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

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

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

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

This Internet-Draft will expire on September 6, 2018.

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.

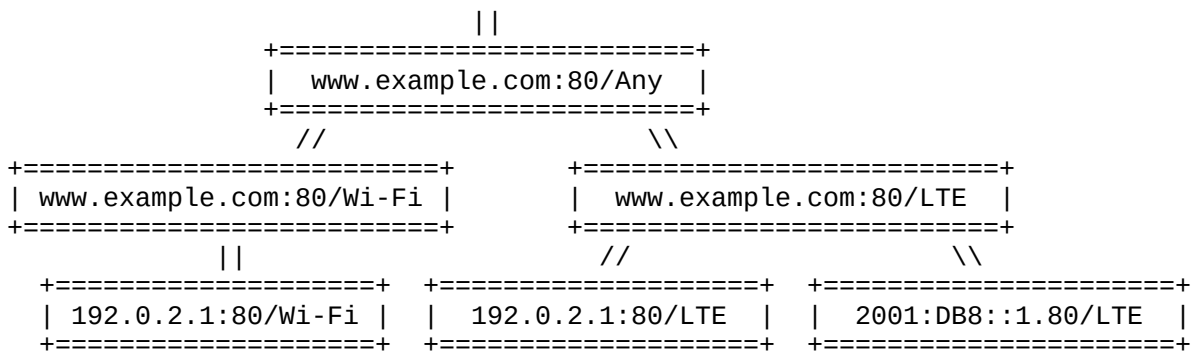
4.1. Candidate Gathering

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:



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:

- 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.

- 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):

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

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

- 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].

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

- 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].

- o 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:

- 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.

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:

- 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, 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.

- 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.

- 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

Transport Services (taps) Working Group
Internet-Draft
Intended status: Informational
Expires: August 31, 2018

F. Gont
SI6 Networks / UTN-FRH
G. Gont
SI6 Networks
M. Garcia Corbo
SITRANS
C. Huitema
Private Octopus Inc.

February 27, 2018

Problem Statement Regarding IPv6 Address Usage
draft-gont-taps-address-usage-problem-statement-00

Abstract

This document analyzes the security and privacy implications of IPv6 addresses based on a number of properties (such as address scope, stability, and usage type), and identifies gaps that currently prevent systems and applications from leveraging the increased flexibility and availability of IPv6 addresses.

Status of This Memo

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

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

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

This Internet-Draft will expire on August 31, 2018.

Copyright Notice

Table of Contents

1. Introduction

IPv6 addresses may differ in a number of properties, such as address scope (e.g. link-local vs. global), stability (e.g. stable addresses vs. temporary addresses), and intended usage type (outgoing communications vs. incomming communications). While often overlooked, these properties have impact on areas such as security, privacy, and performance.

IPv6 hosts typically configure a number of IPv6 addresses of different properties. For example, a host may configure one stable

and one temporary address per each autoconfiguration prefix advertised on the local network. Currently, the addresses to be configured typically depend on local system policy, with the aforementioned policy being static and irrespective of the network the host attaches to. This "one size fits all" approach limits the ability of systems and applications of fully-leveraging the increased flexibility and availability of IPv6 addresses.

Each application running on a given system may have its own set of requirements or expectations for the properties of the IPv6 addresses to be employed. For example, an application meaning to offer a public service might expect to employ global stable addresses for such purpose, while a privacy-sensible client application might prefer short-lived temporary addresses, or might even expect to employ single-use ("throw-away") IPv6 addresses when connecting to public servers. However, the subtetlies associated with IPv6 addresses (and associated properties) are often ignored by application programmers and, in any case, current APIs (such as the BSD Sockets API) tend to be very limited in the amount of control they give applications to select the most appropriate IPv6 addresses for a given task, thus limiting a programmer’s ability to leverage IPv6 address availability and properties.

This document analyzes the impact of a number of properties of IPv6 addresses on areas such as security and privacy, and analyzes how IPv6 addresses are curently generated and employed by different operating systems and applications. Finally, it provides a problem statement by identifying and analyzing gaps that prevent systems and applications from fully-leveraging IPv6 addressing capabilities, setting the basis for further work that could fill those gaps.

2. Disclaimer/Notes

This document is a verbatim copy of [I-D.gont-6man-address-usage-recommendations], modulo minor changes. The aforementioned document was targeted at the 6man working group, and thus this document focuses only on IPv6 addresses. If this document is deemed of interest to the TAPS working group, it might be expanded to discuss properties of IPv4 addresses.

3. Terminology

This document employs the definitions of "public address", "stable address", and "temporary address" from Section 2 of [RFC7721].

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

4. Background

Predictable IPv6 addresses result in a number of security and privacy implications. For example, [Barnes2012] discusses how patterns in network prefixes can be leveraged for IPv6 address scanning. On the other hand, [RFC7707], [RFC7721] and [RFC7217] discuss the security and privacy implications of predictable IPv6 Interface Identifiers (IIDs).

Given the aforementioned previous work in this area, and the formal specification update produced by [RFC8064], we expect (and assume in the rest of this document) that implementations have replaced any schemes that produce predictable addresses with alternative schemes that avoid such patterns (e.g., RFC7217 in replacement of the traditional SLAAC addresses that embed link-layer addresses).

5. IPv6 Address Properties

There are three parameters that affect the security and privacy properties of an IPv6 address:

- Scope
- Stability
- Usage type (client-like "outgoing connections" vs. server-like "incoming connections")

Section 5.1, Section 5.2, and Section 5.3 discuss the security and privacy implications (and associated tradeoffs) of the scope, stability and usage type properties of IPv6 addresses, respectively.

5.1. Address Scope Considerations

The IPv6 address scope can, in some scenarios, limit the attack exposure of a node as a result of the implicit isolation provided by a non-global address scope. For example, a node that only employs link-local addresses may, in principle, only be exposed to attack from other nodes in the local link. Hosts employing only Unique Local Addresses (ULAs) may be more isolated from attack than those employing Global Unicast Addresses (GUAs), assuming that proper packet filtering is enforced at the network edge.

The potential protection provided by a non-global addresses should not be regarded as a complete security strategy, but rather as a form of "prophylactic" security (see [I-D.gont-opsawg-firewalls-analysis]).

We note that the use of non-global addresses is usually limited to a reduced type of applications/protocols that e.g. are only meant to operate on a reduced scope, and hence their applicability may be limited.

A discussion of ULA usage considerations can be found in [I-D.ietf-v6ops-ula-usage-considerations].

5.2. Address Stability Considerations

The stability of an address has two associated security/privacy implications:

- Ability of an attacker to correlate network activity
- Exposure to attack

For obvious reasons, an address that is employed for multiple communication instances allows the aforementioned network activities to be correlated. The longer an address is employed (i.e., the more stable it is), the longer such correlation will be possible. In the worst-case scenario, a stable address that is employed for multiple communication instances over time will allow all such activities to be correlated. On the other hand, if a host were to generate (and eventually "throw away") one new address for each communication instance (e.g., TCP connection), network activity correlation would be mitigated.

NOTE:
The use of constant IIDs (as in traditional SLAAC) result in addresses that, while not constant as a whole (since the prefix changes), contain a globally-unique value that leaks out the node "identity". Such addresses result in the worst possible security and privacy implications, and their use has been deprecated by [RFC8064].

Typically, when it comes to attack exposure, the longer an address is employed the longer an attacker is exposed to attacks (e.g. an attacker has more time to find the address in the first place [RFC7707]). While such exposure is traditionally associated with the stability of the address, the usage type of the address (see Section 5.3) may also have an impact on attack exposure.

A popular approach to mitigate network activity correlation is the use of "temporary addresses" [RFC4941]. Temporary addresses are typically configured and employed along with stable addresses, with the temporary addresses employed for outgoing communications, and the stable addresses employed for incoming communications.

NOTE:
Ongoing work [I-D.gont-6man-non-stable-iids] aims at updating [RFC4941] such that temporary addresses can be employed without the need to configure stable addresses.

We note that the extent to which temporary addresses provide improved mitigation of network activity correlation and/or reduced attack exposure may be questionable and/or limited in some scenarios. For example, a temporary address that is reachable for, say, a few hours has a questionable "reduced exposure" (particularly when automated attack tools do not typically require such a long period of time to complete their task). Similarly, if network activity can be correlated for the life of such address (e.g., on the order of several hours), such period of time might be long enough for the attacker to correlate all the network activity he is meaning to correlate.

In order to better mitigate network activity correlation and/or possibly reduce host exposure, an implementation might want to either reduce the preferred lifetime of a temporary address, or even better, generate one new temporary address for each new transport protocol instance. However, the associated lifetime/stability of an address may have a negative impact on the network. For example, if a node were to employ "throw away" IPv6 addresses, or employ temporary addresses [RFC4941] with a short preferred lifetime, local nodes might need to maintain too many entries in their Neighbor Cache, and a number of devices (possibly enforcing security policies) might also need to cope with such additional state.

Additionally, enforcing a maximum lifetime on IPv6 addresses may cause long-lived TCP connections to fail. For example, an address becoming "Invalid" (after transitioning through the "Preferred" and "Deprecated" states) would cause the TCP connections employing them to break. This, in turn, would cause e.g. long-lived SSH sessions to break/fail.

In some scenarios, attack exposure may be reduced by limiting the usage of temporary addresses to outgoing connections, and prevent such addresses from being used for incoming connections (please see Section 5.3).

5.3. Usage Type Considerations

A node that employs one of its addresses to communicate with an external server (i.e., to perform an "outgoing connection") may cause such address to become exposed to attack. For example, once the external server receives an incoming connection, the corresponding server might launch an attack against the aforementioned address. A

real-world instance of this type of scenario has been documented in [Hein].

However, we note that employing an IPv6 address for outgoing communications need not increase the exposure of local services to other parties. For example, nodes could employ temporary addresses only for outgoing connections, but not for incoming connections. Thus, external nodes that learn about client's addresses could not really leverage such addresses for actively contacting the clients.

There are multiple ways in which this could possibly be achieved, with different implications. Namely:

- Run a host-based or network-based firewall
- Bind services to specific (explicit) addresses
- Bind services only to stable addresses

A client could simply run a host-based firewall that only allows incoming connections on the stable addresses. This is clearly more of an operational way of achieving the desired functionality, and may require good firewall/host integration (e.g., the firewall should be able to tell stable vs. temporary addresses), may require the client to run additional firewall software for this specific purpose, etc. In other scenarios, a network-based firewall could be configured to allow outgoing communications from all internal addresses, but only allow incoming communications to stable addresses. For obvious reasons, this is generally only applicable to networks where incoming communications are allowed to a limited number of hosts/servers.

Services could be bound to specific (explicit) addresses, rather than to all locally-configured addresses. However, there are a number of short-comings associated with this approach. Firstly, an application would need to be able to learn all of its addresses and associated stability properties, something that tends to be non-trivial and non-portable, and that also makes applications protocol-dependent, unnecessarily. Secondly, the BSD Sockets API does not really allow a socket to be bound to a subset of the node's addresses. That is, sockets can be bound to a single address or to all available addresses (wildcard), but not to a subset of all the configured addresses.

Binding services only to stable addresses provides a clean separation between addresses employed for client-like outgoing connections and server-like incoming connections. However, we currently lack an appropriate API for nodes to be able to specify that a socket should only be bound to stable addresses.

6. Default Address Selection in IPv6

Applications use system API's to select the IPv6 addresses that will be used for incoming and outgoing connections. These choices have consequences in terms of privacy, security, stability and performance.

Default Address Selection for IPv6 is specified in [RFC6724]. The selection starts with a set of potential destination addresses, such as returned by getaddrinfo(), and the set of potential source addresses currently configured for the selected interfaces. For each potential destination address, the algorithm will select the source address that provides the best route to the destination, while choosing the appropriate scope and preferring temporary addresses. The algorithm will then select the destination address, while giving a preference to reachable addresses with the smallest scope. The selection may be affected by system settings. We note that [RFC6724] only applies for outgoing connections, such as those made by clients trying to use services offered by other hosts.

We note that [RFC6724] selects IPv6 addresses from all the currently available addresses on the host, and there is currently no way for an application to indicate expected or desirable properties for the IPv6 source addresses employed for such outgoing communications. For example, a privacy-sensitive application might want that each outgoing communication instance employs a new, single-use IPv6 address, or to employ a new reusable address that is not employed or reusable by any other application on the host. Reuse of an IPv6 address by an application would allow the correlation of all network activities corresponding to such application as being performed by the same host, while reuse of an IPv6 address by multiple different applications would allow the correlation of all such network activities as being performed by the host with such IPv6 address.

When devices provide a service, the common pattern is to just wait for connections over all addresses configured on the device. For example, applications using the BSD Sockets API will commonly bind() the listening socket to the undefined address. This long-established behavior is appropriate for devices providing public services, but may have unexpected results for devices providing semi-private services, such as various forms of peer-to-peer or local-only applications.

This behavior leads to three problems: device tracking, discussed in Section 8.1.2; unexpected address discovery, discussed in Section 8.1.3; and availability outside the expected scope, discussed in Section 8.1.4. These problems are caused in part by the

limitations of available address selection API, presented in Section 8.2.

7. Current Possible Approaches for IPv6 Address Usage

7.1. Incoming communications

There are a number of ways in which a system or network may affect which address (and how) may be employed for different services and cases. Namely,

- TCP/IP stack address filtering
- Application-based address filtering
- Firewall-based address filtering

Clearly, the most elegant approach for address selection is for applications to be able to specify the properties of the addresses they are willing to employ by means of an API, such the TCP/IP stack itself can "filter" which addresses are allowed to be employed for the given service/application. This relieves the application from dealing with low level details of networking, improves portability, and avoids duplicate code in applications. However, constraints in the current APIs (see Section 8.2) may limit the ability of application progremmers for leveraging this technique.

Another possible approach is for applications to e.g. bind services to all available addresses, and perform the associated selection/ filtering at the application level. While possible this has a number of drawbacks. Firstly, it would require applications to deal with low-level networking details, require that all the associated code be duplicated in all applications, and also negatively affect portability. Besides, performing address/selection filtering at the application level may not mitigate some possible threats. For example, port scanning will still be possible, since the aforementioned filtering will only be performed e.g. once UDP packets are received or TCP connections are established.

Finally, a firewall may be employed to filter addresses based on their intended usage. For example, a firewall may block incoming requests to all addresses except to some whitelisted addresses (such as the stable addresses of the node). This technique not only requires the use of a firewall (which may or may not be present), but also implies knowledge of the firewall regarding the desired properties of the addresses that each application/service is intended to use.

7.2. Outgoing communications

An application might be able to obtain the list of currently-configured addresses, and subsequently select an address with desired properties, and explicitly "bind" the address to the socket, to override the default source address selection.

However, this approach is problematic for a number of reasons. Firstly, there is no portable way of obtaining the list of currently-configured addresses on the local node, and even less to check for properties such "valid lifetime". Secondly, as discussed in Section 7.1, it would require application programmers to understand all the subtetiles associated with IPv6 addressing, and would also lead to duplicate code on all applications. Finally, applications would be limited to use already-configured addresses and unable to trigger the generation of new addresses where desirable (e.g. the genration of a new temporary address for this application instance or communication instance).

8. Problem Statement

This section elaborates the problem statement on IPv6 address usage. Section 8.1 describes the security and privacy implications of improper IPv6 address usage, while Section 8.2, Section 8.4, Section 8.3, analyze the possible root of such improper address usage, suggesting possible future work.

8.1. Issues Associated with Sub-optimal IPv6 Address Usage

8.1.1. Correlation of Network Activity

As discussed in [RFC7721], a node that reuses an IPv6 address for multiple communication instances would allow the correlation of such network activities. This could be the case when the same IPv6 address is employed by several instances of the same application (e.g., a browser in "privacy" mode and a browser in "normal" mode), or when the same IPv6 address is employed by two different applications on the same node (e.g., a browser in "privacy" mode, and an email client).

Particularly for privacy-sensitive applications, an application or system might want to limit the usage of a given IPv6 address to a single communication instance, a single application, a single user on the system, etc. However, given current APIs, this is practically impossible.

8.1.2. Testing for the Presence of Node in the Network

The stable addresses recommended in [RFC8064] use stable IIDs defined in [RFC7217]. One key part of that algorithm is that if a device connects to a given network at different times, it will always configure the same IPv6 addresses on that network. If the device hosts a service ready to accept connections on that stable address, adversaries can test the presence of the device on the network by attempting connections to that stable address. Stable addresses used by listening services will thus enable testing whether a specific device is returning to a particular network, which in a number of cases might be considered a privacy issue.

8.1.3. Unexpected Address Discovery

Systems like DNS-Based Service Discovery [RFC6763] allow clients to discover services within a limited scope, that can be defined by a domain name. These services are not advertised outside of that scope, and thus do not expect to be discovered by random parties on the Internet. However, such services may be easily discoverable if they listen for connections to IPv6 addresses that a client process also uses as source address when connecting to remote servers.

NOTE:
An example of such unexpected discovery is described in [Hein]. A network manager observed scanning traffic directed at the temporary addresses of local devices. The analysis in [Hein] shows that the scanners learned the addresses by observing the device contact an NTP service ([RFC5905]). The remote scanning was possible because the local devices were also accepting connections directed to the temporary addresses.

It should be obvious from the example that the "attack surface" of the services is increased because they are bond to the same IPv6 addresses that are also used by clients for outgoing communications with remote systems. But the overlap between "client" and "server" addresses is only one part of the problem. Suppose that a device hosts both a video game and a home automation application. The video game users will be able to discover the IPv6 address of the game server. If the home automation server listens to the same IPv6 addresses, it is now exposed to connection attempts by all these users. That, too, increases the attack surface of the home automation server.

8.1.4. Availability Outside the Expected Scope

The IPv6 addressing architecture [RFC4291] defines multiple address scopes. In practice, devices are often configured with globally reachable unicast addresses, link local addresses, and Unique Local IPv6 Unicast Addresses (ULA) [RFC4193]. Availability outside the expected scope happens when a service is expected to be only available in some local scope, but inadvertently becomes available to remote parties. That could happen for example if a service is meant to be available only on a given link, but becomes reachable through ULA or through globally reachable addresses, or if a service is meant to be available only inside some organization's perimeter and becomes reachable through globally reachable addresses. It will happen in particular if a service intended for some local scope is programmed to bind to "unspecified" addresses, which in practice means every address configured for the device (please see Section 8.2).

8.2. Current Limitations in the Address Selection APIs

Application developers using the BSD Sockets API can "bind" a listening socket to a specific address, and ensure that the application is only reachable through that address. In theory, careful selection of the binding address could mitigate the problems described in Section 8.1. Binding services to temporary addresses could mitigate the ability of an attacker from testing for the presence of the node in the network. Binding different services to different addresses could mitigate unexpected discovery. Binding services to link local addresses or ULA could mitigate availability outside the expected scope. However, explicitly managing addresses adds significant complexity to the application development. It requires that application developers master addressing architecture subtleties, and implement logic that reacts adequately to connectivity events and address changes. Experience shows that application developers would probably prefer some much simpler solution.

In addition, we should note that many application developers use high level APIs that listen to TLS, HTTP, or some other application protocol. These high level APIs seldom provide detailed access to specific IP addresses, and typically default to listening to all available addresses.

A more advanced API could allow an application programmer to select desired properties in an address (scope, lifespan, etc.), such that the best-suitable addresses are selected, while relieving the application for low-level IPv6 addressing details. Such API might also trigger the generation of new IPv6 addresses when the specified properties would require so.

8.3. Sub-optimal IPv6 Address Configuration

Most operating systems configure the same types of addresses regardless of the current "operating mode" or "profile" of the device (e.g., device connected to enterprise network vs roaming across untrusted networks). For example, many operating systems configure both stable [RFC8064] and temporary [RFC4941] addresses on all network interfaces. However, this "one size fits all" approach tends to be sub-optimal or inappropriate for some scenarios. For example, enterprise networks typically prefer usage of only stable address, thus meaning that a network administator needs to find the means for disabling the generation of temporary addresses on all those systems that would otherwise generate them. On the other hand, some mobile devices configure both stable and temporary addresses, even when their usage pattern (client-like operation, as opposed to offering services to other nodes) would allow for the more privacy-sensible option of configuring only temporary addresses.

The lack of better tuned address configuration policies has helped the "one size fits all" approach that, as noted, may lead to suboptimal results. Advice in this area might help achieve more optional address generation policies such that IPv6 addressing capabilities are fully leveraged.

8.4. Sub-optimal IPv6 Address Usage

An application programmer, left with the question of which are the most appropriate addresses for a given usage type and application, typically resorts to the Default IPv6 Address Selection for IPv6 (see Section 6) for outgoing communications, and to accepting incoming communications on all available addresses for incoming communications. As discussed throughout this document, this leads to sub-optimal results. Besides, all applications on a node share the same pool of configured addresses, and applications are also prevented from triggering the generation of new addresses (e.g. to be employed for a particular application or communcation instance).

Guidance in this area is warranted such that applications and systems fully-leverage IPv6 addressing.

9. IANA Considerations

There are no IANA registries within this document. The RFC-Editor can remove this section before publication of this document as an RFC.

10. Security Considerations

The security and privacy implications associated with the predictability and lifetime of IPv6 addresses has been analyzed in [RFC7217] [RFC7721], and [RFC7707]. This document complements and extends the aforementioned analysis by considering other IPv6 properties such as the address scope and address usage type, and the associated tradeoffs.

11. Acknowledgements

The authors would like to thank (in alphabetical order) Francis Dupont, Tatuya Jinmei, Erik Kline, Tommy Pauly, and Dave Thaler for providing valuable comments on earlier versions of this document.

Fernando Gont would like to thank Spencer Dawkins for his guidance.

12. References

12.1. Normative References

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

[RFC4193] Hinden, R. and B. Haberman, "Unique Local IPv6 Unicast Addresses", RFC 4193, DOI 10.17487/RFC4193, October 2005, <https://www.rfc-editor.org/info/rfc4193>.

[RFC4291] Hinden, R. and S. Deering, "IP Version 6 Addressing Architecture", RFC 4291, DOI 10.17487/RFC4291, February 2006, <https://www.rfc-editor.org/info/rfc4291>.

[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>.

[RFC5905] Mills, D., Martin, J., Ed., Burbank, J., and W. Kasch, "Network Time Protocol Version 4: Protocol and Algorithms Specification", RFC 5905, DOI 10.17487/RFC5905, June 2010, <https://www.rfc-editor.org/info/rfc5905>.

[RFC6724] Thaler, D., Ed., Draves, R., Matsumoto, A., and T. Chown, "Default Address Selection for Internet Protocol Version 6 (IPv6)", RFC 6724, DOI 10.17487/RFC6724, September 2012, <https://www.rfc-editor.org/info/rfc6724>.

[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>.

[RFC7217] Gont, F., "A Method for Generating Semantically Opaque Interface Identifiers with IPv6 Stateless Address Autoconfiguration (SLAAC)", RFC 7217, DOI 10.17487/RFC7217, April 2014, <https://www.rfc-editor.org/info/rfc7217>.

[RFC8064] Gont, F., Cooper, A., Thaler, D., and W. Liu, "Recommendation on Stable IPv6 Interface Identifiers", RFC 8064, DOI 10.17487/RFC8064, February 2017, <https://www.rfc-editor.org/info/rfc8064>.

12.2. Informative References

[Barnes2012] Barnes, R., Altmann, R., and D. Kerr, "Mapping the Great Void Smarter scanning for IPv6", ISMA 2012 AIMS-4 - Workshop on Active Internet Measurements, February 2012, <https://www.caida.org/workshops/isma/1202/slides/aims1202_rbarnes.pdf>.

[Hein] Hein, B., "The Rising Sophistication of Network Scanning", January 2016, <http://netpatterns.blogspot.be/2016/01/the-rising-sophistication-of-network.html>.

[I-D.gont-6man-address-usage-recommendations] Gont, F., Gont, G., Corbo, M., and C. Huitema, "Problem Statement Regarding IPv6 Address Usage", draft-gont-6man-address-usage-recommendations-04 (work in progress), October 2017.

[I-D.gont-6man-non-stable-iids] Gont, F., Huitema, C., Gont, G., and M. Corbo, "Recommendation on Temporary IPv6 Interface Identifiers", draft-gont-6man-non-stable-iids-01 (work in progress), March 2017.

[I-D.gont-opsawg-firewalls-analysis] Gont, F. and F. Baker, "On Firewalls in Network Security", draft-gont-opsawg-firewalls-analysis-02 (work in progress), February 2016.

[I-D.ietf-v6ops-ula-usage-considerations]
Liu, B. and S. Jiang, "Considerations For Using Unique Local Addresses", draft-ietf-v6ops-ula-usage-considerations-02 (work in progress), March 2017.

[RFC7707] Gont, F. and T. Chown, "Network Reconnaissance in IPv6 Networks", RFC 7707, DOI 10.17487/RFC7707, March 2016, <https://www.rfc-editor.org/info/rfc7707>.

[RFC7721] Cooper, A., Gont, F., and D. Thaler, "Security and Privacy Considerations for IPv6 Address Generation Mechanisms", RFC 7721, DOI 10.17487/RFC7721, March 2016, <https://www.rfc-editor.org/info/rfc7721>.

Authors' Addresses

Fernando Gont
SI6 Networks / UTN-FRH
Evaristo Carriego 2644
Haedo, Provincia de Buenos Aires 1706
Argentina

Phone: +54 11 4650 8472
Email: fgont@si6networks.com
URI: http://www.si6networks.com

Guillermo Gont
SI6 Networks
Evaristo Carriego 2644
Haedo, Provincia de Buenos Aires 1706
Argentina

Phone: +54 11 4650 8472
Email: ggont@si6networks.com
URI: https://www.si6networks.com

Madeleine Garcia Corbo
Servicios de Informacion del Transporte
Neptuno 358
Havana City 10400
Cuba

Email: madelen.garcia16@gmail.com

Christian Huitema
Private Octopus Inc.
Friday Harbor, WA 98250
U.S.A.

Email: huitema@huitema.net
URI: http://privateoctopus.com

TAPS
Internet-Draft
Intended status: Informational
Expires: September 1, 2018

M. Welzl
S. Gjessing
University of Oslo

February 28, 2018

A Minimal Set of Transport Services for TAPS Systems
draft-ietf-taps-minset-02

Abstract

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

Status of This Memo

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

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

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

This Internet-Draft will expire on September 1, 2018.

Table of Contents

1. Introduction

The task of any system that implements TAPS is to offer transport services to its applications, i.e. the applications running on top of the transport system, without binding them to a particular transport protocol. Currently, the set of transport services that most applications use is based on TCP and UDP (and protocols that are layered on top of them); this limits the ability for the network stack to make use of features of other transport protocols. For

example, if a protocol supports out-of-order message delivery but applications always assume that the network provides an ordered bytestream, then the network stack can not immediately deliver a message that arrives out-of-order: doing so would break a fundamental assumption of the application. The net result is unnecessary head-of-line blocking delay.

By exposing the transport services of multiple transport protocols, a TAPS transport system can make it possible to use these services without having to statically bind an application to a specific transport protocol. The first step towards the design of such a system was taken by [RFC8095], which surveys a large number of transports, and [RFC8303] as well as [RFC8304], which identify the specific transport features that are exposed to applications by the protocols TCP, MPTCP, UDP(-Lite) and SCTP as well as the LEDBAT congestion control mechanism. This memo is based on these documents and follows the same terminology (also listed below). Because the considered transport protocols conjointly cover a wide range of transport features, there is reason to hope that the resulting set (and the reasoning that led to it) will also apply to many aspects of other transport protocols.

The number of transport features of current IETF transports is large, and exposing all of them has a number of disadvantages: generally, the more functionality is exposed, the less freedom a transport system has to automate usage of the various functions of its available set of transport protocols. Some functions only exist in one particular protocol, and if an application would use them, this would statically tie the application to this protocol, counteracting the purpose of TAPS. Also, if the number of exposed features is exceedingly large, a transport system might become very difficult to use for an application programmer. Taking [RFC8303] as a basis, this document therefore develops a minimal set of transport features, removing the ones that could be harmful to the purpose of TAPS but keeping the ones that must be retained for applications to benefit from useful transport functionality.

Applications use a wide variety of APIs today. The transport features in the minimal set in this document must be reflected in *all* network APIs in order for the underlying functionality to become usable everywhere. For example, it does not help an application that talks to a middleware if only the Berkeley Sockets API is extended to offer "unordered message delivery", but the middleware only offers an ordered bytestream. Both the Berkeley Sockets API and the middleware would have to expose the "unordered message delivery" transport feature (alternatively, there may be ways for certain types of middleware to use this transport feature without exposing it, based on knowledge about the applications -- but this is

not the general case). In most situations, in the interest of being as flexible and efficient as possible, the best choice will be for a middleware or library to expose at least all of the transport features that are recommended as a "minimal set" here.

This "minimal set" can be implemented one-sided over TCP (or UDP, if certain limitations are put in place). This means that a sender-side TAPS system implementing it can talk to a non-TAPS TCP (or UDP) receiver, and a receiver-side TAPS system implementing it can talk to a non-TAPS TCP (or UDP) sender.

2. Terminology

The following terms are used throughout this document, and in subsequent documents produced by TAPS that describe the composition and decomposition of transport services.

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

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

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

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

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

Application-specific knowledge: knowledge that only applications have.

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

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

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

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

3. The Minimal Set of Transport Features

Based on the categorization, reduction and discussion in Appendix A, this section describes the minimal set of transport features that is offered by end systems supporting TAPS. The described transport system can be implemented over TCP; elements of the system that may prohibit implementation over UDP are marked with "!UDP". To implement a transport system that can also work over UDP, these marked transport features should be excluded.

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

3.1. ESTABLISHMENT, AVAILABILITY and TERMINATION

A connection must first be "created" to allow for some initial configuration to be carried out before the transport system can actively or passively establish communication with a remote endpoint. All configuration parameters in Section 3.2 can be used initially, although some of them may only take effect when a connection has been established with a chosen transport protocol. Configuring a connection early helps a transport system make the right decisions. For example, grouping information can influence the transport system to implement a connection as a stream of a multi-streaming protocol’s existing association or not.

For ungrouped connections, early configuration is necessary because it allows the transport system to know which protocols it should try to use (to steer a mechanism such as "Happy Eyeballs" [I-D.grinnemo-taps-he]). In particular, a transport system that only makes a one-time choice for a particular protocol must know early about strict requirements that must be kept, or it can end up in a deadlock situation (e.g., having chosen UDP and later be asked to support reliable transfer). As a possibility to correctly handle these cases, we provide the following decision tree (this is derived from Appendix A.2.1 excluding authentication, as explained in Section 7):

- Will it ever be necessary to offer any of the following?
  * Reliably transfer data
  * Notify the peer of closing/aborting
  * Preserve data ordering

  Yes: SCTP or TCP can be used.
  - Is any of the following useful to the application?
    * Choosing a scheduler to operate between connections in a group, with the possibility to configure a priority or weight per connection
    * Configurable message reliability
    * Unordered message delivery
    * Request not to delay the acknowledgement (SACK) of a message

    Yes: SCTP is preferred.
    No:
    - Is any of the following useful to the application?
      * Hand over a message to reliably transfer (possibly multiple times) before connection establishment
      * Suggest timeout to the peer
      * Notification of Excessive Retransmissions (early warning below abortion threshold)
      * Notification of ICMP error message arrival

      Yes: TCP is preferred.
      No: SCTP and TCP are equally preferable.

  No: all protocols can be used.
  - Is any of the following useful to the application?
    * Specify checksum coverage used by the sender
    * Specify minimum checksum coverage required by receiver

    Yes: UDP-Lite is preferred.
    No: UDP is preferred.

Note that this decision tree is not optimal for all cases. For example, if an application wants to use "Specify checksum coverage used by the sender", which is only offered by UDP-Lite, and "Configure priority or weight for a scheduler", which is only offered by SCTP, the above decision tree will always choose UDP-Lite, making it impossible to use SCTP's schedulers with priorities between grouped connections. The transport system must know which choice is more important for the application in order to make the best decision. We caution implementers to be aware of the full set of trade-offs, for which we recommend consulting the list in Appendix A.2.1 when deciding how to initialize a connection.

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

- Reliability: a boolean that should be set to true when any of the following will be useful to the application: reliably transfer data; notify the peer of closing/aborting; preserve data ordering.
- Checksum_coverage: a boolean to specify whether it will be useful to the application to specify checksum coverage when sending or receiving.
- Config_msg_prio: a boolean that should be set to true when any of the following per-message configuration or prioritization mechanisms will be useful to the application: choosing a scheduler to operate between grouped connections, with the possibility to configure a priority or weight per connection; configurable message reliability; unordered message delivery; requesting not to delay the acknowledgement (SACK) of a message.
- Earlymsg_timeout_notifications: a boolean that should be set to true when any of the following will be useful to the application: hand over a message to reliably transfer (possibly multiple times) before connection establishment; suggest timeout to the peer; notification of excessive retransmissions (early warning below abortion threshold); notification of ICMP error message arrival.

Once a connection is created, it can be queried for the maximum amount of data that an application can possibly expect to have reliably transmitted before or during transport connection establishment (with zero being a possible answer) (see Section 3.2.1). An application can also give the connection a message for reliable transmission before or during connection establishment (!UDP); the transport system will then try to transmit it as early as possible. An application can facilitate sending a message particularly early by marking it as "idempotent" (see Section 3.3.1); in this case, the receiving application must be prepared to potentially receive multiple copies of the message (because idempotent messages are reliably transferred, asking for idempotence is not necessary for systems that support UDP).

After creation, a transport system can actively establish communication with a peer, or it can passively listen for incoming connection requests. Note that active establishment may or may not trigger a notification on the listening side. It is possible that the first notification on the listening side is the arrival of the first data that the active side sends (a receiver-side transport system could handle this by continuing to block a "Listen" call, immediately followed by issuing "Receive", for example; callback-based implementations could simply skip the equivalent of "Listen"). This also means that the active opening side is assumed to be the first side sending data.

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

3.2. MAINTENANCE

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

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

3.2.1. Connection groups

The following transport features and notifications (some directly from Appendix A.2, some new or changed, based on the discussion in Appendix A.3) automatically apply to all grouped connections:

(!UDP) Configure a timeout: this can be done with the following parameters:

- A timeout value for aborting connections, in seconds
- A timeout value to be suggested to the peer (if possible), in seconds
- The number of retransmissions after which the application should be notifed of "Excessive Retransmissions"

Configure urgency: this can be done with the following parameters:

- A number to identify the type of scheduler that should be used to operate between connections in the group (no guarantees given). Schedulers are defined in [RFC8260].
- A "capacity profile" number to identify how an application wants to use its available capacity. Choices can be "lowest possible latency at the expense of overhead" (which would disable any Nagle-like algorithm), "scavenger", or values that help determine the DSCP value for a connection (e.g. similar to table 1 in [I-D.ietf-tsvwg-rtcweb-qos]).
- A buffer limit (in bytes); when the sender has less then the provided limit of bytes in the buffer, the application may be notified. Notifications are not guaranteed, and it is optional for a transport system to support buffer limit values greater than 0. Note that this limit and its notification should operate across the buffers of the whole transport system, i.e. also any potential buffers that the transport system itself may use on top of the transport's send buffer.

Following Appendix A.3.7, these properties can be queried:

- The maximum message size that may be sent without fragmentation via the configured interface. This is optional for a transport system to offer, and may return an error ("not available"). It can aid applications implementing Path MTU Discovery.
- The maximum transport message size that can be sent, in bytes. Irrespective of fragmentation, there is a size limit for the messages that can be handed over to SCTP or UDP(-Lite); because the service provided by a transport system is independent of the transport protocol, it must allow an application to query this value -- the maximum size of a message in an Application-Framed-Bytestream (see Appendix A.3.1). This may also return an error when data is not delimited ("not available").
- The maximum transport message size that can be received from the configured interface, in bytes (or "not available").
- The maximum amount of data that can possibly be sent before or during connection establishment, in bytes.

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

- Excessive Retransmissions: the configured (or a default) number of retransmissions has been reached, yielding this early warning below an abortion threshold.
- ICMP Arrival (parameter: ICMP message): an ICMP packet carrying the conveyed ICMP message has arrived.

- ECN Arrival (parameter: ECN value): a packet carrying the conveyed ECN value has arrived. This can be useful for applications implementing congestion control.
- Timeout (parameter: s seconds): data could not be delivered for s seconds.
- Drain: the send buffer has either drained below the configured buffer limit or it has become completely empty. This is a generic notification that tries to enable uniform access to "TCP_NOTSENT_LOWAT" as well as the "SENDER DRY" notification (as discussed in Appendix A.3.4 -- SCTP's "SENDER DRY" is a special case where the threshold (for unsent data) is 0 and there is also no more unacknowledged data in the send buffer).

3.2.2. Individual connections

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

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

- A boolean to enable / disable usage of a checksum when sending
- The desired coverage (in bytes) of the checksum used when sending
- A boolean to enable / disable requiring a checksum when receiving
- The required minimum coverage (in bytes) of the checksum when receiving

3.3. DATA Transfer

3.3.1. Sending Data

When sending a message, no guarantees are given about the preservation of message boundaries to the peer; if message boundaries are needed, the receiving application at the peer must know about them beforehand (or the transport system cannot use TCP). Note that an application should already be able to hand over data before the transport system establishes a connection with a chosen transport protocol. Regarding the message that is being handed over, the following parameters can be used:

- Reliability: this parameter is used to convey a choice of: fully reliable (!UDP), unreliable without congestion control, unreliable (!UDP), partially reliable (see [RFC3758] and [RFC7496] for details on how to specify partial reliability) (!UDP). The latter two choices are optional for a transport system to offer and may result in full reliability. Note that applications sending

unreliable data without congestion control should themselves perform congestion control in accordance with [RFC2914].

- o (!UDP) Ordered: this boolean parameter lets an application choose between ordered message delivery (true) and possibly unordered, potentially faster message delivery (false).
- o Bundle: a boolean that expresses a preference for allowing to bundle messages (true) or not (false). No guarantees are given.
- o DelAck: a boolean that, if false, lets an application request that the peer would not delay the acknowledgement for this message.
- o Fragment: a boolean that expresses a preference for allowing to fragment messages (true) or not (false), at the IP level. No guarantees are given.
- o (!UDP) Idempotent: a boolean that expresses whether a message is idempotent (true) or not (false). Idempotent messages may arrive multiple times at the receiver (but they will arrive at least once). When data is idempotent it can be used by the receiver immediately on a connection establishment attempt. Thus, if data is handed over before the transport system establishes a connection with a chosen transport protocol, stating that a message is idempotent facilitates transmitting it to the peer application particularly early.

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

3.3.2. Receiving Data

A receiving application obtains an "Application-Framed Bytestream" (AFra-Bytestream); this concept is further described in Appendix A.3.1). In line with TCP's receiver semantics, an AFra-Bytestream is just a stream of bytes to the receiver. If message boundaries were specified by the sender, a receiver-side transport system implementing only the minimum set of transport services defined here will still not inform the receiving application about them (this limitation is only needed for transport systems that are implemented to directly use TCP).

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

4. Conclusion

By decoupling applications from transport protocols, a TAPS transport system provides a different abstraction level than the Berkeley sockets interface. As with high- vs. low-level programming languages, a higher abstraction level allows more freedom for automation below the interface, yet it takes some control away from the application programmer. This is the design trade-off that a transport system developer is facing, and this document provides guidance on the design of this abstraction level. Some transport features are currently rarely offered by APIs, yet they must be offered or they can never be used ("functional" transport features). Other transport features are offered by the APIs of the protocols covered here, but not exposing them in a TAPS API would allow for more freedom to automate protocol usage in a transport system. The minimal set presented in this document is an effort to find a middle ground that can be recommended for transport systems to implement, on the basis of the transport features discussed in [RFC8303].

5. Acknowledgements

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

6. IANA Considerations

XX RFC ED - PLEASE REMOVE THIS SECTION XXX

This memo includes no request to IANA.

7. Security Considerations

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

discussed further in a separate TAPS document [I-D.pauly-taps-transport-security].

8. References

8.1. Normative References

[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>.

8.2. Informative References

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

[I-D.grinnemo-taps-he] Grinnemo, K., Brunstrom, A., Hurtig, P., Khademi, N., and Z. Bozakov, "Happy Eyeballs for Transport Selection", draft-grinnemo-taps-he-03 (work in progress), July 2017.

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

[I-D.pauly-taps-transport-security] Pauly, T., Rose, K., and C. Wood, "A Survey of Transport Security Protocols", draft-pauly-taps-transport-security-01 (work in progress), January 2018.

[LBE-draft] Bless, R., "A Lower Effort Per-Hop Behavior (LE PHB)", Internet-draft draft-tsvwg-le-phb-03, February 2018.

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

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

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

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

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

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

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

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

[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>.

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

[RFC8304] Fairhurst, G. and T. Jones, "Transport Features of the User Datagram Protocol (UDP) and Lightweight UDP (UDP-Lite)", RFC 8304, DOI 10.17487/RFC8304, February 2018, <https://www.rfc-editor.org/info/rfc8304>.

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

Appendix A. Deriving the minimal set

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

1. Categorization: the superset of transport features from [RFC8303] is presented, and transport features are categorized for later reduction.
2. Reduction: a shorter list of transport features is derived from the categorization in the first step. This removes all transport features that do not require application-specific knowledge or cannot be implemented with TCP or UDP.
3. Discussion: the resulting list shows a number of peculiarities that are discussed, to provide a basis for constructing the minimal set.
4. Construction: Based on the reduced set and the discussion of the transport features therein, a minimal set is constructed.

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

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

Following [RFC8303], we divide the transport features into two main groups as follows:

1. CONNECTION related transport features
   - ESTABLISHMENT
   - AVAILABILITY
   - MAINTENANCE
   - TERMINATION

2. DATA Transfer related transport features
   - Sending Data
   - Receiving Data
   - Errors

We assume that applications have no specific requirements that need knowledge about the network, e.g. regarding the choice of network interface or the end-to-end path. Even with these assumptions, there

are certain requirements that are strictly kept by transport protocols today, and these must also be kept by a transport system. Some of these requirements relate to transport features that we call "Functional".

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

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

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

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

In this description, transport services are presented following the nomenclature "CATEGORY.[SUBCATEGORY].SERVICENAME.PROTOCOL", equivalent to "pass 2" in [RFC8303]. We also sketch how some of the TAPS transport features can be implemented by a transport system. For all transport features that are categorized as "functional" or "optimizing", and for which no matching TCP and/or UDP primitive exists in "pass 2" of [RFC8303], a brief discussion on how to implement them over TCP and/or UDP is included.

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

- Most transport features that are related to multi-streaming were designated as "automatable". This was done because the decision

on whether to use multi-streaming or not does not depend on application-specific knowledge. This means that a connection that is exhibited to an application could be implemented by using a single stream of an SCTP association instead of mapping it to a complete SCTP association or TCP connection. This could be achieved by using more than one stream when an SCTP association is first established (CONNECT.SCTP parameter "outbound stream count"), maintaining an internal stream number, and using this stream number when sending data (SEND.SCTP parameter "stream number"). Closing or aborting a connection could then simply free the stream number for future use. This is discussed further in Appendix A.3.2.

- All transport features that are related to using multiple paths or the choice of the network interface were designated as "automatable". Choosing a path or an interface does not depend on application-specific knowledge. For example, "Listen" could always listen on all available interfaces and "Connect" could use the default interface for the destination IP address.

A.1.1. CONNECTION Related Transport Features

ESTABLISHMENT:

- Connect
  Protocols: TCP, SCTP, UDP(-Lite)
  Functional because the notion of a connection is often reflected in applications as an expectation to be able to communicate after a "Connect" succeeded, with a communication sequence relating to this transport feature that is defined by the application protocol.
  Implementation: via CONNECT.TCP, CONNECT.SCTP or CONNECT.UDP(-Lite).

- Specify which IP Options must always be used
  Protocols: TCP, UDP(-Lite)
  Automatable because IP Options relate to knowledge about the network, not the application.

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

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

- Specify number of attempts and/or timeout for the first establishment message
  Protocols: TCP, SCTP
  Functional because this is closely related to potentially assumed reliable data delivery for data that is sent before or during connection establishment.
  Implementation: Using a parameter of CONNECT.TCP and CONNECT.SCTP.
  Implementation over UDP: Do nothing (this is irrelevant in case of UDP because there, reliable data delivery is not assumed).

- Obtain multiple sockets
  Protocols: SCTP
  Automatable because the usage of multiple paths to communicate to the same end host relates to knowledge about the network, not the application.

- Disable MPTCP
  Protocols: MPTCP
  Automatable because the usage of multiple paths to communicate to the same end host relates to knowledge about the network, not the application.
  Implementation: via a boolean parameter in CONNECT.MPTCP.

- Configure authentication
  Protocols: TCP, SCTP
  Functional because this has a direct influence on security.
  Implementation: via parameters in CONNECT.TCP and CONNECT.SCTP.
  Implementation over TCP: With TCP, this allows to configure Master Key Tuples (MKTs) to authenticate complete segments (including the TCP IPv4 pseudoheader, TCP header, and TCP data). With SCTP, this allows to specify which chunk types must always be authenticated. Authenticating only certain chunk types creates a reduced level of security that is not supported by TCP; to be compatible, this should therefore only allow to authenticate all chunk types. Key material must be provided in a way that is compatible with both [RFC4895] and [RFC5925].

Implementation over UDP: Not possible.

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

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

- Hand over a message to reliably transfer (possibly multiple times) before connection establishment
  Protocols: TCP
  Functional because this is closely tied to properties of the data that an application sends or expects to receive.
  Implementation: via a parameter in CONNECT.TCP.
  Implementation over UDP: not possible.

- Hand over a message to reliably transfer during connection establishment
  Protocols: SCTP
  Functional because this can only work if the message is limited in size, making it closely tied to properties of the data that an application sends or expects to receive.
  Implementation: via a parameter in CONNECT.SCTP.
  Implementation over UDP: not possible.

- Enable UDP encapsulation with a specified remote UDP port number
  Protocols: SCTP
  Automatable because UDP encapsulation relates to knowledge about the network, not the application.

AVAILABILITY:

- Listen
  Protocols: TCP, SCTP, UDP(-Lite)
  Functional because the notion of accepting connection requests is often reflected in applications as an expectation to be able to communicate after a "Listen" succeeded, with a communication sequence relating to this transport feature that is defined by the application protocol.
  ADDED. This differs from the 3 automatable transport features below in that it leaves the choice of interfaces for listening open.
  Implementation: by listening on all interfaces via LISTEN.TCP (not providing a local IP address) or LISTEN.SCTP (providing SCTP port number / address pairs for all local IP addresses). LISTEN.UDP(-Lite) supports both methods.

- Listen, 1 specified local interface
  Protocols: TCP, SCTP, UDP(-Lite)
  Automatable because decisions about local interfaces relate to knowledge about the network and the Operating System, not the application.

- Listen, N specified local interfaces
  Protocols: SCTP
  Automatable because decisions about local interfaces relate to knowledge about the network and the Operating System, not the application.

- Listen, all local interfaces
  Protocols: TCP, SCTP, UDP(-Lite)
  Automatable because decisions about local interfaces relate to knowledge about the network and the Operating System, not the application.

- Specify which IP Options must always be used
  Protocols: TCP, UDP(-Lite)
  Automatable because IP Options relate to knowledge about the network, not the application.

- Disable MPTCP
  Protocols: MPTCP
  Automatable because the usage of multiple paths to communicate to the same end host relates to knowledge about the network, not the application.

- Configure authentication
  Protocols: TCP, SCTP
  Functional because this has a direct influence on security.
  Implementation: via parameters in LISTEN.TCP and LISTEN.SCTP.
  Implementation over TCP: With TCP, this allows to configure Master Key Tuples (MKTs) to authenticate complete segments (including the TCP IPv4 pseudoheader, TCP header, and TCP data). With SCTP, this allows to specify which chunk types must always be authenticated. Authenticating only certain chunk types creates a reduced level of security that is not supported by TCP; to be compatible, this should therefore only allow to authenticate all chunk types. Key material must be provided in a way that is compatible with both [RFC4895] and [RFC5925].
  Implementation over UDP: not possible.

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

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

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

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

MAINTENANCE:

- Change timeout for aborting connection (using retransmit limit or time value)
  Protocols: TCP, SCTP
  Functional because this is closely related to potentially assumed reliable data delivery.
  Implementation: via CHANGE_TIMEOUT.TCP or CHANGE_TIMEOUT.SCTP.
  Implementation over UDP: not possible (UDP is unreliable and there is no connection timeout).

- Suggest timeout to the peer
  Protocols: TCP
  Functional because this is closely related to potentially assumed reliable data delivery.
  Implementation: via CHANGE_TIMEOUT.TCP.
  Implementation over UDP: not possible (UDP is unreliable and there is no connection timeout).

- Disable Nagle algorithm
  Protocols: TCP, SCTP
  Optimizing because this decision depends on knowledge about the size of future data blocks and the delay between them.
  Implementation: via DISABLE_NAGLE.TCP and DISABLE_NAGLE.SCTP.
  Implementation over UDP: do nothing (UDP does not implement the Nagle algorithm).

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

- Notification of Excessive Retransmissions (early warning below abortion threshold)
  Protocols: TCP
  Optimizing because it is an early warning to the application, informing it of an impending functional event.
  Implementation: via ERROR.TCP.
  Implementation over UDP: do nothing (there is no abortion threshold).

- Add path
  Protocols: MPTCP, SCTP
  MPTCP Parameters: source-IP; source-Port; destination-IP; destination-Port
  SCTP Parameters: local IP address
  Automatable because the usage of multiple paths to communicate to the same end host relates to knowledge about the network, not the application.

- Remove path
  Protocols: MPTCP, SCTP
  MPTCP Parameters: source-IP; source-Port; destination-IP; destination-Port
  SCTP Parameters: local IP address
  Automatable because the usage of multiple paths to communicate to the same end host relates to knowledge about the network, not the application.

- Set primary path
  Protocols: SCTP
  Automatable because the usage of multiple paths to communicate to the same end host relates to knowledge about the network, not the application.

- Suggest primary path to the peer
  Protocols: SCTP
  Automatable because the usage of multiple paths to communicate to the same end host relates to knowledge about the network, not the application.

- Configure Path Switchover
  Protocols: SCTP
  Automatable because the usage of multiple paths to communicate to the same end host relates to knowledge about the network, not the application.

- Obtain status (query or notification)
  Protocols: SCTP, MPTCP
  SCTP parameters: association connection state; destination transport address list; destination transport address reachability states; current local and peer receiver window size; current local congestion window sizes; number of unacknowledged DATA chunks; number of DATA chunks pending receipt; primary path; most recent SRTT on primary path; RTO on primary path; SRTT and RTO on other destination addresses; MTU per path; interleaving supported yes/no
  MPTCP parameters: subflow-list (identified by source-IP; source-Port; destination-IP; destination-Port)
  Automatable because these parameters relate to knowledge about the network, not the application.

- Specify DSCP field
  Protocols: TCP, SCTP, UDP(-Lite)
  Optimizing because choosing a suitable DSCP value requires application-specific knowledge.
  Implementation: via SET_DSCP.TCP / SET_DSCP.SCTP / SET_DSCP.UDP(-Lite)

- Notification of ICMP error message arrival
  Protocols: TCP, UDP(-Lite)
  Optimizing because these messages can inform about success or failure of functional transport features (e.g., host unreachable relates to "Connect")
  Implementation: via ERROR.TCP or ERROR.UDP(-Lite).

- Obtain information about interleaving support
  Protocols: SCTP
  Automatable because it requires using multiple streams, but requesting multiple streams in the CONNECTION.ESTABLISHMENT category is automatable.
  Implementation: via STATUS.SCTP.

- Change authentication parameters
  Protocols: TCP, SCTP
  Functional because this has a direct influence on security.
  Implementation: via SET_AUTH.TCP and SET_AUTH.SCTP.
  Implementation over TCP: With SCTP, this allows to adjust key_id, key, and hmac_id. With TCP, this allows to change the preferred outgoing MKT (current_key) and the preferred incoming MKT (rnext_key), respectively, for a segment that is sent on the connection. Key material must be provided in a way that is compatible with both [RFC4895] and [RFC5925].
  Implementation over UDP: not possible.

- Obtain authentication information
  Protocols: SCTP
  Functional because authentication decisions may have been made by the peer, and this has an influence on the necessary application-level measures to provide a certain level of security.
  Implementation: via GET_AUTH.SCTP.
  Implementation over TCP: With SCTP, this allows to obtain key_id and a chunk list. With TCP, this allows to obtain current_key and rnext_key from a previously received segment. Key material must be provided in a way that is compatible with both [RFC4895] and [RFC5925].
  Implementation over UDP: not possible.

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

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

- Reset Association
  Protocols: SCTP
  Automatable because deciding to reset an association does not require application-specific knowledge.
  Implementation: via RESET_ASSOC.SCTP.

- Notification of Association Reset
  Protocols: STCP
  Automatable because this notification does not relate to application-specific knowledge.

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

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

- Choose a scheduler to operate between streams of an association
  Protocols: SCTP
  Optimizing because the scheduling decision requires application-specific knowledge. However, if a transport system would not use this, or wrongly configure it on its own, this would only affect the performance of data transfers; the outcome would still be correct within the "best effort" service model.
  Implementation: using SET_STREAM_SCHEDULER.SCTP.
  Implementation over TCP: do nothing.
  Implementation over UDP: do nothing.

- Configure priority or weight for a scheduler
  Protocols: SCTP
  Optimizing because the priority or weight requires application-specific knowledge. However, if a transport system would not use this, or wrongly configure it on its own, this would only affect the performance of data transfers; the outcome would still be correct within the "best effort" service model.
  Implementation: using CONFIGURE_STREAM_SCHEDULER.SCTP.
  Implementation over TCP: do nothing.
  Implementation over UDP: do nothing.

- Configure send buffer size

Protocols: SCTP
Automatable because this decision relates to knowledge about the network and the Operating System, not the application (see also the discussion in Appendix A.3.4).

- Configure receive buffer (and rwnd) size
  Protocols: SCTP
  Automatable because this decision relates to knowledge about the network and the Operating System, not the application.

- Configure message fragmentation
  Protocols: SCTP
  Automatable because fragmentation relates to knowledge about the network and the Operating System, not the application.
  Implementation: by always enabling it with CONFIG_FRAGMENTATION.SCTP and auto-setting the fragmentation size based on network or Operating System conditions.

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

- Configure delayed SACK timer
  Protocols: SCTP
  Automatable because the receiver-side decision to delay sending SACKs relates to knowledge about the network, not the application (it can be relevant for a sending application to request not to delay the SACK of a message, but this is a different transport feature).

- Set Cookie life value
  Protocols: SCTP
  Functional because it relates to security (possibly weakened by keeping a cookie very long) versus the time between connection establishment attempts. Knowledge about both issues can be application-specific.

Implementation over TCP: the closest specified TCP functionality is the cookie in TCP Fast Open; for this, [RFC7413] states that the server "can expire the cookie at any time to enhance security" and section 4.1.2 describes an example implementation where updating the key on the server side causes the cookie to expire. Alternatively, for implementations that do not support TCP Fast Open, this transport feature could also affect the validity of SYN cookies (see Section 3.6 of [RFC4987]).
Implementation over UDP: do nothing.

- Set maximum burst
  Protocols: SCTP
  Automatable because it relates to knowledge about the network, not the application.

- Configure size where messages are broken up for partial delivery
  Protocols: SCTP
  Functional because this is closely tied to properties of the data that an application sends or expects to receive.
  Implementation over TCP: not possible.
  Implementation over UDP: not possible.

- Disable checksum when sending
  Protocols: UDP
  Functional because application-specific knowledge is necessary to decide whether it can be acceptable to lose data integrity.
  Implementation: via SET_CHECKSUM_ENABLED.UDP.
  Implementation over TCP: do nothing.

- Disable checksum requirement when receiving
  Protocols: UDP
  Functional because application-specific knowledge is necessary to decide whether it can be acceptable to lose data integrity.
  Implementation: via SET_CHECKSUM_REQUIRED.UDP.
  Implementation over TCP: do nothing.

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

Functional because application-specific knowledge is necessary to decide for which parts of the data it can be acceptable to lose data integrity.
Implementation: via SET_CHECKSUM_COVERAGE.UDP-Lite.
Implementation over TCP: do nothing.

- Specify minimum checksum coverage required by receiver
  Protocols: UDP-Lite
  Functional because application-specific knowledge is necessary to decide for which parts of the data it can be acceptable to lose data integrity.
  Implementation: via SET_MIN_CHECKSUM_COVERAGE.UDP-Lite.
  Implementation over TCP: do nothing.

- Specify DF field
  Protocols: UDP(-Lite)
  Optimizing because the DF field can be used to carry out Path MTU Discovery, which can lead an application to choose message sizes that can be transmitted more efficiently.
  Implementation: via MAINTENANCE.SET_DF.UDP(-Lite) and SEND_FAILURE.UDP(-Lite).
  Implementation over TCP: do nothing. With TCP the sender is not in control of transport message sizes, making this functionality irrelevant.

- Get max. transport-message size that may be sent using a non-fragmented IP packet from the configured interface
  Protocols: UDP(-Lite)
  Optimizing because this can lead an application to choose message sizes that can be transmitted more efficiently.
  Implementation over TCP: do nothing: this information is not available with TCP.

- Get max. transport-message size that may be received from the configured interface
  Protocols: UDP(-Lite)
  Optimizing because this can, for example, influence an application’s memory management.
  Implementation over TCP: do nothing: this information is not available with TCP.

- Specify TTL/Hop count field
  Protocols: UDP(-Lite)
  Automatable because a transport system can use a large enough system default to avoid communication failures. Allowing an application to configure it differently can produce notifications of ICMP error message arrivals that yield information which only relates to knowledge about the network, not the application.

- Obtain TTL/Hop count field
  Protocols: UDP(-Lite)
  Automatable because the TTL/Hop count field relates to knowledge about the network, not the application.

- Specify ECN field
  Protocols: UDP(-Lite)
  Automatable because the ECN field relates to knowledge about the network, not the application.

- Obtain ECN field
  Protocols: UDP(-Lite)
  Optimizing because this information can be used by an application to better carry out congestion control (this is relevant when choosing a data transmission transport service that does not already do congestion control).
  Implementation over TCP: do nothing: this information is not available with TCP.

- Specify IP Options
  Protocols: UDP(-Lite)
  Automatable because IP Options relate to knowledge about the network, not the application.

- Obtain IP Options
  Protocols: UDP(-Lite)
  Automatable because IP Options relate to knowledge about the network, not the application.

- Enable and configure a "Low Extra Delay Background Transfer"
  Protocols: A protocol implementing the LEDBAT congestion control mechanism
  Optimizing because whether this service is appropriate or not depends on application-specific knowledge. However, wrongly using this will only affect the speed of data transfers (albeit including other transfers that may compete with the transport system's transfer in the network), so it is still correct within the "best effort" service model.
  Implementation: via CONFIGURE.LEDBAT and/or SET_DSCP.TCP / SET_DSCP.SCTP / SET_DSCP.UDP(-Lite) [LBE-draft].
  Implementation over TCP: do nothing.
  Implementation over UDP: do nothing.

TERMINATION:

- Close after reliably delivering all remaining data, causing an event informing the application on the other side
  Protocols: TCP, SCTP
  Functional because the notion of a connection is often reflected in applications as an expectation to have all outstanding data delivered and no longer be able to communicate after a "Close" succeeded, with a communication sequence relating to this transport feature that is defined by the application protocol.
  Implementation: via CLOSE.TCP and CLOSE.SCTP.
  Implementation over UDP: not possible.

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

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

Protocols: UDP(-Lite)
Functional because the notion of a connection is often reflected in applications as an expectation to potentially not have all outstanding data delivered and no longer be able to communicate after an "Abort" succeeded. On both sides of a connection, an application protocol may define a communication sequence relating to this transport feature.
Implementation: via ABORT.UDP(-Lite).
Implementation over TCP: stop using the connection, wait for a timeout.

- Timeout event when data could not be delivered for too long
  Protocols: TCP, SCTP
  Functional because this notifies that potentially assumed reliable data delivery is no longer provided.
  Implementation: via TIMEOUT.TCP and TIMEOUT.SCTP.
  Implementation over UDP: do nothing: this event will not occur with UDP.

A.1.2. DATA Transfer Related Transport Features

A.1.2.1. Sending Data

- Reliably transfer data, with congestion control
  Protocols: TCP, SCTP
  Functional because this is closely tied to properties of the data that an application sends or expects to receive.
  Implementation: via SEND.TCP and SEND.SCTP.
  Implementation over UDP: not possible.

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

- Unreliably transfer a message
  Protocols: SCTP, UDP(-Lite)
  Optimizing because only applications know about the time criticality of their communication, and reliably transfering a message is never incorrect for the receiver of a potentially unreliable data transfer, it is just slower.
  ADDED. This differs from the 2 automatable transport features below in that it leaves the choice of congestion control open.
  Implementation: via SEND.SCTP or SEND.UDP(-Lite).
  Implementation over TCP: use SEND.TCP. With SEND.TCP, messages will be sent reliably, and they will not be identifiable by the receiver.

- Unreliably transfer a message, with congestion control
  Protocols: SCTP
  Automatable because congestion control relates to knowledge about the network, not the application.

- Unreliably transfer a message, without congestion control
  Protocols: UDP(-Lite)
  Automatable because congestion control relates to knowledge about the network, not the application.

- Configurable Message Reliability
  Protocols: SCTP
  Optimizing because only applications know about the time criticality of their communication, and reliably transfering a message is never incorrect for the receiver of a potentially unreliable data transfer, it is just slower.
  Implementation: via SEND.SCTP.
  Implementation over TCP: By using SEND.TCP and ignoring this configuration: based on the assumption of the best-effort service model, unnecessarily delivering data does not violate application expectations. Moreover, it is not possible to associate the requested reliability to a "message" in TCP anyway.
  Implementation over UDP: not possible.

- Choice of stream
  Protocols: SCTP

Automatable because it requires using multiple streams, but requesting multiple streams in the CONNECTION.ESTABLISHMENT category is automatable. Implementation: see Appendix A.3.2.

- Choice of path (destination address)
  Protocols: SCTP
  Automatable because it requires using multiple sockets, but obtaining multiple sockets in the CONNECTION.ESTABLISHMENT category is automatable.

- Ordered message delivery (potentially slower than unordered)
  Protocols: SCTP
  Functional because this is closely tied to properties of the data that an application sends or expects to receive.
  Implementation: via SEND.SCTP.
  Implementation over TCP: By using SEND.TCP. With SEND.TCP, messages will not be identifiable by the receiver.
  Implementation over UDP: not possible.

- Unordered message delivery (potentially faster than ordered)
  Protocols: SCTP, UDP(-Lite)
  Functional because this is closely tied to properties of the data that an application sends or expects to receive.
  Implementation: via SEND.SCTP.
  Implementation over TCP: By using SEND.TCP and always sending data ordered: based on the assumption of the best-effort service model, ordered delivery may just be slower and does not violate application expectations. Moreover, it is not possible to associate the requested delivery order to a "message" in TCP anyway.

- Request not to bundle messages
  Protocols: SCTP
  Optimizing because this decision depends on knowledge about the size of future data blocks and the delay between them.
  Implementation: via SEND.SCTP.
  Implementation over TCP: By using SEND.TCP and DISABLE_NAGLE.TCP to disable the Nagle algorithm when the request is made and enable it again when the request is no longer made. Note that this is not fully equivalent because it relates to the time of issuing the request rather than a specific message.

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

- Specifying a "payload protocol-id" (handed over as such by the receiver)
  Protocols: SCTP
  Functional because it allows to send extra application data with every message, for the sake of identification of data, which by itself is application-specific.
  Implementation: SEND.SCTP.
  Implementation over TCP: not possible.
  Implementation over UDP: not possible.

- Specifying a key id to be used to authenticate a message
  Protocols: SCTP
  Functional because this has a direct influence on security.
  Implementation: via a parameter in SEND.SCTP.
  Implementation over TCP: This could be emulated by using SET_AUTH.TCP before and after the message is sent. Note that this is not fully equivalent because it relates to the time of issuing the request rather than a specific message.
  Implementation over UDP: not possible.

- Request not to delay the acknowledgement (SACK) of a message
  Protocols: SCTP
  Optimizing because only an application knows for which message it wants to quickly be informed about success / failure of its delivery.
  Implementation over TCP: do nothing.
  Implementation over UDP: do nothing.

A.1.2.2. Receiving Data

- Receive data (with no message delimiting)
  Protocols: TCP
  Functional because a transport system must be able to send and receive data.
  Implementation: via RECEIVE.TCP.
  Implementation over UDP: do nothing (hand over a message, let the application ignore message boundaries).

- Receive a message
  Protocols: SCTP, UDP(-Lite)
  Functional because this is closely tied to properties of the data that an application sends or expects to receive.
  Implementation: via RECEIVE.SCTP and RECEIVE.UDP(-Lite).
  Implementation over TCP: not possible.

- Choice of stream to receive from
  Protocols: SCTP
  Automatable because it requires using multiple streams, but requesting multiple streams in the CONNECTION.ESTABLISHMENT category is automatable.
  Implementation: see Appendix A.3.2.

- Information about partial message arrival
  Protocols: SCTP
  Functional because this is closely tied to properties of the data that an application sends or expects to receive.
  Implementation: via RECEIVE.SCTP.
  Implementation over TCP: do nothing: this information is not available with TCP.
  Implementation over UDP: do nothing: this information is not available with UDP.

A.1.2.3. Errors

This section describes sending failures that are associated with a specific call to in the "Sending Data" category (Appendix A.1.2.1).

- Notification of send failures
  Protocols: SCTP, UDP(-Lite)
  Functional because this notifies that potentially assumed reliable data delivery is no longer provided.
  ADDED. This differs from the 2 automatable transport features below in that it does not distinugish between unsent and unacknowledged messages.
  Implementation: via SENDFAILURE-EVENT.SCTP and SEND_FAILURE.UDP(-Lite).
  Implementation over TCP: do nothing: this notification is not available and will therefore not occur with TCP.

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

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

- Notification that the stack has no more user data to send
  Protocols: SCTP
  Optimizing because reacting to this notification requires the application to be involved, and ensuring that the stack does not run dry of data (for too long) can improve performance.
  Implementation over TCP: do nothing. See also the discussion in Appendix A.3.4.
  Implementation over UDP: do nothing. This notification is not available and will therefore not occur with UDP.

- Notification to a receiver that a partial message delivery has been aborted
  Protocols: SCTP
  Functional because this is closely tied to properties of the data that an application sends or expects to receive.
  Implementation over TCP: do nothing. This notification is not available and will therefore not occur with TCP.
  Implementation over UDP: do nothing. This notification is not available and will therefore not occur with UDP.

A.2. Step 2: Reduction -- The Reduced Set of Transport Features

By hiding automatable transport features from the application, a transport system can gain opportunities to automate the usage of network-related functionality. This can facilitate using the transport system for the application programmer and it allows for optimizations that may not be possible for an application. For instance, system-wide configurations regarding the usage of multiple interfaces can better be exploited if the choice of the interface is

not entirely up to the application. Therefore, since they are not strictly necessary to expose in a transport system, we do not include automatable transport features in the reduced set of transport features. This leaves us with only the transport features that are either optimizing or functional.

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

In the following list, we precede a transport feature with "T:" if an implementation over TCP is possible, "U:" if an implementation over UDP is possible, and "TU:" if an implementation over either TCP or UDP is possible.

A.2.1. CONNECTION Related Transport Features

ESTABLISHMENT:

- T,U: Connect
- T,U: Specify number of attempts and/or timeout for the first establishment message
- T: Configure authentication
- T: Hand over a message to reliably transfer (possibly multiple times) before connection establishment
- T: Hand over a message to reliably transfer during connection establishment

AVAILABILITY:

- T,U: Listen
- T: Configure authentication

MAINTENANCE:

- T: Change timeout for aborting connection (using retransmit limit or time value)
- T: Suggest timeout to the peer
- T,U: Disable Nagle algorithm

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

TERMINATION:

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

A.2.2. DATA Transfer Related Transport Features

A.2.2.1. Sending Data

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

A.2.2.2. Receiving Data

- T,U: Receive data (with no message delimiting)
- U: Receive a message
- T,U: Information about partial message arrival

A.2.2.3. Errors

This section describes sending failures that are associated with a specific call to in the "Sending Data" category (Appendix A.1.2.1).

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

A.3. Step 3: Discussion

The reduced set in the previous section exhibits a number of peculiarities, which we will discuss in the following. This section focuses on TCP because, with the exception of one particular transport feature ("Receive a message" -- we will discuss this in Appendix A.3.1), the list shows that UDP is strictly a subset of TCP. We can first try to understand how to build a transport system that can run over TCP, and then narrow down the result further to allow that the system can always run over either TCP or UDP (which effectively means removing everything related to reliability, ordering, authentication and closing/aborting with a notification to the peer).

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

A.3.1. Sending Messages, Receiving Bytes

For implementing a transport system over TCP, there are several transport features related to sending, but only a single transport feature related to receiving: "Receive data (with no message delimiting)" (and, strangely, "information about partial message arrival"). Notably, the transport feature "Receive a message" is also the only non-automatable transport feature of UDP(-Lite) for which no implementation over TCP is possible.

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

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

For example, if an application requests to transfer fixed-size messages of 100 bytes with partial reliability, this needs the receiving application to be prepared to accept data in chunks of 100 bytes. If, then, some of these 100-byte messages are missing (e.g., if SCTP with Configurable Reliability is used), this is the expected application behavior. With TCP, no messages would be missing, but this is also correct for the application, and the possible retransmission delay is acceptable within the best effort service model [RFC7305]. Still, the receiving application would separate the byte stream into 100-byte chunks.

Note that this usage of messages does not require all messages to be equal in size. Many application protocols use some form of Type-Length-Value (TLV) encoding, e.g. by defining a header including length fields; another alternative is the use of byte stuffing methods such as COBS [COBS]. If an application needs message numbers, e.g. to restore the correct sequence of messages, these must also be encoded by the application itself, as the sequence number related transport features of SCTP are not provided by the "minimum set" (in the interest of enabling usage of TCP).

A.3.2. Stream Schedulers Without Streams

We have already stated that multi-streaming does not require application-specific knowledge. Potential benefits or disadvantages of, e.g., using two streams of an SCTP association versus using two separate SCTP associations or TCP connections are related to knowledge about the network and the particular transport protocol in use, not the application. However, the transport features "Choose a scheduler to operate between streams of an association" and

"Configure priority or weight for a scheduler" operate on streams. Here, streams identify communication channels between which a scheduler operates, and they can be assigned a priority. Moreover, the transport features in the MAINTENANCE category all operate on assocations in case of SCTP, i.e. they apply to all streams in that assocation.

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

To be compatible with multiple transport protocols and uniformly allow access to both transport connections and streams of a multi-streaming protocol, the semantics of opening and closing need to be the most restrictive subset of all of the underlying options. For example, TCP's support of half-closed connections can be seen as a feature on top of the more restrictive "ABORT"; this feature cannot be supported because not all protocols used by a transport system (including streams of an association) support half-closed connections.

A.3.3. Early Data Transmission

There are two transport features related to transferring a message early: "Hand over a message to reliably transfer (possibly multiple times) before connection establishment", which relates to TCP Fast Open [RFC7413], and "Hand over a message to reliably transfer during connection establishment", which relates to SCTP's ability to transfer data together with the COOKIE-Echo chunk. Also without TCP Fast Open, TCP can transfer data during the handshake, together with the SYN packet -- however, the receiver of this data may not hand it over to the application until the handshake has completed. Also, different from TCP Fast Open, this data is not delimited as a message by TCP (thus, not visible as a ''message''). This functionality is commonly available in TCP and supported in several implementations, even though the TCP specification does not explain how to provide it to applications.

A transport system could differentiate between the cases of transmitting data "before" (possibly multiple times) or "during" the handshake. Alternatively, it could also assume that data that are handed over early will be transmitted as early as possible, and "before" the handshake would only be used for messages that are

explicitly marked as "idempotent" (i.e., it would be acceptable to transfer them multiple times).

The amount of data that can successfully be transmitted before or during the handshake depends on various factors: the transport protocol, the use of header options, the choice of IPv4 and IPv6 and the Path MTU. A transport system should therefore allow a sending application to query the maximum amount of data it can possibly transmit before (or, if exposed, during) connection establishment.

A.3.4. Sender Running Dry

The transport feature "Notification that the stack has no more user data to send" relates to SCTP's "SENDER DRY" notification. Such notifications can, in principle, be used to avoid having an unnecessarily large send buffer, yet ensure that the transport sender always has data available when it has an opportunity to transmit it. This has been found to be very beneficial for some applications [WWDC2015]. However, "SENDER DRY" truly means that the entire send buffer (including both unsent and unacknowledged data) has emptied -- i.e., when it notifies the sender, it is already too late, the transport protocol already missed an opportunity to send data. Some modern TCP implementations now include the unspecified "TCP_NOTSENT_LOWAT" socket option that was proposed in [WWDC2015], which limits the amount of unsent data that TCP can keep in the socket buffer; this allows to specify at which buffer filling level the socket becomes writable, rather than waiting for the buffer to run empty.

SCTP allows to configure the sender-side buffer too: the automatable Transport Feature "Configure send buffer size" provides this functionality, but only for the complete buffer, which includes both unsent and unacknowledged data. SCTP does not allow to control these two sizes separately. It therefore makes sense for a transport system to allow for uniform access to "TCP_NOTSENT_LOWAT" as well as the "SENDER DRY" notification.

A.3.5. Capacity Profile

The transport features:

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

all relate to a QoS-like application need such as "low latency" or "scavenger". In the interest of flexibility of a transport system, they could therefore be offered in a uniform, more abstract way,

where a transport system could e.g. decide by itself how to use combinations of LEDBAT-like congestion control and certain DSCP values, and an application would only specify a general "capacity profile" (a description of how it wants to use the available capacity). A need for "lowest possible latency at the expense of overhead" could then translate into automatically disabling the Nagle algorithm.

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

A.3.6. Security

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

Security is discussed in a separate TAPS document [I-D.pauly-taps-transport-security]. The minimal set presented in the present document therefore excludes all security related transport features: "Configure authentication", "Change authentication parameters", "Obtain authentication information" and and "Set Cookie life value" as well as "Specifying a key id to be used to authenticate a message".

A.3.7. Packet Size

UDP(-Lite) has a transport feature called "Specify DF field". This yields an error message in case of sending a message that exceeds the Path MTU, which is necessary for a UDP-based application to be able to implement Path MTU Discovery (a function that UDP-based applications must do by themselves). The "Get max. transport-message size that may be sent using a non-fragmented IP packet from the configured interface" transport feature yields an upper limit for the Path MTU (minus headers) and can therefore help to implement Path MTU Discovery more efficiently.

Appendix B. Revision information

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

-02: implementation suggestions added, discussion section added, terminology extended, DELETED category removed, various other fixes; list of Transport Features adjusted to -01 version of [RFC8303] except that MPTCP is not included.

-03: updated to be consistent with -02 version of [RFC8303].

-04: updated to be consistent with -03 version of [RFC8303]. Reorganized document, rewrote intro and conclusion, and made a first stab at creating a real "minimal set".

-05: updated to be consistent with -05 version of [RFC8303] (minor changes). Fixed a mistake regarding Cookie Life value. Exclusion of security related transport features (to be covered in a separate document). Reorganized the document (now begins with the minset, derivation is in the appendix). First stab at an abstract API for the minset.

draft-ietf-taps-minset-00: updated to be consistent with -08 version of [RFC8303] ("obtain message delivery number" was removed, as this has also been removed in [RFC8303] because it was a mistake in RFC4960. This led to the removal of two more transport features that were only designated as functional because they affected "obtain message delivery number"). Fall-back to UDP incorporated (this was requested at IETF-99); this also affected the transport feature "Choice between unordered (potentially faster) or ordered delivery of messages" because this is a boolean which is always true for one fall-back protocol, and always false for the other one. This was therefore now divided into two features, one for ordered, one for unordered delivery. The word "reliably" was added to the transport features "Hand over a message to reliably transfer (possibly multiple times) before connection establishment" and "Hand over a message to reliably transfer during connection establishment" to make it clearer why this is not supported by UDP. Clarified that the "minset abstract interface" is not proposing a specific API for all TAPS systems to implement, but it is just a way to describe the minimum set. Author order changed.

WG -01: "fall-back to" (TCP or UDP) replaced (mostly with "implementation over"). References to post-sockets removed (these were statments that assumed that post-sockets requires two-sided implementation). Replaced "flow" with "TAPS Connection" and "frame" with "message" to avoid introducing new terminology. Made sections 3 and 4 in line with the categorization that is already used in the

appendix and [RFC8303], and changed style of section 4 to be even shorter and less interface-like. Updated reference draft-ietf-tsvwg-sctp-ndata to RFC8260.

WG -02: rephrased "the TAPS system" and "TAPS connection" etc. to more generally talk about transport after the intro (mostly replacing "TAPS system" with "transport system" and "TAPS connection" with "connection". Merged sections 3 and 4 to form a new section 3.

Authors' Addresses

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

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

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

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

TAPS Working Group
Internet-Draft
Intended status: Informational
Expires: August 30, 2018

T. Pauly, Ed.
Apple Inc.
B. Trammell, Ed.
ETH Zurich
A. Brunstrom
Karlstad University
G. Fairhurst
University of Aberdeen
C. Perkins
University of Glasgow
P. Tiesel
TU Berlin
C. Wood
Apple Inc.
February 26, 2018

An Architecture for Transport Services
draft-pauly-taps-arch-00

Abstract

This document provides an overview of the architecture of Transport Services, a system for exposing the features of transport protocols to applications. This architecture serves as a basis for Application Programming Interfaces (APIs) and implementations that provide flexible transport networking services. It defines the common set of terminology and concepts to be used in more detailed discussion of Transport Services.

Table of Contents

1. Introduction

Many APIs to perform transport networking have been deployed, perhaps the most widely known and imitated being the BSD socket() interface. The names and functions between these APIs are not consistent, and vary depending on the protocol being used. For example, sending and receiving on a stream of data is conceptually the same between operating on an unencrypted TCP stream and operating on an encrypted

TLS stream over TCP, but applications cannot use the same socket send() and recv() calls on top of both kinds of connections. Similarly, terminology for the implementation of protocols offering transport services vary based on the context of the protocols themselves. This variety can lead to confusion when trying to understand the similarities and differences between protocols, and how applications can use them effectively.

The goal of the Transport Services architecture is to provide a common, flexible, and reusable interface for 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.

This document is developed in parallel with the specification of the Transport Services API [draft-trammell-taps-interface] and Implementation [draft-brunstrom-taps-impl] documents.

2. Background

The Transport Services architecture is based on the survey of Services Provided by IETF Transport Protocols and Congestion Control Mechanisms [RFC8095], and the distilled minimal set of the features offered by transport protocols [I-D.ietf-taps-minset]. This work has identified common features and patterns across all transport protocols developed thus far in the IETF.

Since transport security is an increasingly relevant aspect of using transport protocols on the Internet, this architecture also considers the impact of transport security protocols on the feature set exposed by transport services [I-D.pauly-taps-transport-security].

One of the key insights to come from identifying the minimal set of features provided by transport protocols [I-D.ietf-taps-minset] was that features either require application interaction and guidance (referred to as Functional Features), or else can be handled automatically by a system implementing Transport Services (referred to as Automatable Features). Among the Functional Features, some were common across all or nearly all transport protocols, while others could be seen as features that, if specified, would only be useful with a subset of protocols, or perhaps even a single transport protocol, but would not harm the functionality of other protocols. For example, some protocols can deliver messages faster for applications that do not require them to arrive in the order in which they were sent. However, this functionality must be explicitly allowed by the application, since reordering messages would be undesirable in many cases.

3. Design Principles

The goal of the Transport Services architecture is to redefine the interface between applications and transports in a way that allows the transport layer to evolve and improve without fundamentally changing the contract with the application. This requires a careful consideration of how to expose the capabilities of protocols.

There are several degrees in which a Transport Services system can offer flexibility to an application: it can provide access to multiple sets of protocols and protocol features, it can use these protocols across multiple paths that may have different performance and functional characteristics, and it can communicate with different Remote Endpoints to optimize performance. Beyond these, if the API for the system remains the same over time, new protocols and features may be added to the system's implementation without requiring significant changes in applications for adoption.

The following considerations were used in the design of this architecture.

3.1. Common APIs for Common Features

Functionality that is common across multiple transport protocols should be accessible through a unified set of API calls. An application should be able to implement logic for its basic use of transport networking (establishing the transport, and sending and receiving data) once, and expect that implementation to continue to function as the transports change.

Any Transport Services API must allow access to the distilled minimal set of features offered by transport protocols [I-D.ietf-taps-minset].

3.2. Access to Specialized Features

Since applications will often need to control fine-grained details of transport protocols to optimize their behavior and ensure compatibility with remote peers, a Transport Services system also needs to allow more specialized protocol features to be used. The interface for these specialized options should be exposed differently from the common options to ensure flexibility.

A specialized feature may be required by an application only when using a specific protocol, and not when using others. For example, if an application is using UDP, it may require control over the checksum or fragmentation behavior for UDP; if it used a protocol to frame its data over a byte stream like TCP, it would not need these

options. In such cases, the API should expose the features in such a way that they take effect when a particular protocol is selected, but do not imply that only that protocol may be used if there are equivalent options.

Other specialized features, however, may be strictly required by an application and thus constrain the set of protocols that can be used. For example, if an application requires encryption of its transport data, only protocol stacks that include some transport security protocol are eligible to be used. A Transport Services API must allow applications to define such requirements and constrain the system's options. Since such options are not part of the core/common features, it should be simple for an application to modify its set of constraints and change the set of allowable protocol features without changing the core implementation.

3.3. Scope for API and Implementation Definitions

The Transport Services API is envisioned as the abstract model for a family of APIs that share a common way to expose transport features and encourage flexibility. The abstract API definition [draft-trammell-taps-interface] describes this interface and is aimed at application developers.

Implementations that provide the Transport Services API [draft-brunstrom-taps-impl] will vary due to system-specific support and the needs of the deployment scenario. It is expected that all implementations of Transport Services will offer the entire mandatory API, but that some features will not be functional in certain implementations. All implementations must offer sufficient APIs to use the distilled minimal set of features offered by transport protocols [I-D.ietf-taps-minset], including API support for TCP and UDP transport, but it is possible that some very constrained devices might not have, for example, a full TCP implementation.

In order to preserve flexibility and compatibility with future protocols, top-level features in the Transport Services API should avoid referencing particular transport protocols. Mappings of these API features in the Implementation document, on the other hand, must explain the ramifications of each feature on existing protocols. It is expected that the Implementation document will be updated and supplemented as new protocols and protocol features are developed.

It is important to note that neither the Transport Services API nor the Implementation document defines new protocols that require any changes on remote hosts. The Transport Services system must be deployable on one side only, as a way to allow an application to make

better use of available capabilities on a system and protocol features that may be supported by peers across the network.

4. Transport Services Architecture and Concepts

The concepts defined in this document are intended primarily for use in the documents and specifications that describe the Transport Services architecture and API. While the specific terminology may be used in some implementations, it is expected that there will remain a variety of terms used by running code.

The architecture divides the concepts for Transport Services into two categories:

1. API concepts, which are meant to be exposed to applications; and

2. System-implementation concepts, which are meant to be internally used when building systems that implement Transport Services.

The following diagram summarizes the top-level concepts in the architecture and how they relate to one another.

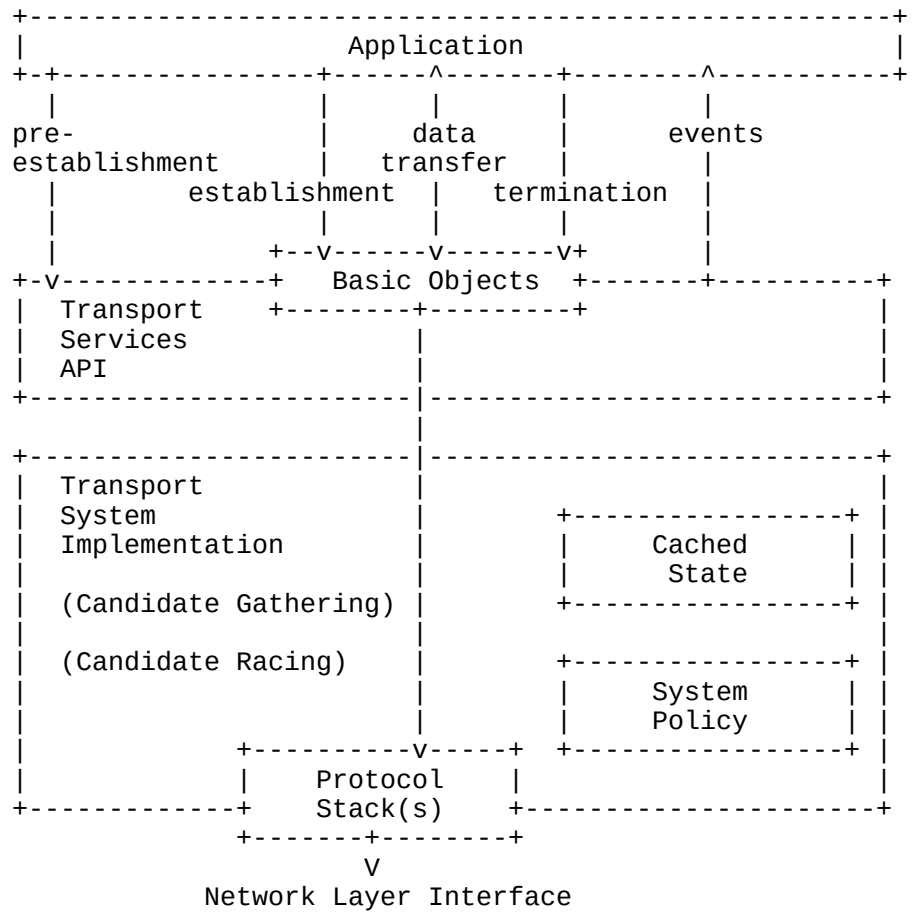


Figure 1: Concepts and Relationships in the Transport Services Architecture

4.1. Transport Services API Concepts

Fundamentally, a Transport Services API needs to provide basic objects (Section 4.1.1) that allow applications to establish communication and send and receive data. These may be exposed as handles or referenced objects, depending on the language.

Beyond the basic objects, there are several high-level groups of actions that any Transport Services API must provide:

- Pre-Establishment (Section 4.1.2) encompasses the properties that an application can pass to describe its intent, requirements, prohibitions, and preferences for its networking operations. For

any system that provides generic Transport Services, these properties should primarily offer knobs that are applicable to multiple transports. Properties may have a large impact on the rest of the aspects of the interface: they can modify how establishment occurs, they can influence the expectations around data transfer, and they determine the set of events that will be supported.

- Establishment (Section 4.1.3) focuses on the actions that an application takes on the basic objects to prepare for data transfer.

- Data Transfer (Section 4.1.4) consists of how an application represents data to be sent and received, the functions required to send and receive that data, and how the application is notified of the status of its data transfer.

- Event Handling (Section 4.1.5) defines the set of properties about which an application can receive notifications during the lifetime of transport objects. Events can also provide opportunities for the application to interact with the underlying transport by querying state or updating maintenance options.

- Termination (Section 4.1.6) focuses on the methods by which data transmission is ceased, and state is torn down in the transport.

The diagram below provides a high-level view of the actions taken during the lifetime of a connection.

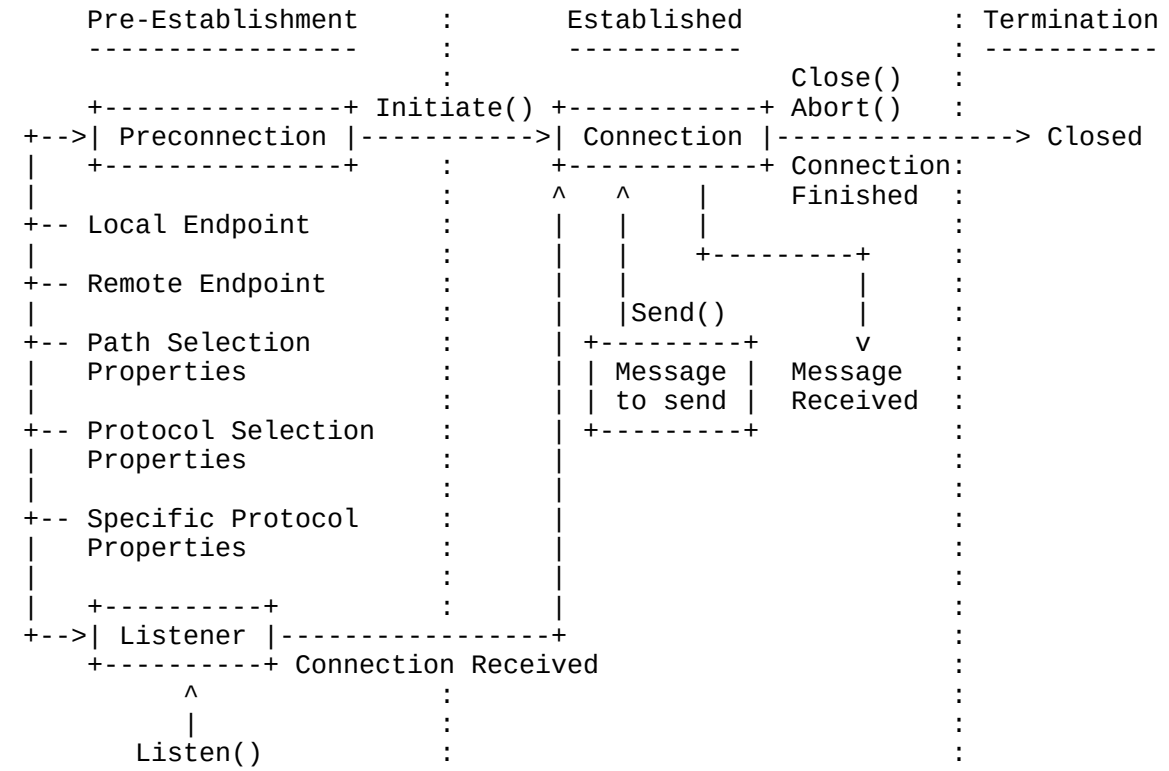


Figure 2: The lifetime of a connection

4.1.1. Basic Objects

- Preconnection: A Preconnection object is a representation of a potential connection. It has state that describes parameters of a Connection that might exist in the future: the Local Endpoint from which that Connection will be established, the Remote Endpoint to which it will connect, and Path Selection Properties, Protocol Selection Properties, and Specific Protocol Properties that influence the choice of transport that a Connection will use. A Preconnection can be fully specified and represent a single possible Connection, or it can be partially specified such that it represents a family of possible Connections. 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. The Remote Endpoint must be specified in the Preconnection is used to Initiate connections, but is optional if it is used to Listen for incoming connections.

- Connection: A Connection object represents an active transport protocol instance that can send and/or receive Messages between a Local Endpoint and a Remote Endpoint. It holds state pertaining to the underlying transport protocol instance and any ongoing data transfer. This represents, for example, an active connection in a connection-oriented protocol such as TCP, or a fully-specified 5-tuple for a connectionless protocol such as UDP.

- Listener: A Listener object accepts incoming transport protocol connections from Remote Endpoints and generates corresponding Connection objects. It is created from a Preconnection object that specifies the type of incoming connections it will accept.

4.1.2. Pre-Establishment

- Endpoint: An Endpoint represents one side of a transport connection. Endpoints can be Local Endpoints or Remote Endpoints, and respectively represent an identity that the application uses for the source or destination of a connection. Endpoint can vary in levels of specificity, and can be resolved to more concrete identities.

- Remote Endpoint: The Remote Endpoint represents the application's name for a peer that can participate in a transport connection. For example, the combination of a DNS name for the peer and a service name/port.

- Local Endpoint: The Local Endpoint represents the application's name for itself that it wants to use for transport connections. For example, a local IP address and port.

- Path Selection Properties: The Path Selection Properties consist of the options that an application may set to influence the selection of paths between itself and the Remote Endpoint. These options can come in the form of requirements, prohibitions, or preferences. Examples of options which may influence path selection include the interface type (such as a Wi-Fi Ethernet connection, or a Cellular LTE connection), characteristics of the path that are locally known like Maximum Transmission Unit (MTU) or discovered like Path MTU (PMTU), or predicted based on cached information like expected throughput or latency.

- Protocol Selection Properties: The Protocol Selection Properties consist of the options that an application may set to influence the selection of transport protocol, or to configure the behavior of generic transport protocol features. These options come in the form of requirements, prohibitions, and preferences. Examples

include reliability, service class, multipath support, and fast open support.

- Specific Protocol Properties: The Specific Protocol Properties refer to the subset of Protocol Properties options that apply to a single protocol (transport protocol, IP, or security protocol). The presence of such Properties does not necessarily require that a specific protocol must be used when a Connection is established, but that if this protocol is employed, a particular set of options should be used.

4.1.3. Establishment Actions

- Initiate is the primary action that an application can take to create a Connection to a remote endpoint, and prepare any required local or remote state to be able to send and/or receive Messages. For some protocols, this may initiate a server-to-client style handshake; for other protocols, this may just establish local state; and for peer-to-peer protocols, this may begin the process of a simultaneous open. The process of identifying options for connecting, such as resolution of the Remote Endpoint, occurs in response the Initiate call.

- Listen is the action of marking a Listener as willing to accept incoming Connections. The Listener will then create Connection objects as incoming connections are accepted (Section 4.1.5).

4.1.4. Data Transfer Objects and Actions

- Message: A Message object is a unit of data that can be represented as bytes that can be transferred between two endpoints over a transport connection. The bytes within a Message are assumed to be ordered within the Message. If an application does not care about the order in which a peer receives two distinct spans of bytes, those spans of bytes are considered independent Messages. Messages may or may not be usable if incomplete or corrupted. Boundaries of a Message may or may not be understood or transmitted by transport protocols. Specifically, what one application considers to be two Messages sent on a stream-based transport may be treated as a single Message by the application on the other side.

- Send is the action to transmit a Message or partial Message over a Connection to a Remote Endpoint. The interface to Send may include options specific to how the Message’s content is to be sent. Status of the Send operation may be delivered back to the application in an event (Section 4.1.5).

- Receive is an action that indicates that the application is ready to asynchronously accept a Message over a Connection from a Remote Endpoint, while the Message content itself will be delivered in an event (Section 4.1.5). The interface to Receive may include options specific to the Message that is to be delivered to the application.

4.1.5. Event Handling

This list of events that can be delivered to an application is not exhaustive, but gives the top-level categories of events. The API may expand this list.

- Connection Ready: Signals to an application that a given Connection is ready to send and/or receive Messages. If the Connection relies on handshakes to establish state between peers, then it is assumed that these steps have been taken.

- Connection Finished: Signals to an application that a given Connection is no longer usable for sending or receiving Messages. This should deliver an error to the application that describes the nature of the termination.

- Connection Received: Signals to an application that a given Listener has passively received a Connection.

- Message Received: Delivers received Message content to the application, based on a Receive action. This may include an error if the Receive action cannot be satisfied due to the Connection being closed.

- Message Sent: Notifies the application of the status of its Send action. This may be an error if the Message cannot be sent, or an indication that Message has been processed by the protocol stack.

- Path Properties Changed: Notifies the application that some property of the Connection has changed that may influence how and where data is sent and/or received.

4.1.6. Termination Actions

- Close is the action an application may take on a Connection to indicate that it no longer intends to send data, is no longer willing to receive data, and that the protocol should signal this state to the remote endpoint if applicable.

- Abort is an action the application may take on a Connection to indicate a Close, but with the additional indication that the

transport system should not attempt to deliver any outstanding data.

4.2. Transport System Implementation Concepts

The Transport System Implementation Concepts define the set of objects used internally to a system or library to provide the functionality of transport networking, as required by the abstract interface.

- Connection Group: A Connections Group is a set of Connections that share properties. For multiplexing transport protocols, the Connection Group defines the set of Connections that can be multiplexed together.

- Path: A Path represents an available set of properties of a network route on which packets may be sent or received.

- Protocol Instance: A Protocol Instance is a single instance of one protocol, including any state it has necessary to establish connectivity or send and receive Messages.

- Protocol Stack: A Protocol Stack is a set of Protocol Instances (including relevant application, security, transport, or Internet protocols) that are used together to establish connectivity or send and receive Messages. A single stack may be simple (a single transport protocol instance over IP), or complex (multiple application protocol streams going through a single security and transport protocol, over IP; or, a multi-path transport protocol over multiple transport sub-flows).

- System Policy: System Policy represents the input from an operating system or other global preferences that can constrain or influence how an implementation will gather candidate paths and protocols (Section 4.2.1) and race the candidates during establishment (Section 4.2.2). Specific aspects of the System Policy may apply to all Connections, or only certain ones depending on the runtime context and properties of the Connection.

- Cached State: Cached State is the state and history that the implementation keeps for each set of associated endpoints that have been used previously. This can include DNS results, TLS session state, previous success and quality of transport protocols over certain paths.

4.2.1. Gathering

- Path Selection: Path Selection represents the act of choosing one or more paths that are available to use based on the Path Selection Properties provided by the application, and a Transport Services system's policies and heuristics.

- Protocol Selection: Protocol Selection represents the act of choosing one or more sets of protocol options that are available to use based on the Protocol Properties provided by the application, and a Transport Services system's policies and heuristics.

4.2.2. Racing

- Protocol Option Racing: Protocol Racing is the act of attempting to establish, or scheduling attempts to establish, multiple Protocol Stacks that differ based on the composition of protocols or the options used for protocols.

- Path Racing: Path Racing is the act of attempting to establish, or scheduling attempts to establish, multiple Protocol Stacks that differ based on a selection from the available Paths.

- Endpoint Racing: Endpoint Racing is the act of attempting to establish, or scheduling attempts to establish, multiple Protocol Stacks that differ based on the specific representation of the Remote Endpoint and the Local Endpoint, such as IP addresses resolved from a DNS hostname.

5. IANA Considerations

RFC-EDITOR: Please remove this section before publication.

This document has no actions for IANA.

6. Security Considerations

The Transport Services architecture does not recommend use of specific security protocols or algorithms. Its goal is to offer ease of use for existing protocols by providing a generic security-related interface. Each provided interface mimics an existing protocol-specific interface provided by supported security protocols. For example, trust verification callbacks are common parts of TLS APIs. Transport Services APIs will expose similar functionality. Clients must take care to use security APIs appropriately. In cases where clients use said interface to provide sensitive keying material, e.g., access to private keys or copies of pre-shared keys (PSKs), key

use must be validated. For example, clients should not use PSK material created for ESP with IETF-QUIC, and clients must not use private keys intended for server authentication as a key for client authentication. Moreover, unlike certain transport features such as TFO or ECN which can fall back to standard configurations, Transport Services systems must not permit fallback for security protocols. For example, if a client requests TLS, yet TLS or the desired version are not available, its connection must fail. Clients are responsible for implementing protocol or version fallback using a Transport Services API if so desired.

7. 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).

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.

8. Informative References

[draft-brunstrom-taps-impl]
"Implementing Interfaces to Transport Services", n.d..

[draft-trammell-taps-interface]
"An Abstract Application Layer Interface to Transport Services", n.d..

[I-D.ietf-taps-minset]
Welzl, M. and S. Gjessing, "A Minimal Set of Transport Services for TAPS Systems", draft-ietf-taps-minset-01 (work in progress), February 2018.

[I-D.pauly-taps-transport-security]
Pauly, T., Rose, K., and C. Wood, "A Survey of Transport Security Protocols", draft-pauly-taps-transport-security-01 (work in progress), January 2018.

[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>.

Authors' Addresses

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

Email: tpauly@apple.com

Brian Trammell (editor)
ETH Zurich
Gloriastrasse 35
8092 Zurich
Switzerland

Email: ietf@trammell.ch

Anna Brunstrom
Karlstad University

Email: anna.brunstrom@kau.se

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/

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

Email: csp@csperkins.org

Philipp S. Tiesel
TU Berlin
Marchstrasse 23
10587 Berlin
Germany

Email: philipp@inet.tu-berlin.de

Chris Wood
Apple Inc.
One Apple Park Way
Cupertino, California 95014
United States of America

Email: cawood@apple.com

Network Working Group
Internet-Draft
Intended status: Informational
Expires: July 7, 2018

T. Pauly
Apple Inc.
K. Rose
Akamai Technologies, Inc.
C. Wood
Apple Inc.

January 03, 2018

A Survey of Transport Security Protocols
draft-pauly-taps-transport-security-01

Abstract

This document provides a survey of commonly used or notable network security protocols, with a focus on how they interact and integrate with applications and transport protocols. Its goal is to supplement efforts to define and catalog transport services [RFC8095] by describing the interfaces required to add security protocols. It examines Transport Layer Security (TLS), Datagram Transport Layer Security (DTLS), Quick UDP Internet Connections with TLS (QUIC + TLS), MinimalT, CurveCP, tcpcrypt, Internet Key Exchange with Encapsulating Security Protocol (IKEv2 + ESP), SRTP (with DTLS), and WireGuard. This survey is not limited to protocols developed within the scope or context of the IETF.

Status of This Memo

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

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

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

This Internet-Draft will expire on July 7, 2018.

Table of Contents

1. Introduction

This document provides a survey of commonly used or notable network security protocols, with a focus on how they interact and integrate with applications and transport protocols. Its goal is to supplement efforts to define and catalog transport services [RFC8095] by describing the interfaces required to add security protocols. It examines Transport Layer Security (TLS), Datagram Transport Layer Security (DTLS), Quick UDP Internet Connections with TLS (QUIC + TLS), MinimalT, CurveCP, tcpcrypt, Internet Key Exchange with Encapsulating Security Protocol (IKEv2 + ESP), SRTP (with DTLS), and WireGuard. This survey is not limited to protocols developed within the scope or context of the IETF.

For each protocol, this document provides a brief description, the security features it provides, and the dependencies it has on the underlying transport. This is followed by defining the set of transport security features shared by these protocols. Finally, we distill the application and transport interfaces provided by the transport security protocols.

2. Terminology

The following terms are used throughout this document to describe the roles and interactions of transport security protocols:

- Transport Feature: a specific end-to-end feature that the transport layer provides to an application. Examples include

confidentiality, reliable delivery, ordered delivery, message-versus-stream orientation, etc.

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

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

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

- Security Feature: a specific feature that a network security layer provides to applications. Examples include authentication, encryption, key generation, session resumption, and privacy. A feature may be considered to be Mandatory or Optional to an application's implementation.

- Security Protocol: a defined network protocol that implements one or more security features. Security protocols may be used alongside transport protocols, and in combination with one another when appropriate.

- Handshake Protocol: a security protocol that performs a handshake to validate peers and establish a shared cryptographic key.

- Record Protocol: a security protocol that allows data to be encrypted in records or datagrams based on a shared cryptographic key.

- Session: an ephemeral security association between applications.

- Connection: the shared state of two or more endpoints that persists across messages that are transmitted between these endpoints. A connection is a transient participant of a session, and a session generally lasts between connection instances.

- Connection Mobility: a property of a connection that allows it to be multihomed or resilient across network interface or address changes.

- Peer: an endpoint application party to a session.

- Client: the peer responsible for initiating a session.

- o Server: the peer responsible for responding to a session initiation.

3. Transport Security Protocol Descriptions

This section contains descriptions of security protocols that currently used to protect data being sent over a network.

For each protocol, we describe the features it provides and its dependencies on other protocols.

3.1. TLS

TLS (Transport Layer Security) [RFC5246] is a common protocol used to establish a secure session between two endpoints. Communication over this session "prevents eavesdropping, tampering, and message forgery." TLS consists of a tightly coupled handshake and record protocol. The handshake protocol is used to authenticate peers, negotiate protocol options, such as cryptographic algorithms, and derive session-specific keying material. The record protocol is used to marshal (possibly encrypted) data from one peer to the other. This data may contain handshake messages or raw application data.

3.1.1. Protocol Description

TLS is the composition of a handshake and record protocol [I-D.ietf-tls-tls13]. The record protocol is designed to marshal an arbitrary, in-order stream of bytes from one endpoint to the other. It handles segmenting, compressing (when enabled), and encrypting data into discrete records. When configured to use an AEAD algorithm, it also handles nonce generation and encoding for each record. The record protocol is hidden from the client behind a byte stream-oriented API.

The handshake protocol serves several purposes, including: peer authentication, protocol option (key exchange algorithm and ciphersuite) negotiation, and key derivation. Peer authentication may be mutual. However, commonly, only the server is authenticated. X.509 certificates are commonly used in this authentication step, though other mechanisms, such as raw public keys [RFC7250], exist. The client is not authenticated unless explicitly requested by the server with a CertificateRequest handshake message.

The handshake protocol is also extensible. It allows for a variety of extensions to be included by either the client or server. These extensions are used to specify client preferences, e.g., the application-layer protocol to be driven with the TLS connection [RFC7301], or signals to the server to aid operation, e.g., the

server name [RFC6066]. Various extensions also exist to tune the parameters of the record protocol, e.g., the maximum fragment length [RFC6066].

Alerts are used to convey errors and other atypical events to the endpoints. There are two classes of alerts: closure and error alerts. A closure alert is used to signal to the other peer that the sender wishes to terminate the connection. The sender typically follows a close alert with a TCP FIN segment to close the connection. Error alerts are used to indicate problems with the handshake or individual records. Most errors are fatal and are followed by connection termination. However, warning alerts may be handled at the discretion of each respective implementation.

Once a session is disconnected all session keying material must be torn down, unless resumption information was previously negotiated. TLS supports stateful and stateless resumption. (Here, the state refers to the information requirements for the server. It is assumed that the client must always store some state information in order to resume a session.)

3.1.2. Protocol Features

- Key exchange and ciphersuite algorithm negotiation.
- Stateful and stateless session resumption.
- Certificate- and raw public-key-based authentication.
- Mutual client and server authentication.
- Byte stream confidentiality and integrity.
- Extensibility via well-defined extensions.
- 0-RTT data support (in TLS 1.3 only).
- Application-layer protocol negotiation.
- Transparent data segmentation.

3.1.3. Protocol Dependencies

- TCP for in-order, reliable transport.
- (Optionally) A PKI trust store for certificate validation.

3.2. DTLS

DTLS (Datagram Transport Layer Security) [RFC6347] is based on TLS, but differs in that it is designed to run over UDP instead of TCP. Since UDP does not guarantee datagram ordering or reliability, DTLS modifies the protocol to make sure it can still provide the same security guarantees as TLS. DTLS was designed to be as close to TLS as possible, so this document will assume that all properties from TLS are carried over except where specified.

3.2.1. Protocol Description

DTLS is modified from TLS to account for packet loss and reordering that occur when operating over a datagram-based transport, i.e., UDP. Each message is assigned an explicit sequence number to be used to reorder on the receiving end. This removes the inter-record dependency and allows each record to be decrypt in isolation of the rest. However, DTLS does not deviate from TLS in that in still provides in-order delivery of data to the application.

With respect to packet loss, if one peer has sent a handshake message and has not yet received its expected response, it will retransmit the handshake message after a configurable timeout.

To account for long records that cannot fit within a single UDP datagram, DTLS supports fragmentation of records across datagrams, keeping track of fragment offsets and lengths in each datagram. The receiving peer must re-assemble records before decrypting.

DTLS relies on UDP’s port numbers to allow peers with multiple DTLS sessions between them to demultiplex ’streams’ of encrypted packets that share a single TLS session.

Since datagrams may be replayed, DTLS provides anti-replay detection based on a window of acceptable sequence numbers [RFC4303].

3.2.2. Protocol Features

- Anti-replay protection between datagrams.
- Basic reliability for handshake messages.
- See also the features from TLS.

3.2.3. Protocol Dependencies

- Since DTLS runs over an unreliable, unordered datagram transport, it does not require any reliability features.
- DTLS contains its own length, so although it runs over a datagram transport, it does not rely on the transport protocol supporting framing.
- UDP for port numbers used for demultiplexing.
- Path MTU discovery.

3.3. QUIC with TLS

QUIC (Quick UDP Internet Connections) is a new transport protocol that runs over UDP, and was originally designed with a tight integration with its security protocol and application protocol mappings. The QUIC transport layer itself provides support for data confidentiality and integrity. This requires keys to be derived with a separate handshake protocol. A mapping for QUIC over TLS 1.3 [I-D.ietf-quic-tls] has been specified to provide this handshake.

3.3.1. Protocol Description

Since QUIC integrates TLS with its transport, it relies on specific integration points between its security and transport sides. Specifically, these points are:

- Starting the handshake to generate keys and provide authentication (and providing the transport for the handshake).
- Client address validation.
- Key ready events from TLS to notify the QUIC transport.
- Exporting secrets from TLS to the QUIC transport.

The QUIC transport layer support multiple streams over a single connection. The first stream is reserved specifically for a TLS connection. The TLS handshake, along with further records, are sent over this stream. This TLS connection follows the TLS standards and inherits the security properties of TLS. The handshake generates keys, which are then exported to the rest of the QUIC connection, and are used to protect the rest of the streams.

Initial QUIC messages (packets) are encrypted using "fixed" keys derived from the QUIC version and public packet information

(Connection ID). Packets are later encrypted using keys derived from the TLS traffic secret upon handshake completion. The TLS 1.3 handshake for QUIC is used in either a single-RTT mode or a fast-open zero-RTT mode. When zero-RTT handshakes are possible, the encryption first transitions to use the zero-RTT keys before using single-RTT handshake keys after the next TLS flight.

3.3.2. Protocol Features

- Handshake properties of TLS.
- Multiple encrypted streams over a single connection without head-of-line blocking.
- Packet payload encryption and complete packet authentication (with the exception of the Public Reset packet, which is not authenticated).

3.3.3. Protocol Dependencies

- QUIC transport relies on UDP.
- QUIC transport relies on TLS 1.3 for authentication and initial key derivation.
- TLS within QUIC relies on a reliable stream abstraction for its handshake.

3.4. MinimalT

MinimalT is a UDP-based transport security protocol designed to offer confidentiality, mutual authentication, DoS prevention, and connection mobility [MinimalT]. One major goal of the protocol is to leverage existing protocols to obtain server-side configuration information used to more quickly bootstrap a connection. MinimalT uses a variant of TCP’s congestion control algorithm.

3.4.1. Protocol Description

MinimalT is a secure transport protocol built on top of a widespread directory service. Clients and servers interact with local directory services to (a) resolve server information and (b) public ephemeral state information, respectively. Clients connect to a local resolver once at boot time. Through this resolver they recover the IP address(es) and public key(s) of each server to which they want to connect.

Connections are instances of user-authenticated, mobile sessions between two endpoints. Connections run within tunnels between hosts. A tunnel is a server-authenticated container that multiplexes multiple connections between the same hosts. All connections in a tunnel share the same transport state machine and encryption. Each tunnel has a dedicated control connection used to configure and manage the tunnel over time. Moreover, since tunnels are independent of the network address information, they may be reused as both ends of the tunnel move about the network. This does however imply that the connection establishment and packet encryption mechanisms are coupled.

Before a client connects to a remote service, it must first establish a tunnel to the host providing or offering the service. Tunnels are established in 1-RTT using an ephemeral key obtained from the directory service. Tunnel initiators provide their own ephemeral key and, optionally, a DoS puzzle solution such that the recipient (server) can verify the authenticity of the request and derive a shared secret. Within a tunnel, new connections to services may be established.

3.4.2. Protocol Features

- 0-RTT forward secrecy for new connections.
- DoS prevention by client-side puzzles.
- Tunnel-based mobility.
- (Transport Feature) Connection multiplexing between hosts across shared tunnels.
- (Transport Feature) Congestion control state is shared across connections between the same host pairs.

3.4.3. Protocol Dependencies

- A DNS-like resolution service to obtain location information (an IP address) and ephemeral keys.
- A PKI trust store for certificate validation.

3.5. CurveCP

CurveCP [CurveCP] is a UDP-based transport security protocol from Daniel J. Bernstein. Unlike other transport security protocols, it is based entirely upon highly efficient public key algorithms. This

removes many pitfalls associated with nonce reuse and key synchronization.

3.5.1. Protocol Description

CurveCP is a UDP-based transport security protocol. It is built on three principal features: exclusive use of public key authenticated encryption of packets, server-chosen cookies to prohibit memory and computation DoS at the server, and connection mobility with a client-chosen ephemeral identifier.

There are two rounds in CurveCP. In the first round, the client sends its first initialization packet to the server, carrying its (possibly fresh) ephemeral public key C', with zero-padding encrypted under the server's long-term public key. The server replies with a cookie and its own ephemeral key S' and a cookie that is to be used by the client. Upon receipt, the client then generates its second initialization packet carrying: the ephemeral key C', cookie, and an encryption of C', the server's domain name, and, optionally, some message data. The server verifies the cookie and the encrypted payload and, if valid, proceeds to send data in return. At this point, the connection is established and the two parties can communicate.

The use of only public-key encryption and authentication, or "boxing", is done to simplify problems that come with symmetric key management and synchronization. For example, it allows the sender of a message to be in complete control of each message's nonce. It does not require either end to share secret keying material. And it allows ephemeral public keys to be associated with connections (or sessions).

The client and server do not perform a standard key exchange. Instead, in the initial exchange of packets, the each party provides its own ephemeral key to the other end. The client can choose a new ephemeral key for every new connection. However, the server must rotate these keys on a slower basis. Otherwise, it would be trivial for an attacker to force the server to create and store ephemeral keys with a fake client initialization packet.

Unlike TCP, the server employs cookies to enable source validation. After receiving the client's initial packet, encrypted under the server's long-term public key, the server generates and returns a stateless cookie that must be echoed back in the client's following message. This cookie is encrypted under the client's ephemeral public key. This stateless technique prevents attackers from hijacking client initialization packets to obtain cookie values to flood clients. (A client would detect the duplicate cookies and

reject the flooded packets.) Similarly, replaying the client's second packet, carrying the cookie, will be detected by the server.

CurveCP supports a weak form of client authentication. Clients are permitted to send their long-term public keys in the second initialization packet. A server can verify this public key and, if untrusted, drop the connection and subsequent data.

Unlike some other protocols, CurveCP data packets only leave the ephemeral public key, i.e., the connection ID, and the per-message nonce in the clear. Everything else is encrypted.

3.5.2. Protocol Features

- Forward-secure data encryption and authentication.
- Per-packet public-key encryption.
- 1-RTT session bootstrapping.
- Connection mobility based on a client-chosen ephemeral identifier.
- Connection establishment message padding to prevent traffic amplification.
- Sender-chosen explicit nonces, e.g., based on a sequence number.

3.5.3. Protocol Dependencies

- An unreliable transport protocol such as UDP.

3.6. tcpcrypt

Tcpcrypt is a lightweight extension to the TCP protocol to enable opportunistic encryption with hooks available to the application layer for implementation of endpoint authentication.

3.6.1. Protocol Description

Tcpcrypt extends TCP to enable opportunistic encryption between the two ends of a TCP connection [I-D.ietf-tcpinc-tcpcrypt]. It is a family of TCP encryption protocols (TEP), distinguished by key exchange algorithm. The use of a TEP is negotiated with a TCP option during the initial TCP handshake via the mechanism described by TCP Encryption Negotiation Option (ENO) [I-D.ietf-tcpinc-tcpeno]. In the case of initial session establishment, once a tcpcrypt TEP has been negotiated the key exchange occurs within the data segments of the first few packets exchanged after the handshake completes. The

initiator of a connection sends a list of supported AEAD algorithms, a random nonce, and an ephemeral public key share. The responder typically chooses a mutually-supported AEAD algorithm and replies with this choice, its own nonce, and ephemeral key share. An initial shared secret is derived from the ENO handshake, the tcpcrypt handshake, and the initial keying material resulting from the key exchange. The traffic encryption keys on the initial connection are derived from the shared secret. Connections can be re-keyed before the natural AEAD limit for a single set of traffic encryption keys is reached.

Each tcpcrypt session is associated with a ladder of resumption IDs, each derived from the respective entry in a ladder of shared secrets. These resumption IDs can be used to negotiate a stateful resumption of the session in a subsequent connection, resulting in use of a new shared secret and traffic encryption keys without requiring a new key exchange. Willingness to resume a session is signaled via the ENO option during the TCP handshake. Given the length constraints imposed by TCP options, unlike stateless resumption mechanisms (such as that provided by session tickets in TLS) resumption in tcpcrypt requires the maintenance of state on the server, and so successful resumption across a pool of servers implies shared state.

Owing to middlebox ossification issues, tcpcrypt only protects the payload portion of a TCP packet. It does not encrypt any header information, such as the TCP sequence number.

Tcpcrypt exposes a universally-unique connection-specific session ID to the application, suitable for application-level endpoint authentication either in-band or out-of-band.

3.6.2. Protocol Features

- Forward-secure TCP payload encryption and integrity protection.
- Session caching and address-agnostic resumption.
- Connection re-keying.
- Application-level authentication primitive.

3.6.3. Protocol Dependencies

- TCP
- TCP Encryption Negotiation Option (ENO)

3.7. IKEv2 with ESP

IKEv2 [RFC7296] and ESP [RFC4303] together form the modern IPsec protocol suite that encrypts and authenticates IP packets, either as for creating tunnels (tunnel-mode) or for direct transport connections (transport-mode). This suite of protocols separates out the key generation protocol (IKEv2) from the transport encryption protocol (ESP). Each protocol can be used independently, but this document considers them together, since that is the most common pattern.

3.7.1. Protocol descriptions

3.7.1.1. IKEv2

IKEv2 is a control protocol that runs on UDP port 500. Its primary goal is to generate keys for Security Associations (SAs). It first uses a Diffie-Hellman key exchange to generate keys for the "IKE SA", which is a set of keys used to encrypt further IKEv2 messages. It then goes through a phase of authentication in which both peers present blobs signed by a shared secret or private key, after which another set of keys is derived, referred to as the "Child SA". These Child SA keys are used by ESP.

IKEv2 negotiates which protocols are acceptable to each peer for both the IKE and Child SAs using "Proposals". Each proposal may contain an encryption algorithm, an authentication algorithm, a Diffie-Hellman group, and (for IKE SAs only) a pseudorandom function algorithm. Each peer may support multiple proposals, and the most preferred mutually supported proposal is chosen during the handshake.

The authentication phase of IKEv2 may use Shared Secrets, Certificates, Digital Signatures, or an EAP (Extensible Authentication Protocol) method. At a minimum, IKEv2 takes two round trips to set up both an IKE SA and a Child SA. If EAP is used, this exchange may be expanded.

Any SA used by IKEv2 can be rekeyed upon expiration, which is usually based either on time or number of bytes encrypted.

There is an extension to IKEv2 that allows session resumption [RFC5723].

MOBIKE is a Mobility and Multihoming extension to IKEv2 that allows a set of Security Associations to migrate over different addresses and interfaces [RFC4555].

When UDP is not available or well-supported on a network, IKEv2 may be encapsulated in TCP [I-D.ietf-ipsecme-tcp-encaps].

3.7.1.2. ESP

ESP is a protocol that encrypts and authenticates IP and IPv6 packets. The keys used for both encryption and authentication can be derived from an IKEv2 exchange. ESP Security Associations come as pairs, one for each direction between two peers. Each SA is identified by a Security Parameter Index (SPI), which is marked on each encrypted ESP packet.

ESP packets include the SPI, a sequence number, an optional Initialization Vector (IV), payload data, padding, a length and next header field, and an Integrity Check Value.

From [RFC4303], "ESP is used to provide confidentiality, data origin authentication, connectionless integrity, an anti-replay service (a form of partial sequence integrity), and limited traffic flow confidentiality."

Since ESP operates on IP packets, it is not directly tied to the transport protocols it encrypts. This means it requires little or no change from transports in order to provide security.

ESP packets are sent directly over IP, except when a NAT is present, in which case they are sent on UDP port 4500, or via TCP encapsulation [I-D.ietf-ipsecme-tcp-encaps].

3.7.2. Protocol features

3.7.2.1. IKEv2

- Encryption and authentication of handshake packets.
- Cryptographic algorithm negotiation.
- Session resumption.
- Mobility across addresses and interfaces.
- Peer authentication extensibility based on Shared Secret, Certificates, Digital Signatures, or EAP methods.

3.7.2.2. ESP

- Data confidentiality and authentication.
- Connectionless integrity.
- Anti-replay protection.
- Limited flow confidentiality.

3.7.3. Protocol dependencies

3.7.3.1. IKEv2

- Availability of UDP to negotiate, or implementation support for TCP-encapsulation.
- Some EAP authentication types require accessing a hardware device, such as a SIM card; or interacting with a user, such as password prompting.

3.7.3.2. ESP

- Since ESP is below transport protocols, it does not have any dependencies on the transports themselves, other than on UDP or TCP for NAT traversal.

3.8. WireGuard

WireGuard is a layer 3 protocol designed to complement or replace IPsec [WireGuard]. Unlike most transport security protocols, which rely on PKI for peer authentication, WireGuard authenticates peers using pre-shared public keys delivered out-of-band, each of which is bound to one or more IP addresses. Moreover, as a protocol suited for VPNs, WireGuard offers no extensibility, negotiation, or cryptographic agility.

3.8.1. Protocol description

WireGuard is a simple VPN protocol that binds a pre-shared public key to one or more IP addresses. Users configure WireGuard by associating peer public keys with IP addresses. These mappings are stored in a CryptoKey Routing Table. (See Section 2 of [WireGuard] for more details and sample configurations.) These keys are used upon WireGuard packet transmission and reception. For example, upon receipt of a Handshake Initiation message, receivers use the static public key in their CryptoKey routing table to perform necessary cryptographic computations.

WireGuard builds on Noise [Noise] for 1-RTT key exchange with identity hiding. The handshake hides peer identities as per the SIGMA construction [SIGMA]. As a consequence of using Noise, WireGuard comes with a fixed set of cryptographic algorithms:

- x25519 [Curve25519] and HKDF [RFC5869] for ECDH and key derivation.
- ChaCha20+Poly1305 [RFC7539] for packet authenticated encryption.
- BLAKE2s [BLAKE2] for hashing.

There is no cryptographic agility. If weaknesses are found in any of these algorithms, new message types using new algorithms must be introduced.

WireGuard is designed to be entirely stateless, modulo the CryptoKey routing table, which has size linear with the number of trusted peers. If a WireGuard receiver is under heavy load and cannot process a packet, e.g., cannot spare CPU cycles for point multiplication, it can reply with a cookie similar to DTLS and IKEv2. This cookie only proves IP address ownership. Any rate limiting scheme can be applied to packets coming from non-spoofed addresses.

3.8.2. Protocol features

- Optional PSK-based session creation.
- Mutual client and server authentication.
- Stateful, timestamp-based replay prevention.
- Cookie-based DoS mitigation similar to DTLS and IKEv2.

3.8.3. Protocol dependencies

- Datagram transport.
- Out-of-band key distribution and management.

3.9. SRTP (with DTLS)

SRTP - Secure RTP - is a profile for RTP that provides confidentiality, message authentication, and replay protection for data and control packets [RFC3711]. SRTP packets are encrypted using a session key, which is derived from a separate master key. Master keys are derived and managed externally, e.g., via DTLS, as specified in RFC 5736 [RFC5763].

3.9.1. Protocol descriptions

SRTP adds confidentiality and, optionally, integrity protection to SRTP packets. This is done by encrypting RTP payloads and optionally appending an authentication tag (MAC) to the packet trailer. Packets are encrypted using session keys, which are ultimately derived from a master key and some additional master salt and session salt. SRTP packets carry a 2-byte sequence number to partially identify the unique packet index. SRTP peers maintain a separate rollover counter (ROC) that is incremented whenever the sequence number wraps. The sequence number and ROC together determine the packet index. Packets also carry

Numerous encryption modes are supported. For popular modes of operation, e.g., AES-CTR, The (unique) initialization vector (IV) used for each encryption mode is a function of the RTP SSRC (synchronization source), packet index, and session "salting key".

SRTP offers replay detection by keeping a Replay List of already seen and processed packet indices. If a packet arrives with an index that matches one in the Replay List, it is silently discarded.

DTLS [RFC5764] is commonly used as a way to perform mutually authentication key establishment for SRTP [RFC5763]. (Here, certificates marshall public keys between endpoints. Thus, self-signed certificates may be used if peers do not mutually trust one another, as is common on the Internet.) When DTLS is used, certificate fingerprints are transmitted out-of-band using SIP. Peers typically verify that DTLS-offered certificates match that which are offered over SIP. This prevents active attacks on RTP, but not on the signalling (SIP) channel.

3.9.2. Protocol features

- Optional replay protection with tunable replay windows.
- Out-of-order packet receipt.
- (RFC5763) Mandatory mutually authenticated key exchange.

3.9.3. Protocol dependencies

- External key derivation and management mechanism or protocol, e.g., DTLS [RFC5763].

4. Common Transport Security Features

There exists a common set of features shared across the transport protocols surveyed in this document. The mandatory features should be provided by any transport security protocol, while the optional features are extensions that a subset of the protocols provide. For clarity, we also distinguish between handshake and record features.

4.1. Mandatory Features

4.1.1. Handshake

- Forward-secure segment encryption and authentication: Transit data must be protected with an authenticated encryption algorithm.
- Private key interface or injection: Authentication based on public key signatures is commonplace for many transport security protocols.
- Endpoint authentication: The endpoint (receiver) of a new connection must be authenticated before any data is sent to said party.
- Source validation: Source validation must be provided to mitigate server-targeted DoS attacks. This can be done with puzzles or cookies.

4.1.2. Record

- Pre-shared key support: A record protocol must be able to use a pre-shared key established out-of-band to encrypt individual messages, packets, or datagrams.

4.2. Optional Features

4.2.1. Handshake

- Mutual authentication: Transport security protocols should allow both endpoints to authenticate one another if needed.
- Application-layer feature negotiation: The type of application using a transport security protocol often requires features configured at the connection establishment layer, e.g., ALPN [RFC7301]. Moreover, application-layer features may often be used to offload the session to another server which can better handle the request. (The TLS SNI is one example of such a feature.) As such, transport security protocols should provide a generic

mechanism to allow for such application-specific features and options to be configured or otherwise negotiated.

- Configuration extensions: The protocol negotiation should be extensible with addition of new configuration options.

- Session caching and management: Sessions should be cacheable to enable reuse and amortize the cost of performing session establishment handshakes.

4.2.2. Record

- Connection mobility: Sessions should not be bound to a network connection (or 5 tuple). This allows cryptographic key material and other state information to be reused in the event of a connection change. Examples of this include a NAT rebinding that occurs without a client's knowledge.

5. Transport Security Protocol Interfaces

This section describes the interface surface exposed by the security protocols described above, with each interface. Note that not all protocols support each interface.

5.1. Configuration Interfaces

Configuration interfaces are used to configure the security protocols before a handshake begins or the keys are negotiated.

- Identity and Private Keys
  The application can provide its identities (certificates) and private keys, or mechanisms to access these, to the security protocol to use during handshakes.
  Protocols: TLS, DTLS, QUIC + TLS, MinimalT, CurveCP, IKEv2, WireGuard, SRTP

- Supported Algorithms (Key Exchange, Signatures and Ciphersuites)
  The application can choose the algorithms that are supported for key exchange, signatures, and ciphersuites.
  Protocols: TLS, DTLS, QUIC + TLS, MinimalT, tcpcrypt, IKEv2, SRTP

- Session Cache
  The application provides the ability to save and retrieve session state (tickets, keying material, server parameters) that may be used to resume the security session.
  Protocols: TLS, DTLS, QUIC + TLS, MinimalT

- Authentication Delegate

The application provides access to a separate module that will provide authentication, using EAP for example.
Protocols: IKEv2, SRTP

5.2. Handshake Interfaces

Handshake interfaces are the points of interaction between a handshake protocol and the application, record protocol, and transport once the handshake is active.

- Send Handshake Messages
  The handshake protocol needs to be able to send messages over a transport to the remote peer to establish trust and negotiate keys.
  Protocols: All (TLS, DTLS, QUIC + TLS, MinimalT, CurveCP, IKEv2, WireGuard, SRTP (DTLS))

- Receive Handshake Messages
  The handshake protocol needs to be able to receive messages from the remote peer over a transport to establish trust and negotiate keys.
  Protocols: All (TLS, DTLS, QUIC + TLS, MinimalT, CurveCP, IKEv2, WireGuard, SRTP (DTLS))

- Identity Validation
  During a handshake, the security protocol will conduct identity validation of the peer. This can call into the application to offload validation. Protocols: All (TLS, DTLS, QUIC + TLS, MinimalT, CurveCP, IKEv2, WireGuard, SRTP (DTLS))

- Source Address Validation
  The handshake protocol may delegate validation of the remote peer that has sent data to the transport protocol or application. This involves sending a cookie exchange to avoid DoS attacks.
  Protocols: QUIC + TLS, DTLS, WireGuard

- Key Update
  The handshake protocol may be instructed to update its keying material, either by the application directly or by the record protocol sending a key expiration event.
  Protocols: TLS, DTLS, QUIC + TLS, MinimalT, tcpcrypt, IKEv2

- Pre-Shared Key Export
  The handshake protocol will generate one or more keys to be used for record encryption/decryption and authentication. These may be explicitly exportable to the application, traditionally limited to direct export to the record protocol, or inherently non-exportable

because the keys must be used directly in conjunction with the record protocol.

* Explict export: TLS (for QUIC), tcpcrypt, IKEv2, DTLS (for SRTP)

* Direct export: TLS, DTLS, MinimalT

* Non-exportable: CurveCP

5.3. Record Interfaces

Record interfaces are the points of interaction between a record protocol and the application, handshake protocol, and transport once in use.

- Pre-Shared Key Import
  Either the handshake protocol or the application directly can supply pre-shared keys for the record protocol use for encryption/decryption and authentication. If the application can supply keys directly, this is considered explicit import; if the handshake protocol traditionally provides the keys directly, it is considered direct import; if the keys can only be shared by the handshake, they are considered non-importable.

  * Explict import: QUIC, ESP

  * Direct import: TLS, DTLS, MinimalT, tcpcrypt, WireGuard

  * Non-importable: CurveCP

- Encrypt application data
  The application can send data to the record protocol to encrypt it into a format that can be sent on the underlying transport. The encryption step may require that the application data is treated as a stream or as datagrams, and that the transport to send the encrypted records present a stream or datagram interface.

  * Stream-to-Stream Protocols: TLS, tcpcrypt

  * Datagram-to-Datagram Protocols: DTLS, ESP, SRTP, WireGuard

  * Stream-to-Datagram Protocols: QUIC ((Editor’s Note: This depends on the interface QUIC exposes to applications.))

- Decrypt application data
  The application can receive data from its transport to be decrypted using record protocol. The decryption step may require

that the incoming transport data is presented as a stream or as datagrams, and that the resulting application data is a stream or datagrams.

* Stream-to-Stream Protocols: TLS, tcpcrypt

* Datagram-to-Datagram Protocols: DTLS, ESP, SRTP, WireGuard

* Datagram-to-Stream Protocols: QUIC ((Editor's Note: This depends on the interface QUIC exposes to applications.))

- Key Expiration
  The record protocol can signal that its keys are expiring due to reaching a time-based deadline, or a use-based deadline (number of bytes that have been encrypted with the key). This interaction is often limited to signaling between the record layer and the handshake layer.
  Protocols: ESP ((Editor's note: One may consider TLS/DTLS to also have this interface))

- Transport mobility
  The record protocol can be signaled that it is being migrated to another transport or interface due to connection mobility, which may reset address and state validation.
  Protocols: QUIC, MinimalT, CurveCP, ESP, WireGuard (roaming)

6. IANA Considerations

This document has on request to IANA.

7. Security Considerations

This document summarizes existing transport security protocols and their interfaces. It does not propose changes to or recommend usage of reference protocols.

8. Acknowledgments

The authors would like to thank Mirja Kuehlewind, Brian Trammell, Yannick Sierra, Frederic Jacobs, and Bob Bradley for their input and feedback on earlier versions of this draft.

9. Normative References

[BLAKE2] "BLAKE2 -- simpler, smaller, fast as MD5", n.d..

[Curve25519] "Curve25519 - new Diffie-Hellman speed records", n.d..

[CurveCP] "CurveCP -- Usable security for the Internet", n.d..

[I-D.ietf-ipsecme-tcp-encaps]
Pauly, T., Touati, S., and R. Mantha, "TCP Encapsulation of IKE and IPsec Packets", draft-ietf-ipsecme-tcp-encaps-10 (work in progress), May 2017.

[I-D.ietf-quic-tls]
Thomson, M. and S. Turner, "Using Transport Layer Security (TLS) to Secure QUIC", draft-ietf-quic-tls-08 (work in progress), December 2017.

[I-D.ietf-quic-transport]
Iyengar, J. and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport", draft-ietf-quic-transport-08 (work in progress), December 2017.

[I-D.ietf-tcpinc-tcpcrypt]
Bittau, A., Giffin, D., Handley, M., Mazieres, D., Slack, Q., and E. Smith, "Cryptographic protection of TCP Streams (tcpcrypt)", draft-ietf-tcpinc-tcpcrypt-11 (work in progress), November 2017.

[I-D.ietf-tcpinc-tcpeno]
Bittau, A., Giffin, D., Handley, M., Mazieres, D., and E. Smith, "TCP-ENO: Encryption Negotiation Option", draft-ietf-tcpinc-tcpeno-18 (work in progress), November 2017.

[I-D.ietf-tls-tls13]
Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", draft-ietf-tls-tls13-22 (work in progress), November 2017.

[MinimalT]
"MinimaLT -- Minimal-latency Networking Through Better Security", n.d..

[Noise] "The Noise Protocol Framework", n.d..

[RFC3711] Baugher, M., McGrew, D., Naslund, M., Carrara, E., and K. Norrman, "The Secure Real-time Transport Protocol (SRTP)", RFC 3711, DOI 10.17487/RFC3711, March 2004, <https://www.rfc-editor.org/info/rfc3711>.

[RFC4303] Kent, S., "IP Encapsulating Security Payload (ESP)", RFC 4303, DOI 10.17487/RFC4303, December 2005, <https://www.rfc-editor.org/info/rfc4303>.

[RFC4555] Eronen, P., "IKEv2 Mobility and Multihoming Protocol (MOBIKE)", RFC 4555, DOI 10.17487/RFC4555, June 2006, <https://www.rfc-editor.org/info/rfc4555>.

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, DOI 10.17487/RFC5246, August 2008, <https://www.rfc-editor.org/info/rfc5246>.

[RFC5723] Sheffer, Y. and H. Tschofenig, "Internet Key Exchange Protocol Version 2 (IKEv2) Session Resumption", RFC 5723, DOI 10.17487/RFC5723, January 2010, <https://www.rfc-editor.org/info/rfc5723>.

[RFC5763] Fischl, J., Tschofenig, H., and E. Rescorla, "Framework for Establishing a Secure Real-time Transport Protocol (SRTP) Security Context Using Datagram Transport Layer Security (DTLS)", RFC 5763, DOI 10.17487/RFC5763, May 2010, <https://www.rfc-editor.org/info/rfc5763>.

[RFC5764] McGrew, D. and E. Rescorla, "Datagram Transport Layer Security (DTLS) Extension to Establish Keys for the Secure Real-time Transport Protocol (SRTP)", RFC 5764, DOI 10.17487/RFC5764, May 2010, <https://www.rfc-editor.org/info/rfc5764>.

[RFC5869] Krawczyk, H. and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", RFC 5869, DOI 10.17487/RFC5869, May 2010, <https://www.rfc-editor.org/info/rfc5869>.

[RFC6066] Eastlake 3rd, D., "Transport Layer Security (TLS) Extensions: Extension Definitions", RFC 6066, DOI 10.17487/RFC6066, January 2011, <https://www.rfc-editor.org/info/rfc6066>.

[RFC6347] Rescorla, E. and N. Modadugu, "Datagram Transport Layer Security Version 1.2", RFC 6347, DOI 10.17487/RFC6347, January 2012, <https://www.rfc-editor.org/info/rfc6347>.

[RFC7250] Wouters, P., Ed., Tschofenig, H., Ed., Gilmore, J., Weiler, S., and T. Kivinen, "Using Raw Public Keys in Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)", RFC 7250, DOI 10.17487/RFC7250, June 2014, <https://www.rfc-editor.org/info/rfc7250>.

[RFC7296] Kaufman, C., Hoffman, P., Nir, Y., Eronen, P., and T. Kivinen, "Internet Key Exchange Protocol Version 2 (IKEv2)", STD 79, RFC 7296, DOI 10.17487/RFC7296, October 2014, <https://www.rfc-editor.org/info/rfc7296>.

[RFC7301] Friedl, S., Popov, A., Langley, A., and E. Stephan, "Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension", RFC 7301, DOI 10.17487/RFC7301, July 2014, <https://www.rfc-editor.org/info/rfc7301>.

[RFC7539] Nir, Y. and A. Langley, "ChaCha20 and Poly1305 for IETF Protocols", RFC 7539, DOI 10.17487/RFC7539, May 2015, <https://www.rfc-editor.org/info/rfc7539>.

[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>.

[SIGMA] "SIGMA -- The 'SIGn-and-MAc' Approach to Authenticated Diffie-Hellman and Its Use in the IKE-Protocols", n.d..

[WireGuard] "WireGuard -- Next Generation Kernel Network Tunnel", n.d..

Authors' Addresses

Tommy Pauly
Apple Inc.
1 Infinite Loop
Cupertino, California 95014
United States of America

Email: tpauly@apple.com

Kyle Rose
Akamai Technologies, Inc.
150 Broadway
Cambridge, MA 02144
United States of America

Email: krose@krose.org

Christopher A. Wood
Apple Inc.
1 Infinite Loop
Cupertino, California 95014
United States of America

Email: cawood@apple.com

TAPS Working Group
Internet-Draft
Intended status: Informational
Expires: September 2, 2018

B. Trammell, Ed.
ETH Zurich
M. Welzl, Ed.
University of Oslo
T. Enghardt
TU Berlin
G. Fairhurst
University of Aberdeen
M. Kuehlewind
ETH Zurich
C. Perkins
University of Glasgow
P. Tiesel
TU Berlin
C. Wood
Apple Inc.

March 01, 2018

An Abstract Application Layer Interface to Transport Services
draft-trammell-taps-interface-00

Abstract

This document describes an abstract programming interface 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 lowest common denominator interface to the transport layer, in an environment where endpoints have multiple interfaces and potential transport protocols to select from.

This Internet-Draft will expire on September 2, 2018.

Table of Contents

1. Introduction

The BSD Unix Sockets API's SOCK_STREAM abstraction, by bringing network sockets into the UNIX programming model, allowing anyone who knew how to write programs that dealt with sequential-access files to also write network applications, was a revolution in simplicity. It would not be an overstatement to say that this simple API is the reason the Internet won the protocol wars of the 1980s. SOCK_STREAM is tied to the Transmission Control Protocol (TCP), specified in 1981 [RFC0793]. TCP has scaled remarkably well over the past three and a half decades, but its total ubiquity has hidden an uncomfortable fact: the network is not really a file, and stream abstractions are too simplistic for many modern application programming models.

In the meantime, the nature of Internet access, and the variety of Internet transport protocols, is evolving. The challenges that new protocols and access paradigms present to the sockets API and to programming models based on them inspire the design principles of a new approach, which we outline in Section 3.

As a first step to realizing this design, [TAPS-ARCH] describes a high-level architecture for transport services. This document builds a modern abstract programming interface atop this architecture, deriving specific path and protocol selection properties and supported transport features from the analysis provided in [RFC8095] and [I-D.ietf-taps-minset].

2. Terminology and Notation

This API is described in terms of Objects, which an application can interact with; 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 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(parameter, parameter, ...) / Event<parameter, parameter, ...> ~~~

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

How these abstract concepts map into concrete implementations of this API in a given language on a given platform is largely dependent 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 callback passing or other asynchronous calling conventions. The method for registering callbacks and handlers is left as an implementation detail, with the caveat that the interface for receiving Messages must require the application to invoke the Connection.Receive() Action once per Message to be received (see Section 8).

This specification treats Events and errors similarly, as 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 which can be immediately detected, such as inconsistency in transport parameters.

3. Interface Design Principles

We begin with the architectural design principles defined in [TAPS-ARCH]; from these, we derive and elaborate a set of principles on which the design of the interface is based. 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;

- Explicit support for security properties as first-order transport features, and for long-term caching of cryptographic identities and parameters for associations among endpoints;

- Asynchronous Connection establishment, transmission, and reception, allowing most application interactions with the transport layer to be Event-driven, in line with developments in modern platforms and programming languages;

- 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; and

- Atomic transmission of data, using application-assisted framing and deframing where the underlying transport does not provide these.

4. API Summary

The Transport Services Interface is the basic common abstract application programming interface to the Transport Services Architecture defined in [TAPS-ARCH]. An application primarily interacts with this interface through two Objects, Preconnections and Connections. A Preconnection represents a set of parameters 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 received from a remote endpoint. 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 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 deframing callback which finds message boundaries in a stream. Messages are received asynchronously through a callback registered by the application. Errors and other notifications also happen asynchronously on the Connection.

In the following sections, we describe the details of application interaction with Objects through Actions and Events in each phase of a Connection, following the phases described in [TAPS-ARCH].

5. Pre-Establishment Phase

The pre-establishment phase allows applications to specify parameters for the Connections they're 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 parameters 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 transport parameters (see Section 5.2), and the security parameters (see Section 5.3):

```
Preconnection := NewPreconnection(LocalEndpoint,
                                  RemoteEndpoint,
                                  TransportParams,
                                  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. The Remote Endpoint MUST be specified in 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.

Framers (see Section 7.2) and deframers (see Section 8.1), if necessary, should be bound to the Preconnection during pre-establishment.

Preconnections, as Connections, can be cloned, in order to establish Connection groups before Connection initiation; see Section 6.4 for details.

5.1. Specifying Endpoints

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

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

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

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

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

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

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 RemoteEndoint. 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. The multiple identifiers refer to the same endpoint.

The transport services API will resolve names internally, when the Initiate(), Listen(), or Rendezvous() method is called establish a Connection. The API does not need the application to resolve names, and premature name resolution can damage performance by limiting the scope for alternate path discovery during Connection establishment. The Resolve() method is, however, provided to resolve a Local Endpoint or a Remote Endpoint in cases where this is required, for example with some NAT traversal protocols (see Section 6.3).

5.2. Specifying Transport Parameters

A Preconnection Object holds parameters reflecting the application’s requirements and preferences for the transport. These include protocol and path selection parameters, as well as Generic and Specific Protocol Properties for configuration of the detailed operation of the selected Protocol Stacks.

All Transport Parameters are organized within a single namespace shared with Send Parameters (see Section 7.1). All transport

parameters take paremeter-specific values. Protocol and Path Selection properties additionally take one of five preference levels, though not all preference levels make sense with all such properties. Note that it is possible for a set of specified transport parameters to be internally inconsistent, or for preferences to be inconsistent with the later use of the API by the application. Application developers should reduce inconsistency by only using the most stringent preference levels when failure to meet a preference would break the application's functionality (e.g. the Reliable Data Transfer preference, which is a core assumption of many application protocols). Implementations of this interface should also raise errors in configuration as early as possible, to help ensure these inconsistencies are caught early in the development process.

The protocol(s) and path(s) selected as candidates during Connection establishment are determined by a set of 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.

To reflect the needs of an individual Connection, they can be specified with five different preference levels:

Preference	Effect
Require	Select only protocols/paths providing the property, fail otherwise
Prefer	Prefer protocols/paths providing the property, proceed otherwise
Ignore	Cancel any default preference for this property
Avoid	Prefer protocols/paths not providing the property, proceed otherwise
Prohibit	Select only protocols/paths not providing the property, fail otherwise

An implementation of this interface must provide sensible defaults for protocol and path selection properties. The defaults given for each property below represent a configuration that can be implemented over TCP. An alternate set of default Protocol Selection Properties would represent a configuration that can be implemented over UDP.

The following properties can be used during Protocol and Path selection:

- Reliable Data Transfer: This boolean property specifies whether the application needs the transport protocol to ensure that data is received completely and without corruption on the other side. This also entails being notified when a Connection is closed or aborted. This property applies to Connections and Connection Groups. This is a strict requirement. The default is to enable Reliable Data Transfer.

- Preservation of data ordering: This boolean property specifies whether the application needs the transport protocol to assure that data is received by the application on the other end in the same order as it was sent. This property applies to Connections and Connection Groups. This is a strict requirement. The default is to preserve data ordering.

- Configure reliability on a per-Message basis: This boolean 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. This is not a strict requirement. The default is to not have this option.

- Use 0-RTT session establishment with an idempotent Message: This boolean 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. See also Section 7.1.4. This is a strict requirement. The default is to not have this option.

- Multiplex Connections: This boolean property specifies that the application would prefer multiple Connections between the same endpoints within a Connection Group to be multiplexed onto a single underlying transport connection where possible, for reasons of efficiency. This is not a strict requirement. The default is to not have this option.

- Notification of excessive retransmissions: This boolean property specifies whether an application considers it useful to be informed in case sent data was retransmitted more often than a certain threshold. This property applies to Connections and Connection Groups. This is not a strict requirement. The default is to have this option.

- Notification of ICMP error message arrival: This boolean property specifies whether an application considers it useful to be informed when an ICMP error message arrives. This property applies to Connections and Connection Groups. This is not a strict requirement. The default is to have this option.

- Control checksum coverage on sending or receiving: This boolean property specifies whether the application considers it useful to enable / disable / configure a checksum when sending data, or decide whether to require a checksum or not when receiving data. This property applies to Connections and Connection Groups. This is not a strict requirement, as it signifies a reduction in reliability. The default is full checksum coverage without being able to change it, and requiring a checksum when receiving.

- Interface Type: This enumerated property specifies which kind of access network interface, e.g., WiFi, Ethernet, or LTE, to prefer over others for this Connection, in case they are available. In general, Interface Types should be used only with the "Prefer" and "Prohibit" preference level. Specifically, using the "Require" preference level for Interface Type may limit path selection in a way that is detrimental to connectivity. The default is to use the default interface configured in the system policy.

- Capacity Profile: This enumerated property specifies the application's expectation of the dominating traffic pattern for this Connection. The Capacity Profile should only be used with the "Prefer" preference level; other preference levels make no sense for profiles. The following values are valid for Capacity Profile:

  Default: The application makes no representation about its expected capacity profile. No special optimizations of the tradeoff between delay, delay variation, and bandwidth efficiency should be made when selecting and configuring stacks.

  Interactive/Low Latency: The application is interactive. Response time (latency) should be optimized at the expense of bandwidth efficiency and delay variation. This can be used by the system to disable the coalescing of multiple small Messages into larger packets (Nagle's algorithm), to prefer lower-latency paths, signal a preference for lower-latency, higher-loss treatment, and so on.

  Constant Rate: The application expects to send/receive data at a constant rate after Connection establishment. Delay and delay variation should be optimized at the expense of bandwidth

efficiency. 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.

Scavenger/Bulk: The application is not interactive. It expects to send/receive a large amount of data, without any urgency. This can be used to select protocol stacks with scavenger transmission control, to signal a preference for less-than-best-effort treatment, and so on.

In addition to protocol and path selection properties, the transport parameters may also contain Generic and/or Specific Protocol Properties (see Section 9.1). These properties will be passed to the selected candidate Protocol Stack(s) to configure them before candidate Connection establishment.

5.2.1. Transport Parameters Object

All transport parameters used in the pre-establishment phase are collected in a TransportParameters Object that is passed to the Preconnection Object.

```
TransportParameters := NewTransportParameters()
```

The Individual parameters are then added to the TransportParameters Object. While Protocol Properties use the "add" call, Transport Preferences use special calls for the levels defined in Section 5.2.

```
TransportParameters.Add(parameter, value)

TransportParameters.Require(preference)
TransportParameters.Prefer(preference)
TransportParameters.Ignore(preference)
TransportParameters.Avoid(preference)
TransportParameters.Prohibit(preference)
```

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

```
TransportParameters := Connection.GetTransportParameters()
```

Note that most properties are only considered for Connection establishment and can not be changed after a Connection is established; however, they can be queried. See Section 9.

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

5.3. Specifying Security Parameters and Callbacks

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.pauly-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 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 HSMs, handshake callbacks MUST be used. See below for details.)

```
SecurityParameters.AddIdentity(identity)
SecurityParameters.AddPrivateKey(privateKey, publicKey)
```

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

```
SecurityParameters.AddSupportedGroup(22)    // secp256k1
SecurityParameters.AddCiphersuite(0xCCA9)   // TLS_ECDHE_ECDSA_WITH_CHACHA20_POL
Y1305_SHA256
SecurityParameters.AddSignatureAlgorithm(7) // ed25519
```

- Session cache: Used to tune cache capacity, lifetime, re-use, and eviction policies, e.g., LRU or FIFO.

```
SecurityParameters.SetSessionCacheCapacity(1024)     // 1024 elements
SecurityParameters.SetSessionCacheLifetime(24*60*60) // 24 hours
SecurityParameters.SetSessionCacheReuse(1)           // One-time use
```

- Pre-shared keying material: 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.AddPreSharedKey(key, identity)
```

Security decisions, especially pertaining to trust, are not static. Thus, once configured, parameters must also be supplied during live handshakes. These are best handled as client-provided callbacks. Security handshake callbacks include:

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

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

- 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)
```

Like transport parameters, security parameters are inherited during cloning (see Section 6.4).

6. Establishing Connections

Before a Connection can be used for data transfer, it must be established. Establishment ends the pre-establishment phase; all transport and cryptographic parameter specification must be complete before establishment, as these parameters 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()
```

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 parameters necessary for candidate selection. After calling Initiate, no further parameters may be bound to the Connection. The Initiate() call consumes the Preconnection and creates a Connection Object. A Preconnection can only be initiated once.

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 7) after calling Initate(); note that any idempotent data 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) will occur before the Ready Event for Connections established using Initiate.

Connection -> InitiateError<>

An InitiateError occurs either when the set of transport and cryptographic 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, or the application is prohibited from opening a Connection by the operating system).

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:

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 parameters necessary for Protocol Stack selection. A Remote Endpoint may optionally be specified, to constrain what Connections are accepted. The Listen() Action consumes the Preconnection. Once Listen() has been called, no further parameters may be bound to the Preconnection, and no subsequent establishment call may be made on the Preconnection.

Preconnection -> ConnectionReceived<Connection>

The ConnectionReceived Event occurs when a Remote Endpoint has established a transport-layer connection to this Preconnection (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.

Preconnection -> ListenError<>

A ListenError occurs either when the Preconnection cannot be fulfilled for listening, when the Local Endpoint (or Remote Endpoint, if specified) cannot be resolved, or when the application is prohibited from listening by policy.

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 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 consumes the Preconnection. Once Rendezvous() has been called, no further parameters may be bound to the Preconnection, and no subsequent establishment call may be made on the Preconnection.

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 -> RendezvousError<msgRef, error>

An RendezvousError occurs either when the Preconnection cannot be fulfilled for listening, 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., ICE [RFC5245], it is expected that the Local Endpoint will be configured with some method of discovering NAT bindings, e.g., a STUN server. In this case, the Local Endpoint may resolve to a mixture of local and server reflexive addresses. The Resolve() method on the Preconnection can be used to discover these bindings:

PreconnectionBindings := 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. This list can be passed to a peer via a signalling protocol, such as SIP or WebRTC, to configure the remote.

6.4. Connection Groups

Groups of Preconnections or Connections can be created using the Clone Action:

Preconnection := Preconnection.Clone()

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 clone 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 group, and so on. Connections in a Connection Group share all their properties, and changing the properties on one Connection in the group changes the property for all others.

Calling Clone on a Preconnection yields a Preconnection with the same parameters, which is entangled with the parent Preconnection: all the Connections created from entangled Preconnections will be entangled as if they had been cloned, and will belong to the same Connection Group.

Establishing a Connection from a cloned Preconnection will not cause Connections for other entangled Preconnections to be established; each such Connection must be established separately. Changes to the parameters of a Preconnection entangled with a Preconnection from which a Connection has already been established will fail. Calling Clone on a Preconnection may be taken by the system an implicit signal that Protocol Stacks supporting multiplexed Connections for efficient Connection Grouping are preferred by the application.

There is only one Protocol Property that is not entangled, i.e., it is a separate per-Connection Property for individual Connections in the group: niceness. Niceness works as in Section 7.1.2: when allocating available network capacity among Connections in a Connection Group, sends on Connections with higher Niceness values will be prioritized over sends on Connections with lower Niceness values. An ideal transport system implementation would assign the Connection the capacity share (M-N) x C / M, where N is the Connection's Niceness value, M is the maximum Niceness value used by all Connections in the group and C is the total available capacity. However, the niceness setting is purely advisory, and no guarantees are given about capacity allocation and each implementation is free to implement exact capacity allocation as it sees fit.

7. Sending Data

Once a Connection has been established, it can be used for sending data. Data is sent by passing a Message Object and additional parameters Section 7.1 to the Send Action on an established Connection:

```
Connection.Send(Message, sendParameters)
```

The type of the Message to be passed is dependent on the implementation, and on the constraints on the Protocol Stacks implied by the Connection's transport parameters. It may itself contain an array of octets to be transmitted in the transport protocol payload, or be transformable to an array of octets by a sender-side framer (see Section 7.2).

Some transport protocols can deliver arbitrarily sized Messages, but other protocols constrain the maximum Message size. Applications can

query the protocol property Maximum Message Size on Send to determine the maximum size.

There may also be system and Protocol Stack dependent limits on the size of a Message which can be transmitted atomically. For that reason, the Message object passed to the Send action may also be a partial Message, either representing the whole data object and information about the range of bytes to send from it, or an object referring back to the larger whole Message. The details of partial Message sending are implementation-dependent.

If Send is called on a Connection which has not yet been established, an Initiate Action will be implicitly performed simultaneously with the Send. Used together with the Idempotent property (see Section 7.1.4), this can be used to send data during establishment for 0-RTT session resumption on Protocol Stacks that support it.

Like all Actions in this interface, the Send Action is asynchronous.

Connection -> Sent<msgRef>

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 the implementation of this interface. The exact disposition of the Message when the Sent Event occurs is specific to the implementation and the constraints on the Protocol Stacks implied by the Connection’s transport parameters. The Sent Event contains an implementation-specific 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 only issues a Send after this Event fires.

Connection -> Expired<msgRef>

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 7.1.1) expired. This is separate from SendError, as it is an expected behavior for partially reliable transports. The Expired Event contains an implementation-specific reference to the Message to which it applies.

Connection -> SendError<msgRef>

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 send parameters not consistent with the Connection's transport parameters. The SendError contains an implementation-specific reference to the Message to which it applies.

7.1. Send Parameters

The Send Action takes per-Message send parameters which control how the contents will be sent down to the underlying Protocol Stack and transmitted.

If Send Parameters should be overridden for a specific Message, an empty sent parameter Object can be acquired and all desired Send Parameters can be added to that Object. A sendParameters Object can be reused for sending multiple contents with the same properties.

```
SendParameters := NewSendParameters()
SendParameters.Add(parameter, value)
```

The Send Parameters share a single namespace with the Transport Parameters (see Section 5.2). This allows the specification of Protocol Properties that can be overridden on a per-Message basis.

Send Parameters may be inconsistent with the properties of the Protocol Stacks underlying the Connection on which a given Message is sent. For example, infinite Lifetime is not possible on a Message over a Connection not providing reliability. Sending a Message with Send Properties inconsistent with the Transport Preferences on the Connection yields an error.

The following send parameters are supported:

7.1.1. Lifetime

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. When a Message's Lifetime is infinite, it must be transmitted reliably. The type and units of Lifetime are implementation-specific.

7.1.2. Niceness

Niceness represents an unbounded hierarchy of priorities of Messages, relative to other Messages sent over the same Connection and/or Connection Group (see Section 6.4). It is most naturally represented as a non-negative integer. A Message with Niceness 0 will yield to a

Message with Niceness 1, which will yield to a Message with Niceness 2, and so on. Niceness 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 inversion of normal schemes for expressing priority has a convenient property: priority increases as both Niceness and Lifetime decrease.

7.1.3. Ordered

Ordered is a boolean property. If true, this Message should be delivered after the last Message passed to the same Connection via the Send Action; if false, this Message may be delivered out of order.

7.1.4. Idempotent

Idempotent is a boolean property. If true, the application-layer entity in the 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.

7.1.5. Corruption Protection Length

This numeric property specifies the length of the section of the Message, starting from byte 0, that the application assumes will be received without corruption due to lower layer errors. It is used to specify options for simple integrity protection via checksums. By default, the entire Message is protected by checksum. A value of 0 means that no checksum is required, and a special value (e.g. -1) can be used to indicate the default. Only full coverage is guaranteed, any other requests are advisory.

7.1.6. Immediate Acknowledgement

This boolean property specifies, if true, that an application wants this Message to be acknowledged immediately by the receiver. In case of reliable transmission, this informs the transport protocol on the sender side faster that it can remove the Message from its buffer; therefore this property can be useful for latency-critical applications that maintain tight control over the send buffer (see Section 7).

7.1.7. Instantaneous Capacity Profile

This enumerated property specifies the application's preferred tradeoffs for sending this Message; it is a per-Message override of the Capacity Profile protocol and path selection property (see Section 5.2).

The following values are valid for Instantaneous Capacity Profile:

Default: No special optimizations of the tradeoff between delay, delay variation, and bandwidth efficiency should be made when sending this message.

Interactive/Low Latency: Response time (latency) should be optimized at the expense of bandwidth efficiency and delay variation when sending this message. This can be used by the system to disable the coalescing of multiple small Messages into larger packets (Nagle's algorithm), to signal a preference for lower-latency, higher-loss treatment, and so on.

Constant Rate: Delay and delay variation should be optimized at the expense of bandwidth efficiency.

Scavenger/Bulk: This Message may be sent at the system's leisure. This can be used to signal a preference for less-than-best-effort treatment, to delay sending until lower-cost paths are available, and so on.

7.2. Sender-side Framing

Sender-side framing allows a caller to provide the interface with a function that takes a Message of an appropriate application-layer type and returns an array of octets, the on-the-wire representation of the Message to be handed down to the Protocol Stack. It consists of a Framer Object with a single Action, Frame. Since the Framer depends on the protocol used at the application layer, it is bound to the Preconnection during the pre-establishment phase:

```
Preconnection.FrameWith(Framer)

OctetArray := Framer.Frame(Message)
```

Sender-side framing is a convenience feature of the interface, for parity with receiver-side framing (see Section 8.1).

8. Receiving Data

Once a Connection is established, Messages may be received on it. The application can indicate that it is ready to receive Messages by calling Receive() on the Connection.

Connection.Receive(ReceiveHandler, maxLength)

Receive takes a ReceiveHandler, which can handle the Received Event and the ReceiveError error. Each call to Receive will result in at most one Received event being sent to the handler, though implementations may provide convenience functions to indicate readiness to receive a larger but finite number of Messages with a single call. This allows an application to provide backpressure to the transport stack when it is temporarily not ready to receive messages.

Receive also takes an optional maxLength argument, the maximum size (in bytes of data) Message 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 received as a partial Message. Note that maxLength does not guarantee that the application will receive that many bytes if they are available; the interface may return partial Messages smaller than maxLength according to implementation constraints.

Connection -> Received<Message>

As with sending, the type of the Message to be passed is dependent on the implementation, and on the constraints on the Protocol Stacks implied by the Connection's transport parameters. The Message may also contain metadata from protocols in the Protocol Stack; which metadata is available is Protocol Stack dependent. In particular, when this information is available, the value of the Explicit Congestion Notification (ECN) field is contained in such metadata. 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.

The Message Object must provide some method to retrieve an octet array containing application data, corresponding to a single message within the underlying Protocol Stack's framing. See Section 8.1 for handling framing in situations where the Protocol Stack provides octet-stream transport only.

The Message Object passed to Received is complete and atomic, unless one of the following conditions holds:

- 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 deframer (see Section 8.1) 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 deframer was supplied by the application

The Message Object passed to Received will indicate one of the following:

1. this is a complete message;
2. this is a partial message containing a section of a message with a known message boundary (made partial for local buffering reasons, either by the underlying Protocol Stack or the deframer). In this case, the Message Object passed to Received may contain the byte offset of the data in the partial Message within the full Message, an indication whether this is the last (highest-offset) partial Message in the full Message, and an optional reference to the full Message it belongs to; or
3. this is a partial message containing data with no definite message boundary, i.e. the only known message boundary is given by termination of the Connection

Note that in the absence of message boundary preservation and without deframing, the entire Connection is represented as one large message of indeterminate length.

Connection -> ReceiveError<>

A ReceiveError occurs when data is received by the underlying Protocol Stack that cannot be fully retrieved or deframed, or when some other indication is received that reception has failed. Such conditions that irrevocably lead the the termination of the Connection are signaled using ConnectionError instead (see Section 10).

8.1. Receiver-side De-framing over Stream Protocols

The Receive Event is intended to be fired once per application-layer Message sent by the remote endpoint; i.e., it is a desired property of this interface that a Send at one end of a Connection maps to exactly one Receive on the other end. This is possible with Protocol

Stacks that provide message boundary preservation, but is not the case over Protocol Stacks that provide a simple octet stream transport.

For preserving message boundaries over stream transports, this interface provides receiver-side de-framing. This facility is based on the observation that, since many of our 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. A Deframer allows an application to push this de-framing down into the interface, in order to transform an octet stream into a sequence of Messages.

Concretely, receiver-side de-framing allows a caller to provide the interface with a function that takes an octet stream, as provided by the underlying Protocol Stack, reads and returns a single Message of an appropriate type for the application and platform, and leaves the octet stream at the start of the next Message to deframe. It consists of a Deframer Object with a single Action, Deframe. Since the Deframer depends on the protocol used at the application layer, it is bound to the Preconnection during the pre-establishment phase:

```
Preconnection.DeframeWith(Deframer)

Message := Deframer.Deframe(OctetStream, ...)
```

9. Setting and Querying of Connection Properties

At any point, the application can set and query the properties of a Connection. Depending on the phase the Connection is in, the Connection properties will include different information.

```
ConnectionProperties := Connection.GetProperties()

Connection.SetProperties()
```

Connection properties include:

- The status of the Connection, which can be one of the following: Establishing, Established, Closing, or Closed.
- Transport Features of the protocols that conform to the Required and Prohibited Transport Preferences, which might be selected by the transport system during Establishment. These features correspond to the properties given in Section 5.2 and can only be queried.

- Transport Features of the Protocol Stacks that were selected and instantiated, once the Connection has been established. These features correspond to the properties given in Section 5.2 and can only be queried. Instead of preference levels, these features have boolean values indicating whether or not they were selected. Note that these transport features may not fully reflect the specified parameters given in the pre-establishment phase. For example, a certain Protocol Selection Property that an application specified as Preferred may not actually be present in the chosen Protocol Stack Instances because none of the currently available transport protocols had this feature.

- Protocol Properties of the Protocol Stack in use (see Section 9.1 below). These can be set or queried. Certain specific procotol queries may be read-only, on a protocol- and property-specific basis.

- Path Properties of the path(s) in use, once the Connection has been established. These properties can be derived from the local provisioning domain, measurements by the Protocol Stack, or other sources. They can only be queried.

9.1. Protocol Properties

Protocol Properties represent the configuration of the selected Protocol Stacks backing a Connection. Some properties apply generically across multiple transport protocols, while other properties only apply to specific protocols. The default settings of these properties will vary based on the specific protocols being used and the system’s configuration.

Note that Protocol Properties are also set during pre-establishment, as transport parameters, to preconfigure Protocol Stacks during establishment.

Generic Protocol Properties include:

- Relative niceness: This numeric property is similar to the Niceness send property (see Section 7.1.2), a non-negative integer representing the relative inverse 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.

- Timeout for aborting Connection: This numeric property specifies how long to wait before aborting a Connection during

establishment, or before deciding that a Connection has failed after establishment. It is given in seconds.

- Retransmission threshold before excessive retransmission notification: This numeric property specifies after how many retransmissions to inform the application about "Excessive Retransmissions".

- Required minimum coverage of the checksum for receiving: This numeric 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, and a special value (e.g., -1) indicates full checksum coverage.

- Connection group transmission scheduler: This enumerated property specifies which scheduler should be used among Connections within a Connection Group. It applies to Connection Groups; the set of schedulers can be taken from [I-D.ietf-tsvwg-sctp-ndata].

- Maximum message size concurrent with Connection establishment: This numeric property represents the maximum Message size that can be sent before or during Connection establishment, see also Section 7.1.4. It is given in Bytes. This property is read-only.

- Maximum Message size before fragmentation or segmentation: This numeric property, if applicable, represents the maximum Message size that can be sent without incurring network-layer fragmentation and/or transport layer segmentation at the sender. This property is read-only.

- Maximum Message size on send: This numeric property represents the maximum Message size that can be sent. This property is read-only.

- Maximum Message size on receive: This numeric property represents the maximum Message size that can be received. This property is read-only.

In order to specify Specific Protocol Properties, Transport System implementations may offer applications to attach a set of options to the Preconnection Object, associated with a specific protocol. For example, an application could specify a set of TCP Options to use if and only if TCP is selected by the system. Such properties 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.

10. 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 be signaled.

Connection -> Closed<>

Abort terminates a Connection without delivering remaining data:

Connection.Abort()

A ConnectionError can inform the application that the other side has aborted the Connection; however, there is no guarantee that an abort will be signaled:

Connection -> ConnectionError<>

11. IANA Considerations

RFC-EDITOR: Please remove this section before publication.

This document has no Actions for IANA.

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 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.

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.

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.

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

[I-D.ietf-tsvwg-sctp-ndata]
Stewart, R., Tuexen, M., Loreto, S., and R. Seggelmann, "Stream Schedulers and User Message Interleaving for the Stream Control Transmission Protocol", draft-ietf-tsvwg-sctp-ndata-13 (work in progress), September 2017.

[TAPS-ARCH]
Pauly, T., Ed., Trammell, B., Ed., Brunstrom, A., Fairhurst, G., Perkins, C., Tiesel, P., and C. Wood, "An Architecture for Transport Services", n.d..

14.2. Informative References

[I-D.pauly-taps-transport-security]
Pauly, T., Rose, K., and C. Wood, "A Survey of Transport Security Protocols", draft-pauly-taps-transport-security-01 (work in progress), January 2018.

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

[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>.

[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>.

Appendix A. Additional Properties

The interface specified by this document represents the minimal common interface to an endpoint in the transport services architecture [TAPS-ARCH], based upon that architecture and on the minimal set of transport service features elaborated in [I-D.ietf-taps-minset]. However, the interface has been designed with extension points to allow the implementation of features beyond those in the minimal common interface: Protocol Selection Properties, Path Selection Properties, and options on Message send are open sets. Implementations of the interface are free to extend these sets to provide additional expressiveness to applications written on top of them.

This appendix enumerates a few 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. Protocol and Path Selection Properties

The following protocol and path selection properties might be made available in addition to those specified in Section 5.2:

- Suggest a timeout to the Remote Endpoint: This boolean property specifies whether an application considers it useful to propose a timeout until the Connection is assumed to be lost. This property applies to Connections and Connection Groups. This is not a strict requirement. The default is to have this option.

[EDITOR'S NOTE: For discussion of this option, see https://github.com/taps-api/drafts/issues/109]

- Request not to delay acknowledgment of Message: This boolean property specifies whether an application considers it useful to request for Message that its acknowledgment be sent out as early as possible instead of potentially being bundled with other acknowledgments. This property applies to Connections and Connection groups. This is not a strict requirement. The default is to not have this option. [EDITOR'S NOTE: For discussion of this option, see https://github.com/taps-api/drafts/issues/90]

A.1.1. Application Intents

Application Intents are a group of transport properties expressing what an application wants to achieve, knows, assumes or prefers regarding its communication. They are not strict requirements. In particular, they should not be used to express any Quality of Service expectations that an application might have. Instead, an application should express its intentions and its expected traffic characteristics in order to help the transport system make decisions that best match it, but on a best-effort basis. Even though Application Intents do not represent Quality of Service requirements, a transport system may use them to determine a DSCP value, e.g. similar to Table 1 in [I-D.ietf-tsvwg-rtcweb-qos].

Application Intents can influence protocol selection, protocol configuration, path selection, and endpoint selection. For example, setting the "Timeliness" Intent to "Interactive" may lead the transport system to disable the Nagle algorithm for a Connection, while setting the "Timeliness" to "Background" may lead it to setting the DSCP value to "scavenger". If the "Size to be Sent" Intent is set on an individual Message, it may influence path selection.

Specifying Application Intents is not mandatory. An application can specify any combination of Application Intents. If specified, Application Intents are defined as parameters passed to the Preconnection Object, and may influence the Connection established from that Preconnection. If a Connection is cloned to form a Connection Group, and associated Application Intents are cloned along with the other transport parameters. Some Intents have also corresponding Message Properties, similar to the properties in Section 7.1.

Application Intents can be added to this interface as Transport Preferences with the "Prefer" preference level.

A.1.1.1. Traffic Category

This Intent specifies what the application expect the dominating traffic pattern to be.

Possible Category values are:

Query: Single request / response style workload, latency bound

Control: Long lasting low bandwidth control channel, not bandwidth bound

Stream: Stream of data with steady data rate

Bulk: Bulk transfer of large Messages, presumably bandwidth bound

The default is to not assume any particular traffic pattern. Most categories suggest the use of other intents to further describe the traffic pattern anticipated, e.g., the bulk category suggesting the use of the Message Size intents or the stream category suggesting the Stream Bitrate and Duration intents.

A.1.1.2. Size to be Sent / Received

This Intent specifies what the application expects the size of a transfer to be. It is a numeric property and given in Bytes.

A.1.1.3. Duration

This Intent specifies what the application expects the lifetime of a transfer to be. It is a numeric property and given in milliseconds.

A.1.1.4. Send / Receive Bit-rate

This Intent specifies what the application expects the bit-rate of a transfer to be. It is a numeric property and given in Bytes per second.

A.1.1.5. Cost Preferences

This Intent describes what an application prefers regarding monetary costs, e.g., whether it considers it acceptable to utilize limited data volume. It provides hints to the transport system on how to handle trade-offs between cost and performance or reliability. This Intent can also apply to an individual Messages.

No Expense: Avoid transports associated with monetary cost

Optimize Cost: Prefer inexpensive transports and accept service degradation

Balance Cost: Use system policy to balance cost and other criteria

Ignore Cost: Ignore cost, choose transport solely based on other criteria

The default is "Balance Cost".

A.2. Protocol Properties

The following protocol properties might be made available in addition to those in Section 9.1:

- Abort timeout to suggest to the Remote Endpoint: This numeric property specifies the timeout to propose to the Remote Endpoint. It is given in seconds. [EDITOR'S NOTE: For discussion of this property, see https://github.com/taps-api/drafts/issues/109]

A.3. Send Parameters

The following send parameters might be made available in addition to those specified in Section 7.1:

- Immediate: Immediate is a boolean property. If true, the caller prefers immediacy to efficient capacity usage for this Message. For example, this means that the Message should not be bundled with other Message into the same transmission by the underlying Protocol Stack.

- Send Bitrate: This numeric property in Bytes per second specifies at what bitrate the application wishes the Message to be sent. A transport supporting this feature will not exceed the requested Send Bitrate even if flow-control and congestion control allow higher bitrates. This helps to avid bursty traffic pattern on busy video streaming servers.

Appendix B. Sample API definition in Go

This document defines an abstract interface. To illustrate how this would map concretely into a programming language, an API interface definition in Go is available online at https://github.com/mami-project/postsocket. Documentation for this API - an illustration of the documentation an application developer would see for an instance of this interface - is available online at https://godoc.org/github.com/mami-project/postsocket. This API

definition will be kept largely in sync with the development of this abstract interface definition.

Authors' Addresses

Brian Trammell (editor)
ETH Zurich
Gloriastrasse 35
8092 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
TU Berlin
Marchstrasse 23
10587 Berlin
Germany

Email: theresa@inet.tu-berlin.de

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
ETH Zurich
Gloriastrasse 35
8092 Zurich
Switzerland

Email: mirja.kuehlewind@tik.ee.ethz.ch

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

Email: csp@csperkins.org

Philipp S. Tiesel
TU Berlin
Marchstrasse 23
10587 Berlin
Germany

Email: philipp@inet.tu-berlin.de

Chris Wood
Apple Inc.
1 Infinite Loop
Cupertino, California 95014
United States of America

Email: cawood@apple.com