TAPS Working Group
Internet-Draft
Intended status: Informational
Expires: 14 January 2021

A. Brunstrom, Ed.
Karlstad University
T. Pauly, Ed.
Apple Inc.
T. Enghardt
Netflix
K-J. Grinnemo
Karlstad University
T. Jones
University of Aberdeen
P. Tiesel
TU Berlin
C. Perkins
University of Glasgow
M. Welzl
University of Oslo
13 July 2020

Implementing Interfaces to Transport Services
draft-ietf-taps-impl-07

Abstract

The Transport Services (TAPS) system enables applications to use transport protocols flexibly for network communication and defines a protocol-independent TAPS Application Programming Interface (API) that is based on an asynchronous, event-driven interaction pattern. This document serves as a guide to implementation on how to build such a system.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

The Transport Services architecture [I-D.ietf-taps-arch] defines a system that allows applications to use transport networking protocols flexibly. The interface such a system exposes to applications is defined as the Transport Services API [I-D.ietf-taps-interface]. This API is designed to be generic across multiple transport protocols and sets of protocols features.

This document serves as a guide to implementation on how to build a system that provides a Transport Services API. It is the job of an implementation of a Transport Services system to turn the requests of an application into decisions on how to establish connections, and how to transfer data over those connections once established. The terminology used in this document is based on the Architecture [I-D.ietf-taps-arch].

2. Implementing Connection Objects

The connection objects that are exposed to applications for Transport Services are:

* the Preconnection, the bundle of Properties that describes the application constraints on the transport;

* the Connection, the basic object that represents a flow of data as Messages in either direction between the Local and Remote Endpoints;

* and the Listener, a passive waiting object that delivers new Connections.

Preconnection objects should be implemented as bundles of properties that an application can both read and write. Once a Preconnection has been used to create an outbound Connection or a Listener, the implementation should ensure that the copy of the properties held by the Connection or Listener is immutable. This may involve performing a deep-copy if the application is still able to modify properties on the original Preconnection object.

Connection objects represent the interface between the application and the implementation to manage transport state, and conduct data transfer. During the process of establishment (Section 4), the Connection will be unbound to a specific transport flow, since there may be multiple candidate Protocol Stacks being raced. Once the Connection is established, the object should be considered mapped to a specific Protocol Stack. The notion of a Connection maps to many different protocols, depending on the Protocol Stack. For example, the Connection may ultimately represent the interface into a TCP connection, a TLS session over TCP, a UDP flow with fully-specified local and remote endpoints, a DTLS session, a SCTP stream, a QUIC stream, or an HTTP/2 stream.

Listener objects are created with a Preconnection, at which point their configuration should be considered immutable by the implementation. The process of listening is described in Section 4.6.

3. Implementing Pre-Establishment

During pre-establishment the application specifies the Endpoints to be used for communication as well as its preferences via Selection Properties and, if desired, also Connection Properties. Generally, Connection Properties should be configured as early as possible, because they can serve as input to decisions that are made by the implementation (e.g., the Capacity Profile can guide usage of a protocol offering scavenger-type congestion control).

The implementation stores these properties as a part of the Preconnection object for use during connection establishment. For Selection Properties that are not provided by the application, the implementation must use the default values specified in the Transport Services API ([I-D.ietf-taps-interface]).

3.1. Configuration-time errors

The transport system should have a list of supported protocols available, which each have transport features reflecting the capabilities of the protocol. Once an application specifies its Transport Properties, the transport system matches the required and prohibited properties against the transport features of the available protocols.

In the following cases, failure should be detected during pre-establishment:

* A request by an application for Protocol Properties that include requirements or prohibitions that cannot be satisfied by any of the available protocols. For example, if an application requires "Configure Reliability per Message", but no such protocol is available on the host running the transport system this should result in an error, e.g., when SCTP is not supported by the operating system.

* A request by an application for Protocol Properties that are in conflict with each other, i.e., the required and prohibited properties cannot be satisfied by the same protocol. For example, if an application prohibits "Reliable Data Transfer" but then requires "Configure Reliability per Message", this mismatch should result in an error.

To avoid allocating resources, it is important that such cases fail as early as possible, e.g., to endpoint resolution, only to find out later that there is no protocol that satisfies the requirements.

3.2. Role of system policy

The properties specified during pre-establishment have a close relationship to system policy. The implementation is responsible for combining and reconciling several different sources of preferences when establishing Connections. These include, but are not limited to:

1. Application preferences, i.e., preferences specified during the pre-establishment via Selection Properties.

2. Dynamic system policy, i.e., policy compiled from internally and externally acquired information about available network interfaces, supported transport protocols, and current/previous Connections. Examples of ways to externally retrieve policy-support information are through OS-specific statistics/ measurement tools and tools that reside on middleboxes and routers.

3. Default implementation policy, i.e., predefined policy by OS or application.

In general, any protocol or path used for a connection must conform to all three sources of constraints. A violation of any of the layers should cause a protocol or path to be considered ineligible for use. For an example of application preferences leading to constraints, an application may prohibit the use of metered network interfaces for a given Connection to avoid user cost. Similarly, the system policy at a given time may prohibit the use of such a metered network interface from the application's process. Lastly, the implementation itself may default to disallowing certain network interfaces unless explicitly requested by the application and allowed by the system.

It is expected that the database of system policies and the method of looking up these policies will vary across various platforms. An implementation should attempt to look up the relevant policies for the system in a dynamic way to make sure it is reflecting an accurate version of the system policy, since the system's policy regarding the application's traffic may change over time due to user or administrative changes.

4. Implementing Connection Establishment

The process of establishing a network connection begins when an application expresses intent to communicate with a remote endpoint by calling Initiate. (At this point, any constraints or requirements the application may have on the connection are available from pre-establishment.) The process can be considered complete once there is at least one Protocol Stack that has completed any required setup to the point that it can transmit and receive the application's data.

Connection establishment is divided into two top-level steps: Candidate Gathering, to identify the paths, protocols, and endpoints to use, and Candidate Racing, in which the necessary protocol handshakes are conducted so that the transport system can select which set to use. This document structures candidates for racing as a tree.

The most simple example of this process might involve identifying the single IP address to which the implementation wishes to connect, using the system's current default interface or path, and starting a TCP handshake to establish a stream to the specified IP address. However, each step may also vary depending on the requirements of the connection: if the endpoint is defined as a hostname and port, then there may be multiple resolved addresses that are available; there may also be multiple interfaces or paths available, other than the default system interface; and some protocols may not need any transport handshake to be considered "established" (such as UDP), while other connections may utilize layered protocol handshakes, such as TLS over TCP.

Whenever an implementation has multiple options for connection establishment, it can view the set of all individual connection establishment options as a single, aggregate connection establishment. The aggregate set conceptually includes every valid combination of endpoints, paths, and protocols. As an example, consider an implementation that initiates a TCP connection to a hostname + port endpoint, and has two valid interfaces available (Wi-Fi and LTE). The hostname resolves to a single IPv4 address on the Wi-Fi network, and resolves to the same IPv4 address on the LTE network, as well as a single IPv6 address. The aggregate set of connection establishment options can be viewed as follows:

```
Aggregate [Endpoint: www.example.com:80] [Interface: Any]   [Protocol: TCP]
|-> [Endpoint: 192.0.2.1:80]       [Interface: Wi-Fi] [Protocol: TCP]
|-> [Endpoint: 192.0.2.1:80]       [Interface: LTE]   [Protocol: TCP]
|-> [Endpoint: 2001:DB8::1.80]     [Interface: LTE]   [Protocol: TCP]
```

Any one of these sub-entries on the aggregate connection attempt would satisfy the original application intent. The concern of this section is the algorithm defining which of these options to try, when, and in what order.

During Candidate Gathering, an implementation first excludes all protocols and paths that match a Prohibit or do not match all Require properties. Then, the implementation will sort branches according to Preferred properties, Avoided properties, and possibly other criteria.

4.1. Candidate Gathering

The step of gathering candidates involves identifying which paths, protocols, and endpoints may be used for a given Connection. This list is determined by the requirements, prohibitions, and preferences of the application as specified in the Selection Properties.

4.1.1. Gathering Endpoint Candidates

Both Local and Remote Endpoint Candidates must be discovered during connection establishment. To support Interactive Connectivity Establishment (ICE) [RFC8445], or similar protocols, that involve out-of-band indirect signalling to exchange candidates with the Remote Endpoint, it's important to be able to query the set of candidate Local Endpoints, and give the protocol stack a set of candidate Remote Endpoints, before it attempts to establish connections.

4.1.1.1. Local Endpoint candidates

The set of possible Local Endpoints is gathered. In the simple case, this merely enumerates the local interfaces and protocols, allocates ephemeral source ports. 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 Local Endpoints becomes broadly equivalent to the ICE candidate gathering phase (see Section 5.1.1. of [RFC8445]). The endpoint determines its server reflexive Local Endpoints (i.e., the translated address of a local, on the other side of a NAT, e.g via a STUN sever [RFC5389]) and relayed locals (e.g., via a TURN server [RFC5766] or other relay), for each interface and network protocol. These are added to the set of candidate Local Endpoints for this connection.

Gathering Local Endpoints is primarily a 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. However, it does not involve communication with the Remote Endpoint.

4.1.1.2. Remote Endpoint Candidates

The Remote Endpoint is typically a name that needs to be resolved into a set of possible addresses that can be used for communication. Resolving the Remote Endpoint is the process of recursively performing such name lookups, until fully resolved, to return the set of candidates for the remote of this connection.

How this is done will depend on the type of the Remote Endpoint, and can also be specific to each Local Endpoint. A common case is when the Remote Endpoint 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 Endpoint 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.

4.1.2. Structuring Options as a Tree

When an implementation responsible for connection establishment needs to consider multiple options, it should logically structure these options as a hierarchical tree. Each leaf node of the tree represents a single, coherent connection attempt, with an Endpoint, a Path, and a set of protocols that can directly negotiate and send data on the network. Each node in the tree that is not a leaf represents a connection attempt that is either underspecified, or else includes multiple distinct options. For example, when connecting on an IP network, a connection attempt to a hostname and port is underspecified, because the connection attempt requires a resolved IP address as its remote endpoint. In this case, the node represented by the connection attempt to the hostname is a parent node, with child nodes for each IP address. Similarly, an implementation that is allowed to connect using multiple interfaces will have a parent node of the tree for the decision between the paths, with a branch for each interface.

The example aggregate connection attempt above can be drawn as a tree by grouping the addresses resolved on the same interface into branches:

```
                             ||
                +==========================+
                |  www.example.com:80/Any  |
                +==========================+
                  //                    \\
+==========================+       +==========================+
| www.example.com:80/Wi-Fi |       |  www.example.com:80/LTE  |
+==========================+       +==========================+
             ||                      //                    \\
  +====================+  +====================+  +======================+
  | 192.0.2.1:80/Wi-Fi |  |  192.0.2.1:80/LTE  |  |  2001:DB8::1.80/LTE  |
  +====================+  +====================+  +======================+
```

The rest of this section will use a notation scheme to represent this tree. The parent (or trunk) node of the tree will be represented by a single integer, such as "1". Each child of that node will have an integer that identifies it, from 1 to the number of children. That child node will be uniquely identified by concatenating its integer to it’s parents identifier with a dot in between, such as "1.1" and "1.2". Each node will be summarized by a tuple of three elements: Endpoint, Path, and Protocol. The above example can now be written more succinctly as:

```
1 [www.example.com:80, Any, TCP]
  1.1 [www.example.com:80, Wi-Fi, TCP]
    1.1.1 [192.0.2.1:80, Wi-Fi, TCP]
  1.2 [www.example.com:80, LTE, TCP]
    1.2.1 [192.0.2.1:80, LTE, TCP]
    1.2.2 [2001:DB8::1.80, LTE, TCP]
```

When an implementation views this aggregate set of connection attempts as a single connection establishment, it only will use one of the leaf nodes to transfer data. Thus, when a single leaf node becomes ready to use, then the entire connection attempt is ready to use by the application. Another way to represent this is that every leaf node updates the state of its parent node when it becomes ready, until the trunk node of the tree is ready, which then notifies the application that the connection as a whole is ready to use.

A connection establishment tree may be degenerate, and only have a single leaf node, such as a connection attempt to an IP address over a single interface with a single protocol.

```
1 [192.0.2.1:80, Wi-Fi, TCP]
```

A parent node may also only have one child (or leaf) node, such as a when a hostname resolves to only a single IP address.

```
1 [www.example.com:80, Wi-Fi, TCP]
  1.1 [192.0.2.1:80, Wi-Fi, TCP]
```

4.1.3. Branch Types

There are three types of branching from a parent node into one or more child nodes. Any parent node of the tree must only use one type of branching.

4.1.3.1. Derived Endpoints

If a connection originally targets a single endpoint, there may be multiple endpoints of different types that can be derived from the original. The connection library creates an ordered list of the derived endpoints according to application preference, system policy and expected performance.

DNS hostname-to-address resolution is the most common method of endpoint derivation. When trying to connect to a hostname endpoint on a traditional IP network, the implementation should send DNS queries for both A (IPv4) and AAAA (IPv6) records if both are supported on the local link. The algorithm for ordering and racing these addresses should follow the recommendations in Happy Eyeballs [RFC8305].

```
1 [www.example.com:80, Wi-Fi, TCP]
  1.1 [2001:DB8::1.80, Wi-Fi, TCP]
  1.2 [192.0.2.1:80, Wi-Fi, TCP]
  1.3 [2001:DB8::2.80, Wi-Fi, TCP]
  1.4 [2001:DB8::3.80, Wi-Fi, TCP]
```

DNS-Based Service Discovery [RFC6763] can also provide an endpoint derivation step. When trying to connect to a named service, the client may discover one or more hostname and port pairs on the local network using multicast DNS [RFC6762]. These hostnames should each be treated as a branch that can be attempted independently from other hostnames. Each of these hostnames might resolve to one or more addresses, which would create multiple layers of branching.

```
1 [term-printer._ipp._tcp.meeting.ietf.org, Wi-Fi, TCP]
  1.1 [term-printer.meeting.ietf.org:631, Wi-Fi, TCP]
    1.1.1 [31.133.160.18.631, Wi-Fi, TCP]
```

4.1.3.2. Alternate Paths

If a client has multiple network interfaces available to it, e.g., a mobile client with both Wi-Fi and Cellular connectivity, it can attempt a connection over any of the interfaces. This represents a branch point in the connection establishment. Similar to a derived endpoint, the interfaces should be ranked based on preference, system policy, and performance. Attempts should be started on one interface, and then on other interfaces successively after delays based on expected round-trip-time or other available metrics.

```
1 [192.0.2.1:80, Any, TCP]
  1.1 [192.0.2.1:80, Wi-Fi, TCP]
  1.2 [192.0.2.1:80, LTE, TCP]
```

This same approach applies to any situation in which the client is aware of multiple links or views of the network. Multiple Paths, each with a coherent set of addresses, routes, DNS server, and more, may share a single interface. A path may also represent a virtual interface service such as a Virtual Private Network (VPN).

The list of available paths should be constrained by any requirements or prohibitions the application sets, as well as system policy.

4.1.3.3. Protocol Options

Differences in possible protocol compositions and options can also provide a branching point in connection establishment. This allows clients to be resilient to situations in which a certain protocol is not functioning on a server or network.

This approach is commonly used for connections with optional proxy server configurations. A single connection might have several options available: an HTTP-based proxy, a SOCKS-based proxy, or no proxy. These options should be ranked and attempted in succession.

```
1 [www.example.com:80, Any, HTTP/TCP]
  1.1 [192.0.2.8:80, Any, HTTP/HTTP Proxy/TCP]
  1.2 [192.0.2.7:10234, Any, HTTP/SOCKS/TCP]
  1.3 [www.example.com:80, Any, HTTP/TCP]
    1.3.1 [192.0.2.1:80, Any, HTTP/TCP]
```

This approach also allows a client to attempt different sets of application and transport protocols that, when available, could provide preferable features. For example, the protocol options could involve QUIC [I-D.ietf-quic-transport] over UDP on one branch, and HTTP/2 [RFC7540] over TLS over TCP on the other:

```
1 [www.example.com:443, Any, Any HTTP]
  1.1 [www.example.com:443, Any, QUIC/UDP]
    1.1.1 [192.0.2.1:443, Any, QUIC/UDP]
  1.2 [www.example.com:443, Any, HTTP2/TLS/TCP]
    1.2.1 [192.0.2.1:443, Any, HTTP2/TLS/TCP]
```

Another example is racing SCTP with TCP:

```
1 [www.example.com:80, Any, Any Stream]
  1.1 [www.example.com:80, Any, SCTP]
    1.1.1 [192.0.2.1:80, Any, SCTP]
  1.2 [www.example.com:80, Any, TCP]
    1.2.1 [192.0.2.1:80, Any, TCP]
```

Implementations that support racing protocols and protocol options should maintain a history of which protocols and protocol options successfully established, on a per-network and per-endpoint basis (see Section 9.2). This information can influence future racing decisions to prioritize or prune branches.

4.1.4. Branching Order-of-Operations

Branch types must occur in a specific order relative to one another to avoid creating leaf nodes with invalid or incompatible settings. In the example above, it would be invalid to branch for derived endpoints (the DNS results for www.example.com) before branching between interface paths, since there are situations when the results will be different across networks due to private names or different supported IP versions. Implementations must be careful to branch in an order that results in usable leaf nodes whenever there are multiple branch types that could be used from a single node.

The order of operations for branching, where lower numbers are acted upon first, should be:

1. Alternate Paths

2. Protocol Options

3. Derived Endpoints

Branching between paths is the first in the list because results across multiple interfaces are likely not related to one another: endpoint resolution may return different results, especially when using locally resolved host and service names, and which protocols are supported and preferred may differ across interfaces. Thus, if multiple paths are attempted, the overall connection can be seen as a race between the available paths or interfaces.

Protocol options are next checked in order. Whether or not a set of protocol, or protocol-specific options, can successfully connect is generally not dependent on which specific IP address is used. Furthermore, the protocol stacks being attempted may influence or altogether change the endpoints being used. Adding a proxy to a connection's branch will change the endpoint to the proxy's IP address or hostname. Choosing an alternate protocol may also modify the ports that should be selected.

Branching for derived endpoints is the final step, and may have multiple layers of derivation or resolution, such as DNS service resolution and DNS hostname resolution.

For example, if the application has indicated both a preference for WiFi over LTE and for a feature only available in SCTP, branches will be first sorted accord to path selection, with WiFi at the top. Then, branches with SCTP will be sorted to the top within their subtree according to the properties influencing protocol selection. However, if the implementation has current cache information that SCTP is not available on the path over WiFi, there is no SCTP node in the WiFi subtree. Here, the path over WiFi will be tried first, and, if connection establishment succeeds, TCP will be used. So the Selection Property of preferring WiFi takes precedence over the Property that led to a preference for SCTP.

```
1. [www.example.com:80, Any, Any Stream]
1.1 [192.0.2.1:80, Wi-Fi, Any Stream]
1.1.1 [192.0.2.1:80, Wi-Fi, TCP]
1.2 [192.0.3.1:80, LTE, Any Stream]
1.2.1 [192.0.3.1:80, LTE, SCTP]
1.2.2 [192.0.3.1:80, LTE, TCP]
```

4.1.5. Sorting Branches

Implementations should sort the branches of the tree of connection options in order of their preference rank, from most preferred to least preferred. Leaf nodes on branches with higher rankings represent connection attempts that will be raced first. Implementations should order the branches to reflect the preferences expressed by the application for its new connection, including Selection Properties, which are specified in [I-D.ietf-taps-interface].

In addition to the properties provided by the application, an implementation may include additional criteria such as cached performance estimates, see Section 9.2, or system policy, see Section 3.2, in the ranking. Two examples of how Selection and Connection Properties may be used to sort branches are provided below:

* "Interface Instance or Type": If the application specifies an interface type to be preferred or avoided, implementations should accordingly rank the paths. If the application specifies an interface type to be required or prohibited, an implementation is expeceted to not include the non-conforming paths.

* "Capacity Profile": An implementation can use the Capacity Profile to prefer paths that match an application's expected traffic pattern. This match will use cached performance estimates, see Section 9.2:

  - Scavenger: Prefer paths with the highest expected available capacity, based on the observed maximum throughput;

  - Low Latency/Interactive: Prefer paths with the lowest expected Round Trip Time, based on observed round trip time estimates;

  - Constant-Rate Streaming: Prefer paths that can are expected to satisy the requested Stream Send or Stream Receive Bitrate, based on the observed maximum throughput.

Implementations process the Properties in the following order: Prohibit, Require, Prefer, Avoid. If Selection Properties contain any prohibited properties, the implementation should first purge branches containing nodes with these properties. For required properties, it should only keep branches that satisfy these requirements. Finally, it should order the branches according to the preferred properties, and finally use any avoided properties as a tiebreaker. When ordering branches, an implementation can give more weight to properties that the application has explicitly set, than to the properties that are default.

The available protocols and paths on a specific system and in a specific context can change; therefore, the result of sorting and the outcome of racing may vary, even when using the same Selection and Connection Properties. However, an implementation ought to provide a consistent outcome to applications, e.g., by preferring protocols and paths that are already used by existing Connections that specified similar Properties.

4.2. Candidate Racing

The primary goal of the Candidate Racing process is to successfully negotiate a protocol stack to an endpoint over an interface--to connect a single leaf node of the tree--with as little delay and as few unnecessary connections attempts as possible. Optimizing these two factors improves the user experience, while minimizing network load.

This section covers the dynamic aspect of connection establishment. The tree described above is a useful conceptual and architectural model. However, an implementation is unable to know the full tree before it is formed and many of the possible branches ultimately might not be used.

There are three different approaches to racing the attempts for different nodes of the connection establishment tree:

1. Immediate

2. Delayed

3. Failover

Each approach is appropriate in different use-cases and branch types. However, to avoid consuming unnecessary network resources, implementations should not use immediate racing as a default approach.

The timing algorithms for racing should remain independent across branches of the tree. Any timers or racing logic is isolated to a given parent node, and is not ordered precisely with regards to other children of other nodes.

4.2.1. Immediate

Immediate racing is when multiple alternate branches are started without waiting for any one branch to make progress before starting the next alternative. This means the attempts are effectively simultaneous. Immediate racing should be avoided by implementations, since it consumes extra network resources and establishes state that might not be used.

4.2.2. Delayed

Delayed racing can be used whenever a single node of the tree has multiple child nodes. Based on the order determined when building the tree, the first child node will be initiated immediately, followed by the next child node after some delay. Once that second child node is initiated, the third child node (if present) will begin after another delay, and so on until all child nodes have been initiated, or one of the child nodes successfully completes its negotiation.

Delayed racing attempts occur in parallel. Implementations should not terminate an earlier child connection attempt upon starting a secondary child.

The delay between starting child nodes should be based on the properties of the previously started child node. For example, if the first child represents an IP address with a known route, and the second child represents another IP address, the delay between starting the first and second IP addresses can be based on the expected retransmission cadence for the first child's connection (derived from historical round-trip-time). Alternatively, if the first child represents a branch on a Wi-Fi interface, and the second child represents a branch on an LTE interface, the delay should be based on the expected time in which the branch for the first interface would be able to establish a connection, based on link quality and historical round-trip-time.

Any delay should have a defined minimum and maximum value based on the branch type. Generally, branches between paths and protocols should have longer delays than branches between derived endpoints. The maximum delay should be considered with regards to how long a user is expected to wait for the connection to complete.

If a child node fails to connect before the delay timer has fired for the next child, the next child should be started immediately.

4.2.3. Failover

If an implementation or application has a strong preference for one branch over another, the branching node may choose to wait until one child has failed before starting the next. Failure of a leaf node is determined by its protocol negotiation failing or timing out; failure of a parent branching node is determined by all of its children failing.

An example in which failover is recommended is a race between a protocol stack that uses a proxy and a protocol stack that bypasses the proxy. Failover is useful in case the proxy is down or misconfigured, but any more aggressive type of racing may end up unnecessarily avoiding a proxy that was preferred by policy.

4.3. Completing Establishment

The process of connection establishment completes when one leaf node of the tree has completed negotiation with the remote endpoint successfully, or else all nodes of the tree have failed to connect. The first leaf node to complete its connection is then used by the application to send and receive data.

Successes and failures of a given attempt should be reported up to parent nodes (towards the trunk of the tree). For example, in the following case, if 1.1.1 fails to connect, it reports the failure to 1.1. Since 1.1 has no other child nodes, it also has failed and reports that failure to 1. Because 1.2 has not yet failed, 1 is not considered to have failed. Since 1.2 has not yet started, it is started and the process continues. Similarly, if 1.1.1 successfully connects, then it marks 1.1 as connected, which propagates to the trunk node 1. At this point, the connection as a whole is considered to be successfully connected and ready to process application data

```
1 [www.example.com:80, Any, TCP]
  1.1 [www.example.com:80, Wi-Fi, TCP]
    1.1.1 [192.0.2.1:80, Wi-Fi, TCP]
  1.2 [www.example.com:80, LTE, TCP]
...
```

If a leaf node has successfully completed its connection, all other attempts should be made ineligible for use by the application for the original request. New connection attempts that involve transmitting data on the network ought not to be started after another leaf node has already successfully completed, because the connection as a whole has now been established. An implementation may choose to let certain handshakes and negotiations complete in order to gather metrics to influence future connections. Keeping additional connections is generally not recommended since those attempts were slower to connect and may exhibit less desirable properties.

4.3.1. Determining Successful Establishment

Implementations may select the criteria by which a leaf node is considered to be successfully connected differently on a per-protocol basis. If the only protocol being used is a transport protocol with a clear handshake, like TCP, then the obvious choice is to declare that node "connected" when the last packet of the three-way handshake has been received. If the only protocol being used is an "unconnected" protocol, like UDP, the implementation may consider the node fully "connected" the moment it determines a route is present, before sending any packets on the network, see further Section 4.5.

For protocol stacks with multiple handshakes, the decision becomes more nuanced. If the protocol stack involves both TLS and TCP, an implementation could determine that a leaf node is connected after the TCP handshake is complete, or it can wait for the TLS handshake to complete as well. The benefit of declaring completion when the TCP handshake finishes, and thus stopping the race for other branches of the tree, is that there will be less burden on the network from other connection attempts. On the other hand, by waiting until the TLS handshake is complete, an implementation avoids the scenario in which a TCP handshake completes quickly, but TLS negotiation is either very slow or fails altogether in particular network conditions or to a particular endpoint. To avoid the issue of TLS possibly failing, the implementation should not generate a Ready event for the Connection until TLS is established.

If all of the leaf nodes fail to connect during racing, i.e. none of the configurations that satisfy all requirements given in the Transport Properties actually work over the available paths, then the transport system should notify the application with an InitiateError event. An InitiateError event should also be generated in case the transport system finds no usable candidates to race.

4.4. Establishing multiplexed connections

Multiplexing several Connections over a single underlying transport connection requires that the Connections to be multiplexed belong to the same Connection Group (as is indicated by the application using the Clone call). When the underlying transport connection supports multi-streaming, the Transport System can map each Connection in the Connection Group to a different stream. Thus, when the Connections that are offered to an application by the Transport System are multiplexed, the Transport System may implement the establishment of a new Connection by simply beginning to use a new stream of an already established transport connection and there is no need for a connection establishment procedure. This, then, also means that there may not be any "establishment" message (like a TCP SYN), but the application can simply start sending or receiving. Therefore, when the Initiate action of a Transport System is called without Messages being handed over, it cannot be guaranteed that the other endpoint will have any way to know about this, and hence a passive endpoint's ConnectionReceived event may not be called upon an active endpoint's Inititate. Instead, calling the ConnectionReceived event may be delayed until the first Message arrives.

4.5. Handling racing with "unconnected" protocols

While protocols that use an explicit handshake to validate a Connection to a peer can be used for racing multiple establishment attempts in parallel, "unconnected" protocols such as raw UDP do not offer a way to validate the presence of a peer or the usability of a Connection without application feedback. An implementation should consider such a protocol stack to be established as soon as a local route to the peer endpoint is confirmed.

However, if a peer is not reachable over the network using the unconnected protocol, or data cannot be exchanged for any other reason, the application may want to attempt using another candidate Protocol Stack. The implementation should maintain the list of other candidate Protocol Stacks that were eligible to use.

4.6. Implementing listeners

When an implementation is asked to Listen, it registers with the system to wait for incoming traffic to the Local Endpoint. If no Local Endpoint is specified, the implementation should use an ephemeral port.

If the Selection Properties do not require a single network interface or path, but allow the use of multiple paths, the Listener object should register for incoming traffic on all of the network interfaces or paths that conform to the Properties. The set of available paths can change over time, so the implementation should monitor network path changes and register and de-register the Listener across all usable paths. When using multiple paths, the Listener is generally expected to use the same port for listening on each.

If the Selection Properties allow multiple protocols to be used for listening, and the implementation supports it, the Listener object should support receiving inbound connections for each eligible protocol on each eligible path.

4.6.1. Implementing listeners for Connected Protocols

Connected protocols such as TCP and TLS-over-TCP have a strong mapping between the Local and Remote Endpoints (five-tuple) and their protocol connection state. These map into Connection objects. Whenever a new inbound handshake is being started, the Listener should generate a new Connection object and pass it to the application.

4.6.2. Implementing listeners for Unconnected Protocols

Unconnected protocols such as UDP and UDP-lite generally do not provide the same mechanisms that connected protocols do to offer Connection objects. Implementations should wait for incoming packets for unconnected protocols on a listening port and should perform five-tuple matching of packets to either existing Connection objects or the creation of new Connection objects. On platforms with facilities to create a "virtual connection" for unconnected protocols implementations should use these mechanisms to minimise the handling of datagrams intended for already created Connection objects.

4.6.3. Implementing listeners for Multiplexed Protocols

Protocols that provide multiplexing of streams into a single five-tuple can listen both for entirely new connections (a new HTTP/2 stream on a new TCP connection, for example) and for new sub-connections (a new HTTP/2 stream on an existing connection). If the abstraction of Connection presented to the application is mapped to the multiplexed stream, then the Listener should deliver new Connection objects in the same way for either case. The implementation should allow the application to introspect the Connection Group marked on the Connections to determine the grouping of the multiplexing.

5. Implementing Sending and Receiving Data

The most basic mapping for sending a Message is an abstraction of datagrams, in which the transport protocol naturally deals in discrete packets. Each Message here corresponds to a single datagram. Generally, these will be short enough that sending and receiving will always use a complete Message.

For protocols that expose byte-streams, the only delineation provided by the protocol is the end of the stream in a given direction. Each Message in this case corresponds to the entire stream of bytes in a direction. These Messages may be quite long, in which case they can be sent in multiple parts.

Protocols that provide the framing (such as length-value protocols, or protocols that use delimiters) provide data boundaries that may be longer than a traditional packet datagram. Each Message for framing protocols corresponds to a single frame, which may be sent either as a complete Message, or in multiple parts.

5.1. Sending Messages

The effect of the application sending a Message is determined by the top-level protocol in the established Protocol Stack. That is, if the top-level protocol provides an abstraction of framed messages over a connection, the receiving application will be able to obtain multiple Messages on that connection, even if the framing protocol is built on a byte-stream protocol like TCP.

5.1.1. Message Properties

* Lifetime: this should be implemented by removing the Message from the queue of pending Messages after the Lifetime has expired. A queue of pending Messages within the transport system implementation that have yet to be handed to the Protocol Stack can always support this property, but once a Message has been sent into the send buffer of a protocol, only certain protocols may support removing a message. For example, an implementation cannot remove bytes from a TCP send buffer, while it can remove data from a SCTP send buffer using the partial reliability extension [RFC8303]. When there is no standing queue of Messages within the system, and the Protocol Stack does not support the removal of a Message from the stack's send buffer, this property may be ignored.

* Priority: this represents the ability to prioritize a Message over other Messages. This can be implemented by the system re-ordering Messages that have yet to be handed to the Protocol Stack, or by giving relative priority hints to protocols that support priorities per Message. For example, an implementation of HTTP/2 could choose to send Messages of different Priority on streams of different priority.

* Ordered: when this is false, this disables the requirement of in-order-delivery for protocols that support configurable ordering.

* Safely Replayable: when this is true, this means that the Message can be used by mechanisms that might transfer it multiple times - e.g., as a result of racing multiple transports or as part of TCP Fast Open. Also, protocols that do not protect against duplicated messages, such as UDP, can only be used with Messages that are Safely Replayable.

* Final: when this is true, this means that a transport connection can be closed immediately after transmission of the message.

* Corruption Protection Length: when this is set to any value other than "Full Coverage", it sets the minimum protection in protocols that allow limiting the checksum length (e.g. UDP-Lite).

* Reliable Data Transfer (Message): When true, the property specifies that the Message must be reliably transmitted. When false, and if unreliable transmission is supported by the underlying protocol, then the Message should be unreliably transmitted. If the underlying protocol does not support unreliable transmission, the Message should be reliably transmitted.

* Message Capacity Profile Override: When true, this expresses a wish to override the Generic Connection Property "Capacity Profile" for this Message. Depending on the value, this can, for example, be implemented by changing the DSCP value of the associated packet (note that the he guidelines in Section 6 of [RFC7657] apply; e.g., the DSCP value should not be changed for different packets within a reliable transport protocol session or DCCP connection).

* No Fragmentation: When set, this property limits the message size to the Maximum Message Size Before Fragmentation or Segmentation (see Section 10.1.7 of [I-D.ietf-taps-interface]). Messages larger than this size generate an error. Setting this avoids transport-layer segmentation or network-layer fragmentation. When used with transports running over IP version 4 the Don't Fragment bit will be set to avoid on-path IP fragmentation ([RFC8304]).

5.1.2. Send Completion

The application should be notified whenever a Message or partial Message has been consumed by the Protocol Stack, or has failed to send. The meaning of the Message being consumed by the stack may vary depending on the protocol. For a basic datagram protocol like UDP, this may correspond to the time when the packet is sent into the interface driver. For a protocol that buffers data in queues, like TCP, this may correspond to when the data has entered the send buffer.

5.1.3. Batching Sends

Since sending a Message may involve a context switch between the application and the transport system, sending patterns that involve multiple small Messages can incur high overhead if each needs to be enqueued separately. To avoid this, the application can indicate a batch of Send actions through the API. When this is used, the implementation should hold off on processing Messages until the batch is complete.

5.2. Receiving Messages

Similar to sending, Receiving a Message is determined by the top-level protocol in the established Protocol Stack. The main difference with Receiving is that the size and boundaries of the Message are not known beforehand. The application can communicate in its Receive action the parameters for the Message, which can help the implementation know how much data to deliver and when. For example, if the application only wants to receive a complete Message, the implementation should wait until an entire Message (datagram, stream, or frame) is read before delivering any Message content to the application. This requires the implementation to understand where messages end, either via a supplied deframer or because the top-level protocol in the established Protocol Stack preserves message boundaries. If the top-level protocol only supports a byte-stream and no framers were supported, the application can control the flow of received data by specifying the minimum number of bytes of Message content it wants to receive at one time.

If a Connection becomes finished before a requested Receive action can be satisfied, the implementation should deliver any partial Message content outstanding, or if none is available, an indication that there will be no more received Messages.

5.3. Handling of data for fast-open protocols

Several protocols allow sending higher-level protocol or application data within the first packet of their protocol establishment, such as TCP Fast Open [RFC7413] and TLS 1.3 [RFC8446]. This approach is referred to as sending Zero-RTT (0-RTT) data. This is a desirable property, but poses challenges to an implementation that uses racing during connection establishment.

If the application has 0-RTT data to send in any protocol handshakes, it needs to provide this data before the handshakes have begun. When racing, this means that the data should be provided before the process of connection establishment has begun. If the application wants to send 0-RTT data, it must indicate this to the implementation by setting the "Safely Replayable" send parameter to true when sending the data. In general, 0-RTT data may be replayed (for example, if a TCP SYN contains data, and the SYN is retransmitted, the data will be retransmitted as well but may be considered as a new connection instead of a retransmission). Also, when racing connections, different leaf nodes have the opportunity to send the same data independently. If data is truly safely replayable, this should be permissible.

Once the application has provided its 0-RTT data, an implementation should keep a copy of this data and provide it to each new leaf node that is started and for which a 0-RTT protocol is being used.

It is also possible that protocol stacks within a particular leaf node use 0-RTT handshakes without any safely replayable application data. For example, TCP Fast Open could use a Client Hello from TLS as its 0-RTT data, shortening the cumulative handshake time.

0-RTT handshakes often rely on previous state, such as TCP Fast Open cookies, previously established TLS tickets, or out-of-band distributed pre-shared keys (PSKs). Implementations should be aware of security concerns around using these tokens across multiple addresses or paths when racing. In the case of TLS, any given ticket or PSK should only be used on one leaf node, since servers will likely reject duplicate tickets in order to prevent replays (see section-8.1 [RFC8446]). If implementations have multiple tickets available from a previous connection, each leaf node attempt can use a different ticket. In effect, each leaf node will send the same early application data, yet encoded (encrypted) differently on the wire.

6. Implementing Message Framers

Message Framers are pieces of code that define simple transformations between application Message data and raw transport protocol data. A Framer can encapsulate or encode outbound Messages, and decapsulate or decode inbound data into Messages.

While many protocols can be represented as Message Framers, for the purposes of the Transport Services interface these are ways for applications or application frameworks to define their own Message parsing to be included within a Connection's Protocol Stack. As an example, TLS can serve the purpose of framing data over TCP, but is exposed as a protocol natively supported by the Transport Services interface.

Most Message Framers fall into one of two categories:

* Header-prefixed record formats, such as a basic Type-Length-Value (TLV) structure

* Delimiter-separated formats, such as HTTP/1.1.

Common Message Framers can be provided by the Transport Services implementation, but an implementation ought to allow custom Message Framers to be defined by the application or some other piece of software. This section describes one possible interface for defining Message Framers as an example.

6.1. Defining Message Framers

A Message Framer is primarily defined by the set of code that handles events for a framer implementation, specifically how it handles inbound and outbound data parsing. The piece of code that implements custom framing logic will be referred to as the "framer implementation", which may be provided by the Transport Services implementation or the application itself. The Message Framer refers to the object or piece of code within the main Connection implementation that delivers events to the custom framer implementation whenever data is ready to be parsed or framed.

When a Connection establishment attempt begins, an event can be delivered to notify the framer implementation that a new Connection is being created. Similarly, a stop event can be delivered when a Connection is being torn down. The framer implementation can use the Connection object to look up specific properties of the Connection or the network being used that may influence how to frame Messages.

```
MessageFramer -> Start(Connection)
MessageFramer -> Stop(Connection)
```

When a Message Framer generates a "Start" event, the framer implementation has the opportunity to start writing some data prior to the Connection delivering its "Ready" event. This allows the implementation to communicate control data to the remote endpoint that can be used to parse Messages.

```
MessageFramer.MakeConnectionReady(Connection)
```

Similarly, when a Message Framer generates a "Stop" event, the framer implementation has the opportunity to write some final data or clear up its local state before the "Closed" event is delivered to the Application. The framer implementation can indicate that it has finished with this.

```
MessageFramer.MakeConnectionClosed(Connection)
```

At any time if the implementation encounters a fatal error, it can also cause the Connection to fail and provide an error.

```
MessageFramer.FailConnection(Connection, Error)
```

Should the framer implementation deem the candidate selected during racing unsuitable it can signal this by failing the Connection prior to marking it as ready. If there are no other candidates available, the Connection will fail. Otherwise, the Connection will select a different candidate and the Message Framer will generate a new "Start" event.

Before an implementation marks a Message Framer as ready, it can also dynamically add a protocol or framer above it in the stack. This allows protocols like STARTTLS, that need to add TLS conditionally, to modify the Protocol Stack based on a handshake result.

```
otherFramer := NewMessageFramer()
MessageFramer.PrependFramer(Connection, otherFramer)
```

6.2. Sender-side Message Framing

Message Framers generate an event whenever a Connection sends a new Message.

```
MessageFramer -> NewSentMessage<Connection, MessageData, MessageContext, IsEndOfM
essage>
```

Upon receiving this event, a framer implementation is responsible for performing any necessary transformations and sending the resulting data back to the Message Framer, which will in turn send it to the next protocol. Implementations SHOULD ensure that there is a way to pass the original data through without copying to improve performance.

```
MessageFramer.Send(Connection, Data)
```

To provide an example, a simple protocol that adds a length as a header would receive the "NewSentMessage" event, create a data representation of the length of the Message data, and then send a block of data that is the concatenation of the length header and the original Message data.

6.3. Receiver-side Message Framing

In order to parse a received flow of data into Messages, the Message Framer notifies the framer implementation whenever new data is available to parse.

```
MessageFramer -> HandleReceivedData&lt;Connection>
```

Upon receiving this event, the framer implementation can inspect the inbound data. The data is parsed from a particular cursor representing the unprocessed data. The application requests a specific amount of data it needs to have available in order to parse. If the data is not available, the parse fails.

```
MessageFramer.Parse(Connection, MinimumIncompleteLength, MaximumLength) -> (Data,
 MessageContext, IsEndOfMessage)
```

The framer implementation can directly advance the receive cursor once it has parsed data to effectively discard data (for example, discard a header once the content has been parsed).

To deliver a Message to the application, the framer implementation can either directly deliver data that it has allocated, or deliver a range of data directly from the underlying transport and simultaneously advance the receive cursor.

```
MessageFramer.AdvanceReceiveCursor(Connection, Length)
MessageFramer.DeliverAndAdvanceReceiveCursor(Connection, MessageContext, Length,
IsEndOfMessage)
MessageFramer.Deliver(Connection, MessageContext, Data, IsEndOfMessage)
```

Note that "MessageFramer.DeliverAndAdvanceReceiveCursor" allows the framer implementation to earmark bytes as part of a Message even before they are received by the transport. This allows the delivery of very large Messages without requiring the implementation to directly inspect all of the bytes.

To provide an example, a simple protocol that parses a length as a header value would receive the "HandleReceivedData" event, and call "Parse" with a minimum and maximum set to the length of the header field. Once the parse succeeded, it would call "AdvanceReceiveCursor" with the length of the header field, and then call "DeliverAndAdvanceReceiveCursor" with the length of the body that was parsed from the header, marking the new Message as complete.

7. Implementing Connection Management

Once a Connection is established, the Transport Services system allows applications to interact with the Connection by modifying or inspecting Connection Properties. A Connection can also generate events in the form of Soft Errors.

The set of Connection Properties that are supported for setting and getting on a Connection are described in [I-D.ietf-taps-interface]. For any properties that are generic, and thus could apply to all protocols being used by a Connection, the Transport System should store the properties in a generic storage, and notify all protocol instances in the Protocol Stack whenever the properties have been modified by the application. For protocol-specfic properties, such as the User Timeout that applies to TCP, the Transport System only needs to update the relevant protocol instance.

If an error is encountered in setting a property (for example, if the application tries to set a TCP-specific property on a Connection that is not using TCP), the action should fail gracefully. The application may be informed of the error, but the Connection itself should not be terminated.

The Transport Services implementation should allow protocol instances in the Protocol Stack to pass up arbitrary generic or protocol-specific errors that can be delivered to the application as Soft Errors. These allow the application to be informed of ICMP errors, and other similar events.

7.1. Pooled Connection

For protocols that employ request/response pairs and do not require in-order delivery of the responses, like HTTP, the transport implementation may distribute interactions across several underlying transport connections. For these kinds of protocols, implementations may hide the connection management and only expose a single Connection object and the individual requests/responses as messages. These Pooled Connections can use multiple connections or multiple streams of multi-streaming connections between endpoints, as long as all of these satisfy the requirements, and prohibitions specified in the Selection Properties of the Pooled Connection. This enables implementations to realize transparent connection coalescing, connection migration, and to perform per-message endpoint and path selection by choosing among these underlying connections.

7.2. Handling Path Changes

When a path change occurs, the Transport Services implementation is responsible for notifying Protocol Instances in the Protocol Stack. If the Protocol Stack includes a transport protocol that supports multipath connectivity, an update to the available paths should inform the Protocol Instance of the new set of paths that are permissible based on the Selection Properties passed by the application. A multipath protocol can establish new subflows over new paths, and should tear down subflows over paths that are no longer available. Pooled Connections Section 7.1 may add or remove underlying transport connections in a similar manner. If the Protocol Stack includes a transport protocol that does not support multipath, but support migrating between paths, the update to available paths can be used as the trigger to migrating the connection. For protocols that do not support multipath or migration, the Protocol Instances may be informed of the path change, but should not be forcibly disconnected if the previously used path becomes unavailable. An exception to this case is if the System Policy changes to prohibit traffic from the Connection based on its properties, in which case the Protocol Stack should be disconnected.

8. Implementing Connection Termination

With TCP, when an application closes a connection, this means that it has no more data to send (but expects all data that has been handed over to be reliably delivered). However, with TCP only, "close" does not mean that the application will stop receiving data. This is related to TCP's ability to support half-closed connections.

SCTP is an example of a protocol that does not support such half-closed connections. Hence, with SCTP, the meaning of "close" is stricter: an application has no more data to send (but expects all data that has been handed over to be reliably delivered), and will also not receive any more data.

Implementing a protocol independent transport system means that the exposed semantics must be the strictest subset of the semantics of all supported protocols. Hence, as is common with all reliable transport protocols, after a Close action, the application can expect to have its reliability requirements honored regarding the data it has given to the Transport System, but it cannot expect to be able to read any more data after calling Close.

Abort differs from Close only in that no guarantees are given regarding data that the application has handed over to the Transport System before calling Abort.

As explained in Section 4.4, when a new stream is multiplexed on an already existing connection of a Transport Protocol Instance, there is no need for a connection establishment procedure. Because the Connections that are offered by the Transport System can be implemented as streams that are multiplexed on a transport protocol's connection, it can therefore not be guaranteed that one Endpoint's Initiate action provokes a ConnectionReceived event at its peer.

For Close (provoking a Finished event) and Abort (provoking a ConnectionError event), the same logic applies: while it is desirable to be informed when a peer closes or aborts a Connection, whether this is possible depends on the underlying protocol, and no guarantees can be given. With SCTP, the transport system can use the stream reset procedure to cause a Finish event upon a Close action from the peer [NEAT-flow-mapping].

9. Cached State

Beyond a single Connection’s lifetime, it is useful for an implementation to keep state and history. This cached state can help improve future Connection establishment due to re-using results and credentials, and favoring paths and protocols that performed well in the past.

Cached state may be associated with different Endpoints for the same Connection, depending on the protocol generating the cached content. For example, session tickets for TLS are associated with specific endpoints, and thus should be cached based on a Connection’s hostname Endpoint (if applicable). On the other hand, performance characteristics of a path are more likely tied to the IP address and subnet being used.

9.1. Protocol state caches

Some protocols will have long-term state to be cached in association with Endpoints. This state often has some time after which it is expired, so the implementation should allow each protocol to specify an expiration for cached content.

Examples of cached protocol state include:

* The DNS protocol can cache resolution answers (A and AAAA queries, for example), associated with a Time To Live (TTL) to be used for future hostname resolutions without requiring asking the DNS resolver again.

* TLS caches session state and tickets based on a hostname, which can be used for resuming sessions with a server.

* TCP can cache cookies for use in TCP Fast Open.

Cached protocol state is primarily used during Connection establishment for a single Protocol Stack, but may be used to influence an implementation's preference between several candidate Protocol Stacks. For example, if two IP address Endpoints are otherwise equally preferred, an implementation may choose to attempt a connection to an address for which it has a TCP Fast Open cookie.

Applications must have a way to flush protocol cache state if desired. This may be necessary, for example, if application-layer identifiers rotate and clients wish to avoid linkability via trackable TLS tickets or TFO cookies.

9.2. Performance caches

In addition to protocol state, Protocol Instances should provide data into a performance-oriented cache to help guide future protocol and path selection. Some performance information can be gathered generically across several protocols to allow predictive comparisons between protocols on given paths:

* Observed Round Trip Time

* Connection Establishment latency

* Connection Establishment success rate

These items can be cached on a per-address and per-subnet granularity, and averaged between different values. The information should be cached on a per-network basis, since it is expected that different network attachments will have different performance characteristics. Besides Protocol Instances, other system entities may also provide data into performance-oriented caches. This could for instance be signal strength information reported by radio modems like Wi-Fi and mobile broadband or information about the battery-level of the device. Furthermore, the system may cache the observed maximum throughput on a path as an estimate of the available bandwidth.

An implementation should use this information, when possible, to determine preference between candidate paths, endpoints, and protocol options. Eligible options that historically had significantly better performance than others should be selected first when gathering candidates (see Section 4.1) to ensure better performance for the application.

The reasonable lifetime for cached performance values will vary depending on the nature of the value. Certain information, like the connection establishment success rate to a Remote Endpoint using a given protocol stack, can be stored for a long period of time (hours or longer), since it is expected that the capabilities of the Remote Endpoint are not changing very quickly. On the other hand, the Round Trip Time observed by TCP over a particular network path may vary over a relatively short time interval. For such values, the implementation should remove them from the cache more quickly, or treat older values with less confidence/weight.

[I-D.ietf-tcpm-2140bis] provides guidance about sharing of TCP Control Block information between connections on initialization.

10. Specific Transport Protocol Considerations

Each protocol that can run as part of a Transport Services implementation defines both its API mapping as well as implementation details. API mappings for a protocol apply most to Connections in which the given protocol is the "top" of the Protocol Stack. For example, the mapping of the "Send" function for TCP applies to Connections in which the application directly sends over TCP. If HTTP/2 is used on top of TCP, the HTTP/2 mappings take precendence.

Each protocol has a notion of Connectedness. Possible values for Connectedness are:

* Unconnected. Unconnected protocols do not establish explicit state between endpoints, and do not perform a handshake during Connection establishment.

* Connected. Connected protocols establish state between endpoints, and perform a handshake during Connection establishment. The handshake may be 0-RTT to send data or resume a session, but bidirectional traffic is required to confirm connectedness.

* Multiplexing Connected. Multiplexing Connected protocols share properties with Connected protocols, but also explictly support opening multiple application-level flows. This means that they can support cloning new Connection objects without a new explicit handshake.

Protocols also define a notion of Data Unit. Possible values for Data Unit are:

* Byte-stream. Byte-stream protocols do not define any Message boundaries of their own apart from the end of a stream in each direction.

* Datagram. Datagram protocols define Message boundaries at the same level of transmission, such that only complete (not partial) Messages are supported.

* Message. Message protocols support Message boundaries that can be sent and received either as complete or partial Messages. Maximum Message lengths can be defined, and Messages can be partially reliable.

Below, terms in capitals with a dot (e.g., "CONNECT.SCTP") refer to the primitives with the same name in section 4 of [RFC8303]. For further implementation details, the description of these primitives in [RFC8303] points to section 3 of [RFC8303] and section 3 of [RFC8304], which refers back to the relevant specifications for each protocol. This back-tracking method applies to all elements of [I-D.ietf-taps-minset] (see appendix D of [I-D.ietf-taps-interface]): they are listed in appendix A of [I-D.ietf-taps-minset] with an implementation hint in the same style, pointing back to section 4 of [RFC8303].

10.1. TCP

Connectedness: Connected

Data Unit: Byte-stream

API mappings for TCP are as follows:

Connection Object: TCP connections between two hosts map directly to Connection objects.

Initiate: CONNECT.TCP. Calling "Initiate" on a TCP Connection causes it to reserve a local port, and send a SYN to the Remote Endpoint.

InitiateWithSend: CONNECT.TCP with parameter "user message". Early safely replayable data is sent on a TCP Connection in the SYN, as TCP Fast Open data.

Ready: A TCP Connection is ready once the three-way handshake is complete.

InitiateError: Failure of CONNECT.TCP. TCP can throw various errors during connection setup. Specifically, it is important to handle a RST being sent by the peer during the handshake.

ConnectionError: Once established, TCP throws errors whenever the connection is disconnected, such as due to receiving a RST from the peer; or hitting a TCP retransmission timeout.

Listen: LISTEN.TCP. Calling "Listen" for TCP binds a local port and prepares it to receive inbound SYN packets from peers.

ConnectionReceived: TCP Listeners will deliver new connections once they have replied to an inbound SYN with a SYN-ACK.

Clone: Calling "Clone" on a TCP Connection creates a new Connection with equivalent parameters. The two Connections are otherwise independent.

Send: SEND.TCP. TCP does not on its own preserve Message boundaries. Calling "Send" on a TCP connection lays out the bytes on the TCP send stream without any other delineation. Any Message marked as Final will cause TCP to send a FIN once the Message has been completely written, by calling CLOSE.TCP immediately upon successful termination of SEND.TCP.

Receive: With RECEIVE.TCP, TCP delivers a stream of bytes without any Message delineation. All data delivered in the "Received" or "ReceivedPartial" event will be part of a single stream-wide Message that is marked Final (unless a Message Framer is used). EndOfMessage will be delivered when the TCP Connection has received a FIN (CLOSE-EVENT.TCP or ABORT-EVENT.TCP) from the peer.

Close: Calling "Close" on a TCP Connection indicates that the Connection should be gracefully closed (CLOSE.TCP) by sending a FIN to the peer and waiting for a FIN-ACK before delivering the "Closed" event.

Abort: Calling "Abort" on a TCP Connection indicates that the Connection should be immediately closed by sending a RST to the peer (ABORT.TCP).

10.2. UDP

Connectedness: Unconnected

Data Unit: Datagram

API mappings for UDP are as follows:

Connection Object: UDP connections represent a pair of specific IP addresses and ports on two hosts.

Initiate: CONNECT.UDP. Calling "Initiate" on a UDP Connection causes it to reserve a local port, but does not generate any traffic.

InitiateWithSend: Early data on a UDP Connection does not have any special meaning. The data is sent whenever the Connection is Ready.

Ready: A UDP Connection is ready once the system has reserved a local port and has a path to send to the Remote Endpoint.

InitiateError: UDP Connections can only generate errors on initiation due to port conflicts on the local system.

ConnectionError: Once in use, UDP throws "soft errors" (ERROR.UDP(-Lite)) upon receiving ICMP notifications indicating failures in the network.

Listen: LISTEN.UDP. Calling "Listen" for UDP binds a local port and prepares it to receive inbound UDP datagrams from peers.

ConnectionReceived: UDP Listeners will deliver new connections once they have received traffic from a new Remote Endpoint.

Clone: Calling "Clone" on a UDP Connection creates a new Connection with equivalent parameters. The two Connections are otherwise independent.

Send: SEND.UDP(-Lite). Calling "Send" on a UDP connection sends the data as the payload of a complete UDP datagram. Marking Messages as Final does not change anything in the datagram's contents. Upon sending a UDP datagram, some relevant fields and flags in the IP header can be controlled: DSCP (SET_DSCP.UDP(-Lite)), DF in IPv4 (SET_DF.UDP(-Lite)) and ECN flag (SET_ECN.UDP(-Lite)).

Receive: RECEIVE.UDP(-Lite). UDP only delivers complete Messages to "Received", each of which represents a single datagram received in a UDP packet. Upon receiving a UDP datagram, the ECN flag from the IP header can be obtained (GET_ECN.UDP(-Lite)).

Close: Calling "Close" on a UDP Connection (ABORT.UDP(-Lite)) releases the local port reservation.

Abort: Calling "Abort" on a UDP Connection (ABORT.UDP(-Lite)) is identical to calling "Close".

10.3. UDP Multicast Receive

Connectedness: Unconnected

Data Unit: Datagram

API mappings for Receiving Multicast UDP are as follows:

Connection Object: Established UDP Multicast Receive connections represent a pair of specific IP addresses and ports. The "unidirectional receive" transport property is required, and the local endpoint must be configured with a group IP address and a port.

Initiate: Calling "Initiate" on a UDP Multicast Receive Connection causes an immediate InitiateError. This is an unsupported operation.

InitiateWithSend: Calling "InitiateWithSend" on a UDP Multicast Receive Connection causes an immediate InitiateError. This is an unsupported operation.

Ready: A UDP Multicast Receive Connection is ready once the system has received traffic for the appropriate group and port.

InitiateError: UDP Multicast Receive Connections generate an InitiateError if Initiate is called.

ConnectionError: Once in use, UDP throws "soft errors" (ERROR.UDP(-Lite)) upon receiving ICMP notifications indicating failures in the network.

Listen: LISTEN.UDP. Calling "Listen" for UDP Multicast Receive binds a local port, prepares it to receive inbound UDP datagrams from peers, and issues a multicast host join. If a remote endpoint with an address is supplied, the join is Source-specific Multicast, and the path selection is based on the route to the remote endpoint. If a remote endpoint is not supplied, the join is Any-source Multicast, and the path selection is based on the outbound route to the group supplied in the local endpoint.

ConnectionReceived: UDP Multicast Receive Listeners will deliver new connections once they have received traffic from a new Remote Endpoint.

Clone: Calling "Clone" on a UDP Multicast Receive Connection creates a new Connection with equivalent parameters. The two Connections are otherwise independent.

Send: SEND.UDP(-Lite). Calling "Send" on a UDP Multicast Receive connection causes an immediate SendError. This is an unsupported operation.

Receive: RECEIVE.UDP(-Lite). The Receive operation in a UDP Multicast Receive connection only delivers complete Messages to "Received", each of which represents a single datagram received in a UDP packet. Upon receiving a UDP datagram, the ECN flag from the IP header can be obtained (GET_ECN.UDP(-Lite)).

Close: Calling "Close" on a UDP Multicast Receive Connection (ABORT.UDP(-Lite)) releases the local port reservation and leaves the group.

Abort: Calling "Abort" on a UDP Multicast Receive Connection (ABORT.UDP(-Lite)) is identical to calling "Close".

10.4. TLS

The mapping of a TLS stream abstraction into the application is equivalent to the contract provided by TCP (see Section 10.1), and builds upon many of the actions of TCP connections.

Connectedness: Connected

Data Unit: Byte-stream

Connection Object: Connection objects represent a single TLS connection running over a TCP connection between two hosts.

Initiate: Calling "Initiate" on a TLS Connection causes it to first initiate a TCP connection. Once the TCP protocol is Ready, the TLS handshake will be performed as a client (starting by sending a "client_hello", and so on).

InitiateWithSend: Early safely replayable data is supported by TLS 1.3, and sends encrypted application data in the first TLS message when performing session resumption. For older versions of TLS, or if a session is not being resumed, the initial data will be delayed until the TLS handshake is complete. TCP Fast Open can also be enabled automatically.

Ready: A TLS Connection is ready once the underlying TCP connection is Ready, and TLS handshake is also complete and keys have been established to encrypt application data.

InitiateError: In addition to TCP initiation errors, TLS can generate errors during its handshake. Examples of error include a failure of the peer to successfully authenticate, the peer rejecting the local authentication, or a failure to match versions or algorithms.

ConnectionError: TLS connections will generate TCP errors, or errors due to failures to rekey or decrypt received messages.

Listen: Calling "Listen" for TLS listens on TCP, and sets up received connections to perform server-side TLS handshakes.

ConnectionReceived: TLS Listeners will deliver new connections once they have successfully completed both TCP and TLS handshakes.

Clone: As with TCP, calling "Clone" on a TLS Connection creates a new Connection with equivalent parameters. The two Connections are otherwise independent.

Send: Like TCP, TLS does not preserve message boundaries. Although application data is framed natively in TLS, there is not a general guarantee that these TLS messages represent semantically meaningful application stream boundaries. Rather, sending data on a TLS Connection only guarantees that the application data will be transmitted in an encrypted form. Marking Messages as Final causes a "close_notify" to be generated once the data has been written.

Receive: Like TCP, TLS delivers a stream of bytes without any Message delineation. The data is decrypted prior to being delivered to the application. If a "close_notify" is received, the stream-wide Message will be delivered with EndOfMessage set.

Close: Calling "Close" on a TLS Connection indicates that the Connection should be gracefully closed by sending a "close_notify" to the peer and waiting for a corresponding "close_notify" before delivering the "Closed" event.

Abort: Calling "Abort" on a TCP Connection indicates that the Connection should be immediately closed by sending a "close_notify", optionally preceded by "user_canceled", to the peer. Implementations do not need to wait to receive "close_notify" before delivering the "Closed" event.

10.5. DTLS

DTLS follows the same behavior as TLS (Section 10.4), with the notable exception of not inheriting behavior directly from TCP. Differences from TLS are detailed below, and all cases not explicitly mentioned should be considered the same as TLS.

Connectedness: Connected

Data Unit: Datagram

Connection Object: Connection objects represent a single DTLS connection running over a set of UDP ports between two hosts.

Initiate: Calling "Initiate" on a DTLS Connection causes it reserve a UDP local port, and begin sending handshake messages to the peer over UDP. These messages are reliable, and will be automatically retransmitted.

Ready: A DTLS Connection is ready once the TLS handshake is complete and keys have been established to encrypt application data.

Send: Sending over DTLS does preserve message boundaries in the same way that UDP datagrams do. Marking a Message as Final does send a "close_notify" like TLS.

Receive: Receiving over DTLS delivers one decrypted Message for each received DTLS datagram. If a "close_notify" is received, a Message will be delivered that is marked as Final.

10.6. HTTP

HTTP requests and responses map naturally into Messages, since they are delineated chunks of data with metadata that can be sent over a transport. To that end, HTTP can be seen as the most prevalent framing protocol that runs on top of streams like TCP, TLS, etc.

In order to use a transport Connection that provides HTTP Message support, the establishment and closing of the connection can be treated as it would without the framing protocol. Sending and receiving of Messages, however, changes to treat each Message as a well-delineated HTTP request or response, with the content of the Message representing the body, and the Headers being provided in Message metadata.

Connectedness: Multiplexing Connected

Data Unit: Message

Connection Object: Connection objects represent a flow of HTTP messages between a client and a server, which may be an HTTP/1.1 connection over TCP, or a single stream in an HTTP/2 connection.

Initiate: Calling "Initiate" on an HTTP connection intiates a TCP or TLS connection as a client.

Clone: Calling "Clone" on an HTTP Connection opens a new stream on an existing HTTP/2 connection when possible. If the underlying version does not support multiplexed streams, calling "Clone" simply creates a new parallel connection.

Send: When an application sends an HTTP Message, it is expected to provide HTTP header values as a MessageContext in a canonical form, along with any associated HTTP message body as the Message data. The HTTP header values are encoded in the specific version format upon sending.

Receive: HTTP Connections deliver Messages in which HTTP header values attached to MessageContexts, and HTTP bodies in Message data.

Close: Calling "Close" on an HTTP Connection will only close the underlying TLS or TCP connection if the HTTP version does not support multiplexing. For HTTP/2, for example, closing the connection only closes a specific stream.

10.7. QUIC

QUIC provides a multi-streaming interface to an encrypted transport. Each stream can be viewed as equivalent to a TLS stream over TCP, so a natural mapping is to present each QUIC stream as an individual Connection. The protocol for the stream will be considered Ready whenever the underlying QUIC connection is established to the point that this stream's data can be sent. For streams after the first stream, this will likely be an immediate operation.

Closing a single QUIC stream, presented to the application as a Connection, does not imply closing the underlying QUIC connection itself. Rather, the implementation may choose to close the QUIC connection once all streams have been closed (often after some timeout), or after an individual stream Connection sends an Abort.

Connectedness: Multiplexing Connected

Data Unit: Stream

Connection Object: Connection objects represent a single QUIC stream on a QUIC connection.

10.8. HTTP/2 transport

Similar to QUIC (Section 10.7), HTTP/2 provides a multi-streaming interface. This will generally use HTTP as the unit of Messages over the streams, in which each stream can be represented as a transport Connection. The lifetime of streams and the HTTP/2 connection should be managed as described for QUIC.

It is possible to treat each HTTP/2 stream as a raw byte-stream instead of a carrier for HTTP messages, in which case the Messages over the streams can be represented similarly to the TCP stream (one Message per direction, see Section 10.1).

Connectedness: Multiplexing Connected

Data Unit: Stream

Connection Object: Connection objects represent a single HTTP/2 stream on a HTTP/2 connection.

10.9. SCTP

Connectedness: Connected

Data Unit: Message

API mappings for SCTP are as follows:

Connection Object: Connection objects represent a flow of SCTP messages between a client and a server, which may be an SCTP association or a stream in a SCTP association. How to map Connection objects to streams is described in [NEAT-flow-mapping]; in the following, a similar method is described. To map Connection objects to SCTP streams without head-of-line blocking on the sender side, both the sending and receiving SCTP implementation must support message interleaving [RFC8260]. Both SCTP implementations must also support stream reconfiguration. Finally, both communicating endpoints must be aware of this intended multiplexing; [NEAT-flow-mapping] describes a way for a Transport System to negotiate the stream mapping capability using SCTP's adaptation layer indication, such that this functionality would only take effect if both ends sides are aware of it. The first flow, for which the SCTP association has been created, will always use stream id zero. All additional flows are assigned to unused stream ids in growing order. To avoid a conflict when both endpoints map new flows simultaneously, the peer which initiated the transport connection will use even stream numbers whereas the remote side will map its flows to odd stream numbers. Both sides maintain a status map of the assigned stream numbers. Generally, new streams must consume the lowest available (even or odd, depending on the side) stream number; this rule is relevant when lower numbers become available because Connection objects associated to the streams are closed.

Initiate: If this is the only Connection object that is assigned to the SCTP association or stream mapping has not been negotiated, CONNECT.SCTP is called. Else, a new stream is used: if there are enough streams available, "Initiate" is just a local operation that assigns a new stream number to the Connection object. The number of streams is negotiated as a parameter of the prior CONNECT.SCTP call, and it represents a trade-off between local resource usage and the number of Connection objects that can be mapped without requiring a reconfiguration signal. When running out of streams, ADD_STREAM.SCTP must be called.

InitiateWithSend: If this is the only Connection object that is assigned to the SCTP association or stream mapping has not been negotiated, CONNECT.SCTP is called with the "user message" parameter. Else, a new stream is used (see "Initiate" for how to handle running out of streams), and this just sends the first message on a new stream.

Ready: "Initiate" or "InitiateWithSend" returns without an error, i.e. SCTP's four-way handshake has completed. If an association with the peer already exists, and stream mapping has been negotiated and enough streams are available, a Connection Object instantly becomes Ready after calling "Initiate" or "InitiateWithSend".

InitiateError: Failure of CONNECT.SCTP.

ConnectionError: TIMEOUT.SCTP or ABORT-EVENT.SCTP.

Listen: LISTEN.SCTP. If an association with the peer already exists and stream mapping has been negotiated, "Listen" just expects to receive a new message on a new stream id (chosen in accordance with the stream number assignment procedure described above).

ConnectionReceived: LISTEN.SCTP returns without an error (a result of successful CONNECT.SCTP from the peer), or, in case of stream mapping, the first message has arrived on a new stream (in this case, "Receive" is also invoked).

Clone: Calling "Clone" on an SCTP association creates a new

Connection object and assigns it a new stream number in accordance with the stream number assignment procedure described above. If there are not enough streams available, ADD_STREAM.SCTP must be called.

Priority (Connection): When this value is changed, or a Message with Message Property "Priority" is sent, and there are multiple Connection objects assigned to the same SCTP association, CONFIGURE_STREAM_SCHEDULER.SCTP is called to adjust the priorities of streams in the SCTP association.

Send: SEND.SCTP. Message Properties such as "Lifetime" and "Ordered" map to parameters of this primitive.

Receive: RECEIVE.SCTP. The "partial flag" of RECEIVE.SCTP invokes a "ReceivedPartial" event.

Close: If this is the only Connection object that is assigned to the SCTP association, CLOSE.SCTP is called. Else, the Connection object is one out of several Connection objects that are assigned to the same SCTP assocation, and RESET_STREAM.SCTP must be called, which informs the peer that the stream will no longer be used for mapping and can be used by future "Initiate", "InitiateWithSend" or "Listen" calls. At the peer, the event RESET_STREAM-EVENT.SCTP will fire, which the peer must answer by issuing RESET_STREAM.SCTP too. The resulting local RESET_STREAM-EVENT.SCTP informs the transport system that the stream number can now be re-used by the next "Initiate", "InitiateWithSend" or "Listen" calls.

Abort: If this is the only Connection object that is assigned to the SCTP association, ABORT.SCTP is called. Else, the Connection object is one out of several Connection objects that are assigned to the same SCTP assocation, and shutdown proceeds as described under "Close".

11. IANA Considerations

RFC-EDITOR: Please remove this section before publication.

This document has no actions for IANA.

12. Security Considerations

[I-D.ietf-taps-arch] outlines general security consideration and requirements for any system that implements the TAPS archtecture. [I-D.ietf-taps-interface] provides further discussion on security and privacy implications of the TAPS API. This document provides additional guidance on implementation specifics for the TAPS API and as such the security considerations in both of these documents apply. The next two subsections discuss further considerations that are specific to mechanisms specified in this document.

12.1. Considerations for Candidate Gathering

Implementations should avoid downgrade attacks that allow network interference to cause the implementation to select less secure, or entirely insecure, combinations of paths and protocols.

12.2. Considerations for Candidate Racing

See Section 5.3 for security considerations around racing with 0-RTT data.

An attacker that knows a particular device is racing several options during connection establishment may be able to block packets for the first connection attempt, thus inducing the device to fall back to a secondary attempt. This is a problem if the secondary attempts have worse security properties that enable further attacks. Implementations should ensure that all options have equivalent security properties to avoid incentivizing attacks.

Since results from the network can determine how a connection attempt tree is built, such as when DNS returns a list of resolved endpoints, it is possible for the network to cause an implementation to consume significant on-device resources. Implementations should limit the maximum amount of state allowed for any given node, including the number of child nodes, especially when the state is based on results from the network.

13. Acknowledgements

This work has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No. 644334 (NEAT).

This work has been supported by Leibniz Prize project funds of DFG - German Research Foundation: Gottfried Wilhelm Leibniz-Preis 2011 (FKZ FE 570/4-1).

This work has been supported by the UK Engineering and Physical Sciences Research Council under grant EP/R04144X/1.

This work has been supported by the Research Council of Norway under its "Toppforsk" programme through the "OCARINA" project.

Thanks to Stuart Cheshire, Josh Graessley, David Schinazi, and Eric Kinnear for their implementation and design efforts, including Happy Eyeballs, that heavily influenced this work.

14. References

14.1. Normative References

[I-D.ietf-taps-arch]
Pauly, T., Trammell, B., Brunstrom, A., Fairhurst, G., Perkins, C., Tiesel, P., and C. Wood, "An Architecture for Transport Services", Work in Progress, Internet-Draft, draft-ietf-taps-arch-07, 9 March 2020, <http://www.ietf.org/internet-drafts/draft-ietf-taps-arch-07.txt>.

[I-D.ietf-taps-interface]
Trammell, B., Welzl, M., Enghardt, T., Fairhurst, G., Kuehlewind, M., Perkins, C., Tiesel, P., Wood, C., and T. Pauly, "An Abstract Application Layer Interface to Transport Services", Work in Progress, Internet-Draft, draft-ietf-taps-interface-06, 9 March 2020, <http://www.ietf.org/internet-drafts/draft-ietf-taps-interface-06.txt>.

[I-D.ietf-taps-minset]
Welzl, M. and S. Gjessing, "A Minimal Set of Transport Services for End Systems", Work in Progress, Internet-Draft, draft-ietf-taps-minset-11, 27 September 2018, <http://www.ietf.org/internet-drafts/draft-ietf-taps-minset-11.txt>.

[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>.

[RFC7540] Belshe, M., Peon, R., and M. Thomson, Ed., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, DOI 10.17487/RFC7540, May 2015, <https://www.rfc-editor.org/info/rfc7540>.

[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>.

[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>.

[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>.

[RFC8305] Schinazi, D. and T. Pauly, "Happy Eyeballs Version 2: Better Connectivity Using Concurrency", RFC 8305, DOI 10.17487/RFC8305, December 2017, <https://www.rfc-editor.org/info/rfc8305>.

[RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <https://www.rfc-editor.org/info/rfc8446>.

14.2. Informative References

[I-D.ietf-quic-transport]
Iyengar, J. and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport", Work in Progress, Internet-Draft, draft-ietf-quic-transport-29, 9 June 2020, <http://www.ietf.org/internet-drafts/draft-ietf-quic-transport-29.txt>.

[I-D.ietf-tcpm-2140bis]
Touch, J., Welzl, M., and S. Islam, "TCP Control Block Interdependence", Work in Progress, Internet-Draft, draft-ietf-tcpm-2140bis-05, 29 April 2020, <http://www.ietf.org/internet-drafts/draft-ietf-tcpm-2140bis-05.txt>.

[NEAT-flow-mapping]
"Transparent Flow Mapping for NEAT", Workshop on Future of Internet Transport (FIT 2017) , 2017.

[RFC5389] Rosenberg, J., Mahy, R., Matthews, P., and D. Wing, "Session Traversal Utilities for NAT (STUN)", RFC 5389, DOI 10.17487/RFC5389, October 2008, <https://www.rfc-editor.org/info/rfc5389>.

[RFC5766] Mahy, R., Matthews, P., and J. Rosenberg, "Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN)", RFC 5766, DOI 10.17487/RFC5766, April 2010, <https://www.rfc-editor.org/info/rfc5766>.

[RFC6762] Cheshire, S. and M. Krochmal, "Multicast DNS", RFC 6762, DOI 10.17487/RFC6762, February 2013, <https://www.rfc-editor.org/info/rfc6762>.

[RFC6763] Cheshire, S. and M. Krochmal, "DNS-Based Service Discovery", RFC 6763, DOI 10.17487/RFC6763, February 2013, <https://www.rfc-editor.org/info/rfc6763>.

[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>.

[RFC8445] Keranen, A., Holmberg, C., and J. Rosenberg, "Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal", RFC 8445, DOI 10.17487/RFC8445, July 2018, <https://www.rfc-editor.org/info/rfc8445>.

Appendix A. Additional Properties

This appendix discusses implementation considerations for additional parameters and properties that could be used to enhance transport protocol and/or path selection, or the transmission of messages given a Protocol Stack that implements them. These are not part of the interface, and may be removed from the final document, but are presented here to support discussion within the TAPS working group as to whether they should be added to a future revision of the base specification.

A.1. Properties Affecting Sorting of Branches

In addition to the Protocol and Path Selection Properties discussed in Section 4.1.5, the following properties under discussion can influence branch sorting:

* Bounds on Send or Receive Rate: If the application indicates a bound on the expected Send or Receive bitrate, an implementation may prefer a path that can likely provide the desired bandwidth, based on cached maximum throughput, see Section 9.2. The application may know the Send or Receive Bitrate from metadata in adaptive HTTP streaming, such as MPEG-DASH.

* Cost Preferences: If the application indicates a preference to avoid expensive paths, and some paths are associated with a monetary cost, an implementation should decrease the ranking of such paths. If the application indicates that it prohibits using expensive paths, paths that are associated with a cost should be purged from the decision tree.

Appendix B. Reasons for errors

The Transport Services API [I-D.ietf-taps-interface] allows for the several generic error types to specify a more detailed reason as to why an error occurred. This appendix lists some of the possible reasons.

* InvalidConfiguration: The transport properties and endpoints provided by the application are either contradictory or incomplete. Examples include the lack of a remote endpoint on an active open or using a multicast group address while not requesting a unidirectional receive.

* NoCandidates: The configuration is valid, but none of the available transport protocols can satisfy the transport properties provided by the application.

* ResolutionFailed: The remote or local specifier provided by the application can not be resolved.

* EstablishmentFailed: The TAPS system was unable to establish a transport-layer connection to the remote endpoint specified by the application.

* PolicyProhibited: The system policy prevents the transport system from performing the action requested by the application.

* NotCloneable: The protocol stack is not capable of being cloned.

* MessageTooLarge: The message size is too big for the transport system to handle.

* ProtocolFailed: The underlying protocol stack failed.

* InvalidMessageProperties: The message properties are either contradictory to the transport properties or they can not be satisfied by the transport system.

* DeframingFailed: The data that was received by the underlying protocol stack could not be deframed.

* ConnectionAborted: The connection was aborted by the peer.

* Timeout: Delivery of a message was not possible after a timeout.

Appendix C. Existing Implementations

This appendix gives an overview of existing implementations, at the time of writing, of transport systems that are (to some degree) in line with this document.

* Apple's Network.framework:

  - Network.framework is a transport-level API built for C, Objective-C, and Swift. It a connect-by-name API that supports transport security protocols. It provides userspace implementations of TCP, UDP, TLS, DTLS, proxy protocols, and allows extension via custom framers.

  - Documentation: https://developer.apple.com/documentation/network (https://developer.apple.com/documentation/network)

* NEAT and NEATPy:

  - NEAT is the output of the European H2020 research project "NEAT"; it is a user-space library for protocol-independent communication on top of TCP, UDP and SCTP, with many more features such as a policy manager.

  - Code: https://github.com/NEAT-project/neat (https://github.com/NEAT-project/neat)

  - NEAT project: https://www.neat-project.org (https://www.neat-project.org)

  - NEATPy is a Python shim over NEAT which updates the NEAT API to be in line with version 6 of the TAPS interface draft.

  - Code: https://github.com/theagilepadawan/NEATPy (https://github.com/theagilepadawan/NEATPy)

* PyTAPS:

  - A TAPS implementation based on Python asyncio, offering protocol-independent communication to applications on top of TCP, UDP and TLS, with support for multicast.

  - Code: https://github.com/fg-inet/python-asyncio-taps (https://github.com/fg-inet/python-asyncio-taps)

Authors' Addresses

Anna Brunstrom (editor)
Karlstad University
Universitetsgatan 2
651 88 Karlstad
Sweden

Email: anna.brunstrom@kau.se

Tommy Pauly (editor)
Apple Inc.
One Apple Park Way
Cupertino, California 95014,
United States of America

Email: tpauly@apple.com

Theresa Enghardt
Netflix
121 Albright Way
Los Gatos, CA 95032,
United States of America

Email: ietf@tenghardt.net

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

Email: karl-johan.grinnemo@kau.se

Tom Jones
University of Aberdeen
Fraser Noble Building
Aberdeen, AB24 3UE
United Kingdom

Email: tom@erg.abdn.ac.uk

Philipp S. Tiesel
TU Berlin
Einsteinufer 25
10587 Berlin
Germany

Email: philipp@tiesel.net

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

Email: csp@csperkins.org

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

Email: michawe@ifi.uio.no

TAPS Working Group
Internet-Draft
Intended status: Standards Track
Expires: January 28, 2021

B. Trammell, Ed.
Google Switzerland GmbH
M. Welzl, Ed.
University of Oslo
T. Enghardt
Netflix
G. Fairhurst
University of Aberdeen
M. Kuehlewind
Ericsson
C. Perkins
University of Glasgow
P. Tiesel
TU Berlin
C. Wood
Cloudflare
T. Pauly
Apple Inc.
July 27, 2020

An Abstract Application Layer Interface to Transport Services
draft-ietf-taps-interface-09

Abstract

This document describes an abstract application programming interface, API, to the transport layer, following the Transport Services Architecture. It supports the asynchronous, atomic transmission of messages over transport protocols and network paths dynamically selected at runtime. It is intended to replace the traditional BSD sockets API as the common interface to the transport layer, in an environment where endpoints could select from multiple interfaces and potential transport protocols.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

This document specifies a modern abstract application programming interface (API) atop the high-level architecture for transport services defined in [I-D.ietf-taps-arch]. It supports the asynchronous, atomic transmission of messages over transport protocols and network paths dynamically selected at runtime. It is intended to replace the traditional BSD sockets API as the common interface to the transport layer, in environments where endpoints select from multiple interfaces and potential transport protocols.

As applications adopt this interface, they will benefit from a wide set of transport features that can evolve over time, and ensure that the system providing the interface can optimize its behavior based on the application requirements and network conditions, without requiring changes to the applications. This flexibility enables faster deployment of new features and protocols. It can also support applications by offering racing and fallback mechanisms, which otherwise need to be separately implemented in each application.

It derives specific path and protocol selection properties and supported transport features from the analysis provided in [RFC8095], [I-D.ietf-taps-minset], and [I-D.ietf-taps-transport-security]. The design encourages implementations underneath the interface to dynamically choose a transport protocol depending on an application's choices rather than statically binding applications to a protocol at compile time. The transport system implementations should provide applications with a way to override transport selection and instantiate a specific stack, e.g., to support servers wishing to listen to a specific protocol. This specific transport stack choice is discouraged for general use, because it can reduce the portability.

2. Terminology and Notation

This API is described in terms of Objects with which an application can interact; Actions the application can perform on these Objects; Events, which an Object can send to an application asynchronously; and Parameters associated with these Actions and Events.

The following notations, which can be combined, are used in this document:

- An Action creates an Object:

```
Object := Action()
```

- An Action creates an array of Objects:

```
[]Object := Action()
```

- An Action is performed on an Object:

```
Object.Action()
```

- An Object sends an Event:

```
Object -> Event<>
```

- An Action takes a set of Parameters; an Event contains a set of Parameters. Action and Event parameters whose names are suffixed with a question mark are optional.

```
Action(param0, param1?, ...) / Event<param0, param1, ...>
```

Actions associated with no Object are Actions on the abstract interface itself; they are equivalent to Actions on a per-application global context.

The way these abstract concepts map into concrete implementations of this API in a given language on a given platform largely depends on the features of the language and the platform. Actions could be implemented as functions or method calls, for instance, and Events could be implemented via event queues, handler functions or classes, communicating sequential processes, or other asynchronous calling conventions.

This specification treats Events and errors similarly. Errors, just as any other Events, may occur asynchronously in network applications. However, it is recommended that implementations of this interface also return errors immediately, according to the error handling idioms of the implementation platform, for errors that can be immediately detected, such as inconsistency in Transport Properties. Errors can provide an optional reason to give the application further details as to why the error occurred.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

3. Overview of Interface Design

The design of the interface specified in this document is based on a set of principles, themselves an elaboration on the architectural design principles defined in [I-D.ietf-taps-arch]. The interface defined in this document provides:

- A single interface to a variety of transport protocols to be used in a variety of application design patterns, independent of the properties of the application and the Protocol Stacks that will be used at runtime, such that all common specialized features of these protocol stacks are made available to the application as necessary in a transport-independent way, to enable applications written to a single API to make use of transport protocols in terms of the features they provide;

- Message-orientation, as opposed to stream-orientation, using application-assisted framing and deframing where the underlying transport does not provide these;

- Asynchronous Connection establishment, transmission, and reception, allowing concurrent operations during establishment and supporting event-driven application interactions with the transport layer, in line with developments in modern platforms and programming languages;

- Explicit support for security properties as first-order transport features, and for configuration of cryptographic identities and transport security parameters persistent across multiple Connections; and

- Explicit support for multistreaming and multipath transport protocols, and the grouping of related Connections into Connection Groups through cloning of Connections, to allow applications to take full advantage of new transport protocols supporting these features.

4. API Summary

The Transport Services API is the basic common abstract application programming interface to the Transport Services Architecture defined in the TAPS Architecture [I-D.ietf-taps-arch].

An application primarily interacts with this API through two Objects: Preconnections and Connections. A Preconnection represents a set of properties and constraints on the selection and configuration of paths and protocols to establish a Connection with a remote Endpoint. A Connection represents a transport Protocol Stack on which data can be sent to and/or received from a remote Endpoint (i.e., depending on the kind of transport, connections can be bi-directional or unidirectional). Connections can be created from Preconnections in three ways: by initiating the Preconnection (i.e., actively opening, as in a client), through listening on the Preconnection (i.e., passively opening, as in a server), or rendezvousing on the Preconnection (i.e. peer to peer establishment).

Once a Connection is established, data can be sent and received on it in the form of Messages. The interface supports the preservation of message boundaries both via explicit Protocol Stack support, and via application support through a Message Framer which finds message boundaries in a stream. Messages are received asynchronously through event handlers registered by the application. Errors and other notifications also happen asynchronously on the Connection. It is not necessary for an application to handle all events; some events may have implementation-specific default handlers. The application should not assume that ignoring events (e.g. errors) is always safe.

Section 5, Section 6, Section 8.2, Section 8.3, and Section 9 describe the details of application interaction with Objects through Actions and Events in each phase of a Connection, following the phases (Pre-Establishment, Establishment, Data Transfer, and Termination) described in Section 4.1 of [I-D.ietf-taps-arch].

4.1. Usage Examples

The following usage examples illustrate how an application might use a Transport Services Interface to:

- Act as a server, by listening for incoming connections, receiving requests, and sending responses, see Section 4.1.1.

- Act as a client, by connecting to a remote endpoint using Initiate, sending requests, and receiving responses, see Section 4.1.2.

- Act as a peer, by connecting to a remote endpoint using Rendezvous while simultaneously waiting for incoming Connections, sending Messages, and receiving Messages, see Section 4.1.3.

The examples in this section presume that a transport protocol is available between the endpoints that provides Reliable Data Transfer, Preservation of data ordering, and Preservation of Message Boundaries. In this case, the application can choose to receive only complete messages.

If none of the available transport protocols provides Preservation of Message Boundaries, but there is a transport protocol that provides a reliable ordered byte stream, an application may receive this byte stream as partial Messages and transform it into application-layer Messages. Alternatively, an application may provide a Message Framer, which can transform a byte stream into a sequence of Messages (Section 8.1.2).

4.1.1. Server Example

This is an example of how an application might listen for incoming Connections using the Transport Services Interface, receive a request, and send a response.

```
LocalSpecifier := NewLocalEndpoint()
LocalSpecifier.WithInterface("any")
LocalSpecifier.WithService("https")

TransportProperties := NewTransportProperties()
TransportProperties.Require(preserve-msg-boundaries)
// Reliable Data Transfer and Preserve Order are Required by default

SecurityParameters := NewSecurityParameters()
SecurityParameters.AddIdentity(identity)
SecurityParameters.AddPrivateKey(privateKey, publicKey)

// Specifying a remote endpoint is optional when using Listen()
Preconnection := NewPreconnection(LocalSpecifier,
                                  TransportProperties,
                                  SecurityParameters)

Listener := Preconnection.Listen()

Listener -> ConnectionReceived<Connection>

// Only receive complete messages in a Conn.Received handler
Connection.Receive()

Connection -> Received<messageDataRequest, messageContext>

//---- Receive event handler begin ----
Connection.Send(messageDataResponse)
Connection.Close()

// Stop listening for incoming Connections
// (this example supports only one Connection)
Listener.Stop()
//---- Receive event handler end ----
```

4.1.2. Client Example

This is an example of how an application might connect to a remote application using the Transport Services Interface, send a request, and receive a response.

```
RemoteSpecifier := NewRemoteEndpoint()
RemoteSpecifier.WithHostname("example.com")
RemoteSpecifier.WithService("https")

TransportProperties := NewTransportProperties()
TransportProperties.Require(preserve-msg-boundaries)
// Reliable Data Transfer and Preserve Order are Required by default

SecurityParameters := NewSecurityParameters()
TrustCallback := NewCallback({
  // Verify identity of the remote endpoint, return the result
})
SecurityParameters.SetTrustVerificationCallback(TrustCallback)

// Specifying a local endpoint is optional when using Initiate()
Preconnection := NewPreconnection(RemoteSpecifier,
                                  TransportProperties,
                                  SecurityParameters)

Connection := Preconnection.Initiate()

Connection -> Ready<>

//---- Ready event handler begin ----
Connection.Send(messageDataRequest)

// Only receive complete messages
Connection.Receive()
//---- Ready event handler end ----

Connection -> Received<messageDataResponse, messageContext>

// Close the Connection in a Receive event handler
Connection.Close()
```

4.1.3. Peer Example

This is an example of how an application might establish a connection with a peer using Rendezvous(), send a Message, and receive a Message.

```
LocalSpecifier := NewLocalEndpoint()
LocalSpecifier.WithPort(9876)

RemoteSpecifier := NewRemoteEndpoint()
RemoteSpecifier.WithHostname("example.com")
RemoteSpecifier.WithPort(9877)

TransportProperties := NewTransportProperties()
TransportProperties.Require(preserve-msg-boundaries)
// Reliable Data Transfer and Preserve Order are Required by default

SecurityParameters := NewSecurityParameters()
SecurityParameters.AddIdentity(identity)
SecurityParameters.AddPrivateKey(privateKey, publicKey)

TrustCallback := New Callback({
  // Verify identity of the remote endpoint, return the result
})
SecurityParameters.SetTrustVerificationCallback(trustCallback)

// Both local and remote endpoint must be specified
Preconnection := NewPreconnection(LocalSpecifier,
                                  RemoteSpecifier,
                                  TransportProperties,
                                  SecurityParameters)

Preconnection.Rendezvous()

Preconnection -> RendezvousDone<Connection>

//---- Ready event handler begin ----
Connection.Send(messageDataRequest)

// Only receive complete messages
Connection.Receive()
//---- Ready event handler end ----

Connection -> Received<messageDataResponse, messageContext>

// Close the Connection in a Receive event handler
Connection.Close()
```

4.2. Transport Properties

Each application using the Transport Services Interface declares its preferences for how the transport service should operate using properties at each stage of the lifetime of a connection using Transport Properties, as defined in [I-D.ietf-taps-arch].

Transport Properties are divided into Selection, Connection, and Message Properties. Selection Properties (see Section 5.2) can only be set during pre-establishment. They are only used to specify which paths and protocol stacks can be used and are preferred by the application. Connection Properties (see Section 7.1) can also be set during pre-establishment but may be changed later. They are used to inform decisions made during establishment and to fine-tune the established connection.
The behavior of the selected protocol stack(s) when sending Messages is controlled by Message Properties (see Section 8.1.3).

All Transport Properties, regardless of the phase in which they are used, are organized within a single namespace. This enables setting them as defaults in earlier stages and querying them in later stages:

o Connection Properties can be set on Preconnections

o Message Properties can be set on Preconnections, Connections and Messages

o The effect of Selection Properties can be queried on Connections and Messages

Note that configuring Connection Properties and Message Properties on Preconnections is preferred over setting them later. Early specification of Connection Properties allows their use as additional input to the selection process. Protocol Specific Properties, which enable configuration of specialized features of a specific protocol, see Section 3.2 of [I-D.ietf-taps-arch], are not used as an input to the selection process but only support configuration if the respective protocol has been selected.

4.2.1. Transport Property Names

Transport Properties are referred to by property names. For the purposes of this document, these names are alphanumeric strings in which words may be separated by hyphens. These names serve two purposes:

o Allowing different components of a TAPS implementation to pass Transport Properties, e.g., between a language frontend and a policy manager, or as a representation of properties retrieved from a file or other storage.

o Making code of different TAPS implementations look similar. While individual programming languages may preclude strict adherence to the aforementioned naming convention (for instance, by prohibiting the use of hyphens in symbols), users interacting with multiple implementations will still benefit from the consistency resulting from the use of visually similar symbols.

Transport Property Names are hierarchically organized in the form [<Namespace>.]<PropertyName>.

- The Namespace component MUST be empty for well-known, generic properties, i.e., for properties that are not specific to a protocol and are defined in an RFC.

- Protocol Specific Properties MUST use the protocol acronym as Namespace, e.g., "tcp" for TCP specific Transport Properties. For IETF protocols, property names under these namespaces SHOULD be defined in an RFC.

- Vendor or implementation specific properties MUST use a string identifying the vendor or implementation as Namespace.

Namespaces for each of the keywords provided in the IANA protocol numbers registry (see https://www.iana.org/assignments/protocol-numbers/protocol-numbers.xhtml), reformatted where necessary to conform to an implementation's naming conventions, are reserved for Protocol Specific Properties and MUST not be used for vendor or implementation-specific properties.

4.2.2. Transport Property Types

Transport Properties can have one of a set of data types:

- Boolean: can take the values "true" and "false"; representation is implementation-dependent.

- Integer: can take positive or negative numeric integer values; range and representation is implementation-dependent.

- Numeric: can take positive or negative numeric values; range and representation is implementation-dependent.

- Enumeration: can take one value of a finite set of values, dependent on the property itself. The representation is implementation dependent; however, implementations MUST provide a method for the application to determine the entire set of possible values for each property.

- Preference: can take one of five values (Prohibit, Avoid, Ignore, Prefer, Require) for the level of preference of a given property during protocol selection; see Section 5.2. When querying, a

Preference is of type Boolean, with "true" indicating that the Selection Property has been applied.

For types Integer and Numeric, special values can be defined per property; it is up to implementations how these special values are represented (e.g., by using -1 for an otherwise non-negative value).

4.3. Scope of the Interface Definition

This document defines a language- and platform-independent interface to a Transport Services system. Given the wide variety of languages and language conventions used to write applications that use the transport layer to connect to other applications over the Internet, this independence makes this interface necessarily abstract.

There is no interoperability benefit in tightly defining how the interface is presented to application programmers across diverse platforms. However, maintaining the "shape" of the abstract interface across these platforms reduces the effort for programmers who learn the transport services interface to then apply their knowledge across multiple platforms.

We therefore make the following recommendations:

- Actions, Events, and Errors in implementations of this interface SHOULD use the names given for them in the document, subject to capitalization, punctuation, and other typographic conventions in the language of the implementation, unless the implementation itself uses different names for substantially equivalent objects for networking by convention.

- Implementations of this interface SHOULD implement each Selection Property, Connection Property, and Message Context Property specified in this document. Each interface SHOULD be implemented even when this will always result in no operation, e.g. there is no action when the API specifies a Property that is not available in a transport protocol implemented on a specific platform. For example, if TCP is the only underlying transport protocol, the Message Property "msgOrdered" can be implemented even if disabling ordering will not have any effect TCP because the API does not guarantee out-of-order delivery. Similarly, the "msg-lifetime" Message Property can be implemented but ignored, as the description of this Property states that "it is not guaranteed that a Message will not be sent when its Lifetime has expired".

- Implementations may use other representations for Transport Property Names, e.g., by providing constants, but should provide a straight-forward mapping between their representation and the property names specified here.

5. Pre-Establishment Phase

The Pre-Establishment phase allows applications to specify properties for the Connections they are about to make, or to query the API about potential Connections they could make.

A Preconnection Object represents a potential Connection. It has state that describes properties of a Connection that might exist in the future. This state comprises Local Endpoint and Remote Endpoint Objects that denote the endpoints of the potential Connection (see Section 5.1), the Selection Properties (see Section 5.2), any preconfigured Connection Properties (Section 7.1), and the security parameters (see Section 5.3):

```
Preconnection := NewPreconnection(LocalEndpoint?,
                                   RemoteEndpoint?,
                                   TransportProperties,
                                   SecurityParams)
```

The Local Endpoint MUST be specified if the Preconnection is used to Listen() for incoming Connections, but is OPTIONAL if it is used to Initiate() connections. If no Local Endpoint is specified, the Transport System will assign an ephemeral local port to the Connection. The Remote Endpoint MUST be specified if the Preconnection is used to Initiate() Connections, but is OPTIONAL if it is used to Listen() for incoming Connections. The Local Endpoint and the Remote Endpoint MUST both be specified if a peer-to-peer Rendezvous is to occur based on the Preconnection.

Transport Properties MUST always be specified while security parameters are OPTIONAL.

If Message Framers are used (see Section 8.1.2), they MUST be added to the Preconnection during pre-establishment.

5.1. Specifying Endpoints

The transport services API uses the Local Endpoint and Remote Endpoint Objects to refer to the endpoints of a transport connection. Actions on these Objects can be used to represent various different types of endpoint identifiers, such as IP addresses, DNS names, and interface names, as well as port numbers and service names.

Specify a Remote Endpoint using a hostname and service name:

```
RemoteSpecifier := NewRemoteEndpoint()
RemoteSpecifier.WithHostname("example.com")
RemoteSpecifier.WithService("https")
```

Specify a Remote Endpoint using an IPv6 address and remote port:

```
RemoteSpecifier := NewRemoteEndpoint()
RemoteSpecifier.WithIPv6Address(2001:db8:4920:e29d:a420:7461:7073:0a)
RemoteSpecifier.WithPort(443)
```

Specify a Remote Endpoint using an IPv4 address and remote port:

```
RemoteSpecifier := NewRemoteEndpoint()
RemoteSpecifier.WithIPv4Address(192.0.2.21)
RemoteSpecifier.WithPort(443)
```

Specify a Local Endpoint using a local interface name and local port:

```
LocalSpecifier := NewLocalEndpoint()
LocalSpecifier.WithInterface("en0")
LocalSpecifier.WithPort(443)
```

As an alternative to specifying an interface name for the Local Endpoint, an application can express more fine-grained preferences using the "Interface Instance or Type" Selection Property, see Section 5.2.10. However, if the application specifies Selection Properties which are inconsistent with the Local Endpoint, this will result in an error once the application attempts to open a Connection.

Specify a Local Endpoint using a STUN server:

```
LocalSpecifier := NewLocalEndpoint()
LocalSpecifier.WithStunServer(address, port, credentials)
```

Specify a Local Endpoint using a Any-Source Multicast group to join on a named local interface:

```
LocalSpecifier := NewLocalEndpoint()
LocalSpecifier.WithIPv4Address(233.252.0.0)
LocalSpecifier.WithInterface("en0")
```

Source-Specific Multicast requires setting both a Local and Remote Endpoint:

```
LocalSpecifier := NewLocalEndpoint()
LocalSpecifier.WithIPv4Address(232.1.1.1)
LocalSpecifier.WithInterface("en0")

RemoteSpecifier := NewRemoteEndpoint()
RemoteSpecifier.WithIPv4Address(192.0.2.22)
```

Implementations may also support additional endpoint representations and provide a single NewEndpoint() call that takes different endpoint representations.

Multiple endpoint identifiers can be specified for each Local Endpoint and Remote Endpoint. For example, a Local Endpoint could be configured with two interface names, or a Remote Endpoint could be specified via both IPv4 and IPv6 addresses. These multiple identifiers refer to the same transport endpoint.

The transport services API resolves names internally, when the Initiate(), Listen(), or Rendezvous() method is called to establish a Connection. The API explicitly does not require the application to resolve names, though there is a tradeoff between early and late binding of addresses to names. Early binding allows the API implementation to reduce connection setup latency, at the cost of potentially limited scope for alternate path discovery during Connection establishment, as well as potential additional information leakage about application interest when used with a resolution method (such as DNS without TLS) which does not protect query confidentiality.

The Resolve() action on Preconnection can be used by the application to force early binding when required, for example with some Network Address Translator (NAT) traversal protocols (see Section 6.3).

Specifying a multicast group address on the Local Endpoint will indicate to the transport system that the resulting connection will be used to receive multicast messages. The Remote Endpoint can be used to filter by specific senders. This will restrict the application to establishing the Preconnection by calling Listen(). The accepted Connections are receive-only.

Similarly, specifying a multicast group address on the Remote Endpoint will indicate that the resulting connection will be used to send multicast messages.

5.2. Specifying Transport Properties

A Preconnection Object holds properties reflecting the application's requirements and preferences for the transport. These include Selection Properties for selecting protocol stacks and paths, as well as Connection Properties for configuration of the detailed operation of the selected Protocol Stacks.

The protocol(s) and path(s) selected as candidates during establishment are determined and configured using these properties. Since there could be paths over which some transport protocols are unable to operate, or remote endpoints that support only specific network addresses or transports, transport protocol selection is necessarily tied to path selection. This may involve choosing between multiple local interfaces that are connected to different access networks.

Most Selection Properties are represented as preferences, which can have one of five preference levels:

Preference	Effect
Require	Select only protocols/paths providing the property, fail otherwise
Prefer	Prefer protocols/paths providing the property, proceed otherwise
Ignore	No preference
Avoid	Prefer protocols/paths not providing the property, proceed otherwise
Prohibit	Select only protocols/paths not providing the property, fail otherwise

In addition, the pseudo-level "Default" can be used to reset the property to the default level used by the implementation. This level will never show up when queuing the value of a preference - the effective preference must be returned instead.

The implementation MUST ensure an outcome that is consistent with application requirements as expressed using Require and Prohibit. While preferences expressed using Prefer and Avoid influence protocol and path selection as well, outcomes may vary given the same Selection Properties, as the available protocols and paths may vary across systems and contexts. However, implementations are RECOMMENDED to aim to provide a consistent outcome to an application, given the same Selection Properties.

Note that application preferences may conflict with each other. For example, if an application indicates a preference for a specific path by specifying an interface, but also a preference for a protocol, a situation might occur in which the preferred protocol is not available on the preferred path. In such cases, implementations SHOULD prioritize Selection Properties that select paths over those that select protocols. Therefore, the transport system SHOULD race the path first, ignoring the protocol preference if the protocol does not work on the path.

Selection and Connection Properties, as well as defaults for Message Properties, can be added to a Preconnection to configure the selection process and to further configure the eventually selected protocol stack(s). They are collected into a TransportProperties object to be passed into a Preconnection object:

```
TransportProperties := NewTransportProperties()
```

Individual properties are then added to the TransportProperties Object:

```
TransportProperties.Add(property, value)
```

Selection Properties of type "Preference" can be frequently used. Implementations MAY therefore provide additional convenience functions, see Appendix A.1 for examples. In addition, implementations MAY provide a mechanism to create TransportProperties objects that are preconfigured for common use cases as outlined in Appendix A.2.

For an existing Connection, the Transport Properties can be queried any time by using the following call on the Connection Object:

```
TransportProperties := Connection.GetTransportProperties()
```

A Connection gets its Transport Properties either by being explicitly configured via a Preconnection, by configuration after establishment, or by inheriting them from an antecedent via cloning; see Section 6.4 for more.

Section 7.1 provides a list of Connection Properties, while Selection Properties are listed in the subsections below. Note that many properties are only considered during establishment, and can not be changed after a Connection is established; however, they can be queried. The return type of a queried Selection Property is Boolean, where "true" means that the Selection Property has been applied and "false" means that the Selection Property has not been applied. Note that "true" does not mean that a request has been honored. For example, if "Congestion control" was requested with preference level "Prefer", but congestion control could not be supported, querying the "congestionControl" property yields the value "false". If preference level "Avoid" was used for "Congestion control", and, as requested, the Connection is not congestion controlled, querying the "congestionControl" property also yields the value "false".

An implementation of this interface must provide sensible defaults for Selection Properties. The recommended default values for each property below represent a configuration that can be implemented over TCP. If these default values are used and TCP is not supported by a Transport Services implementation, then an application using the default set of Properties might not succeed in establishing a connection. Using the same default values for independent Transport Services implementations can be beneficical when application are ported between different implementations, even if this default could lead to a connection failure, as, for example, an application needs to be explicitly designed to support a connectionless mode. In this case the application can regonize the failure and explicitly specify a different set of Protocol Selection Properties that result in a usable protocol. If default values other than those recommended below are used, it is recommended to clearly document the differences.

5.2.1. Reliable Data Transfer (Connection)

Name: reliability

Type: Preference

Default: Require

This property specifies whether the application needs to use a transport protocol that ensures that all data is received on the other side without corruption. This also entails being notified when a Connection is closed or aborted when reliable data transfer is enabled.

5.2.2. Preservation of Message Boundaries

Name: preserveMsgBoundaries

Type: Preference

Default: Prefer

This property specifies whether the application needs or prefers to use a transport protocol that preserves message boundaries.

5.2.3. Configure Per-Message Reliability

Name: perMsgReliability

Type: Preference

Default: Ignore

This property specifies whether an application considers it useful to indicate its reliability requirements on a per-Message basis. This property applies to Connections and Connection Groups.

5.2.4. Preservation of Data Ordering

Name: preserveOrder

Type: Preference

Default: Require

This property specifies whether the application wishes to use a transport protocol that can ensure that data is received by the application on the other end in the same order as it was sent.

5.2.5. Use 0-RTT Session Establishment with a Safely Replayable Message

Name: zeroRttMsg

Type: Preference

Default: Ignore

This property specifies whether an application would like to supply a Message to the transport protocol before Connection establishment, which will then be reliably transferred to the other side before or during Connection establishment, potentially multiple times (i.e., multiple copies of the message data may be passed to the Remote Endpoint). See also Section 8.1.3.4. Note that disabling this property has no effect for protocols that are not connection-oriented and do not protect against duplicated messages, e.g., UDP.

5.2.6. Multistream Connections in Group

Name: multistreaming

Type: Preference

Default: Prefer

This property specifies that the application would prefer multiple Connections within a Connection Group to be provided by streams of a single underlying transport connection where possible.

5.2.7. Full Checksum Coverage on Sending

Name: perMsgChecksumLenSend

Type: Preference

Default: Require

This property specifies whether the application desires protection against corruption for all data transmitted on this Connection. Disabling this property may enable to control checksum coverage later (see Section 8.1.3.6).

5.2.8. Full Checksum Coverage on Receiving

Name: perMsgChecksumLenRecv

Type: Preference

Default: Require

This property specifies whether the application desires protection against corruption for all data received on this Connection.

5.2.9. Congestion control

Name: congestionControl

Type: Preference

Default: Require

This property specifies whether the application would like the Connection to be congestion controlled or not. Note that if a Connection is not congestion controlled, an application using such a Connection should itself perform congestion control in accordance with [RFC2914]. Also note that reliability is usually combined with congestion control in protocol implementations, rendering "reliable but not congestion controlled" a request that is unlikely to succeed.

5.2.10. Interface Instance or Type

Name: interface

Type: Set (Preference, Enumeration)

Default: Empty set (not setting a preference for any interface)

This property allows the application to select which specific network interfaces or categories of interfaces it wants to "Require", "Prohibit", "Prefer", or "Avoid". Note that marking a specific interface as "Require" strictly limits path selection to a single interface, and may often lead to less flexible and resilient connection establishment.

In contrast to other Selection Properties, this property is a tuple of an (Enumerated) interface identifier and a preference, and can either be implemented directly as such, or for making one preference available for each interface and interface type available on the system.

The set of valid interface types is implementation- and system-specific. For example, on a mobile device, there may be "Wi-Fi" and "Cellular" interface types available; whereas on a desktop computer, there may be "Wi-Fi" and "Wired Ethernet" interface types available. An implementation should provide all types that are supported on the local system to all remote systems, to allow applications to be written generically. For example, if a single implementation is used on both mobile devices and desktop devices, it should define the "Cellular" interface type for both systems, since an application may want to always "Prohibit Cellular".

The set of interface types is expected to change over time as new access technologies become available. The taxonomy of interface types on a given Transport Services system is implementation-specific.

Interface types should not be treated as a proxy for properties of interfaces such as metered or unmetered network access. If an application needs to prohibit metered interfaces, this should be specified via Provisioning Domain attributes (see Section 5.2.11) or another specific property.

5.2.11. Provisioning Domain Instance or Type

Name: pvd

Type: Set (Preference, Enumeration)

Default: Empty set (not setting a preference for any PvD)

Similar to interface instances and types (see Section 5.2.10), this property allows the application to control path selection by selecting which specific Provisioning Domains or categories of Provisioning Domains it wants to "Require", "Prohibit", "Prefer", or "Avoid". Provisioning Domains define consistent sets of network properties that may be more specific than network interfaces [RFC7556].

As with interface instances and types, this property is a tuple of an (Enumerated) PvD identifier and a preference, and can either be implemented directly as such, or for making one preference available for each interface and interface type available on the system.

The identification of a specific Provisioning Domain (PvD) is defined to be implementation- and system-specific, since there is not a portable standard format for a PvD identifier. For example, this identifier may be a string name or an integer. As with requiring specific interfaces, requiring a specific PvD strictly limits path selection.

Categories or types of PvDs are also defined to be implementation- and system-specific. These may be useful to identify a service that is provided by a PvD. For example, if an application wants to use a PvD that provides a Voice-Over-IP service on a Cellular network, it can use the relevant PvD type to require some PvD that provides this service, without needing to look up a particular instance. While this does restrict path selection, it is broader than requiring specific PvD instances or interface instances, and should be preferred over these options.

5.2.12. Use Temporary Local Address

Name: useTemporaryLocalAddress

Type: Preference

Default: Avoid for Listeners and Rendezvous Connections. Prefer for other Connections.

This property allows the application to express a preference for the use of temporary local addresses, sometimes called "privacy" addresses [RFC4941]. Temporary addresses are generally used to prevent linking connections over time when a stable address, sometimes called "permanent" address, is not needed. Note that if an application Requires the use of temporary addresses, the resulting Connection cannot use IPv4, as temporary addresses do not exist in IPv4.

5.2.13. Multi-Paths Transport

Name: multipath

Type: Enumeration

Default: Disabled for connections created through initiate and rendezvous, Passive for listeners

This property specifies whether and how applications want to take advantage of transferring data across multiple paths between the same end hosts. Using multiple paths allows connections to migrate between interfaces or aggregate bandwidth as availability and performance properties change. Possible values are:

Disabled: The connection will not use multiple paths once established, even if the chosen transport supports using multiple paths.

Active: The connection will negotiate the use of multiple paths if the chosen transport supports this.

Passive: The connection will support the use of multiple paths if the remote endpoint requests it.

The policy for using multiple paths is specified using the separate "multipath-policy" property, see Section 7.1.7 below. To enable the peer endpoint to initiate additional paths towards a local address other than the one initially used, it is necessary to set the Alternative Addresses property (see Section 5.2.14 below).

Setting this property to "Active", may have privacy implications: It enables the transport to establish connectivity using alternate paths that may make users linkable across multiple paths, even if the Advertisement of Alternative Addresses property (see Section 5.2.14 below) is set to false.

Enumeration values other than "Disabled" are interpreted as a preference for choosing protocols that can make use of multiple paths. The "Disabled" value implies a requirement not to use multiple paths in parallel but does not prevent choosing a protocol that is capable of using multiple paths, e.g., it does not prevent choosing TCP, but prevents sending the "MP_CAPABLE" option in the TCP handshake.

5.2.14. Advertisement of Alternative Addresses

Name: advertises-altaddr

Type: Boolean

Default: False

This property specifies whether alternative addresses, e.g., of other interfaces, should be advertised to the peer endpoint by the protocol stack. Advertising these addresses enables the peer-endpoint to establish additional connectivity, e.g., for connection migration or using multiple paths.

Note that this may have privacy implications because it may make users linkable across multiple paths. Also, note that setting this to false does not prevent the local transport system from _establishing_ connectivity using alternate paths (see Section 5.2.13 above); it only prevents _procative advertisement_ of addresses.

5.2.15. Direction of communication

Name: direction

Type: Enumeration

Default: Bidirectional

This property specifies whether an application wants to use the connection for sending and/or receiving data. Possible values are:

Bidirectional: The connection must support sending and receiving data

Unidirectional send: The connection must support sending data, and the application cannot use the connection to receive any data

Unidirectional receive: The connection must support receiving data, and the application cannot use the connection to send any data

Since unidirectional communication can be supported by transports offering bidirectional communication, specifying unidirectional communication may cause a transport stack that supports bidirectional communication to be selected.

5.2.16. Notification of excessive retransmissions

Name: retransmitNotify

Type: Preference

Default: Ignore

This property specifies whether an application considers it useful to be informed in case sent data was retransmitted more often than a certain threshold (see Section 7.1.1 for configuration of this threshold).

5.2.17. Notification of ICMP soft error message arrival

Name: softErrorNotify

Type: Preference

Default: Ignore

This property specifies whether an application considers it useful to be informed when an ICMP error message arrives that does not force termination of a connection. When set to true, received ICMP errors will be available as SoftErrors, see Section 7.3.1. Note that even if a protocol supporting this property is selected, not all ICMP errors will necessarily be delivered, so applications cannot rely on receiving them.

5.2.18. Initiating side is not the first to write

Name: activeReadBeforeSend

Type: Preference

Default: Ignore

The most common client-server communication pattern involves the client actively opening a connection, then sending data to the server. The server listens (passive open), reads, and then answers. This property specifies whether an application wants to diverge from this pattern - either by actively opening with Initiate(), immediately followed by reading, or passively opening with Listen(), immediately followed by writing. This property is ignored when establishing connections using Rendezvous(). Requiring this property limits the choice of mappings to underlying protocols, which can reduce efficiency. For example, it prevents the transport system from mapping Connections to SCTP streams, where the first transmitted data takes the role of an active open signal [I-D.ietf-taps-impl].

5.3. Specifying Security Parameters and Callbacks

Most security parameters, e.g., TLS ciphersuites, local identity and private key, etc., may be configured statically. Others are dynamically configured during connection establishment. Thus, we partition security parameters and callbacks based on their place in the lifetime of connection establishment. Similar to Transport Properties, both parameters and callbacks are inherited during cloning (see Section 6.4).

5.3.1. Pre-Connection Parameters

Common parameters such as TLS ciphersuites are known to implementations. Clients should use common safe defaults for these values whenever possible. However, as discussed in [I-D.ietf-taps-transport-security], many transport security protocols require specific security parameters and constraints from the client at the time of configuration and actively during a handshake. These configuration parameters need to be specified in the pre-connection phase and are created as follows:

```
SecurityParameters := NewSecurityParameters()
```

Security configuration parameters and sample usage follow:

- Local identity and private keys: Used to perform private key operations and prove one's identity to the Remote Endpoint. (Note, if private keys are not available, e.g., since they are stored in hardware security modules (HSMs), handshake callbacks must be used. See below for details.)

```
SecurityParameters.Add('identity', identity)
SecurityParameters.Add('keypair', privateKey, publicKey)
```

- Supported algorithms: Used to restrict what parameters are used by underlying transport security protocols. When not specified, these algorithms should use known and safe defaults for the system. Parameters include: ciphersuites, supported groups, and signature algorithms.

```
SecurityParameters.Add('supported-group', 'secp256k1')
SecurityParameters.Add('ciphersuite, 'TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA2
56')
SecurityParameters.Add('signature-algorithm', 'ed25519')
```

o Pre-Shared Key import: Used to install pre-shared keying material established out-of-band. Each pre-shared keying material is associated with some identity that typically identifies its use or has some protocol-specific meaning to the Remote Endpoint.

```
SecurityParameters.Add('pre-shared-key', key, identity)
```

o Session cache management: Used to tune cache capacity, lifetime, re-use, and eviction policies, e.g., LRU or FIFO.may also me changed, but are implementation-specific.

5.3.2. Connection Establishment Callbacks

Security decisions, especially pertaining to trust, are not static. Once configured, parameters may also be supplied during connection establishment. These are best handled as client-provided callbacks. Security handshake callbacks that may be invoked during connection establishment include:

o Trust verification callback: Invoked when a Remote Endpoint's trust must be validated before the handshake protocol can continue.

```
TrustCallback := NewCallback({
  // Handle trust, return the result
})
SecurityParameters.SetTrustVerificationCallback(trustCallback)
```

o Identity challenge callback: Invoked when a private key operation is required, e.g., when local authentication is requested by a remote.

```
ChallengeCallback := NewCallback({
  // Handle challenge
})
SecurityParameters.SetIdentityChallengeCallback(challengeCallback)
```

6. Establishing Connections

Before a Connection can be used for data transfer, it must be established. Establishment ends the pre-establishment phase; all transport properties and cryptographic parameter specification must be complete before establishment, as these will be used to select candidate Paths and Protocol Stacks for the Connection. Establishment may be active, using the Initiate() Action; passive, using the Listen() Action; or simultaneous for peer-to-peer, using the Rendezvous() Action. These Actions are described in the subsections below.

6.1. Active Open: Initiate

Active open is the Action of establishing a Connection to a Remote Endpoint presumed to be listening for incoming Connection requests. Active open is used by clients in client-server interactions. Active open is supported by this interface through the Initiate Action:

```
Connection := Preconnection.Initiate(timeout?)
```

The timeout parameter specifies how long to wait before aborting Active open. Before calling Initiate, the caller must have populated a Preconnection Object with a Remote Endpoint specifier, optionally a Local Endpoint specifier (if not specified, the system will attempt to determine a suitable Local Endpoint), as well as all properties necessary for candidate selection.

The Initiate() Action returns a Connection object. Once Initiate() has been called, any changes to the Preconnection MUST NOT have any effect on the Connection. However, the Preconnection can be reused, e.g., to Initiate another Connection.

Once Initiate is called, the candidate Protocol Stack(s) may cause one or more candidate transport-layer connections to be created to the specified remote endpoint. The caller may immediately begin sending Messages on the Connection (see Section 8.2) after calling Initiate(); note that any data marked "Safely Replayable" that is sent while the Connection is being established may be sent multiple times or on multiple candidates.

The following Events may be sent by the Connection after Initiate() is called:

```
Connection -> Ready<>
```

The Ready Event occurs after Initiate has established a transport-layer connection on at least one usable candidate Protocol Stack over at least one candidate Path. No Receive Events (see Section 8.3) will occur before the Ready Event for Connections established using Initiate.

```
Connection -> EstablishmentError<reason?>
```

An EstablishmentError occurs either when the set of transport properties and security parameters cannot be fulfilled on a Connection for initiation (e.g. the set of available Paths and/or Protocol Stacks meeting the constraints is empty) or reconciled with the local and/or remote Endpoints; when the remote specifier cannot be resolved; or when no transport-layer connection can be established to the remote Endpoint (e.g. because the remote Endpoint is not accepting connections, the application is prohibited from opening a Connection by the operating system, or the establishment attempt has timed out for any other reason).

See also Section 8.2.6 to combine Connection establishment and transmission of the first message in a single action.

6.2. Passive Open: Listen

Passive open is the Action of waiting for Connections from remote Endpoints, commonly used by servers in client-server interactions. Passive open is supported by this interface through the Listen Action and returns a Listener object:

```
Listener := Preconnection.Listen()
```

Before calling Listen, the caller must have initialized the Preconnection during the pre-establishment phase with a Local Endpoint specifier, as well as all properties necessary for Protocol Stack selection. A Remote Endpoint may optionally be specified, to constrain what Connections are accepted.

The Listen() Action returns a Listener object. Once Listen() has been called, any changes to the Preconnection MUST NOT have any effect on the Listener. The Preconnection can be disposed of or reused, e.g., to create another Listener.

Listening continues until the global context shuts down, or until the Stop action is performed on the Listener object:

```
Listener.Stop()
```

After Stop() is called, the Listener can be disposed of.

```
Listener -> ConnectionReceived<Connection>
```

The ConnectionReceived Event occurs when a Remote Endpoint has established a transport-layer connection to this Listener (for Connection-oriented transport protocols), or when the first Message has been received from the Remote Endpoint (for Connectionless protocols), causing a new Connection to be created. The resulting Connection is contained within the ConnectionReceived Event, and is ready to use as soon as it is passed to the application via the event.

```
Listener.SetNewConnectionLimit(value)
```

If the caller wants to rate-limit the number of inbound Connections that will be delivered, it can set a cap using SetNewConnectionLimit(). This mechanism allows a server to protect itself from being drained of resources. Each time a new Connection is delivered by the ConnectionReceived Event, the value is automatically decremented. Once the value reaches zero, no further Connections will be delivered until the caller sets the limit to a higher value. By default, this value is Infinite. The caller is also able to reset the value to Infinite at any point.

```
Listener -> EstablishmentError<reason?>
```

An EstablishmentError occurs either when the Properties and Security Parameters of the Preconnection cannot be fulfilled for listening or cannot be reconciled with the Local Endpoint (and/or Remote Endpoint, if specified), when the Local Endpoint (or Remote Endpoint, if specified) cannot be resolved, or when the application is prohibited from listening by policy.

```
Listener -> Stopped<>
```

A Stopped Event occurs after the Listener has stopped listening.

6.3. Peer-to-Peer Establishment: Rendezvous

Simultaneous peer-to-peer Connection establishment is supported by the Rendezvous() Action:

```
Preconnection.Rendezvous()
```

The Preconnection Object must be specified with both a Local Endpoint and a Remote Endpoint, and also the transport properties and security parameters needed for Protocol Stack selection.

The Rendezvous() Action causes the Preconnection to listen on the Local Endpoint for an incoming Connection from the Remote Endpoint, while simultaneously trying to establish a Connection from the Local Endpoint to the Remote Endpoint. This corresponds to a TCP simultaneous open, for example.

The Rendezvous() Action returns a Connection object. Once Rendezvous() has been called, any changes to the Preconnection MUST NOT have any effect on the Connection. However, the Preconnection can be reused, e.g., for Rendezvous of another Connection.

```
Preconnection -> RendezvousDone<Connection>
```

The RendezvousDone<> Event occurs when a Connection is established with the Remote Endpoint. For Connection-oriented transports, this occurs when the transport-layer connection is established; for Connectionless transports, it occurs when the first Message is received from the Remote Endpoint. The resulting Connection is contained within the RendezvousDone<> Event, and is ready to use as soon as it is passed to the application via the Event.

```
Preconnection -> EstablishmentError<reason?>
```

An EstablishmentError occurs either when the Properties and Security Parameters of the Preconnection cannot be fulfilled for rendezvous or cannot be reconciled with the Local and/or Remote Endpoints, when the Local Endpoint or Remote Endpoint cannot be resolved, when no transport-layer connection can be established to the Remote Endpoint, or when the application is prohibited from rendezvous by policy.

When using some NAT traversal protocols, e.g., Interactive Connectivity Establishment (ICE) [RFC5245], it is expected that the Local Endpoint will be configured with some method of discovering NAT bindings, e.g., a Session Traversal Utilities for NAT (STUN) server. In this case, the Local Endpoint may resolve to a mixture of local and server reflexive addresses. The Resolve() action on the Preconnection can be used to discover these bindings:

```
[]Preconnection := Preconnection.Resolve()
```

The Resolve() call returns a list of Preconnection Objects, that represent the concrete addresses, local and server reflexive, on which a Rendezvous() for the Preconnection will listen for incoming Connections. These resolved Preconnections will share all other Properties with the Preconnection from which they are derived, though some Properties may be made more-specific by the resolution process. This list can be passed to a peer via a signalling protocol, such as SIP [RFC3261] or WebRTC [RFC7478], to configure the remote.

6.4. Connection Groups

Entangled Connections can be created using the Clone Action:

```
Connection := Connection.Clone()
```

Calling Clone on a Connection yields a group of two Connections: the parent Connection on which Clone was called, and the resulting cloned Connection. These connections are "entangled" with each other, and become part of a Connection Group. Calling Clone on any of these two Connections adds a third Connection to the Connection Group, and so on. Connections in a Connection Group generally share Connection

Properties. However, there may be exceptions, such as "Priority (Connection)", see Section 7.1.3. Like all other Properties, Priority is copied to the new Connection when calling Clone(), but it is not entangled: Changing Priority on one Connection does not change it on the other Connections in the same Connection Group.

It is also possible to check which Connections belong to the same Connection Group. Calling GroupedConnections() on a specific Connection returns a set of all Connections in the same group.

```
[]Connection := Connection.GroupedConnections()
```

Connections will be in the same group if the application previously called Clone. Passive Connections can also be added to the same group - e.g., when a Listener receives a new Connection that is just a new stream of an already active multi-streaming protocol instance.

Changing one of the Connection Properties on one Connection in the group changes it for all others. Message Properties, however, are not entangled. For example, changing "Timeout for aborting Connection" (see Section 7.1.4) on one Connection in a group will automatically change this Connection Property for all Connections in the group in the same way. However, changing "Lifetime" (see Section 8.1.3.1) of a Message will only affect a single Message on a single Connection, entangled or not.

If the underlying protocol supports multi-streaming, it is natural to use this functionality to implement Clone. In that case, entangled Connections are multiplexed together, giving them similar treatment not only inside endpoints but also across the end-to-end Internet path.

Note that calling Clone() may result in on-the-wire signaling, e.g., to open a new connection, depending on the underlying Protocol Stack. When Clone() leads to multiple connections being opened instead of multi-streaming, the transport system will ensure consistency of Connection Properties by uniformly applying them to all underlying connections in a group. Even in such a case, there are possibilities for a transport system to implement prioritization within a Connection Group [TCP-COUPLING] [RFC8699].

Attempts to clone a Connection can result in a CloneError:

```
Connection -> CloneError<reason?>
```

The Connection Property "Priority" operates on entangled Connections as in Section 8.1.3.2: when allocating available network capacity among Connections in a Connection Group, sends on Connections with higher Priority values will be prioritized over sends on Connections with lower Priority values. A transport system implementation should, if possible, assign each Connection the capacity share (M-N) x C / M, where N is the Connection's Priority value, M is the maximum Priority value used by all Connections in the group and C is the total available capacity. However, the Priority setting is purely advisory, and no guarantees are given about the way capacity is shared. Each implementation is free to implement a way to share capacity that it sees fit.

7. Managing Connections

During pre-establishment and after establishment, connections can be configured and queried using Connection Properties, and asynchronous information may be available about the state of the connection via Soft Errors.

Connection Properties represent the configuration and state of the selected Protocol Stack(s) backing a Connection. These Connection Properties may be Generic, applying regardless of transport protocol, or Specific, applicable to a single implementation of a single transport protocol stack. Generic Connection Properties are defined in Section 7.1 below. Specific Protocol Properties are defined in a transport- and implementation-specific way, and must not be assumed to apply across different protocols. Attempts to set Specific Protocol Properties on a protocol stack not containing that specific protocol are simply ignored, and do not raise an error; however, too much reliance by an application on Specific Protocol Properties may significantly reduce the flexibility of a transport services implementation.

The application can set and query Connection Properties on a per-Connection basis. Connection Properties that are not read-only can be set during pre-establishment (see Section 5.2), as well as on connections directly using the SetProperty action:

```
Connection.SetProperty(property, value)
```

Note that changing one of the Connection Properties on one Connection in a Connection Group will also change it for all other Connections of that group; see further Section 6.4.

At any point, the application can query Connection Properties.

```
ConnectionProperties := Connection.GetProperties()
```

Depending on the status of the connection, the queried Connection Properties will include different information:

- The connection state, which can be one of the following: Establishing, Established, Closing, or Closed.

- Whether the connection can be used to send data. A connection can not be used for sending if the connection was created with the Selection Property "Direction of Communication" set to "unidirectional receive" or if a Message marked as "Final" was sent over this connection, see Section 8.1.3.5.

- Whether the connection can be used to receive data. A connection can not be used for reading if the connection was created with the Selection Property "Direction of Communication" set to "unidirectional send" or if a Message marked as "Final" was received, see Section 8.3.3.3. The latter is only supported by certain transport protocols, e.g., by TCP as half-closed connection.

- For Connections that are Establishing: Transport Properties that the application specified on the Preconnection, see Section 5.2.

- For Connections that are Established, Closing, or Closed: Selection (Section 5.2) and Connection Properties (Section 7.1) of the actual protocols that were selected and instantiated. Selection Properties indicate whether or not the Connection has or offers a certain Selection Property. Note that the actually instantiated protocol stack may not match all Protocol Selection Properties that the application specified on the Preconnection. For example, a certain Protocol Selection Property that an application specified as Preferred may not actually be present in the chosen protocol stack because none of the currently available transport protocols had this feature.

- For Connections that are Established, additional properties of the path(s) in use. These properties can be derived from the local provisioning domain [RFC7556], measurements by the Protocol Stack, or other sources.

7.1. Generic Connection Properties

Generic Connection Properties are defined independent of the chosen protocol stack and therefore available on all Connections.

Note that many Connection Properties have a corresponding Selection Property which enables applications to express their preference for protocols providing a supporting transport feature.

7.1.1. Retransmission Threshold Before Excessive Retransmission Notification

Name: retransmitNotifyThreshold

Type: Integer, with special value "Disabled"

Default: Disabled

This property specifies after how many retransmissions to inform the application about "Excessive Retransmissions".

7.1.2. Required Minimum Corruption Protection Coverage for Receiving

Name: recvChecksumLen

Type: Integer, with special value "Full Coverage"

Default: Full Coverage

This property specifies the part of the received data that needs to be covered by a checksum. It is given in Bytes. A value of 0 means that no checksum is required.

7.1.3. Priority (Connection)

Name: connPrio

Type: Integer

Default: 100

This Property is a non-negative integer representing the relative inverse priority (i.e., a lower value reflects a higher priority) of this Connection relative to other Connections in the same Connection Group. It has no effect on Connections not part of a Connection Group. As noted in Section 6.4, this property is not entangled when Connections are cloned, i.e., changing the Priority on one Connection in a Connection Group does not change it on the other Connections in the same Connection Group.

7.1.4. Timeout for Aborting Connection

Name: connTimeout

Type: Numeric, with special value "Disabled"

Default: Disabled

This property specifies how long to wait before deciding that a Connection has failed when trying to reliably deliver data to the destination. Adjusting this Property will only take effect when the underlying stack supports reliability. The special value "Disabled" means that this timeout is not scheduled to happen. This can be a valid choice with unreliable data transfer (e.g., when UDP is the underlying transport protocol).

7.1.5. Connection Group Transmission Scheduler

Name: connScheduler

Type: Enumeration

Default: Weighted Fair Queueing (see Section 3.6 in [RFC8260])

This property specifies which scheduler should be used among Connections within a Connection Group, see Section 6.4. The set of schedulers can be taken from [RFC8260].

7.1.6. Capacity Profile

Name: connCapacityProfile

This property specifies the desired network treatment for traffic sent by the application and the tradeoffs the application is prepared to make in path and protocol selection to receive that desired treatment. When the capacity profile is set to a value other than Default, the transport system SHOULD select paths and configure protocols to optimize the tradeoff between delay, delay variation, and efficient use of the available capacity based on the capacity profile specified. How this is realized is implementation-specific. The Capacity Profile MAY also be used to set priorities on the wire for Protocol Stacks supporting prioritization. Recommendations for use with DSCP are provided below for each profile; note that when a Connection is multiplexed, the guidelines in Section 6 of [RFC7657] apply.

The following values are valid for the Capacity Profile:

Default: The application provides no information about its expected capacity profile. Transport system implementations that map the requested capacity profile onto per-connection DSCP signaling SHOULD assign the DSCP Default Forwarding [RFC2474] PHB.

Scavenger: The application is not interactive. It expects to send and/or receive data without any urgency. This can, for example, be used to select protocol stacks with scavenger transmission control and/or to assign the traffic to a lower-effort service. Transport system implementations that map the requested capacity profile onto per-connection DSCP signaling SHOULD assign the DSCP Less than Best Effort [RFC8622] PHB.

Low Latency/Interactive: The application is interactive, and prefers loss to latency. Response time should be optimized at the expense of delay variation and efficient use of the available capacity when sending on this connection. This can be used by the system to disable the coalescing of multiple small Messages into larger packets (Nagle's algorithm); to prefer immediate acknowledgment from the peer endpoint when supported by the underlying transport; and so on. Transport system implementations that map the requested capacity profile onto per-connection DSCP signaling without multiplexing SHOULD assign a DSCP Assured Forwarding (AF41,AF42,AF43,AF44) [RFC2597] PHB. Inelastic traffic that is expected to conform to the configured network service rate could be mapped to the DSCP Expedited Forwarding [RFC3246] or [RFC5865] PHBs.

Low Latency/Non-Interactive: The application prefers loss to latency but is not interactive. Response time should be optimized at the expense of delay variation and efficient use of the available capacity when sending on this connection. Transport system implementations that map the requested capacity profile onto per-connection DSCP signaling without multiplexing SHOULD assign a DSCP Assured Forwarding (AF21,AF22,AF23,AF24) [RFC2597] PHB.

Constant-Rate Streaming: The application expects to send/receive data at a constant rate after Connection establishment. Delay and delay variation should be minimized at the expense of efficient use of the available capacity. This implies that the Connection may fail if the desired rate cannot be maintained across the Path. A transport may interpret this capacity profile as preferring a circuit breaker [RFC8084] to a rate-adaptive congestion controller. Transport system implementations that map the requested capacity profile onto per-connection DSCP signaling without multiplexing SHOULD assign a DSCP Assured Forwarding (AF31,AF32,AF33,AF34) [RFC2597] PHB.

Capacity-Seeking: The application expects to send/receive data at the maximum rate allowed by its congestion controller over a relatively long period of time. Transport system implementations that map the requested capacity profile onto per-connection DSCP signaling without multiplexing SHOULD assign a DSCP Assured Forwarding (AF11,AF12,AF13,AF14) [RFC2597] PHB per Section 4.8 of [RFC4594].

The Capacity Profile for a selected protocol stack may be modified on a per-Message basis using the Transmission Profile Message Property; see Section 8.1.3.8.

7.1.7. Policy for using Multi-Path Transports

Name: multipath-policy

Type: Enumeration

Default: Handover

This property specifies the local policy of transferring data across multiple paths between the same end hosts if Parallel Use of Multiple Paths not set to Disabled (see Section 5.2.13). Possible values are:

Handover: The connection should only attempt to migrate between different paths when the original path is lost or becomes unusable. The actual thresholds to declare a path unusable are implementation specific.

Interactive: The connection should attempt to minimize the latency for interactive traffic patterns by transmitting data across multiple paths when it is beneficial to do so. The goal of minimizing the latency will be balanced against the cost of each of these paths, meaning that depending on the cost of the lower-latency path, the scheduling might choose to use a higher-latency path. Traffic can be scheduled such that data may be transmitted on multiple paths in parallel to achieve the lowest latency possible. The specific scheduling algorithm is implementation-specific.

Aggregate: The connection should attempt to use multiple paths in parallel in order to maximize bandwidth and possibly overcome bandwidth limitations of the individual paths. The actual strategy is implementation specific.

Note that this is a local choice - the peer endpoint can choose a different policy.

7.1.8. Bounds on Send or Receive Rate

Name: maxSendRate / maxRecvRate

Type: Numeric (with special value "Unlimited") / Numeric (with special value "Unlimited")

Default: Unlimited / Unlimited

This property specifies an upper-bound rate that a transfer is not expected to exceed (even if flow control and congestion control allow higher rates), and/or a lower-bound rate below which the application does not deem a data transfer useful. It is given in bits per second. The special value "Unlimited" indicates that no bound is specified.

7.1.9. Read-only Connection Properties

The following generic Connection Properties are read-only, i.e. they cannot be changed by an application.

7.1.9.1. Maximum Message Size Concurrent with Connection Establishment

Name: zeroRttMsgMaxLen

Type: Integer

This property represents the maximum Message size that can be sent before or during Connection establishment, see also Section 8.1.3.4. It is given in Bytes.

7.1.9.2. Maximum Message Size Before Fragmentation or Segmentation

Name: singularTransmissionMsgMaxLen

Type: Integer

This property, if applicable, represents the maximum Message size that can be sent without incurring network-layer fragmentation or transport layer segmentation at the sender. This property exposes the Maximum Packet Size (MPS) as described in Datagram PLPMTUD [I-D.ietf-tsvwg-datagram-plpmtud].

7.1.9.3. Maximum Message Size on Send

Name: sendMsgMaxLen

Type: Integer

This property represents the maximum Message size that can be sent using a send operation.

7.1.9.4. Maximum Message Size on Receive

Name: recvMsgMaxLen

Type: Integer

This numeric property represents the maximum Message size that can be received.

7.2. TCP-specific Properties: User Timeout Option (UTO)

These properties specify configurations for the User Timeout Option (UTO), in case TCP becomes the chosen transport protocol. Implementation is optional and of course only sensible if TCP is implemented in the transport system.

These TCP-specific properties are included here because the feature "Suggest timeout to the peer" is part of the minimal set of transport services [I-D.ietf-taps-minset], where this feature was categorized as "functional". This means that when an implementation offers this feature, it has to expose an interface to it to the application. Otherwise, the implementation might violate assumptions by the application, which could cause the application to fail.

All of the below properties are optional (e.g., it is possible to specify "User Timeout Enabled" as true, but not specify an Advertised User Timeout value; in this case, the TCP default will be used).

7.2.1. Advertised User Timeout

Name: tcp.userTimeoutValue

Type: Integer

Default: the TCP default

This time value is advertised via the TCP User Timeout Option (UTO) [RFC5482] at the remote endpoint to adapt its own "Timeout for aborting Connection" (see Section 7.1.4) value accordingly.

7.2.2. User Timeout Enabled

Name: tcp.userTimeout

Type: Boolean

Default: false

This property controls whether the UTO option is enabled for a connection. This applies to both sending and receiving.

7.2.3. Timeout Changeable

Name: tcp.userTimeoutRecv

Type: Boolean

Default: true

This property controls whether the "Timeout for aborting Connection" (see Section 7.1.4) may be changed based on a UTO option received from the remote peer. This boolean becomes false when "Timeout for aborting Connection" (see Section 7.1.4) is used.

7.3. Connection Lifecycle Events

During the lifetime of a connection there are events that can occur when configured.

7.3.1. Soft Errors

Asynchronous introspection is also possible, via the SoftError Event. This event informs the application about the receipt and contents of an ICMP error message related to the Connection. This will only happen if the underlying protocol stack supports access to soft errors; however, even if the underlying stack supports it, there is no guarantee that a soft error will be signaled.

Connection -> SoftError<>

7.3.2. Excessive retransmissions

This event notifies the application of excessive retransmissions, based on a configured threshold (see Section 7.1.1). This will only happen if the underlying protocol stack supports reliability and, with it, such notifications.

Connection -> ExcessiveRetransmission<>

8. Data Transfer

Data is sent and received as Messages, which allows the application to communicate the boundaries of the data being transferred.

8.1. Messages and Framers

Each Message has an optional Message Context, which allows to add Message Properties, identify Send Events related to a specific Message or to inspect meta-data related to the Message sent. Framers can be used to extend or modify the message data with additional information that can be processed at the receiver to detect message boundaries.

8.1.1. Message Contexts

Using the MessageContext object, the application can set and retrieve meta-data of the message, including Message Properties (see Section 8.1.3) and framing meta-data (see Section 8.1.2.2). Therefore, a MessageContext object can be passed to the Send action and is returned by each Send and Receive related event.

Message Properties can be set and queried using the Message Context:

```
MessageContext.add(scope?, parameter, value)
PropertyValue := MessageContext.get(scope?, property)
```

To get or set Message Properties, the optional scope parameter is left empty. To get or set meta-data for a Framer, the application has to pass a reference to this Framer as the scope parameter.

For MessageContexts returned by send events (see Section 8.2.3) and receive events (see Section 8.3.2), the application can query information about the local and remote endpoint:

```
RemoteEndpoint := MessageContext.GetRemoteEndpoint()
LocalEndpoint := MessageContext.GetLocalEndpoint()
```

Message Contexts can also be used to send messages in reply to other messages, see Section 8.2.2 for details.

8.1.2. Message Framers

Although most applications communicate over a network using well-formed Messages, the boundaries and metadata of the Messages are often not directly communicated by the transport protocol itself. For example, HTTP applications send and receive HTTP messages over a byte-stream transport, requiring that the boundaries of HTTP messages be parsed out from the stream of bytes.

Message Framers allow extending a Connection's Protocol Stack to define how to encapsulate or encode outbound Messages, and how to decapsulate or decode inbound data into Messages. Message Framers allow message boundaries to be preserved when using a Connection object, even when using byte-stream transports. This facility is designed based on the fact that many of the current application protocols evolved over TCP, which does not provide message boundary preservation, and since many of these protocols require message boundaries to function, each application layer protocol has defined its own framing.

To use a Message Framer, the application adds it to its Preconnection object. Then, the Message Framer can intercept all calls to Send() or Receive() on a Connection to add Message semantics, in addition to interacting with the setup and teardown of the Connection. A Framer can start sending data before the application sends data if the framing protocol requires a prefix or handshake (see [RFC8229] for an example of such a framing protocol).

```
     Initiate()   Send()   Receive()   Close()
         |          |         ^          |
         |          |         |          |
    +----v----------v---------+----------v-----+
    |                Connection                |
    +----+----------+---------^----------+-----+
         |          |         |          |
         |      +-----------------+      |
         |      |    Messages     |      |
         |      +-----------------+      |
         |          |         |          |
    +----v----------v---------+----------v-----+
    |                Framer(s)                 |
    +----+----------+---------^----------+-----+
         |          |         |          |
         |      +-----------------+      |
         |      |   Byte-stream   |      |
         |      +-----------------+      |
         |          |         |          |
    +----v----------v---------+----------v-----+
    |         Transport Protocol Stack         |
    +------------------------------------------+
```

Note that while Message Framers add the most value when placed above a protocol that otherwise does not preserve message boundaries, they can also be used with datagram- or message-based protocols. In these cases, they add an additional transformation to further encode or encapsulate, and can potentially support packing multiple application-layer Messages into individual transport datagrams.

The API to implement a Message Framer can vary depending on the implementation; guidance on implementing Message Framers can be found in [I-D.ietf-taps-impl].

8.1.2.1. Adding Message Framers to Connections

The Message Framer object can be added to one or more Preconnections to run on top of transport protocols. Multiple Framers may be added. If multiple Framers are added, the last one added runs first when framing outbound messages, and last when parsing inbound data.

The following example adds a basic HTTP Message Framer to a Preconnection:

```
framer := NewHTTPMessageFramer()
Preconnection.AddFramer(framer)
```

8.1.2.2. Framing Meta-Data

When sending Messages, applications can add specific Message values to a MessageContext (Section 8.1.1) that is intended for a Framer. This can be used, for example, to set the type of a Message for a TLV format. The namespace of values is custom for each unique Message Framer.

```
messageContext := NewMessageContext()
messageContext.add(framer, key, value)
Connection.Send(messageData, messageContext)
```

When an application receives a MessageContext in a Receive event, it can also look to see if a value was set by a specific Message Framer.

```
messageContext.get(framer, key) -> value
```

For example, if an HTTP Message Framer is used, the values could correspond to HTTP headers:

```
httpFramer := NewHTTPMessageFramer()
...
messageContext := NewMessageContext()
messageContext.add(httpFramer, "accept", "text/html")
```

8.1.3. Message Properties

Applications may need to annotate the Messages they send with extra information to control how data is scheduled and processed by the transport protocols in the Connection. Therefore a message context containing these properties can be passed to the Send Action. For other uses of the message context, see Section 8.1.1.

Note that Message Properties are per-Message, not per-Send if partial Messages are sent (Section 8.2.4). All data blocks associated with a single Message share properties specified in the Message Contexts. For example, it would not make sense to have the beginning of a Message expire, but allow the end of a Message to still be sent.

A MessageContext object contains metadata for Messages to be sent or received.

```
messageData := "hello"
messageContext := NewMessageContext()
messageContext.add(parameter, value)
Connection.Send(messageData, messageContext)
```

The simpler form of Send, which does not take any messageContext, is equivalent to passing a default MessageContext without adding any Message Properties to it.

If an application wants to override Message Properties for a specific message, it can acquire an empty MessageContext Object and add all desired Message Properties to that Object. It can then reuse the same messageContext Object for sending multiple Messages with the same properties.

Properties may be added to a MessageContext object only before the context is used for sending. Once a messageContext has been used with a Send call, modifying any of its properties is invalid.

Message Properties may be inconsistent with the properties of the Protocol Stacks underlying the Connection on which a given Message is sent. For example, a Connection must provide reliability to allow setting an infinite value for the lifetime property of a Message. Sending a Message with Message Properties inconsistent with the Selection Properties of the Connection yields an error.

Connection Properties describe the default behavior for all Messages on a Connection. If a Message Property contradicts a Connection Property, and if this per-Message behavior can be supported, it overrides the Connection Property for the specific Message. For example, if "Reliable Data Transfer (Connection)" is set to "Require" and a protocol with configurable per-Message reliability is used, setting "Reliable Data Transfer (Message)" to "false" for a particular Message will allow this Message to be unreliably delivered. Note that changing the Reliable Data Transfer property on Messages is only possible for Connections that were established with the Selection Property "Configure Per-Message Reliability" enabled.

The following Message Properties are supported:

8.1.3.1. Lifetime

Name: msgLifetime

Type: Numeric

Default: infinite

Lifetime specifies how long a particular Message can wait to be sent to the remote endpoint before it is irrelevant and no longer needs to be (re-)transmitted. This is a hint to the transport system - it is not guaranteed that a Message will not be sent when its Lifetime has expired.

Setting a Message's Lifetime to infinite indicates that the application does not wish to apply a time constraint on the transmission of the Message, but it does not express a need for reliable delivery; reliability is adjustable per Message via the "Reliable Data Transfer (Message)" property (see Section 8.1.3.7). The type and units of Lifetime are implementation-specific.

8.1.3.2. Priority

Name: msgPrio

Type: Integer (non-negative)

Default: 100

This property represents a hierarchy of priorities. It can specify the priority of a Message, relative to other Messages sent over the same Connection.

A Message with Priority 0 will yield to a Message with Priority 1, which will yield to a Message with Priority 2, and so on. Priorities may be used as a sender-side scheduling construct only, or be used to specify priorities on the wire for Protocol Stacks supporting prioritization.

Note that this property is not a per-message override of the connection Priority - see Section 7.1.3. Both Priority properties may interact, but can be used independently and be realized by different mechanisms.

8.1.3.3. Ordered

Name: msgOrdered

Type: Boolean

Default: true

If true, it specifies that the receiver-side transport protocol stack may only deliver the Message to the receiving application after the previous ordered Message which was passed to the same Connection via the Send Action, when such a Message exists. If false, the Message may be delivered to the receiving application out of order. This property is used for protocols that support preservation of data ordering, see Section 5.2.4, but allow out-of-order delivery for certain messages, e.g., by multiplexing independent messages onto different streams.

8.1.3.4. Safely Replayable

Name: safelyReplayable

Type: Boolean

Default: false

If true, it specifies that a Message is safe to send to the remote endpoint more than once for a single Send Action. It is used to mark data safe for certain 0-RTT establishment techniques, where retransmission of the 0-RTT data may cause the remote application to receive the Message multiple times.

Note that for protocols that do not protect against duplicated messages, e.g., UDP, all messages MUST be marked as "Safely Replayable". In order to enable protocol selection to choose such a protocol, "Safely Replayable" MUST be added to the TransportProperties passed to the Preconnection. If such a protocol was chosen, disabling "Safely Replayable" on individual messages MUST result in a SendError.

8.1.3.5. Final

Name: final

Type: Boolean

Default: false

If true, this Message is the last one that the application will send on a Connection. This allows underlying protocols to indicate to the Remote Endpoint that the Connection has been effectively closed in the sending direction. For example, TCP-based Connections can send a FIN once a Message marked as Final has been completely sent, indicated by marking endOfMessage. Protocols that do not support signalling the end of a Connection in a given direction will ignore this property.

Note that a Final Message must always be sorted to the end of a list of Messages. The Final property overrides Priority and any other property that would re-order Messages. If another Message is sent after a Message marked as Final has already been sent on a Connection, the Send Action for the new Message will cause a SendError Event.

8.1.3.6. Corruption Protection Length

Name: msgChecksumLen

Type: Integer (non-negative with special value "Full Coverage")

Default: Full Coverage

This property specifies the minimum length of the section of the Message, starting from byte 0, that the application requires to be delivered without corruption due to lower layer errors. It is used to specify options for simple integrity protection via checksums. A value of 0 means that no checksum is required, and "Full Coverage" means that the entire Message is protected by a checksum. Only "Full Coverage" is guaranteed, any other requests are advisory, meaning that "Full Coverage" is applied anyway.

8.1.3.7. Reliable Data Transfer (Message)

Name: msgReliable

Type: Boolean

Default: true

When true, this property specifies that a message should be sent in such a way that the transport protocol ensures all data is received on the other side without corruption. Changing the "Reliable Data Transfer" property on Messages is only possible for Connections that were established with the Selection Property "Configure Per-Message Reliability" enabled. When this is not the case, changing it will generate an error. Disabling this property indicates that the transport system may disable retransmissions or other reliability mechanisms for this particular Message, but such disabling is not guaranteed.

8.1.3.8. Message Capacity Profile Override

Name: msgCapacityProfile

Type: Enumeration

This enumerated property specifies the application's preferred tradeoffs for sending this Message; it is a per-Message override of the Capacity Profile connection property (see Section 7.1.6).

8.1.3.9. No Fragmentation

Name: noFragmentation

Type: Boolean

Default: false

This property specifies that a message should be sent and received as a single packet without network-layer fragmentation, if possible. Attempts to send a message with this property set with a size greater to the transport's current estimate of its maximum transmission segment size will result in a "SendError". When used with transports supporting this functionality and running over IP version 4, the Don't Fragment bit will be set.

8.2. Sending Data

Once a Connection has been established, it can be used for sending Messages. By default, Send enqueues a complete Message, and takes optional per-Message properties (see Section 8.2.1). All Send actions are asynchronous, and deliver events (see Section 8.2.3). Sending partial Messages for streaming large data is also supported (see Section 8.2.4).

Messages are sent on a Connection using the Send action:

```
Connection.Send(messageData, messageContext?, endOfMessage?)
```

where messageData is the data object to send, and messageContext allows adding Message Properties, identifying Send Events related to a specific Message or inspecting meta-data related to the Message sent (see Section 8.1.1).

The optional endOfMessage parameter supports partial sending and is described in Section 8.2.4.

8.2.1. Basic Sending

The most basic form of sending on a connection involves enqueuing a single Data block as a complete Message, with default Message Properties.

```
messageData := "hello"
Connection.Send(messageData)
```

The interpretation of a Message to be sent is dependent on the implementation, and on the constraints on the Protocol Stacks implied by the Connection's transport properties. For example, a Message may be a single datagram for UDP Connections; or an HTTP Request for HTTP Connections.

Some transport protocols can deliver arbitrarily sized Messages, but other protocols constrain the maximum Message size. Applications can query the Connection Property "Maximum Message size on send" (Section 7.1.9.3) to determine the maximum size allowed for a single Message. If a Message is too large to fit in the Maximum Message Size for the Connection, the Send will fail with a SendError event (Section 8.2.3.3). For example, it is invalid to send a Message over a UDP connection that is larger than the available datagram sending size.

8.2.2. Sending Replies

When a message is sent in response to a message received, the application may use the Message Context of the received Message to construct a Message Context for the reply.

```
replyMessageContext := requestMessageContext.reply()
```

By using the "replyMessageContext", the transport system is informed that the message to be sent is a response and can map the response to the same underlying transport connection or stream the request was received from. The concept of Message Contexts is described in Section 8.1.1.

8.2.3. Send Events

Like all Actions in this interface, the Send Action is asynchronous. There are several Events that can be delivered in response to Sending a Message. Exactly one Event (Sent, Expired, or SendError) will be delivered in response to each call to Send.

Note that if partial Sends are used (Section 8.2.4), there will still be exactly one Send Event delivered for each call to Send. For example, if a Message expired while two requests to Send data for that Message are outstanding, there will be two Expired events delivered.

The interface should allow the application to correlate which Send Action resulted in a particular Send Event. The manner in which this correlation is indicated is implementation-specific.

8.2.3.1. Sent

Connection -> Sent<messageContext>

The Sent Event occurs when a previous Send Action has completed, i.e., when the data derived from the Message has been passed down or through the underlying Protocol Stack and is no longer the responsibility of this interface. The exact disposition of the Message (i.e., whether it has actually been transmitted, moved into a buffer on the network interface, moved into a kernel buffer, and so on) when the Sent Event occurs is implementation-specific. The Sent Event contains a reference to the Message to which it applies.

Sent Events allow an application to obtain an understanding of the amount of buffering it creates. That is, if an application calls the Send Action multiple times without waiting for a Sent Event, it has created more buffer inside the transport system than an application that always waits for the Sent Event before calling the next Send Action.

8.2.3.2. Expired

Connection -> Expired<messageContext>

The Expired Event occurs when a previous Send Action expired before completion; i.e. when the Message was not sent before its Lifetime (see Section 8.1.3.1) expired. This is separate from SendError, as it is an expected behavior for partially reliable transports. The Expired Event contains a reference to the Message to which it applies.

8.2.3.3. SendError

Connection -> SendError<messageContext, reason?>

A SendError occurs when a Message could not be sent due to an error condition: an attempt to send a Message which is too large for the system and Protocol Stack to handle, some failure of the underlying

Protocol Stack, or a set of Message Properties not consistent with the Connection's transport properties. The SendError contains a reference to the Message to which it applies.

8.2.4. Partial Sends

It is not always possible for an application to send all data associated with a Message in a single Send Action. The Message data may be too large for the application to hold in memory at one time, or the length of the Message may be unknown or unbounded.

Partial Message sending is supported by passing an endOfMessage boolean parameter to the Send Action. This value is always true by default, and the simpler forms of Send are equivalent to passing true for endOfMessage.

The following example sends a Message in two separate calls to Send.

```
messageContext := NewMessageContext()
messageContext.add(parameter, value)

messageData := "hel"
endOfMessage := false
Connection.Send(messageData, messageContext, endOfMessage)

messageData := "lo"
endOfMessage := true
Connection.Send(messageData, messageContext, endOfMessage)
```

All data sent with the same MessageContext object will be treated as belonging to the same Message, and will constitute an in-order series until the endOfMessage is marked.

8.2.5. Batching Sends

To reduce the overhead of sending multiple small Messages on a Connection, the application may want to batch several Send Actions together. This provides a hint to the system that the sending of these Messages should be coalesced when possible, and that sending any of the batched Messages may be delayed until the last Message in the batch is enqueued.

The semantics for starting and ending a batch can be implementation-specific, but need to allow multiple Send Actions to be enqueued.

```
Connection.StartBatch()
Connection.Send(messageData)
Connection.Send(messageData)
Connection.EndBatch()
```

8.2.6. Send on Active Open: InitiateWithSend

For application-layer protocols where the Connection initiator also sends the first message, the InitiateWithSend() action combines Connection initiation with a first Message sent:

```
Connection := Preconnection.InitiateWithSend(messageData, messageContext?, timeou
t?)
```

Whenever possible, a messageContext should be provided to declare the Message passed to InitiateWithSend as "Safely Replayable". This allows the transport system to make use of 0-RTT establishment in case this is supported by the available protocol stacks. When the selected stack(s) do not support transmitting data upon connection establishment, InitiateWithSend is identical to Initiate() followed by Send().

Neither partial sends nor send batching are supported by InitiateWithSend().

The Events that may be sent after InitiateWithSend() are equivalent to those that would be sent by an invocation of Initiate() followed immediately by an invocation of Send(), with the caveat that a send failure that occurs because the Connection could not be established will not result in a SendError separate from the InitiateError signaling the failure of Connection establishment.

8.3. Receiving Data

Once a Connection is established, it can be used for receiving data (unless the "Direction of Communication" property is set to "unidirectional send"). As with sending, data is received in terms of Messages. Receiving is an asynchronous operation, in which each call to Receive enqueues a request to receive new data from the connection. Once data has been received, or an error is encountered, an event will be delivered to complete any pending Receive requests (see Section 8.3.2). If Messages arrive at the transport system before Receive requests are issued, ensuing Receive requests will first operate on these Messages before awaiting any further Messages.

8.3.1. Enqueuing Receives

Receive takes two parameters to specify the length of data that an application is willing to receive, both of which are optional and have default values if not specified.

```
Connection.Receive(minIncompleteLength?, maxLength?)
```

By default, Receive will try to deliver complete Messages in a single event (Section 8.3.2.1).

The application can set a minIncompleteLength value to indicate the smallest partial Message data size in bytes that should be delivered in response to this Receive. By default, this value is infinite, which means that only complete Messages should be delivered (see Section 8.3.2.2 and Section 8.1.2 for more information on how this is accomplished). If this value is set to some smaller value, the associated receive event will be triggered only when at least that many bytes are available, or the Message is complete with fewer bytes, or the system needs to free up memory. Applications should always check the length of the data delivered to the receive event and not assume it will be as long as minIncompleteLength in the case of shorter complete Messages or memory issues.

The maxLength argument indicates the maximum size of a Message in bytes the application is currently prepared to receive. The default value for maxLength is infinite. If an incoming Message is larger than the minimum of this size and the maximum Message size on receive for the Connection’s Protocol Stack, it will be delivered via ReceivedPartial events (Section 8.3.2.2).

Note that maxLength does not guarantee that the application will receive that many bytes if they are available; the interface may return ReceivedPartial events with less data than maxLength according to implementation constraints. Note also that maxLength and minIncompleteLength are intended only to manage buffering, and are not interpreted as a receiver preference for message reordering.

8.3.2. Receive Events

Each call to Receive will be paired with a single Receive Event, which can be a success or an error. This allows an application to provide backpressure to the transport stack when it is temporarily not ready to receive messages.

The interface should allow the application to correlate which call to Receive resulted in a particular Receive Event. The manner in which this correlation is indicated is implementation-specific.

8.3.2.1. Received

Connection -> Received<messageData, messageContext>

A Received event indicates the delivery of a complete Message. It contains two objects, the received bytes as messageData, and the metadata and properties of the received Message as messageContext.

The messageData object provides access to the bytes that were received for this Message, along with the length of the byte array. The messageContext is provided to enable retrieving metadata about the message and referring to the message, e.g., to send replies and map responses to their requests. See Section 8.1.1 for details.

See Section 8.1.2 for handling Message framing in situations where the Protocol Stack only provides a byte-stream transport.

8.3.2.2. ReceivedPartial

Connection -> ReceivedPartial<messageData, messageContext, endOfMessage>

If a complete Message cannot be delivered in one event, one part of the Message may be delivered with a ReceivedPartial event. In order to continue to receive more of the same Message, the application must invoke Receive again.

Multiple invocations of ReceivedPartial deliver data for the same Message by passing the same MessageContext, until the endOfMessage flag is delivered or a ReceiveError occurs. All partial blocks of a single Message are delivered in order without gaps. This event does not support delivering discontiguous partial Messages.

If the minIncompleteLength in the Receive request was set to be infinite (indicating a request to receive only complete Messages), the ReceivedPartial event may still be delivered if one of the following conditions is true:

- the underlying Protocol Stack supports message boundary preservation, and the size of the Message is larger than the buffers available for a single message;

- the underlying Protocol Stack does not support message boundary preservation, and the Message Framer (see Section 8.1.2) cannot determine the end of the message using the buffer space it has available; or

- the underlying Protocol Stack does not support message boundary preservation, and no Message Framer was supplied by the application

Note that in the absence of message boundary preservation or a Message Framer, all bytes received on the Connection will be represented as one large Message of indeterminate length.

8.3.2.3. ReceiveError

Connection -> ReceiveError<messageContext, reason?>

A ReceiveError occurs when data is received by the underlying Protocol Stack that cannot be fully retrieved or parsed, or when some other indication is received that reception has failed. In contrast, conditions that irrevocably lead to the termination of the Connection are signaled using ConnectionError instead (see Section 9).

The ReceiveError event passes an optional associated MessageContext. This may indicate that a Message that was being partially received previously, but had not completed, encountered an error and will not be completed.

8.3.3. Receive Message Properties

Each Message Context may contain metadata from protocols in the Protocol Stack; which metadata is available is Protocol Stack dependent. These are exposed though additional read-only Message Properties that can be queried from the MessageContext object (see Section 8.1.1) passed by the receive event. The following metadata values are supported:

8.3.3.1. UDP(-Lite)-specific Property: ECN

When available, Message metadata carries the value of the Explicit Congestion Notification (ECN) field. This information can be used for logging and debugging purposes, and for building applications which need access to information about the transport internals for their own operation. This property is specific to UDP and UDP-Lite because these protocols do not implement congestion control, and hence expose this functionality to the application.

8.3.3.2. Early Data

In some cases it may be valuable to know whether data was read as part of early data transfer (before connection establishment has finished). This is useful if applications need to treat early data separately, e.g., if early data has different security properties than data sent after connection establishment. In the case of TLS 1.3, client early data can be replayed maliciously (see [RFC8446]). Thus, receivers may wish to perform additional checks for early data to ensure it is safely replayable. If TLS 1.3 is available and the recipient Message was sent as part of early data, the corresponding metadata carries a flag indicating as such. If early data is enabled, applications should check this metadata field for Messages received during connection establishment and respond accordingly.

8.3.3.3. Receiving Final Messages

The Message Context can indicate whether or not this Message is the Final Message on a Connection. For any Message that is marked as Final, the application can assume that there will be no more Messages received on the Connection once the Message has been completely delivered. This corresponds to the Final property that may be marked on a sent Message, see Section 8.1.3.5.

Some transport protocols and peers may not support signaling of the Final property. Applications therefore should not rely on receiving a Message marked Final to know that the other endpoint is done sending on a connection.

Any calls to Receive once the Final Message has been delivered will result in errors.

9. Connection Termination

Close terminates a Connection after satisfying all the requirements that were specified regarding the delivery of Messages that the application has already given to the transport system. For example, if reliable delivery was requested for a Message handed over before calling Close, the transport system will ensure that this Message is indeed delivered. If the Remote Endpoint still has data to send, it cannot be received after this call.

```
Connection.Close()
```

The Closed Event can inform the application that the Remote Endpoint has closed the Connection; however, there is no guarantee that a remote Close will indeed be signaled.

```
Connection -> Closed<>
```

Abort terminates a Connection without delivering remaining data:

```
Connection.Abort()
```

A ConnectionError informs the application that data to could not be delivered after a timeout, or the other side has aborted the Connection; however, there is no guarantee that an Abort will indeed be signaled.

```
Connection -> ConnectionError<reason?>
```

10. Connection State and Ordering of Operations and Events

As this interface is designed to be independent of an implementation's concurrency model, the details of how exactly actions are handled, and how events are dispatched, are implementation dependent.

Each transition of connection state is associated with one of more events:

- Ready<> occurs when a Connection created with Initiate() or InitiateWithSend() transitions to Established state.
- ConnectionReceived<> occurs when a Connection created with Listen() transitions to Established state.
- RendezvousDone<> occurs when a Connection created with Rendezvous() transitions to Established state.
- Closed<> occurs when a Connection transitions to Closed state without error.
- InitiateError<> occurs when a Connection created with Initiate() transitions from Establishing state to Closed state due to an error.
- ConnectionError<> occurs when a Connection transitions to Closed state due to an error in all other circumstances.

The following diagram shows the possible states of a Connection and the events that occur upon a transition from one state to another.

```
                (*)                 (**)
Establishing -----> Established -----> Closed
      |                                  ^
      |                                  |
      +----------------------------------+
                  InitiateError<>

(*) Ready<>, ConnectionReceived<>, RendezvousDone<>
(**) Closed<>, ConnectionError<>
```

Figure 1: Connection State Diagram

The interface provides the following guarantees about the ordering of operations:

- Sent<> events will occur on a Connection in the order in which the Messages were sent (i.e., delivered to the kernel or to the network interface, depending on implementation).

- Received<> will never occur on a Connection before it is Established; i.e. before a Ready<> event on that Connection, or a ConnectionReceived<> or RendezvousDone<> containing that Connection.

- No events will occur on a Connection after it is Closed; i.e., after a Closed<> event, an InitiateError<> or ConnectionError<> on that connection. To ensure this ordering, Closed<> will not occur on a Connection while other events on the Connection are still locally outstanding (i.e., known to the interface and waiting to be dealt with by the application). ConnectionError<> may occur after Closed<>, but the interface must gracefully handle all cases where application ignores these errors.

11. IANA Considerations

RFC-EDITOR: Please remove this section before publication.

This document has no Actions for IANA. Later versions of this document may create IANA registries for generic transport property names and transport property namespaces (see Section 4.2.1).

12. Security Considerations

This document describes a generic API for interacting with a transport services (TAPS) system. Part of this API includes configuration details for transport security protocols, as discussed in Section 5.3. It does not recommend use (or disuse) of specific algorithms or protocols. Any API-compatible transport security protocol should work in a TAPS system. Security consideration for these protocols should be discussed in the respective specifications.

The desribed API is used to exchange information between an application and the transport system. While it is not necessarily expected that both systems are implemented by the same authority, it is expected that the transport system implementation is either provided as a library that is selected by the application from a trusted party, or that it is part of the operating system that the application also relies on for other tasks.

In either case, the TAPS API is an internal interface that is used to change information locally between two systems. However, as the transport system is responsible for network communication, it is in the position to potentially share any information provided by the application with the network or another communication peer. Most of the information provided over the TAPS API are useful to configure and select protocols and paths and are not necessarily privacy sensitive. Still, there is some information that could be privacy sensitve because this might reveal usage characteristics and habits of the user of an application.

Of course any communication over a network reveals usage characteristics, as all packets as well as their timing and size are part of the network-visible wire image [RFC8546]. However, the selection of a protocol and its configuration also impacts which information is visible, potentially in clear text, and which other enties can access it. In most cases information that is provided for protocol and path selection should not directly translate to information that is can be observed by network devices on the path. But there might be specific configuration information that are intended for path exposure, such as e.g. a DiffServ codepoint setting, that is either povided directly by the application or indirectly configured over a traffic profile.

Further, applications should be aware that communication attempts can lead to more than one connection establishment. This is for example the case when the transport system also excecutes name resolution; or when support mechanisms such as TURN or ICE are used to establish connectivity; or if protocols or paths are raised; or if a path fails and fallback or re-establishment is supported in the transport system.

These communication activities are not different from what is used today, however, the goal of a TAPS transport system is to support such mechanisms as a generic service within the transport layer. This enables applications to more dynamically benefit from innovations and new protocols in the transport system but at the same time may reduce transparency of the underlying communication actions to the application itself. The TAPS API is designed such that protocol and path selection can be limited to a small and controlled set if required by the application for functional or security purposes. Further, TAPS implementations should provide an interface to poll information about which protocol and path is currently in use as well as provide logging about the communication events of each connection.

13. Acknowledgements

This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreements No. 644334 (NEAT) and No. 688421 (MAMI).

This work has been supported by Leibniz Prize project funds of DFG - German Research Foundation: Gottfried Wilhelm Leibniz-Preis 2011 (FKZ FE 570/4-1).

This work has been supported by the UK Engineering and Physical Sciences Research Council under grant EP/R04144X/1.

This work has been supported by the Research Council of Norway under its "Toppforsk" programme through the "OCARINA" project.

Thanks to Stuart Cheshire, Josh Graessley, David Schinazi, and Eric Kinnear for their implementation and design efforts, including Happy Eyeballs, that heavily influenced this work. Thanks to Laurent Chuat and Jason Lee for initial work on the Post Sockets interface, from which this work has evolved. Thanks to Maximilian Franke for asking good questions based on implementation experience and for contributing text, e.g., on multicast.

14. References

14.1. Normative References

[I-D.ietf-taps-arch]
Pauly, T., Trammell, B., Brunstrom, A., Fairhurst, G., Perkins, C., Tiesel, P., and C. Wood, "An Architecture for Transport Services", draft-ietf-taps-arch-08 (work in progress), July 2020.

[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>.

[RFC4941] Narten, T., Draves, R., and S. Krishnan, "Privacy Extensions for Stateless Address Autoconfiguration in IPv6", RFC 4941, DOI 10.17487/RFC4941, September 2007, <https://www.rfc-editor.org/info/rfc4941>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[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>.

[RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <https://www.rfc-editor.org/info/rfc8446>.

14.2. Informative References

[I-D.ietf-taps-impl]
Brunstrom, A., Pauly, T., Enghardt, T., Grinnemo, K., Jones, T., Tiesel, P., Perkins, C., and M. Welzl, "Implementing Interfaces to Transport Services", draft-ietf-taps-impl-07 (work in progress), July 2020.

[I-D.ietf-taps-minset]
Welzl, M. and S. Gjessing, "A Minimal Set of Transport Services for End Systems", draft-ietf-taps-minset-11 (work in progress), September 2018.

[I-D.ietf-taps-transport-security]
Enghardt, T., Pauly, T., Perkins, C., Rose, K., and C. Wood, "A Survey of the Interaction Between Security Protocols and Transport Services", draft-ietf-taps-transport-security-12 (work in progress), April 2020.

[I-D.ietf-tsvwg-datagram-plpmtud]
Fairhurst, G., Jones, T., Tuexen, M., Ruengeler, I., and T. Voelker, "Packetization Layer Path MTU Discovery for Datagram Transports", draft-ietf-tsvwg-datagram-plpmtud-22 (work in progress), June 2020.

[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>.

[RFC2597] Heinanen, J., Baker, F., Weiss, W., and J. Wroclawski, "Assured Forwarding PHB Group", RFC 2597, DOI 10.17487/RFC2597, June 1999, <https://www.rfc-editor.org/info/rfc2597>.

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

[RFC3246] Davie, B., Charny, A., Bennet, J., Benson, K., Le Boudec, J., Courtney, W., Davari, S., Firoiu, V., and D. Stiliadis, "An Expedited Forwarding PHB (Per-Hop Behavior)", RFC 3246, DOI 10.17487/RFC3246, March 2002, <https://www.rfc-editor.org/info/rfc3246>.

[RFC3261] Rosenberg, J., Schulzrinne, H., Camarillo, G., Johnston, A., Peterson, J., Sparks, R., Handley, M., and E. Schooler, "SIP: Session Initiation Protocol", RFC 3261, DOI 10.17487/RFC3261, June 2002, <https://www.rfc-editor.org/info/rfc3261>.

[RFC4594] Babiarz, J., Chan, K., and F. Baker, "Configuration Guidelines for DiffServ Service Classes", RFC 4594, DOI 10.17487/RFC4594, August 2006, <https://www.rfc-editor.org/info/rfc4594>.

[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>.

[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>.

[RFC5865] Baker, F., Polk, J., and M. Dolly, "A Differentiated Services Code Point (DSCP) for Capacity-Admitted Traffic", RFC 5865, DOI 10.17487/RFC5865, May 2010, <https://www.rfc-editor.org/info/rfc5865>.

[RFC7478] Holmberg, C., Hakansson, S., and G. Eriksson, "Web Real-Time Communication Use Cases and Requirements", RFC 7478, DOI 10.17487/RFC7478, March 2015, <https://www.rfc-editor.org/info/rfc7478>.

[RFC7556] Anipko, D., Ed., "Multiple Provisioning Domain Architecture", RFC 7556, DOI 10.17487/RFC7556, June 2015, <https://www.rfc-editor.org/info/rfc7556>.

[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>.

[RFC8084] Fairhurst, G., "Network Transport Circuit Breakers", BCP 208, RFC 8084, DOI 10.17487/RFC8084, March 2017, <https://www.rfc-editor.org/info/rfc8084>.

[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>.

[RFC8229] Pauly, T., Touati, S., and R. Mantha, "TCP Encapsulation of IKE and IPsec Packets", RFC 8229, DOI 10.17487/RFC8229, August 2017, <https://www.rfc-editor.org/info/rfc8229>.

[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>.

[RFC8546] Trammell, B. and M. Kuehlewind, "The Wire Image of a Network Protocol", RFC 8546, DOI 10.17487/RFC8546, April 2019, <https://www.rfc-editor.org/info/rfc8546>.

[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>.

[RFC8699] Islam, S., Welzl, M., and S. Gjessing, "Coupled Congestion Control for RTP Media", RFC 8699, DOI 10.17487/RFC8699, January 2020, <https://www.rfc-editor.org/info/rfc8699>.

[TCP-COUPLING]
"ctrlTCP: Reducing Latency through Coupled, Heterogeneous Multi-Flow TCP Congestion Control", IEEE INFOCOM Global Internet Symposium (GI) workshop (GI 2018) , April 2018.

Appendix A. Convenience Functions

A.1. Adding Preference Properties

As Selection Properties of type "Preference" will be added to a TransportProperties object quite frequently, implementations should provide special actions for adding each preference level i.e, "TransportProperties.Add(some_property, avoid)" is equivalent to "TransportProperties.Avoid(some_property)":

```
TransportProperties.Require(property)
TransportProperties.Prefer(property)
TransportProperties.Ignore(property)
TransportProperties.Avoid(property)
TransportProperties.Prohibit(property)
TransportProperties.Default(property)
```

A.2. Transport Property Profiles

To ease the use of the interface specified by this document, implementations should provide a mechanism to create Transport Property objects (see Section 5.2) that are pre-configured with frequently used sets of properties. Implementations should at least offer short-hands to specify the following property profiles:

A.2.1. reliable-inorder-stream

This profile provides reliable, in-order transport service with congestion control. An example of a protocol that provides this service is TCP. It should consist of the following properties:

Property	Value
reliability	require
preserveOrder	require
congestionControl	require
preserveMsgBoundaries	ignore

A.2.2. reliable-message

This profile provides message-preserving, reliable, in-order transport service with congestion control. An example of a protocol that provides this service is SCTP. It should consist of the following properties:

Property	Value
reliability	require
preserveOrder	require
congestionControl	require
preserveMsgBoundaries	require

A.2.3. unreliable-datagram

This profile provides unreliable datagram transport service. An example of a protocol that provides this service is UDP. It should consist of the following properties:

Property	Value
reliability	ignore
preserveOrder	ignore
congestionControl	ignore
preserveMsgBoundaries	require
safely replayable	true

Applications that choose this Transport Property Profile for latency reasons should also consider setting the Capacity Profile Property, see Section 7.1.6 accordingly and my benefit from controlling checksum coverage, see Section 5.2.7 and Section 5.2.8.

Appendix B. Relationship to the Minimal Set of Transport Services for End Systems

[I-D.ietf-taps-minset] identifies a minimal set of transport services that end systems should offer. These services make all non-security-related transport features of TCP, MPTCP, UDP, UDP-Lite, SCTP and LEDBAT available that 1) require interaction with the application, and 2) do not get in the way of a possible implementation over TCP (or, with limitations, UDP). The following text explains how this minimal set is reflected in the present API. For brevity, it is based on the list in Section 4.1 of [I-D.ietf-taps-minset], updated according to the discussion in Section 5 of [I-D.ietf-taps-minset]. This list is a subset of the transport features in Appendix A of [I-D.ietf-taps-minset], which refers to the primitives in "pass 2" (Section 4) of [RFC8303] for further details on the implementation with TCP, MPTCP, UDP, UDP-Lite, SCTP and LEDBAT.

- Connect: "Initiate" Action (Section 6.1).

- Listen: "Listen" Action (Section 6.2).

- Specify number of attempts and/or timeout for the first establishment message: "timeout" parameter of "Initiate" (Section 6.1) or "InitiateWithSend" Action (Section 8.2.6).

- Disable MPTCP: "Parallel Use of Multiple Paths" Property (Section 5.2.13).

- Hand over a message to reliably transfer (possibly multiple times) before connection establishment: "InitiateWithSend" Action (Section 8.2.6).

- Change timeout for aborting connection (using retransmit limit or time value): "Timeout for Aborting Connection" property, using a time value (Section 7.1.4).

- Timeout event when data could not be delivered for too long: "ConnectionError" Event (Section 9).

- Suggest timeout to the peer: "TCP-specific Property: User Timeout" (Section 7.2).

- Notification of Excessive Retransmissions (early warning below abortion threshold): "Notification of excessive retransmissions" property (Section 5.2.16).

- Notification of ICMP error message arrival: "Notification of ICMP soft error message arrival" property (Section 5.2.17).

- Choose a scheduler to operate between streams of an association: "Connection Group Transmission Scheduler" property (Section 7.1.5).

- Configure priority or weight for a scheduler: "Priority (Connection)" property (Section 7.1.3).

- "Specify checksum coverage used by the sender" and "Disable checksum when sending": "Corruption Protection Length" property (Section 8.1.3.6) and "Full Checksum Coverage on Sending" property (Section 5.2.7).

- "Specify minimum checksum coverage required by receiver" and "Disable checksum requirement when receiving": "Required Minimum Corruption Protection Coverage for Receiving" property (Section 7.1.2) and "Full Checksum Coverage on Receiving" property (Section 5.2.8).

- "Specify DF" field and "Request not to bundle messages": the "No Fragmentation" Message Property combines both of these requests, i.e. if a request not to bundle messages is made, this also turns off fragmentation (i.e., sets DF=1) in the case of a protocol that allows this (only UDP and UDP-Lite, which cannot bundle messages anyway) (Section 8.1.3.9).

- Get max. transport-message size that may be sent using a non-fragmented IP packet from the configured interface: "Maximum Message Size Before Fragmentation or Segmentation" property (Section 7.1.9.2).

- Get max. transport-message size that may be received from the configured interface: "Maximum Message Size on Receive" property (Section 7.1.9.4).

- Obtain ECN field: "ECN" is a defined UDP(-Lite)-specific read-only Message Property of the MessageContext object (Section 8.3.3.1).

- "Specify DSCP field", "Disable Nagle algorithm", "Enable and configure a "Low Extra Delay Background Transfer"": as suggested in Section 5.5 of [I-D.ietf-taps-minset], these transport features are collectively offered via the "Capacity Profile" property (Section 7.1.6). Per-Message control is offered via the "Message Capacity Profile Override" property (Section 8.1.3.8).

- Close after reliably delivering all remaining data, causing an event informing the application on the other side: this is offered by the "Close" Action with slightly changed semantics in line with the discussion in Section 5.2 of [I-D.ietf-taps-minset] (Section 9).

- "Abort without delivering remaining data, causing an event informing the application on the other side" and "Abort without delivering remaining data, not causing an event informing the application on the other side": this is offered by the "Abort" action without promising that this is signaled to the other side.

If it is, a "ConnectionError" Event will fire at the peer (Section 9).

- "Reliably transfer data, with congestion control", "Reliably transfer a message, with congestion control" and "Unreliably transfer a message": data is transferred via the "Send" action (Section 8.2). Reliability is controlled via the "Reliable Data Transfer (Connection)" (Section 5.2.1) property and the "Reliable Data Transfer (Message)" Message Property (Section 8.1.3.7). Transmitting data as a message or without delimiters is controlled via Message Framers (Section 8.1.2). The choice of congestion control is provided via the "Congestion control" property (Section 5.2.9).

- Configurable Message Reliability: the "Lifetime" Message Property implements a time-based way to configure message reliability (Section 8.1.3.1).

- "Ordered message delivery (potentially slower than unordered)" and "Unordered message delivery (potentially faster than ordered)": these two transport features are controlled via the Message Property "Ordered" (Section 8.1.3.3).

- Request not to delay the acknowledgement (SACK) of a message: should the protocol support it, this is one of the transport features the transport system can apply when an application uses the "Capacity Profile" Property (Section 7.1.6) or the "Message Capacity Profile Override" Message Property (Section 8.1.3.8) with value "Low Latency/Interactive".

- Receive data (with no message delimiting): "Received" Event (Section 8.3.2.1). See Section 8.1.2 for handling Message framing in situations where the Protocol Stack only provides a byte-stream transport.

- Receive a message: "Received" Event (Section 8.3.2.1), using Message Framers (Section 8.1.2).

- Information about partial message arrival: "ReceivedPartial" Event (Section 8.3.2.2).

- Notification of send failures: "Expired" Event (Section 8.2.3.2) and "SendError" Event (Section 8.2.3.3).

- Notification that the stack has no more user data to send: applications can obtain this information via the "Sent" Event (Section 8.2.3.1).

o Notification to a receiver that a partial message delivery has been aborted: "ReceiveError" Event (Section 8.3.2.3).

Authors' Addresses

Brian Trammell (editor)
Google Switzerland GmbH
Gustav-Gull-Platz 1
8004 Zurich
Switzerland

Email: ietf@trammell.ch

Michael Welzl (editor)
University of Oslo
PO Box 1080 Blindern
0316 Oslo
Norway

Email: michawe@ifi.uio.no

Theresa Enghardt
Netflix
121 Albright Way
Los Gatos, CA 95032
United States of America

Email: ietf@tenghardt.net

Godred Fairhurst
University of Aberdeen
Fraser Noble Building
Aberdeen, AB24 3UE
Scotland

Email: gorry@erg.abdn.ac.uk
URI: http://www.erg.abdn.ac.uk/

Mirja Kuehlewind
Ericsson
Ericsson-Allee 1
Herzogenrath
Germany

Email: mirja.kuehlewind@ericsson.com

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

Email: csp@csperkins.org

Philipp S. Tiesel
TU Berlin
Einsteinufer 25
10587 Berlin
Germany

Email: philipp@tiesel.net

Christopher A. Wood
Cloudflare
101 Townsend St
San Francisco
United States of America

Email: caw@heapingbits.net

Tommy Pauly
Apple Inc.
One Apple Park Way
Cupertino, California 95014
United States of America

Email: tpauly@apple.com