Internet Engineering Task Force (IETF)
Request for Comments: 8923
Category: Informational
ISSN: 2070-1721

M. Welzl
S. Gjessing
University of Oslo

October 2020

A Minimal Set of Transport Services for End Systems

Abstract

This document recommends a minimal set of Transport Services offered by end systems and gives guidance on choosing among the available mechanisms and protocols. It is based on the set of transport features in RFC 8303.

Status of This Memo

This document is not an Internet Standards Track specification; it is published for informational purposes.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Not all documents approved by the IESG are candidates for any level of Internet Standard; see Section 2 of RFC 7841.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at https://www.rfc-editor.org/info/rfc8923.

Table of Contents

1. Introduction

Currently, the set of Transport Services that most applications use is based on TCP and UDP (and protocols that are layered on top of them); this limits the ability for the network stack to make use of features of other transport protocols. For example, if a protocol supports out-of-order message delivery but applications always assume that the network provides an ordered byte stream, then the network stack can not immediately deliver a message that arrives out of order; doing so would break a fundamental assumption of the application. The net result is unnecessary head-of-line blocking delay.

By exposing the Transport Services of multiple transport protocols, a transport system can make it possible for applications to use these services without being statically bound 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 [RFC8303] as well as [RFC8304], which identify the specific transport features that are exposed to applications by the protocols TCP, Multipath TCP (MPTCP), UDP(-Lite), and Stream Control Transmission Protocol (SCTP), as well as the Low Extra Delay Background Transport (LEDBAT) congestion control mechanism. LEDBAT was included as the only congestion control mechanism in this list because the "low extra delay background transport" service that it offers is significantly

different from the typical service provided by other congestion control mechanisms. This memo is based on these documents and follows the same terminology (also listed below). Because the considered transport protocols conjointly cover a wide range of transport features, there is reason to hope that the resulting set (and the reasoning that led to it) will also apply to many aspects of other transport protocols that may be in use today or may be designed in the future.

By decoupling applications from transport protocols, a transport system provides a different abstraction level than the Berkeley sockets interface [POSIX]. 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 transport 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. Other transport features are offered by the APIs of the protocols covered here, but not exposing them in an API would allow for more freedom to automate protocol usage in a transport system. The minimal set presented here is an effort to find a middle ground that can be recommended for transport systems to implement, on the basis of the transport features discussed in [RFC8303].

Applications use a wide variety of APIs today. While this document was created to ensure the API developed in the Transport Services (TAPS) Working Group [TAPS-INTERFACE] includes the most important transport features, the minimal set presented here 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 library that offers its own communication interface if the underlying Berkeley Sockets API is extended to offer "unordered message delivery", but the library only exposes an ordered byte stream. Both the Berkeley Sockets API and the library would have to expose the "unordered message delivery" transport feature (alternatively, there may be ways for certain types of libraries to use this transport feature without exposing it, based on knowledge about the applications, but this is not the general case). Similarly, transport protocols such as the Stream Control Transmission Protocol (SCTP) offer multi-streaming, which cannot be utilized, e.g., to prioritize messages between streams, unless applications communicate the priorities and the group of connections upon which these priorities should be applied. In most situations, in the interest of being as flexible and efficient as possible, the best choice will be for a 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" over TCP. This means that a sender-side transport system can talk to a standard TCP receiver, and a receiver-side transport system can talk to a standard TCP sender. If certain limitations are put in place, the "minimal set" can also be implemented "one-sided" over UDP. While the possibility of such "one-sided" implementation may help deployment, it comes at the cost of limiting the set to services that can also be

provided by TCP (or, with further limitations, UDP). Thus, the minimal set of transport features here is applicable for many, but not all, applications; some application protocols have requirements that are not met by this "minimal set".

Note that, throughout this document, protocols are meant to be used natively. For example, when transport features of TCP, or "implementation over" TCP is discussed, this refers to native usage of TCP rather than TCP being encapsulated in some other transport protocol such as UDP.

2. 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, that 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.

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

Application-specific knowledge: Knowledge that only applications have.

End system: An entity that communicates with one or more other end systems using a transport protocol. An end system provides a transport-layer interface to applications.

Connection: Shared state of two or more end systems that persists across messages that are transmitted between these end systems.

Connection Group: A set of connections that share the same configuration (configuring one of them causes all other connections in the same group to be configured in the same way). We call connections that belong to a connection group "grouped", while "ungrouped" connections are not a part of a connection group.

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

Moreover, throughout the document, the protocol name "UDP(-Lite)" is used when discussing transport features that are equivalent for UDP and UDP-Lite; similarly, the protocol name "TCP" refers to both TCP and MPTCP.

3. Deriving the Minimal Set

We assume that 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 transport 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, ordered message delivery is a functional transport feature: it cannot be configured without the application knowing about it because the application's assumption could be that messages always 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 transport system autonomously decides to enable or disable them, an application will not fail, but a transport 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 utilized by a transport system on its own without involving the application are called "Automatable".

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

1. Categorization (Appendix A): The superset of transport features from [RFC8303] is presented, and transport features are categorized as Functional, Optimizing, or Automatable for later reduction.

2. Reduction (Section 4): 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 would result in semantically incorrect behavior if they were implemented over TCP or UDP.

3. Discussion (Section 5): The resulting list shows a number of peculiarities that are discussed, to provide a basis for constructing the minimal set.

4. Construction (Section 6): Based on the reduced set and the discussion of the transport features therein, a minimal set is constructed.

Following [RFC8303] and retaining its terminology, 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

4. The Reduced Set of Transport Features

By hiding automatable transport features from the application, a transport system can gain opportunities to automate the usage of network-related functionality. This can facilitate using the transport 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 transport 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 transport system should be able to communicate via TCP or UDP if alternative transport protocols are found not to work. For many transport features, this is possible, often by simply not doing anything when a specific request is made. For some transport features, however, it was identified that direct usage of neither TCP nor UDP is possible; in these cases, even not doing anything would incur semantically incorrect behavior. Whenever an application would make use of one of these transport features, this would eliminate the possibility to use TCP or UDP. Thus, we only keep the functional and optimizing transport features for which an implementation over either TCP or UDP is possible in our reduced set.

The following list contains the transport features from Appendix A, reduced using these rules. The "minimal set" derived in this document is meant to be implementable "one-sided" over TCP and, with limitations, UDP. In the list, we therefore precede a transport feature with "T:" if an implementation over TCP is possible, "U:" if an implementation over UDP is possible, and "T,U:" if an implementation over either TCP or UDP is possible.

4.1. CONNECTION-Related Transport Features

ESTABLISHMENT:

* T,U: Connect
* T,U: Specify number of attempts and/or timeout for the first establishment message

* T,U: Disable MPTCP
* T: Configure authentication
* T: Hand over a message to reliably transfer (possibly multiple times) before connection establishment
* T: Hand over a message to reliably transfer during connection establishment

AVAILABILITY:

* T,U: Listen
* T,U: Disable MPTCP
* T: Configure authentication

MAINTENANCE:

* T: Change timeout for aborting connection (using retransmit limit or time value)
* T: Suggest timeout to the peer
* T,U: Disable Nagle algorithm
* T,U: Notification of Excessive Retransmissions (early warning below abortion threshold)
* T,U: Specify DSCP field
* T,U: Notification of ICMP error message arrival
* T: Change authentication parameters
* T: Obtain authentication information
* T,U: Set Cookie life value
* T,U: Choose a scheduler to operate between streams of an association
* T,U: Configure priority or weight for a scheduler
* T,U: Disable checksum when sending
* T,U: Disable checksum requirement when receiving
* T,U: Specify checksum coverage used by the sender
* T,U: Specify minimum checksum coverage required by receiver
* T,U: Specify DF field
* T,U: Get max. transport-message size that may be sent using a non-fragmented IP packet from the configured interface
* T,U: Get max. transport-message size that may be received from the configured interface
* T,U: Obtain ECN field
* T,U: Enable and configure a "Low Extra Delay Background Transfer"

TERMINATION:

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

4.2. DATA-Transfer-Related Transport Features

4.2.1. Sending Data

* T: Reliably transfer data, with congestion control

* T: Reliably transfer a message, with congestion control
* T,U: Unreliably transfer a message
* T: Configurable Message Reliability
* T: Ordered message delivery (potentially slower than unordered)
* T,U: Unordered message delivery (potentially faster than ordered)
* T,U: Request not to bundle messages
* T: Specifying a key id to be used to authenticate a message
* T,U: Request not to delay the acknowledgement (SACK) of a message

4.2.2. Receiving Data

* T,U: Receive data (with no message delimiting)
* U: Receive a message
* T,U: 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 (Appendix A.2.1).

* T,U: Notification of send failures
* T,U: Notification that the stack has no more user data to send
* T,U: Notification to a receiver that a partial message delivery has been aborted

5. Discussion

The reduced set in the previous section exhibits a number of peculiarities, which we will discuss in the following. This section focuses on TCP because, with the exception of one particular transport feature ("Receive a message"; we will discuss this in Section 5.1), the list shows that UDP is strictly a subset of TCP. We can first try to understand how to build a transport system that can run over TCP, and then narrow down the result further to allow that the system can always run over either TCP or UDP (which effectively means removing everything related to reliability, ordering, authentication, and closing/aborting with a notification to the peer).

Note that, because the functional transport features of UDP are, with the exception of "Receive a message", a subset of TCP, TCP can be used as a replacement for UDP whenever an application does not need message delimiting (e.g., because the application-layer protocol already does it). This has been recognized by many applications that already do this in practice, by trying to communicate with UDP at first and falling back to TCP in case of a connection failure.

5.1. Sending Messages, Receiving Bytes

For implementing a transport system over TCP, there are several transport features related to sending, but only a single transport feature related to receiving: "Receive data (with no message delimiting)" (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) for which no implementation over TCP is possible.

To support these TCP receiver semantics, we define an "Application-Framed Byte Stream" (AFra Byte Stream). AFra Byte Streams 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 Byte Stream, the sending application can optionally inform the transport about message boundaries and required properties per message (configurable order and reliability, or embedding a request not to delay the acknowledgement of a message). Whenever the sending application specifies per-message properties that relax the notion of reliable in-order delivery of bytes, it must assume that the receiving application is 1) able to determine message boundaries, provided that messages are always kept intact, and 2) able to accept these relaxed per-message 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. Then, if 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 (see Section 3.5 of [RFC7305]). 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 Consistent Overhead Byte Stuffing (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 SCTP's transport features that are related to the sequence number are not provided by the "minimum set" (in the interest of enabling usage of TCP).

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

With only these semantics necessary to represent, the interface to a transport system becomes easier if we assume that connections may be not only a transport protocol's connection or association, but could also be a stream of an existing SCTP association, for example. We only need to allow for a way to define a possible grouping of connections. Then, all MAINTENANCE transport features can be said to operate on connection groups, not connections, and a scheduler operates on the connections within a group.

To be compatible with multiple transport protocols and uniformly allow access to both transport connections and streams of a multi-streaming protocol, the semantics of opening and closing need to be the most restrictive subset of all of the underlying options. 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 transport system (including streams of an association) support half-closed connections.

5.3. Early Data Transmission

There are two transport features related to transferring a message early: "Hand over a message to reliably transfer (possibly multiple times) before connection establishment", which relates to TCP Fast Open [RFC7413], and "Hand over a message to reliably 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. Also, different from TCP Fast Open, this data is not delimited as a message by TCP (thus, not visible as a "message"). 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 transport 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 messages that are explicitly marked as "idempotent" (i.e., it would be acceptable to transfer them 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 transport 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.

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 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 that was proposed in [WWDC2015], which limits the amount of unsent data that TCP can keep in the socket buffer; this allows specifying at which buffer filling level the socket becomes writable, rather than waiting for the buffer to run empty.

SCTP allows configuring 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. It therefore makes sense for a transport system to 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 transport system, they could therefore be offered in a uniform, more abstract way, where a transport 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. TCP authenticates complete segments. SCTP allows configuring 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 transport system should only allow to configure complete transport layer packets, including headers, IP pseudo-header (if any)

and payload.

Security is discussed in a separate document [RFC8922]. The minimal set presented in the present document excludes all security-related transport features from Appendix A: "Configure authentication", "Change authentication parameters", "Obtain authentication information", and "Set Cookie life value", as well as "Specifying a key id to be used to authenticate a message". It also excludes security transport features not listed in Appendix A, including content privacy to in-path devices.

5.7. Packet Size

UDP(-Lite) has a transport feature called "Specify DF field". This yields an error message in the 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.

6. The Minimal Set of Transport Features

Based on the categorization, reduction, and discussion in Section 3, this section describes a minimal set of transport features that end systems should offer. Any configuration based on the described minimum set of transport feature can always be realized over TCP but also gives the transport system flexibility to choose another transport if implemented. In the text of this section, "not UDP" is used to indicate elements of the system that cannot be implemented over UDP. Conversely, all elements of the system that are not marked with "not UDP" can also be implemented over UDP.

The arguments laid out in Section 5 ("discussion") were used to make the final representation of the minimal set as short, simple, and general as possible. There may be situations where these arguments do not apply, e.g., implementers may have specific reasons to expose multi-streaming as a visible functionality to applications, or the restrictive open/close semantics may be problematic under some circumstances. In such cases, the representation in Section 4 ("reduction") should be considered.

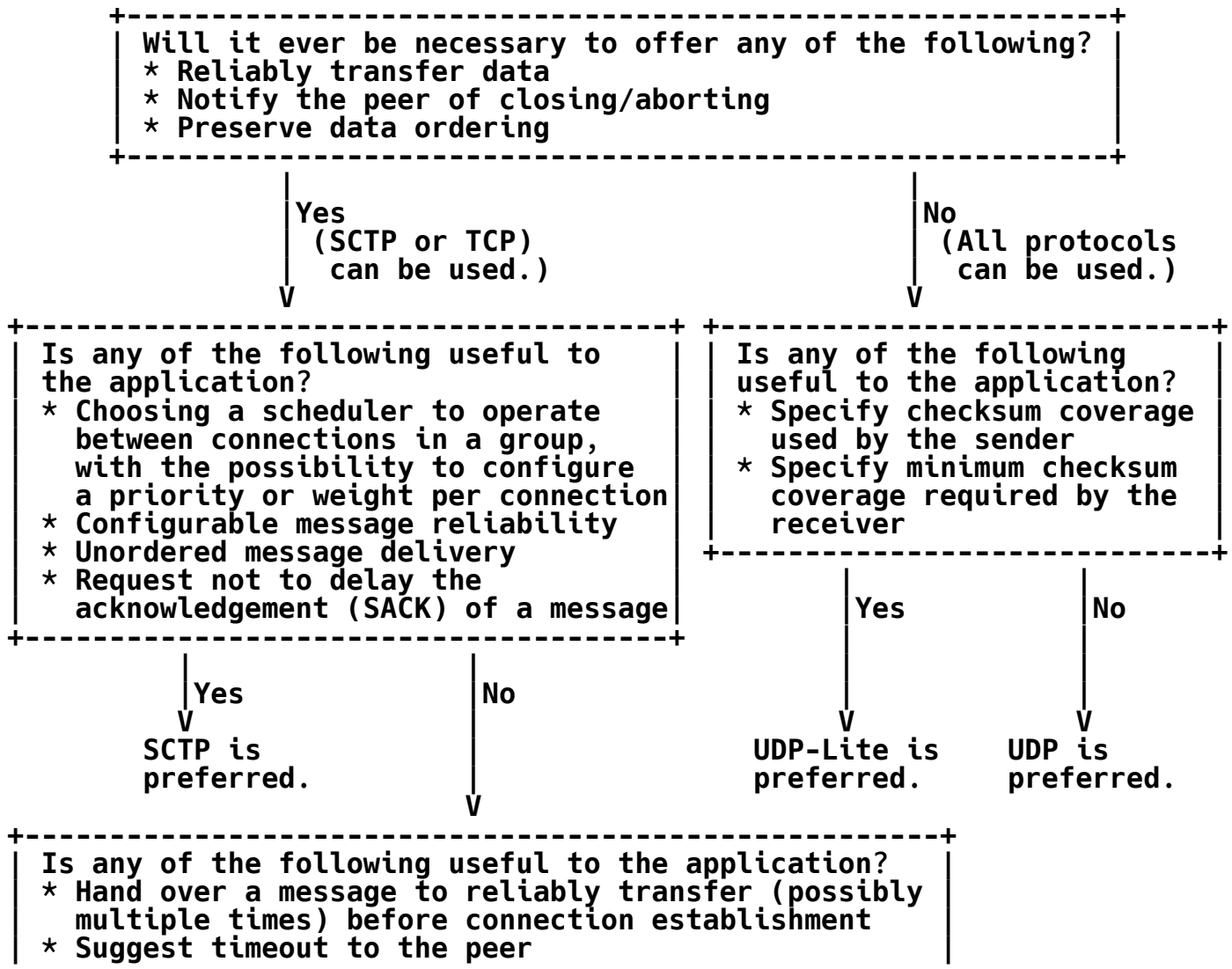
As in Section 3, Section 4, and [RFC8303], we categorize the minimal set of transport features as 1) CONNECTION related (ESTABLISHMENT, AVAILABILITY, MAINTENANCE, TERMINATION) and 2) DATA Transfer related (Sending Data, Receiving Data, Errors). Here, the focus is on connections that the transport system offers as an abstraction to the application, as opposed to connections of transport protocols that the transport system uses.

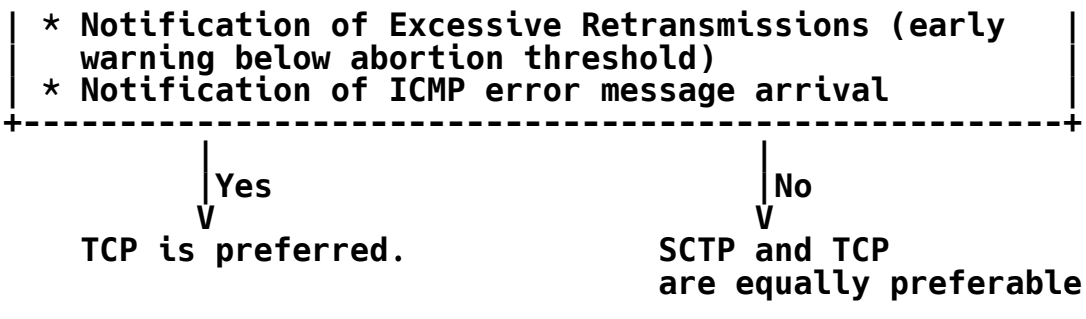
6.1. ESTABLISHMENT, AVAILABILITY, and TERMINATION

A connection must first be "created" to allow for some initial configuration to be carried out before the transport system can

actively or passively establish communication with a remote end system. As a configuration of the newly created connection, an application can choose to disallow usage of MPTCP. Furthermore, all configuration parameters in Section 6.2 can be used initially, although some of them may only take effect when a connection has been established with a chosen transport protocol. Configuring a connection early helps a transport system make the right decisions. For example, grouping information can influence whether or not the transport system implements a connection as a stream of a multi-streaming protocol's existing association.

For ungrouped connections, early configuration is necessary because it allows the transport system to know which protocols it should try to use. In particular, a transport system that only makes a one-time choice for a particular protocol must know early about strict requirements that must be kept, or it can end up in a deadlock situation (e.g., having chosen UDP and later be asked to support reliable transfer). As an example description of how to correctly handle these cases, we provide the following decision tree (this is derived from Section 4.1 excluding authentication, as explained in Section 8):





Note that this decision tree is not optimal for all cases. For example, if an application wants to use "Specify checksum coverage used by the sender", which is only offered by UDP-Lite, and "Configure priority or weight for a scheduler", which is only offered by SCTP, the above decision tree will always choose UDP-Lite, making it impossible to use SCTP's schedulers with priorities between grouped connections. Also, several other factors may influence the decisions for or against a protocol, e.g., penetration rates, the ability to work through NATs, etc. We caution implementers to be aware of the full set of trade-offs, for which we recommend consulting the list in Section 4.1 when deciding how to initialize a connection.

To summarize, the following parameters serve as input for the transport system to help it choose and configure a suitable protocol:

Reliability: a boolean that should be set to true when any of the following will be useful to the application: reliably transfer data; notify the peer of closing/aborting; or preserve data ordering.

Checksum coverage: a boolean to specify whether it will be useful to the application to specify checksum coverage when sending or receiving.

Configure message priority: a boolean that should be set to true when any of the following per-message configuration or prioritization mechanisms will be useful to the application: choosing a scheduler to operate between grouped connections, with the possibility to configure a priority or weight per connection; configurable message reliability; unordered message delivery; or requesting not to delay the acknowledgement (SACK) of a message.

Early message timeout notifications: a boolean that should be set to true when any of the following will be useful to the application: hand over a message to reliably transfer (possibly multiple times) before connection establishment; suggest timeout to the peer; notification of excessive retransmissions (early warning below abortion threshold); or notification of ICMP error message arrival.

Once a connection is created, it can be queried for the maximum amount of data that an application can possibly expect to have reliably transmitted before or during transport connection establishment (with zero being a possible answer) (see Section 6.2.1). An application can also give the connection a

message for reliable transmission before or during connection establishment (not UDP); the transport system will then try to transmit it as early as possible. An application can facilitate sending a message particularly early by marking it as "idempotent" (see Section 6.3.1); in this case, the receiving application must be prepared to potentially receive multiple copies of the message (because idempotent messages are reliably transferred, asking for idempotence is not necessary for systems that support UDP).

After creation, a transport system can actively establish communication with a peer, or it can passively listen for incoming connection requests. Note that active establishment 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 transport system could handle this by continuing to block a "Listen" call, immediately followed, for example, by issuing "Receive"; callback-based implementations could simply skip the equivalent of "Listen"). This also means that the active opening side is assumed to be the first side sending data.

A transport system can actively close a connection, i.e., terminate it after reliably delivering all remaining data to the peer (if reliable data delivery was requested earlier (not UDP)), in which case the peer is notified that the connection is closed. Alternatively, a connection can be aborted without delivering outstanding data to the peer. In case reliable or partially reliable data delivery was requested earlier (not UDP), the peer is notified that the connection is aborted. A timeout can be configured to abort a connection when data could not be delivered for too long (not UDP); however, timeout-based abortion does not notify the peer application that the connection has been aborted. Because half-closed connections are not supported, when a host implementing a transport system receives a notification that the peer is closing or aborting the connection (not UDP), its peer may not be able to read outstanding data. This means that unacknowledged data residing in a transport system's send buffer may have to be dropped from that buffer upon arrival of a "close" or "abort" notification from the peer.

6.2. MAINTENANCE

A transport system must offer means to group connections, but it cannot guarantee truly grouping them using the transport protocols that it uses (e.g., it cannot be guaranteed that connections become multiplexed as streams on a single SCTP association when SCTP may not be available). The transport system must therefore ensure that group- versus non-group-configurations are handled correctly in some way (e.g., by applying the configuration to all grouped connections even when they are not multiplexed, or informing the application about grouping success or failure).

As a general rule, any configuration described below should be carried out as early as possible to aid the transport system's decision making.

6.2.1. Connection Groups

The following transport features and notifications (some directly from Section 4; some new or changed, based on the discussion in Section 5) automatically apply to all grouped connections:

Configure a timeout (not UDP)
This can be done with the following parameters:

* A timeout value for aborting connections, in seconds.

* A timeout value to be suggested to the peer (if possible), in seconds.

* The number of retransmissions after which the application should be notified of "Excessive Retransmissions".

Configure urgency
This can be done with the following parameters:

* A number to identify the type of scheduler that should be used to operate between connections in the group (no guarantees given). Schedulers are defined in [RFC8260].

* A "capacity profile" number to identify 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", or values that help determine the DSCP value for a connection.

* A buffer limit (in bytes); when the sender has less than the provided limit of bytes in the buffer, the application may be notified. Notifications are not guaranteed, and it is optional for a transport system to support buffer limit values greater than 0. Note that this limit and its notification should operate across the buffers of the whole transport system, i.e., also any potential buffers that the transport system itself may use on top of the transport's send buffer.

Following Section 5.7, these properties can be queried:

* The maximum message size that may be sent without fragmentation via the configured interface. This is optional for a transport system to offer and may return an error ("not available"). It can aid applications implementing Path MTU Discovery.

* The maximum transport message size that can be sent, in bytes. Irrespective of fragmentation, there is a size limit for the messages that can be handed over to SCTP or UDP(-Lite); because the service provided by a transport system is independent of the transport protocol, it must allow an application to query this value: the maximum size of a message in an Application-Framed Byte Stream (see Section 5.1). This may also return an error when data is not delimited ("not available").

* The maximum transport message size that can be received from the

configured interface, in bytes (or "not available").

* The maximum amount of data that can possibly be sent before or during connection establishment, in bytes.

In addition to the already mentioned closing/aborting notifications and possible send errors, the following notifications can occur:

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 Explicit Congestion Notification (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.

Drain: The send buffer has either drained below the configured buffer limit or it has become completely empty. This 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 (for unsent data) is 0 and there is also no more unacknowledged data in the send buffer).

6.2.2. Individual Connections

Configure priority or weight for a scheduler, as described in [RFC8260].

Configure checksum usage: This can be done with the following parameters, but there is no guarantee that any checksum limitations will indeed be enforced (the default behavior is "full coverage, checksum enabled"):

* a boolean to enable/disable usage of a checksum when sending

* the desired coverage (in bytes) of the checksum used when sending

* a boolean to enable/disable requiring a checksum when receiving

* the required minimum coverage (in bytes) of the checksum when receiving

6.3. DATA Transfer

6.3.1. Sending Data

When sending a message, no guarantees are given about the preservation of message boundaries to the peer; if message boundaries are needed, the receiving application at the peer must know about

them beforehand (or the transport system cannot use TCP). Note that an application should already be able to hand over data before the transport system establishes a connection with a chosen transport protocol. Regarding the message that is being handed over, the following parameters can be used:

Reliability: This parameter is used to convey a choice of: fully reliable with congestion control (not UDP), unreliable without congestion control, unreliable with congestion control (not UDP), and partially reliable with congestion control (see [RFC3758] and [RFC7496] for details on how to specify partial reliability) (not UDP). The latter two choices are optional for a transport system to offer and may result in full reliability. Note that applications sending unreliable data without congestion control should themselves perform congestion control in accordance with [RFC8085].

Ordered (not UDP): This boolean lets an application choose between ordered message delivery (true) and possibly unordered, potentially faster message delivery (false).

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

DelAck: This boolean, if false, lets an application request that the peer not delay the acknowledgement for this message.

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

Idempotent (not UDP): This boolean expresses whether a message is idempotent (true) or not (false). Idempotent messages may arrive multiple times at the receiver (but they will arrive at least once). When data is idempotent, it can be used by the receiver immediately on a connection establishment attempt. Thus, if data is handed over before the transport system establishes a connection with a chosen transport protocol, stating that a message is idempotent facilitates transmitting it to the peer application particularly early.

An application can be notified of a failure to send a specific message. There is no guarantee of such notifications, i.e., send failures can also silently occur.

6.3.2. Receiving Data

A receiving application obtains an "Application-Framed Byte Stream" (AFra Byte Stream); this concept is further described in Section 5.1. In line with TCP's receiver semantics, an AFra Byte Stream is just a stream of bytes to the receiver. If message boundaries were specified by the sender, a receiver-side transport system implementing only the minimum set of Transport Services defined here will still not inform the receiving application about them (this limitation is only needed for transport systems that are implemented to directly use TCP).

Different from TCP's semantics, if the sending application has allowed that messages are not fully reliably transferred, or delivered out of order, then such reordering or unreliability may be reflected per message in the arriving data. Messages will always stay intact, i.e., if an incomplete message is contained at the end of the arriving data block, this message is guaranteed to continue in the next arriving data block.

7. IANA Considerations

This document has no IANA actions.

8. Security Considerations

Authentication, confidentiality protection, and integrity protection are identified as transport features by [RFC8095]. Often, these features are provided by a protocol or layer on top of the transport protocol; none of the full-featured standards-track transport protocols in [RFC8303], which this document is based upon, provide all of these transport features on its own. Therefore, they 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. The minimum requirements for a secure transport system are discussed in a separate document [RFC8922].

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, <https://www.rfc-editor.org/info/rfc8095>.

[RFC8303] Welzl, M., Tuexen, M., and N. Khademi, "On the Usage of Transport Features Provided by IETF Transport Protocols", RFC 8303, DOI 10.17487/RFC8303, February 2018, <https://www.rfc-editor.org/info/rfc8303>.

[RFC8922] Enghardt, T., Pauly, T., Perkins, C., Rose, K., and C. Wood, "A Survey of the Interaction between Security Protocols and Transport Services", RFC 8922, DOI 10.17487/RFC8922, October 2020, <https://www.rfc-editor.org/info/rfc8922>.

9.2. Informative References

[COBS] Cheshire, S. and M. Baker, "Consistent overhead byte stuffing", IEEE/ACM Transactions on Networking, Volume 7, Issue 2, DOI 10.1109/90.769765, April 1999, <https://doi.org/10.1109/90.769765>.

[POSIX] The Open Group, "IEEE Standard for Information

Technology--Portable Operating System Interface (POSIX(R)) Base Specifications, Issue 7", (Revision of IEEE Std 1003.1-2008), IEEE Std 1003.1-2017, January 2018, <https://www.opengroup.org/onlinepubs/9699919799/functions/contents.html>.

[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>.

[RFC4987] Eddy, W., "TCP SYN Flooding Attacks and Common Mitigations", RFC 4987, DOI 10.17487/RFC4987, August 2007, <https://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, <https://www.rfc-editor.org/info/rfc5925>.

[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>.

[RFC7305] Lear, E., Ed., "Report from the IAB Workshop on Internet Technology Adoption and Transition (ITAT)", RFC 7305, DOI 10.17487/RFC7305, July 2014, <https://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, <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>.

[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>.

[RFC8260] Stewart, R., Tuexen, M., Loreto, S., and R. Seggelmann, "Stream Schedulers and User Message Interleaving for the Stream Control Transmission Protocol", RFC 8260, DOI 10.17487/RFC8260, November 2017, <https://www.rfc-editor.org/info/rfc8260>.

[RFC8304] Fairhurst, G. and T. Jones, "Transport Features of the User Datagram Protocol (UDP) and Lightweight UDP (UDP-

Lite)", RFC 8304, DOI 10.17487/RFC8304, February 2018, <https://www.rfc-editor.org/info/rfc8304>.

[RFC8622] Bless, R., "A Lower-Effort Per-Hop Behavior (LE PHB) for Differentiated Services", RFC 8622, DOI 10.17487/RFC8622, June 2019, <https://www.rfc-editor.org/info/rfc8622>.

[SCTP-STREAM-1] Weinrank, F. and M. Tuexen, "Transparent Flow Mapping for NEAT", IFIP Networking 2017, Workshop on Future of Internet Transport (FIT 2017), June 2017.

[SCTP-STREAM-2] Welzl, M., Niederbacher, F., and S. Gjessing, "Beneficial Transparent Deployment of SCTP: The Missing Pieces", IEEE GlobeCom 2011, DOI 10.1109/GLOCOM.2011.6133554, December 2011, <https://doi.org/10.1109/GLOCOM.2011.6133554>.

[TAPS-INTERFACE] Trammell, B., Welzl, M., Enghardt, T., Fairhurst, G., Kuehlewind, M., Perkins, C., Tiesel, P. S., Wood, C. A., and T. Pauly, "An Abstract Application Layer Interface to Transport Services", Work in Progress, Internet-Draft, draft-ietf-taps-interface-09, 27 July 2020, <https://tools.ietf.org/html/draft-ietf-taps-interface-09>.

[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. The Superset of Transport Features

In this description, transport features are presented following the nomenclature "CATEGORY.[SUBCATEGORY].FEATURENAME.PROTOCOL", equivalent to "pass 2" in [RFC8303]. We also sketch how functional or optimizing transport features can be implemented by a transport system. The "minimal set" derived in this document is meant to be implementable "one-sided" over TCP and, with limitations, UDP. Hence, for all transport features that are categorized as "functional" or "optimizing", and for which no matching TCP and/or UDP primitive exists in "pass 2" of [RFC8303], a brief discussion on how to implement them over TCP and/or UDP 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 or not to use multi-streaming 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.

* With the exception of "Disable MPTCP", all transport features that are related to using multiple paths or the choice of the network interface were designated as "automatable". For example, "Listen" could always listen on all available interfaces and "Connect" could use the default interface for the destination IP address.

Finally, in three cases, transport features are aggregated and/or slightly changed from [RFC8303] in the description below. These transport features are marked as "CHANGED FROM RFC 8303". These do not add any new functionality but just represent a simple refactoring step that helps to streamline the derivation process (e.g., by removing a choice of a parameter for the sake of applications that may not care about this choice). The corresponding transport features are automatable, and they are listed immediately below the "CHANGED FROM RFC 8303" transport feature.

A.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 (example implementations of using multi-streaming without involving the application are described in [SCTP-STREAM-1] and [SCTP-STREAM-2]).

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.

  Implementation over UDP: do nothing (this is irrelevant in the case of UDP because there, reliable data delivery is not assumed).

* **Obtain multiple sockets**

  Protocols: SCTP

  Automatable because the non-parallel usage of multiple paths to communicate between the same end hosts relates to knowledge about the network, not the application.

* **Disable MPTCP**

  Protocols: MPTCP

  Optimizing because the parallel usage of multiple paths to communicate between the same end hosts can improve performance. Whether or not to use this feature depends on knowledge about the network as well as application-specific knowledge (see Section 3.1 of [RFC6897]).

  Implementation: via a boolean parameter in CONNECT.MPTCP.

  Implementation over TCP: do nothing.

  Implementation over UDP: 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.

With TCP, this allows configuring Master Key Tuples (MKTs) to authenticate complete segments (including the TCP IPv4 pseudoheader, TCP header, and TCP data). With SCTP, this allows specifying 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].

Implementation over UDP: not possible (UDP does not offer this functionality).

* Indicate (and/or obtain upon completion) an Adaptation Layer via an adaptation code point

  Protocols: SCTP

  Functional because it allows sending extra data for the sake of identifying an adaptation layer, which by itself is application specific.

  Implementation: via a parameter in CONNECT.SCTP.

  Implementation over TCP: not possible. (TCP does not offer this functionality.)

  Implementation over UDP: not possible. (UDP does not offer this functionality.)

* 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: controlled via a parameter in CONNECT.SCTP. One possible implementation is to always try to enable interleaving.

* Hand over a message to reliably 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.

  Implementation over UDP: not possible. (UDP does not provide reliability.)

* Hand over a message to reliably 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.

Implementation over TCP: transmit the message with the SYN packet, sacrificing the ability to identify message boundaries.

Implementation over UDP: not possible. (UDP is unreliable.)

* 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.

CHANGED FROM RFC 8303. 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.UDP(-Lite) supports both methods.

* 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

  Optimizing because the parallel usage of multiple paths to communicate between the same end hosts can improve performance. Whether or not to use this feature depends on knowledge about the network as well as application-specific knowledge (see Section 3.1 of [RFC6897]).

  Implementation: via a boolean parameter in LISTEN.MPTCP.

  Implementation over TCP: do nothing.

  Implementation over UDP: do nothing.

* Configure authentication

  Protocols: TCP, SCTP

  Functional because this has a direct influence on security.

  Implementation: via parameters in LISTEN.TCP and LISTEN.SCTP.

  Implementation over TCP: with TCP, this allows configuring Master Key Tuples (MKTs) to authenticate complete segments (including the TCP IPv4 pseudoheader, TCP header, and TCP data). With SCTP, this allows specifying 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].

  Implementation over UDP: not possible. (UDP does not offer authentication.)

* 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 sending extra data for the sake of identifying an adaptation layer, which by itself is application specific.

  Implementation: via a parameter in LISTEN.SCTP.

  Implementation over TCP: not possible. (TCP does not offer this functionality.)

  Implementation over UDP: not possible. (UDP does not offer this functionality.)

* **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.

  Implementation over UDP: not possible. (UDP is unreliable and

there is no connection timeout.)

* Suggest timeout to the peer

  Protocols: TCP

  Functional because this is closely related to potentially assumed reliable data delivery.

  Implementation: via CHANGE_TIMEOUT.TCP.

  Implementation over UDP: not possible. (UDP is unreliable and there is no connection timeout.)

* 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.

  Implementation over UDP: do nothing (UDP does not implement the Nagle algorithm).

* 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.

  Implementation over UDP: do nothing (there is no abortion threshold).

* Add path

  Protocols: MPTCP, SCTP

  MPTCP Parameters: source-IP; source-Port; destination-IP; destination-Port

  SCTP Parameters: local IP address

  Automatable because the choice of paths to communicate between the same end hosts 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 choice of paths to communicate between the same end host relates to knowledge about the network, not the application.

* Set primary path

  Protocols: SCTP

  Automatable because the choice of paths to communicate between the same end hosts relates to knowledge about the network, not the application.

* Suggest primary path to the peer

  Protocols: SCTP

  Automatable because the choice of paths to communicate between the same end hosts relates to knowledge about the network, not the application.

* Configure Path Switchover

  Protocols: SCTP

  Automatable because the choice of paths to communicate between the same end hosts 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 STATUS.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.

  Implementation over TCP: with SCTP, this allows adjusting key_id, key, and hmac_id. With TCP, this allows changing 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].

  Implementation over UDP: not possible. (UDP does not offer authentication.)

* **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 GET_AUTH.SCTP.

Implementation over TCP: with SCTP, this allows obtaining key_id and a chunk list. With TCP, this allows obtaining 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].

Implementation over UDP: not possible. (UDP does not offer authentication.)

* 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: SCTP

  Automatable because using multi-streaming does not require application-specific knowledge.

  Implementation: see Section 5.2.

* Reset Association

  Protocols: SCTP

  Automatable because deciding to reset an association does not require application-specific knowledge.

  Implementation: via RESET_ASSOC.SCTP.

* Notification of Association Reset

  Protocols: SCTP

  Automatable because this notification does not relate to application-specific knowledge.

* 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: SCTP

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 transport 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 SET_STREAM_SCHEDULER.SCTP.

  Implementation over TCP: do nothing (streams are not available in TCP, but no guarantee is given that this transport feature has any effect).

  Implementation over UDP: do nothing (streams are not available in UDP, but no guarantee is given that this transport feature has any effect).

* **Configure priority or weight for a scheduler**

  Protocols: SCTP

  Optimizing because the priority or weight requires application-specific knowledge. However, if a transport 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 CONFIGURE_STREAM_SCHEDULER.SCTP.

  Implementation over TCP: do nothing (streams are not available in TCP, but no guarantee is given that this transport feature has any effect).

  Implementation over UDP: do nothing (streams are not available in UDP, but no guarantee is given that this transport feature has any effect).

* **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 this relates to knowledge about the network and the Operating System, not the application. Note that this SCTP feature does not control IP-level fragmentation, but decides on fragmentation of messages by SCTP, in the end system.

  Implementation: done 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.

  Implementation over 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 of [RFC7413] 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]).

  Implementation over UDP: not possible. (UDP does not offer this

functionality.)

* 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.

  Implementation over TCP: not possible. (TCP does not offer identification of message boundaries.)

  Implementation over UDP: not possible. (UDP does not fragment messages.)

* Disable checksum when sending

  Protocols: UDP

  Functional because application-specific knowledge is necessary to decide whether it can be acceptable to lose data integrity with respect to random corruption.

  Implementation: via SET_CHECKSUM_ENABLED.UDP.

  Implementation over TCP: do nothing (TCP does not offer to disable the checksum, but transmitting data with an intact checksum will not yield a semantically wrong result).

* 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 with respect to random corruption.

  Implementation: via SET_CHECKSUM_REQUIRED.UDP.

  Implementation over TCP: do nothing (TCP does not offer to disable the checksum, but transmitting data with an intact checksum will not yield a semantically wrong result).

* 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 with respect to random corruption.

Implementation: via SET_CHECKSUM_COVERAGE.UDP-Lite.

Implementation over TCP: do nothing (TCP does not offer to limit the checksum length, but transmitting data with an intact checksum will not yield a semantically wrong result).

Implementation over UDP: if checksum coverage is set to cover payload data, do nothing. Else, either do nothing (transmitting data with an intact checksum will not yield a semantically wrong result), or use the transport feature "Disable checksum when sending".

* 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 with respect to random corruption.

Implementation: via SET_MIN_CHECKSUM_COVERAGE.UDP-Lite.

Implementation over TCP: do nothing (TCP does not offer to limit the checksum length, but transmitting data with an intact checksum will not yield a semantically wrong result).

Implementation over UDP: if checksum coverage is set to cover payload data, do nothing. Else, either do nothing (transmitting data with an intact checksum will not yield a semantically wrong result), or use the transport feature "Disable checksum requirement when receiving".

* 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).

Implementation over TCP: do nothing (with TCP, the sending application 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.

Implementation over TCP: do nothing (this information is not available with TCP).

* 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.

  Implementation over TCP: do nothing (this information is not available with TCP).

* Specify TTL/Hop count field

  Protocols: UDP(-Lite)

  Automatable because a transport 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 that 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).

  Implementation over TCP: do nothing (this information is not available with TCP).

* 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 feature 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 transport system's 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) [RFC8622].

  Implementation over TCP: do nothing (TCP does not support LEDBAT congestion control, but not implementing this functionality will not yield a semantically wrong behavior).

  Implementation over UDP: do nothing (UDP does not offer congestion control).

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.

  Implementation over UDP: not possible. (UDP is unreliable and hence does not know when all remaining data is delivered; it does also not offer to cause an event related to closing at the peer.)

* 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.

Implementation over UDP: not possible. (UDP does not offer to cause an event related to aborting at the peer.)

* 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).

Implementation over 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.

Implementation over UDP: do nothing (this event will not occur with UDP).

A.2. DATA-Transfer-Related Transport Features

A.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.

Implementation over UDP: not possible. (UDP is unreliable.)

* 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.

Implementation over TCP: via SEND.TCP. With SEND.TCP, message boundaries will not be identifiable by the receiver, because TCP provides a byte-stream service.

Implementation over UDP: not possible. (UDP is unreliable.)

* Unreliably transfer a message

Protocols: SCTP, UDP(-Lite)

Optimizing because only applications know about the time criticality of their communication, and reliably transferring a message is never incorrect for the receiver of a potentially unreliable data transfer, it is just slower.

CHANGED FROM RFC 8303. 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(-Lite).

Implementation over TCP: use SEND.TCP. With SEND.TCP, messages will be sent reliably, and message boundaries will not be identifiable by the receiver.

* Unreliably transfer a message, with congestion control

Protocols: SCTP

Automatable because congestion control relates to knowledge about the network, not the application.

* Unreliably transfer a message, without congestion control

Protocols: UDP(-Lite)

Automatable because congestion control relates to knowledge about the network, not the application.

* Configurable Message Reliability

Protocols: SCTP

Optimizing because only applications know about the time criticality of their communication, and reliably transferring a message is never incorrect for the receiver of a potentially unreliable data transfer, it is just slower.

Implementation: via SEND.SCTP.

Implementation over TCP: done 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.

Implementation over UDP: not possible. (UDP is unreliable.)

* 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.

* Ordered message delivery (potentially slower than unordered)

  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.

  Implementation over TCP: done by using SEND.TCP. With SEND.TCP, messages will not be identifiable by the receiver.

  Implementation over UDP: not possible. (UDP does not offer any guarantees regarding ordering.)

* Unordered message delivery (potentially faster than ordered)

  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 SEND.SCTP.

  Implementation over TCP: done 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.

  Implementation over TCP: done 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.

  Implementation over UDP: do nothing (UDP never bundles messages).

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

  Protocols: SCTP

  Functional because it allows sending extra application data with every message, for the sake of identification of data, which by itself is application specific.

  Implementation: SEND.SCTP.

  Implementation over TCP: not possible. (This functionality is not available in TCP.)

  Implementation over UDP: not possible. (This functionality is not available in UDP.)

* 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.

  Implementation over 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.

  Implementation over UDP: not possible. (UDP does not offer authentication.)

* 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.

Implementation over TCP: do nothing (TCP does not offer this functionality, but ignoring this request from the application will not yield a semantically wrong behavior).

Implementation over UDP: do nothing (UDP does not offer this functionality, but ignoring this request from the application will not yield a semantically wrong behavior).

A.2.2. Receiving Data

* Receive data (with no message delimiting)

  Protocols: TCP

  Functional because a transport system must be able to send and receive data.

  Implementation: via RECEIVE.TCP.

  Implementation over UDP: do nothing (UDP only works on messages; these can be handed over, the application can still ignore the message boundaries).

* 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).

  Implementation over TCP: not possible. (TCP does not support identification of message boundaries.)

* 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.

Implementation over TCP: do nothing (this information is not available with TCP).

Implementation over UDP: do nothing (this information is not available with UDP).

A.2.3. Errors

This section describes sending failures that are associated with a specific call to in the "Sending Data" category (Appendix A.2.1).

* Notification of send failures

  Protocols: SCTP, UDP(-Lite)

  Functional because this notifies that potentially assumed reliable data delivery is no longer provided.

  CHANGED FROM RFC 8303. This differs from the 2 automatable transport features below in that it does not distinguish between unsent and unacknowledged messages.

  Implementation: via SENDFAILURE-EVENT.SCTP and SEND_FAILURE.UDP(-Lite).

  Implementation over 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 does not relate to application-specific knowledge.

* Notification of an unacknowledged (part of a) message

  Protocols: SCTP

  Automatable because the distinction between unsent and unacknowledged does not relate to application-specific knowledge.

* 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.

  Implementation over TCP: do nothing (see the discussion in Section 5.4).

Implementation over UDP: do nothing (this notification is not available and will therefore not occur with UDP).

* 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.

  Implementation over TCP: do nothing (this notification is not available and will therefore not occur with TCP).

  Implementation over UDP: do nothing (this notification is not available and will therefore not occur with UDP).

Acknowledgements

The authors would like to thank all the participants of the TAPS Working Group and the NEAT and MAMI research projects for valuable input to this document. We especially thank Michael Tüxen for help with connection establishment/teardown, Gorry Fairhurst for his suggestions regarding fragmentation and packet sizes, and Spencer Dawkins for his extremely detailed and constructive review. This work has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No. 644334 (NEAT).

Authors' Addresses

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

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

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

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