        An Abstract Application Layer Interface to Transport Services
                    draft-ietf-taps-interface-04

Abstract

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

Status of This Memo

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

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

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

   This Internet-Draft will expire on January 9, 2020.

Copyright Notice

Table of Contents

1. Introduction

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

a file, and stream abstractions are too simplistic for many modern
application programming models.

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

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

2. Terminology and Notation

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

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

o An Action creates an Object:

Object := Action()

o An Action creates an array of Objects:

[]Object := Action()

o An Action is performed on an Object:

Object.Action()

o An Object sends an Event:

Object -> Event<>

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

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

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

How these abstract concepts map into concrete implementations of this
API in a given language on a given platform is largely dependent on
the features of the language and the platform. Actions could be
implemented as functions or method calls, for instance, and Events
could be implemented via callbacks, communicating sequential
processes, or other asynchronous calling conventions. The method for
dispatching and handling Events is left as an implementation detail,
with the caveat that the interface for receiving Messages must
require the application to invoke the Connection.Receive() Action
once per Message to be received (see Section 8).

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

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

3. Interface Design Principles

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

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

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

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

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

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

[4](#). **API Summary**

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

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

   Once a Connection is established, data can be sent on it in the form
   of Messages. The interface supports the preservation of message
   boundaries both via explicit Protocol Stack support, and via
   application support through a Message Framer which finds message
   boundaries in a stream. Messages are received asynchronously through
   a callback registered by the application. Errors and other
   notifications also happen asynchronously on the Connection.

   [Section 5](#), [Section 6](#), [Section 7](#), [Section 8](#), and [Section 12](#) describe
   the details of application interaction with Objects through Actions
   and Events in each phase of a Connection, following the phases
   described in [[I-D.ietf-taps-arch](#)].

4.1. Usage Examples

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

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

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

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

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

If none of the available transport protocols provides Preservation of
Message Boundaries, but there is a transport protocol which provides
a reliable ordered byte stream, an application may receive this byte
stream as partial Messages and transform it into application-layer
Messages. Alternatively, an application may provide a Message
Framer, which can transform a byte stream into a sequence of Messages
(Section 10.6).

4.1.1. Server Example

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

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

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

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

// Specifying a remote endpoint is optional when using Listen()
Preconnection := NewPreconnection(LocalSpecifier,
                                  None,
                                  TransportProperties,
                                  SecurityParameters)

Listener := Preconnection.Listen()

Listener -> ConnectionReceived<Connection>

// Only receive complete messages
Connection.Receive()

Connection -> Received(messageDataRequest, messageContext)

Connection.Send(messageDataResponse)

Connection.Close()

// Stop listening for incoming Connections
Listener.Stop()
```

4.1.2. Client Example

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

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

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

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

// Specifying a local endpoint is optional when using Initiate()
Preconnection := NewPreconnection(None,
                                  RemoteSpecifier,
                                  TransportPreperties,
                                  SecurityParameters)

Connection := Preconnection.Initiate()

Connection -> Ready<>

Connection.Send(messageDataRequest)

// Only receive complete messages
Connection.Receive()

Connection -> Received(messageDataResponse, messageContext)

Connection.Close()
```

4.1.3. Peer Example

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

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

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

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

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

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

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

   Preconnection.Rendezvous()

   Preconnection -> RendezvousDone<Connection>

   Connection.Send(messageDataRequest)

   // Only receive complete messages
   Connection.Receive()

   Connection -> Received(messageDataResponse, messageContext)

   Connection.Close()
```

4.2. Transport Properties

   Each application using the Transport Services Interface declares its
   preferences for how the transport service should operate using
   properties at each stage of the lifetime of a connection. During
   pre-establishment, Selection Properties (see Section 5.2) are used to
   specify which paths and protocol stacks can be used and are preferred
   by the application, and Connection Properties (see Section 11.1) can
   be used to influence decisions made during establishment and to fine-

tune the eventually established connection. These Connection
Properties can also be used later, to monitor and fine-tune
established connections. The behavior of the selected protocol
stack(s) when sending Messages is controlled by Message Properties
(see Section 7.4).

Collectively, Selection, Connection, and Message Properties can be
referred to as Transport Properties. All Transport Properties,
regardless of the phase in which they are used, are organized within
a single namespace. This enables setting them as defaults in earlier
stages and querying them in later stages: - Connection Properties can
be set on Preconnections - Message Properties can be set on
Preconnections and Connections - The effect of Selection Properties
can be queried on Connections and Messages

Note that Configuring Connection Properties and Message Properties on
Preconnections is preferred over setting them later. Connection
Properties specified early on may be used as additional input to the
selection process. Also note that Protocol Specific Properties, see
Section 4.2.1, should not be used as an input to the selection
process.

4.2.1. Transport Property Names

Transport Properties are referred to by property names. These names
are lower-case strings whereby words are separated by hyphens. These
names serve two purposes:

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

o Make code of different TAPS implementations look similar.

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

o The Namespace part is empty for well known, generic properties,
   i.e., for properties defined by an RFC which are not protocol
   specific.

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

   o Vendor or implementation specific properties must use a a string
      identifying the vendor or implementation as Namespace.

4.2.2. Transport Property Types

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

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

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

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

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

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

4.3. Scope of the Interface Definition

   This document defines a language- and platform-independent interface
   to a Transport Services system. Given the wide variety of languages
   and language conventions used to write applications that use the
   transport layer to connect to other applications over the Internet,
   this independence makes this interface necessarily abstract. While
   there is no interoperability benefit to tightly defining how the
   interface be presented to application programmers in diverse
   platforms, maintaining the "shape" of the abstract interface across
   these platforms reduces the effort for programmers who learn the
   transport services interface to apply their knowledge in multiple
   platforms. We therefore make the following recommendations:

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

   o Implementations of this interface SHOULD implement each Selection
      Property, Connection Property, and Message Context Property
      specified in this document, exclusive of appendices, even if said
      implementation is a non-operation, e.g. because transport
      protocols implementing a given Property are not available on the
      platform.

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

5. Pre-Establishment Phase

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

   A Preconnection Object represents a potential Connection. It has
   state that describes properties of a Connection that might exist in
   the future. This state comprises Local Endpoint and Remote Endpoint
   Objects that denote the endpoints of the potential Connection (see
   Section 5.1), the Selection Properties (see Section 5.2), any
   preconfigured Connection Properties (Section 11.1), and the security
   parameters (see Section 5.3):

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

   The Local Endpoint MUST be specified if the Preconnection is used to
   Listen() for incoming Connections, but is OPTIONAL if it is used to
   Initiate() connections. The Remote Endpoint MUST be specified if the
   Preconnection is used to Initiate() Connections, but is OPTIONAL if
   it is used to Listen() for incoming Connections. The Local Endpoint
   and the Remote Endpoint MUST both be specified if a peer-to-peer
   Rendezvous is to occur based on the Preconnection.

   Message Framers (see Section 10), if required, should be added to the
   Preconnection during pre-establishment.

5.1. Specifying Endpoints

   The transport services API uses the Local Endpoint and Remote
   Endpoint types to refer to the endpoints of a transport connection.
   Subtypes of these represent various different types of endpoint

identifiers, such as IP addresses, DNS names, and interface names, as
well as port numbers and service names.

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

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

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

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

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

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

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

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

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

5.2. Specifying Transport Properties

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

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

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

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

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

   Internally, the transport system will first exclude all protocols and
   paths that match a Prohibit, then exclude all protocols and paths
   that do not match a Require, then sort candidates according to
   Preferred properties, and then use Avoided properties as a
   tiebreaker. Selection Properties which select paths take preference

over those which select protocols. For example, if an application
indicates a preference for a specific path by specifying an
interface, but also a preference for a protocol not available on this
path, the transport system will try the path first, ignoring the
preference.

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

TransportProperties := NewTransportProperties()

Individual properties are then added to the TransportProperties
Object:

TransportProperties.Add(property, value)

Selection Properties can be added to a TransportProperties object
using special actions for each preference level i.e,
"TransportProperties.Add(some_property, avoid)" is equivalent to
"TransportProperties.Avoid(some_property)":

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

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

TransportProperties := Connection.GetTransportProperties()

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

Section 11.1 provides a list of Connection Properties, while
Selection Properties are listed in the subsections below. Note that
many properties are only considered during establishment, and can not
be changed after a Connection is established; however, they can be
queried. Querying a Selection Property after establishment yields
the value Required for properties of the selected protocol and path,

Avoid for properties avoided during selection, and Ignore for all
other properties.

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

5.2.1. Reliable Data Transfer (Connection)

Name: reliability

This property specifies whether the application needs to use a
transport protocol that ensures that all data is received on the
other side without corruption. This also entails being notified when
a Connection is closed or aborted. The recommended default is to
Require Reliable Data Transfer.

5.2.2. Preservation of Message Boundaries

Name: preserve-msg-boundaries

This property specifies whether the application needs or prefers to
use a transport protocol that preserves message boundaries. The
recommended default is to Prefer Preservation of Message Boundaries.

5.2.3. Configure Per-Message Reliability

Name: per-msg-reliability

This property specifies whether an application considers it useful to
indicate its reliability requirements on a per-Message basis. This
property applies to Connections and Connection Groups. The
recommended default is to Ignore this option.

5.2.4. Preservation of Data Ordering

Name: preserve-order

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

5.2.5. Use 0-RTT Session Establishment with an Idempotent Message

   Name: zero-rtt-msg

   This property specifies whether an application would like to supply a
   Message to the transport protocol before Connection establishment,
   which will then be reliably transferred to the other side before or
   during Connection establishment, potentially multiple times (i.e.,
   multiple copies of the message data may be passed to the Remote
   Endpoint). See also Section 7.4.4. The recommended default is to
   Ignore this option. Note that disabling this property has no effect
   for protocols that are not connection-oriented and do not protect
   against duplicated messages, e.g., UDP.

5.2.6. Multistream Connections in Group

   Name: multistreaming

   This property specifies that the application would prefer multiple
   Connections within a Connection Group to be provided by streams of a
   single underlying transport connection where possible. The
   recommended default is to Prefer this option.

5.2.7. Full Checksum Coverage on Sending

   Name: per-msg-checksum-len-send

   This property specifies whether the application desires protection
   against corruption for all data transmitted on this Connection.
   Disabling this property may enable to control checksum coverage later
   (see Section 7.4.6). The recommended default is to Require this
   option.

5.2.8. Full Checksum Coverage on Receiving

   Name: per-msg-checksum-len-recv

   This property specifies whether the application desires protection
   against corruption for all data received on this Connection. The
   recommended default is to Require this option.

5.2.9. Congestion control

   Name: congestion-control

   This property specifies whether the application would like the
   Connection to be congestion controlled or not. Note that if a
   Connection is not congestion controlled, an application using such a

Connection should itself perform congestion control in accordance
with [RFC2914]. Also note that reliability is usually combined with
congestion control in protocol implementations, rendering "reliable
but not congestion controlled" a request that is unlikely to succeed.
The recommended default is to Require that the Connection is
congestion controlled.

5.2.10. Interface Instance or Type

Name: interface

Type: Set (Preference, Enumeration)

This property allows the application to select which specific network
interfaces or categories of interfaces it wants to "Require",
"Prohibit", "Prefer", or "Avoid".

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

Note that marking a specific interface as "Required" strictly limits
path selection to a single interface, and leads to less flexible and
resilient connection establishment.

The set of valid interface types is implementation- and system-
specific. For example, on a mobile device, there may be "Wi-Fi" and
"Cellular" interface types available; whereas on a desktop computer,
there may be "Wi-Fi" and "Wired Ethernet" interface types available.
Implementations should provide all types that are supported on some
system to all systems, in order to allow applications to write
generic code. For example, if a single implementation is used on
both mobile devices and desktop devices, it should define the
"Cellular" interface type for both systems, since an application may
want to always "Prohibit Cellular". Note that marking a specific
interface type as "Required" limits path selection to a small set of
interfaces, and leads to less flexible and resilient connection
establishment.

The set of interface types is expected to change over time as new
access technologies become available.

Interface types should not be treated as a proxy for properties of
interfaces such as metered or unmetered network access. If an
application needs to prohibit metered interfaces, this should be

specified via Provisioning Domain attributes (see Section 5.2.11) or
another specific property.

5.2.11. Provisioning Domain Instance or Type

   Name: pvd

   Type: Set (Preference, Enumeration)

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

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

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

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

5.2.12. Parallel Use of Multiple Paths

   Name: multipath

   This property specifies whether an application considers it useful to
   transfer data across multiple paths between the same end hosts.
   Generally, in most cases, this will improve performance (e.g.,
   achieve greater throughput). One possible side-effect is increased

jitter, which may be problematic for delay-sensitive applications.
The recommended default is to Prefer this option.

[5.2.13](#). Direction of communication

Name: direction

Type: Enumeration

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

Bidirectional (default): The connection must support sending and
   receiving data

Unidirectional send: The connection must support sending data

Unidirectional receive: The connection must support receiving data

In case a unidirectional connection is requested, but unidirectional
connections are not supported by the transport protocol, the system
should fall back to bidirectional transport.

[5.2.14](#). Notification of excessive retransmissions

Name: :retransmit-notify

This property specifies whether an application considers it useful to
be informed in case sent data was retransmitted more often than a
certain threshold. The recommended default is to Ignore this option.

[5.2.15](#). Notification of ICMP soft error message arrival

Name: :soft-error-notify

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

[5.3](#). Specifying Security Parameters and Callbacks

Most security parameters, e.g., TLS ciphersuites, local identity and
private key, etc., may be configured statically. Others are
dynamically configured during connection establishment. Thus, we

partition security parameters and callbacks based on their place in
the lifetime of connection establishment. Similar to Transport
Properties, both parameters and callbacks are inherited during
cloning (see Section 6.4).

5.3.1. Pre-Connection Parameters

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

SecurityParameters := NewSecurityParameters()

Security configuration parameters and sample usage follow:

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

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

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

SecurityParameters.AddSupportedGroup(secp256k1)
SecurityParameters.AddCiphersuite(TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256)
SecurityParameters.AddSignatureAlgorithm(ed25519)

o Session cache management: Used to tune cache capacity, lifetime,
   re-use, and eviction policies, e.g., LRU or FIFO. Constants and
   policies for these interfaces are implementation-specific.

SecurityParameters.SetSessionCacheCapacity(MAX_CACHE_ELEMENTS)
SecurityParameters.SetSessionCacheLifetime(SECONDS_PER_DAY)
SecurityParameters.SetSessionCachePolicy(CachePolicyOneTimeUse)

o Pre-Shared Key import: Used to install pre-shared keying material
   established out-of-band. Each pre-shared keying material is

associated with some identity that typically identifies its use or
has some protocol-specific meaning to the Remote Endpoint.

```
SecurityParameters.AddPreSharedKey(key, identity)
```

5.3.2. Connection Establishment Callbacks

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

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

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

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

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

6. Establishing Connections

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

6.1. Active Open: Initiate

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

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

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

The Initiate() Action consumes the Preconnection. Once Initiate()
has been called, no further properties may be added to the
Preconnection, and no subsequent establishment call may be made on
the Preconnection.

Once Initiate is called, the candidate Protocol Stack(s) may cause
one or more candidate transport-layer connections to be created to
the specified remote endpoint. The caller may immediately begin
sending Messages on the Connection (see Section 7) after calling
Initate(); note that any idempotent data sent while the Connection is
being established may be sent multiple times or on multiple
candidates.

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

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

The Ready Event occurs after Initiate has established a transport-
layer connection on at least one usable candidate Protocol Stack over
at least one candidate Path. No Receive Events (see Section 8) will
occur before the Ready Event for Connections established using
Initiate.

```
Connection -> InitiateError<>
```

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

See also Section 7.7 to combine Connection establishment and
transmission of the first message in a single action.

6.2. Passive Open: Listen

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

Listener := Preconnection.Listen()

Before calling Listen, the caller must have initialized the Preconnection during the pre-establishment phase with a Local Endpoint specifier, as well as all properties necessary for Protocol Stack selection. A Remote Endpoint may optionally be specified, to constrain what Connections are accepted. The Listen() Action returns a Listener object. Once Listen() has been called, properties added to the Preconnection have no effect on the Listener and the Preconnection can be disposed of or reused.

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

Listener.Stop()

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

Listener -> ConnectionReceived<Connection>

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

Listener -> ListenError<>

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

Listener -> Stopped<>

A Stopped event occurs after the Listener has stopped listening.

6.3. Peer-to-Peer Establishment: Rendezvous

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

Preconnection.Rendezvous()

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

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

The Rendezvous() Action consumes the Preconnection. Once
Rendezvous() has been called, no further properties may be added to
the Preconnection, and no subsequent establishment call may be made
on the Preconnection.

Preconnection -> RendezvousDone<Connection>

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

Preconnection -> RendezvousError<messageContext, error>

An RendezvousError occurs either when the Preconnection cannot be
fulfilled for listening, when the Local Endpoint or Remote Endpoint
cannot be resolved, when no transport-layer connection can be
established to the Remote Endpoint, or when the application is
prohibited from rendezvous by policy.

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

[]Preconnection := Preconnection.Resolve()

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

6.4. Connection Groups

Entangled Connections can be created using the Clone Action:

Connection := Connection.Clone()

Calling Clone on a Connection yields a group of two Connections: the
parent Connection on which Clone was called, and the resulting cloned
Connection. These connections are "entangled" with each other, and
become part of a Connection Group. Calling Clone on any of these two
Connections adds a third Connection to the Connection Group, and so
on. Connections in a Connection Group share all Protocol Properties
that are not applicable to a Message.

In addition, incoming entangled Connections can be received by
creating a Listener on an existing connection:

Listener := Connection.Listen()

Changing one of these Protocol Properties on one Connection in the
group changes it for all others. Per-Message Protocol Properties,
however, are not entangled. For example, changing "Timeout for
aborting Connection" (see Section 11.1.4) on one Connection in a
group will automatically change this Protocol Property for all
Connections in the group in the same way. However, changing
"Lifetime" (see Section 7.4.1) of a Message will only affect a single
Message on a single Connection, entangled or not.

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

If the underlying Protocol Stack does not support cloning, or cannot
create a new stream on the given Connection, then attempts to clone a
Connection will result in a CloneError:

```
Connection -> CloneError<>
```

The Protocol Property "Priority" operates on entangled Connections as
in [Section 7.4.2](): when allocating available network capacity among
Connections in a Connection Group, sends on Connections with higher
Priority values will be prioritized over sends on Connections with
lower Priority values. An ideal transport system implementation
would assign each Connection the capacity share $(M-N) \times C / M$, where
N is the Connection's Priority value, M is the maximum Priority value
used by all Connections in the group and C is the total available
capacity. However, the Priority setting is purely advisory, and no
guarantees are given about the way capacity is shared. Each
implementation is free to implement a way to share capacity that it
sees fit.

[7](). Sending Data

Once a Connection has been established, it can be used for sending
data. Data is sent as Messages, which allow the application to
communicate the boundaries of the data being transferred. By
default, Send enqueues a complete Message, and takes optional per-
Message properties (see [Section 7.1]()). All Send actions are
asynchronous, and deliver events (see [Section 7.3]()). Sending partial
Messages for streaming large data is also supported (see
[Section 7.5]()).

Messages are sent on a Connection using the Send action:

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

where messageData is the data object to send.

The optional messageContext parameter supports per-message properties
and is described in [Section 7.4](). If provided, the Message Context
object returned is identical to the one that was passed.

The optional endOfMessage parameter supports partial sending and is
described in [Section 7.5]().

The MessageContext returned by Send can be used to identify send
events (see [Section 7.3]()) related to a specific message or to inspect
meta-data related to the message sent.

[7.1](). Basic Sending

The most basic form of sending on a connection involves enqueuing a
single Data block as a complete Message, with default Message
Properties. Message data is transferred as an array of bytes, and

   the resulting object contains both the byte array and the length of
   the array.

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

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

   Some transport protocols can deliver arbitrarily sized Messages, but
   other protocols constrain the maximum Message size. Applications can
   query the Connection Property "Maximum Message size on send"
   (Section 11.1.8) to determine the maximum size allowed for a single
   Message. If a Message is too large to fit in the Maximum Message
   Size for the Connection, the Send will fail with a SendError event
   (Section 7.3.3). For example, it is invalid to send a Message over a
   UDP connection that is larger than the available datagram sending
   size.

7.2. Sending Replies

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

   replyMessageContext := requestMessageContext.reply()

   By using the "replyMessageContext", the transport system is informed
   that the message to be sent is a response and can map the response to
   the same underlying transport connection or stream the request was
   received from. The concept of Message Contexts is described in
   Section 9.

7.3. Send Events

   Like all Actions in this interface, the Send Action is asynchronous.
   There are several Events that can be delivered in response to Sending
   a Message.

   Note that if partial Sends are used (Section 7.5), there will still
   be exactly one Send Event delivered for each call to Send. For
   example, if a Message expired while two requests to Send data for
   that Message are outstanding, there will be two Expired events
   delivered.

[7.3.1](#). Sent

   Connection -> Sent<messageContext>

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

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

[7.3.2](#). Expired

   Connection -> Expired<messageContext>

   The Expired Event occurs when a previous Send Action expired before
   completion; i.e. when the Message was not sent before its Lifetime
   (see [Section 7.4.1](#)) expired. This is separate from SendError, as it
   is an expected behavior for partially reliable transports. The
   Expired Event contains an implementation-specific reference to the
   Message to which it applies.

[7.3.3](#). SendError

   Connection -> SendError<messageContext>

   A SendError occurs when a Message could not be sent due to an error
   condition: an attempt to send a Message which is too large for the
   system and Protocol Stack to handle, some failure of the underlying
   Protocol Stack, or a set of Message Properties not consistent with
   the Connection's transport properties. The SendError contains an
   implementation-specific reference to the Message to which it applies.

[7.4](#). Message Properties

   Applications may need to annotate the Messages they send with extra
   information to control how data is scheduled and processed by the
   transport protocols in the Connection. Therefore a message context

containing these properties can be passed to the Send Action. For
other uses of the message context, see Section 9.

Note that message properties are per-Message, not per-Send if partial
Messages are sent (Section 7.5). All data blocks associated with a
single Message share properties specified in the Message Contexts.
For example, it would not make sense to have the beginning of a
Message expire, but allow the end of a Message to still be sent.

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

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

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

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

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

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

The following Message Properties are supported:

7.4.1. Lifetime

Name: msg-lifetime

Type: Integer

Recommended default: infinite

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

Setting a Message's Lifetime to infinite indicates that the
application does not wish to apply a time constraint on the
transmission of the Message, but it does not express a need for
reliable delivery; reliability is adjustable per Message via the
"Reliable Data Transfer (Message)" property (see Section 7.4.7). The
type and units of Lifetime are implementation-specific.

7.4.2. Priority

   Name: msg-prio

   Type: Integer (non-negative)

   Recommended default: 100

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

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

   Note that this property is not a per-message override of the
   connection Priority - see Section 11.1.3. Both Priority properties
   may interact, but can be used independently and be realized by
   different mechanisms.

7.4.3. Ordered

   Name: msg-ordered

   Type: Boolean

   Default: true

   If true, it specifies that the receiver-side transport protocol stack
   only deliver the Message to the receiving application after the
   previous ordered Message which was passed to the same Connection via
   the Send Action, when such a Message exists. If false, the Message

may be delivered to the receiving application out of order. This
property is used for protocols that support preservation of data
ordering, see [Section 5.2.4](#), but allow out-of-order delivery for
certain messages.

[7.4.4](#). Idempotent

Name: idempotent

Type: Boolean

Default: false

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

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

[7.4.5](#). Final

Type: Boolean

Name: final

Default: false

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

Note that a Final Message must always be sorted to the end of a list
of Messages. The Final property overrides Priority and any other
property that would re-order Messages. If another Message is sent
after a Message marked as Final has already been sent on a

Connection, the Send Action for the new Message will cause a
SendError Event.

7.4.6. Corruption Protection Length

   Name: msg-checksum-len

   Type: Integer (non-negative with -1 as special value)

   Default: full coverage

   This property specifies the minimum length of the section of the
   Message, starting from byte 0, that the application requires to be
   delivered without corruption due to lower layer errors. It is used
   to specify options for simple integrity protection via checksums. A
   value of 0 means that no checksum is required, and -1 means that the
   entire Message is protected by a checksum. Only full coverage is
   guaranteed, any other requests are advisory.

7.4.7. Reliable Data Transfer (Message)

   Name: msg-reliable

   Type: Boolean

   Default: true

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

7.4.8. Message Capacity Profile Override

   Name: msg-capacity-profile

   Type: Enumeration

   This enumerated property specifies the application's preferred
   tradeoffs for sending this Message; it is a per-Message override of
   the Capacity Profile protocol and path selection property (see
   Section 11.1.10).

The following values are valid for Transmission Profile:

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

Low Latency: Response time (latency) should be optimized at the
   expense of efficiently using the available capacity when sending
   this message. This can be used by the system to disable the
   coalescing of multiple small Messages into larger packets (Nagle's
   algorithm); to prefer immediate acknowledgment from the peer
   endpoint when supported by the underlying transport; to signal a
   preference for lower-latency, higher-loss treatment; and so on.

   [TODO: This is inconsistent with {prop-cap-profile}} - needs to be
   fixed]

7.4.9. Singular Transmission

Name: singular-transmission

Type: Boolean

Default: false

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

7.5. Partial Sends

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

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

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

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

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

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

All data sent with the same MessageContext object will be treated as belonging to the same Message, and will constitute an in-order series until the endOfMessage is marked. Once the end of the Message is marked, the MessageContext object may be re-used as a new Message with identical parameters.

7.6. Batching Sends

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

```
Connection.Batch(
    Connection.Send(messageData)
    Connection.Send(messageData)
)
```

7.7. Send on Active Open: InitiateWithSend

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

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

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

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

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

8. Receiving Data

Once a Connection is established, it can be used for receiving data.
As with sending, data is received in terms of Messages. Receiving is
an asynchronous operation, in which each call to Receive enqueues a
request to receive new data from the connection. Once data has been
received, or an error is encountered, an event will be delivered to
complete the Receive request (see Section 8.2).

As with sending, the type of the Message to be passed is dependent on
the implementation, and on the constraints on the Protocol Stacks
implied by the Connection's transport parameters.

8.1. Enqueuing Receives

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

Connection.Receive(minIncompleteLength?, maxLength?)

By default, Receive will try to deliver complete Messages in a single
event (Section 8.2.1).

The application can set a minIncompleteLength value to indicates the
smallest partial Message data size in bytes that should be delivered
in response to this Receive. By default, this value is infinite,
which means that only complete Messages should be delivered (see
Section 8.2.2 and Section 10.6 for more information on how this is
accomplished). If this value is set to some smaller value, the
associated receive event will be triggered only when at least that
many bytes are available, or the Message is complete with fewer
bytes, or the system needs to free up memory. Applications should
always check the length of the data delivered to the receive event
and not assume it will be as long as minIncompleteLength in the case
of shorter complete Messages or memory issues.

The maxLength argument indicates the maximum size of a Message in
bytes the application is currently prepared to receive. The default
value for maxLength is infinite. If an incoming Message is larger
than the minimum of this size and the maximum Message size on receive
for the Connection's Protocol Stack, it will be delivered via
ReceivedPartial events (Section 8.2.2).

Note that maxLength does not guarantee that the application will
receive that many bytes if they are available; the interface may
return ReceivedPartial events with less data than maxLength according
to implementation constraints.

8.2. Receive Events

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

8.2.1. Received

Connection -> Received<messageData, messageContext>

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

The messageData object provides access to the bytes that were
received for this Message, along with the length of the byte array.
The messageContext is provided to enable retrieving metadata about
the message and referring to the message, e.g., to send replies and
map responses to their requests. See Section 9 for details.

See Section 10.6 for handling Message framing in situations where the
Protocol Stack only provides a byte-stream transport.

8.2.2. ReceivedPartial

Connection -> ReceivedPartial<messageData, messageContext, endOfMessage>

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

Multiple invocations of ReceivedPartial deliver data for the same
Message by passing the same MessageContext, until the endOfMessage
flag is delivered or a ReceiveError occurs. All partial blocks of a

single Message are delivered in order without gaps. This event does
not support delivering discontiguous partial Messages.

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

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

o the underlying Protocol Stack does not support message boundary
   preservation, and the Message Framer (see Section 10.6) cannot
   determine the end of the message using the buffer space it has
   available; or

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

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

8.2.3. ReceiveError

    Connection -> ReceiveError<messageContext>

A ReceiveError occurs when data is received by the underlying
Protocol Stack that cannot be fully retrieved or parsed, or when some
other indication is received that reception has failed. Such
conditions that irrevocably lead to the termination of the Connection
are signaled using ConnectionError instead (see Section 12).

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

8.3. Receive Message Properties

Each Message Context may contain metadata from protocols in the
Protocol Stack; which metadata is available is Protocol Stack
dependent. These are exposed though additional read-only Message
Properties that can be queried from the MessageContext object (see
Section 9) passed by the receive event. The following metadata
values are supported:

8.3.1. ECN

When available, Message metadata carries the value of the Explicit
Congestion Notification (ECN) field. This information can be used
for logging and debugging purposes, and for building applications
which need access to information about the transport internals for
their own operation.

8.3.2. Early Data

In some cases it may be valuable to know whether data was read as
part of early data transfer (before connection establishment has
finished). This is useful if applications need to treat early data
separately, e.g., if early data has different security properties
than data sent after connection establishment. In the case of TLS
1.3, client early data can be replayed maliciously (see [RFC8446]).
Thus, receivers may wish to perform additional checks for early data
to ensure it is idempotent or not replayed. If TLS 1.3 is available
and the recipient Message was sent as part of early data, the
corresponding metadata carries a flag indicating as such. If early
data is enabled, applications should check this metadata field for
Messages received during connection establishment and respond
accordingly.

8.3.3. Receiving Final Messages

The Message Context can indicate whether or not this Message is the
Final Message on a Connection. For any Message that is marked as
Final, the application can assume that there will be no more Messages
received on the Connection once the Message has been completely
delivered. This corresponds to the Final property that may be marked
on a sent Message Section 7.4.5.

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

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

9. Message Contexts

Using the MessageContext object, the application can set and retrieve
meta-data of the message, including Message Properties (see
Section 7.4) and framing meta-data (see Section 10.3). Therefore, a
MessageContext object can be passed to the Send action and is retuned
by each Send and Receive related events.

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

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

To get or set Message Properties, the optional scope parameter is
left empty, for framing meta-data, the framer is passed.

For MessageContexts returned by send events (see Section 7.3) and
receive events (see Section 8.2), the application can query
information about the local and remote endpoint:

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

Message Contexts can also be used to send messages that are flagged
as a reply to other messages, see Section 7.2 for details. If the
message received was send by the remote endpoint as a reply to an
earlier message and the transports provides this information, the
MessageContext of the original request can be accessed using the
Message Context of the reply:

RequestMessageContext := MessageContext.GetOriginalRequest()

10. Message Framers

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

Message Framers allow message boundaries to be preserved when using a
Connection object, even when using byte-stream transports. This
facility is designed based on the fact that many of the current
application protocols evolved over TCP, which does not provide
message boundary preservation, and since many of these protocols
require message boundaries to function, each application layer
protocol has defined its own framing.

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

Most Message Framers fall into one of two categories: - Header-
prefixed record formats, such as a basic Type-Length-Value (TLV)
structure - Delimeter-separated formats, such as HTTP/1.1.

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

10.1. Defining Message Framers

A Message Framer is primarily defined by the set of code that handles
events for a framer implementation, specifically how it handles
inbound and outbound data parsing.

Applications can instantiate a Message Framer upon which they will
receive framing events, or use a Message Framer defined by another
library.

    framer := NewMessageFramer()

Message Framer objects will deliver events to code that is written
either as part of the application or a helper library. This piece of
code will be referred to as the "framer implementation".

10.2. Adding Message Framers to Connections

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

    Preconnection.AddFramer(framer)

Framers have the ability to also dynamically modify Protocol Stacks,
as described in Section 10.4.

10.3. Framing Meta-Data

When sending Messages, applications can add specific Message values
to a MessageContext (Section 9) that is intended for a Framer. This
can be used, for example, to set the type of a Message for a TLV
format. The namespace of values is custom for each unique Message
Framer.

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

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

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

10.4. Message Framer Lifetime

When a Connection establishment attempt begins, an event is delivered to notify the framer implementation that a new Connection is being created. Similarly, a stop event is 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 Message Framer generates a "Start" event, the framer implementation has the opportunity to start writing some data prior to the Connection delivering its "Ready" event. This allows the implementation to communicate control data to the remote endpoint that can be used to parse Messages.

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

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)
```

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

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

10.5. Sender-side Message Framing

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

```
MessageFramer -> NewSentMessage<Connection, MessageData, MessageContext,
IsEndOfMessage>
```

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

MessageFramer.Send(Connection, Data)

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

10.6. Receiver-side Message Framing

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

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) ->
(Data, MessageContext, IsEndOfMessage)

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

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

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

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

directly inspect all of the bytes.

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

11. Managing Connections

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

Connection Properties represent the configuration and state of the
selected Protocol Stack(s) backing a Connection. These Connection
Properties may be Generic, applying regardless of transport protocol,
or Specific, applicable to a single implementation of a single
transport protocol stack. Generic Connection Properties are defined
in Section 11.1 below. Specific Protocol Properties are defined in a
transport- and implementation-specific way, and must not be assumed
to apply across different protocols. Attempts to set Specific
Protocol Properties on a protocol stack not containing that specific
protocol are simply ignored, and do not raise an error; however, too
much reliance by an application on Specific Protocol Properties may
significantly reduce the flexibility of a transport services
implementation.

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

Connection.SetProperty(property, value)

At any point, the application can query Connection Properties.

ConnectionProperties := Connection.GetProperties()

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

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

o Whether the connection can be used to send data. A connection can
   not be used for sending if the connection was created with the

Selection Property "Direction of Communication" set to
"unidirectional receive" or if a Message marked as "Final" was
sent over this connection, see Section 7.4.5.

o Whether the connection can be used to receive data. A connection
   can not be used for reading if the connection was created with the
   Selection Property "Direction of Communication" set to
   "unidirectional send" or if a Message marked as "Final" was
   received, see Section 8.3.3. The latter is only supported by
   certain transport protocols, e.g., by TCP as half-closed
   connection.

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

o For Connections that are Established, Closing, or Closed:
   Selection (Section 5.2) and Connection Properties (Section 11.1)
   of the actual protocols that were selected and instantiated.
   Selection Properties indicate whether or not the Connection has or
   offers a certain Selection Property. Note that the actually
   instantiated protocol stack may not match all Protocol Selection
   Properties that the application specified on the Preconnection.
   For example, a certain Protocol Selection Property that an
   application specified as Preferred may not actually be present in
   the chosen protocol stack because none of the currently available
   transport protocols had this feature.

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

11.1. Generic Connection Properties

The Connection Properties defined as independent, and available on
all Connections are defined in the subsections below.

Note that many protocol properties have a corresponding selection
property, which prefers protocols providing a specific transport
feature that controlled by that protocol property. [EDITOR'S NOTE:
todo: add these cross-references up to Section 5.2]

11.1.1. Retransmission Threshold Before Excessive Retransmission
        Notification

Name: retransmit-notify-threshold

Type: Integer

Default: -1

This property specifies after how many retransmissions to inform the
application about "Excessive Retransmissions". The special value -1
means that this notification is disabled.

11.1.2. Required Minimum Corruption Protection Coverage for Receiving

Name: recv-checksum-len

Type: Integer

Default: -1

This property specifies the part of the received data that needs to
be covered by a checksum. It is given in Bytes. A value of 0 means
that no checksum is required, and the special value -1 indicates full
checksum coverage.

11.1.3. Priority (Connection)

Name: conn-prio

Type: Integer

Default: 100

This Property is a non-negative integer representing the relative
inverse priority of this Connection relative to other Connections in
the same Connection Group. It has no effect on Connections not part
of a Connection Group. As noted in Section 6.4, this property is not
entangled when Connections are cloned.

11.1.4. Timeout for Aborting Connection

Name: conn-timeout

Type: Numeric

Default: -1

This property specifies how long to wait before deciding that a
Connection has failed when trying to reliably deliver data to the
destination. Adjusting this Property will only take effect when the
underlying stack supports reliability. The special value -1 means
that this timeout is not scheduled to happen.

11.1.5. Connection Group Transmission Scheduler

   Name: conn-scheduler

   Type: Enumeration

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

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

11.1.6. Maximum Message Size Concurrent with Connection Establishment

   Name: zero-rtt-msg-max-len

   Type: Integer (read only)

   This property represents the maximum Message size that can be sent
   before or during Connection establishment, see also Section 7.4.4.
   It is given in Bytes.

11.1.7. Maximum Message Size Before Fragmentation or Segmentation

   Name: singular-transmission-msg-max-len

   Type: Integer (read only)

   This property, if applicable, represents the maximum Message size
   that can be sent without incurring network-layer fragmentation or
   transport layer segmentation at the sender.

11.1.8. Maximum Message Size on Send

   Name: send-msg-max-len

   Type: Integer (read only)

   This property represents the maximum Message size that can be sent.

11.1.9. Maximum Message Size on Receive

   Name: recv-msg-max-len

   Type: Integer (read only)

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

11.1.10. Capacity Profile

   Name: conn-capacity-profile

   This property specifies the desired network treatment for traffic
   sent by the application and the tradeoffs the application is prepared
   to make in path and protocol selection to receive that desired
   treatment. When the capacity profile is set to a value other than
   Default, the transport system should select paths and profiles to
   optimize for the capacity profile specified. The following values
   are valid for the Capacity Profile:

   Default: The application makes no representation about its expected
      capacity profile. No special optimizations of the tradeoff
      between delay, delay variation, and bandwidth efficiency should be
      made when selecting and configuring transport protocol stacks.
      Transport system implementations that map the requested capacity
      profile onto per-connection DSCP signaling without multiplexing
      SHOULD assign the DSCP Default Forwarding [RFC2474] PHB; when the
      Connection is multiplexed, the guidelines in Section 6 of
      [RFC7657] apply.

   Scavenger: The application is not interactive. It expects to send
      and/or receive data without any urgency. This can, for example,
      be used to select protocol stacks with scavenger transmission
      control and/or to assign the traffic to a lower-effort service.
      Transport system implementations that map the requested capacity
      profile onto per-connection DSCP signaling without multiplexing
      SHOULD assign the DSCP Less than Best Effort [LE-PHB] PHB; when
      the Connection is multiplexed, the guidelines in Section 6 of
      [RFC7657] apply.

   Low Latency/Interactive: The application is interactive, and prefers
      loss to latency. Response time should be optimized at the expense
      of bandwidth efficiency and delay variation when sending on this
      connection. This can be used by the system to disable the
      coalescing of multiple small Messages into larger packets (Nagle's
      algorithm); to prefer immediate acknowledgment from the peer
      endpoint when supported by the underlying transport; and so on.
      Transport system implementations that map the requested capacity
      profile onto per-connection DSCP signaling without multiplexing
      SHOULD assign the DSCP Expedited Forwarding [RFC3246] PHB; when
      the Connection is multiplexed, the guidelines in Section 6 of
      [RFC7657] apply.

   Low Latency/Non-Interactive: The application prefers loss to latency
      but is not interactive. Response time should be optimized at the
      expense of bandwidth efficiency and delay variation when sending

on this connection.Transport system implementations that map the
requested capacity profile onto per-connection DSCP signaling
without multiplexing SHOULD assign a DSCP Assured Forwarding
(AF21,AF22,AF23,AF24) [RFC2597] PHB; when the Connection is
multiplexed, the guidelines in Section 6 of [RFC7657] apply.

Constant-Rate Streaming: The application expects to send/receive
data at a constant rate after Connection establishment. Delay and
delay variation should be minimized at the expense of bandwidth
efficiency. This implies that the Connection may fail if the
desired rate cannot be maintained across the Path. A transport
may interpret this capacity profile as preferring a circuit
breaker [RFC8084] to a rate-adaptive congestion controller.
Transport system implementations that map the requested capacity
profile onto per-connection DSCP signaling without multiplexing
SHOULD assign a DSCP Assured Forwarding (AF31,AF32,AF33,AF34)
[RFC2597] PHB; when the Connection is multiplexed, the guidelines
in Section 6 of [RFC7657] apply.

High Throughput Data: The application expects to send/receive data
at the maximum rate allowed by its congestion controller over a
relatively long period of time. Transport system implementations
that map the requested capacity profile onto per-connection DSCP
signaling without multiplexing SHOULD assign a DSCP Assured
Forwarding (AF11,AF12,AF13,AF14) [RFC2597] PHB per Section 4.8 of
[RFC4594]. When the Connection is multiplexed, the guidelines in
Section 6 of [RFC7657] apply.

The Capacity Profile for a selected protocol stack may be modified on
a per-Message basis using the Transmission Profile Message Property;
see Section 7.4.8.

11.1.11. Bounds on Send or Receive Rate

Name: max-send-rate / max-recv-rate

Type: Numeric / Numeric

Default: -1 / -1 (unlimited, for both values)

This property specifies an upper-bound rate that a transfer is not
expected to exceed (even if flow control and congestion control allow
higher rates), and/or a lower-bound rate below which the application
does not deem a data transfer useful. It is given in bits per
second. The special value -1 indicates that no bound is specified.

11.1.12. **TCP-specific Property: User Timeout**

   This property specifies, for the case TCP becomes the chosen
   transport protocol:

   Advertised User Timeout (name: tcp.user-timeout-value, type:
   Integer):
      a time value (default: the TCP default) to be advertised via the
      User Timeout Option (UTO) for the TCP at the remote endpoint to
      adapt its own "Timeout for aborting Connection" (see
      Section 11.1.4) value accordingly.

   User Timeout Enabled (name: tcp.user-timeout, type: Boolean): a bool
      ean (default false) to control whether the UTO option is enabled
      for a connection. This applies to both sending and receiving.

   Changeable (name: tcp.user-timeout-recv, type: Boolean): a boolean
      (default true) which controls whether the "Timeout for aborting
      Connection" (see Section 11.1.4) may be changed based on a UTO
      option received from the remote peer. This boolean becomes false
      when "Timeout for aborting Connection" (see Section 11.1.4) is
      used.

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

11.2. **Soft Errors**

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

   Connection -> SoftError<>

11.3. **Excessive retransmissions**

   This event notifies the application of excessive retransmissions,
   based on a configured threshold (see Section 11.1.1). This will only
   happen if the underlying protocol stack supports reliability and,
   with it, such notifications.

   Connection -> ExcessiveRetransmission<>

12. Connection Termination

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

   Connection.Close()

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

   Connection -> Closed<>

   Abort terminates a Connection without delivering remaining data:

   Connection.Abort()

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

   Connection -> ConnectionError<>

13. Connection State and Ordering of Operations and Events

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

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

   o Ready<> occurs when a Connection created with Initiate() or
      InitiateWithSend() transitions to Established state.

   o ConnectionReceived<> occurs when a Connection created with
      Listen() transitions to Established state.

   o RendezvousDone<> occurs when a Connection created with
      Rendezvous() transitions to Established state.

o Closed<> occurs when a Connection transitions to Closed state
   without error.

o InitiateError<> occurs when a Connection created with Initiate()
   transitions from Establishing state to Closed state due to an
   error.

o ConnectionError<> occurs when a Connection transitions to Closed
   state due to an error in all other circumstances.

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

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

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

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

14. IANA Considerations

RFC-EDITOR: Please remove this section before publication.

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

15. Security Considerations

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

16. Acknowledgements

17. References

17.1. Normative References

[I-D.ietf-taps-arch]
           Pauly, T., Trammell, B., Brunstrom, A., Fairhurst, G.,
           Perkins, C., Tiesel, P., and C. Wood, "An Architecture for
           Transport Services", draft-ietf-taps-arch-03 (work in
           progress), March 2019.

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

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

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

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

17.2. Informative References

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

   [I-D.ietf-taps-transport-security]
              Wood, C., Enghardt, T., Pauly, T., Perkins, C., and K.
              Rose, "A Survey of Transport Security Protocols", draft-
              ietf-taps-transport-security-06 (work in progress), March
              2019.

   [LE-PHB]   Bless, R., "A Lower Effort Per-Hop Behavior (LE PHB) for
              Differentiated Services", draft-ietf-tsvwg-le-phb-10 (work
              in progress), March 2019.

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

   [RFC2474] Nichols, K., Blake, S., Baker, F., and D. Black,
              "Definition of the Differentiated Services Field (DS
              Field) in the IPv4 and IPv6 Headers", RFC 2474,
              DOI 10.17487/RFC2474, December 1998,
              <https://www.rfc-editor.org/info/rfc2474>.

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

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

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

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

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

   [RFC5245] Rosenberg, J., "Interactive Connectivity Establishment
              (ICE): A Protocol for Network Address Translator (NAT)
              Traversal for Offer/Answer Protocols", RFC 5245,
              DOI 10.17487/RFC5245, April 2010,
              <https://www.rfc-editor.org/info/rfc5245>.

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

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

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

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

   [RFC8095] Fairhurst, G., Ed., Trammell, B., Ed., and M. Kuehlewind,
              Ed., "Services Provided by IETF Transport Protocols and
              Congestion Control Mechanisms", RFC 8095,
              DOI 10.17487/RFC8095, March 2017,
              <https://www.rfc-editor.org/info/rfc8095>.

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

Appendix A. Additional Properties

   The interface specified by this document represents the minimal
   common interface to an endpoint in the transport services
   architecture [I-D.ietf-taps-arch], based upon that architecture and
   on the minimal set of transport service features elaborated in
   [I-D.ietf-taps-minset]. However, the interface has been designed

with extension points to allow the implementation of features beyond
those in the minimal common interface: Protocol Selection Properties,
Path Selection Properties, and Message Properties are open sets.
Implementations of the interface are free to extend these sets to
provide additional expressiveness to applications written on top of
them.

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

A.1. Experimental Transport Properties

The following Transport Properties might be made available in
addition to those specified in Section 5.2, Section 11.1, and
Section 7.4.

A.1.1. Cost Preferences

[EDITOR'S NOTE: At IETF 103, opinions were that this property should
stay, but it was also said that this is maybe not "on the right
level". If / when moving it to the main text, note that this is
meant to be applicable to a Preconnection or a Message.]

Name: cost-preferences

Type: Enumeration

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

Possible values are:

No Expense: Avoid transports associated with monetary cost

Optimize Cost: Prefer inexpensive transports and accept service
   degradation

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

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

The default is "Balance Cost".

Appendix B. Sample API definition in Go

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

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

[I-D.ietf-taps-minset] identifies a minimal set of transport services
that end systems should offer. These services make all transport
features offered by TCP, MPTCP, UDP, UDP-Lite, SCTP and LEDBAT
available that 1) require interaction with the application, and 2) do
not get in the way of a possible implementation over TCP or, with
limitations, UDP. The following text explains how this minimal set
is reflected in the present API. For brevity, this uses the list in
Section 4.1 of [I-D.ietf-taps-minset], updated according to the
discussion in Section 5 of [I-D.ietf-taps-minset].

[EDITOR'S NOTE: This is early text. In the future, this section will
contain backward references, which we currently avoid because things
are still being moved around and names / categories etc. are
changing.]

o Connect:
   "Initiate" Action.

o Listen:
   "Listen" Action.

o Specify number of attempts and/or timeout for the first
   establishment message:
   "Timeout for aborting Connection Establishment" Property, using a
   time value.

o Disable MPTCP:
   "Parallel Use of Multiple Paths" Property.

o Hand over a message to reliably transfer (possibly multiple times)
   before connection establishment:

"InitiateWithSend" Action.

o Hand over a message to reliably transfer during connection
   establishment:
   "InitiateWithSend" Action.

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

o Timeout event when data could not be delivered for too long:
   "ConnectionError" Event.

o Suggest timeout to the peer:
   TCP-specific Property: User Timeout.

o Notification of Excessive Retransmissions (early warning below
   abortion threshold):
   "Notification of excessive retransmissions" property.

o Notification of ICMP error message arrival:
   "Notification of ICMP soft error message arrival" property.

o Choose a scheduler to operate between streams of an association:
   "Connection group transmission scheduler" property.

o Configure priority or weight for a scheduler:
   "Priority (Connection)" property.

o "Specify checksum coverage used by the sender" and "Disable
   checksum when sending":
   "Corruption Protection Length" property (value 0 to disable).

o "Specify minimum checksum coverage required by receiver" and
   "Disable checksum requirement when receiving":
   "Required minimum coverage of the checksum for receiving" property
   (value 0 to disable).

o "Specify DF" field and "Request not to bundle messages:"
   The "Singular Transmission" Message property combines both of
   these requests, i.e. if a request not to bundle messages is made,
   this also turns off DF in case of protocols that allow this (only
   UDP and UDP-Lite, which cannot bundle messages anyway).

o Get max. transport-message size that may be sent using a non-
   fragmented IP packet from the configured interface:
   "Maximum Message size before fragmentation or segmentation"
   property.

o Get max. transport-message size that may be received from the
   configured interface:
   "Maximum Message size on receive" property.

o Obtain ECN field:
   "ECN" is a defined metadata value as part of the Message Receive
   Context.

o "Specify DSCP field", "Disable Nagle algorithm", "Enable and
   configure a 'Low Extra Delay Background Transfer'":
   As suggested in Section 5.5 of [I-D.ietf-taps-minset], these
   transport features are collectively offered via the "Capacity
   profile" property.

o Close after reliably delivering all remaining data, causing an
   event informing the application on the other side:
   This is offered by the "Close" Action with slightly changed
   semantics in line with the discussion in Section 5.2 of
   [I-D.ietf-taps-minset].

o "Abort without delivering remaining data, causing an event
   informing the application on the other side" and "Abort without
   delivering remaining data, not causing an event informing the
   application on the other side":
   This is offered by the "Abort" action without promising that this
   is signaled to the other side. If it is, a "ConnectionError"
   Event will fire at the peer.

o "Reliably transfer data, with congestion control", "Reliably
   transfer a message, with congestion control" and "Unreliably
   transfer a message":
   Reliability is controlled via the "Reliable Data Transfer
   (Message)" Message property. Transmitting data without delimiters
   is done by not using a Framer. The choice of congestion control
   is provided via the "Congestion control" property.

o Configurable Message Reliability:
   The "Lifetime" Message Property implements a time-based way to
   configure message reliability.

o "Ordered message delivery (potentially slower than unordered)" and
   "Unordered message delivery (potentially faster than ordered)":
   The two transport features are controlled via the Message property
   "Ordered".

o Request not to delay the acknowledgement (SACK) of a message:

Should the protocol support it, this is one of the transport
features the transport system can use when an application uses the
Capacity Profile Property with value "Low Latency/Interactive".

o Receive data (with no message delimiting):
   "Received" Event without using a Message Framer.

o Receive a message:
   "Received" Event. Section 5.1 of [I-D.ietf-taps-minset] discusses
   how messages can be obtained from a bytestream in case of
   implementation over TCP. Here, this is dealt with by Message
   Framers.

o Information about partial message arrival:
   "ReceivedPartial" Event.

o Notification of send failures:
   "Expired" and "SendError" Events.

o Notification that the stack has no more user data to send:
   Applications can obtain this information via the "Sent" Event.

o Notification to a receiver that a partial message delivery has
   been aborted:
   "ReceiveError" Event.

Authors' Addresses

Brian Trammell (editor)
Google
Gustav-Gull-Platz 1
8004 Zurich
Switzerland

Email: ietf@trammell.ch

Michael Welzl (editor)
University of Oslo
PO Box 1080 Blindern
0316 Oslo
Norway

Email: michawe@ifi.uio.no

Theresa Enghardt
TU Berlin
Marchstrasse 23
10587 Berlin
Germany

Email: theresa@inet.tu-berlin.de

Godred Fairhurst
University of Aberdeen
Fraser Noble Building
Aberdeen, AB24 3UE
Scotland

Email: gorry@erg.abdn.ac.uk
URI: http://www.erg.abdn.ac.uk/

Mirja Kuehlewind
ETH Zurich
Gloriastrasse 35
8092 Zurich
Switzerland

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

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

Email: csp@csperkins.org

Philipp S. Tiesel
TU Berlin
Einsteinufer 25
10587 Berlin
Germany

Email: philipp@tiesel.net

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

Email: cawood@apple.com

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

Email: tpauly@apple.com