that the underlying transport protocol stack must support object framing, in order to delimit Messages carried by protocol stacks that are not themselves message-oriented.

Another goal of Post Sockets is to work over unmodified TCP. We could simply define a Message Carrier over TCP to support only stream morphing, but this would fall far short of our goal to transport independence. Another approach is to recognize that almost every protocol using TCP already has its own message delimiters, and to allow the receiver of a Message to provide a deframing primitive to the API. Experimentation with the best way to achieve this within Post Sockets is underway.

3.2.2. Message Size Limitations

Ideally, Messages can be of infinite size. However, protocol stacks and protocol stack implementations may impose their own limits on message sizing; For example, SCTP [RFC4960] and TLS [I-D.ietf-tls-tls13] impose record size limitations of 64kB and 16kB, respectively. Message sizes may also be limited by the available buffer at the receiver, since a Message must be fully assembled by the transport layer before it can be passed on to the application layer. Since not every transport protocol stack implements the signaling necessary to negotiate or expose message size limitations, these are currently configured out of band, and are probably best exposed through the policy context.

A truly infinite message service - e.g. large file transfer where both endpoints have committed persistent storage to the message - is probably best realized as a layer above Post Sockets, and may be added as a new type of Message Carrier to a future revision of this document.

3.2.3. Backpressure

Regardless of how asynchronous reception is implemented, it is important for an application to be able to apply receiver backpressure, to allow the protocol stack to perform receiver flow control. Depending on how asynchronous I/O works in the platform, this could be implemented by having a maximum number of concurrent receive callbacks, for example.

4. Acknowledgments

Many thanks to Laurent Chuat and Jason Lee at the Network Security Group at ETH Zurich for contributions to the initial design of Post Sockets. Thanks to Joe Hildebrand, Martin Thomson, and Michael Welzl for their feedback, as well as the attendees of the Post Sockets workshop in February 2017 in Zurich for the discussions, which have improved the design described herein.

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

5. References

5.1. Normative References

[I-D.ietf-taps-transports]
Fairhurst, G., Trammell, B., and M. Kuehlewind, "Services provided by IETF transport protocols and congestion control mechanisms", draft-ietf-taps-transports-14 (work in progress), December 2016.

5.2. Informative References

[I-D.ietf-quic-transport]
Iyengar, J. and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport", draft-ietf-quic-transport-01 (work in progress), January 2017.

[I-D.ietf-tls-tls13]
Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", draft-ietf-tls-tls13-18 (work in progress), October 2016.

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

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

[I-D.trammell-plus-statefulness]
Kuehlewind, M., Trammell, B., and J. Hildebrand, "Transport-Independent Path Layer State Management", draft-trammell-plus-statefulness-02 (work in progress), December 2016.

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

[NEAT] Grinnemo, K-J., Tom Jones, ., Gorry Fairhurst, ., David Ros, ., Anna Brunstrom, ., and . Per Hurtig, "Towards a Flexible Internet Transport Layer Architecture", June 2016.

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

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

[RFC5245] Rosenberg, J., "Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols", RFC 5245, DOI 10.17487/RFC5245, April 2010, <http://www.rfc-editor.org/info/rfc5245>.

[RFC6555] Wing, D. and A. Yourtchenko, "Happy Eyeballs: Success with Dual-Stack Hosts", RFC 6555, DOI 10.17487/RFC6555, April 2012, <http://www.rfc-editor.org/info/rfc6555>.

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

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

[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>.

[RFC7556] Anipko, D., Ed., "Multiple Provisioning Domain Architecture", RFC 7556, DOI 10.17487/RFC7556, June 2015, <http://www.rfc-editor.org/info/rfc7556>.

Appendix A. API sketch in Golang

The following sketch is a snapshot of an API currently under development in Go, available at https://github.com/mami-project/postsocket. The details of the API are still under development; once the API definition stabilizes, this will be expanded into prose in a future revision of this draft.

```
// The interface to path information is TBD
type Path interface{}

// An association encapsulates an endpoint pair and the set of paths between the
m.
type Association interface {
    Local() Local
    Remote() Remote
    Paths() []Path
}

// A message together with with metadata needed to send it
type OutMessage struct {
    // The content of this message, as a byte array
    Content []byte
    // The niceness of this message. 0 is highest priority.
    Niceness uint
    // The lifetime of this message. After this duration, the message may expire
.
    Lifetime time.Duration
    // Pointers to messages that must be sent before this one.
    Antecedent []*OutMessage
    // True if the message is safe to send such that it may be received multiple
 times (i.e. for 0-RTT).
    Idempotent bool
}

// A message received from a stream
type InMessage []byte

// A Carrier is a transport protocol stack-independent interface for sending and
// receiving messages between an application and a remote endpoint; it is roughl
y
// analogous to a socket in the present sockets API.
type Carrier interface {
    // Send a byte array on this Carrier as a message with default metadata
    // and no notifications.
    Send(buf []byte) error

    // Send a message on this Carrier. The optional onSent function will be
```

```
    // called when the protocol stack instance has sent the message. The
    // optional onAcked function will be called when the receiver has
    // acknowledged the message. The optional onExpired function will be
    // called if the message's lifetime expired before the message coult be
    // sent. If the Carrier is not active, attempt to activate the Carrier
    // before sending.
    Sendmsg(msg *OutMessage, onSent func(), onAcked func(), onExpired func()) er
ror

    // Signal that an application is ready to receive messages via a given callb
ack.
    // Messages will be given to the callback until it returns false, or until t
he
    // Carrier is closed.
    Ready(receive func(InMessage) bool) error

    // Retrieve the Association over which this Carrier is running.
    Association() *Association

    // Retrieve the active Transients over which this carrier is running, if act
ive.
    Transients() []Transient

    // Determine whether the Carrier is currently active
    IsActive() bool

    // Ensure that the Carrier is active and ready to send and receive messages.
    // Attempts to bring up at least one Transient.
    Activate(isActive func()) error

    // Terminate the Carrier
    Close()

    // Mutate to a file-like object
    AsStream() io.ReadWriteCloser

    // Attempt to fork a new Carrier for communicating with the same Remote
    Fork() (Carrier, error)

    // Signal that an application is ready to accept forks via a given callback.
    // Forked carriers will be given to the callback until it returns false or
    // until the Carrier is closed.
    Accept(accept func(Carrier) bool) error
}

// Initiate a Carrier from a given Local to a given Remote. Returns a new
// Carrier, which may be bound to an existing or a new Association. The
// initiated Carrier is not yet active.
func Initiate(local Local, remote Remote) (Carrier, error)

type Listener interface {
    // Signal that an application is ready to accept forks via a given callback.
```

```
    // Accept will terminate when the callback returns false, or until the
    // Listener is closed.
    Accept(accept func(Carrier) bool) error

    // Terminate this Listener
    Close()
}

// Create a Listener on a given Local which will pass new Carriers to the
// given channel until that channel is closed.
func Listen(local Local) (Listener, error)

// A Source is a unidirectional, send-only Carrier.
type Source interface {
    // Send a byte array on this Source as a message with default metadata
    // and no notifications.
    Send(buf []byte) error

    // Send a message on this Source. The optional onSent function will be
    // called when the protocol stack instance has sent the message. The
    // optional onAcked function will be called when the receiver has
    // acknowledged the message. The optional onExpired function will be
    // called if the message's lifetime expired before the message coult be
    // sent. If the Source is not active, attempt to activate the Source
    // before sending.
    Sendmsg(msg *OutMessage, onSent func(), onAcked func(), onExpired func()) er
ror

    // Retrieve the Association over which this Source is running.
    Association() *Association

    // Determine whether the Source is currently active
    IsActive() bool

    // Ensure that the Source is active and ready to send messages.
    // Attempts to bring up at least one Transient.
    Activate() error

    // Terminate the Source
    Close()
}

// Initiate a Source from a given Local to a given Remote. Returns a new
// Source, which may be bound to an existing or a new Association. The
// initiated Source is not yet active.
func NewSource(local Local, remote Remote) (Source, error)

// A Sink is a unidirectional, receive-only Carrier, bound only to a local.
type Sink interface {
```

```
    // Signal that an application is ready to receive messages via a given callb
ack.
    // Messages will be given to the callback until it returns false, or until t
he
    // Sink is closed.
    Ready(receive func(InMessage) bool) error

    // Retrieve the Association over which this Sink is running.
    Association() *Association

    // Terminate the Sink
    Close()
}

// Initiate a Sink on a given Local. Returns a new
// Sink, which may be bound to an existing or a new Association.
func NewSink(local Local) (Sink, error)

// Initiate a Responder on a given Local. For each incoming Message, calls the
// respond function with the Message and a Sink to send replies to. Calls the
// Responder until it returns False, then terminates
func Respond(local Local, respond func(msg InMessage, reply Sink) bool) error

// A local identity
type Local struct {
    // A string identifying an interface or set of interfaces to accept messages
 and new carriers on.
    Interface string
    // A transport layer port
    Port int
    // A set of zero or more end entity certificates, together with private
    // keys, to identify this application with.
    Certificates []tls.Certificate
}

// Encapsulate a remote identity. Since the contents of a Remote are highly
// dependent on its level of resolution; some examples are below.
type Remote interface {
    // Resolve this Remote Identity to a
    Resolve() ([]RemoteIdentity, error)
    // Returns True if the Remote is completely resolved; i.e., cannot be resol
    Complete() bool
}

// Remote consisting of a URL
type URLRemote struct {
    URL string
}

// Remote encapsulating a name and port number
type NamedEndpointRemote struct {
```

```
    Hostname string
    Port     int
}

// Remote encapsulating an IP address and port number
type IPEndpointRemote struct {
    Address net.IP
    Port    int
}

// Remote encapsulating an IP address and port number, and a set of presented ce
rtificates
type IPEndpointCertRemote struct {
    Address      net.IP
    Port         int
    Certificates []tls.Certificate
}
```

Authors' Addresses

Brian Trammell
ETH Zurich
Gloriastrasse 35
8092 Zurich
Switzerland

Email: ietf@trammell.ch

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

Email: csp@csperkins.org

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

Email: tpauly@apple.com

Mirja Kuehlewind
ETH Zurich
Gloriastrasse 35
8092 Zurich
Switzerland

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

TAPS Working Group
Internet-Draft
Intended status: Informational
Expires: April 30, 2018

B. Trammell
ETH Zurich
C. Perkins
University of Glasgow
T. Pauly
Apple Inc.
M. Kuehlewind
ETH Zurich
C. Wood
Apple Inc.

October 27, 2017

Post Sockets, An Abstract Programming Interface for the Transport Layer
draft-trammell-taps-post-sockets-03

Abstract

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

Status of This Memo

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

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

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

This Internet-Draft will expire on April 30, 2018.

Copyright Notice

Copyright (c) 2017 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (https://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction 3
2. Abstractions and Terminology 5
2.1. Message Carrier 6
2.2. Message 8
2.3. Association 11
2.4. Remote 11
2.5. Local 12
2.6. Configuration 12
2.7. Transient 13
2.8. Path 14
3. Abstract Programming Interface 15
3.1. Example Connection Patterns 16
3.1.1. Client-Server 16
3.1.2. Client-Server with Happy Eyeballs and 0-RTT establishment 18
3.1.3. Peer to Peer with Network Address Translation 18
3.1.4. Multicast Receiver 18
3.1.5. Association Bootstrapping 19
3.2. API Dynamics 20
4. Implementation Considerations 22
4.1. Protocol Stack Instance (PSI) 23
4.2. Message Framing, Parsing, and Serialization 24
4.3. Message Size Limitations 25
4.4. Back-pressure 25
4.5. Associations, Transients, Racing, and Rendezvous 26
5. Acknowledgments 28
6. References 28
6.1. Normative References 28
6.2. Informative References 28
Appendix A. Open Issues 30
Authors' Addresses 30

1. Introduction

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

In the meantime, the nature of Internet access, and the variety of Internet transport protocols, is evolving. The challenges that new protocols and access paradigms present to the sockets API and to programming models based on them inspire the design elements of a new approach.

Many end-user devices are connected to the Internet via multiple interfaces, which suggests it is time to promote the paths by which two endpoints are connected to each other to a first-order object. While implicit multipath communication is available for these multihomed nodes in the present Internet architecture with the Multipath TCP extension (MPTCP) [RFC6824], MPTCP was specifically designed to hide multipath communication from the application for purposes of compatibility. Since many multihomed nodes are connected to the Internet through access paths with widely different properties with respect to bandwidth, latency and cost, adding explicit path control to MPTCP's API would be useful in many situations.

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

To meet these challenges, we present the Post-Sockets Application Programming Interface (API), described in detail in this work. Post is designed to be language, transport protocol, and architecture independent, allowing applications to be written to a common abstract interface, easily ported among different platforms, and used even in environments where transport protocol selection may be done dynamically, as proposed in the IETF's Transport Services working group.

Post replaces the traditional SOCK_STREAM abstraction with a Message abstraction, which can be seen as a generalization of the Stream Control Transmission Protocol's [RFC4960] SOCK_SEQPACKET service. Messages are sent and received on Carriers, which logically group Messages for transmission and reception. For backward compatibility, bidirectional byte stream protocols are represented as a pair of Messages, one in each direction, that can only be marked complete when the sending peer has finished transmitting data.

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

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

- Explicit Message orientation, with framing and atomicity guarantees for Message transmission.
- Asynchronous reception, allowing all receiver-side interactions to be event-driven.
- Explicit support for multistreaming and multipath transport protocols and network architectures.
- Long-lived Associations, whose lifetimes may not be bound to underlying transport connections. This allows associations to cache state and cryptographic key material to enable fast resumption of communication, and for the implementation of the API to explicitly take care of connection establishment mechanics such as connection racing [RFC6555] and peer-to-peer rendezvous [RFC5245].

- Transport protocol stack independence, allowing applications to be written in terms of the semantics best for the application's own design, separate from the protocol(s) used on the wire to achieve them. This enables applications written to a single API to make use of transport protocols in terms of the features they provide, as in [I-D.ietf-taps-transports].

This work is the synthesis of many years of Internet transport protocol research and development. It is inspired by concepts from the Stream Control Transmission Protocol (SCTP) [RFC4960], TCP Minion [I-D.iyengar-minion-protocol], and MinimaLT [MinimaLT], among other transport protocol modernization efforts. We present Post as an illustration of what is possible with present developments in transport protocols when freed from the strictures of the current sockets API. While much of the work for building parts of the protocols needed to implement Post are already ongoing in other IETF working groups (e.g. MPTCP, QUIC, TLS), we argue that an abstract programming interface unifying access all these efforts is necessary to fully exploit their potential.

2. Abstractions and Terminology

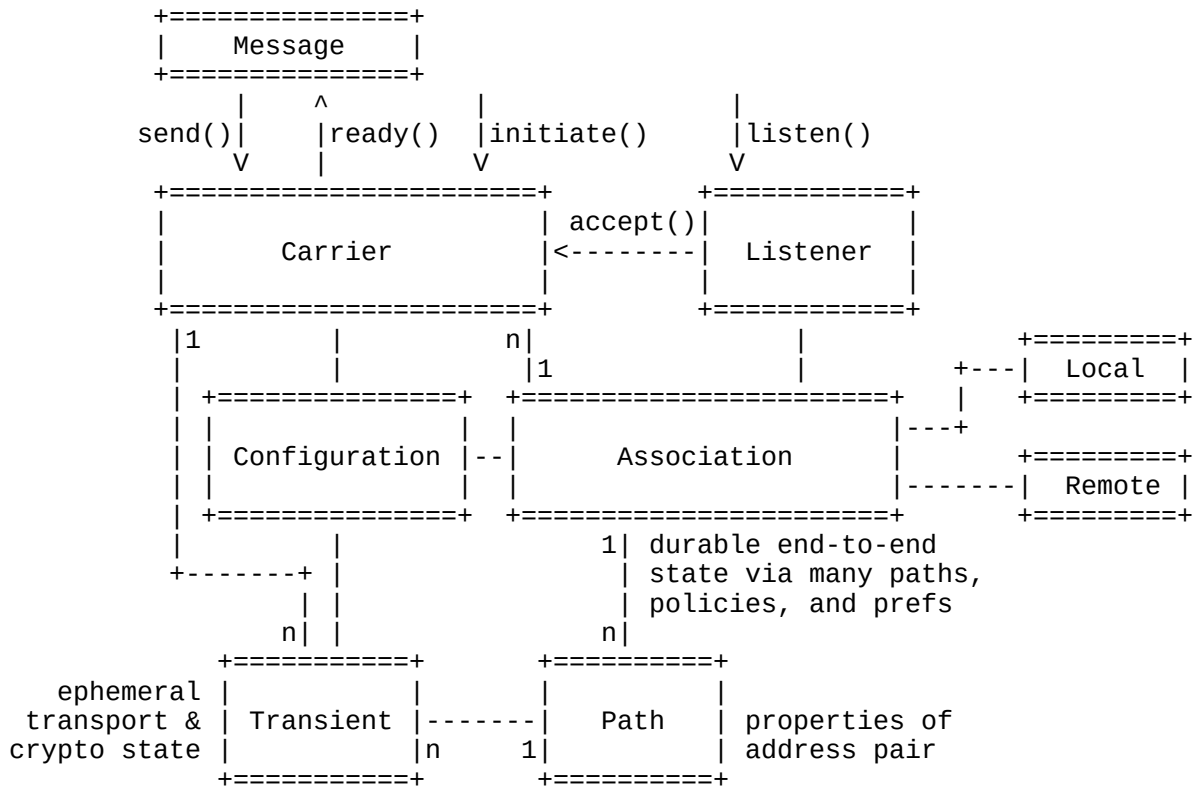


Figure 1: Abstractions and relationships in Post Sockets

Post is based on a small set of abstractions, centered around a Message Carrier as the entry point for an application to the networking API. The relationships among them are shown in Figure Figure 1 and detailed in this section.

2.1. Message Carrier

A Message Carrier (or simply Carrier) is a transport protocol stack-independent interface for sending and receiving messages between an application and a remote endpoint; it is roughly analogous to a socket in the present sockets API.

Sending a Message over a Carrier is driven by the application, while receipt is driven by the arrival of the last packet that allows the Message to be assembled, decrypted, and passed to the application. Receipt is therefore asynchronous; given the different models for asynchronous I/O and concurrency supported by different platforms, it

may be implemented in any number of ways. The abstract API provides only for a way for the application to register how it wants to handle incoming messages.

All the Messages sent to a Carrier will be received on the corresponding Carrier at the remote endpoint, though not necessarily reliably or in order, depending on Message properties and the underlying transport protocol stack.

A Carrier that is backed by current transport protocol stack state (such as a TCP connection; see Section 2.7) is said to be "active": messages can be sent and received over it. A Carrier can also be "dormant": there is long-term state associated with it (via the underlying Association; see Section 2.3), and it may be able to reactivated, but messages cannot be sent and received immediately. Carriers become dormant when the underlying transport protocol stack determines that an underlying connection has been lost and there is insufficient state in the Association to re-establish it (e.g., in the case of a server-side Carrier where the client’s address has changed unexpectedly). Passive close can be handled by the application via an event on the carrier. Attempting to use a carrier after passive close results in an error.

If supported by the underlying transport protocol stack, a Carrier may be forked: creating a new Carrier associated with a new Carrier at the same remote endpoint. The semantics of the usage of multiple Carriers based on the same Association are application-specific. When a Carrier is forked, its corresponding Carrier at the remote endpoint receives a fork request, which it must accept in order to fully establish the new carrier. Multiple Carriers between endpoints are implemented differently by different transport protocol stacks, either using multiple separate transport-layer connections, or using multiple streams of multistreaming transport protocols.

To exchange messages with a given remote endpoint, an application may initiate a Carrier given its remote (see Section 2.4 and local (see Section 2.5) identities; this is an equivalent to an active open. There are four special cases of Carriers, as well, supporting different initiation and interaction patterns, defined in the subsections below.

- Listener: A Listener is a special case of Message Carrier which only responds to requests to create a new Carrier from a remote endpoint, analogous to a server or listening socket in the present sockets API. Instead of being bound to a specific remote endpoint, it is bound only to a local identity; however, its interface for accepting fork requests is identical to that for fully fledged Carriers.

- Source: A Source is a special case of Message Carrier over which messages can only be sent, intended for unidirectional applications such as multicast transmitters. Sources cannot be forked, and need not accept forks.

- Sink: A Sink is a special case of Message Carrier over which messages can only be received, intended for unidirectional applications such as multicast receivers. Sinks cannot be forked, and need not accept forks.

- Responder: A Responder is a special case of Message Carrier which may receive messages from many remote sources, for cases in which an application will only ever send Messages in reply back to the source from which a Message was received. This is a common implementation pattern for servers in client-server applications. A Responder's receiver gets a Message, as well as a Source to send replies to. Responders cannot be forked, and need not accept forks.

2.2. Message

A Message is the unit of communication between applications. Messages can represent relatively small structures, such as requests in a request/response protocol such as HTTP; relatively large structures, such as files of arbitrary size in a filesystem; and structures of indeterminate length, such as a stream of bytes in a protocol like TCP.

In the general case, there is no mapping between a Message and packets sent by the underlying protocol stack on the wire: the transport protocol may freely segment messages and/or combine messages into packets. However, a message may be marked as immediate, which will cause it to be sent in a single packet when possible.

Content may be sent and received either as Complete or Partial Messages. Dealing with Complete Messages should be preferred for simplicity whenever possible based on the underlying protocol. It is always possible to send Complete Messages, but only protocols that have a fixed maximum message length may allow clients to receive Messages using an API that guarantees Complete Messages. Sending and receiving Partial Messages (that is, a Message whose content spans multiple calls or callbacks) is always possible.

To send a Message, either Complete or Partial, the Message content is passed into the Carrier, and client provides a set of callbacks to know when the Message was delivered or acknowledged. The client of the API may use the callbacks to pace the sending of Messages.

To receive a Message, the client of the API schedules a completion to be called when a Complete or Partial Message is available. If the client is willing to accept Partial Messages, it can specify the minimum incomplete Message length it is willing to receive at once, and the maximum number of bytes it is willing to receive at once. If the client wants Complete Messages, there are no values to tune. The scheduling of the receive completion indicates to the Carrier that there is a desire to receive bytes, effectively creating a "pull model" in which backpressure may be applied if the client is not receiving Messages or Partial Messages quickly enough to match the peer's sending rate. The Carrier may have some minimal buffer of incoming Messages ready for the client to read to reduce latency.

When receiving a Complete Message, the entire content of the Message must be delivered at once, and the Message is not delivered at all if the full Message is not received. This implies that both the sending and receiving endpoint, whether in the application or the carrier, must guarantee storage for the full size of a Message.

Partial Messages may be sent or received in several stages, with a handle representing the total Message being associated with each portion of the content. Each call to send or receive also indicates whether or not the Message is now complete. This approach is necessary whenever the size of the Message does not have a known bound, or the size is too large to process and hold in memory. Protocols that only present a concept of byte streams represent their data as single Messages with unknown bounds. In the case of TCP, the client will receive a single Message in pieces using the Partial Message API, and that Message will only be marked as complete when the peer has sent a FIN.

Messages are sent over and received from Message Carriers (see Section 2.1).

On sending, Messages have properties that allow the application to specify its requirements with respect to reliability, ordering, priority, idempotence, and immediacy; these are described in detail below. Messages may also have arbitrary properties which provide additional information to the underlying transport protocol stack on how they should be handled, in a protocol-specific way. These stacks may also deliver or set properties on received messages, but in the general case a received messages contains only a sequence of ordered bytes. Message properties include:

- Lifetime and Partial Reliability: A Message may have a "lifetime" - a wall clock duration before which the Message must be available to the application layer at the remote end. If a lifetime cannot be met, the Message is discarded as soon as possible. Messages

without lifetimes are sent reliably if supported by the transport protocol stack. Lifetimes are also used to prioritize Message delivery.

There is no guarantee that a Message will not be delivered after the end of its lifetime; for example, a Message delivered over a strictly reliable transport will be delivered regardless of its lifetime. Depending on the transport protocol stack used to transmit the message, these lifetimes may also be signalled to path elements by the underlying transport, so that path elements that realize a lifetime cannot be met can discard frames containing the Messages instead of forwarding them.

- Priority: Messages have a "niceness" - a priority among other messages sent over the same Carrier in an unbounded hierarchy most naturally represented as a non-negative integer. By default, Messages are in niceness class 0, or highest priority. Niceness class 1 Messages will yield to niceness class 0 Messages sent over the same Carrier, class 2 to class 1, and so on. Niceness may be translated to a priority signal for exposure to path elements (e.g. DSCP code point) to allow prioritization along the path as well as at the sender and receiver. This inversion of normal schemes for expressing priority has a convenient property: priority increases as both niceness and lifetime decrease. A Message may have both a niceness and a lifetime - Messages with higher niceness classes will yield to lower classes if resource constraints mean only one can meet the lifetime.

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

- Idempotence: A sending application may mark a Message as "idempotent" to signal to the underlying transport protocol stack that its application semantics make it safe to send in situations that may cause it to be received more than once (i.e., for 0-RTT session resumption as in TCP Fast Open, TLS 1.3, and QUIC).

- Immediacy: A sending application may mark a Message as "immediate" to signal to the underlying transport protocol stack that its application semantics require it to be placed in a single packet, on its own, instead of waiting to be combined with other messages or parts thereof (i.e., for media transports and interactive sessions with small messages).

Senders may also be asynchronously notified of three events on Messages they have sent: that the Message has been transmitted, that the Message has been acknowledged by the receiver, or that the Message has expired before transmission/acknowledgement. Not all transport protocol stacks will support all of these events.

2.3. Association

An Association contains the long-term state necessary to support communications between a Local (see Section 2.5) and a Remote (see Section 2.4) endpoint, such as trust model information, including pinned public keys or anchor certificates, cryptographic session resumption parameters, or rendezvous information. It uses information from the Configuration (see Section 2.6) to constrain the selection of transport protocols and local interfaces to create Transients (see Section 2.7) to carry Messages; and information about the paths through the network available available between them (see Section 2.8).

All Carriers are bound to an Association. New Carriers will reuse an Association if they can be carried from the same Local to the same Remote over the same Paths; this re-use of an Association may implies the creation of a new Transient.

Associations may exist and be created without a Carrier. This may be done if peer cryptographic state such as a pre-shared key is established out-of-band. Thus, Associations may be created without the need to send application data to a peer, that is, without a Carrier. Associations are mutable. Association state may expire over time, after which it is removed from the Association, and Transients may export cryptographic state to store in an Association as needed. Moreover, this state may be exported directly into the Association or modified before insertion. This may be needed to diversify ephemeral Transient keying material from the longer-term Association keying material.

A primary use of Association state is to allow new Associations and their derived Carriers to be quickly created without performing in-band cryptographic handshakes. See [I-D.kuehlewind-taps-crypto-sep] for more details about this separation.

2.4. Remote

A Remote represents information required to establish and maintain a connection with the far end of an Association: name(s), address(es), and transport protocol parameters that can be used to establish a Transient; transport protocols to use; trust model information, inherited from the relevant Association, used to identify the remote

on connection establishment; and so on. Each Association is associated with a single Remote, either explicitly by the application (when created by the initiation of a Carrier) or a Listener (when created by forking a Carrier on passive open).

A Remote may be resolved, which results in zero or more Remotes with more specific information. For example, an application may want to establish a connection to a website identified by a URL https://www.example.com. This URL would be wrapped in a Remote and passed to a call to initiate a Carrier. The first pass resolution might parse the URL, decomposing it into a name, a transport port, and a transport protocol to try connecting with. A second pass resolution would then look up network-layer addresses associated with that name through DNS, and store any certificates available from DANE. Once a Remote has been resolved to the point that a transport protocol stack can use it to create a Transient, it is considered fully resolved.

2.5. Local

A Local represents all the information about the local endpoint necessary to establish an Association or a Listener. It encapsulates the Provisioning Domain (PvD) of a single interface in the multiple provisioning domain architecture [RFC7556], and adds information about the service endpoint (transport protocol port), and, per [I-D.pauly-taps-transport-security], cryptographic identities (certificates and associated private keys) bound to this endpoint.

2.6. Configuration

A Configuration encapsulates an application’s preferences around Path selection and protocol options.

Each Association has exactly one Configuration, and all Carriers belonging to that Association share the same Configuration.

The application cannot modify the Configuration for a Carrier or Association once it is set. If a new set of options needs to be used, then the application needs a new Carrier or Association instance. This is necessary to ensure that a single Carrier can consistently track the Paths and protocol options it uses, since it is usually not possible to modify these properties without breaking connectivity.

To influence Path selection, the application can configure a set of requirements, preferences, and restrictions concerning which Paths may be selected by the Association to use for creating Transients between a Local and a Remote. For example, a Configuration can

specify that the application prefers Wi-Fi access over LTE when roaming on a foreign LTE network, due to monetary cost to the user.

The Association uses the Configuration's Path preferences as a key part of determining the Paths to use for its Transients. The Configuration is provided as input when examining the complete list of available Paths on the system (to limit the list, or order the Paths by preference). The system's policy will further restrict and modify the Path that is ultimately selected, using other aspects of the Configuration (protocol options and originating application) to select the most appropriate Path.

To influence protocol selection and options, the Configuration contains one or more allowed Protocol Stack Configurations. Each of these is comprised of application- and transport-layer protocols that may be used together to communicate to the Remote, along with any protocol-specific options. For example, a Configuration could specify two alternate, but equivalent, protocol stacks: one using HTTP/2 over TLS over TCP, and the other using QUIC over UDP. Alternatively, the Configuration could specify two protocol stacks with the same protocols, but different protocol options: one using TLS with TLS 1.3 0-RTT enabled and TCP with TCP Fast-Open enabled, and one using TLS with out 0-RTT and TCP without TCP Fast-Open.

Protocol-specific options within the Configuration include trust settings and acceptable cryptographic algorithms to be used by security protocols. These may be configured for specific protocols to allow different settings for each (such as between TLS over TCP and TLS for use with QUIC), or set as default security settings on the Configuration to be used by any protocol that needs to evaluate trust. Trust settings may include certificate anchors and certificate pinning options.

2.7. Transient

A Transient represents a binding between a Carrier and the instance of the transport protocol stack that implements it. As an Association contains long-term state for communications between two endpoints, a Transient contains ephemeral state for a single transport protocol over a one or more Paths at a given point in time.

A Carrier may be served by multiple Transients at once, e.g. when implementing multipath communication such that the separate paths are exposed to the API by the underlying transport protocol stack. Each Transient serves only one Carrier, although multiple Transients may share the same underlying protocol stack; e.g. when multiplexing Carriers over streams in a multistreaming protocol.

Transients are generally not exposed by the API to the application, though they may be accessible for debugging and logging purposes.

2.8. Path

A Path represents information about a single path through the network used by an Association, in terms of source and destination network and transport layer addresses within an addressing context, and the provisioning domain [RFC7556] of the local interface. This information may be learned through a resolution, discovery, or rendezvous process (e.g. DNS, ICE), by measurements taken by the transport protocol stack, or by some other path information discovery mechanism. It is used by the transport protocol stack to maintain and/or (re-)establish communications for the Association.

The set of available properties is a function of the transport protocol stacks in use by an association. However, the following core properties are generally useful for applications and transport layer protocols to choose among paths for specific Messages:

- Maximum Transmission Unit (MTU): the maximum size of an Message's payload (subtracting transport, network, and link layer overhead) which will likely fit into a single frame. Derived from signals sent by path elements, where available, and/or path MTU discovery processes run by the transport layer.

- Latency Expectation: expected one-way delay along the Path. Generally provided by inline measurements performed by the transport layer, as opposed to signaled by path elements.

- Loss Probability Expectation: expected probability of a loss of any given single frame along the Path. Generally provided by inline measurements performed by the transport layer, as opposed to signaled by path elements.

- Available Data Rate Expectation: expected maximum data rate along the Path. May be derived from passive measurements by the transport layer, or from signals from path elements.

- Reserved Data Rate: Committed, reserved data rate for the given Association along the Path. Requires a bandwidth reservation service in the underlying transport protocol stack.

- Path Element Membership: Identifiers for some or all nodes along the path, depending on the capabilities of the underlying network layer protocol to provide this.

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

3. Abstract Programming Interface

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

- Flexibility is paramount. So is simplicity. Applications must be given as many controls and as much information as they may need, but they must be able to ignore controls and information irrelevant to their operation. This implies that the "default" interface must be no more complicated than BSD sockets, and must do something reasonable.

- Reception is an inherently asynchronous activity. While the API is designed to be as platform-independent as possible, one key insight it is based on is that an Message receiver's behavior in a packet-switched network is inherently asynchronous, driven by the receipt of packets, and that this asynchronicity must be reflected in the API. The actual implementation of receive and event handling will need to be aligned to the method a given platform provides for asynchronous I/O.

- A new API cannot be bound to a single transport protocol and expect wide deployment. As the API is transport-independent and may support runtime transport selection, it must impose the minimum possible set of constraints on its underlying transports, though some API features may require underlying transport features to work optimally. It must be possible to implement Post over vanilla TCP in the present Internet architecture.

The API we design from these principles is centered around a Carrier, which can be created actively via initiate() or passively via a listen(); the latter creates a Listener from which new Carriers can be accept()ed. Messages may be created explicitly and passed to this Carrier, or implicitly through a simplified interface which uses default message properties (reliable transport without priority or

deadline, which guarantees ordered delivery over a single Carrier when the underlying transport protocol stack supports it).

For each connection between a Local and a Remote a new Carrier is created and destroyed when the connection is closed. However, a new Carrier may use an existing Association if present for the requested Local-Remote pair and permitted by the PolicyContext that can be provided at Carrier initiation. Further the system-wide PolicyContext can contain more information that determine when to create or destroy Associations other than at Carrier initiation. E.g. an Association can be created at system start, based on the configured PolicyContext or also by a manual action of an single application, for Local-Remote pairs that are known to be likely used soon, and to pre-establish, e.g., cryptographic context as well as potentially collect current information about path capabilities. Every time an actual connection with a specific PSI is established between the Local and Remote, the Association learns new Path information and stores them. This information can be used when a new transient is created, e.g. to decide which PSI to use (to provide the highest probably for a successful connection attempt) or which PSIs to probe for (first). A Transient is created when an application actually sends a Message over a Carrier. As further explained below this step can actually create multiple transients for probing or assign a new transient to an already active PSI, e.g. if multi-streaming is provided and supported for these kind of use on both sides.

3.1. Example Connection Patterns

Here, we illustrate the usage of the API for common connection patterns. Note that error handling is ignored in these illustrations for ease of reading.

3.1.1. Client-Server

Here’s an example client-server application. The server echoes messages. The client sends a message and prints what it receives.

The client in Figure 2 connects, sends a message, and sets up a receiver to print messages received in response. The carrier is inactive after the Initiate() call; the Send() call blocks until the carrier can be activated.

```
// connect to a server given a remote
func sayHello() {

    carrier := Initiate(local, remote)

    carrier.Send([]byte("Hello!"))
    carrier.Ready(func (msg InMessage) {
        fmt.Println(string([]byte(msg))
        return false
    })
    carrier.Close()
}
```

Figure 2: Example client

The server in Figure 3 creates a Listener, which accepts Carriers and passes them to a server. The server echos the content of each message it receives.

```
// run a server for a specific carrier, echo all its messages
func runMyServerOn(carrier Carrier) {
    carrier.Ready(func (msg InMessage) {
        carrier.Send(msg)
    })
}

// accept connections forever, spawn servers for them
func acceptConnections() {
    listener := Listen(local)
    listener.Accept(func(carrier Carrier) bool {
        go runMyServerOn(carrier)
        return true
    })
}
```

Figure 3: Example server

The Responder allows the server to be significantly simplified, as shown in Figure 4.

```
func echo(msg InMessage, reply Sink) {
    reply.Send(msg)
}

Respond(local, echo)
```

Figure 4: Example responder

3.1.2. Client-Server with Happy Eyeballs and 0-RTT establishment

The fundamental design of a client need not change at all for happy eyeballs [RFC6555] (selection of multiple potential protocol stacks through connection racing); this is handled by the Post Sockets implementation automatically. If this connection racing is to use 0-RTT data (i.e., as provided by TCP Fast Open [RFC7413], the client must mark the outgoing message as idempotent.

```
// connect to a server given a remote and send some 0-RTT data
func sayHelloQuickly() {

    carrier := Initiate(local, remote)

    carrier.SendMsg(OutMessage{Content: []byte("Hello!"), Idempotent: true}, nil
, nil, nil)
    carrier.Ready(func (msg InMessage) {
        fmt.Println(string([]byte(msg)))
        return false
    })
    carrier.Close()
}
```

3.1.3. Peer to Peer with Network Address Translation

In the client-server examples shown above, the Remote given to the Initiate call refers to the name and port of the server to connect to. This need not be the case, however; a Remote may also refer to an identity and a rendezvous point for rendezvous as in ICE [RFC5245]. Here, each peer does its own Initiate call simultaneously, and the result on each side is a Carrier attached to an appropriate Association.

3.1.4. Multicast Receiver

A multicast receiver is implemented using a Sink attached to a Local encapsulating a multicast address on which to receive multicast datagrams. The following example prints messages received on the multicast address forever.

```
func receiveMulticast() {
    sink = NewSink(local)
    sink.Ready(func (msg InMessage) {
        fmt.Println(string([]byte(msg)))
        return true
    })
}
```

3.1.5. Association Bootstrapping

Here, we show how Association state may be initialized without a carrier. The goal is to create a long-term Association from which Carriers may be derived and, if possible, used immediately. Per [I-D.pauly-taps-transport-security], a first step is to specify trust model constraints, such as pinned public keys and anchor certificates, which are needed to create Remote connections.

We begin by creating shared security parameters that will be used later for creating a remote connection.

```
// create security parameters with a set of trusted certificates
func createParameters(trustedCerts []Certificate) Parameters {
    parameters := Parameters()
    parameters = parameters.SetTrustedCerts(trustedCerts)
    return parameters
}
```

Using these statically configured parameters, we now show how to create an Association between a Local and Remote using these parameters.

```
// create an Association using shared parameters
func createAssociation(local Local, remote Remote, parameters Parameters) Associ
ation {
    association := NewAssociation(local, remote, parameters)
    return association
}
```

We may also create an Association with a pre-shared key configured out-of-band.

```
// create an Association using a pre-shared key
func createAssociationWithPSK(local Local, remote Remote, parameters Parameters,
 preSharedKey []byte) Association {
    association := NewAssociation(local, remote, parameters)
    association = association.SetPreSharedKey(preSharedKey)
    return association
}
```

We now show how to create a Carrier from an existing, pre-configured Association. This Association may or may not contain shared cryptographic static between the Local and Remote, depending on how it was configured.

```
// open a connection to a server using an existing Association and send some dat
a,
// which will be sent early if possible.
func sayHelloWithAssociation(association Association) {
    carrier := association.Initiate()

    carrier.SendMsg(OutMessage{Content: []byte("Hello!"), Idempotent: true}, nil
, nil, nil)
    carrier.Ready(func (msg InMessage) {
        fmt.Println(string([]byte(msg)))
        return false
    })
    carrier.Close()
}
```

3.2. API Dynamics

As Carriers provide the central entry point to Post, they are key to API dynamics. The lifecycle of a carrier is shown in Figure 5. Carriers are created by active openers by calling Initiate() given a Local and a Remote, and by passive openers by calling Listen() given a Local; the .Accept() method on the listener Carrier can then be used to create active carriers. By default, the underlying Association is automatically created and managed by the underlying API. This underlying Association can be accessed by the Carrier's .Association() method. Alternately, an association can be explicitly created using NewAssociation(), and a Carrier on the association may be accessed or initiated by calling the association's .Initiate() method.

Once a Carrier has been created (via Initiate(), Association.Initiate(), NewSource(), NewSink(), or Listen()/Accept()), it may be used to send and receive Messages. The existence of a Carrier does not imply the existence of an active Transient or associated transport-layer connection; these may be created when the carrier is, or may be deferred, depending on the network environment, configuration, and protocol stacks available.

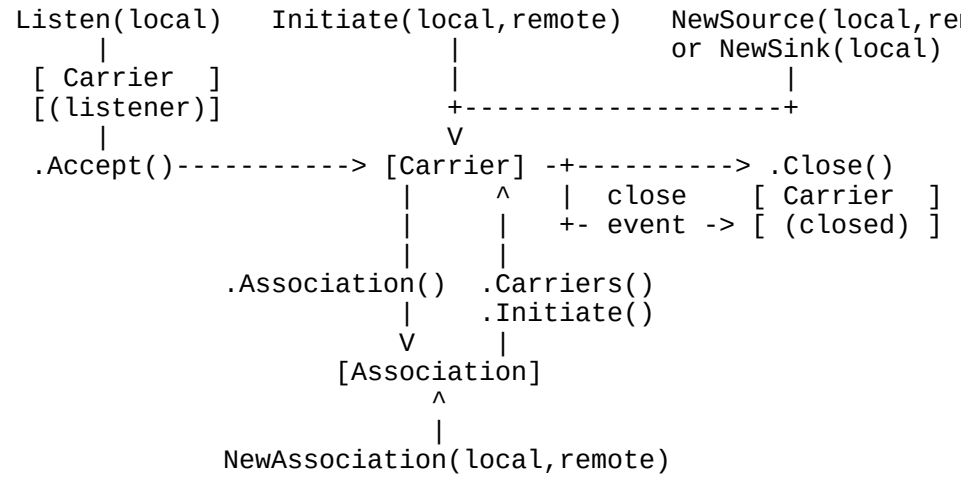


Figure 5: Carrier and Association Life Cycle

Access to more detailed information is possible through accessors on Carriers and Associations, as shown in Figure 6. The set of currently active Transients can be accessed through the Carrier's .Transients() methods. The active path(s) used by a Transient can be accessed through the Transient's .Paths() method, and the set of all paths for which properties are cached by an Association can be accessed through the Association's .Paths() method. The set of active carriers on an association can be accessed through the Association's .Carriers() method. Access to transients and paths is not necessary in normal operation; these accessors are provided primarily for logging and debugging purposes.

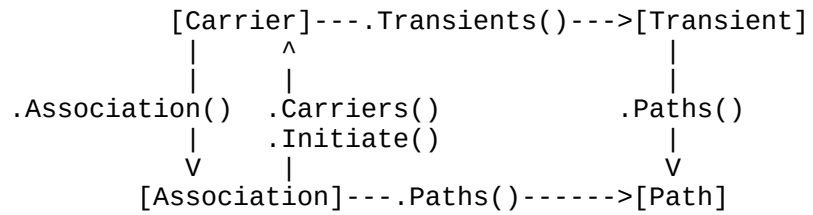


Figure 6: Accessors on Carriers and Associations

Each Carrier has a .Send() method, by which Messages can be sent with given properties, and a .Ready() method, which supplies a callback for reading Messages from the remote side. .Send() is not available on Sinks, and .Ready() is not available on Sources. Carriers also provide .OnSent(), .OnAcked(), and .OnExpired() calls for binding default send event handlers to the Carrier, and .OnClosed() for handling passive close notifications.

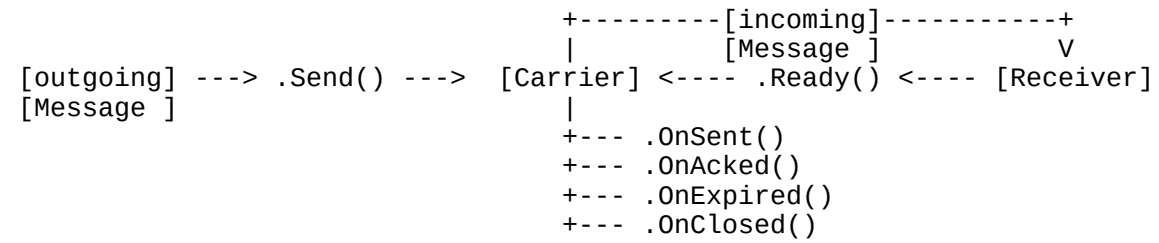


Figure 7: Sending and Receiving Messages and Events

An application may have a global Configuation, as well as more specific Configurations to apply to the establishment of a given Association or Carrier. These Configurations are optional arguments to the Association and Carrier creation calls.

In order to initiate a connection with a remote endpoint, a user of Post Sockets must start from a Remote (see Section 2.4). A Remote encapsulates identifying information about a remote endpoint at a specific level of resolution. A new Remote can be wrapped around some identifying information by via the NewRemote() call. A Remote has a .Resolve() method, which can be iteratively revoked to increase the level of resolution; a call to Resolve on a given Remote may result in one to many Remotes, as shown in Figure 8. Remotes at any level of resolution may be passed to Post Sockets calls; each call will continue resolution to the point necessary to establish or resume a Carrier.

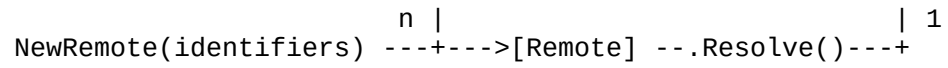


Figure 8: Recursive resolution of Remotes

Information about the local endpoint is also necessary to establish an Association, whether explicitly or implicitly through the creation of a Carrier or Listener. This is passed in the form of a Local (see Section 2.5). A Local is created with a NewLocal() call, which takes a Configuration (including certificates to present and secret keys associated with them) and identifying information (interface(s) and port(s) to use).

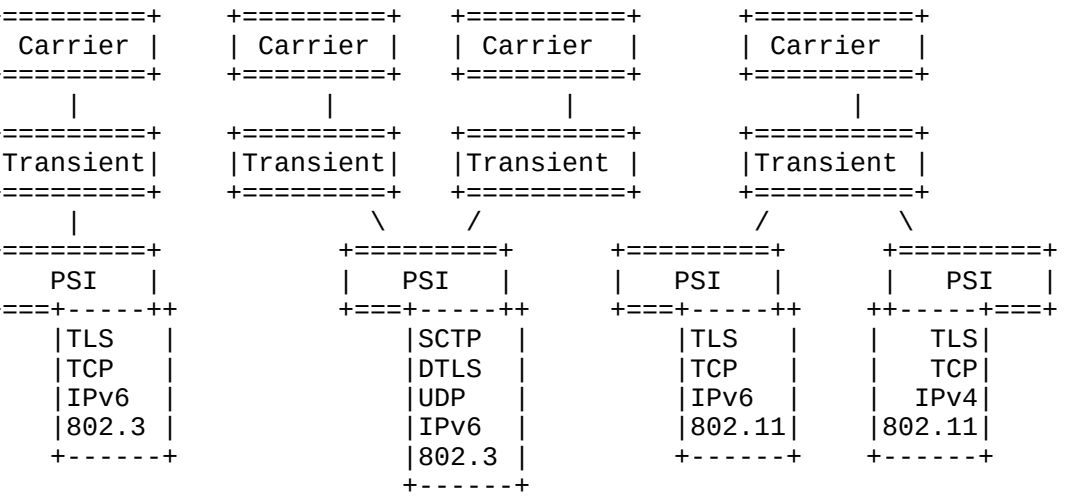
4. Implementation Considerations

Here we discuss an incomplete list of API implementation considerations that have arisen with experimentation with prototype implementations of Post.

4.1. Protocol Stack Instance (PSI)

A PSI encapsulates an arbitrary stack of protocols (e.g., TCP over IPv6, SCTP over DTLS over UDP over IPv4). PSIs provide the bridge between the interface (Carrier) plus the current state (Transients) and the implementation of a given set of transport services [I-D.ietf-taps-transports].

A given implementation makes one or more possible protocol stacks available to its applications. Selection and configuration among multiple PSIs is based on system-level or application policies, as well as on network conditions in the provisioning domain in which a connection is made.



(a) Transient bound to PSI (b) Carrier multiplexing over a multi-streaming transport protocol (c) Multiple candidates racing during session establishment

Figure 9: Example Protocol Stack Instances

For example, Figure 9(a) shows a TLS over TCP stack, usable on most network connections. Protocols are layered to ensure that the PSI provides all the transport services required by the application. A single PSI may be bound to multiple Carriers, as shown in Figure 9(b): a multi-streaming transport protocol like QUIC or SCTP can support one carrier per stream. Where multi-streaming transport is not available, these carriers could be serviced by different PSIs on different flows. On the other hand, multiple PSIs are bound to a single transient during establishment, as shown in Figure 9(c). Here, the losing PSI in a happy-eyeballs race will be terminated, and the carrier will continue using the winning PSI.

4.2. Message Framing, Parsing, and Serialization

While some transports expose a byte stream abstraction, most higher level protocols impose some structure onto that byte stream. That is, the higher level protocol operates in terms of messages, protocol data units (PDUs), rather than using unstructured sequences of bytes, with each message being processed in turn. Protocols are specified in terms of state machines acting on semantic messages, with parsing the byte stream into messages being a necessary annoyance, rather than a semantic concern. Accordingly, Post Sockets exposes a message-based API to applications as the primary abstraction. Protocols that deal only in byte streams, such as TCP, represent their data in each direction as a single, long message. When framing protocols are placed on top of byte streams, the messages used in the API represent the framed messages within the stream.

There are other benefits of providing a message-oriented API beyond framing PDUs that Post Sockets should provide when supported by the underlying transport. These include:

- the ability to associate deadlines with messages, for transports that care about timing;
- the ability to provide control of reliability, choosing what messages to retransmit in the event of packet loss, and how best to make use of the data that arrived;
- the ability to manage dependencies between messages, when some messages may not be delivered due to either packet loss or missing a deadline, in particular the ability to avoid (re-)sending data that relies on a previous transmission that was never received.

All require explicit message boundaries, and application-level framing of messages, to be effective. Once a message is passed to Post Sockets, it can not be cancelled or paused, but prioritization as well as lifetime and retransmission management will provide the protocol stack with all needed information to send the messages as quickly as possible without blocking transmission unnecessarily. Post Sockets provides this by handling message, with known identity (sequence numbers, in the simple case), lifetimes, niceness, and antecedents.

Transport protocols such as SCTP provide a message-oriented API that has similar features to those we describe. Other transports, such as TCP, do not. To support a message oriented API, while still being compatible with stream-based transport protocols, Post Sockets must provide APIs for parsing and serialising messages that understand the protocol data. That is, we push message parsing and serialisation

down into the Post Sockets stack, allowing applications to send and receive strongly typed data objects (e.g., a receive call on an HTTP Message Carrier should return an object representing the HTTP response, with pre-parsed status code, headers, and any message body, rather than returning a byte array that the application has to parse itself). This is backwards compatible with existing protocols and APIs, since the wire format of messages does not change, but gives a Post Sockets stack additional information to allow it to make better use of modern transport services.

The Post Sockets approach is therefore to raise the semantic level of the transport API: applications should send and receive messages in the form of meaningful, strongly typed, protocol data. Parsing and serialising such messages should be a re-usable function of the protocol stack instance not the application. This is well-suited to implementation in modern systems languages, such as Swift, Go, Rust, or C++, but can also be implemented with some loss of type safety in C.

4.3. Message Size Limitations

Ideally, Messages can be of infinite size. However, protocol stacks and protocol stack implementations may impose their own limits on message sizing; For example, SCTP [RFC4960] and TLS [I-D.ietf-tls-tls13] impose record size limitations of 64kB and 16kB, respectively. Message sizes may also be limited by the available buffer at the receiver, since a Message must be fully assembled by the transport layer before it can be passed on to the application layer. Since not every transport protocol stack implements the signaling necessary to negotiate or expose message size limitations, these may need to be defined out of band, and are probably best exposed through the Configuration.

A truly infinite message service - e.g. large file transfer where both endpoints have committed persistent storage to the message - is probably best realized as a layer above Post Sockets, and may be added as a new type of Message Carrier to a future revision of this document.

4.4. Back-pressure

Regardless of how asynchronous reception is implemented, it is important for an application to be able to apply receiver back-pressure, to allow the protocol stack to perform receiver flow control. Depending on how asynchronous I/O works in the platform, this could be implemented by having a maximum number of concurrent receive callbacks, or by bounding the maximum number of outstanding, unread bytes at any given time, for example.

4.5. Associations, Transients, Racing, and Rendezvous

As the network has evolved, even the simple act of establishing a connection has become increasingly complex. Clients now regularly race multiple connections, for example over IPv4 and IPv6, to determine which protocol to use. The choice of outgoing interface has also become more important, with differential reachability and performance from multiple interfaces. Name resolution can also give different outcomes depending on the interface the query was issued from. Finally, but often most significantly, NAT traversal, relay discovery, and path state maintenance messages are an essential part of connection establishment, especially for peer-to-peer applications.

Post Sockets accordingly breaks communication establishment down into multiple phases:

- Gathering Locals

  The set of possible Locals is gathered. In the simple case, this merely enumerates the local interfaces and protocols, and allocates ephemeral source ports for transients. For example, a system that has WiFi and Ethernet and supports IPv4 and IPv6 might gather four candidate locals (IPv4 on Ethernet, IPv6 on Ethernet, IPv4 on WiFi, and IPv6 on WiFi) that can form the source for a transient.

  If NAT traversal is required, the process of gathering locals becomes broadly equivalent to the ICE candidate gathering phase [RFC5245]. The endpoint determines its server reflexive locals (i.e., the translated address of a local, on the other side of a NAT) and relayed locals (e.g., via a TURN server or other relay), for each interface and network protocol. These are added to the set of candidate locals for this association.

  Gathering locals is primarily an endpoint local operation, although it might involve exchanges with a STUN server to derive server reflexive locals, or with a TURN server or other relay to derive relayed locals. It does not involve communication with the remote.

- Resolving the Remote

  The remote is typically a name that needs to be resolved into a set of possible addresses that can be used for communication. Resolving the remote is the process of recursively performing such name lookups, until fully resolved, to return the set of candidates for the remote of this association.

How this is done will depend on the type of the Remote, and can also be specific to each local. A common case is when the Remote is a DNS name, in which case it is resolved to give a set of IPv4 and IPv6 addresses representing that name. Some types of remote might require more complex resolution. Resolving the remote for a peer-to-peer connection might involve communication with a rendezvous server, which in turn contacts the peer to gain consent to communicate and retrieve its set of candidate locals, which are returned and form the candidate remote addresses for contacting that peer.

Resolving the remote is _not_ a local operation. It will involve a directory service, and can require communication with the remote to rendezvous and exchange peer addresses. This can expose some or all of the candidate locals to the remote.

- Establishing Transients

  The set of candidate locals and the set of candidate remotes are paired, to derive a priority ordered set of Candidate Paths that can potentially be used to establish a connection.

  Then, communication is attempted over each candidate path, in priority order. If there are multiple candidates with the same priority, then transient establishment proceeds simultaneously and uses the transient that wins the race to be established. Otherwise, transients establishment is sequential, paced at a rate that should not congest the network. Depending on the chosen transport, this phase might involve racing TCP connections to a server over IPv4 and IPv6 [RFC6555], or it could involve a STUN exchange to establish peer-to-peer UDP connectivity [RFC5245], or some other means.

- Confirming and Maintaining Transients

  Once connectivity has been established, unused resources can be released and the chosen path can be confirmed. This is primarily required when establishing peer-to-peer connectivity, where connections supporting relayed locals that were not required can be closed, and where an associated signalling operation might be needed to inform middleboxes and proxies of the chosen path. Keep-alive messages may also be sent, as appropriate, to ensure NAT and firewall state is maintained, so the transient remains operational.

By encapsulating these four phases of communication establishment into the PSI, Post Sockets aims to simplify application development. It can provide reusable implementations of connection racing for TCP,

to enable happy eyeballs, that will be automatically used by all TCP clients, for example. With appropriate callbacks to drive the rendezvous signalling as part of resolving the remote, we believe a generic ICE implementation ought also to be possible. This procedure can even be repeated fully or partially during a connection to enable seamless hand-over and mobility within the network stack.

5. Acknowledgments

Many thanks to Laurent Chuat and Jason Lee at the Network Security Group at ETH Zurich for contributions to the initial design of Post Sockets. Thanks to Joe Hildebrand, Martin Thomson, and Michael Welzl for their feedback, as well as the attendees of the Post Sockets workshop in February 2017 in Zurich for the discussions, which have improved the design described herein.

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

6. References

6.1. Normative References

[I-D.ietf-taps-transports]
Fairhurst, G., Trammell, B., and M. Kuehlewind, "Services provided by IETF transport protocols and congestion control mechanisms", draft-ietf-taps-transports-14 (work in progress), December 2016.

6.2. Informative References

[I-D.ietf-quic-transport]
Iyengar, J. and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport", draft-ietf-quic-transport-07 (work in progress), October 2017.

[I-D.ietf-tls-tls13]
Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", draft-ietf-tls-tls13-21 (work in progress), July 2017.

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

[I-D.kuehlewind-taps-crypto-sep]
Kuehlewind, M., Pauly, T., and C. Wood, "Separating Crypto Negotiation and Communication", draft-kuehlewind-taps-crypto-sep-00 (work in progress), July 2017.

[I-D.pauly-taps-transport-security]
Pauly, T. and C. Wood, "A Survey of Transport Security Protocols", draft-pauly-taps-transport-security-00 (work in progress), July 2017.

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

[I-D.trammell-plus-statefulness]
Kuehlewind, M., Trammell, B., and J. Hildebrand, "Transport-Independent Path Layer State Management", draft-trammell-plus-statefulness-03 (work in progress), March 2017.

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

[NEAT] Grinnemo, K-J., Tom Jones, ., Gorry Fairhurst, ., David Ros, ., Anna Brunstrom, ., and . Per Hurtig, "Towards a Flexible Internet Transport Layer Architecture", June 2016.

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

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

[RFC5245] Rosenberg, J., "Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols", RFC 5245, DOI 10.17487/RFC5245, April 2010, <https://www.rfc-editor.org/info/rfc5245>.

[RFC6555] Wing, D. and A. Yourtchenko, "Happy Eyeballs: Success with Dual-Stack Hosts", RFC 6555, DOI 10.17487/RFC6555, April 2012, <https://www.rfc-editor.org/info/rfc6555>.

[RFC6698] Hoffman, P. and J. Schlyter, "The DNS-Based Authentication of Named Entities (DANE) Transport Layer Security (TLS) Protocol: TLSA", RFC 6698, DOI 10.17487/RFC6698, August 2012, <https://www.rfc-editor.org/info/rfc6698>.

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

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

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

[RFC7556] Anipko, D., Ed., "Multiple Provisioning Domain Architecture", RFC 7556, DOI 10.17487/RFC7556, June 2015, <https://www.rfc-editor.org/info/rfc7556>.

Appendix A. Open Issues

This document is under active development; a list of current open issues is available at https://github.com/mami-project/draft-trammell-post-sockets/issues

Authors' Addresses

Brian Trammell
ETH Zurich
Gloriastrasse 35
8092 Zurich
Switzerland

Email: ietf@trammell.ch

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

Email: csp@csperkins.org

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

Email: tpauly@apple.com

Mirja Kuehlewind
ETH Zurich
Gloriastrasse 35
8092 Zurich
Switzerland

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

Chris Wood
Apple Inc.
1 Infinite Loop
Cupertino, California 95014
United States of America

Email: cawood@apple.com