TAPS Working Group                                    A. Brunstrom, Ed.
Internet-Draft                                      Karlstad University
Intended status: Informational                           T. Pauly, Ed.
Expires: 16 June 2024                                       Apple Inc.
                                                          R. Enghardt
                                                              Netflix
                                                           P. Tiesel
                                                              SAP SE
                                                            M. Welzl
                                                    University of Oslo
                                                    14 December 2023

           Implementing Interfaces to Transport Services
                      draft-ietf-taps-impl-18

Abstract

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

Status of This Memo

Copyright Notice

Table of Contents

1.  Introduction

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

   This document serves as a guide to implementing a system that
   provides a Transport Services API.  This guide offers suggestions to
   developers, but it is not prescriptive: implementations are free to
   take any desired form as long as the API specification in
   [I-D.ietf-taps-interface] is honored.  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 Transport Services
   architecture [I-D.ietf-taps-arch].

2.  Implementing Connection Objects

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

   *  the Preconnection, the bundle of properties that describes the
      application constraints on, and preferences for, the transport;

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

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

   Preconnection objects should be implemented as bundles of properties
   that an application can both read and write.  A Preconnection object
   influences a Connection only at one point in time: when the
   Connection is created.  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 not necessarily be
   immediately bound to a transport protocol instance, since multiple
   candidate Protocol Stacks might be raced.

   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 cannot be mutated by
   the application making changes to the original Preconnection object.
   This may involve the implementation performing a deep-copy, copying
   the object with all the objects that it references.

   Once the Connection is established, the Transport Services
   Implementation maps actions and events to the details of the chosen
   Protocol Stack.  For example, the same Connection object may
   ultimately represent a single transport protocol instance (e.g., a
   TCP connection, a TLS session over TCP, a UDP flow with fully-
   specified Local and Remote Endpoint Identifiers, a DTLS session, a
   SCTP stream, a QUIC stream, or an HTTP/2 stream).  The Connection
   Properties held by a Connection or Listener are independent of other
   Connections that are not part of the same Connection Group.

   Connection establishment is only a local operation for a
   connectionless protocols, which serves to simplify the local send/
   receive functions and to filter the traffic for the specified
   addresses and ports [RFC8085] (for example using UDP or UDP-Lite
   transport without a connection handshake procedure).

   Once Initiate has been called, the Selection Properties and Endpoint
   information of the created Connection are immutable (i.e, an
   application is not able to later modify the properties of a
   Connection by manipulating the original Preconnection object).
   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 4.7.

3.  Implementing Pre-Establishment

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

   During pre-establishment the application specifies one or more
   Endpoints to be used for communication as well as protocol
   preferences and constraints via Selection Properties and, if desired,
   also Connection Properties.  Section 4 of [I-D.ietf-taps-interface]
   states that Connection Properties should preferably be configured
   during pre-establishment, because they can serve as input to
   decisions that are made by the implementation (e.g., the capacity
   profile can guide usage of a protocol offering scavenger-type
   congestion control).

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

3.1.  Configuration-time errors

   The Transport Services system should have a list of supported
   protocols available, which each have transport features reflecting
   the capabilities of the protocol.  Once an application specifies its
   Transport Properties, the Transport Services system matches the
   required and prohibited properties against the transport features of
   the available protocols (see Section 6.2 of [I-D.ietf-taps-interface]
   for the definition of property preferences).

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

* A request by an application for properties that cannot be
  satisfied by any of the available protocols.  For example, if an
  application requires perMsgReliability, but no such feature is
  available in any protocol on the host running the Transport
  Services system this should result in an error.

* A request by an application for properties that are in conflict
  with each other, such as specifying required and prohibited
  properties that cannot be satisfied by any protocol.  For example,
  if an application prohibits reliability but then requires
  perMsgReliability, this mismatch should result in an error.

To avoid allocating resources that are not finally needed, it is
important that configuration-time errors fail as early as possible.

3.2.  Role of system policy

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

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

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

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

In general, any protocol or path used for a Connection must conform
to all three sources of constraints.  A violation that occurs at any
of the policy layers should cause a protocol or path to be considered
ineligible for use.  If such a violation prevents a Connection from
being established, this should be communicated to the application,
e.g. via the EstablishmentError event.  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.

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 which point the Preconnection object contains
all constraints or requirements the application has configured.  The
establishment 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 (defined in Section 4.2.1 of
[I-D.ietf-taps-arch]), to identify the paths, protocols, and
endpoints to use (see Section 4.2); and Candidate Racing (defined in
Section 4.2.2 of [I-D.ietf-taps-arch]), in which the necessary
protocol handshakes are conducted so that the Transport Services
system can select which set to use (see Section 4.3).  Candidate
Racing involves attempting multiple options for connection
establishment, and choosing the first option to succeed as the
Protocol Stack to use for the connection.  These attempts are usually
staggered, starting each next option after a delay, but they can also
be performed in parallel or only after waiting for failures.

For ease of illustration, this document structures the candidates for
racing as a tree (see Section 4.1).  This is not meant to restrict
implementations from structuring racing candidates differently.

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 path (i.e., using the default
interface), and starting a TCP handshake to establish a stream to the
specified IP address.  However, each step may also differ depending
on the requirements of the connection: if the Endpoint Identifier is
a hostname and port, then there may be multiple resolved addresses
that are available; there may also be multiple paths available, (in
this case using an interface 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 Identifier, 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 Identifier: www.example.com:443] [Interface: Any]   [Protocol
: TCP]
|-> [Endpoint Identifier: [2001:db8:23::1]:443]          [Interface: Wi-Fi] [Protoc
ol: TCP]
|-> [Endpoint Identifier: 192.0.2.1:443]                 [Interface: LTE]   [Protocol
: TCP]
|-> [Endpoint Identifier: [2001:db8:42::1]:443]          [Interface: LTE]   [Protoc
ol: TCP]
```

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

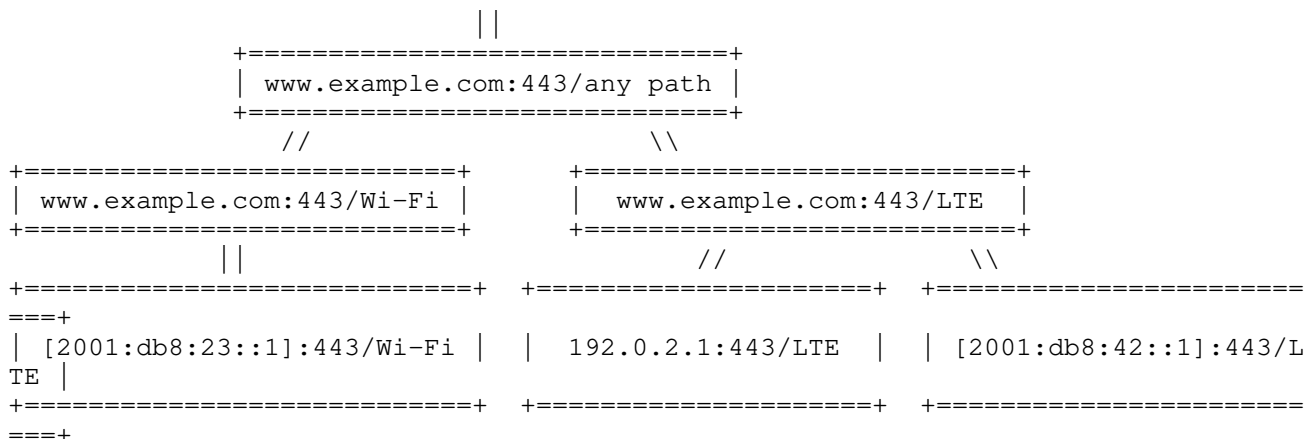During Candidate Gathering (Section 4.2), an implementation prunes
and sorts branches according to the Selection Property preferences
(Section 6.2 of [I-D.ietf-taps-interface].  It first excludes all
protocols and paths that match a Prohibit property or do not match
all Require properties.  Then it will sort branches according to
Preferred properties, Avoided properties, and possibly other
criteria.

4.1.  Structuring Candidates as a Tree

   As noted above, the consideration of multiple candidates in a
   gathering and racing process can be conceptually structured as a
   tree; this terminological convention is used throughout this
   document.

   Each leaf node of the tree represents a single, coherent connection
   attempt, with an endpoint, a network 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
   Identifier.  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 network paths, with a branch for each interface.

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

```
                               ||
                 +==============================+
                 | www.example.com:443/any path |
                 +==============================+
                      //                      \\
 +===========================+       +===========================+
 | www.example.com:443/Wi-Fi |       |   www.example.com:443/LTE |
 +===========================+       +===========================+
              ||                         //                \\
 +===========================+  +====================+  +=====================
 ===+
 | [2001:db8:23::1]:443/Wi-Fi |  |  192.0.2.1:443/LTE  |  | [2001:db8:42::1]:443/L
 TE |
 +===========================+  +====================+  +=====================
 ===+
```

   The rest of this section will use a notation scheme to represent this
   tree.  The root node (or parent node) of the tree will be represented
   by a single integer, such as "1". ("1" is used assuming that this is
   the first connection made by the system; future connections created
   by the application would allocate numbers in an increasing manner.)
   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 its parent's 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 (labeled here

by interface), and protocol.  In Protocol Stacks, the layers are
separated by '/' and ordered with the protocol closest to the
application first.  The above example can now be written more
succinctly as:

```
1 [www.example.com:443, any path, TCP]
  1.1 [www.example.com:443, Wi-Fi, TCP]
    1.1.1 [[2001:db8:23::1]:443, Wi-Fi, TCP]
  1.2 [www.example.com:443, LTE, TCP]
    1.2.1 [192.0.2.1:443, LTE, TCP]
    1.2.2 [[2001:db8.42::1]:443, LTE, TCP]
```

When an implementation is asked to establish a single connection,
only one of the leaf nodes in the candidate set is needed to transfer
data.  Thus, once a single leaf node becomes ready to use, then the
connection establishment tree is considered ready.  One way to
implement this is by having every leaf node update the state of its
parent node when it becomes ready, until the root 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 consist of only a single node,
such as a connection attempt to an IP address over a single interface
with a single protocol.

```
1 [[2001:db8:23::1]:443, Wi-Fi, TCP]
```

A root 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:443, Wi-Fi, TCP]
  1.1 [[2001:db8:23::1]:443, Wi-Fi, TCP]
```

4.1.1.  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.

4.1.1.1.  Derived Endpoints

If a connection originally targets a single Endpoint Identifer, there
may be multiple endpoint candidates of different types that can be
derived from the original.  This creates an ordered list of the
derived endpoint candidates 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
Identifer on a traditional IP network, the implementation should send
all applicable DNS queries.  Commonly, this will include both A
(IPv4) and AAAA (IPv6) records if both address families are supported
on the local interface.  This can also include SRV records [RFC2782],
SVCB and HTTPS records [I-D.ietf-dnsop-svcb-https], or other future
record types.  The algorithm for ordering and racing these addresses
should follow the recommendations in Happy Eyeballs [RFC8305].

```
1 [www.example.com:443, Wi-Fi, TCP]
  1.1 [[2001:db8::1]:443, Wi-Fi, TCP]
  1.2 [192.0.2.1:443, Wi-Fi, TCP]
  1.3 [[2001:db8::2]:443, Wi-Fi, TCP]
  1.4 [[2001:db8::3]:443, Wi-Fi, TCP]
```

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

```
1 [term-printer._ipp._tcp.meeting.example.com, Wi-Fi, TCP]
  1.1 [term-printer.meeting.example.com:631, Wi-Fi, TCP]
    1.1.1 [31.133.160.18:631, Wi-Fi, TCP]
```

Applications can influence which derived Endpoints are allowed and
preferred via Selection Properties set on the Preconnection.  For
example, setting a preference for useTemporaryLocalAddress would
prefer the use of IPv6 over IPv4, and requiring
useTemporaryLocalAddress would eliminate IPv4 options, since IPv4
does not support temporary addresses.

4.1.1.2.  Network Paths

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

```
1 [192.0.2.1:443, any path, TCP]
  1.1 [192.0.2.1:443, Wi-Fi, TCP]
  1.2 [192.0.2.1:443, LTE, TCP]
```

The same approach applies to any situation in which the client is aware of multiple links or views of the network.  A single interface may be shared by multiple network paths, each with a coherent set of addresses, routes, DNS server, and more.  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 the application sets, as well as by the system policy.

## 4.1.1.3.  Protocol Options

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

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

```
1 [www.example.com:443, any path, HTTP/TCP]
  1.1 [192.0.2.8:443, any path, HTTP/HTTP Proxy/TCP]
  1.2 [192.0.2.7:10234, any path, HTTP/SOCKS/TCP]
  1.3 [www.example.com:443, any path, HTTP/TCP]
    1.3.1 [192.0.2.1:443, any path, HTTP/TCP]
```

This approach also allows a client to attempt different sets of application and transport protocols that, when available, could provide preferable features.  For example, the protocol options could involve QUIC [RFC9000] over UDP on one branch, and HTTP/2 [RFC7540] over TLS over TCP on the other:

```
1 [www.example.com:443, any path, HTTP]
  1.1 [www.example.com:443, any path, HTTP3/QUIC/UDP]
    1.1.1 [192.0.2.1:443, any path, HTTP3/QUIC/UDP]
  1.2 [www.example.com:443, any path, HTTP2/TLS/TCP]
    1.2.1 [192.0.2.1:443, any path, HTTP2/TLS/TCP]
```

Another example is racing SCTP with TCP:

```
1 [www.example.com:4740, any path, reliable-inorder-stream]
  1.1 [www.example.com:4740, any path, SCTP]
    1.1.1 [192.0.2.1:4740, any path, SCTP]
  1.2 [www.example.com:4740, any path, TCP]
    1.2.1 [192.0.2.1:4740, any path, TCP]
```

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

## 4.1.2.  Branching Order-of-Operations

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

This document recommends the following order of operations for
branching:

1.  Network Paths

2.  Protocol Options

3.  Derived Endpoints

where a lower number indicates higher precedence and therefore higher
placement in the tree.  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 establishment process can be seen as a race
between the available paths or interfaces.

Protocol options are next checked in order.  Whether or not a set of
protocols, 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 Endpoint Identifers being used.  Adding a proxy
to a connection's branch will change the Endpoint Identifer to the
proxy's IP address or hostname.  Choosing an alternate protocol may
also modify the ports that should be selected.

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

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

```
1. [www.example.com:80, any path, reliable-inorder-stream]
1.1 [192.0.2.1:443, Wi-Fi, reliable-inorder-stream]
1.1.1 [192.0.2.1:443, Wi-Fi, TCP]
1.2 [192.0.3.1:443, LTE, reliable-inorder-stream]
1.2.1 [192.0.3.1:443, LTE, SCTP]
1.2.2 [192.0.3.1:443, LTE, TCP]
```

4.1.3.  Sorting Branches

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

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

*   "Interface Instance or Type" (property name interface): If the
    application specifies an interface type to be preferred or
    avoided, implementations should accordingly rank the paths.  If
    the application specifies an interface type to be required or
    prohibited, an implementation is expected to exclude the non-
    conforming paths.

*   "Capacity Profile" (property name connCapacityProfile): An
    implementation can use the capacity profile to prefer paths that
    match an application's expected traffic profile.  This match will
    use cached performance estimates, see Section 9.2.  Some examples
    of path preferences based on capacity profiles include:

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

    -   Low Latency/Non-Interactive: Prefer paths with a low expected
        Round Trip Time, but can tolerate delay variation;

    -   Constant-Rate Streaming: Prefer paths that are expected to
        satisfy the requested stream send or receive bitrate, based on
        the observed maximum throughput;

    -   Capacity-Seeking: Prefer adapting to paths to determine the
        highest available capacity, based on the observed maximum
        throughput.

As another example, branch sorting can also be influenced by bounds
on the send or receive rate (Selection Properties minSendRate /
minRecvRate / maxSendRate / maxRecvRate): if the application
indicates a bound on the expected send or receive bitrate, an
implementation may prefer a path that can likely provide the desired
bandwidth, based on cached maximum throughput, see Section 9.2.  The
application may know the send or receive bitrate from metadata in
adaptive HTTP streaming, such as MPEG-DASH.

Implementations process the Properties (Section 6.2 of
[I-D.ietf-taps-interface]) in the following order: Prohibit, Require,
Prefer, Avoid.  If Selection Properties contain any prohibited
properties, the implementation should first purge branches containing
nodes with these properties.  For required properties, it should only
keep branches that satisfy these requirements.  Finally, it should
order the branches according to the preferred properties, and finally
use any avoided properties as a tiebreaker.  When ordering branches,
an implementation can give more weight to properties that the
application has explicitly set, than to the properties that are
default.

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

## 4.2.  Candidate Gathering

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

## 4.2.1.  Gathering Endpoint Candidates

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

## 4.2.1.1.  Local Endpoint candidates

The set of possible Local Endpoints is gathered.  In a simple case,
this merely enumerates the local interfaces and protocols, and
allocates ephemeral source ports.  For example, a system that has
WiFi and Ethernet and supports IPv4 and IPv6 might gather four
candidate Local Endpoints (IPv4 on Ethernet, IPv6 on Ethernet, IPv4
on WiFi, and IPv6 on WiFi) that can form the source for a transient.

If NAT traversal is required, the process of gathering Local
Endpoints becomes broadly equivalent to the ICE Candidate Gathering
phase (see Section 5.1.1 of [RFC8445]).  The endpoint determines its
server reflexive Local Endpoints (i.e., the translated address of a
Local Endpoint, on the other side of a NAT, e.g via a STUN sever
[RFC5389]) and relayed Local Endpoints (e.g., via a TURN server
[RFC5766] or other relay), for each interface and network protocol.
These are added to the set of candidate Local Endpoint Identifers for
this connection.

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

4.2.1.2.  Remote Endpoint Candidates

The Remote Endpoint Identifer 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 Endpoint of this Connection.

How this resolution 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 Identifer 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 Endpoint Identifers 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 Local Endpoints, which are returned and form the candidate remote addresses for contacting that peer.

Resolving the Remote Endpoint is not a local operation.  It will involve a directory service, and can require communication with the Remote Endpoint to rendezvous and exchange peer addresses.  This can expose some or all of the candidate Local Endpoints to the Remote Endpoint.

4.3.  Candidate Racing

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

This section covers the dynamic aspect of connection establishment. The tree described above is a useful conceptual and architectural model.  However, an implementation is unable to know all of the nodes that will be used until steps like name resolution have occurred, and many of the possible branches ultimately might not be attempted.

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

1.  Simultaneous

2.  Staggered

3.  Failover

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

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

## 4.3.1.  Simultaneous

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

## 4.3.2.  Staggered

Staggered 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.

Staggered racing attempts can proceed in parallel.  Implementations
should not terminate an earlier child connection attempt upon
starting a secondary child.

If a child node fails to establish connectivity (as in Section 4.4.1)
before the delay time has expired for the next child, the next child
should be started immediately.

Staggered racing between IP addresses for a generic Connection should
follow the Happy Eyeballs algorithm described in [RFC8305].
[RFC8421] provides guidance for racing when performing Interactive
Connectivity Establishment (ICE).

Generally, the delay before starting a given child node ought to be
based on the length of time the previously started child node is
expected to take before it succeeds or makes progress in connection
establishment.  Algorithms like Happy Eyeballs choose a delay based
on how long the transport connection handshake is expected to take.
When performing staggered races in multiple branch types (such as
racing between network interfaces, and then racing between IP
addresses), a longer delay may be chosen for some branch types.  For
example, when racing between network interfaces, the delay should
also take into account the amount of time it takes to prepare the
network interface (such as radio association) and name resolution
over that interface, in addition to the delay that would be added for
a single transport connection handshake.

Since the staggered delay can be chosen based on dynamic information,
such as predicted Round Trip Time, implementations should define
upper and lower bounds for delay times.  These bounds are
implementation-specific, and may differ based on which branch type is
being used.

### 4.3.3.  Failover

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

An example in which failover is recommended is a race between a
preferred Protocol Stack that uses a proxy and an alternate 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.

### 4.4.  Completing Establishment

The process of connection establishment completes when one leaf node
of the tree has successfully completed negotiation with the Remote
Endpoint, 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.  This is signalled to the
application using the Ready event in the API (Section 7.1 of

[I-D.ietf-taps-interface]).

Successes and failures of a given attempt should be reported up to
parent nodes (towards the root 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
root 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:443, Any, TCP]
  1.1 [www.example.com:443, Wi-Fi, TCP]
    1.1.1 [192.0.2.1:443, Wi-Fi, TCP]
  1.2 [www.example.com:443, LTE, TCP]
...
```

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

4.4.1.  Determining Successful Establishment

On a per-protocol basis, implementations may select different
criteria by which a leaf node is considered to be successfully
connected.  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 three-way handshake has been
completed.  If the only protocol being used is an connectionless
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 4.6.

When the Initiate action is called without any Messages being sent at
the same time, depending on the protocols involved, it is not
guaranteed that the Remote Endpoint will be notified of this, and
hence a passive endpoint's application may not receive a
ConnectionReceived event until it receives the first Message on the
new Connection.

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 reduced burden on the network and Remote Endpoints
from further connection attempts that are likely to be abandoned.  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 the TLS handshake
is complete.

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

## 4.5.  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 action).  When the underlying transport connection supports
multi-streaming, the Transport Services System can map each
Connection in the Connection Group to a different stream of this
connection.

For such streams, there is often no explicit connection establishment
procedure for the new stream prior to sending data on it (e.g., with
SCTP).  In this case, the same considerations apply to determining
stream establishment as apply to establishing a UDP connection, as
discussed in Section 4.4.1.  This means that there might not be any
"establishment" message (like a TCP SYN).

4.6.  Handling connectionless 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, connectionless 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 the
   Transport Services system has selected a path on which to send data.

   However, this can cause a problem if a specific peer is not reachable
   over the network using the connectionless protocol, or data cannot be
   exchanged with the peer for any other reason.  To handle the lack of
   an explicit handshake in the underlying protocol, an application can
   use a Message Framer (Section 6) on top of a connectionless protocol
   to only mark a specific connection attempt as ready when some data
   has been received, or after some application-level handshake has been
   performed by the Message Framer.

4.7.  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 Identifer is specified, the implementation should use
   an ephemeral port.

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

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

4.7.1.  Implementing Listeners for Connected Protocols

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

4.7.2.  Implementing Listeners for Connectionless Protocols

   Connectionless 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 connectionless protocols on a listening port and should perform
   four-tuple matching of packets to existing Connection objects if
   possible.  If a matching Connection object does not exist, an
   incoming packet from a connectionless protocol should cause a new
   Connection object to be created.

4.7.3.  Implementing Listeners for Multiplexed Protocols

   Protocols that provide multiplexing of streams can listen for
   entirely new connections as well as for new sub-connections (streams
   of an already existing connection).  A new stream arrival on an
   existing connection is presented to the application as a new
   Connection.  This new Connection is grouped with all other
   Connections that are multiplexed via the same protocol.

5.  Implementing Sending and Receiving Data

   The most basic mapping for sending a Message is an abstraction of
   datagrams, in which the transport protocol naturally deals in
   discrete packets (such as UDP).  Each Message here corresponds to a
   single datagram.

   For protocols that expose byte-streams (such as TCP), 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 framing (such as length-value protocols, or
   protocols that use delimiters like HTTP/1.1) may support Message
   sizes that do not fit within a single datagram.  Each Message for
   framing protocols corresponds to a single frame, which may be sent
   either as a complete Message in the underlying protocol, or in
   multiple parts.

   Messages themselves generally consist of bytes passed in the
   messageData parameter intended to be processed at an application
   layer.  However, Message objects presented through the API can carry
   associated Message Properties passed through the messageContext
   parameter.  When these are Protocol Specific Properties, they can
   include metadata that exists separately from a byte encoding.  For
   example, these Properties can include name-value pairs of
   information, like HTTP header fields.  In such cases, Messages might

be "empty", insofar as they contain zero bytes in the messageData
parameter, but can still include data in the messageContext that is
interpreted by the Protocol Stack.

## 5.1.  Sending Messages

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

### 5.1.1.  Message Properties

The API allows various properties to be associated with each Message,
which should be implemented as discussed below.

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

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

*  msgOrdered: when this is false, this disables the requirement of
   in-order-delivery for protocols that support configurable
   ordering.  When the Protocol Stack does not support configurable
   ordering, this property may be ignored.

*  safelyReplayable: when this is true, this means that the Message
   can be used by a transport mechanism that might deliver it
   multiple times -- e.g., as a result of racing multiple transports

or as part of TCP Fast Open.  Also, protocols that do not protect
against duplicated Messages, such as UDP (when used directly,
without a protocol layered atop), can only be used with Messages
that are Safely Replayable.  When a Transport Services system is
permitted to replay Messages, replay protection could be provided
by the application.

*  final: when this is true, this means that the sender will not send
   any further Messages.  The Connection need not be closed (in case
   the Protocol Stack supports half-close operation, like TCP).  Any
   Messages sent after a Message marked final will result in a
   SendError.

*  msgChecksumLen: when this is set to any value other than Full
   Coverage, it sets the minimum protection in protocols that allow
   limiting the checksum length (e.g.  UDP-Lite).  If the Protocol
   Stack does not support checksum length limitation, this property
   may be ignored.

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

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

*  noFragmentation: Setting this avoids network-layer fragmentation.
   Messages exceeding the transports current estimate of its maximum
   packet size (the singularTransmissionMsgMaxLen Connection
   Property) can result in transport segmentation when permitted, or
   generate an error.  When used with transports running over IP
   version 4, the Don't Fragment bit should be set to avoid on-path
   IP fragmentation ([RFC8304]).

   *  noSegmentation: When set, this property limits the Message size to
      the transports current estimate of its maximum packet size (the
      singularTransmissionMsgMaxLen Connection Property).  Messages
      larger than this size generate an error.  Setting this avoids
      transport-layer segmentation and network-layer fragmentation.
      When used with transports running over IP version 4, the Don't
      Fragment bit should be set to avoid on-path IP fragmentation
      ([RFC8304]).

## 5.1.2.  Send Completion

   The application should be notified (using a Sent, Expired or
   SendError event) whenever a Message or partial Message has been
   consumed by the Protocol Stack, or has failed to send.  The time at
   which a Message is considered to have been consumed by the Protocol
   Stack may vary depending on the protocol.  For example, 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.  The time at which a Message failed
   to send is when the Transport Services Implementation (including the
   Protocol Stack) has experienced a failure related to sending; this
   can depend on protocol-specific timeouts.

## 5.1.3.  Batching Sends

   Sending multiple Messages can incur high overhead if each needs to be
   enqueued separately (e.g., each Message might involve a context
   switch between the application and the Transport Services System).
   To avoid this, the application can indicate a batch of Send actions
   through the API.  When this is used, the implementation can defer the
   processing of Messages until the batch is complete.

## 5.2.  Receiving Messages

   Similar to sending, receiving a Message is determined by the top-
   level protocol in the established Protocol Stack.  The main
   difference with receiving is that the size and boundaries of the
   Message are not known beforehand.  The application can communicate in
   its Receive action the parameters for the Message, which can help the
   Transport Services 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 Message
   Framer or because the top-level protocol in the established Protocol
   Stack preserves message boundaries.  The application can also control

the flow of received data by specifying the minimum and maximum
number of bytes of Message content it wants to receive at one time.

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

## 5.3.  Handling of data for fast-open protocols

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

The application can express its preference for sending messagess as
0-RTT data by using the zeroRttMsg Selection Property on the
Preconnection.  Then, the application can provide the message to send
as 0-RTT data via the InitiateWithSend action.  In order to be sent
as 0-RTT data, the message needs to be marked with the
safelyReplayable send paramteter.  In general, 0-RTT data may be
replayed (for example, if a TCP SYN contains data, and the SYN is
retransmitted, the data will be retransmitted as well but may be
considered as a new connection instead of a retransmission).  When
racing connections, different leaf nodes have the opportunity to send
the same data independently.  If data is truly safely replayable,
this is permissible.

Once the application has provided its 0-RTT data, a Transport
Services Implementation should keep a copy of this data and provide
it to each new leaf node that is started and for which a protocol
instance supporting 0-RTT is being used.  Note that the amount of
data that can actually be sent as 0-RTT data varies by protocol, so
any given Protocol Stack might only consume part of the saved data
prior to becoming established.  The implementation needs to keep
track of how much data a particular Protocol Stack has consumed, and
ensure that any pending 0-RTT-eligible data from the application is
handled before subsequent Messages.

It is also possible for Protocol Stacks within a particular leaf node
to use a 0-RTT handshakes in a lower-level protocol without any
safely replayable application data if a higher-level protocol in the
stack has idempotent handshake data to send.  For example, TCP Fast
Open could use a Client Hello from TLS as its 0-RTT data, without any
data being provided by the application.

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

6.  Implementing Message Framers

   Message Framers are functions that define simple transformations between application Message data and raw transport protocol data. Generally, a Message Framer implements a simple application protocol that can either be provided by the Transport Services implementation or by the application.  It is optional for Transport Services system implementations to provide Message Framers: the specification [I-D.ietf-taps-interface] does not prescribe any particular Message Framers to be implemented.  A Framer can encapsulate or encode outbound Messages, decapsulate or decode inbound data into Messages, and implement parts of protocols that do not directly map to application Messages (such as protocol handshakes or preludes before Message exchange).

   While many protocols can be represented as Message Framers, for the purposes of the Transport Services API, these are ways for applications or application frameworks to define their own Message parsing to be included within a Connection's Protocol Stack.  As an example, TLS is a protocol that is by default built into the Transport Services API, even though it could also serve the purpose of framing data over TCP.

   Most Message Framers fall into one of two categories:

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

   *  Delimiter-separated formats, such as HTTP/1.1

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

6.1.  Defining Message Framers

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

   The API examples in this section use the notation conventions for the
   Transport Services API defined in Section 1.1 of
   [I-D.ietf-taps-interface].

   The Transport Services Implementation needs to ensure that all of the
   events and actions taken on a Message Framer are synchronized to
   ensure consistent behavior.  For example, some of the actions defined
   below (such as PrependFramer and StartPassthrough) modify how data
   flows in a protocol stack, and require synchronization with sending
   and parsing data in the Message Framer.

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

   MessageFramer -> Start<connection>
   MessageFramer -> Stop<connection>

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

   Once the framer implementation has completed its setup or handshake,
   it can indicate to the application that it is ready for handling data
   with this call.

   MessageFramer.MakeConnectionReady(connection)

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

MessageFramer.MakeConnectionClosed(connection)

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

MessageFramer.FailConnection(connection, error)

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

Before an implementation marks a Message Framer as ready, it can also
dynamically add a protocol or framer above it in the stack.  This
allows protocols that need to add TLS conditionally, like STARTTLS
[RFC3207], to modify the Protocol Stack based on a handshake result.

otherFramer := NewMessageFramer()
MessageFramer.PrependFramer(connection, otherFramer)

A Message Framer might also choose to go into a passthrough mode once
an initial exchange or handshake has been completed, such as the
STARTTLS case mentioned above.  This can also be useful for proxy
protocols like SOCKS [RFC1928] or HTTP CONNECT [RFC7230].  In such
cases, a Message Framer implementation can intercept sending and
receiving of Messages at first, but then indicate that no more
processing is needed.

MessageFramer.StartPassthrough()

6.2.  Sender-side Message Framing

Message Framers generate an event whenever a Connection sends a new
Message.  The parameters to the event align with the Send action in
the API (Section 9.2 of [I-D.ietf-taps-interface]).

```
                         MessageFramer
                              |
                              V
NewSentMessage<connection, messageData, messageContext, endOfMessage>
```

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

    MessageFramer.Send(connection, messageData)

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

6.3.  Receiver-side Message Framing

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

The parameters to the events and calls for receiving data with a
framer align with the Receive action in the API (Section 9.3 of
[I-D.ietf-taps-interface]).

    MessageFramer -> HandleReceivedData<connection>

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

```
 MessageFramer.Parse(connection, minimumIncompleteLength, maximumLength)
                                 |
                                 V
            (messageData, messageContext, endOfMessage)
```

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

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

```
MessageFramer.AdvanceReceiveCursor(connection, length)
MessageFramer.DeliverAndAdvanceReceiveCursor(connection, messageContext, length,
endOfMessage)
MessageFramer.Deliver(connection, messageContext, messageData, endOfMessage)
```

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

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

7.   Implementing Connection Management

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

   The set of Connection Properties that are supported for setting and
   getting on a Connection are described in [I-D.ietf-taps-interface].
   For any properties that are generic, and thus could apply to all
   protocols being used by a Connection, the Transport Services
   Implementation should store the properties in storage common to all
   protocols, and notify the Protocol Stack as a whole whenever the
   properties have been modified by the application.  [RFC8303] and
   [RFC8304] offer guidance on how to do this for TCP, MPTCP, SCTP, UDP
   and UDP-Lite; see Section 10 for a description of a back-tracking
   method to find the relevant protocol primitives using these
   documents.  For Protocol-specific Properties, such as the User
   Timeout that applies to TCP, the Transport Services Implementation
   only needs to update the relevant protocol instance.

   Some Connection Properties might apply to multiple protocols within a
   Protocol Stack.  Depending on the specific property, it might be
   appropriate to apply the property across multiple protocols
   simultaneously, or else only apply it to one protocol.  In general,
   the Transport Services Implementation should allow the protocol
   closest to the application to interpret Connection Properties, and
   potentially modify the set of Connection Properties passed down to
   the next protocol in the stack.  For example, if the application has

requested to use keepalives with the keepAlive property, and the
Protocol Stack contains both HTTP/2 and TCP, the HTTP/2 protocol can
choose to enable its own keepalives to satisfy the application
request, and disable TCP-level keepalives.  For cases where the
application needs to have fine-grained per-protocol control, the
Transport Services Implementation can expose Protocol-specific
Properties.

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

When protocol instances in the Protocol Stack report generic or
protocol-specific errors, the API will deliver them to the
application as SoftError events.  These allow the application to be
informed of ICMP errors, and other similar events.

## 7.1.  Pooled Connection

For applications that do not need in-order delivery of Messages, the
Transport Services Implementation may distribute Messages of a single
Connection across several underlying transport connections or
multiple streams of multi-streaming connections between endpoints, as
long as all of these satisfy the Selection Properties.  The Transport
Services Implementation will then hide this connection management and
only expose a single Connection object, which we here call a "Pooled
Connection".  This is in contrast to Connection Groups, which
explicitly expose combined treatment of Connections, giving the
application control over multiplexing, for example.

Pooled Connections can be useful when the application using the
Transport Services system implements a protocol such as HTTP, which
employs request/response pairs and does not require in-order delivery
of responses.  This enables implementations of Transport Services
systems to realize transparent connection coalescing, connection
migration, and to perform per-message endpoint and path selection by
choosing among multiple underlying connections.

## 7.2.  Handling Path Changes

When a path change occurs, e.g., when the IP address of an interface
changes or a new interface becomes available, the Transport Services
Implementation is responsible for notifying the Protocol Instance of
the change.  The path change may interrupt connectivity on a path for
an active Connection or provide an opportunity for a transport that
supports multipath or migration to adapt to the new paths.  Note

that, in the model of the Transport Services API, migration is
considered a part of multipath connectivity; it is just a limiting
policy on multipath usage.  If the multipath Selection Property is
set to Disabled, migration is disallowed.

For protocols that do not support multipath or migration, the
Protocol Instances should be informed of the path change, but should
not be forcibly disconnected if the previously used path becomes
unavailable.  There are many common usage scenarios that can lead to
a path becoming temporarily unavailable, and then recovering before
the transport protocol reaches a timeout error.  These are
particularly common using mobile devices.  Examples include: an
Ethernet cable becoming unplugged and then plugged back in; a device
losing a Wi-Fi signal while a user is in an elevator, and reattaching
when the user leaves the elevator; and a user losing the radio signal
while riding a train through a tunnel.  If the device is able to
rejoin a network with the same IP address, a stateful transport
connection can generally resume.  Thus, while it is useful for a
Protocol Instance to be aware of a temporary loss of connectivity,
the Transport Services Implementation should not aggressively close
Connections in these scenarios.

If the Protocol Stack includes a transport protocol that supports
multipath connectivity, the Transport Services Implementation should
also inform the Protocol Instance about potentially new paths that
become permissible based on the multipath Selection Property and the
multipathPolicy Connection Property choices made by the application.
A protocol can then establish new subflows over new paths while an
active path is still available or, if migration is supported, also
after a break has been detected, and should attempt to tear down
subflows over paths that are no longer used.  The Connection Property
multipathPolicy of the Transport Services API allows an application
to indicate when and how different paths should be used.  However,
detailed handling of these policies is implementation-specific.  For
example, if the multipath Selection Property is set to active, the
decision about when to create a new path or to announce a new path or
set of paths to the Remote Endpoint, e.g., in the form of additional
IP addresses, is implementation-specific.  If the Protocol Stack
includes a transport protocol that does not support multipath, but
does support migrating between paths, the update to the set of
available paths can trigger the connection to be migrated.

In the case of a Pooled Connection Section 7.1, the Transport
Services Implementation may add connections over new paths to the
pool if permissible based on the multipath policy and Selection
Properties.  In the case that a previously used path becomes
unavailable, the Transport Services system may disconnect all
connections that require this path, but should not disconnect the

pooled Connection object exposed to the application.  The strategy to
do so is implementation-specific, but should be consistent with the
behavior of multipath transports.

8.  Implementing Connection Termination

For Close (which leads to a Closed event) and Abort (which leads to a
ConnectionError event), the application might find it useful 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.  When an underlying transport connection supports multi-
streaming (such as SCTP), the Transport Services system can use a
stream reset procedure to cause a Finish event upon a Close action
from the peer [NEAT-flow-mapping].

9.  Cached State

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

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

9.1.  Protocol state caches

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

Examples of cached protocol state include:

*  The DNS protocol can cache resolved addresses (such as those
   retrieved from A and AAAA queries), 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 Endpoint Identifers
   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 can use the Transport Services API to request that a
   Connection Group maintain a separate cache for protocol state.
   Connections in the group will not use cached state from Connections
   outside the group, and Connections outside the group will not use
   state cached from Connections inside the group.  This may be
   necessary, for example, if application-layer identifiers rotate and
   clients wish to avoid linkability via trackable TLS tickets or TFO
   cookies.

9.2.  Performance caches

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

   *  Observed Round Trip Time

   *  Connection establishment latency

   *  Connection establishment success rate

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

   An implementation should use this information, when possible, to
   influence 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 4.2) to ensure better performance for the application.

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

[RFC9040] provides guidance about sharing of TCP Control Block information between connections on initialization.

10.  Specific Transport Protocol Considerations

   Each protocol that is supported by a Transport Services Implementation should have a well-defined API mapping.  API mappings for a protocol are important for Connections in which a given protocol is the "top" of the Protocol Stack.  For example, the mapping of the Send function for TCP applies to Connections in which the application directly sends over TCP.

   Each protocol has a notion of Connectedness.  Possible definitions of Connectedness for various types of protocols are:

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

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

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

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

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

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

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

Below, terms in capitals with a dot (e.g., "CONNECT.SCTP") refer to
the primitives with the same name in Section 4 of [RFC8303].  For
further implementation details, the description of these primitives
in [RFC8303] points to Section 3 of [RFC8303] and Section 3 of
[RFC8304], which refers back to the relevant specifications for each
protocol.  This back-tracking method applies to all elements of
[RFC8923] (see appendix D of [I-D.ietf-taps-interface]): they are
listed in appendix A of [RFC8923] with an implementation hint in the
same style, pointing back to Section 4 of [RFC8303].

This document presents the protocol mappings defined in [RFC8923].
Other protocol mappings can be provided as separate documents,
following the mapping template in Appendix A.

10.1.  TCP

Connectedness: Connected

Data Unit: Byte-stream

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

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

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

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

EstablishmentError:  Failure of CONNECT.TCP.  TCP can throw various

errors during connection setup.  Specifically, it is important to
handle a RST being sent by the peer during the handshake.

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

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

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

Clone:  Calling Clone on a TCP Connection creates a new Connection
with equivalent parameters.  These Connections, and Connections
generated via later calls to Clone on an Established Connection,
form a Connection Group.  To realize entanglement for these
Connections, with the exception of connPriority, changing a
Connection Property on one of them must affect the Connection
Properties of the others too.  No guarantees of honoring the
Connection Property connPriority are given, and thus it is safe
for an implementation of a Transport Services system to ignore
this property.  When it is reasonable to assume that Connections
traverse the same path (e.g., when they share the same
encapsulation), support for it can also experimentally be
implemented using a congestion control coupling mechanism (see for
example [TCP-COUPLING] or [RFC3124]).

Send:  SEND.TCP.  TCP does not on its own preserve message
boundaries.  Calling Send on a TCP connection lays out the bytes
on the TCP send stream without any other delineation.  Any Message
marked as Final will cause TCP to send a FIN once the Message has
been completely written, by calling CLOSE.TCP immediately upon
successful termination of SEND.TCP.  Note that transmitting a
Message marked as Final should not cause the Closed event to be
delivered to the application, as it will still be possible to
receive data until the peer closes or aborts the TCP connection.

Receive:  With RECEIVE.TCP, TCP delivers a stream of bytes without
any Message delineation.  All data delivered in the Received or
ReceivedPartial event will be part of a single stream-wide Message
that is marked Final (unless a Message Framer is used).
EndOfMessage will be delivered when the TCP Connection has
received a FIN (CLOSE-EVENT.TCP) from the peer.  Note that
reception of a FIN should not cause the Closed event to be
delivered to the application, as it will still be possible for the
application to send data.

Close:  Calling Close on a TCP Connection indicates that the
    Connection should be gracefully closed (CLOSE.TCP) by sending a
    FIN to the peer.  It will then still be possible to receive data
    until the peer closes or aborts the TCP connection.  The Closed
    event will be issued upon reception of a FIN.

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

CloseGroup:  Calling CloseGroup on a TCP Connection (CLOSE.TCP) is
    identical to calling Close on this Connection and on all
    Connections in the same ConnectionGroup.

AbortGroup:  Calling AbortGroup on a TCP Connection (ABORT.TCP) is
    identical to calling Abort on this Connection and on all
    Connections in the same ConnectionGroup.

## 10.2.  MPTCP

Connectedness: Connected

Data Unit: Byte-stream

The Transport Services API mappings for MPTCP are identical to TCP.
MPTCP adds support for multipath properties, such as multipath and
multipathPolicy, and actions for managing paths, such as AddRemote
and RemoveRemote.

## 10.3.  UDP

Connectedness: Connectionless

Data Unit: Datagram

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

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

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

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

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

ConnectionError:  UDP Connections can only generate Connection errors
   in response to Abort calls.  (Once in use, UDP Connections can
   also generate SoftError events (ERROR.UDP) upon receiving ICMP
   notifications indicating failures in the network.)

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

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

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

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

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

Close:  Calling Close on a UDP Connection (ABORT.UDP) releases the
   local port reservation.  The Connection then issues a Closed
   event.

Abort:  Calling Abort on a UDP Connection (ABORT.UDP) is identical to
   calling Close, except that the Connection will send a
   ConnectionError event rather than a Closed event.

CloseGroup:  Calling CloseGroup on a UDP Connection (ABORT.UDP) is
   identical to calling Close on this Connection and on all
   Connections in the same ConnectionGroup.

AbortGroup:  Calling AbortGroup on a UDP Connection (ABORT.UDP) is
   identical to calling Close on this Connection and on all
   Connections in the same ConnectionGroup.

10.4.  UDP-Lite

   Connectedness: Connectionless

   Data Unit: Datagram

   The Transport Services API mappings for UDP-Lite are identical to
   UDP.  In addition, UDP-Lite supports the msgChecksumLen and
   recvChecksumLen Properties that allow an application to specify the
   minimum number of bytes in a Message that need to be covered by a
   checksum.

   This includes: CONNECT.UDP-Lite; LISTEN.UDP-Lite; SEND.UDP-Lite;
   RECEIVE.UDP-Lite; ABORT.UDP-Lite; ERROR.UDP-Lite; SET_DSCP.UDP-Lite;
   SET_DF.UDP-Lite; SET_ECN.UDP-Lite; GET_ECN.UDP-Lite.

10.5.  UDP Multicast Receive

   Connectedness: Connectionless

   Data Unit: Datagram

   Connection Object:  Established UDP Multicast Receive connections
      represent a pair of specific IP addresses and ports.  The
      direction Selection Property must be set to unidirectional
      receive, and the Local Endpoint must be configured with a group IP
      address and a port.

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

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

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

   EstablishmentError:  UDP Multicast Receive Connections generate an
      EstablishmentError indicating that joining a multicast group
      failed if Initiate is called.

   ConnectionError:  The only ConnectionError generated by a UDP
      Multicast Receive Connection is in response to an Abort call.

   Listen:  LISTEN.UDP.  Calling Listen for UDP Multicast Receive binds

a local port, prepares it to receive inbound UDP datagrams from
peers, and issues a multicast host join.  If a Remote Endpoint
Identifer with an address is supplied, the join is Source-specific
Multicast, and the path selection is based on the route to the
Remote Endpoint.  If a Remote Endpoint Identifer is not supplied,
the join is Any-source Multicast, and the path selection is based
on the outbound route to the group supplied in the Local Endpoint.

There are cases where it is required to open multiple connections for
the same address(es).  For example, one Connection might be opened
for a multicast group to for a multicast control bus, and another
application later opens a separate Connection to the same group to
send signals to and/or receive signals from the common bus.  In such
cases, the Transport Services system needs to explicitly enable re-
use of the same set of addresses (equivalent to setting SO_REUSEADDR
in the socket API).

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

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

Send:  SEND.UDP.  Calling Send on a UDP Multicast Receive connection
   causes an immediate SendError.  This is an unsupported operation.

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

Close:  Calling Close on a UDP Multicast Receive Connection
   (ABORT.UDP) releases the local port reservation and leaves the
   group.  The Connection then issues a Closed event.

Abort:  Calling Abort on a UDP Multicast Receive Connection
   (ABORT.UDP) is identical to calling Close, except that the
   Connection will send a ConnectionError event rather than a Closed
   event.

CloseGroup:  Calling CloseGroup on a UDP Multicast Receive Connection
   (ABORT.UDP) is identical to calling Close on this Connection and
   on all Connections in the same ConnectionGroup.

AbortGroup:  Calling AbortGroup on a UDP Multicast Receive Connection

        (ABORT.UDP) is identical to calling Close on this Connection and
        on all Connections in the same ConnectionGroup.

10.6.  SCTP

     Connectedness: Connected

     Data Unit: Message

     Connection Object:  Connection objects can be mapped to an SCTP
        association or a stream in an SCTP association.  Mapping
        Connection objects to SCTP streams is called "stream mapping" and
        has additional requirements as follows.  The following explanation
        assumes a client-server communication model.

     Stream mapping requires an association to already be in place between
     the client and the server, and it requires the server to understand
     that a new incoming stream should be represented as a new Connection
     object by the Transport Services system.  A new SCTP stream is
     created by sending an SCTP message with a new stream id.  Thus, to
     implement stream mapping, the Transport Services API must provide a
     newly created Connection object to the application upon the reception
     of such a message.  The necessary semantics to implement a Transport
     Services system's Close and Abort primitives are provided by the
     stream reconfiguration (reset) procedure described in [RFC6525].
     This also allows to re-use a stream id after resetting ("closing")
     the stream.  To implement this functionality, SCTP stream
     reconfiguration [RFC6525] must be supported by both the client and
     the server side.

     To avoid head-of-line blocking, stream mapping should only be
     implemented when both sides support message interleaving [RFC8260].
     This allows a sender to schedule transmissions between multiple
     streams without risking that transmission of a large message on one
     stream might block transmissions on other streams for a long time.

     To avoid conflicts between stream ids, the following procedure is
     recommended: the first Connection, for which the SCTP association has
     been created, must always use stream id zero.  All additional
     Connections are assigned to unused stream ids in growing order.  To
     avoid a conflict when both endpoints map new Connections
     simultaneously, the peer which initiated association must use even
     stream ids whereas the remote side must map its Connections to odd
     stream ids.  Both sides maintain a status map of the assigned stream
     ids.  Generally, new streams should consume the lowest available
     (even or odd, depending on the side) stream id; this rule is relevant
     when lower ids become available because Connection objects associated
     with the streams are closed.

SCTP stream mapping as described here has been implemented in a
research prototype; a desription of this implementation is given in
[NEAT-flow-mapping].

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

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

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

EstablishmentError:  Failure of CONNECT.SCTP.

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

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

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

Clone:  Calling Clone on an SCTP association creates a new Connection
   object and assigns it a new stream id in accordance with the
   stream id assignment procedure described above.  If there are not
   enough streams available, ADD_STREAM.SCTP must be called.

Send:  SEND.SCTP.  Message Properties such as msgLifetime and

   msgOrdered map to parameters of this primitive.

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

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

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

   CloseGroup: Calling CloseGroup calls CLOSE.SCTP, closing all
   Connections in the SCTP association.

   AbortGroup: Calling AbortGroup calls ABORT.SCTP, immediately closing
   all Connections in the SCTP association.

   In addition to the API mappings described above, when there are
   multiple Connection objects assigned to the same SCTP association,
   SCTP can support Connection properties such as connPriority and
   connScheduler where CONFIGURE_STREAM_SCHEDULER.SCTP can be called to
   adjust the priorities of streams in the SCTP association.

11.  IANA Considerations

   This document has no actions for IANA.

12.  Security Considerations

   [I-D.ietf-taps-arch] outlines general security consideration and
   requirements for any system that implements the Transport Services
   archtecture.  [I-D.ietf-taps-interface] provides further discussion
   on security and privacy implications of the Transport Services API.
   This document provides additional guidance on implementation
   specifics for the Transport Services API and as such the security

considerations in both of these documents apply.  The next two
subsections discuss further considerations that are specific to
mechanisms specified in this document.

## 12.1.  Considerations for Candidate Gathering

The Security Considerations of the Transport Services Architecture
[I-D.ietf-taps-arch] forbids gathering and racing with Protocol
Stacks that do not have equivalent security properties.  Therefore,
implementations need to avoid downgrade attacks that allow network
interference to cause the implementation to select less secure, or
entirely insecure, combinations of paths and protocols.

## 12.2.  Considerations for Candidate Racing

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

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

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

## 13.  Acknowledgements

14.  References

14.1.  Normative References

[I-D.ietf-taps-arch]
          Pauly, T., Trammell, B., Brunstrom, A., Fairhurst, G., and
          C. Perkins, "Architecture and Requirements for Transport
          Services", Work in Progress, Internet-Draft, draft-ietf-
          taps-arch-19, 9 November 2023,
          <https://datatracker.ietf.org/doc/html/draft-ietf-taps-
          arch-19>.

[I-D.ietf-taps-interface]
          Trammell, B., Welzl, M., Enghardt, R., Fairhurst, G.,
          Kühlewind, M., Perkins, C., Tiesel, P. S., and T. Pauly,
          "An Abstract Application Layer Interface to Transport
          Services", Work in Progress, Internet-Draft, draft-ietf-
          taps-interface-23, 14 November 2023,
          <https://datatracker.ietf.org/doc/html/draft-ietf-taps-
          interface-23>.

[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/rfc/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/rfc/rfc7540>.

[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/rfc/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/rfc/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/rfc/rfc8305>.

   [RFC8421]  Martinsen, P., Reddy, T., and P. Patil, "Guidelines for
              Multihomed and IPv4/IPv6 Dual-Stack Interactive
              Connectivity Establishment (ICE)", BCP 217, RFC 8421,
              DOI 10.17487/RFC8421, July 2018,
              <https://www.rfc-editor.org/rfc/rfc8421>.

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

   [RFC8923]  Welzl, M. and S. Gjessing, "A Minimal Set of Transport
              Services for End Systems", RFC 8923, DOI 10.17487/RFC8923,
              October 2020, <https://www.rfc-editor.org/rfc/rfc8923>.

14.2.  Informative References

   [I-D.ietf-dnsop-svcb-https]
              Schwartz, B. M., Bishop, M., and E. Nygren, "Service
              Binding and Parameter Specification via the DNS (SVCB and
              HTTPS Resource Records)", Work in Progress, Internet-
              Draft, draft-ietf-dnsop-svcb-https-12, 11 March 2023,
              <https://datatracker.ietf.org/doc/html/draft-ietf-dnsop-
              svcb-https-12>.

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

   [RFC1928]  Leech, M., Ganis, M., Lee, Y., Kuris, R., Koblas, D., and
              L. Jones, "SOCKS Protocol Version 5", RFC 1928,
              DOI 10.17487/RFC1928, March 1996,
              <https://www.rfc-editor.org/rfc/rfc1928>.

   [RFC2782]  Gulbrandsen, A., Vixie, P., and L. Esibov, "A DNS RR for
              specifying the location of services (DNS SRV)", RFC 2782,
              DOI 10.17487/RFC2782, February 2000,
              <https://www.rfc-editor.org/rfc/rfc2782>.

   [RFC3124]  Balakrishnan, H. and S. Seshan, "The Congestion Manager",
              RFC 3124, DOI 10.17487/RFC3124, June 2001,
              <https://www.rfc-editor.org/rfc/rfc3124>.

   [RFC3207]  Hoffman, P., "SMTP Service Extension for Secure SMTP over
              Transport Layer Security", RFC 3207, DOI 10.17487/RFC3207,
              February 2002, <https://www.rfc-editor.org/rfc/rfc3207>.

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

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

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

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

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

   [RFC7230]  Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer
              Protocol (HTTP/1.1): Message Syntax and Routing",
              RFC 7230, DOI 10.17487/RFC7230, June 2014,
              <https://www.rfc-editor.org/rfc/rfc7230>.

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

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

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

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

   [RFC9000]  Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based
              Multiplexed and Secure Transport", RFC 9000,
              DOI 10.17487/RFC9000, May 2021,
              <https://www.rfc-editor.org/rfc/rfc9000>.

   [RFC9040]  Touch, J., Welzl, M., and S. Islam, "TCP Control Block
              Interdependence", RFC 9040, DOI 10.17487/RFC9040, July
              2021, <https://www.rfc-editor.org/rfc/rfc9040>.

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

Appendix A.  API Mapping Template

   Any protocol mapping for the Transport Services API should follow a
   common template.

   Connectedness: (Connectionless/Connected/Multiplexing Connected)

   Data Unit: (Byte-stream/Datagram/Message)

   Connection Object:

   Initiate:

   InitiateWithSend:

   Ready:

   EstablishmentError:

   ConnectionError:

   Listen:

   ConnectionReceived:

   Clone:

   Send:

   Receive:

   Close:

   Abort:

   CloseGroup:

   AbortGroup:

Appendix B.  Reasons for errors

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

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

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

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

   *  EstablishmentFailed: The Transport Services system was unable to
      establish a transport-layer connection to the Remote Endpoint
      specified by the application.

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

   *  NotCloneable: The Protocol Stack is not capable of being cloned.

   *  MessageTooLarge: The Message is too big for the Transport Services
      system to handle.

   *  ProtocolFailed: The underlying Protocol Stack failed.

   *  InvalidMessageProperties: The Message Properties either contradict
      the Transport Properties or they can not be satisfied by the
      Transport Services system.

   *  DeframingFailed: The data that was received by the underlying
      Protocol Stack could not be processed by the Message Framer.

   *  ConnectionAborted: The connection was aborted by the peer.

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

Appendix C.  Existing Implementations

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

   *  Apple's Network.framework:

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

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

   *  NEAT and NEATPy:

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

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

      -  Code at the Software Heritage Archive:
         https://archive.softwareheritage.org/swh:1:dir:737820840f83c4ec
         9493a8c0cc89b3159e2e1a57;origin=https://github.com/NEAT-
         project/neat;visit=swh:1:snp:bbb611b04e355439d47e426e8ad5d07cdb
         f647e0;anchor=swh:1:rev:652ee991043ce3560a6e5715fa2a5c211139d15
         c (https://archive.softwareheritage.org/swh:1:dir:737820840f83c
         4ec9493a8c0cc89b3159e2e1a57;origin=https://github.com/NEAT-
         project/neat;visit=swh:1:snp:bbb611b04e355439d47e426e8ad5d07cdb
         f647e0;anchor=swh:1:rev:652ee991043ce3560a6e5715fa2a5c211139d15
         c)

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

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

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

   -  Code at the Software Heritage Archive:
      https://archive.softwareheritage.org/swh:1:dir:295ccd148cf918cc
      b9ed7ad14b5ae968a8d2c370;origin=https://github.com/
      theagilepadawan/NEATPy;visit=swh:1:snp:6e1a3a9dd4c532ba6c0f52c8
      f734c1256a06cedc;anchor=swh:1:rev:cd0788d7f7f34a0e9b8654516da7c
      002c44d2e95 (https://archive.softwareheritage.org/swh:1:dir:295
      ccd148cf918ccb9ed7ad14b5ae968a8d2c370;origin=https://github.com
      /theagilepadawan/NEATPy;visit=swh:1:snp:6e1a3a9dd4c532ba6c0f52c
      8f734c1256a06cedc;anchor=swh:1:rev:cd0788d7f7f34a0e9b8654516da7
      c002c44d2e95)

*  PyTAPS:

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

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

   -  Code at the Software Heritage Archive:
      https://archive.softwareheritage.org/swh:1:dir:a7151096d91352b4
      39b092ef116d04f38e52e556;origin=https://github.com/fg-inet/
      python-asyncio-taps;visit=swh:1:snp:4841e59b53b28bb385726e7d3a5
      69bee0fea7fc4;anchor=swh:1:rev:63571fd7545da25142bc1a6371b8f130
      97cba38e (https://archive.softwareheritage.org/swh:1:dir:a71510
      96d91352b439b092ef116d04f38e52e556;origin=https://github.com/
      fg-inet/python-asyncio-taps;visit=swh:1:snp:4841e59b53b28bb3857
      26e7d3a569bee0fea7fc4;anchor=swh:1:rev:63571fd7545da25142bc1a63
      71b8f13097cba38e)

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


Reese Enghardt
Netflix
121 Albright Way
Los Gatos, CA 95032,
United States of America
Email: ietf@tenghardt.net


Philipp S. Tiesel
SAP SE
George-Stephenson-Straße 7-13
10557 Berlin
Germany
Email: philipp@tiesel.net


Michael Welzl
University of Oslo
PO Box 1080 Blindern
0316  Oslo
Norway
Email: michawe@ifi.uio.no