Stream: Internet Engineering Task Force (IETF)
RFC: 9622
Category: Standards Track
Published: January 2025
ISSN: 2070-1721
Authors: B. Trammell, Ed. *Google Switzerland GmbH*
M. Welzl, Ed. *University of Oslo*
R. Enghardt *Netflix*
G. Fairhurst *University of Aberdeen*
M. Kühlewind *Ericsson*
C. S. Perkins *University of Glasgow*
P.S. Tiesel *SAP SE*
T. Pauly *Apple Inc.*

RFC 9622
An Abstract Application Programming Interface (API) for Transport Services

Abstract

This document describes an abstract Application Programming Interface (API) to the transport layer that enables the selection of transport protocols and network paths dynamically at runtime. This API enables faster deployment of new protocols and protocol features without requiring changes to the applications. The specified API follows the Transport Services Architecture by providing asynchronous, atomic transmission of Messages. It is intended to replace the BSD Socket API as the common interface to the transport layer, in an environment where endpoints could select from multiple network paths and potential transport protocols.

Status of This Memo

This is an Internet Standards Track document.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Further information on Internet Standards is available in Section 2 of RFC 7841.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at https://www.rfc-editor.org/info/rfc9622.

Copyright Notice

Table of Contents

1. Introduction

This document specifies an abstract Application Programming Interface (API) that describes the interface component of the high-level Transport Services Architecture defined in [RFC9621]. A Transport Services System supports asynchronous, atomic transmission of Messages over transport protocols and network paths dynamically selected at runtime, in environments where an endpoint selects from multiple network paths and potential transport protocols.

Applications that adopt this API will benefit from a wide set of transport features that can evolve over time. This protocol-independent API ensures that the system providing the API 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 and can support applications by offering racing and fallback mechanisms, which otherwise need to be separately implemented in each application. Transport Services Implementations are free to take any desired form as long as the API specification in this document is honored; a non-prescriptive guide to implementing a Transport Services System is available (see [RFC9623]).

The Transport Services System derives specific path and Protocol Selection Properties and supported transport features from the analysis provided in [RFC8095], [RFC8923], and [RFC8922]. The Transport Services API enables an implementation to dynamically choose a transport protocol rather than statically binding applications to a protocol at compile time. The Transport Services API also provides 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. However, forcing a choice to use a specific Protocol Stack is discouraged for general use because it can reduce portability.

1.1. Terminology and Notation

The Transport Services 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 to be processed asynchronously; and
- Parameters associated with these actions and events.

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

- An action that creates an object:

```
Object := Action()
```

- An action that creates an array of objects:

```
[]Object := Action()
```

- An action that is performed on an object:

```
Object.Action()
```

- An object sends an event:

```
Object -> Event<>
```

- An action takes a set of parameters; an event contains a set of parameters. Action and event parameters whose names are suffixed with a question mark are optional:

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

Objects that are passed as parameters to actions use call-by-value behavior. Actions not associated with an object are actions on the API; they are equivalent to actions on a per-application global context.

Events are sent to the application or application-supplied code (e.g., Framers; see Section 9.1.2) for processing; the details of event interfaces are specific to the platform or implementation and can be implemented using other forms of asynchronous processing, as idiomatic for the implementing platform.

We also make use of the following basic types:

Boolean: Instances take the value `true` or `false`.

Integer: Instances take integer values.

Numeric: Instances take real number values.

String: Instances are represented in UTF-8.

IP Address: An IPv4 address [RFC791] or IPv6 address [RFC4291].

Enumeration: A family of types in which each instance takes one of a fixed, predefined set of values specific to a given enumerated type.

Tuple: An ordered grouping of multiple value types, represented as a comma-separated list in parentheses, e.g., `(Enumeration, Preference)`. Instances take a sequence of values, each valid for the corresponding value type.

Array: Denoted [ ]Type, an instance takes a value for each of zero or more elements in a sequence of the given Type. An array can be of fixed or variable length.

Set: An unordered grouping of one or more different values of the same type.

For guidance on how these abstract concepts can be implemented in languages in accordance with language-specific design patterns and platform features, see Appendix A.

1.2. Specification of Requirements

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.

2. Overview of the API Design

The design of the API specified in this document is based on a set of principles, themselves an elaboration on the architectural design principles defined in [RFC9621]. The API defined in this document provides:

- A Transport Services System that can offer a variety of transport protocols, independent of the Protocol Stacks that will be used at runtime. To the degree possible, all common features of these Protocol Stacks are made available to the application in a transport-independent way. This enables applications written for a single API to make use of transport protocols in terms of the features they provide.
- A unified API to datagram and stream-oriented transports, allowing the use of a common API for Connection establishment and closing.
- Message-orientation, as opposed to stream-orientation, using application-assisted framing and deframing where the underlying transport does not itself provide the required framing.
- Asynchronous Connection establishment, transmission, and reception. This allows concurrent operations during establishment and event-driven application interactions with the transport layer.
- Selection between alternate network paths, using additional information about the networks over which a Connection can operate (e.g., Provisioning Domain (PvD) information [RFC7556]) where available.
- Explicit support for transport-specific features to be applied, when that particular transport is part of a chosen Protocol Stack.
- Explicit support for security properties as first-order transport features.
- Explicit support for configuration of cryptographic identities and transport Security Parameters persistent across multiple Connections.
- Explicit support for multistreaming and multipath transport protocols, and the grouping of related Connections into Connection Groups through "cloning" of Connections (see Section 7.4). This function allows applications to take full advantage of new transport protocols supporting these features.

3. API Summary

An application primarily interacts with this API through two objects: Preconnections and Connections. A Preconnection object (Section 6) represents a set of Properties and constraints on the selection and configuration of paths and protocols to establish a Connection with an Endpoint. A Connection object represents an instance of a transport Protocol Stack on which data can be sent to and/or received from a Remote Endpoint (i.e., a logical connection that, depending on the kind of transport, can be bidirectional or unidirectional, and that can use a stream protocol or a datagram protocol). Connections are presented consistently to the application, irrespective of whether the underlying transport is connectionless or connection oriented. Connections can be created from Preconnections in three ways:

- initiating the Preconnection (i.e., creating a Connection from the Preconnection, actively opening, as in a client; see `Initiate` in Section 7.1),
- listening on the Preconnection (i.e., creating a Listener based on the Preconnection, passively opening, as in a server; see `Listen` in Section 7.2), or
- a rendezvous for the Preconnection (i.e., peer-to-peer Connection establishment; see `Rendezvous` in Section 7.3).

Once a Connection is established, data can be sent and received on it in the form of Messages. The API supports the preservation of Message boundaries via both explicit Protocol Stack support and application support through a Message Framer that 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 can have implementation-specific default handlers.

The application **SHOULD NOT** assume that ignoring events (e.g., errors) is always safe.

3.1. Usage Examples

The following usage examples illustrate how an application might use the Transport Services API to act as:

- a server, by listening for incoming Connections, receiving requests, and sending responses; see Section 3.1.1.
- a client, by connecting to a Remote Endpoint using `Initiate`, sending requests, and receiving responses; see Section 3.1.2.
- a peer, by connecting to a Remote Endpoint using `Rendezvous` while simultaneously waiting for incoming Connections, sending Messages, and receiving Messages; see Section 3.1.3.

The examples in this section presume that a transport protocol is available between the Local and Remote Endpoints and that this protocol 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 provide preservation of Message boundaries, but there is a transport protocol that provides a reliable ordered byte-stream, an application could receive this byte-stream as partial Messages and transform it into application-layer Messages. Alternatively, an application might provide a Message Framer, which can transform a sequence of Messages into a byte-stream and vice versa (Section 9.1.2).

3.1.1. Server Example

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

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

TransportProperties := NewTransportProperties()
TransportProperties.Require(preserveMsgBoundaries)
// Reliable data transfer and preserve order are required by default

SecurityParameters := NewSecurityParameters()
SecurityParameters.Set(serverCertificate, myCertificate)

// 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 ----
```

3.1.2. Client Example

This is an example of how an application might open two Connections to a remote application using the Transport Services API, send a request, and receive a response for each of the two Connections. The code designated with comments as "Ready event handler" could, for example, be implemented as a callback function. This function would receive the Connection that it expects to operate on ("Connection" and "Connection2" in the example) handed over using the variable name "C".

```
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 the identity of the Remote Endpoint and return the result
})
SecurityParameters.SetTrustVerificationCallback(TrustCallback)

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

Connection := Preconnection.Initiate()
Connection2 := Connection.Clone()

Connection -> Ready<>
Connection2 -> Ready<>

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

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

Connection -> Received<messageDataResponse, messageContext>
Connection2 -> Received<messageDataResponse, messageContext>

// Close the Connection in a Receive event handler
Connection.Close()
Connection2.Close()
```

A Preconnection serves as a template for creating a Connection via initiating, listening, or via rendezvous. Once a Connection has been created, changes made to the Preconnection that was used to create it do not affect this Connection. Preconnections are reusable after being used to create a Connection, whether or not this Connection was closed. Hence, in the above example, it would be correct for the client to initiate a third Connection to the `example.com` server by continuing as follows:

```
//.. carry out adjustments to the Preconnection, if desired
Connection3 := Preconnection.Initiate()
```

3.1.3. 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.

```
// Configure local candidates: a port on the Local Endpoint
// and via a Session Traversal Utilities for NAT (STUN) server
HostCandidate := NewLocalEndpoint()
HostCandidate.WithPort(9876)

StunCandidate := NewLocalEndpoint()
StunCandidate.WithStunServer(address, port, credentials)

LocalCandidates = [HostCandidate, StunCandidate]

TransportProperties := // ...Configure transport properties
SecurityParameters  := // ...Configure security properties

Preconnection := NewPreconnection(LocalCandidates,
                                  [], // No remote candidates yet
                                  TransportProperties,
                                  SecurityParameters)

// Resolve the LocalCandidates.  The Preconnection.Resolve()
// call resolves both local and remote candidates; however,
// because the remote candidates have not yet been specified,
// the ResolvedRemote list returned will be empty and is not
// used.
ResolvedLocal, ResolvedRemote = Preconnection.Resolve()

// Application-specific code goes here to send the ResolvedLocal
// list to the peer via some out-of-band signaling channel (e.g.,
// in a SIP message).
...

// Application-specific code goes here to receive RemoteCandidates
// (type []RemoteEndpoint, a list of RemoteEndpoint objects) from
// the peer via the signaling channel.
...

// Add remote candidates and initiate the rendezvous:
Preconnection.AddRemote(RemoteCandidates)
Preconnection.Rendezvous()

Preconnection -> RendezvousDone<Connection>

//---- RendezvousDone event handler begin ----
Connection.Send(messageDataRequest)
Connection.Receive()
//---- RendezvousDone event handler end ----

Connection -> Received<messageDataResponse, messageContext>

// If new Remote Endpoint candidates are received from the
// peer over the signaling channel -- for example, if using
// Trickle Interactive Connectivity Establishment (ICE) --
// then add them to the Connection:
Connection.AddRemote(NewRemoteCandidates)

// On a PathChange event, resolve the Local Endpoint Identifiers to
// see if a new Local Endpoint has become available and, if
// so, send to the peer as a new candidate and add to the
```

```
// Connection:
Connection -> PathChange<>

//---- PathChange event handler begin ----
ResolvedLocal, ResolvedRemote = Preconnection.Resolve()
if ResolvedLocal has changed:
  // Application-specific code goes here to send the
  // ResolvedLocal list to the peer via the signaling channel
  ...

  // Add the new Local Endpoints to the Connection:
  Connection.AddLocal(ResolvedLocal)
//---- PathChange event handler end ----

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

4. Transport Properties

Each application using the Transport Services API declares its preferences for how the Transport Services System is to operate. This is done by using Transport Properties, as defined in [RFC9621], at each stage of the lifetime of a Connection.

Transport Properties are divided into Selection, Connection, and Message Properties.

Selection Properties (see Section 6.2) can only be set during preestablishment. They are only used to specify which paths and Protocol Stacks can be used and are preferred by the application. Calling `Initiate` on a Preconnection creates an outbound Connection, and the Selection Properties remain readable from the Connection but become immutable. Selection Properties can be set on Preconnections, and the effect of Selection Properties can be queried on Connections and Messages.

Connection Properties (see Section 8.1) are used to inform decisions made during establishment and to fine-tune the established Connection. They can be set during preestablishment and can be changed later. Connection Properties can be set on Connections and Preconnections; when set on Preconnections, they act as an initial default for the resulting Connections.

Message Properties (see Section 9.1.3) control the behavior of the selected Protocol Stack(s) when sending Messages. Message Properties can be set on Messages, Connections, and Preconnections; when set on the latter two, they act as an initial default for the Messages sent over those Connections.

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 [RFC9621]), are not used as input to the selection process; they only support configuration if the respective protocol has been selected.

4.1. Transport Property Names

Transport Properties are referred to by names, represented as case-insensitive strings. These names serve two purposes:

- Allowing different components of a Transport Services Implementation to pass Transport Properties (e.g., between a language frontend and a policy manager) or to enable a Transport Services Implementation to represent Properties retrieved from a file or other storage to the application.
- Making the code of different Transport Services Implementations look similar. While individual programming languages might preclude strict adherence to the naming convention of representing Property names as case-insensitive strings (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>.

- The optional Namespace component and its trailing dot character (`"."`) **MUST** be omitted for well-known generic Properties, i.e., for Properties that are not specific to a protocol.
- Protocol-specific Properties **MUST** use the protocol acronym as the Namespace (e.g., a Connection that uses TCP could support a TCP-specific Transport Property, such as the TCP User Timeout value, in a Protocol-specific Property called `tcp.userTimeoutValue` (see Section 8.2)).
- Vendor-specific or implementation-specific Properties **MUST** be placed in a Namespace starting with the underscore character (`"_"`) and **SHOULD** use a string identifying the vendor or implementation.
- For IETF protocols, the name of a Protocol-specific Property **MUST** be specified in an RFC from the IETF Stream (after IETF Review [RFC8126]). An IETF protocol Namespace does not start with an underscore character (`"_"`).

Namespaces for each of the keywords provided in the "Protocol Numbers" registry (see <https://www.iana.org/assignments/protocol-numbers/>) are reserved for Protocol-specific Properties and **MUST NOT** be used for vendor-specific or implementation-specific Properties. Terms listed as keywords, as in the "Protocol Numbers" registry, **SHOULD** be avoided as any part of a vendor-specific or implementation-specific Property name.

Though Transport Property names are case insensitive, it is recommended to use camelCase to improve readability. Implementations may transpose Transport Property names into snake_case or PascalCase to blend into the language environment.

4.2. Transport Property Types

Each Transport Property has one of the basic types described in Section 1.1.

Most Selection Properties (see Section 6.2) are of the Enumeration type, and they use the Preference Enumeration, which takes one of five possible values (`Prohibit`, `Avoid`, `No Preference`, `Prefer`, or `Require`) denoting the level of preference for a given Property during protocol selection.

5. Scope of the API Definition

This document defines a language- and platform-independent API of 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 API necessarily abstract.

There is no interoperability benefit in tightly defining how the API is presented to application programmers across diverse platforms. However, maintaining the "shape" of the abstract API across different platforms reduces the effort for programmers who learn to use the Transport Services API to then apply their knowledge to another platform. That said, implementations have significant freedom in presenting this API to programmers, balancing the conventions of the protocol with the shape of the API. We make the following recommendations:

- Actions, events, and errors in implementations of the Transport Services API **SHOULD** use the names assigned to them in this 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.
- Transport Services Systems **SHOULD** implement each Selection Property, Connection Property, and MessageContext Property specified in this document. These features **SHOULD** be implemented even when, in a specific implementation, it 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 (trivially, as a no-op) as disabling the requirement for ordering will not have any effect on delivery order for Connections over TCP. Similarly, the `msgLifetime` Message Property can be implemented but ignored, as the description of this Property (Section 9.1.3.1) states that "it is not guaranteed that a Message will not be sent when its Lifetime has expired".
- Implementations can use other representations for Transport Property names, e.g., by providing constants, but should provide a straightforward mapping between their representation and the Property names specified here.

6. Preestablishment Phase

The preestablishment phase allows applications to specify Properties for the Connections that they are about to make or to query the API about potential Connections they could make.

A Preconnection object represents a potential Connection. It is a passive object (a data structure) that merely maintains the state that describes the 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 6.1), the Selection Properties (see Section 6.2), any preconfigured Connection Properties (Section 8.1), and the Security Parameters (see Section 6.3):

```
Preconnection := NewPreconnection([]LocalEndpoint,
                                  []RemoteEndpoint,
                                  TransportProperties,
                                  SecurityParameters)
```

At least one Local Endpoint **MUST** be specified if the Preconnection is used to `Listen` for incoming Connections, but the list of Local Endpoints **MAY** be empty if the Preconnection is used to `Initiate` connections. If no Local Endpoint is specified, the Transport Services System will assign an ephemeral local port to the Connection on the appropriate interface(s). At least one Remote Endpoint **MUST** be specified if the Preconnection is used to `Initiate` Connections, but the list of Remote Endpoints **MAY** be empty if the Preconnection is used to `Listen` for incoming Connections. At least one Local Endpoint and one Remote Endpoint **MUST** be specified if a peer-to-peer `Rendezvous` is to occur based on the Preconnection.

If more than one Local Endpoint is specified on a Preconnection, then the application is indicating that all of the Local Endpoints are eligible to be used for Connections. For example, their Endpoint Identifiers might correspond to different interfaces on a multihomed host or their Endpoint Identifiers might correspond to local interfaces and a STUN server that can be resolved to a server-reflexive address for a Preconnection used to make a peer-to-peer `Rendezvous`.

If more than one Remote Endpoint is specified on the Preconnection, the application is indicating that it expects all of the Remote Endpoints to offer an equivalent service and that the Transport Services System can choose any of them for a Connection. For example, a Remote Endpoint might represent various network interfaces of a host, or a server-reflexive address that can be used to reach a host, or a set of hosts that provide equivalent local balanced service.

In most cases, it is expected that a single Remote Endpoint will be specified by name, and a later call to `Initiate` on the Preconnection (see Section 7.1) will internally resolve that name to a list of concrete Endpoint Identifiers. Specifying multiple Remote Endpoints on a Preconnection allows applications to override this for more detailed control.

If Message Framers are used (see Section 9.1.2), they **MUST** be added to the Preconnection during preestablishment.

6.1. Specifying Endpoints

The Transport Services API uses the Local Endpoint and Remote Endpoint objects to refer to the Endpoints of a Connection. Endpoints can be created as either remote or local:

```
RemoteSpecifier := NewRemoteEndpoint()
LocalSpecifier := NewLocalEndpoint()
```

A single Endpoint object represents the identity of a network host. That Endpoint can be more or less specific, depending on which Endpoint Identifiers are set. For example, an Endpoint that only specifies a hostname can, in fact, finally correspond to several different IP addresses on different hosts.

An Endpoint object can be configured with the following identifiers:

- HostName (string):

```
RemoteSpecifier.WithHostName("example.com")
```

- Port (a 16-bit unsigned Integer):

```
RemoteSpecifier.WithPort(443)
```

- Service (an identifier string that maps to a port; either a service name associated with a port number (from <https://www.iana.org/assignments/service-names-port-numbers/>) or a DNS SRV service name to be resolved):

```
RemoteSpecifier.WithService("https")
```

- IP address (an IPv4 or IPv6 address type; note that the examples here show the human-readable form of the IP addresses, but the functions can take a binary encoding of the addresses):

```
RemoteSpecifier.WithIPAddress(192.0.2.21)
```

```
RemoteSpecifier.WithIPAddress(2001:db8:4920:e29d:a420:7461:7073:a)
```

- Interface identifier (which can be a string name or other platform-specific identifier), e.g., to qualify link-local addresses (see Section 6.1.2 for details):

```
LocalSpecifier.WithInterface("en0")
```

The `Resolve` action on a Preconnection can be used to obtain a list of available local interfaces.

Note that an IPv6 address specified with a scope zone ID (e.g., `fe80::2001:db8%en0`) is equivalent to `WithIPAddress` with an unscoped address and `WithInterface` together.

Applications creating Endpoint objects using `WithHostName` **SHOULD** provide Fully Qualified Domain Names (FQDNs). Not providing an FQDN will result in the Transport Services Implementation needing to use DNS search domains for name resolution, which might lead to inconsistent or unpredictable behavior.

The design of the API **MUST NOT** permit an Endpoint object to be configured with multiple Endpoint Identifiers of the same type. For example, an Endpoint object cannot specify two IP addresses. Two separate IP addresses are represented as two Endpoint objects. If a Preconnection specifies a Remote Endpoint with a specific IP address set, it will only establish Connections to that IP address. If, on the other hand, a Remote Endpoint specifies a hostname but no addresses, the Transport Services Implementation can perform name resolution and attempt using any address derived from the original hostname of the Remote Endpoint. Note that multiple Remote Endpoints can be added to a Preconnection, as discussed in Section 7.5.

The Transport Services System resolves names internally, when the `Initiate`, `Listen`, or `Rendezvous` action is called to establish a Connection. Privacy considerations for the timing of this resolution are given in Section 13.

The `Resolve` action on a 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 7.3).

6.1.1. Using Multicast Endpoints

To use multicast, a Preconnection is first created with the Local or Remote Endpoint Identifier specifying the Any-Source Multicast (ASM) or Source-Specific Multicast (SSM) group and destination port number. This is then followed by a call to either `Initiate`, `Listen`, or `Rendezvous`, depending on whether the resulting Connection is to be used to send Messages to the multicast group, receive Messages from the group, or both send and receive Messages (as is the case for an ASM group).

Note that the Transport Services API has separate specifier calls for multicast groups to avoid introducing filter Properties for single-source multicast and seeks to avoid confusion that can be caused by overloading the unicast specifiers.

Calling `Initiate` on that Preconnection creates a Connection that can be used to send Messages to the multicast group. The Connection object that is created will support `Send` but not `Receive`. Any Connections created this way are send-only and do not join the multicast group. The resulting Connection will have a Local Endpoint identifying the local interface to which the Connection is bound and a Remote Endpoint identifying the multicast group.

The following API calls can be used to configure a Preconnection before calling `Initiate`:

```
RemoteSpecifier.WithMulticastGroupIP(GroupAddress)
RemoteSpecifier.WithPort(PortNumber)
RemoteSpecifier.WithHopLimit(HopLimit)
```

Calling `Listen` on a Preconnection with a multicast group address specified as the Remote Endpoint Identifier will trigger the Transport Services Implementation to join the multicast group to receive Messages. This Listener will create one Connection for each Remote Endpoint sending to the group, with the Local Endpoint Identifier specified as a group address. The set of Connection objects created forms a Connection Group. The receiving interface can be restricted by passing it as part of the LocalSpecifier or queried through the MessageContext on the Messages received (see Section 9.1.1 for further details).

Specifying `WithHopLimit` sets the Time To Live (TTL) field in the header of IPv4 packets or the Hop Count field in the header of IPv6 packets.

The following API calls can be used to configure a Preconnection before calling `Listen`:

```
LocalSpecifier.WithSingleSourceMulticastGroupIP(GroupAddress,
                                                SourceAddress)
LocalSpecifier.WithAnySourceMulticastGroupIP(GroupAddress)
LocalSpecifier.WithPort(PortNumber)
```

Calling `Rendezvous` on a Preconnection with an ASM group address as the Remote Endpoint Identifier will trigger the Transport Services Implementation to join the multicast group and also indicates that the resulting Connection can be used to send Messages to the multicast group. The `Rendezvous` action will return both:

1. a Connection that can be used to send to the group and that acts the same as a Connection returned by calling `Initiate` with a multicast Remote Endpoint and
2. a Listener that acts as if `Listen` had been called with a multicast Remote Endpoint.

Calling `Rendezvous` on a Preconnection with an SSM group address as the Local Endpoint Identifier results in an `EstablishmentError`.

The following API calls can be used to configure a Preconnection before calling `Rendezvous`:

```
RemoteSpecifier.WithMulticastGroupIP(GroupAddress)
RemoteSpecifier.WithPort(PortNumber)
RemoteSpecifier.WithHopLimit(HopLimit)
LocalSpecifier.WithAnySourceMulticastGroupIP(GroupAddress)
LocalSpecifier.WithPort(PortNumber)
LocalSpecifier.WithHopLimit(HopLimit)
```

See Section 6.1.5 for more examples.

6.1.2. Constraining Interfaces for Endpoints

Note that this API has multiple ways to constrain and prioritize Endpoint candidates based on the network interface:

- Specifying an interface on a Remote Endpoint qualifies the scope zone of the Remote Endpoint, e.g., for link-local addresses.

- Specifying an interface on a Local Endpoint explicitly binds all candidates derived from this Endpoint to use the specified interface.
- Specifying an interface using the `interface` Selection Property (Section 6.2.11) or indirectly via the `pvd` Selection Property (Section 6.2.12) influences the selection among the available candidates.

While specifying an interface on an Endpoint restricts the candidates available for Connection establishment in the preestablishment phase, the Selection Properties prioritize and constrain the Connection establishment.

6.1.3. Protocol-Specific Endpoints

An Endpoint can have an alternative definition when using different protocols. For example, a server that supports both TLS/TCP and QUIC could be accessible on two different port numbers, depending on which protocol is used.

To scope an Endpoint to apply conditionally to a specific transport protocol (such as defining an alternate port to use when QUIC is selected, as opposed to TCP), an Endpoint can be associated with a protocol identifier. Protocol identifiers are objects or Enumeration values provided by the Transport Services API that will vary based on which protocols are implemented in a particular system.

```
AlternateRemoteSpecifier.WithProtocol(QUIC)
```

The following example shows a case where `example.com` has a server running on port 443 with an alternate port of 8443 for QUIC. Both endpoints can be passed when creating a Preconnection.

```
RemoteSpecifier := NewRemoteEndpoint()
RemoteSpecifier.WithHostName("example.com")
RemoteSpecifier.WithPort(443)

QUICRemoteSpecifier := NewRemoteEndpoint()
QUICRemoteSpecifier.WithHostName("example.com")
QUICRemoteSpecifier.WithPort(8443)
QUICRemoteSpecifier.WithProtocol(QUIC)

RemoteSpecifiers := [ RemoteSpecifier, QUICRemoteSpecifier ]
```

6.1.4. Endpoint Examples

The following examples of Endpoints show common usage patterns.

Specify a Remote Endpoint using a hostname `example.com` and a service name `https`, which tells the system to use the default port for HTTPS (443):

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

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

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

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

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

Specify a Local Endpoint using a local interface name and no local port to let the system assign an ephemeral local port:

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

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` Selection Property; see Section 6.2.11. However, if the application specifies Selection Properties that 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)
```

6.1.5. Multicast Examples

The following examples show how multicast groups can be used.

Join an ASM group in receive-only mode, bound to a known port on a named local interface:

```
    RemoteSpecifier := NewRemoteEndpoint()

    LocalSpecifier := NewLocalEndpoint()
    LocalSpecifier.WithAnySourceMulticastGroupIP(233.252.0.0)
    LocalSpecifier.WithPort(5353)
    LocalSpecifier.WithInterface("en0")

    TransportProperties := ...
    SecurityParameters  := ...

    Preconnection := NewPreconnection(LocalSpecifier,
                                      RemoteSpecifier,
                                      TransportProperties,
                                      SecurityProperties)
    Listener := Preconnection.Listen()
```

Join an SSM group in receive-only mode, bound to a known port on a named local interface:

```
    RemoteSpecifier := NewRemoteEndpoint()

    LocalSpecifier := NewLocalEndpoint()

    LocalSpecifier.WithSingleSourceMulticastGroupIP(233.252.0.0,
                                                    198.51.100.10)
    LocalSpecifier.WithPort(5353)
    LocalSpecifier.WithInterface("en0")

    TransportProperties := ...
    SecurityParameters  := ...

    Preconnection := NewPreconnection(LocalSpecifier,
                                      RemoteSpecifier,
                                      TransportProperties,
                                      SecurityProperties)
    Listener := Preconnection.Listen()
```

Create an SSM group as a sender:

```
RemoteSpecifier := NewRemoteEndpoint()
RemoteSpecifier.WithMulticastGroupIP(233.251.240.1)
RemoteSpecifier.WithPort(5353)
RemoteSpecifier.WithHopLimit(8)

LocalSpecifier := NewLocalEndpoint()
LocalSpecifier.WithIPAddress(192.0.2.22)
LocalSpecifier.WithInterface("en0")

TransportProperties := ...
SecurityParameters  := ...

Preconnection := NewPreconnection(LocalSpecifier,
                                  RemoteSpecifier,
                                  TransportProperties,
                                  SecurityProperties)
Connection := Preconnection.Initiate()
```

Join an ASM group as both a sender and a receiver:

```
RemoteSpecifier := NewRemoteEndpoint()
RemoteSpecifier.WithMulticastGroupIP(233.252.0.0)
RemoteSpecifier.WithPort(5353)
RemoteSpecifier.WithHopLimit(8)

LocalSpecifier := NewLocalEndpoint()
LocalSpecifier.WithAnySourceMulticastGroupIP(233.252.0.0)
LocalSpecifier.WithIPAddress(192.0.2.22)
LocalSpecifier.WithPort(5353)
LocalSpecifier.WithInterface("en0")


TransportProperties := ...
SecurityParameters  := ...

Preconnection := NewPreconnection(LocalSpecifier,
                                  RemoteSpecifier,
                                  TransportProperties,
                                  SecurityProperties)
Connection, Listener := Preconnection.Rendezvous()
```

6.2. 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 and Message Properties for configuration of the detailed operation of the selected Protocol Stacks on a per-Connection and per-Message level.

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 could involve choosing between multiple local interfaces that are connected to different access networks.

When additional information (such as PvD information [RFC7556]) is available about the networks over which an Endpoint can operate, this can inform the selection between alternate network paths. Path information can include the Path MTU (PMTU), the set of supported Differentiated Services Code Points (DSCPs), expected usage, cost, etc. The usage of this information by the Transport Services System is generally independent of the specific mechanism or protocol used to receive the information (e.g., zero-conf, DHCP, or IPv6 Router Advertisements (RAs)).

Most Selection Properties are represented as Preferences, which can take one of five values:

Preference	Effect
Require	Select only protocols/paths providing the Property; otherwise, fail
Prefer	Prefer protocols/paths providing the Property; otherwise, proceed
No Preference	No preference
Avoid	Prefer protocols/paths not providing the Property; otherwise, proceed
Prohibit	Select only protocols/paths not providing the Property; otherwise, fail

Table 1: Selection Property Preference Levels

The implementation **MUST** ensure an outcome that is consistent with all application requirements expressed using `Require` and `Prohibit`. While preferences expressed using `Prefer` and `Avoid` influence protocol and path selection as well, outcomes can vary, even given the same Selection Properties, because the available protocols and paths can differ across systems and contexts. However, implementations are **RECOMMENDED** to seek to provide a consistent outcome to an application, when provided with the same set of Selection Properties.

Note that application preferences can 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, applications can expect Properties that determine path selection to be prioritized over Properties that determine protocol selection. The Transport Services System **SHOULD** determine the preferred path first, regardless of protocol preferences. This ordering is chosen to provide consistency across implementations; this is based on the fact that it is more common for the use of a given network path to determine cost to the user (i.e., an interface type preference might be based on a user's preference to avoid being charged more for a cellular data plan).

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 set on the TransportProperties object. Setting a Transport Property to a value overrides the previous value of this Transport Property.

```
TransportProperties.Set(property, value)
```

To aid readability, implementations **MAY** provide additional convenience functions to simplify the use of Selection Properties: see Appendix B.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 B.2.

Transport Properties for an established Connection can be queried via the Connection object, as outlined in Section 8.

A Connection gets its Transport Properties by either being explicitly configured via a Preconnection, being configured after establishment, or inheriting them from an antecedent via cloning; see Section 7.4 for more details.

Section 8.1 provides a list of Connection Properties, while Selection Properties are listed in the subsections below. Selection Properties are only considered during establishment and cannot be changed after a Connection is established. At this point, Selection Properties can only be read to check the Properties used by the Connection. Upon reading, the Preference type of a Selection Property changes into Boolean, where:

- `true` means that the selected Protocol Stack supports the feature or uses the path associated with the Selection Property, and
- `false` means that the Protocol Stack does not support the feature or use the path.

Implementations of Transport Services Systems could alternatively use the `Require` and `Prohibit` Preference values to represent `true` and `false`, respectively. Other types of Selection Properties remain unchanged when they are made available for reading after a Connection is established.

An implementation of the Transport Services API needs to provide sensible defaults for Selection Properties. The 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 System, 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 Systems can be beneficial when applications are ported between different implementations/

platforms, even if this default could lead to a Connection failure when TCP is not available. If default values other than those suggested below are used, it is **RECOMMENDED** to clearly document any differences.

6.2.1. 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 at the Remote Endpoint in order, without loss or duplication. When reliable data transfer is enabled, this also entails being notified when a Connection is closed or aborted.

6.2.2. Preservation of Message Boundaries

Name: preserveMsgBoundaries

Type: Preference

Default: `No Preference`

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

6.2.3. Configure Per-Message Reliability

Name: perMsgReliability

Type: Preference

Default: `No Preference`

This Property specifies whether an application considers it useful to specify different reliability requirements for individual Messages in a Connection.

6.2.4. 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 at the Remote Endpoint in the same order as it was sent.

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

Name: zeroRttMsg

Type: Preference

Default: `No Preference`

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 Remote Endpoint before or during connection establishment. This Message can potentially be received multiple times (i.e., multiple copies of the Message data could be passed to the Remote Endpoint). See also Section 9.1.3.4.

6.2.6. Multistream Connections in a Group

Name: multistreaming

Type: Preference

Default: `Prefer`

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

6.2.7. Full Checksum Coverage on Sending

Name: fullChecksumSend

Type: Preference

Default: `Require`

This Property specifies the application's need for protection against corruption for all data transmitted on this Connection. Disabling this Property could enable the application to influence the sender checksum coverage after Connection establishment (see Section 9.1.3.6).

6.2.8. Full Checksum Coverage on Receiving

Name: fullChecksumRecv

Type: Preference

Default: `Require`

This Property specifies the application's need for protection against corruption for all data received on this Connection. Disabling this Property could enable the application to influence the required minimum receiver checksum coverage after Connection establishment (see Section 8.1.1).

6.2.9. Congestion Control

Name: congestionControl

Type: Preference

Default: `Require`

This Property specifies whether or not the application would like the Connection to be congestion controlled. Note that if a Connection is not congestion controlled, an application using such a Connection **SHOULD** itself perform congestion control in accordance with [RFC2914] or use a circuit breaker in accordance with [RFC8084], whichever is appropriate. 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. If the Connection is congestion controlled, performing additional congestion control in the application can have negative performance implications.

6.2.10. Keep-Alive Packets

Name: keepAlive

Type: Preference

Default: `No Preference`

This Property specifies whether or not the application would like the Connection to send keep-alive packets. Note that if a Connection determines that keep-alive packets are being sent, the application itself **SHOULD** avoid generating additional keep-alive Messages. Note that, when supported, the system will use the default period for generation of the keep-alive packets. (See also Section 8.1.4.)

6.2.11. Interface Instance or Type

Name: interface

Type: Set of (Preference, Enumeration)

Default: Empty (not setting a Preference for any interface)

This Property allows the application to select any 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 that single interface, and often leads to less flexible and resilient Connection establishment.

In contrast to other Selection Properties, this Property is a set of tuples of (enumerated) interface identifier and Preference. It can either be implemented directly as such or be implemented to make one Preference available for each interface and interface type available on the system.

The set of valid interface types is specific to the implementation or system. For example, on a mobile device, there could be `Wi-Fi` and `Cellular` interface types available; whereas, on a desktop computer, `Wi-Fi` and `Wired Ethernet` interface types might be available. An implementation should provide all types that are supported on the local system to allow applications to be written generically. For example, if a single implementation is used on both mobile devices and desktop devices, it ought to define the `Cellular` interface type for both systems, since an application might wish 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 6.2.12) or another specific Property.

Note that this Property is not used to specify an interface scope zone for a particular Endpoint. Section 6.1.2 provides details about how to qualify endpoint candidates on a per-interface basis.

6.2.12. Provisioning Domain Instance or Type

Name: pvd

Type: Set of (Preference, Enumeration)

Default: Empty (not setting a Preference for any PvD)

Similar to `interface` (see Section 6.2.11), this Property allows the application to control path selection by selecting which specific PvD or categories of PvDs it wants to `Require`, `Prohibit`, `Prefer`, or `Avoid`. Provisioning Domains define consistent sets of network properties that might be more specific than network interfaces [RFC7556].

As with `interface`, this Property is a set of tuples of (enumerated) PvD identifier and Preference. It can either be implemented directly as such or be implemented to make one Preference available for each interface and interface type available on the system.

The identification of a specific PvD is specific to the implementation or system. [RFC8801] defines how to use an FQDN to identify a PvD when advertised by a network, but systems might also use other locally relevant identifiers such as string names or Integers to identify PvDs. As with requiring specific interfaces, requiring a specific PvD strictly limits the path selection.

Categories or types of PvDs are also defined to be specific to the implementation or system. These can 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 (VoIP) service on a Cellular network, it can use the

relevant PvD type to require a 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.

6.2.13. 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 [RFC8981]. Temporary addresses are generally used to prevent linking connections over time when a stable address, sometimes called a "permanent" address, is not needed. There are some caveats to note when specifying this Property. First, if an application requires the use of temporary addresses, the resulting Connection cannot use IPv4 because temporary addresses do not exist in IPv4. Second, temporary local addresses might involve trading off privacy for performance. For instance, temporary addresses (e.g., [RFC8981]) can interfere with resumption mechanisms that some protocols rely on to reduce initial latency.

6.2.14. Multipath 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 as follows:

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 it.

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 `multipathPolicy` Property; see Section 8.1.7. To enable the peer Endpoint to initiate additional paths toward a local address other than the one initially used, it is necessary to set the `advertisesAltaddr` Property (see Section 6.2.15).

Setting this Property to `Active` can have privacy implications. It enables the transport to establish connectivity using alternate paths that might result in users being linkable across the multiple paths, even if the `advertisesAltaddr` Property (see Section 6.2.15) is set to `false`.

Note that this Property has no corresponding Selection Property of type "Preference". 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.

6.2.15. Advertisement of Alternative Addresses

Name: advertisesAltaddr

Type: Boolean

Default: `false`

This Property specifies whether alternative addresses, e.g., of other interfaces, ought to 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 can have privacy implications because it might result in users being linkable across the multiple paths. Also, note that setting this to `false` does not prevent the local Transport Services System from *establishing* connectivity using alternate paths (see Section 6.2.14); it only prevents *proactive advertisement* of addresses.

6.2.16. 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 as follows:

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 might cause a Protocol Stack that supports bidirectional communication to be selected.

6.2.17. Notification of ICMP Soft Error Message Arrival

Name: softErrorNotify

Type: Preference

Default: `No Preference`

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 are available as `SoftError` events; see Section 8.3.1. Note that even if a protocol supporting this Property is selected, not all ICMP errors will necessarily be delivered, so applications cannot rely upon receiving them [RFC8085].

6.2.18. Initiating Side Is Not the First to Write

Name: activeReadBeforeSend

Type: Preference

Default: `No Preference`

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 by either:

1. actively opening with `Initiate`, immediately followed by reading or
2. 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 Services System from mapping Connections to Stream Control Transmission Protocol (SCTP) streams, where the first transmitted data takes the role of an active open signal.

6.3. Specifying Security Parameters and Callbacks

Most Security Parameters, e.g., TLS ciphersuites, local identity and private key, etc., can be configured statically. Others are dynamically configured during Connection establishment. Security Parameters and callbacks are partitioned based on their place in the lifetime of Connection establishment. Similar to Transport Properties, both parameters and callbacks are inherited during cloning (see Section 7.4).

This document specifies an abstract API, which could appear to conflict with the need for Security Parameters to be unambiguous. The Transport Services System **SHOULD** provide reasonable, secure defaults for each enumerated Security Parameter, such that users of the system only need to specify parameters required to establish a secure connection (e.g., `serverCertificate` or `clientCertificate`). Specifying Security Parameters from enumerated values (e.g., specific ciphersuites) might constrain which transport protocols can be selected during Connection establishment.

Security Parameters are specified in the preestablishment phase and are created as follows:

```
SecurityParameters := NewSecurityParameters()
```

Specific parameters are added using a call to `Set` on the `SecurityParameters`.

As with the rest of the Transport Services API, the exact names of parameters and/or values of Enumerations (e.g., ciphersuites) used in the Security Parameters are specific to the system or implementation and ought to be chosen to follow the principle of least surprise for users of the platform/language environment in question.

For Security Parameters that are Enumerations of known values, such as TLS ciphersuites, implementations are responsible for exposing the set of values they support. For Security Parameters that are not simple value types, such as certificates and keys, implementations are responsible for exposing types appropriate for the platform/language environment.

Applications **SHOULD** use common safe defaults for values such as TLS ciphersuites whenever possible. However, as discussed in [RFC8922], many transport security protocols require specific Security Parameters and constraints from the client at the time of configuration and actively during a handshake.

The set of Security Parameters defined here is not exhaustive, but illustrative. Implementations **SHOULD** expose an equivalent to the parameters listed below to allow for sufficient configuration of Security Parameters, but the details are expected to vary based on platform and implementation constraints. Applications **MUST** be able to constrain the security protocols and versions that the Transport Services System will use.

Representation of Security Parameters in implementations ought to parallel that chosen for Transport Property names as suggested in Section 5.

Connections that use Transport Services **SHOULD** use security in general. However, for compatibility with endpoints that do not support transport security protocols (such as a TCP endpoint that does not support TLS), applications can initialize their Security Parameters to indicate that security can be disabled or opportunistic. If security is disabled, the Transport Services System will not attempt to add transport security automatically. If security is opportunistic, it will allow Connections without transport security, but it will still attempt to use unauthenticated security if available.

```
SecurityParameters := NewDisabledSecurityParameters()

SecurityParameters := NewOpportunisticSecurityParameters()
```

6.3.1. Allowed Security Protocols

Name: allowedSecurityProtocols

Type: Implementation-specific Enumeration of security protocol names and/or versions

Default: Implementation-specific best available security protocols

This Property allows applications to restrict which security protocols and security protocol versions can be used in the Protocol Stack. Applications **MUST** be able to constrain the security protocols used by this or an equivalent mechanism, in order to prevent the use of security protocols with unknown or weak security properties.

```
SecurityParameters.Set(allowedSecurityProtocols, [ tls_1_2, tls_1_3 ])
```

6.3.2. Certificate Bundles

Names: serverCertificate, clientCertificate

Type: Array of certificate objects

Default: Empty array

One or more certificate bundles identifying the Local Endpoint as a server certificate or a client certificate. Multiple bundles may be provided to allow selection among different Protocol Stacks that may require differently formatted bundles. The form and format of the certificate bundle are implementation specific. Note that if the private keys associated with a bundle are not available, e.g., since they are stored in Hardware Security Modules (HSMs), handshake callbacks are necessary. See below for details.

```
SecurityParameters.Set(serverCertificate, myCertificateBundle[])
SecurityParameters.Set(clientCertificate, myCertificateBundle[])
```

6.3.3. Pinned Server Certificate

Name: pinnedServerCertificate

Type: Array of certificate chain objects

Default: Empty array

Zero or more certificate chains to use as pinned server certificates, such that connecting will fail if the presented server certificate does not match one of the supplied pinned certificates. The form and format of the certificate chain are implementation specific.

```
SecurityParameters.Set(pinnedServerCertificate, yourCertificateChain[])
```

6.3.4. Application-Layer Protocol Negotiation

Name: alpn

Type: Array of strings

Default: Automatic selection

Application-Layer Protocol Negotiation (ALPN) values: used to indicate which application-layer protocols are negotiated by the security protocol layer. See [ALPN] for a definition of the ALPN field. Note that the Transport Services System can provide ALPN values automatically based on the protocols being used, if not explicitly specified by the application.

```
SecurityParameters.Set(alpn, ["h2"])
```

6.3.5. Groups, Ciphersuites, and Signature Algorithms

Names: supportedGroup, ciphersuite, signatureAlgorithm

Types: Arrays of implementation-specific Enumerations

Default: Automatic selection

These are used to restrict what cryptographic parameters are used by underlying transport security protocols. When not specified, these algorithms should use known and safe defaults for the system.

```
SecurityParameters.Set(supportedGroup, secp256r1)
SecurityParameters.Set(ciphersuite, TLS_AES_128_GCM_SHA256)
SecurityParameters.Set(signatureAlgorithm, ecdsa_secp256r1_sha256)
```

6.3.6. Session Cache Options

Names: maxCachedSessions, cachedSessionLifetimeSeconds

Type: Integer

Default: Automatic selection

These values are used to tune session cache capacity and lifetime and can be extended to include other policies.

```
SecurityParameters.Set(maxCachedSessions, 16)
SecurityParameters.Set(cachedSessionLifetimeSeconds, 3600)
```

6.3.7. Pre-Shared Key

Name: preSharedKey

Type: Key and identity (platform specific)

Default: `None`

Used to install pre-shared keying material established out of band. Each instance of pre-shared keying material is associated with some identity that typically identifies its use or has some protocol-specific meaning to the Remote Endpoint. Note that the use of a pre-shared key will tend to select a single security protocol and, therefore, directly select a single underlying Protocol Stack. A Transport Services API could express `None` in an environment-typical way, e.g., as a Union type or special value.

```
SecurityParameters.Set(preSharedKey, key, myIdentity)
```

6.3.8. Connection Establishment Callbacks

Security decisions, especially pertaining to trust, are not static. Once configured, parameters can also be supplied during Connection establishment. These are best handled as client-provided callbacks. Callbacks block the progress of the Connection establishment, which distinguishes them from other events in the Transport Services System. How callbacks and events are implemented is specific to each implementation. Security handshake callbacks that could be invoked during Connection establishment include:

- Trust verification callback: Invoked when a Remote Endpoint's trust must be verified before the handshake protocol can continue. For example, the application could verify an X.509 certificate as described in [RFC5280].

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

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

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

7. Establishing Connections

Before a Connection can be used for data transfer, it needs to be established. Establishment ends the preestablishment 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 can be active, using the `Initiate` action; passive, using the `Listen` action; or simultaneous for peer-to-peer connections, using the `Rendezvous` action. These actions are described in the subsections below.

7.1. 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 the Transport Services API 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 object to identify the Endpoint, optionally a Local Endpoint object (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) can cause one or more candidate transport-layer connections to be created to the specified Remote Endpoint. The caller could immediately begin sending Messages on the Connection (see Section 9.2) after calling `Initiate`; note that any data marked as "safely replayable" that is sent while the Connection is being established could be sent multiple times or using multiple candidates.

The following events can 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 9.3) will occur before the `Ready` event for Connections established using `Initiate`.

```
Connection -> EstablishmentError<reason?>
```

An `EstablishmentError` occurs 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,
- a Remote Endpoint Identifier cannot be resolved, or
- 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).

Connection establishment and transmission of the first Message can be combined in a single action (Section 9.2.5).

7.2. 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 the Transport Services API through the `Listen` action and returns a Listener object:

```
Listener := Preconnection.Listen()
```

Before calling `Listen`, the caller must have initialized the Preconnection during the preestablishment phase with a Local Endpoint object, as well as all Properties necessary for Protocol Stack selection. A Remote Endpoint can 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.

```
Listener.Stop()
```

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

```
Listener -> ConnectionReceived<Connection>
```

The `ConnectionReceived` event occurs when:

- a Remote Endpoint has established or cloned (e.g., by creating a new stream in a multi-stream transport; see Section 7.4) a transport-layer connection to this Listener (for connection-oriented transport protocols), or
- the first Message has been received from the Remote Endpoint (for connectionless protocols or streams of a multi-streaming transport) 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 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),
- the Local Endpoint (or Remote Endpoint, if specified) cannot be resolved, or
- the application is prohibited from listening by policy.

```
Listener -> Stopped<>
```

A `Stopped` event occurs after the Listener has stopped listening.

7.3. Peer-to-Peer Establishment: Rendezvous

Simultaneous peer-to-peer Connection establishment is supported by the `Rendezvous` action:

```
Preconnection.Rendezvous()
```

A Preconnection object used in a `Rendezvous` **MUST** have both the Local Endpoint candidates and the Remote Endpoint candidates specified, along with the Transport Properties and Security Parameters needed for Protocol Stack selection before the `Rendezvous` action is initiated.

The `Rendezvous` action listens on the Local Endpoint candidates for an incoming Connection from the Remote Endpoint candidates, while also simultaneously trying to establish a Connection from the Local Endpoint candidates to the Remote Endpoint candidates.

If there are multiple Local Endpoints or Remote Endpoints configured, then initiating a `Rendezvous` action will cause the Transport Services Implementation to systematically probe the reachability of those endpoint candidates following an approach such as that used in Interactive Connectivity Establishment (ICE) [RFC8445].

If the endpoints are suspected to be behind a NAT, and the Local Endpoint supports a method of discovering NAT bindings, such as STUN [RFC8489] or Traversal Using Relays around NAT (TURN) [RFC8656], then the `Resolve` action on the Preconnection can be used to discover such bindings:

```
[]LocalEndpoint, []RemoteEndpoint := Preconnection.Resolve()
```

The `Resolve` action returns lists of Local Endpoints and Remote Endpoints that represent the concrete addresses, local and server reflexive, on which a `Rendezvous` for the Preconnection will listen for incoming Connections and to which it will attempt to establish Connections.

Note that the set of Local Endpoints returned by `Resolve` might or might not contain information about all possible local interfaces, depending on how the Preconnection is configured. The set of available local interfaces can also change over time, so care needs to be taken when using stored interface names.

An application that uses `Rendezvous` to establish a peer-to-peer Connection in the presence of NATs will configure the Preconnection object with at least one Local Endpoint that supports NAT binding discovery. It will then `Resolve` the Preconnection and pass the resulting list of Local Endpoint candidates to the peer via a signaling protocol, for example, as part of an ICE exchange [RFC8445] within SIP [RFC3261] or WebRTC [RFC7478]. The peer will then, via the same signaling channel, return the Remote Endpoint candidates. The set of Remote Endpoint candidates is then configured on the Preconnection:

```
Preconnection.AddRemote([]RemoteEndpoint)
```

Once the application has added both the Local Endpoint candidates and the Remote Endpoint candidates retrieved from the peer via the signaling channel to the Preconnection, the `Rendezvous` action is initiated and causes the Transport Services Implementation to begin connectivity checks.

If successful, the `Rendezvous` action returns a Connection object via a `RendezvousDone` event:

```
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. Changes made to a Preconnection after `Rendezvous` has been called **MUST NOT** have any effect on existing Connections.

An `EstablishmentError` occurs 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,
- the Local Endpoint or Remote Endpoint cannot be resolved,
- no transport-layer connection can be established to the Remote Endpoint, or
- the application is prohibited from rendezvous by policy.

```
Preconnection -> EstablishmentError<reason?>
```

7.4. Connection Groups

Connection Groups can be created using the `Clone` action:

```
Connection := Connection.Clone(framer?, connectionProperties?)
```

Calling `Clone` on a Connection yields a Connection Group containing two Connections: the parent Connection on which `Clone` was called and a resulting cloned Connection. The new Connection is actively opened, and it will locally send a `Ready` event or an `EstablishmentError` event. Calling `Clone` on any of these Connections adds another Connection to the Connection Group. Connections in a Connection Group share all Connection Properties except `connPriority` (see Section 8.1.2), and these Connection Properties are entangled: changing one of the Connection Properties on one Connection in the Connection Group automatically changes the Connection Property for all others. For example, changing `connTimeout` (see Section 8.1.3) on one Connection in a Connection Group will automatically make the same change to this Connection Property for

all other Connections in the Connection Group. Like all other Properties, `connPriority` is copied to the new Connection when calling `Clone`, but, in this case, a later change to the `connPriority` on one Connection does not change it on the other Connections in the same Connection Group.

The optional `connectionProperties` parameter allows passing Transport Properties that control the behavior of the underlying stream or connection to be created, e.g., Protocol-specific Properties to request specific stream IDs for SCTP or QUIC.

Message Properties set on a Connection also apply only to that Connection.

A new Connection created by `Clone` can have a Message Framer assigned via the optional `framer` parameter of the `Clone` action. If this parameter is not supplied, the stack of Message Framers associated with a Connection is copied to the cloned Connection when calling `Clone`. Then, a cloned Connection has the same stack of Message Framers as the Connection from which they are cloned, but these Framers can internally maintain per-Connection state.

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 belong to 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.

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

Note that calling `Clone` can result in on-the-wire signaling, e.g., to open a new transport connection, depending on the underlying Protocol Stack. When `Clone` leads to the opening of multiple such connections, the Transport Services System will ensure consistency of Connection Properties by uniformly applying them to all underlying connections in a group. Even in such a case, it is possible for a Transport Services System to implement prioritization within a Connection Group (see [TCP-COUPLING] and [RFC8699]).

Attempts to clone a Connection can result in a `CloneError`:

```
Connection -> CloneError<reason?>
```

A `CloneError` can also occur later, after `Clone` was successfully called. In this case, it informs the application that the Connection that sends the `CloneError` is no longer a part of any Connection Group. For example, this can occur when the Transport Services system is unable to implement

entanglement (a Connection Property was changed on a different Connection in the Connection Group, but this change could not be successfully applied to the Connection that sends the `CloneError`).

The `connPriority` Connection Property operates on Connections in a Connection Group using the same approach as that used in Section 9.1.3.2: when allocating available network capacity among Connections in a Connection Group, sends on Connections with numerically lower priority values will be prioritized over sends on Connections that have numerically higher priority values. Capacity will be shared among these Connections according to the `connScheduler` Property (Section 8.1.5). See Section 9.2.6 for more details.

7.5. Adding and Removing Endpoints on a Connection

Transport protocols that are explicitly multipath-aware are expected to automatically manage the set of remote endpoints that they are communicating with and the paths to those endpoints. A `PathChange` event, described in Section 8.3.2, will be generated when the path changes.

However, in some cases, it is necessary to explicitly indicate to a Connection that a new Remote Endpoint has become available for use or indicate that a Remote Endpoint is no longer available. This is most common in the case of peer-to-peer connections using Trickle ICE [RFC8838].

The `AddRemote` action can be used to add one or more new Remote Endpoints to a Connection:

```
Connection.AddRemote([]RemoteEndpoint)
```

Endpoints that are already known to the Connection are ignored. A call to `AddRemote` makes the new Remote Endpoints available to the Connection, but whether the Connection makes use of those Endpoints will depend on the underlying transport protocol.

Similarly, the `RemoveRemote` action can be used to tell a Connection to stop using one or more Remote Endpoints:

```
Connection.RemoveRemote([]RemoteEndpoint)
```

Removing all known Remote Endpoints can have the effect of aborting the connection. The effect of removing the active Remote Endpoint(s) depends on the underlying transport: multipath-aware transports might be able to switch to a new path if other reachable Remote Endpoints exist or the connection might abort.

Similarly, the `AddLocal` and `RemoveLocal` actions can be used to add and remove Local Endpoints to or from a Connection.

8. Managing Connections

During preestablishment and after establishment, Preconnections or Connections can be configured and queried using Connection Properties, and asynchronous information could be available about the state of the Connection via `SoftError` events.

Connection Properties represent the configuration and state of the selected Protocol Stack(s) backing a Connection. These Connection Properties can 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 8.1.

Protocol-specific Properties are defined in a way that is specific to the transport or implementation to permit more specialized protocol features to be used. Too much reliance by an application on Protocol-specific Properties can significantly reduce the flexibility of a Transport Services System to make appropriate selection and configuration choices. Therefore, it is **RECOMMENDED** that Generic Connection Properties be used for properties common across different protocols and that Protocol-specific Connection Properties are only used where specific protocols or properties are necessary.

The application can set and query Connection Properties on a per-Connection basis. Connection Properties that are not read-only can be set during preestablishment (see Section 6.2), as well as on Connections directly using the `SetProperty` action:

```
ErrorCode := Connection.SetProperty(property, value)
```

If an error is encountered in setting a Property (for example, if the application tries to set a TCP-specific Property on a Connection that is not using TCP), the application **MUST** be informed about this error via the `ErrorCode` object. Such errors **MUST NOT** cause the Connection to be terminated. 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 Section 7.4.

At any point, the application can query Connection Properties.

```
ConnectionProperties := Connection.GetProperties()
value := ConnectionProperties.Get(property)
if ConnectionProperties.Has(boolean_or_preference_property) then...
```

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

- The Connection state, which can be one of the following: `Establishing`, `Established`, `Closing`, or `Closed` (see Section 8.1.11.1).

- Whether the Connection can be used to send data (see Section 8.1.11.2). A Connection cannot be used for sending if the Connection was created with the Selection Property `direction` set to `Unidirectional receive` or if a Message marked as `Final` was sent over this Connection. See also Section 9.1.3.5.
- Whether the Connection can be used to receive data (see Section 8.1.11.3). A Connection cannot be used for receiving if the Connection was created with the Selection Property `direction` set to `Unidirectional send` or if a Message marked as `Final` was received (see Section 9.3.3.3). The latter is only supported by certain transport protocols, e.g., by TCP as a half-closed connection.
- For Connections that are `Established`, `Closing`, or `Closed`: Connection Properties (Section 8.1) of the actual protocols that were selected and instantiated, and Selection Properties that the application specified on the Preconnection. Selection Properties of type "Preference" will be exposed as Boolean values indicating whether or not the Property applies to the selected transport. Note that the instantiated Protocol Stack might not match all Protocol Selection Properties that the application specified on the Preconnection.
- For Connections that are `Established`: Transport Services Implementations ought to provide information concerning the path(s) used by the Protocol Stack. This can be derived from local PvD information, measurements by the Protocol Stack, or other sources. For example, a Transport Services System that is configured to receive and process PvD information [RFC7556] could also provide network configuration information for the chosen path(s).

8.1. Generic Connection Properties

Generic Connection Properties are defined independently of the chosen Protocol Stack; therefore, they are available on all Connections.

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

8.1.1. Required Minimum Corruption Protection Coverage for Receiving

Name: recvChecksumLen

Type: Integer (non-negative) or `Full Coverage`

Default: `Full Coverage`

If this Property is an Integer, it specifies the minimum number of bytes in a received Message that need to be covered by a checksum. A receiving Endpoint will not forward Messages that have less coverage to the application. The application is responsible for handling any corruption within the non-protected part of the Message [RFC8085]. A special value of 0 means that a received packet might also have a zero checksum field, and the enumerated value `Full Coverage` means that the entire Message needs to be protected by a checksum. An implementation is supposed to express `Full Coverage` in an environment-typical way, e.g., as a Union type or special value.

8.1.2. Connection Priority

Name: connPriority

Type: Integer (non-negative)

Default: 100

This Property is a non-negative Integer representing the priority of this Connection relative to other Connections in the same Connection Group. A numerically lower value reflects a higher priority. It has no effect on Connections not part of a Connection Group. As noted in Section 7.4, 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. No guarantees of a specific behavior regarding Connection Priority are given; a Transport Services System could ignore this Property. See Section 9.2.6 for more details.

8.1.3. Timeout for Aborting Connection

Name: connTimeout

Type: Numeric (positive) or `Disabled`

Default: `Disabled`

If this Property is Numeric, it specifies how long to wait before deciding that an active Connection has failed when trying to reliably deliver data to the Remote Endpoint. Adjustments to this Property will only take effect if the underlying stack supports reliability. If this Property has the enumerated value `Disabled`, it means that no timeout is scheduled. A Transport Services API could express `Disabled` in an environment-typical way, e.g., as a Union type or special value.

8.1.4. Timeout for Keep-Alive Packets

Name: keepAliveTimeout

Type: Numeric (positive) or `Disabled`

Default: `Disabled`

A Transport Services API can request a protocol that supports sending keep-alive packets (Section 6.2.10). If this Property is Numeric, it specifies the maximum length of time an idle Connection (one for which no transport packets have been sent) ought to wait before the Local Endpoint sends a keep-alive packet to the Remote Endpoint. Adjustments to this Property will only take effect if the underlying stack supports sending keep-alive packets. Guidance on setting this value for connectionless transports is provided in [RFC8085]. A value greater than the Connection timeout (Section 8.1.3) or the enumerated value `Disabled` will disable the sending of keep-alive packets. A Transport Services API could express `Disabled` in an environment-typical way, e.g., as a Union type or special value.

8.1.5. Connection Group Transmission Scheduler

Name: connScheduler

Type: Enumeration

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

This Property specifies which scheduler is used among Connections within a Connection Group to apportion the available capacity according to Connection priorities (see Sections 7.4 and 8.1.2). A set of schedulers is described in [RFC8260].

8.1.6. Capacity Profile

Name: connCapacityProfile

Type: Enumeration

Default: Default Profile (Best Effort)

This Property specifies the desired network treatment for traffic sent by the application and the trade-offs 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 Services System **SHOULD** select paths and configure protocols to optimize the trade-off 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 markings on the wire for Protocol Stacks supporting this action. Recommendations for use with DSCPs 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 Services Systems that map the requested capacity profile to per-connection DSCP signaling **SHOULD** assign the DSCP Default Forwarding Per Hop Behavior (PHB) [RFC2474].

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 Services Systems that map the requested capacity profile to per-connection DSCP signaling **SHOULD** assign the DSCP "Less than best effort" PHB [RFC8622].

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. The `Low Latency/Interactive` value of the capacity profile can be used by the system to disable the coalescing of multiple small

Messages into larger packets (Nagle algorithm (see Section 4.2.3.4 of [RFC1122])); to prefer immediate acknowledgement from the peer Endpoint when supported by the underlying transport; and so on. Transport Services Systems that map the requested capacity profile to per-connection DSCP signaling without multiplexing **SHOULD** assign a DSCP Assured Forwarding (AF41, AF42, AF43, and AF44) PHB [RFC2597]. Inelastic traffic that is expected to conform to the configured network service rate could be mapped to the DSCP Expedited Forwarding PHBs [RFC3246] or PHBs as discussed in [RFC5865].

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 to per-connection DSCP signaling without multiplexing **SHOULD** assign a DSCP Assured Forwarding (AF21, AF22, AF23, and AF24) PHB [RFC2597].

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 might fail if the path is unable to maintain the desired rate. A transport can 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 to per-connection DSCP signaling without multiplexing **SHOULD** assign a DSCP Assured Forwarding (AF31, AF32, AF33, and AF34) PHB [RFC2597].

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 Services Systems that map the requested capacity profile to per-connection DSCP signaling without multiplexing **SHOULD** assign a DSCP Assured Forwarding (AF11, AF12, AF13, and AF14) PHB [RFC2597] per Section 4.8 of [RFC4594].

The capacity profile for a selected Protocol Stack may be modified on a per-Message basis using the `msgCapacityProfile` Message Property; see Section 9.1.3.8.

8.1.7. Policy for Using Multipath Transports

Name: multipathPolicy

Type: Enumeration

Default: Handover

This Property specifies the local policy for transferring data across multiple paths between the same end hosts if the `multipath` Property is not set to `Disabled` (see Section 6.2.14). Possible values are as follows:

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

Interactive: The Connection ought only to attempt to minimize the latency for interactive traffic patterns by transmitting data across multiple paths when this is beneficial. The goal of minimizing the latency will be balanced against the cost of each of these paths. 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 a lower latency. The specific scheduling algorithm is implementation specific.

Aggregate: The Connection ought to attempt to use multiple paths in parallel to maximize available capacity and possibly overcome the capacity limitations of the individual paths. The actual strategy is implementation specific.

Note that this is a local choice: the Remote Endpoint can choose a different policy.

8.1.8. Bounds on Send or Receive Rate

Name: minSendRate / minRecvRate / maxSendRate / maxRecvRate

Type: Numeric (positive) or `Unlimited` / Numeric (positive) or `Unlimited` / Numeric (positive) or `Unlimited` / Numeric (positive) or `Unlimited`

Default: `Unlimited` / `Unlimited` / `Unlimited` / `Unlimited`

Numeric values of these Properties specify 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 application-layer rate below which the application does not deem it will be useful. These rate values are measured at the application layer, i.e., do not consider the header overhead from protocols used by the Transport Services System. The values are specified in bits per second and assumed to be measured over one-second time intervals. For example, specifying a maxSendRate of X bits per second means that, from the moment at which the Property value is chosen, not more than X bits will be sent in any following second. The enumerated value `Unlimited` indicates that no bound is specified. A Transport Services API could express `Unlimited` in an environment-typical way, e.g., as a Union type or special value.

8.1.9. Group Connection Limit

Name: groupConnLimit

Type: Numeric (positive) or `Unlimited`

Default: `Unlimited`

If this Property is Numeric, it controls the number of Connections that can be accepted from a peer as new members of the Connection's group. Similar to `SetNewConnectionLimit`, this limits the number of `ConnectionReceived` events that will occur, but constrained to the group of the Connection associated with this Property. For a multi-streaming transport, this limits the number of allowed streams. A Transport Services API could express `Unlimited` in an environment-typical way, e.g., as a Union type or special value.

8.1.10. Isolate Session

Name: isolateSession

Type: Boolean

Default: `false`

When set to `true`, this Property will initiate new Connections using as little cached information (such as session tickets or cookies) as possible from previous Connections that are not in the same Connection Group. Any state generated by this Connection will only be shared with Connections in the same Connection Group. Cloned Connections will use saved state from within the Connection Group. This is used for separating Connection Contexts as specified in Section 4.2.3 of [RFC9621].

Note that this does not guarantee that information will not leak because implementations might not be able to fully isolate all caches (e.g., RTT estimates). Note that this Property could degrade Connection performance.

8.1.11. Read-Only Connection Properties

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

8.1.11.1. Connection State

Name: connState

Type: Enumeration

This Property provides information about the current state of the Connection. Possible values are `Establishing`, `Established`, `Closing`, or `Closed`. For more details on Connection state, see Section 11.

8.1.11.2. Can Send Data

Name: canSend

Type: Boolean

This Property can be queried to learn whether the Connection can be used to send data.

8.1.11.3. Can Receive Data

Name: canReceive

Type: Boolean

This Property can be queried to learn whether the Connection can be used to receive data.

8.1.11.4. Maximum Message Size Before Fragmentation or Segmentation

Name: singularTransmissionMsgMaxLen

Type: Integer (non-negative) or `Not applicable`

This Property, if applicable, represents the maximum Message size that can be sent without incurring network-layer fragmentation at the sender. It is specified as a number of bytes and is less than or equal to the maximum Message Size on `Send`. It exposes a readable value to the application based on the Maximum Packet Size (MPS). The value of this Property can change over time (and can be updated via Datagram Packetization Layer Path MTU Discovery (DPLPMTUD) [RFC8899]). This value allows a sending stack to avoid unwanted fragmentation at the network layer or segmentation by the transport layer before choosing the Message size and/or after a `SendError` occurs indicating an attempt to send a Message that is too large. A Transport Services API could express `Not applicable` in an environment-typical way, e.g., as a Union type or special value (e.g., 0).

8.1.11.5. Maximum Message Size on Send

Name: sendMsgMaxLen

Type: Integer (non-negative)

This Property represents the maximum Message size that an application can send. It is specified as the number of bytes. A value of 0 indicates that sending is not possible.

8.1.11.6. Maximum Message Size on Receive

Name: recvMsgMaxLen

Type: Integer (non-negative)

This Property represents the maximum Message size that an application can receive. It is specified as the number of bytes. A value of 0 indicates that receiving is not possible.

8.2. TCP-Specific Properties: User Timeout Option (UTO)

These Properties specify configurations for the TCP User Timeout Option (UTO). This is a TCP-specific Property that is only used in the case that TCP becomes the chosen transport protocol. It is useful only if TCP is implemented in the Transport Services System. Protocol-specific options could also be defined for other transport protocols.

These Properties are included here because the feature `Suggest timeout to the peer` is part of the minimal set of Transport Services [RFC8923], where this feature was categorized as "functional". This means that when a Transport Services System offers this feature, the Transport Services API has to expose an interface 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 `tcp.userTimeoutValue` as `true` but not specify a `tcp.userTimeoutValue` value; in this case, the TCP default will be used). These Properties reflect the API extension specified in Section 3 of [RFC5482].

8.2.1. Advertised User Timeout

Name: tcp.userTimeoutValue

Type: Integer (positive)

Default: the TCP default

This time value is advertised via the TCP User Timeout Option (UTO) [RFC5482] to the Remote Endpoint, which can use it to adapt its own `connTimeout` (see Section 8.1.3) value.

8.2.2. User Timeout Enabled

Name: tcp.userTimeoutEnabled

Type: Boolean

Default: `false`

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

8.2.3. Timeout Changeable

Name: tcp.userTimeoutChangeable

Type: Boolean

Default: `true`

This Property controls whether the TCP `connTimeout` (see Section 8.1.3) can be changed based on a UTO received from the remote peer. This Boolean becomes `false` when `connTimeout` (see Section 8.1.3) is used.

8.3. Connection Lifecycle Events

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

8.3.1. 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<>
```

8.3.2. Path Change

This event notifies the application when at least one of the paths underlying a Connection has changed. Changes occur on a single path when the PMTU changes as well as when multiple paths are used and paths are added or removed, the set of Local Endpoints changes, or a handover has been performed.

```
Connection -> PathChange<>
```

9. Data Transfer

Data is sent and received as Messages, which allows the application to communicate the boundaries of the data being transferred.

9.1. Messages and Framers

Each Message has an optional MessageContext, which allows adding Message Properties, to identify `Send` events related to a specific Message or to inspect metadata related to the Message sent. 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.

9.1.1. Message Contexts

Using the MessageContext object, the application can set and retrieve metadata of the Message, including Message Properties (see Section 9.1.3) and framing metadata (see Section 9.1.2.2). Therefore, a MessageContext object can be passed to the `Send` action and is returned by each event related to `Send` and `Receive`.

Message Properties can be set and queried using the MessageContext:

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

These Message Properties can be generic Properties or Protocol-specific Properties.

For MessageContexts returned by `Send` events (see Section 9.2.2) and `Receive` events (see Section 9.3.2), the application can query information about the Local and Remote Endpoint:

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

9.1.2. 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 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 is designed based on the fact that many of the application protocols in use at the time of writing evolved over TCP, which does not provide Message boundary preservation; because 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 [RFC9329] for an example of such a framing protocol).

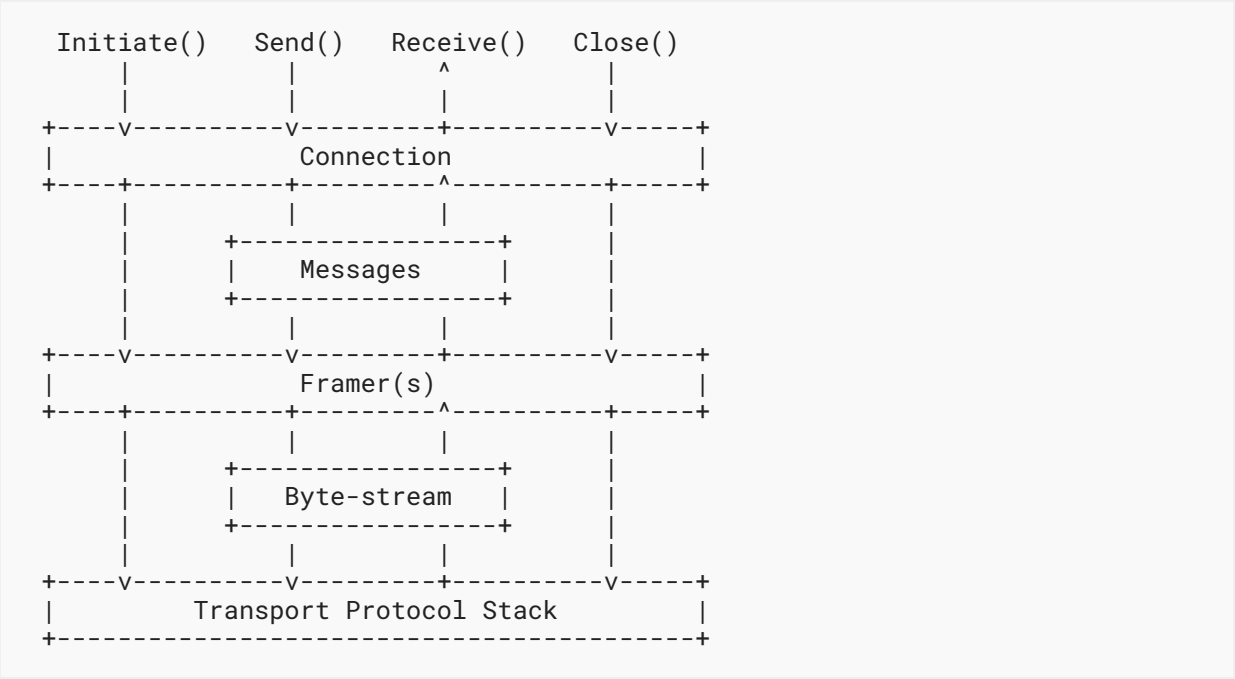

Figure 1: Protocol Stack Showing a Message Framer

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 a 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 [RFC9623].

9.1.2.1. Adding Message Framers to Preconnections

The Message Framer object can be added to one or more Preconnections to run on top of transport protocols. Multiple Framers can be added to a Preconnection; in this case, the Framers operate as a framing stack, i.e., 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)
```

Since Message Framers pass from Preconnection to Listener or Connection, addition of Framers must happen before any operation that might result in the creation of a Connection.

9.1.2.2. Framing Metadata

When sending Messages, applications can add Framer-specific Properties to a MessageContext (Section 9.1.1) with the `add` action. To avoid naming conflicts, the Property names **SHOULD** be prefixed with a Namespace referencing the Framer implementation or the protocol it implements as described in Section 4.1.

This mechanism 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")
```

9.1.3. Message Properties

Applications needing to annotate the Messages they send with extra information (for example, to control how data is scheduled and processed by the transport protocols supporting the Connection) can include this information in the MessageContext passed to the `Send` action. For other uses of the MessageContext, see Section 9.1.1.

Message Properties are per-Message, not per-`Send`, if partial Messages are sent (Section 9.2.3). All data blocks associated with a single Message share Properties specified in the MessageContexts. For example, it would not make sense to have the beginning of a Message expire and then allow the end of the Message to still be sent.

A MessageContext object contains metadata for the 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.

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 can be added to a MessageContext object only before the context is used for sending. Once a MessageContext has been used with a `Send` action, further modifications to the MessageContext object do not have any effect on this `Send` action. Message Properties that are not added to a MessageContext object before using the context for sending will either take a specific default value or be configured based on Selection or Connection Properties of the Connection that is associated with the `Send` action. This initialization behavior is defined per Message Property below.

The Message Properties could be inconsistent with the properties of the Protocol Stacks underlying the Connection on which a given Message is sent. For example, a Protocol Stack must be able to provide ordering if the `msgOrdered` Property of a Message is enabled. Sending a Message with Message Properties inconsistent with the Selection Properties of the Connection yields an error.

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 `reliability` is set to `Require` and a protocol with configurable per-Message reliability is used, setting `msgReliable` to `false` for a particular Message will allow this Message to be sent without any reliability guarantees. Changing the `msgReliable` Message Property is only possible for Connections that were established enabling the Selection Property `perMsgReliability`. If the contradicting Message Property cannot be supported by the Connection (such as requiring reliability on a Connection that uses an unreliable protocol), the `Send` action will result in a `SendError` event.

The Message Properties in the following subsections are supported.

9.1.3.1. Lifetime

Name: msgLifetime

Type: Numeric (positive)

Default: Infinite

The Lifetime specifies how long a particular Message can wait in the Transport Services System before it is sent to the Remote Endpoint. After this time, it is irrelevant and no longer needs to be (re-)transmitted. This is a hint to the Transport Services 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 `perMsgReliability` Property (see Section 9.1.3.7). The type and units of Lifetime are implementation specific.

9.1.3.2. Priority

Name: msgPriority

Type: Integer (non-negative)

Default: 100

This Property specifies the priority of a Message, relative to other Messages sent over the same Connection. A numerically lower value represents a higher priority.

A Message with priority 2 will yield to a Message with priority 1, which will yield to a Message with priority 0, and so on. Priorities can 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 `connPriority`; see Section 8.1.2. The priority Properties might interact, but they can be used independently and be realized by different mechanisms; see Section 9.2.6.

9.1.3.3. Ordered

Name: msgOrdered

Type: Boolean

Default: the queried Boolean value of the Selection Property `preserveOrder` (Section 6.2.4)

The order in which Messages were submitted for transmission via the `Send` action will be preserved on delivery via `Receive` events for all Messages on a Connection that have this Message Property set to `true`.

If `false`, the Message is delivered to the receiving application without preserving the ordering. This Property is used for protocols that support preservation of data ordering (see Section 6.2.4) but allow out-of-order delivery for certain Messages, e.g., by multiplexing independent Messages onto different streams.

If it is not configured by the application before sending, this Property's default value will be based on the Selection Property `preserveOrder` of the Connection associated with the `Send` action.

9.1.3.4. Safely Replayable

Name: safelyReplayable

Type: Boolean

Default: `false`

If `true`, `safelyReplayable` specifies that a Message is safe to send to the Remote Endpoint more than once for a single `Send` action. It marks the data as safe for certain 0-RTT establishment techniques, where retransmission of the 0-RTT data could cause the remote application to receive the Message multiple times.

For protocols that do not protect against duplicated Messages, e.g., UDP, all Messages need to be marked as "safely replayable" by enabling this Property. To enable protocol selection to choose such a protocol, `safelyReplayable` needs to be added to the TransportProperties passed to the Preconnection. If such a protocol was chosen, disabling `safelyReplayable` on individual Messages **MUST** result in a `SendError`.

9.1.3.5. Final

Name: final

Type: Boolean

Default: `false`

If `true`, this indicates a Message is the last 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 signaling the end of a Connection in a given direction will ignore this Property.

A `Final` Message must always be sorted to the end of a list of Messages. The `final` Property overrides `connPriority`, `msgPriority`, and any other Property that would reorder 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.

9.1.3.6. Sending Corruption Protection Length

Name: msgChecksumLen

Type: Integer (non-negative) or `Full Coverage`

Default: `Full Coverage`

If this Property is an Integer, it specifies the minimum length of the section of a sent 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 needs to be calculated, and the enumerated value `Full`

`Coverage` means that the entire Message needs to be protected by a checksum. Only `Full Coverage` is guaranteed: any other requests are advisory, which may result in `Full Coverage` being applied.

9.1.3.7. Reliable Data Transfer (Message)

Name: msgReliable

Type: Boolean

Default: the queried Boolean value of the Selection Property `reliability` (Section 6.2.1)

When `true`, this Property specifies that a Message should be sent in such a way that the transport protocol ensures that all data is received by the Remote Endpoint. Changing the `msgReliable` Property on Messages is only possible for Connections that were established enabling the Selection Property `perMsgReliability`. When this is not the case, changing `msgReliable` will generate an error.

Disabling this Property indicates that the Transport Services System could disable retransmissions or other reliability mechanisms for this particular Message, but such disabling is not guaranteed.

If it is not configured by the application before sending, this Property's default value will be based on the Selection Property `reliability` of the Connection associated with the `Send` action.

9.1.3.8. Message Capacity Profile Override

Name: msgCapacityProfile

Type: Enumeration

Default: inherited from the Connection Property `connCapacityProfile` (Section 8.1.6)

This enumerated Property specifies the application's preferred trade-offs for sending this Message; it is a per-Message override of the `connCapacityProfile` Connection Property (see Section 8.1.6). If it is not configured by the application before sending, this Property's default value will be based on the Connection Property `connCapacityProfile` of the Connection associated with the `Send` action.

9.1.3.9. No Network-Layer Fragmentation

Name: noFragmentation

Type: Boolean

Default: `false`

This Property specifies that a Message should be sent and received without network-layer fragmentation, if possible. It can be used to avoid network-layer fragmentation when transport segmentation is preferred.

This only takes effect when the transport uses a network layer that supports this functionality. When it does take effect, setting this Property to `true` will cause the sender to avoid network-layer source fragmentation. When using IPv4, this will result in the Don't Fragment (DF) bit being set in the IP header.

Attempts to send a Message with this Property that result in a size greater than the transport's current estimate of its maximum packet size (`singularTransmissionMsgMaxLen`) can result in transport segmentation when permitted or in a `SendError`.

> Note: `noSegmentation` is used when it is desired to send a Message within a single network packet.

9.1.3.10. No Segmentation

Name: noSegmentation

Type: Boolean

Default: `false`

When set to `true`, this Property requests that the transport layer not provide segmentation of Messages larger than the maximum size permitted by the network layer and that it avoid network-layer source fragmentation of Messages. When running over IPv4, setting this Property to `true` will result in a sending Endpoint setting the Don't Fragment bit in the IPv4 header of packets generated by the transport layer.

An attempt to send a Message that results in a size greater than the transport's current estimate of its maximum packet size (`singularTransmissionMsgMaxLen`) will result in a `SendError`. This only takes effect when the transport and network layers support this functionality.

9.2. Sending Data

Once a Connection has been established, it can be used for sending Messages. By default, `Send` enqueues a complete Message and takes optional per-Message Properties (see Section 9.2.1). All `Send` actions are asynchronous and deliver events (see Section 9.2.2). Sending partial Messages for streaming large data is also supported (see Section 9.2.3).

Messages are sent on a Connection using the `Send` action:

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

where `messageData` is the data object to send and `messageContext` allows adding Message Properties, identifying `Send` events related to a specific Message or inspecting metadata related to the Message sent (see Section 9.1.1).

The optional `endOfMessage` parameter supports partial sending and is described in Section 9.2.3.

9.2.1. 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 could be the payload of a single datagram for a UDP connection. Another example would be an HTTP Request for an HTTP Connection.

Some transport protocols can deliver arbitrarily sized Messages, but other protocols constrain the maximum Message size. Applications can query the Connection Property `sendMsgMaxLen` (Section 8.1.11.5) 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 9.2.2.3). For example, it is invalid to send a Message over a UDP connection that is larger than the available datagram sending size.

9.2.2. Send Events

Like all actions in the Transport Services API, 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 `Send` calls are used (Section 9.2.3), 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 Transport Services API should allow the application to correlate a `Send` event to the particular call to `Send` that triggered the event. The manner in which this correlation is indicated is implementation specific.

9.2.2.1. 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 the Transport Services API. 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 Context of 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 Services System than an application that always waits for the `Sent` event before calling the next `Send` action.

9.2.2.2. 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 9.1.3.1) expired. This is separate from `SendError`, as it is an expected behavior for partially reliable transports. The `Expired` event contains a reference to the MessageContext of the Message to which it applies.

9.2.2.3. SendError

```
Connection -> SendError<messageContext, reason?>
```

A `SendError` occurs when a Message was not sent due to an error condition: an attempt to send a Message that 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 MessageContext of the Message to which it applies.

9.2.3. 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 might be too large for the application to hold in memory at one time or the length of the Message might 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.

9.2.4. Batching Sends

To reduce the overhead of sending multiple small Messages on a Connection, the application could batch several `Send` actions together. This provides a hint to the system that the sending of these Messages ought to be coalesced when possible and that sending any of the batched Messages can 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()
```

9.2.5. 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" using the `safelyReplayable` Property. This allows the Transport Services System to make use of 0-RTT establishment in case this is supported by the available Protocol Stacks. When the selected stack or stacks 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 are 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 `EstablishmentError` signaling the failure of Connection establishment.

9.2.6. Priority and the Transport Services API

The Transport Services API provides two Properties to allow a sender to signal the relative priority of data transmission: `msgPriority` (see Section 9.1.3.2) and `connPriority` (see Section 8.1.2). These Properties are designed to allow the expression and implementation of a wide variety of approaches to transmission priority in the transport and application layers, including those that do not appear on the wire (affecting only sender-side transmission scheduling) as well as those that do (e.g., [RFC9218]). A Transport Services System gives no guarantees about how its expression of relative priorities will be realized.

The Transport Services API does order `connPriority` over `msgPriority`. In the absence of other externalities (e.g., transport-layer flow control), a priority 1 Message on a priority 0 Connection will be sent before a priority 0 Message on a priority 1 Connection in the same group.

9.3. Receiving Data

Once a Connection is established, it can be used for receiving data (unless the `direction` Property is set to `unidirectional send`). As with sending, the data is received in 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 9.3.2). If Messages arrive at the Transport Services System before `Receive` requests are issued, ensuing `Receive` requests will first operate on these Messages before awaiting any further Messages.

9.3.1. 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 9.3.2.1).

The application can set a `minIncompleteLength` value to indicate the smallest partial Message data size in bytes to be delivered in response to this `Receive`. By default, this value is Infinite, which means that only complete Messages should be delivered. See Sections 9.3.2.2 and 9.1.2 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:

1. when at least that many bytes are available,
2. the Message is complete with fewer bytes, or

3. 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 that 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 9.3.2.2).

Note that `maxLength` does not guarantee that the application will receive that many bytes if they are available; the Transport Services API could 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.

9.3.2. Receive Events

Each call to `Receive` will be paired with a single `Receive` event. This allows an application to provide backpressure to the Protocol Stack when it is temporarily not ready to receive Messages. For example, an application that will later be able to handle multiple `Receive` events at the same time can make multiple calls to `Receive` without waiting for, or processing, any `Receive` events. An application that is temporarily unable to process received events for a connection could refrain from calling `Receive` or could delay calling it. This would lead to a buildup of unread data, which, in turn, could result in backpressure to the sender via a transport protocol's flow control.

The Transport Services API should allow the application to correlate a `Receive` event to the particular call to `Receive` that triggered the event. The manner in which this correlation is indicated is implementation specific.

9.3.2.1. 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` value provides access to the bytes that were received for this Message, along with the length of the byte array. The `messageContext` value is provided to enable retrieving metadata about the Message and referring to the Message. The MessageContext object is described in Section 9.1.1.

See Section 9.1.2 regarding how to handle Message framing in situations where the Protocol Stack only provides a byte-stream transport.

9.3.2.2. ReceivedPartial

```
Connection -> ReceivedPartial<messageData, messageContext,
                              endOfMessage>
```

If a complete Message cannot be delivered in one event, one part of the Message can be delivered with a `ReceivedPartial` event. 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 value of the `endOfMessage` Property 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 non-contiguous partial Messages. For example, if Message A is divided into three pieces (A1, A2, and A3), Message B is divided into three pieces (B1, B2, and B3), and `preserveOrder` is not `Require`, the `ReceivedPartial` could deliver them in a sequence like this: A1, B1, B2, A2, A3, B3. This is because the MessageContext allows the application to identify the pieces as belonging to Message A and B, respectively. However, a sequence like A1, A3 will never occur.

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

- the underlying Protocol Stack supports Message boundary preservation and the size of the Message is larger than the buffers available for a single Message;
- the underlying Protocol Stack does not support Message boundary preservation and the Message Framer (see Section 9.1.2) cannot determine the end of the Message using the buffer space it has available; or
- the underlying Protocol Stack does not support Message boundary preservation and no 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.

In the following example, an application only wants to receive up to 1000 bytes at a time from a Connection. If a 1500-byte Message arrives, it would receive the Message in two separate `ReceivedPartial` events.

```
Connection.Receive(1, 1000)

// Receive the first 1000 bytes; Message is incomplete
Connection -> ReceivedPartial<messageData(1000 bytes),
                              messageContext, false>

Connection.Receive(1, 1000)

// Receive the last 500 bytes; Message is now complete
Connection -> ReceivedPartial<messageData(500 bytes),
                              messageContext, true>
```

9.3.2.3. ReceiveError

```
Connection -> ReceiveError<messageContext, reason?>
```

A `ReceiveError` occurs when:

- data is received by the underlying Protocol Stack that cannot be fully retrieved or parsed, and
- it is useful for the application to be notified of such errors.

For example, a `ReceiveError` can indicate that a Message (identified via the `messageContext` value) that was being partially received previously, but had not completed, encountered an error and will not be completed. This can be useful for an application, which might wish to use this error as a hint to remove previously received Message parts from memory. As another example, if an incoming Message does not fulfill the `recvChecksumLen` Property (see Section 8.1.1), an application can use this error as a hint to inform the peer application to adjust the `msgChecksumLen` Property (see Section 9.1.3.6).

In contrast, internal protocol reception errors (e.g., loss causing retransmissions in TCP) are not signaled by this event. Conditions that irrevocably lead to the termination of the Connection are signaled using `ConnectionError` (see Section 10).

9.3.3. Receive Message Properties

Each MessageContext could contain metadata from protocols in the Protocol Stack; which metadata is available is Protocol Stack dependent. These are exposed through additional read-only Message Properties that can be queried from the MessageContext object (see Section 9.1.1) passed by the `Receive` event. The metadata values in the following subsections are supported.

9.3.3.1. Property Specific to UDP and UDP-Lite: ECN

When available, Message metadata carries the value of the Explicit Congestion Notification (ECN) field. This information can be used for logging and debugging as well as building applications that 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; hence, they expose this functionality to the application (see [RFC8293], following the guidance in [RFC8085]).

9.3.3.2. Early Data

In some cases, it can 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 might wish to perform additional checks for early data to ensure that 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.

9.3.3.3. Receiving Final Messages

The MessageContext can indicate whether or not this Message is the last 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 can be marked on a sent Message; see Section 9.1.3.5.

Some transport protocols and peers do not support signaling of the `final` Property. Therefore, applications **SHOULD NOT** rely on receiving a Message marked `Final` to know that the sending Endpoint is done sending on a Connection.

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

10. Connection Termination

A Connection can be terminated:

1. by the Local Endpoint (i.e., the application calls the `Close`, `CloseGroup`, `Abort`, or `AbortGroup` action),
2. by the Remote Endpoint (i.e., the remote application calls the `Close`, `CloseGroup`, `Abort`, or `AbortGroup` action), or
3. because of an error (e.g., a timeout).

A local call of the `Close` action will cause the Connection to send either a `Closed` event or a `ConnectionError` event; a local call of the `CloseGroup` action will cause all of the Connections in the group to send either a `Closed` event or a `ConnectionError` event. A local call of the `Abort` action will cause the Connection to send a `ConnectionError` event, indicating local `Abort` as a `reason`; a local call of the `AbortGroup` action will cause all of the Connections in the group to send a `ConnectionError` event, indicating local `Abort` as a `reason`.

Remote action calls map to events similar to local calls (e.g., a remote `Close` causes the Connection to send either a `Closed` event or a `ConnectionError` event), but in contrast to local action calls, it is not guaranteed that such events will indeed be invoked. When an application needs to free resources associated with a Connection, it ought not rely on the invocation of such events due to termination calls from the Remote Endpoint; instead, it should use the local termination actions.

`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 Services System. Upon successfully satisfying all these requirements, the Connection will send the `Closed` event. For example, if reliable delivery was requested for a Message handed over before calling `Close`, the `Closed` event will signify that this Message has indeed been delivered. This action does not affect any other Connection in the same Connection Group.

An application **MUST NOT** assume that it can receive any further data on a Connection for which it has called `Close`, even if such data is already in flight.

```
Connection.Close()
```

The `Closed` event informs the application that a `Close` action has successfully completed or that the Remote Endpoint has closed the Connection. There is no guarantee that a remote `Close` will be signaled.

```
Connection -> Closed<>
```

`Abort` terminates a Connection without delivering any remaining Messages. This action does not affect any other Connection that is entangled with this one in a Connection Group. When the `Abort` action has finished, the Connection will send a `ConnectionError` event, indicating local `Abort` as a `reason`.

```
Connection.Abort()
```

`CloseGroup` gracefully terminates a Connection and any other Connections in the same Connection Group. For example, all of the Connections in a group might be streams of a single session for a multistreaming protocol; closing the entire group will close the underlying session. See also Section 7.4. All Connections in the group will send a `Closed` event when the `CloseGroup` action was successful. As with `Close`, any Messages remaining to be processed on a Connection will be handled prior to closing.

```
Connection.CloseGroup()
```

`AbortGroup` terminates a Connection and any other Connections that are in the same Connection Group without delivering any remaining Messages. When the `AbortGroup` action has finished, all Connections in the group will send a `ConnectionError` event, indicating local `Abort` as a `reason`.

```
Connection.AbortGroup()
```

A `ConnectionError` informs the application that:

1. data could not be delivered to the peer after a timeout or
2. the Connection has been aborted (e.g., because the peer has called `Abort`).

There is no guarantee that an `Abort` from the peer will be signaled.

```
Connection -> ConnectionError<reason?>
```

11. Connection State and Ordering of Operations and Events

This Transport Services API is designed to be independent of an implementation's concurrency model. The exact details regarding how actions are handled, and how events are dispatched, are implementation dependent.

Some transitions of Connection states are associated with events:

- A `Ready` event occurs when a Connection created with `Initiate` or `InitiateWithSend` transitions to `Established` state.
- A `ConnectionReceived` event occurs when a Connection created with `Listen` transitions to `Established` state.
- A `RendezvousDone` event occurs when a Connection created with `Rendezvous` transitions to `Established` state.
- A `Closed` event occurs when a Connection transitions to `Closed` state without error.
- An `EstablishmentError` event occurs when a Connection created with `Initiate` transitions from `Establishing` state to `Closed` state due to an error.
- A `ConnectionError` event 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.

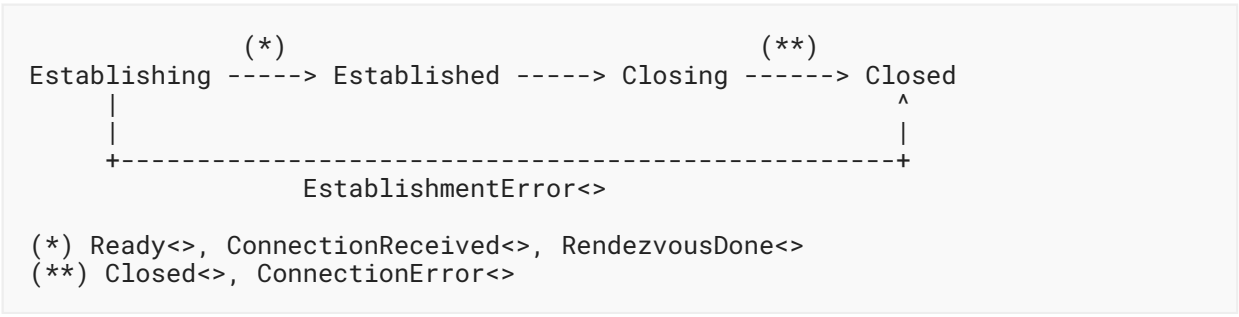

Figure 2: Connection State Diagram

The Transport Services API provides the following guarantees about the ordering of operations:

- `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 the implementation).
- `Received` events will never occur on a Connection before it is `Established`, i.e., before a `Ready` event on that Connection or a `ConnectionReceived` or `RendezvousDone` event containing that Connection.
- No events will occur on a Connection after it is closed, i.e., after a `Closed` event, an `EstablishmentError` or `ConnectionError` event will not occur on that Connection. To ensure this ordering, a `Closed` event will not occur on a Connection while other events on the Connection are still locally outstanding (i.e., known to the Transport Services API and waiting to be dealt with by the application).

12. IANA Considerations

This document has no IANA actions.

Future works might create IANA registries for generic Transport Property names and Transport Property Namespaces (see Section 4.1).

13. Privacy and Security Considerations

This document describes a generic API for interacting with a Transport Services System. Part of this API includes configuration details for transport security protocols, as discussed in Section 6.3. It does not recommend use (or disuse) of specific algorithms or protocols. Any API-compatible transport security protocol ought to work in a Transport Services System. Security considerations for these protocols are discussed in the respective specifications.

[RFC9621] provides general security considerations and requirements for any system that implements the Transport Services Architecture. These include recommendations of relevance to the API, e.g., regarding the use of keying material.

The described API is used to exchange information between an application and the Transport Services System. The same authority implementing both systems is not necessarily expected. However, there is an expectation that the Transport Services Implementation would either:

- be provided as a library that is selected by the application from a trusted party or
- be part of the operating system that the application also relies on for other tasks.

In either case, the Transport Services API is an internal interface that is used to exchange information locally between two systems. However, as the Transport Services 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 Transport Services API is useful to configure and select protocols and paths and is not necessarily privacy sensitive. Still, some information could be privacy sensitive because it might reveal usage characteristics and habits of the user of an application.

Of course, any communication over a network reveals usage characteristics, because 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 entities can access it. How Transport Services Systems ought to choose protocols -- depending on the security Properties required -- is out of scope for this specification, as it is limited to transport protocols. The choice of a security protocol can be informed by the survey provided in [RFC8922].

In most cases, information provided for protocol and path selection does not directly translate to information that can be observed by network devices on the path. However, there might be specific configuration information that is intended for path exposure, e.g., a Diffserv codepoint setting that is either provided directly by the application or indirectly configured for a traffic profile.

Applications should be aware that a single communication attempt can lead to more than one connection establishment procedure. For example, this is the case when:

- the Transport Services System also executes name resolution,
- support mechanisms such as TURN or ICE are used to establish connectivity if protocols or paths are raced or if a path fails and fallback or re-establishment is supported in the Transport Services System.

Applications should take special care when using 0-RTT session resumption (see Section 6.2.5), as early data sent across multiple paths during Connection establishment could reveal information that can be used to correlate Endpoints on these paths.

Applications should also take care to not assume that all data received using the Transport Services API is always complete or well-formed. Specifically, Messages that are received partially (see Section 9.3.2.2) could be a source of truncation attacks if applications do not distinguish between partial Messages and complete Messages.

The Transport Services API explicitly does not require the application to resolve names, though there is a trade-off between early and late binding of addresses to names. Early binding allows the Transport Services Implementation to reduce Connection setup latency. This is 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) that does not protect query confidentiality. Names used with the Transport Services API **SHOULD** be FQDNs; not providing an FQDN will result in the Transport Services Implementation needing to use DNS search domains for name resolution, which might lead to inconsistent or unpredictable behavior.

These communication activities are not different from what is used at the time of writing. However, the goal of a Transport Services 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, although it reduces transparency of the underlying communication actions to the application itself. The Transport Services API is designed such that protocol and path selection can be limited to a small and controlled set if required by the application to perform a function or to provide security. Further, introspection on the Properties of Connection objects allows an application to determine which protocol(s) and path(s) are in use. A Transport Services System **SHOULD** provide a facility logging the communication events of each Connection.

14. References

14.1. Normative References

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

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

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

[RFC9621] Pauly, T., Ed., Trammell, B., Ed., Brunstrom, A., Fairhurst, G., and C. S. Perkins, "Architecture and Requirements for Transport Services", RFC 9621, DOI 10.17487/RFC9621, January 2025, <https://www.rfc-editor.org/info/RFC9621>.

14.2. Informative References

[RFC1122] Braden, R., Ed., "Requirements for Internet Hosts - Communication Layers", STD 3, RFC 1122, DOI 10.17487/RFC1122, October 1989, <https://www.rfc-editor.org/info/rfc1122>.

[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.C.R., Benson, K., Le Boudec, J.Y., 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>.

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

[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>.

[RFC5280] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, DOI 10.17487/RFC5280, May 2008, <https://www.rfc-editor.org/info/rfc5280>.

[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>.

[RFC791] Postel, J., "Internet Protocol", STD 5, RFC 791, DOI 10.17487/RFC0791, September 1981, <https://www.rfc-editor.org/info/rfc791>.

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

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

[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>.

[RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <https://www.rfc-editor.org/info/rfc8126>.

[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>.

[RFC8293] Ghanwani, A., Dunbar, L., McBride, M., Bannai, V., and R. Krishnan, "A Framework for Multicast in Network Virtualization over Layer 3", RFC 8293, DOI 10.17487/RFC8293, January 2018, <https://www.rfc-editor.org/info/rfc8293>.

[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>.

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

[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>.

[RFC8489] Petit-Huguenin, M., Salgueiro, G., Rosenberg, J., Wing, D., Mahy, R., and P. Matthews, "Session Traversal Utilities for NAT (STUN)", RFC 8489, DOI 10.17487/RFC8489, February 2020, <https://www.rfc-editor.org/info/rfc8489>.

[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>.

[RFC8656] Reddy, T., Ed., Johnston, A., Ed., Matthews, P., and J. Rosenberg, "Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN)", RFC 8656, DOI 10.17487/RFC8656, February 2020, <https://www.rfc-editor.org/info/rfc8656>.

[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>.

[RFC8801] Pfister, P., Vyncke, É., Pauly, T., Schinazi, D., and W. Shao, "Discovering Provisioning Domain Names and Data", RFC 8801, DOI 10.17487/RFC8801, July 2020, <https://www.rfc-editor.org/info/rfc8801>.

[RFC8838] Ivov, E., Uberti, J., and P. Saint-Andre, "Trickle ICE: Incremental Provisioning of Candidates for the Interactive Connectivity Establishment (ICE) Protocol", RFC 8838, DOI 10.17487/RFC8838, January 2021, <https://www.rfc-editor.org/info/rfc8838>.

[RFC8899] Fairhurst, G., Jones, T., Tüxen, M., Rüngeler, I., and T. Völker, "Packetization Layer Path MTU Discovery for Datagram Transports", RFC 8899, DOI 10.17487/RFC8899, September 2020, <https://www.rfc-editor.org/info/rfc8899>.

[RFC8922] Enghardt, T., Pauly, T., Perkins, C., Rose, K., and C. Wood, "A Survey of the Interaction between Security Protocols and Transport Services", RFC 8922, DOI 10.17487/RFC8922, October 2020, <https://www.rfc-editor.org/info/rfc8922>.

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

[RFC8981] Gont, F., Krishnan, S., Narten, T., and R. Draves, "Temporary Address Extensions for Stateless Address Autoconfiguration in IPv6", RFC 8981, DOI 10.17487/RFC8981, February 2021, <https://www.rfc-editor.org/info/rfc8981>.

[RFC9218] Oku, K. and L. Pardue, "Extensible Prioritization Scheme for HTTP", RFC 9218, DOI 10.17487/RFC9218, June 2022, <https://www.rfc-editor.org/info/rfc9218>.

[RFC9329] Pauly, T. and V. Smyslov, "TCP Encapsulation of Internet Key Exchange Protocol (IKE) and IPsec Packets", RFC 9329, DOI 10.17487/RFC9329, November 2022, <https://www.rfc-editor.org/info/rfc9329>.

[RFC9623] Brunstrom, A., Ed., Pauly, T., Ed., Enghardt, R., Tiesel, P. S., and M. Welzl, "Implementing Interfaces to Transport Services", RFC 9623, DOI 10.17487/RFC9623, January 2025, <https://www.rfc-editor.org/info/rfc9623>.

[TCP-COUPLING] Islam, S., Welzl, M., Hiorth, K., Hayes, D., Armitage, G., and S. Gjessing, "ctrlTCP: Reducing latency through coupled, heterogeneous multi-flow TCP congestion control", IEEE INFOCOM 2018 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), DOI 10.1109/INFCOMW.2018.8406887, 2018, <https://ieeexplore.ieee.org/document/8406887>.

Appendix A. Implementation Mapping

The way the concepts from this abstract API map to concrete APIs in a given language on a given platform largely depends on the features and norms of the language and the platform. Actions could be implemented as either functions or method calls. For instance, actions could be implemented via event queues, handler functions or classes, communicating sequential processes, or other asynchronous calling conventions.

A.1. Types

The basic types mentioned in Section 1.1 typically have natural correspondences in practical programming languages, perhaps constrained by implementation-specific limitations. For example:

- Typically, an Integer can be represented in C by an `int` or `long`; this is subject to the underlying platform's ranges for each.
- In C, a Tuple may be represented as a `struct` with one member for each of the value types in the ordered grouping. However, in Python, a Tuple may be represented as a `tuple`, which is a sequence of dynamically typed elements.
- A Set may be represented as a `std::set` in C++ or as a `set` in Python. In C, it may be represented as an array or as a higher-level data structure with appropriate accessors defined.

The objects described in Section 1.1 can also be represented in different ways, depending on which programming language is used. Objects like Preconnections, Connections, and Listeners can be long-lived and benefit from using object-oriented constructs. Note that, in C, these objects may need to provide a way to release or free their underlying memory when the application is done using them. For example, since a Preconnection can be used to initiate multiple Connections, it is the responsibility of the application to clean up the Preconnection memory if necessary.

A.2. Events and Errors

This specification treats events and errors similarly. Errors, just as any other events, may occur asynchronously in network applications. However, implementations of this API may report errors synchronously. This is done according to the error-handling idioms of the implementation platform, where they can be immediately detected. An example of this is to generate an exception when attempting to initiate a Connection with inconsistent Transport Properties. An error can provide an optional `reason` to the application with further details about why the error occurred.

A.3. Time Duration

Time duration types are implementation specific. For instance, it could be a number of seconds, a number of milliseconds, or a `struct timeval` in C; in C++, it could be a user-defined `Duration` class.

Appendix B. Convenience Functions

B.1. Adding Preference Properties

TransportProperties will frequently need to set Selection Properties of type "Preference"; therefore, implementations can provide special actions for adding each preference level, i.e., `TransportProperties.Set(some_property, avoid)` is equivalent to `TransportProperties.Avoid(some_property)`:

```
TransportProperties.Require(property)
TransportProperties.Prefer(property)
TransportProperties.NoPreference(property)
TransportProperties.Avoid(property)
TransportProperties.Prohibit(property)
```

B.2. Transport Property Profiles

To ease the use of the Transport Services API, implementations can provide a mechanism to create Transport Property objects (see Section 6.2) that are preconfigured with frequently used sets of Properties; the following subsections list those that are in common use in applications at the time of writing.

B.2.1. reliable-inorder-stream

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

Property	Value
reliability	Require
preserveOrder	Require
congestionControl	Require
preserveMsgBoundaries	No Preference

Table 2: reliable-inorder-stream Preferences

B.2.2. reliable-message

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

Property	Value
reliability	Require
preserveOrder	Require
congestionControl	Require
preserveMsgBoundaries	Require

Table 3: reliable-message Preferences

B.2.3. unreliable-datagram

This profile provides a datagram transport service without any reliability guarantee. An example of a protocol that provides this service is UDP. It consists of the following Properties:

Property	Value
reliability	Avoid
preserveOrder	Avoid
congestionControl	No Preference
preserveMsgBoundaries	Require
safelyReplayable	true

Table 4: unreliable-datagram Preferences

Applications that choose this Transport Property Profile would avoid the additional latency that could be introduced by retransmission or reordering in a transport protocol.

Applications that choose this Transport Property Profile to reduce latency should also consider setting an appropriate capacity profile Property (see Section 8.1.6) and might benefit from controlling checksum coverage (see Sections 6.2.7 and 6.2.8).

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

[RFC8923] identifies a minimal set of Transport Services that end systems should offer. These services make all non-security-related transport features of TCP, Multipath TCP (MPTCP), UDP, UDP-Lite, SCTP, and Low Extra Delay Background Transport (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 [RFC8923] and updated according to the discussion in Section 5 of [RFC8923]. The present API covers all elements of this section. This list is a subset of the transport features in Appendix A of [RFC8923], which refers to the primitives in "pass 2". See Section 4 of [RFC8303] for 1) further details on the implementation with TCP, MPTCP, UDP, UDP-Lite, SCTP, and LEDBAT and 2) how to facilitate finding the specifications for implementing the services listed below with these protocols.

- Connect: `Initiate` action (Section 7.1).
- Listen: `Listen` action (Section 7.2).
- Specify number of attempts and/or timeout for the first establishment Message: `timeout` parameter of `Initiate` (Section 7.1) or `InitiateWithSend` action (Section 9.2.5).
- Disable MPTCP: `multipath` Property (Section 6.2.14).
- Hand over a Message to reliably transfer (possibly multiple times) before Connection establishment: `InitiateWithSend` action (Section 9.2.5).
- Change timeout for aborting connection (using retransmit limit or time value): `connTimeout` Property, using a time value (Section 8.1.3).
- Timeout event when data could not be delivered for too long: `ConnectionError` event (Section 10).
- Suggest timeout to the peer: See "TCP-Specific Properties: User Timeout Option (UTO)" (Section 8.2).
- Notification of ICMP error message arrival: `softErrorNotify` (Section 6.2.17) and `SoftError` event (Section 8.3.1).
- Choose a scheduler to operate between streams of an association: `connScheduler` Property (Section 8.1.5).
- Configure priority or weight for a scheduler: `connPriority` Property (Section 8.1.2).
- "Specify checksum coverage used by the sender" and "Disable checksum when sending": `msgChecksumLen` Property (Section 9.1.3.6) and `fullChecksumSend` Property (Section 6.2.7).
- "Specify minimum checksum coverage required by receiver" and "Disable checksum requirement when receiving": `recvChecksumLen` Property (Section 8.1.1) and `fullChecksumRecv` Property (Section 6.2.8).
- Specify DF field: `noFragmentation` Property (Section 9.1.3.9).

- Get maximum transport-message size that may be sent using a non-fragmented IP packet from the configured interface: `singularTransmissionMsgMaxLen` Property (Section 8.1.11.4).
- Get maximum transport-message size that may be received from the configured interface: `recvMsgMaxLen` Property (Section 8.1.11.6).
- Obtain ECN field: This is a read-only Message Property of the MessageContext object (see "Property Specific to UDP and UDP-Lite: ECN" (Section 9.3.3.1)).
- "Specify DSCP field", "Disable Nagle algorithm", and "Enable and configure a `Low Extra Delay Background Transfer`": as suggested in Section 5.5 of [RFC8923], these transport features are collectively offered via the `connCapacityProfile` Property (Section 8.1.6). Per-Message control ("Request not to bundle messages") is offered via the `msgCapacityProfile` Property (Section 9.1.3.8).
- 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 [RFC8923] (see also Section 10).
- "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": these are offered by the `Abort` action without promising that these are signaled to the other side. If they are, a `ConnectionError` event will be invoked at the peer (Section 10).
- "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 9.2). Reliability is controlled via the `reliability` (Section 6.2.1) Property and the `msgReliable` Message Property (Section 9.1.3.7). Transmitting data as a Message or without delimiters is controlled via Message Framers (Section 9.1.2). The choice of congestion control is provided via the `congestionControl` Property (Section 6.2.9).
- Configurable Message Reliability: the `msgLifetime` Message Property implements a time-based way to configure Message reliability (Section 9.1.3.1).
- "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 `msgOrdered` (Section 9.1.3.3).
- Request not to delay the acknowledgement (SACK) of a message: should the protocol support it, this is one of the transport features the Transport Services System can apply when an application uses the `connCapacityProfile` Property (Section 8.1.6) or the `msgCapacityProfile` Message Property (Section 9.1.3.8) with value `Low Latency/ Interactive`.
- Receive data (with no message delimiting): `Receive` action (Section 9.3.1) and `Received` event (Section 9.3.2.1).
- Receive a message: `Receive` action (Section 9.3.1) and `Received` event (Section 9.3.2.1) using Message Framers (Section 9.1.2).
- Information about partial message arrival: `Receive` action (Section 9.3.1) and `ReceivedPartial` event (Section 9.3.2.2).
- Notification of send failures: `Expired` event (Section 9.2.2.2) and `SendError` event (Section 9.2.2.3).

- Notification that the stack has no more user data to send: applications can obtain this information via the `Sent` event (Section 9.2.2.1).
- Notification to a receiver that a partial message delivery has been aborted: `ReceiveError` event (Section 9.3.2.3).
- Notification of Excessive Retransmissions (early warning below abortion threshold): `SoftError` event (Section 8.3.1).

Acknowledgements

This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreements No. 644334 (NEAT) and No. 688421 (MAMI).

This work has been supported by:

- Leibniz Prize project funds from the DFG - German Research Foundation: Gottfried Wilhelm Leibniz-Preis 2011 (FKZ FE 570/4-1).
- the UK Engineering and Physical Sciences Research Council under grant EP/R04144X/1.
- the Research Council of Norway under its "Toppforsk" programme through the "OCARINA" project.

Thanks to Stuart Cheshire, Josh Graessley, David Schinazi, and Eric Kinnear for their implementation and design efforts, including Happy Eyeballs, that heavily influenced this work. Thanks to Laurent Chuat and Jason Lee for initial work on the Post Sockets interface, from which this work has evolved. Thanks to Maximilian Franke for asking good questions based on implementation experience and for contributing text, e.g., on multicast.

Authors' Addresses

Brian Trammell (EDITOR)
Google Switzerland GmbH
Gustav-Gull-Platz 1
CH-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

Reese 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
United Kingdom
Email: gorry@erg.abdn.ac.uk
URI: https://erg.abdn.ac.uk/

Mirja Kühlewind
Ericsson
Ericsson-Allee 1
Herzogenrath
Germany
Email: mirja.kuehlewind@ericsson.com

Colin S. Perkins
University of Glasgow
School of Computing Science
Glasgow
G12 8QQ
United Kingdom
Email: csp@csperkins.org

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

Tommy Pauly
Apple Inc.
One Apple Park Way
Cupertino, CA 95014
United States of America
Email: tpauly@apple.com