TAPS Working Group
Internet-Draft
Intended status: Informational
Expires: September 6, 2018

A. Brunstrom, Ed.
Karlstad University
T. Pauly, Ed.
Apple Inc.
T. Enghardt
TU Berlin
K-J. Grinnemo
Karlstad University
T. Jones
University of Aberdeen
P. Tiesel
TU Berlin
C. Perkins
University of Glasgow
M. Welzl
University of Oslo
March 05, 2018

Implementing Interfaces to Transport Services
draft-brunstrom-taps-impl-00

Abstract

The Transport Services architecture [I-D.pauly-taps-arch] defines a system that allows applications to use transport networking protocols flexibly. 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.pauly-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.trammell-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.pauly-taps-arch].

2. Implementing Basic Objects

The basic 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 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 4.8.

3. Implementing Pre-Establishment

During pre-establishment the application specifies the Endpoints to be used for communication as well as its preferences regarding Protocol and Path Selection. The implementation stores these objects and properties as part of the Preconnection object for use during connection establishment. For Protocol and Path Selection Properties that are not provided by the application, the implementation must use the default values specified in the Transport Services API ([I-D.trammell-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 Parameters, the transport system should match the required and prohibited properties against the transport features of the available protocols.

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

- The application requested 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, e.g., because SCTP is not supported by the operating system, this should result in an error.

- The application requested 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.

It is important to fail as early as possible in such cases in order to avoid allocating resources, 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 has a close connection 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 such as Local Endpoint, Remote Endpoint, Path Selection Properties, and Protocol 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. Any 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 in order to select which set to use.

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.

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 Path Selection Properties and Protocol Selection Properties.

4.1.1. 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]
```

4.1.2. 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.

4.1.2.1. 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 should order 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 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. These hostnames should each be treated as a branch which can be attempted independently from other hostnames. Each of these hostnames may also resolve to one or more addresses, thus creating 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]
```

4.1.2.2. Alternate Paths

If a client has multiple network interfaces available to it, such as mobile client with both Wi-Fi and Cellular connectivity, it can attempt a connection over either interface. This represents a branch point in the connection establishment. Like with derived endpoints, 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.

4.1.2.3. 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 may be allowed to use an HTTP-based proxy, a SOCKS-based proxy, or connect directly. 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 may provide preferable characteristics when available. 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 basis (see Section 8.2). This information can influence future racing decisions to prioritize or prune branches.

4.2. 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 usable DNS results on one network may not necessarily be the same as DNS results on another network due to local network entities, supported address families, or enterprise network configurations. 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 checked next 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.

4.3. Sorting Branches

Implementations should sort the branches of the tree of connection options in order of their preference rank. 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 Protocol and Path Selection Properties, which are specified in [I-D.trammell-taps-interface]. In addition to the properties provided by the application, an implementation may include additional criteria such as cached performance estimates, see Section 8.2, or system policy, see Section 3.2, in the ranking. Two examples of how the Protocol and Path Selection Properties may be used to sort branches are provided below:

- o Interface Type: If the application specifies an interface type to be preferred or avoided, implementations should rank paths accordingly. If the application specifies an interface type to be required or prohibited, we expect an implementation to not include the non-conforming paths into the three.

- o Capacity Profile: An implementation may use the Capacity Profile to prefer paths optimized for the application's expected traffic pattern according to cached performance estimates, see Section 8.2:

  - * Interactive/Low Latency: Prefer paths with the lowest expected Round Trip Time

  - * Constant Rate: Prefer paths that can satisfy the requested Stream Send or Stream Receive Bitrate, based on observed maximum throughput

* Scavenger/Bulk: Prefer paths with the highest expected available bandwidth, based on observed maximum throughput

[Note: See Appendix A.1 for additional examples related to Properties under discussion.]

4.4. 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. While the tree described above is a useful conceptual and architectural model, an implementation does not know what the full tree may become up front, nor will many of the possible branches be used in the common case.

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.

4.4.1. Delayed Racing

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.

4.4.2. 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.

4.5. 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.

It is useful to process success and failure throughout the tree by child nodes reporting to their 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 should not be started after another leaf node has completed successfully, as the connection as a whole has been established. An implementation may choose to let certain handshakes and negotiations complete in order to gather metrics to influence future connections. Similarly, an implementation may choose to hold onto fully established leaf nodes that were not the first to establish for use in future connections, but this approach is not recommended since those attempts were slower to connect and may exhibit less desirable properties.

4.5.1. 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 4.7.

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 Parameters 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.

4.6. 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.

4.7. 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. In the case that the application signals that the initial Protocol Stack is failing for some reason and that another option should be attempted, the Connection can be updated to point to the next candidate Protocol Stack. This can be viewed as an application-driven form of Protocol Stack racing.

4.8. 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 either use an ephemeral port or generate an error.

If the Path 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 Path Selection 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 Protocol Selection Properties allow multiple protocols to be used for listening, and the implementation supports it, the Listener object should register across the eligble protocols for each path. This means that inbound Connections delivered by the implementation may have heterogeneous protocol stacks.

4.8.1. 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 well 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.

4.8.2. 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.

4.8.3. 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 Data Transfer

5.1. Data transfer for streams, datagrams, and frames

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 delimeters) 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.

5.1.1. 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.

5.1.1.1. Send Parameters

- Lifetime: this should be implemented by removing the Message from its 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 de-queueing a message. For example, TCP cannot remove bytes from its send buffer, while in case of SCTP, such control over the SCTP send buffer can be exercised using the partial reliability extension [RFC8303]. When there is no standing queue of Messages within the system, and the Protocol Stack does not support removing a Message from its buffer, this property may be ignored.

- Niceness: this represents the ability to de-prioritize a Message in favor of 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 niceness on streams of different priority.

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

- Idempotent: when this is true, it 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.

- Corruption Protection Length: when this is set to any value other than -1, it limits the required checksum in protocols that allow limiting the checksum length (e.g. UDP-Lite).

- Immediate Acknowledgement: this informs the implementation that the sender intends to execute tight control over the send buffer, and therefore wants to avoid delayed acknowledgements. In case of SCTP, a request to immediately send acknowledgements can be implemented using the "sack-immediately flag" described in Section 4.2 of [RFC8303] for the SEND.SCTP primitive.

- Instantaneous Capacity Profile: when this is set to "Interactive/ Low Latency", the Message should be sent immediately, even when this comes at the cost of using the network capacity less efficiently. For example, small messages can sometimes be bundled to fit into a single data packet for the sake of reducing header overhead; such bundling should not be used. For example, in case of TCP, the Nagle algorithm should be disabled when Interactive/ Low Latency is selected as the capacity profile. Scavenger/Bulk can translate into usage of a congestion control mechanism such as LEDBAT, and/or the capacity profile can lead to a choice of a DSCP value as described in [I-D.ietf-taps-minset]).

[Note: See also Appendix A.2 for additional Send Parameters under discussion.]

5.1.1.2. 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.

5.1.2. 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, on the other hand, the top-level protocol only supports a byte-stream and no deframers were supported, the application must specify the minimum number of bytes of Message content it wants to receive (which may be just a single byte) to control the flow of received data.

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.

5.2. 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 [I-D.ietf-tls-tls13]. 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 Idempotent 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 racing means that different leaf nodes have the opportunity to send the same data independently. If data is truly idempotent, 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 idempotent 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. If implementations have multiple tickets available from a previous connection, each leaf node attempt must 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 Maintenance

Maintenance encompasses changes that the application can request to a Connection, or that a Connection can react to based on system and network changes.

6.1. Changing Protocol Properties

Appendix A.1 of [I-D.ietf-taps-minset] explains, using primitives that are described in [RFC8303] and [RFC8304], how to implement changing the following protocol properties of an established connection with TCP and UDP. Below, we amend this description for other protocols (if applicable):

- o Relative niceness: for SCTP, this can be done using the primitive CONFIGURE_STREAM_SCHEDULER.SCTP described in section 4 of [RFC8303].

- o Timeout for aborting Connection: for SCTP, this can be done using the primitive CHANGE_TIMEOUT.SCTP described in section 4 of [RFC8303].

- o Abort timeout to suggest to the Remote Endpoint: for TCP, this can be done using the primitive CHANGE_TIMEOUT.TCP described in section 4 of [RFC8303].

- o Retransmission threshold before excessive retransmission notification: for TCP, this can be done using ERROR.TCP described in section 4 of [RFC8303].

- o Required minimum coverage of the checksum for receiving: for UDP-Lite, this can be done using the primitive SET_MIN_CHECKSUM_COVERAGE.UDP-Lite described in section 4 of [RFC8303].

- Connection group transmission scheduler: for SCTP, this can be done using the primitive SET_STREAM_SCHEDULER.SCTP described in section 4 of [RFC8303].

It may happen that the application attempts to set a Protocol Property which does not apply to the actually chosen protocol. In this case, the implementation should fail gracefully, i.e., it may give a warning to the application, but it should not terminate the Connection.

6.2. 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 Path 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. 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.

7. Implementing 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 Tranport System before calling Abort.

As explained in section Section 4.6, 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].

8. 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.

8.1. 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:

- o 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.

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

- o 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.

8.2. 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:

- o Observed Round Trip Time

- o Connection Establishment latency

- o 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, 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.

9. Specific Transport Protocol Considerations

9.1. TCP

Connection lifetime for TCP translates fairly simply into the the abstraction presented to an application. When the TCP three-way handshake is complete, its layer of the Protocol Stack can be considered Ready (established). This event will cause racing of Protocol Stack options to complete if TCP is the top-level protocol, at which point the application can be notified that the Connection is Ready to send and receive.

If the application sends a Close, that can translate to a graceful termination of the TCP connection, which is performed by sending a FIN to the remote endpoint. If the application sends an Abort, then the TCP state can be closed abruptly, leading to a RST being sent to the peer.

Without a layer of framing (a top-level protocol in the established Protocol Stack that preserves message boundaries, or an application-supplied deframer) on top of TCP, the receiver side of the transport system implementation can only treat the incoming stream of bytes as a single Message, terminated by a FIN when the Remote Endpoint closes the Connection.

9.2. UDP

UDP as a direct transport does not provide any handshake or connectivity state, so the notion of the transport protocol becoming Ready or established is degenerate. Once the system has validated that there is a route on which to send and receive UDP datagrams, the protocol is considered Ready. Similarly, a Close or Abort has no meaning to the on-the-wire protocol, but simply leads to the local state being torn down.

When sending and receiving messages over UDP, each Message should correspond to a single UDP datagram. The Message can contain metadata about the packet, such as the ECN bits applied to the packet.

9.3. SCTP

To support sender-side stream schedulers (which are implemented on the sender side), a receiver-side Transport System should always support message interleaving [RFC8260].

SCTP messages can be very large. To allow the reception of large messages in pieces, a "partial flag" can be used to inform a (native SCTP) receiving application that a message is incomplete. After receiving the "partial flag", this application would know that the next receive calls will only deliver remaining parts of the same message (i.e., no messages or partial messages will arrive on other streams until the message is complete) (see Section 8.1.20 in [RFC6458]). The "partial flag" can therefore facilitate the implementation of the receiver buffer in the receiving application, at the cost of limiting multiplexing and temporarily creating head-of-line blocking delay at the receiver.

When a Transport System transfers a Message, it seems natural to map the Message object to SCTP messages in order to support properties such as "Ordered" or "Lifetime" (which maps onto partially reliable delivery with a SCTP_PR_SCTP_TTL policy [RFC6458]). However, since multiplexing of Connections onto SCTP streams may happen, and would be hidden from the application, the Transport System requires a per-stream receiver buffer anyway, so this potential benefit is lost and the "partial flag" becomes unnecessary for the system.

The problem of long messages either requiring large receiver-side buffers or getting in the way of multiplexing is addressed by message interleaving [RFC8260], which is yet another reason why a receivers-side transport system supporting SCTP should implement this mechanism.

9.4. TLS

The mapping of a TLS stream abstraction into the application is equivalent to the contract provided by TCP (see Section 9.1). The Ready state should be determined by the completion of the TLS handshake, which involves potentially several more round trips beyond the TCP handshake. The application should not be notified that the Connection is Ready until TLS is established.

9.5. 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.

9.6. 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 (possibly after some timeout), or after an individual stream Connection sends an Abort.

Messages over a direct QUIC stream should be represented similarly to the TCP stream (one Message per direction, see Section 9.1), unless a framing mapping is used on top of QUIC.

9.7. HTTP/2 transport

Similar to QUIC (Section 9.6), 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 9.1).

10. Rendezvous and Environment Discovery

The connection establishment process outlined in Section 4 is appropriate for client-server connections, but needs to be expanded in peer-to-peer Rendezvous scenarios, as follows:

- Gathering 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 [RFC5245]. The endpoint determines its server reflexive Local Endpoints (i.e., the translated address of a local, on the other side of a NAT) and relayed locals (e.g., via a TURN server or other relay), for each interface and network protocol. These are added to the set of candidate Local Endpoints for this connection.

  Gathering locals is primarily an endpoint local operation, although it might involve exchanges with a STUN server to derive server reflexive locals, or with a TURN server or other relay to derive relayed locals. It does not involve communication with the Remote Endpoint.

- Gathering 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.

o Establishing Connections

The set of candidate Local Endpoints and the set of candidate Remote Endpoints are paired, to derive a priority ordered set of Candidate Paths that can potentially be used to establish a Connection.

Then, communication is attempted over each candidate path, in priority order. If there are multiple candidates with the same priority, then connection establishment proceeds simultaneously and uses the transient that wins the race to be established. Otherwise, connection establishment is sequential, paced at a rate that should not congest the network. Depending on the chosen transport, this phase might involve racing TCP connections to a server over IPv4 and IPv6 [RFC8305], or it could involve a STUN exchange to establish peer-to-peer UDP connectivity [RFC5245], or some other means.

o Confirming and Maintaining Connections

Once connectivity has been established, unused resources can be released and the chosen path can be confirmed. This is primarily required when establishing peer-to-peer connectivity, where connections supporting relayed locals that were not required can be closed, and where an associated signalling operation might be needed to inform middleboxes and proxies of the chosen path. Keep-alive messages may also be sent, as appropriate, to ensure NAT and firewall state is maintained, so the Connection remains operational.

To support ICE, or similar protocols, that involve an out-of-band indirect signalling exchange 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.

(TO-DO: It is expected that a single abstract algorithm can be identified that supports both the peer-to-peer and client-server connection racing, allowing this text to be merged with Section 4)

11. IANA Considerations

RFC-EDITOR: Please remove this section before publication.

This document has no actions for IANA.

12. Security Considerations

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 5.2 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.

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-minset]
Welzl, M. and S. Gjessing, "A Minimal Set of Transport Services for TAPS Systems", draft-ietf-taps-minset-02 (work in progress), February 2018.

[I-D.pauly-taps-arch]
Pauly, T., Trammell, B., Brunstrom, A., Fairhurst, G., Perkins, C., Tiesel, P., and C. Wood, "An Architecture for Transport Services", draft-pauly-taps-arch-00 (work in progress), February 2018.

[I-D.trammell-taps-interface]
Trammell, B., Welzl, M., Enghardt, T., Fairhurst, G., Kuehlewind, M., Perkins, C., Tiesel, P., and C. Wood, "An Abstract Application Layer Interface to Transport Services", draft-trammell-taps-interface-00 (work in progress), March 2018.

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

[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>.

14.2. Informative References

[I-D.ietf-quic-transport] Iyengar, J. and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport", draft-ietf-quic-transport-10 (work in progress), March 2018.

[I-D.ietf-tls-tls13] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", draft-ietf-tls-tls13-26 (work in progress), March 2018.

[NEAT-flow-mapping] Weinrank, F. and M. Tuexen, "Transparent Flow Mapping for NEAT (in Workshop on Future of Internet Transport (FIT 2017))", June 2017.

[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>.

[Trickle] Ghobadi, M., Cheng, Y., Jain, A. and M. Mathis, "Trickle - Rate Limiting YouTube Video Streaming (ATC 2012)", June 2012.

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 4.3, the following properties under discussion can influence branch sorting:

- Size to be Sent or Received: An implementation may use the Size to be Sent or Received in combination with cached performance estimates, see Section 8.2, e.g. the observed Round Trip Time and the observed maximum throughput, to compute an estimate of the completion time of a transfer over different available paths. It may then prefer the path with the shorter expected completion time. This property may be used instead of the Capacity profile, as the application does not always know whether its transfer will be latency-bound or bandwidth-bound, and thus may not be able to specify a Capacity Profile. However, the application may know the Size to be Sent or Received from metadata, e.g., in adaptive HTTP streaming such as MPEG-DASH, or in operating system upgrades. A related paper is currently under submission.

- Send / Receive Bitrate: If the application indicates an 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 8.2. 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.

A.2. Send Parameters

In addition to the Send Parameters listed in Section 5.1.1.1, the following Send Parameters are under discussion:

- Send Bitrate: If an application indicates a certain bitrate it wants to send on the connection, the implementation may limit the bitrate of the outgoing communication to that rate, for example by setting an upper bound for the TCP congestion window of a connection calculated from the Send Bitrate and the Round Trip Time. This helps to avoid bursty traffic patterns on video streaming servers, see [Trickle].

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
TU Berlin
Marchstrasse 23
10587 Berlin
Germany

Email: theresa@inet.tu-berlin.de

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
UK

Email: tom@erg.abdn.ac.uk

Philipp S. Tiesel
TU Berlin
Marchstrasse 23
10587 Berlin
Germany

Email: philipp@inet.tu-berlin.de

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