TAPS Working Group   B. Trammell, Ed.
Internet-Draft   Google Switzerland GmbH
Intended status: Standards Track   M. Welzl, Ed.
Expires: January 28, 2021   University of Oslo
                                                         T. Enghardt
                                                             Netflix
                                                        G. Fairhurst
                                               University of Aberdeen
                                                       M. Kuehlewind
                                                            Ericsson
                                                          C. Perkins
                                               University of Glasgow
                                                           P. Tiesel
                                                           TU Berlin
                                                            C. Wood
                                                          Cloudflare
                                                           T. Pauly
                                                          Apple Inc.
                                                      July 27, 2020

        An Abstract Application Layer Interface to Transport Services
                      draft-ietf-taps-interface-08

Abstract

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

Status of This Memo

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 28, 2021.

Copyright Notice

Table of Contents

1. Introduction

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

   interface to the transport layer, in environments where endpoints
   select from multiple interfaces and potential transport protocols.

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

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

2. Terminology and Notation

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

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

   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 and Event parameters whose names are suffixed
      with a question mark are optional.

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

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

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

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

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

3. Overview of Interface Design

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

   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-orientation, 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 configuration of cryptographic identities and
   transport security parameters persistent across multiple
   Connections; 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 API is the basic common abstract application
programming interface to the Transport Services Architecture defined
in the TAPS Architecture [I-D.ietf-taps-arch].

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

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

event handlers registered by the application. Errors and other
notifications also happen asynchronously on the Connection. It is
not necessary for an application to handle all events; some events
may have implementation-specific default handlers. The application
should not assume that ignoring events (e.g. errors) is always safe.

Section 5, Section 6, Section 7, Section 8, and Section 11 describe
the details of application interaction with Objects through Actions
and Events in each phase of a Connection, following the phases (Pre-
Establishment, Establishment, Data Transfer, and Termination)
described in Section 4.1 of [I-D.ietf-taps-arch].

4.1. Usage Examples

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

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 that provides Reliable Data Transfer,
Preservation of data ordering, and Preservation of Message
Boundaries. In this case, the application can choose to receive only
complete messages.

If none of the available transport protocols provides Preservation of
Message Boundaries, but there is a transport protocol that provides a
reliable ordered byte stream, an application may receive this byte
stream as partial Messages and transform it into application-layer
Messages. Alternatively, an application may provide a Message
Framer, which can transform a byte stream into a sequence of Messages
(Section 9).

4.1.1. Server Example

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

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

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

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

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

Listener := Preconnection.Listen()

Listener -> ConnectionReceived<Connection>

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

Connection -> Received<messageDataRequest, messageContext>

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

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

4.1.2. Client Example

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

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

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

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

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

   Connection := Preconnection.Initiate()

   Connection -> Ready<>

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

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

   Connection -> Received<messageDataResponse, messageContext>

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

4.1.3. Peer Example

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

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

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

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

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

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

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

   Preconnection.Rendezvous()

   Preconnection -> RendezvousDone<Connection>

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

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

   Connection -> Received<messageDataResponse, messageContext>

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

4.2. Transport Properties

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

Transport Properties are divided into Selection, Connection, and
Message Properties. Selection Properties (see Section 5.2) can only
be set during pre-establishment. They are only used to specify which
paths and protocol stacks can be used and are preferred by the
application. Connection Properties (see Section 10.1) can also be
set during pre-establishment but may be changed later. They are used
to inform decisions made during establishment and to fine-tune the
established connection.
The behavior of the selected protocol stack(s) when sending Messages
is controlled by Message Properties (see Section 7.5).

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

o Connection Properties can be set on Preconnections

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

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

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

4.2.1. Transport Property Names

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

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

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

implementations will still benefit from the consistency resulting
from the use of visually similar symbols.

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

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

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 string
   identifying the vendor or implementation as Namespace.

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

4.2.2. Transport Property Types

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

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. When querying, a

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

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

4.3. Scope of the Interface Definition

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

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

We therefore make the following recommendations:

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

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

o Implementations may use other representations for Transport
   Property Names, e.g., by providing constants, but should provide a

straight-forward mapping between their representation and the
property names specified here.

5. Pre-Establishment Phase

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

A Preconnection Object represents a potential Connection. It has
state that describes properties of a Connection that might exist in
the future. This state comprises Local Endpoint and Remote Endpoint
Objects that denote the endpoints of the potential Connection (see
Section 5.1), the Selection Properties (see Section 5.2), any
preconfigured Connection Properties (Section 10.1), and the security
parameters (see Section 5.3):

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

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

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

If Message Framers are used (see Section 9), they MUST be added to
the Preconnection during pre-establishment.

5.1. Specifying Endpoints

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

Specify a Remote Endpoint using a hostname and service name:

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

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

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

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

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

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

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

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

Specify a Local Endpoint using a STUN server:

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

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

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

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

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

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

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

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

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

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

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

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

5.2. Specifying Transport Properties

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

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

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

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

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

   The implementation MUST ensure an outcome that is consistent with
   application requirements as expressed using Require and Prohibit.
   While preferences expressed using Prefer and Avoid influence protocol
   and path selection as well, outcomes may vary given the same
   Selection Properties, as the available protocols and paths may vary

across systems and contexts. However, implementations are
RECOMMENDED to aim to provide a consistent outcome to an application,
given the same Selection Properties.

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

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

TransportProperties := NewTransportProperties()

Individual properties are then added to the TransportProperties
Object:

TransportProperties.Add(property, value)

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

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

TransportProperties := Connection.GetTransportProperties()

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

Section 10.1 provides a list of Connection Properties, while
Selection Properties are listed in the subsections below. Note that
many properties are only considered during establishment, and can not
be changed after a Connection is established; however, they can be

queried. The return type of a queried Selection Property is Boolean,
where "true" means that the Selection Property has been applied and
"false" means that the Selection Property has not been applied. Note
that "true" does not mean that a request has been honored. For
example, if "Congestion control" was requested with preference level
"Prefer", but congestion control could not be supported, querying the
"congestionControl" property yields the value "false". If preference
level "Avoid" was used for "Congestion control", and, as requested,
the Connection is not congestion controlled, querying the
"congestionControl" property also yields the value "false".

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

5.2.1. Reliable Data Transfer (Connection)

Name: reliability

Type: Preference

Default: Require

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

5.2.2. Preservation of Message Boundaries

Name: preserveMsgBoundaries

Type: Preference

Default: Prefer

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

5.2.3. Configure Per-Message Reliability

Name: perMsgReliability

Type: Preference

Default: Ignore

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

5.2.4. Preservation of Data Ordering

Name: preserveOrder

Type: Preference

Default: Require

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

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

Name: zeroRttMsg

Type: Preference

Default: Ignore

This property specifies whether an application would like to supply a
Message to the transport protocol before Connection establishment,
which will then be reliably transferred to the other side before or
during Connection establishment, potentially multiple times (i.e.,
multiple copies of the message data may be passed to the Remote
Endpoint). See also Section 7.5.4. Note that disabling this
property has no effect for protocols that are not connection-oriented
and do not protect against duplicated messages, e.g., UDP.

5.2.6. Multistream Connections in Group

   Name: multistreaming

   Type: Preference

   Default: Prefer

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

5.2.7. Full Checksum Coverage on Sending

   Name: perMsgChecksumLenSend

   Type: Preference

   Default: Require

   This property specifies whether the application desires protection
   against corruption for all data transmitted on this Connection.
   Disabling this property may enable to control checksum coverage later
   (see Section 7.5.6).

5.2.8. Full Checksum Coverage on Receiving

   Name: perMsgChecksumLenRecv

   Type: Preference

   Default: Require

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

5.2.9. Congestion control

   Name: congestionControl

   Type: Preference

   Default: Require

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

with [RFC2914]. Also note that reliability is usually combined with
congestion control in protocol implementations, rendering "reliable
but not congestion controlled" a request that is unlikely to succeed.

5.2.10. Interface Instance or Type

Name: interface

Type: Set (Preference, Enumeration)

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

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

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

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

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

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

5.2.11. Provisioning Domain Instance or Type

   Name: pvd

   Type: Set (Preference, Enumeration)

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

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

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

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

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

5.2.12. Use Temporary Local Address

   Name: useTemporaryLocalAddress

   Type: Preference

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

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

5.2.13. Multi-Paths Transport

Name: multipath

Type: Enumeration

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

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

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

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

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

The policy for using multiple paths is specified using the separate
"multipath-policy" property, see Section 10.1.7 below. To enable the
peer endpoint to initiate additional paths towards a local address
other than the one initially used, it is necessary to set the
Alternative Addresses property (see Section 5.2.14 below).

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

Enumeration values other than "Disabled" are interpreted as a
preference for choosing protocols that can make use of multiple

paths. The "Disabled" value implies a requirement not to use
multiple paths in parallel but does not prevent choosing a protocol
that is capable of using multiple paths, e.g., it does not prevent
choosing TCP, but prevents sending the "MP_CAPABLE" option in the TCP
handshake.

5.2.14. Advertisement of Alternative Addresses

   Name: advertises-altaddr

   Type: Boolean

   Default: False

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

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

5.2.15. Direction of communication

   Name: direction

   Type: Enumeration

   Default: Bidirectional

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

   Bidirectional: The connection must support sending and receiving
      data

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

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

   Since unidirectional communication can be supported by transports
   offering bidirectional communication, specifying unidirectional

communication may cause a transport stack that supports bidirectional
communication to be selected.

5.2.16. Notification of excessive retransmissions

Name: retransmitNotify

Type: Preference

Default: Ignore

This property specifies whether an application considers it useful to
be informed in case sent data was retransmitted more often than a
certain threshold (see Section 10.1.1 for configuration of this
threshold).

5.2.17. Notification of ICMP soft error message arrival

Name: softErrorNotify

Type: Preference

Default: Ignore

This property specifies whether an application considers it useful to
be informed when an ICMP error message arrives that does not force
termination of a connection. When set to true, received ICMP errors
will be available as SoftErrors, see Section 10.3.1. Note that even
if a protocol supporting this property is selected, not all ICMP
errors will necessarily be delivered, so applications cannot rely on
receiving them.

5.2.18. Initiating side is not the first to write

Name: activeReadBeforeSend

Type: Preference

Default: Ignore

The most common client-server communication pattern involves the
client actively opening a connection, then sending data to the
server. The server listens (passive open), reads, and then answers.
This property specifies whether an application wants to diverge from
this pattern - either by actively opening with Initiate(),
immediately followed by reading, or passively opening with Listen(),
immediately followed by writing. This property is ignored when
establishing connections using Rendezvous(). Requiring this property

limits the choice of mappings to underlying protocols, which can
reduce efficiency. For example, it prevents the transport system
from mapping Connections to SCTP streams, where the first transmitted
data takes the role of an active open signal [I-D.ietf-taps-impl].

5.3. Specifying Security Parameters and Callbacks

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

5.3.1. Pre-Connection Parameters

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

SecurityParameters := NewSecurityParameters()

Security configuration parameters and sample usage follow:

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

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

SecurityParameters.Add('supported-group', 'secp256k1')
SecurityParameters.Add('ciphersuite,
'TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256')
SecurityParameters.Add('signature-algorithm', 'ed25519')

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

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

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

5.3.2. Connection Establishment Callbacks

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

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

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

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

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

6. Establishing Connections

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

6.1. Active Open: Initiate

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

   Connection := Preconnection.Initiate(timeout?)

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

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

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

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

   Connection -> Ready<>

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

   Connection -> EstablishmentError<reason?>

   An EstablishmentError occurs either when the set of transport
   properties and security parameters cannot be fulfilled on a
   Connection for initiation (e.g. the set of available Paths and/or
   Protocol Stacks meeting the constraints is empty) or reconciled with
   the local and/or remote Endpoints; when the remote specifier cannot
   be resolved; or when no transport-layer connection can be established

to the remote Endpoint (e.g. because the remote Endpoint is not
accepting connections, the application is prohibited from opening a
Connection by the operating system, or the establishment attempt has
timed out for any other reason).

See also Section 7.8 to combine Connection establishment and
transmission of the first message in a single action.

6.2. Passive Open: Listen

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

Listener := Preconnection.Listen()

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

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

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

Listener.Stop()

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

Listener -> ConnectionReceived<Connection>

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

Listener.SetNewConnectionLimit(value)

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

Listener -> EstablishmentError<reason?>

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

Listener -> Stopped<>

A Stopped Event occurs after the Listener has stopped listening.

6.3. Peer-to-Peer Establishment: Rendezvous

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

Preconnection.Rendezvous()

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

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

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

Preconnection -> RendezvousDone<Connection>

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

Preconnection -> EstablishmentError<reason?>

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

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

[]Preconnection := Preconnection.Resolve()

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

6.4. Connection Groups

Entangled Connections can be created using the Clone Action:

Connection := Connection.Clone()

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

Properties. However, there may be exceptions, such as "Priority
(Connection)", see Section 10.1.3. Like all other Properties,
Priority is copied to the new Connection when calling Clone(), but it
is not entangled: Changing Priority on one Connection does not change
it on the other Connections in the same Connection Group.

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

[]Connection := Connection.GroupedConnections()

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

Changing one of the Connection Properties on one Connection in the
group changes it for all others. Message Properties, however, are
not entangled. For example, changing "Timeout for aborting
Connection" (see Section 10.1.4) on one Connection in a group will
automatically change this Connection Property for all Connections in
the group in the same way. However, changing "Lifetime" (see
Section 7.5.1) of a Message will only affect a single Message on a
single Connection, entangled or not.

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

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

Attempts to clone a Connection can result in a CloneError:

Connection -> CloneError<reason?>

The Connection Property "Priority" operates on entangled Connections
as in Section 7.5.2: when allocating available network capacity among
Connections in a Connection Group, sends on Connections with higher

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

7. 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 7.6).

Messages are sent on a Connection using the Send action:

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

where messageData is the data object to send.

The optional messageContext parameter allows adding Message
Properties as described in Section 7.5. Moreover, the messageContext
can be used to identify Send Events related to a specific Message
(see Section 7.3) or to inspect meta-data related to the Message sent
(see Section 7.4).

The optional endOfMessage parameter supports partial sending and is
described in Section 7.6.

Framers can be used to extend or modify the message data with
additional information that can be processed at the receiver to
detect message boundaries. This is further decribed in Section 9.

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.

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

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

Some transport protocols can deliver arbitrarily sized Messages, but
other protocols constrain the maximum Message size. Applications can
query the Connection Property "Maximum Message size on send"
(Section 10.1.9.3) to determine the maximum size allowed for a single
Message. If a Message is too large to fit in the Maximum Message
Size for the Connection, the Send will fail with a SendError event
(Section 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 7.4.

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. Exactly one Event (Sent, Expired, or SendError) will be
delivered in response to each call to Send.

Note that if partial Sends are used (Section 7.6), there will still
be exactly one Send Event delivered for each call to Send. For
example, if a Message expired while two requests to Send data for
that Message are outstanding, there will be two Expired events
delivered.

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

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 a reference to the Message to which it applies.

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

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

7.3.3. SendError

    Connection -> SendError<messageContext, reason?>

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

7.4. Message Contexts

    Using the MessageContext object, the application can set and retrieve
    meta-data of the message, including Message Properties (see
    Section 7.5) and framing meta-data (see Section 9.2). Therefore, a

MessageContext object can be passed to the Send action and is
returned by each Send and Receive related event.

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

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

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

For MessageContexts returned by send events (see Section 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 in reply to other
messages, see Section 7.2 for details.

7.5. 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 7.4.

Note that Message Properties are per-Message, not per-Send if partial
Messages are sent (Section 7.6). All data blocks associated with a
single Message share properties specified in the Message Contexts.
For example, it would not make sense to have the beginning of a
Message expire, but allow the end of a Message to still be sent.

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

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

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

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

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

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

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

The following Message Properties are supported:

7.5.1. Lifetime

Name: msgLifetime

Type: Numeric

Default: infinite

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

Setting a Message's Lifetime to infinite indicates that the
application does not wish to apply a time constraint on the
transmission of the Message, but it does not express a need for

reliable delivery; reliability is adjustable per Message via the
"Reliable Data Transfer (Message)" property (see Section 7.5.7).   The
type and units of Lifetime are implementation-specific.

7.5.2. Priority

   Name: msgPrio

   Type: Integer (non-negative)

   Default: 100

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

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

   Note that this property is not a per-message override of the
   connection Priority - see Section 10.1.3. Both Priority properties
   may interact, but can be used independently and be realized by
   different mechanisms.

7.5.3. Ordered

   Name: msgOrdered

   Type: Boolean

   Default: true

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

[7.5.4](#). **Safely Replayable**

   Name: safelyReplayable

   Type: Boolean

   Default: false

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

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

[7.5.5](#). **Final**

   Name: final

   Type: Boolean

   Default: false

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

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

7.5.6. Corruption Protection Length

   Name: msgChecksumLen

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

   Default: Full Coverage

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

7.5.7. Reliable Data Transfer (Message)

   Name: msgReliable

   Type: Boolean

   Default: true

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

7.5.8. Message Capacity Profile Override

   Name: msgCapacityProfile

   Type: Enumeration

   This enumerated property specifies the application's preferred
   tradeoffs for sending this Message; it is a per-Message override of
   the Capacity Profile connection property (see Section 10.1.6).

7.5.9. No Fragmentation

   Name: noFragmentation

   Type: Boolean

   Default: false

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

7.6. Partial Sends

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

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

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

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

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

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

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

7.7. Batching Sends

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

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

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

7.8. 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 "Safely Replayable". This
allows the transport system to make use of 0-RTT establishment in
case this is supported by the available protocol stacks. When the
selected stack(s) do not support transmitting data upon connection
establishment, InitiateWithSend is identical to Initiate() followed
by Send().

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

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

8. Receiving Data

Once a Connection is established, it can be used for receiving data
(unless the "Direction of Communication" property is set to
"unidirectional send"). As with sending, data is received in terms

of Messages. Receiving is an asynchronous operation, in which each
call to Receive enqueues a request to receive new data from the
connection. Once data has been received, or an error is encountered,
an event will be delivered to complete any pending Receive requests
(see Section 8.2). If Messages arrive at the transport system before
Receive requests are issued, ensuing Receive requests will first
operate on these Messages before awaiting any further Messages.

8.1. 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 indicate the
smallest partial Message data size in bytes that should be delivered
in response to this Receive. By default, this value is infinite,
which means that only complete Messages should be delivered (see
Section 8.2.2 and Section 9 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. Note also that maxLength and
minIncompleteLength are intended only to manage buffering, and are
not interpreted as a receiver preference for message reordering.

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.

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

8.2.1. 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 7.4 for details.

See Section 9 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 9) 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, reason?>

A ReceiveError occurs when data is received by the underlying
Protocol Stack that cannot be fully retrieved or parsed, or when some
other indication is received that reception has failed. In contrast,
conditions that irrevocably lead to the termination of the Connection
are signaled using ConnectionError instead (see Section 11).

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 7.4) passed by the receive event. The following metadata
values are supported:

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

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

because these protocols do not implement congestion control, and
hence expose this functionality to the application.

8.3.2. Early Data

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

8.3.3. Receiving Final Messages

The Message Context can indicate whether or not this Message is the
Final Message on a Connection. For any Message that is marked as
Final, the application can assume that there will be no more Messages
received on the Connection once the Message has been completely
delivered. This corresponds to the Final property that may be marked
on a sent Message, see Section 7.5.5.

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 Framers

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

Message Framers allow extending a Connection's Protocol Stack to
define how to encapsulate or encode outbound Messages, and how to
decapsulate or decode inbound data into Messages. Message Framers
allow message boundaries to be preserved when using a Connection

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

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

```
   Initiate()   Send()   Receive()   Close()
       |          |          ^          |
       |          |          |          |
   +----v----------v---------+----------v-----+
   |                 Connection                |
   +----+----------+---------^----------+-----+
        |          |          |          |
        |      +----------------+        |
        |      |   Messages     |        |
        |      +----------------+        |
        |          |          |          |
   +----v----------v---------+----------v-----+
   |                  Framer(s)                |
   +----+----------+---------^----------+-----+
        |          |          |          |
        |      +----------------+        |
        |      |  Byte-stream   |        |
        |      +----------------+        |
        |          |          |          |
   +----v----------v---------+----------v-----+
   |          Transport Protocol Stack         |
   +-------------------------------------------+
```
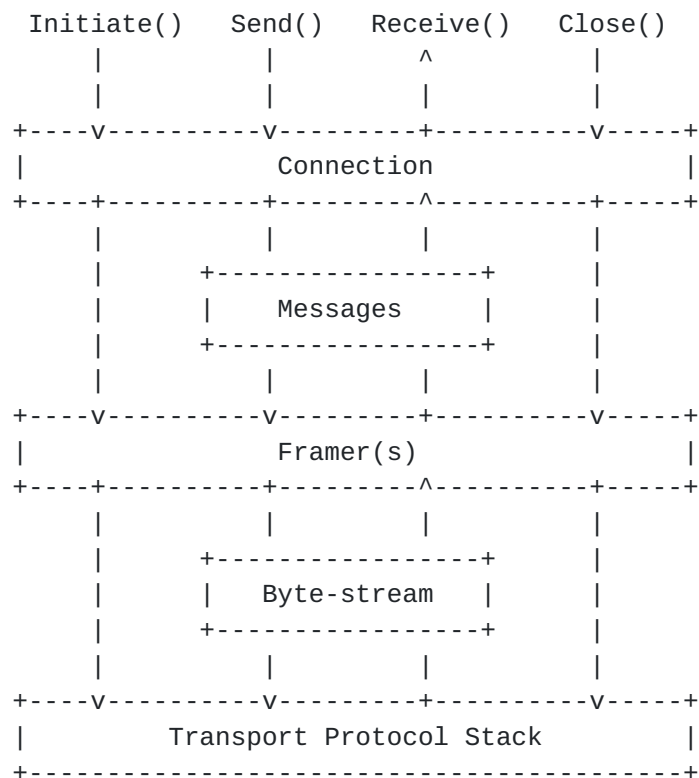
Note that while Message Framers add the most value when placed above
a protocol that otherwise does not preserve message boundaries, they
can also be used with datagram- or message-based protocols. In these
cases, they add an additional transformation to further encode or
encapsulate, and can potentially support packing multiple
application-layer Messages into individual transport datagrams.

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

9.1. Adding Message Framers to Connections

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

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

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

9.2. Framing Meta-Data

   When sending Messages, applications can add specific Message values
   to a MessageContext (Section 7.4) that is intended for a Framer.
   This can be used, for example, to set the type of a Message for a TLV
   format. The namespace of values is custom for each unique Message
   Framer.

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

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

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

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

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

10. 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 10.1 below. Specific Protocol Properties are defined in a
transport- and implementation-specific way, and must not be assumed
to apply across different protocols. Attempts to set Specific
Protocol Properties on a protocol stack not containing that specific
protocol are simply ignored, and do not raise an error; however, too
much reliance by an application on Specific Protocol Properties may
significantly reduce the flexibility of a transport services
implementation.

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

Connection.SetProperty(property, value)

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

At any point, the application can query Connection Properties.

ConnectionProperties := Connection.GetProperties()

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

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 7.5.5.

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 10.1)
      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.

10.1. Generic Connection Properties

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

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

10.1.1. Retransmission Threshold Before Excessive Retransmission
         Notification

   Name: retransmitNotifyThreshold

   Type: Integer, with special value "Disabled"

   Default: Disabled

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

10.1.2. Required Minimum Corruption Protection Coverage for Receiving

   Name: recvChecksumLen

   Type: Integer, with special value "Full Coverage"

   Default: Full Coverage

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

10.1.3. Priority (Connection)

Name: connPrio

Type: Integer

Default: 100

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

10.1.4. Timeout for Aborting Connection

Name: connTimeout

Type: Numeric, with special value "Disabled"

Default: Disabled

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

10.1.5. Connection Group Transmission Scheduler

Name: connScheduler

Type: Enumeration

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

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

10.1.6. Capacity Profile

   Name: connCapacityProfile

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

   The following values are valid for the Capacity Profile:

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

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

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

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

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

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

The Capacity Profile for a selected protocol stack may be modified on
a per-Message basis using the Transmission Profile Message Property;
see Section 7.5.8.

10.1.7. Policy for using Multi-Path Transports

Name: multipath-policy

Type: Enumeration

Default: Handover

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

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

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

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

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

10.1.8. Bounds on Send or Receive Rate

Name: maxSendRate / maxRecvRate

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

Default: Unlimited / Unlimited

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

10.1.9. Read-only Connection Properties

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

10.1.9.1. Maximum Message Size Concurrent with Connection Establishment

Name: zeroRttMsgMaxLen

Type: Integer

This property represents the maximum Message size that can be sent
before or during Connection establishment, see also Section 7.5.4.
It is given in Bytes.

10.1.9.2. Maximum Message Size Before Fragmentation or Segmentation

Name: singularTransmissionMsgMaxLen

Type: Integer

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

10.1.9.3. Maximum Message Size on Send

Name: sendMsgMaxLen

Type: Integer

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

10.1.9.4. Maximum Message Size on Receive

Name: recvMsgMaxLen

Type: Integer

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

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

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

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

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

10.2.1. Advertised User Timeout

Name: tcp.userTimeoutValue

Type: Integer

Default: the TCP default

This time value is advertised via the TCP User Timeout Option (UTO)
[RFC5482] at the remote endpoint to adapt its own "Timeout for
aborting Connection" (see Section 10.1.4) value accordingly.

10.2.2. User Timeout Enabled

Name: tcp.userTimeout

Type: Boolean

Default: false

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

10.2.3. Timeout Changeable

Name: tcp.userTimeoutRecv

Type: Boolean

Default: true

This property controls whether the "Timeout for aborting Connection"
(see Section 10.1.4) may be changed based on a UTO option received
from the remote peer. This boolean becomes false when "Timeout for
aborting Connection" (see Section 10.1.4) is used.

10.3. Connection Lifecycle Events

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

10.3.1. Soft Errors

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

   Connection -> SoftError<>

10.3.2. Excessive retransmissions

   This event notifies the application of excessive retransmissions,
   based on a configured threshold (see Section 10.1.1). This will only
   happen if the underlying protocol stack supports reliability and,
   with it, such notifications.

   Connection -> ExcessiveRetransmission<>

11. Connection Termination

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

   Connection.Close()

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

   Connection -> Closed<>

   Abort terminates a Connection without delivering remaining data:

   Connection.Abort()

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

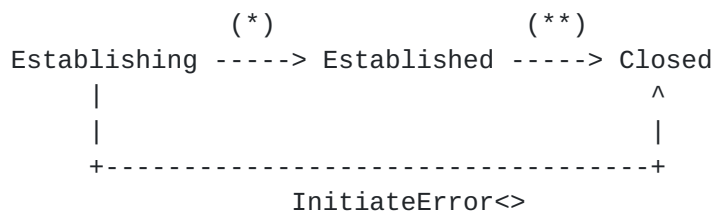   Connection -> ConnectionError<reason?>

12. Connection State and Ordering of Operations and Events

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

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

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

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

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

                  Figure 1: Connection State Diagram

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

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.

13. 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).

14. Security Considerations

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

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

In either case, the TAPS API is an internal interface that is used to
change information locally between two systems. However, as the
transport system is responsible for network communication, it is in
the position to potentially share any information provided by the

   application with the network or another communication peer. Most of
   the information provided over the TAPS API are useful to configure
   and select protocols and paths and are not necessarily privacy
   sensitive. Still, there is some information that could be privacy
   sensitve because this might reveal usage characteristics and habits
   of the user of an application.

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

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

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

15. Acknowledgements

16. References

16.1. Normative References

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

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

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

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

[RFC8303] Welzl, M., Tuexen, M., and N. Khademi, "On the Usage of
          Transport Features Provided by IETF Transport Protocols",
          RFC 8303, DOI 10.17487/RFC8303, February 2018,
          <https://www.rfc-editor.org/info/rfc8303>.

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

16.2. **Informative References**

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

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

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

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

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

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

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

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

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

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

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

   [RFC5482] Eggert, L. and F. Gont, "TCP User Timeout Option",
              RFC 5482, DOI 10.17487/RFC5482, March 2009,
              <https://www.rfc-editor.org/info/rfc5482>.

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

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

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

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

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

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

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

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

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

   [RFC8622]   Bless, R., "A Lower-Effort Per-Hop Behavior (LE PHB) for
               Differentiated Services", RFC 8622, DOI 10.17487/RFC8622,
               June 2019, <https://www.rfc-editor.org/info/rfc8622>.

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

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

Appendix A. Convenience Functions

A.1. Adding Preference Properties

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

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

A.2. Transport Property Profiles

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

A.2.1. reliable-inorder-stream

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

```
+-----------------------+---------+
| Property              | Value   |
+-----------------------+---------+
| reliability           | require |
|                       |         |
| preserveOrder         | require |
|                       |         |
| congestionControl     | require |
|                       |         |
| preserveMsgBoundaries | ignore  |
+-----------------------+---------+
```

A.2.2. reliable-message

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

```
              +-----------------------+---------+
              | Property              | Value   |
              +-----------------------+---------+
              | reliability           | require |
              |                       |         |
              | preserveOrder         | require |
              |                       |         |
              | congestionControl     | require |
              |                       |         |
              | preserveMsgBoundaries | require |
              +-----------------------+---------+
```

[A.2.3](#). unreliable-datagram

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

```
              +-----------------------+---------+
              | Property              | Value   |
              +-----------------------+---------+
              | reliability           | ignore  |
              |                       |         |
              | preserveOrder         | ignore  |
              |                       |         |
              | congestionControl     | ignore  |
              |                       |         |
              | preserveMsgBoundaries | require |
              |                       |         |
              | safely replayable     | true    |
              +-----------------------+---------+
```

Applications that choose this Transport Property Profile for latency
reasons should also consider setting the Capacity Profile Property,
see [Section 10.1.6](#) accordingly and my benefit from controlling
checksum coverage, see [Section 5.2.7](#) and [Section 5.2.8](#).

[Appendix B](#). Relationship to the Minimal Set of Transport Services for
            End Systems

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

according to the discussion in Section 5 of [I-D.ietf-taps-minset].
This list is a subset of the transport features in Appendix A of
[I-D.ietf-taps-minset], which refers to the primitives in "pass 2"
(Section 4) of [RFC8303] for further details on the implementation
with TCP, MPTCP, UDP, UDP-Lite, SCTP and LEDBAT.

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

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

o Specify number of attempts and/or timeout for the first
   establishment message: "timeout" parameter of "Initiate"
   (Section 6.1) or "InitiateWithSend" Action (Section 7.8).

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

o Hand over a message to reliably transfer (possibly multiple times)
   before connection establishment: "InitiateWithSend" Action
   (Section 7.8).

o Change timeout for aborting connection (using retransmit limit or
   time value): "Timeout for Aborting Connection" property, using a
   time value (Section 10.1.4).

o Timeout event when data could not be delivered for too long:
   "ConnectionError" Event (Section 11).

o Suggest timeout to the peer: "TCP-specific Property: User Timeout"
   (Section 10.2).

o Notification of Excessive Retransmissions (early warning below
   abortion threshold): "Notification of excessive retransmissions"
   property (Section 5.2.16).

o Notification of ICMP error message arrival: "Notification of ICMP
   soft error message arrival" property (Section 5.2.17).

o Choose a scheduler to operate between streams of an association:
   "Connection Group Transmission Scheduler" property
   (Section 10.1.5).

o Configure priority or weight for a scheduler: "Priority
   (Connection)" property (Section 10.1.3).

o "Specify checksum coverage used by the sender" and "Disable
   checksum when sending": "Corruption Protection Length" property

(Section 7.5.6) and "Full Checksum Coverage on Sending" property
(Section 5.2.7).

o "Specify minimum checksum coverage required by receiver" and
   "Disable checksum requirement when receiving": "Required Minimum
   Corruption Protection Coverage for Receiving" property
   (Section 10.1.2) and "Full Checksum Coverage on Receiving"
   property (Section 5.2.8).

o "Specify DF" field and "Request not to bundle messages": the "No
   Fragmentation" Message Property combines both of these requests,
   i.e. if a request not to bundle messages is made, this also turns
   off fragmentation (i.e., sets DF=1) in the case of a protocol that
   allows this (only UDP and UDP-Lite, which cannot bundle messages
   anyway) (Section 7.5.9).

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
   (Section 10.1.9.2).

o Get max. transport-message size that may be received from the
   configured interface: "Maximum Message Size on Receive" property
   (Section 10.1.9.4).

o Obtain ECN field: "ECN" is a defined UDP(-Lite)-specific read-only
   Message Property of the MessageContext object (Section 8.3.1).

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
   (Section 10.1.6). Per-Message control is offered via the "Message
   Capacity Profile Override" property (Section 7.5.8).

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]
   (Section 11).

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
   (Section 11).

o "Reliably transfer data, with congestion control", "Reliably
   transfer a message, with congestion control" and "Unreliably
   transfer a message": data is transferred via the "Send" action
   (Section 7). Reliability is controlled via the "Reliable Data
   Transfer (Connection)" (Section 5.2.1) property and the "Reliable
   Data Transfer (Message)" Message Property (Section 7.5.7).
   Transmitting data as a message or without delimiters is controlled
   via Message Framers (Section 9). The choice of congestion control
   is provided via the "Congestion control" property (Section 5.2.9).

o Configurable Message Reliability: the "Lifetime" Message Property
   implements a time-based way to configure message reliability
   (Section 7.5.1).

o "Ordered message delivery (potentially slower than unordered)" and
   "Unordered message delivery (potentially faster than ordered)":
   these two transport features are controlled via the Message
   Property "Ordered" (Section 7.5.3).

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 apply when an application uses
   the "Capacity Profile" Property (Section 10.1.6) or the "Message
   Capacity Profile Override" Message Property (Section 7.5.8) with
   value "Low Latency/Interactive".

o Receive data (with no message delimiting): "Received" Event
   (Section 8.2.1). See Section 9 for handling Message framing in
   situations where the Protocol Stack only provides a byte-stream
   transport.

o Receive a message: "Received" Event (Section 8.2.1), using Message
   Framers (Section 9).

o Information about partial message arrival: "ReceivedPartial" Event
   (Section 8.2.2).

o Notification of send failures: "Expired" Event (Section 7.3.2) and
   "SendError" Event (Section 7.3.3).

o Notification that the stack has no more user data to send:
   applications can obtain this information via the "Sent" Event
   (Section 7.3.1).

o Notification to a receiver that a partial message delivery has
   been aborted: "ReceiveError" Event (Section 8.2.3).

Authors' Addresses

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

    Email: ietf@trammell.ch

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

    Email: michawe@ifi.uio.no

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

    Email: ietf@tenghardt.net

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

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

    Mirja Kuehlewind
    Ericsson
    Ericsson-Allee 1
    Herzogenrath
    Germany

    Email: mirja.kuehlewind@ericsson.com

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

Email: csp@csperkins.org

Philipp S. Tiesel
TU Berlin
Einsteinufer 25
10587 Berlin
Germany

Email: philipp@tiesel.net

Christopher A. Wood
Cloudflare
101 Townsend St
San Francisco
United States of America

Email: caw@heapingbits.net

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

Email: tpauly@apple.com