TAPS Working Group
Internet-Draft
Intended status: Standards Track
Expires: 10 September 2023

B. Trammell, Ed.
Google Switzerland GmbH
M. Welzl, Ed.
University of Oslo
T. Enghardt
Netflix
G. Fairhurst
University of Aberdeen
M. Kuehlewind
Ericsson
C. Perkins
University of Glasgow
P. Tiesel
SAP SE
T. Pauly
Apple Inc.
9 March 2023

An Abstract Application Layer Interface to Transport Services
draft-ietf-taps-interface-19

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 sockets API as the common interface to the transport layer, in an environment where endpoints could select from multiple interfaces and potential transport protocols.

Status of This Memo

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

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

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

This Internet-Draft will expire on 10 September 2023.

Copyright Notice

Copyright (c) 2023 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (https://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

Table of Contents

1. Introduction 4
1.1. Terminology and Notation 5
1.2. Specification of Requirements 7
2. Overview of the API Design 7
3. API Summary 8
3.1. Usage Examples 9
3.1.1. Server Example 9
3.1.2. Client Example 10
3.1.3. Peer Example 12
4. Transport Properties 13
4.1. Transport Property Names 14
4.2. Transport Property Types 15
5. Scope of the API Definition 15
6. Pre-Establishment Phase 16
6.1. Specifying Endpoints 17
6.1.1. Using Multicast Endpoints 19
6.1.2. Constraining Interfaces for Endpoints 20
6.1.3. Endpoint Aliases 21
6.1.4. Endpoint Examples 21
6.1.5. Multicast Examples 22
6.2. Specifying Transport Properties 25
6.2.1. Reliable Data Transfer (Connection) 28
6.2.2. Preservation of Message Boundaries 28
6.2.3. Configure Per-Message Reliability 28
6.2.4. Preservation of Data Ordering 28

6.2.5. Use 0-RTT Session Establishment with a Safely Replayable Message 29
6.2.6. Multistream Connections in Group 29
6.2.7. Full Checksum Coverage on Sending 29
6.2.8. Full Checksum Coverage on Receiving 29
6.2.9. Congestion control 30
6.2.10. Keep alive 30
6.2.11. Interface Instance or Type 30
6.2.12. Provisioning Domain Instance or Type 31
6.2.13. Use Temporary Local Address 32
6.2.14. Multipath Transport 33
6.2.15. Advertisement of Alternative Addresses 34
6.2.16. Direction of communication 34
6.2.17. Notification of ICMP soft error message arrival . . . 35
6.2.18. Initiating side is not the first to write 35
6.3. Specifying Security Parameters and Callbacks 36
6.3.1. Specifying Security Parameters on a Pre-Connection . 36
6.3.2. Connection Establishment Callbacks 38
7. Establishing Connections 38
7.1. Active Open: Initiate 39
7.2. Passive Open: Listen 40
7.3. Peer-to-Peer Establishment: Rendezvous 41
7.4. Connection Groups 43
7.5. Adding and Removing Endpoints on a Connection 45
8. Managing Connections 45
8.1. Generic Connection Properties 47
8.1.1. Required Minimum Corruption Protection Coverage for Receiving 47
8.1.2. Connection Priority 48
8.1.3. Timeout for Aborting Connection 48
8.1.4. Timeout for keep alive packets 48
8.1.5. Connection Group Transmission Scheduler 49
8.1.6. Capacity Profile 49
8.1.7. Policy for using Multipath Transports 51
8.1.8. Bounds on Send or Receive Rate 52
8.1.9. Group Connection Limit 52
8.1.10. Isolate Session 52
8.1.11. Read-only Connection Properties 53
8.2. TCP-specific Properties: User Timeout Option (UTO) . . . 54
8.2.1. Advertised User Timeout 54
8.2.2. User Timeout Enabled 54
8.2.3. Timeout Changeable 55
8.3. Connection Lifecycle Events 55
8.3.1. Soft Errors 55
8.3.2. Path change 55
9. Data Transfer 55
9.1. Messages and Framers 56
9.1.1. Message Contexts 56

9.1.2. Message Framers 56
9.1.3. Message Properties 59
9.2. Sending Data 65
9.2.1. Basic Sending 65
9.2.2. Send Events 66
9.2.3. Partial Sends 67
9.2.4. Batching Sends 68
9.2.5. Send on Active Open: InitiateWithSend 68
9.2.6. Priority and the Transport Services API 69
9.3. Receiving Data 69
9.3.1. Enqueuing Receives 70
9.3.2. Receive Events 70
9.3.3. Receive Message Properties 73
10. Connection Termination 74
11. Connection State and Ordering of Operations and Events . . . 76
12. IANA Considerations 77
13. Privacy and Security Considerations 77
14. Acknowledgements 79
15. References 79
15.1. Normative References 79
15.2. Informative References 80
Appendix A. Implementation Mapping 84
A.1. Types 84
A.2. Events and Errors 84
A.3. Time Duration 85
Appendix B. Convenience Functions 85
B.1. Adding Preference Properties 85
B.2. Transport Property Profiles 85
B.2.1. reliable-inorder-stream 85
B.2.2. reliable-message 86
B.2.3. unreliable-datagram 86
Appendix C. Relationship to the Minimal Set of Transport Services for End Systems 87
Authors' Addresses 90

1. Introduction

This document specifies an abstract application programming interface (API) that specifies the interface component of the high-level Transport Services architecture defined in [I-D.ietf-taps-arch]. 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 interfaces 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.

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 transport 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 aynchronously; 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 associated with no 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 platform- and implementation-specific, and may 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 positive or negative integer values.

* Numeric: Instances take positive or negative real number values.

* 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 may be of fixed or variable length.

* Collection: An unordered grouping of one or more values of the same type.

For guidance on how these abstract concepts may be implemented in languages in accordance with native design patterns and language 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 [I-D.ietf-taps-arch]. 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 to 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 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 provide these.

* 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, should that particular transport be 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 bi-directional 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 connection-less or connection-oriented. Connections can be created from Preconnections in three ways:

* by initiating the Preconnection (i.e., actively opening, as in a client; Section 7.1),

* through listening on the Preconnection (i.e., passively opening, as in a server Section 7.2),

* or rendezvousing on the Preconnection (i.e., peer to peer establishment; 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 both via explicit Protocol Stack support, and via 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 may 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.

* Act as a client, by connecting to a Remote Endpoint using Initiate, sending requests, and receiving responses, see Section 3.1.2.

* Act as a peer, by connecting to a Remote Endpoint using Rendezvous while simultaneously waiting for incoming Connections, sending Messages, and receiving Messages, see Section 3.1.3.

The examples in this section presume that a transport protocol is available between the Local and Remote Endpoints that provides Reliable Data Transfer, Preservation of Data Ordering, and Preservation of Message Boundaries. In this case, the application can choose to receive only complete Messages.

If none of the available transport protocols provides Preservation of Message Boundaries, but there is a transport protocol that provides a reliable ordered byte stream, an application 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, and receive a request, and send a response.

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

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

SecurityParameters := NewSecurityParameters()
SecurityParameters.Set(identity, myIdentity)
SecurityParameters.Set(key-pair, myPrivateKey, myPublicKey)

// 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, and send a request as well as receive a response on each of them.

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

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

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

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

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

Preconnections are reusable after being used to initiate a Connection. Hence, for example, after the Connections were closed, the following would be correct:

```
//.. carry out adjustments to the Preconnection, if desire
Connection := 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 STUN server
HostCandidate := NewLocalEndpoint()
HostCandidate.WithPort(9876)

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

LocalCandidates = [HostCandidate, StunCandidate]

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

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

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

// ...Send the ResolvedLocal list to peer via signalling channel
// ...Receive a list of RemoteCandidates from peer via
//    signalling channel

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 signalling channel, for example if using Trickle ICE, then add
// them to the Connection:
Connection.AddRemote(NewRemoteCandidates)

// On a PathChange<> events, resolve the local endpoints 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:
  // ...Send the ResolvedLocal list to peer via signalling 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 should operate. This is done by using Transport Properties, as defined in [I-D.ietf-taps-arch], 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 pre-establishment. They are only used to specify which paths and protocol stacks can be used and are preferred by the application. Although Connection Properties (see Section 8.1) can be set during pre-establishment, they may be changed later. They are used to inform decisions made during establishment and to fine-tune the established connection. Calling Initiate on a Preconnection creates an outbound Connection or a Listener, and the Selection Properties remain readable from the Connection or Listener, but become immutable.

The behavior of the selected protocol stack(s) when sending Messages is controlled by Message Properties (see Section 9.1.3).

Selection Properties can be set on Preconnections, and the effect of Selection Properties can be queried on Connections and Messages. 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 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 [I-D.ietf-taps-arch], are not used as an input to the selection process, but only support configuration if the respective protocol has been selected.

4.1. Transport Property Names

Transport Properties are referred to by property names. For the purposes of this document, these names are alphanumeric strings in which words may be separated by hyphens. Specifically, the following characters are allowed: lowercase letters a-z, uppercase letters A-Z, digits 0-9, the hyphen -, and the underscore _. 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 as a representation of properties retrieved from a file or other storage.

* Making the code of different Transport Services implementations look similar. While individual programming languages may preclude strict adherence to the aforementioned naming convention (for instance, by prohibiting the use of hyphens in symbols), users interacting with multiple implementations will still benefit from the consistency resulting from the use of visually similar symbols.

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

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

* Protocol-specific Properties MUST use the protocol acronym as the Namespace (e.g., a tcp Connection could support a TCP-specific Transport Property, such as the user timeout value, in a Protocol-specific Property called tcp.userTimeoutValue (see Section 8.2).

* Vendor or implementation specific properties MUST use a string identifying the vendor or implementation as the Namespace.

* For IETF protocols, the name of a Protocol-specific Property SHOULD be specified in an IETF document published in the RFC Series.

Namespaces for each of the keywords provided in the IANA protocol numbers registry (see https://www.iana.org/assignments/protocol-numbers/protocol-numbers.xhtml) are reserved for Protocol-specific Properties and MUST NOT be used for vendor or implementation-specific properties. Avoid using any of the terms listed as keywords in the protocol numbers registry as any part of a vendor- or implementation-specific property name.

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 use the Preference Enumeration, which takes one of five possible values (Prohibit, Avoid, Ignore, 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.

We therefore make the following recommendations:

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

* Transport Services systems SHOULD implement each Selection Property, Connection Property, and Message Context Property specified in this document. The Transport Services API SHOULD be implemented even when in a specific implementation/platform 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 states that "it is not guaranteed that a Message will not be sent when its Lifetime has expired".

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

6. Pre-Establishment Phase

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

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 all the Local Endpoints on the Preconnection MUST represent the same host. For example, they might correspond to different interfaces on a multi-homed host, of they 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, then all the Remote Endpoints on the Preconnection SHOULD represent the same service. For example, the Remote Endpoints 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 endpoints. 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 pre-establishment.

6.1. Specifying Endpoints

The transport services API uses the Local Endpoint and Remote Endpoint Objects to refer to the endpoints of a transport 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 identifiers are set. For example, an Endpoint that only specifies a hostname may in fact end up corresponding 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 integer):

RemoteSpecifier.WithPort(443)

* Service (an identifier that maps to a port; either a the name of a well-known service, or a DNS SRV service name to be resolved):

RemoteSpecifier.WithService("https")

* IP address (IPv4 or IPv6 address):

RemoteSpecifier.WithIPv4Address(192.0.2.21)

RemoteSpecifier.WithIPv6Address(2001:db8:4920:e29d:a420:7461:7073:0a)

* Interface name (string), e.g., to qualify link-local or multicast addresses (see Section 6.1.2 for details):

LocalSpecifier.WithInterface("en0")

Note that an IPv6 address specified with a scope (e.g. 2001:db8:4920:e29d:a420:7461:7073:0a%en0) is equivalent to WithIPv6Address with an unscoped address and WithInterface together.

An Endpoint MUST NOT be configured with multiple identifiers of the same type. For example, an endpoint cannot have two IP addresses specified. 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, the Remote Endpoint specifies a hostname but no addresses, the Connection 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() method 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/Remote Endpoint specifying the any-source multicast (ASM) or source-specific multicast (SSM) multicast group and destination port number.

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 indicating the local interface to which the connection is bound and a Remote Endpoint indicating the multicast group.

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

```
RemoteSpecifier.WithMulticastGroupIPv4(GroupAddress)
RemoteSpecifier.WithMulticastGroupIPv6(GroupAddress)
RemoteSpecifier.WithPort(PortNumber)
RemoteSpecifier.WithTTL(TTL)
```

Calling Listen() on a Preconnection with a multicast group specified on the Remote Endpoint will 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 set to the 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 MessagContext on the messages received (see Section 9.1.1 for further details).

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

```
LocalSpecifier.WithSingleSourceMulticastGroupIPv4(GroupAddress, SourceAddress)
LocalSpecifier.WithSingleSourceMulticastGroupIPv6(GroupAddress, SourceAddress)
LocalSpecifier.WithAnySourceMulticastGroupIPv4(GroupAddress)
LocalSpecifier.WithAnySourceMulticastGroupIPv6(GroupAddress)
LocalSpecifier.WithPort(PortNumber)
```

Calling Rendezvous() on a Preconnection with an any-source multicast group address as the Remote Endpoint will join the multicast group, and also indicates that the resulting connection can be used to send messages to the multicast group. The Rendezvous() call will return both a Connection that can be used to send to the group, that acts the same as a connection returned by calling Initiate() with a multicast Remote Endpoint, and a Listener that acts as if Listen() had been called with a multicast Remote Endpoint. Calling Rendezvous() on a Preconnection with a source-specific multicast group address as the Local Endpoint results in an EstablishmentError.

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

```
RemoteSpecifier.WithMulticastGroupIPv4(GroupAddress)
RemoteSpecifier.WithMulticastGroupIPv6(GroupAddress)
RemoteSpecifier.WithPort(PortNumber)
RemoteSpecifier.WithTTL(TTL)
LocalSpecifier.WithAnySourceMulticastGroupIPv4(GroupAddress)
LocalSpecifier.WithAnySourceMulticastGroupIPv6(GroupAddress)
LocalSpecifier.WithPort(PortNumber)
LocalSpecifier.WithTTL(TTL)
```

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 RemoteEndpoint qualifies the scope of the remote endpoint, e.g., for link-local addresses.

* Specifying an interface on a LocalEndpoint 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 Pre-Establishment Phase, the Selection Properties prioritize and constrain the connection establishment.

6.1.3. Endpoint Aliases

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

To support this, Endpoint Objects can specify "aliases". An Endpoint can have multiple aliases set.

```
RemoteSpecifier.AddAlias(AlternateRemoteSpecifier)
```

In order to scope an alias to a specific transport protocol, an Endpoint can specify a protocol identifier.

```
RemoteSpecifier.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.

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

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

RemoteSpecifier.AddAlias(QUICRemoteSpecifier)
```

6.1.4. Endpoint Examples

The following examples of Endpoints show common usage patterns.

Specify a Remote Endpoint using a hostname and service name:

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

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

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

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

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

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

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

As an alternative to specifying an interface name for the Local Endpoint, an application can express more fine-grained preferences using the Interface Instance or Type Selection Property, see Section 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 Any-Source Multicast group in receive-only mode, bound to a known port on a named local interface:

```
RemoteSpecifier := NewRemoteEndpoint()

LocalSpecifier := NewLocalEndpoint()
LocalSpecifier.WithAnySourceMulticastGroupIPv4(233.252.0.0)
LocalSpecifier.WithPort(5353)
LocalSpecifier.WithInterface("en0")

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

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

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

```
RemoteSpecifier := NewRemoteEndpoint()

LocalSpecifier := NewLocalEndpoint()
LocalSpecifier.WithSingleSourceMulticastGroupIPv4(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 a Source-Specific Multicast group as a sender:

```
RemoteSpecifier := NewRemoteEndpoint()
RemoteSpecifier.WithMulticastGroupIPv4(232.1.1.1)
RemoteSpecifier.WithPort(5353)
RemoteSpecifier.WithTTL(8)

LocalSpecifier := NewLocalEndpoint()
LocalSpecifier.WithIPv4Address(192.0.2.22)
LocalSpecifier.WithInterface("en0")

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

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

Join an any-source multicast group as both a sender and a receiver:

```
RemoteSpecifier := NewRemoteEndpoint()
RemoteSpecifier.WithMulticastGroupIPv4(233.252.0.0)
RemoteSpecifier.WithPort(5353)
RemoteSpecifier.WithTTL(8)

LocalSpecifier := NewLocalEndpoint()
LocalSpecifier.WithAnySourceMulticastGroupIPv4(233.252.0.0)
LocalSpecifier.WithIPv4Address(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 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 may involve choosing between multiple local interfaces that are connected to different access networks.

When additional information (such as Provisioning Domain (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 PMTU, set of supported DSCPs, expected usage, cost, etc. The usage of this information by the Transport Services System is generally independent of the specific mechanism/protocol used to receive the information (e.g. zero-conf, DHCP, or IPv6 RA).

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

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

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 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 system SHOULD determine the preferred path first, regardless of protocol preferences. This ordering is chosen to provide consistency across implementations, 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 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 either by being explicitly configured via a Preconnection, by configuration after establishment, or by inheriting them from an antecedent via cloning; see Section 7.4 for more.

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 can not be changed after a Connection is established. After a Connection is established, 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 may alternatively use the two Preference values Require and Prohibit to represent true and false, respectively.

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 implementations 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 without corruption. 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: Ignore

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: Ignore

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: Ignore

This property specifies whether an application would like to supply a Message to the transport protocol before Connection establishment that will then be reliably transferred to the other side before or during Connection establishment. This Message can potentially be received multiple times (i.e., multiple copies of the message data may be passed to the Remote Endpoint). See also Section 9.1.3.4.

6.2.6. Multistream Connections in Group

Name: multistreaming

Type: Preference

Default: Prefer

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

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 later control of the sender checksum coverage (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 later control of the required minimum receiver checksum coverage (see Section 8.1.1).

6.2.9. Congestion control

Name: congestionControl

Type: Preference

Default: Require

This property specifies whether the application would like the Connection to be congestion controlled or not. Note that if a Connection is not congestion controlled, an application using such a Connection SHOULD itself perform congestion control in accordance with [RFC2914] 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

Name: keepAlive

Type: Preference

Default: Ignore

This property specifies whether the application would like the Connection to send keep-alive packets or not. Note that if a Connection determines that keep-alive packets are being sent, the applicaton should itself 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: Collection 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 tuple of an (Enumerated) interface identifier and a preference, and can either be implemented directly as such, or for making one preference available for each interface and interface type available on the system.

The set of valid interface types is implementation- and system-specific. For example, on a mobile device, there may be Wi-Fi and Cellular interface types available; whereas on a desktop computer, 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 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: Collection 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 Provisioning Domain (PvD) or categories of PVDs it wants to Require, Prohibit, Prefer, or Avoid. Provisioning Domains define consistent sets of network properties that may be more specific than network interfaces [RFC7556].

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

The identification of a specific PvD is implementation- and system-specific, because there is currently no portable standard format for a PvD identifier. For example, this identifier might be a string name or an integer. As with requiring specific interfaces, requiring a specific PvD strictly limits the path selection.

Categories or types of PvDs are also defined to be implementation- and system-specific. 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 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 "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:

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

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

Passive: The connection will support the use of multiple paths if the Remote Endpoint requests it.

The policy for using multiple paths is specified using the separate multipathPolicy property, see Section 8.1.7 below. To enable the peer endpoint to initiate additional paths towards a local address other than the one initially used, it is necessary to set the advertisesAltaddr property (see Section 6.2.15 below).

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 below) is set to false.

Note that Multipath Transport 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, should be advertised to the peer endpoint by the protocol stack. Advertising these addresses enables the peer-endpoint to establish additional connectivity, e.g., for connection migration or using multiple paths.

Note that this 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 above); 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:

Bidirectional: The connection must support sending and receiving

data

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

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

Since unidirectional communication can be supported by transports offering bidirectional communication, specifying unidirectional communication may cause a transport stack that supports bidirectional communication to be selected.

6.2.17. Notification of ICMP soft error message arrival

Name: softErrorNotify

Type: Preference

Default: Ignore

This property specifies whether an application considers it useful to be informed when an ICMP error message arrives that does not force termination of a connection. When set to true, received ICMP errors are available as SoftErrors, 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: Ignore

The most common client-server communication pattern involves the client actively opening a Connection, then sending data to the server. The server listens (passive open), reads, and then answers. This property specifies whether an application wants to diverge from this pattern -- either by actively opening with Initiate(), immediately followed by reading, or passively opening with Listen(), immediately followed by writing. This property is ignored when establishing connections using Rendezvous(). Requiring this property limits the choice of mappings to underlying protocols, which can reduce efficiency. For example, it prevents the Transport Services system from mapping Connections to SCTP streams, where the first transmitted data takes the role of an active open signal [I-D.ietf-taps-impl].

6.3. Specifying Security Parameters and Callbacks

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

6.3.1. Specifying Security Parameters on a Pre-Connection

Common security parameters such as TLS ciphersuites are known to implementations. Clients should use common safe defaults for these values 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. These configuration parameters need to be specified in the pre-connection phase and are created as follows:

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

Security configuration parameters and sample usage follow:

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

```
SecurityParameters.Set(identity, myIdentity)
SecurityParameters.Set(key-pair, myPrivateKey, myPublicKey)
```

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

```
SecurityParameters.Set(supported-group, "secp256r1")
SecurityParameters.Set(ciphersuite, "TLS_AES_128_GCM_SHA256")
SecurityParameters.Set(signature-algorithm, "ecdsa_secp256r1_sha256")
```

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

```
SecurityParameters.Set(pre-shared-key, key, identity)
```

* Session cache management: Used to tune session cache capacity, lifetime, and other policies.

```
SecurityParameters.Set(max-cached-sessions, 16)
SecurityParameters.Set(cached-session-lifetime-seconds, 3600)
```

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 can be 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 will still attempt to use security if available.

```
SecurityParameters := NewDisabledSecurityParameters()

SecurityParameters := NewOpportunisticSecurityParameters()
```

Representation of security parameters in implementations should parallel that chosen for Transport Property names as sugggested in Section 5.

6.3.2. Connection Establishment Callbacks

Security decisions, especially pertaining to trust, are not static. Once configured, parameters may also be supplied during connection establishment. These are best handled as client-provided callbacks. Callbacks block the progress of the connection establishment, which distinguishes them from other Events in the transport system. How callbacks and events are implemented is specific to each implementation. Security handshake callbacks that may 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 trust, return the result
})
SecurityParameters.SetTrustVerificationCallback(trustCallback)
```

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

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

7. Establishing Connections

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

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 specifier, optionally a Local Endpoint specifier (if not specified, the system will attempt to determine a suitable Local Endpoint), as well as all properties necessary for candidate selection.

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

Once Initiate is called, the candidate Protocol Stack(s) may cause one or more candidate transport-layer connections to be created to the specified Remote Endpoint. The caller may immediately begin sending Messages on the Connection (see Section 9.2) after calling Initiate(); note that any data marked as "safely replayable" that is sent while the Connection is being established may be sent multiple times or on multiple candidates.

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

Connection -> Ready<>

The Ready Event occurs after Initiate has established a transport-layer connection on at least one usable candidate Protocol Stack over at least one candidate Path. No Receive Events (see Section 9.3) will occur before the Ready Event for Connections established using Initiate.

Connection -> EstablishmentError<reason?>

An EstablishmentError occurs either when the set of transport properties and security parameters cannot be fulfilled on a Connection for initiation (e.g., the set of available Paths and/or Protocol Stacks meeting the constraints is empty) or reconciled with the Local and/or Remote Endpoints; when the remote specifier cannot

be resolved; or when no transport-layer connection can be established to the Remote Endpoint (e.g., because the Remote Endpoint is not accepting connections, the application is prohibited from opening a Connection by the operating system, or the establishment attempt has timed out for any other reason).

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 pre-establishment phase with a Local Endpoint specifier, as well as all properties necessary for Protocol Stack selection. A Remote Endpoint may optionally be specified, to constrain what Connections are accepted.

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

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

Listener -> Stopped<>

A Stopped Event occurs after the Listener has stopped listening.

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 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, Rendezvous() can be initiated using Local Endpoints that support a method of discovering NAT bindings such as Session Traversal Utilities for NAT (STUN) [RFC8489] or Traversal Using Relays around NAT (TURN) [RFC8656]. In this case, the Local Endpoint will resolve to a mixture of local and server reflexive addresses. The Resolve() action on the Preconnection can be used to discover these bindings:

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

The Resolve() call 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; it is valid only for a Rendezvous happening at the same time as the resolution. Care should be taken in using these values in any other context.

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 a 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 signalling protocol, for example as part of an ICE [RFC8445] exchange within SIP [RFC3261] or WebRTC [RFC7478]. The peer will then, via the same signalling channel, return the Remote Endpoint candidates. The set of Remote Endpoint candidates are then configured onto the Preconnection:

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

The Rendezvous() Action can be initiated once both the Local Endpoint candidates and the Remote Endpoint candidates retrieved from the peer via the signalling channel have been added to the Preconnection.

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 do not have any effect on existing Connections.

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

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 openend, 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 may 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, there are possibilities for a Transport Services system to implement prioritization within a Connection Group [TCP-COUPLING] [RFC8699].

Attempts to clone a Connection can result in a CloneError:

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

The connPriority Connection Property operates on Connections in a Connection Group using the same approach as in Section 9.1.3.2: when allocating available network capacity among Connections in a Connection Group, sends on Connections with higher Priority values will be prioritized over sends on Connections that have lower 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.

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.

In some cases, however, it is necessary to explicitly indicate to a Connection that a new remote endpoint has become available for use, or to indicate that some 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/from a Connection.

8. Managing Connections

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

Connection Properties represent the configuration and state of the selected Protocol Stack(s) backing a Connection. These Connection Properties may be Generic, applying regardless of transport protocol,

or Specific, applicable to a single implementation of a single transport protocol stack. Generic Connection Properties are defined in Section 8.1 below.

Protocol-specific Properties are defined in a transport- and implementation-specific way 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 implementation to make appropriate selection and configuration choices. Therefore, it is RECOMMENDED that Protocol-specific Properties are used for properties common across different protocols and that Protocol-specific 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 pre-establishment (see Section 6.2), as well as on connections directly using the SetProperty action:

```
Connection.SetProperty(property, value)
```

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

* Whether the connection can be used to send data. A connection can not be used for sending if the connection was created with the Selection Property direction 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. A connection cannot be used for reading 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 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: 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 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 independent of the chosen protocol stack and therefore 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 may also have a zero checksum field, and the enumerated value Full Coverage means that the entire Message needs to be protected by a checksum.

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 higher 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 may ignore this property. See Section 9.2.6 for more details.

8.1.3. Timeout for Aborting Connection

Name: connTimeout

Type: Numeric (non-negative) 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. Adjusting this property will only take effect when the underlying stack supports reliability. If this property has the enumerated value Disabled, it means that no timeout is scheduled.

8.1.4. Timeout for keep alive packets

Name: keepAliveTimeout

Type: Numeric (non-negative) 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) should wait before the Local Endpoint sends a keep-alive packet to the Remote Endpoint. Adjusting this property will only take effect when the underlying stack supports sending keep-alive packets. Guidance on setting this value for connection-less 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.

8.1.5. Connection Group Transmission Scheduler

Name: connScheduler

Type: Enumeration

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

This property specifies which scheduler should be used among Connections within a Connection Group, see Section 7.4. The set of schedulers can be taken from [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 tradeoffs the application is prepared to make in path and protocol selection to receive that desired treatment. When the capacity profile is set to a value other than Default, the Transport Services system SHOULD select paths and configure protocols to optimize the tradeoff between delay, delay variation, and efficient use of the available capacity based on the capacity profile specified. How this is realized is implementation-specific. The Capacity Profile MAY also be used to set markings on the wire for Protocol Stacks supporting this. Recommendations for use with DSCP are provided below for each profile; note that when a Connection is multiplexed, the guidelines in Section 6 of [RFC7657] apply.

The following values are valid for the Capacity Profile:

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

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

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

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

Constant-Rate Streaming: The application expects to send/receive data at a constant rate after Connection establishment. Delay and delay variation should be minimized at the expense of efficient use of the available capacity. This implies that the Connection 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 onto per-connection DSCP signaling without multiplexing SHOULD assign a DSCP Assured Forwarding (AF31,AF32,AF33,AF34) [RFC2597] PHB.

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

The Capacity Profile for a selected protocol stack may be modified on a per-Message basis using the Transmission Profile Message Property; see Section 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 Multipath Transport is not set to Disabled (see Section 6.2.14). Possible values are:

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 â\200\223 the Remote Endpoint can choose a different policy.

8.1.8. Bounds on Send or Receive Rate

Name: minSendRate / minRecvRate / maxSendRate / maxRecvRate

Type: Numeric (non-negative) or Unlimited / Numeric (non-negative) or Unlimited / Numeric (non-negative) or Unlimited / Numeric (non-negative) or Unlimited

Default: Unlimited / Unlimited / Unlimited / Unlimited

Numeric values of this property 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 rate below which the application does not deem it will be useful. These are specified in bits per second. The enumerated value Unlimited indicates that no bound is specified.

8.1.9. Group Connection Limit

Name: groupConnLimit

Type: Numeric (non-negative) 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.

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 [I-D.ietf-taps-arch].

Note that this does not guarantee no leakage of information, as implementations may not be able to fully isolate all caches (e.g. RTT estimates). Note that this property may 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. Maximum Message Size Concurrent with Connection Establishment

Name: zeroRttMsgMaxLen

Type: Integer (non-negative)

This property represents the maximum Message size that can be sent before or during Connection establishment, see also Section 9.1.3.4. It is specified as the number of bytes.

8.1.11.2. 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 the number of bytes. It exposes a value to the application based on the Maximum Packet Size (MPS) as described in Datagram PLPMTUD [RFC8899]. This can allow a sending stack to avoid unwanted fragmentation at the network-layer or segmentation by the transport layer.

8.1.11.3. 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.

8.1.11.4. Maximum Message Size on Receive

Name: recvMsgMaxLen

Type: Integer (non-negative)

This property represents the maximum Message size that an application can receive. It specified as the number of bytes.

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

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

These TCP-specific properties are included here because the feature Suggest timeout to the peer is part of the minimal set of transport services [RFC8923], where this feature was categorized as "functional". This means that when an Transport Services implementation 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 User Timeout Enabled as true, but not specify an Advertised User Timeout 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 (non-negative)

Default: the TCP default

This time value is advertised via the TCP User Timeout Option (UTO) [RFC5482] at the Remote Endpoint to adapt its own Timeout for aborting Connection (see Section 8.1.3) value.

8.2.2. User Timeout Enabled

Name: tcp.userTimeoutEnabled

Type: Boolean

Default: false

This property controls whether the UTO option 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 connTimeout (see Section 8.1.3) may be changed based on a UTO option 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 Message Context, which allows to add Message Properties, identify Send Events related to a specific Message or to inspect meta-data 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 meta-data of the message, including Message Properties (see Section 9.1.3) and framing meta-data (see Section 9.1.2.2). Therefore, a MessageContext object can be passed to the Send action and is returned by each Send and Receive related event.

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

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

These Message Properties may 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 current application protocols evolved over TCP, which does not provide message boundary

preservation, and since many of these protocols require message boundaries to function, each application layer protocol has defined its own framing.

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

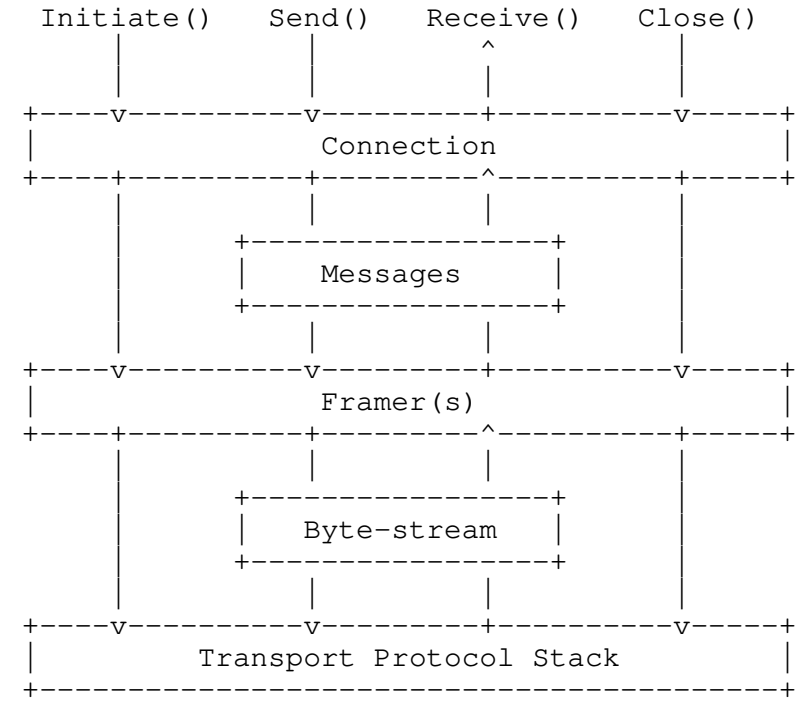


Figure 1: Protocol Stack showing a Message Framer

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 an additional transformation to further encode or encapsulate, and can potentially support packing multiple application-layer Messages into individual transport datagrams.

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

9.1.2.1. Adding Message Framers to Pre-Connections

The Message Framer object can be added to one or more Preconnections to run on top of transport protocols. Multiple Framers may be added 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 may result in the creation of a Connection.

9.1.2.2. Framing Meta-Data

When sending Messages, applications can add Framer-specific properties to a MessageContext (Section 9.1.1). In order to set these properties, the add and get actions on the MessageContext. 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 Message Context passed to the Send Action. For other uses of the message context, 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 Message Contexts. For example, it would not make sense to have the beginning of a Message expire, but allow the end of a Message to still be sent.

A MessageContext object contains metadata for 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 call, further modifications to the MessageContext object do not have any effect on this Send call. 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 call. 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.

The following Message Properties are supported:

9.1.3.1. Lifetime

Name: msgLifetime

Type: Numeric (non-negative)

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

Note that this property is not a per-message override of connPriority - see Section 8.1.2. The priority properties may interact, but 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 may 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 signalling 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 Priority and any other property that would re-order Messages. If another Message is sent after a Message marked as Final has already been sent on a Connection, the Send Action for the new Message will cause a SendError Event.

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 all data is received on the other side without corruption. 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 may 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 tradeoffs 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 prefered.

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 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 should be used when it is desired to only 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 the transport layer to not provide segmentation of messages larger than the maximum size permitted by the network layer, and also to avoid network-layer source fragmentation of messages. When running over IPv4, setting this property to true will result in a sending endpount 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 layer 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 meta-data 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 Connectionâ\200\231s transport properties. For example, a Message may be a single datagram for UDP Connections; or an HTTP Request for HTTP Connections.

Some transport protocols can deliver arbitrarily sized Messages, but other protocols constrain the maximum Message size. Applications can query the Connection Property sendMsgMaxLen (Section 8.1.11.3) to determine the maximum size allowed for a single Message. If a Message is too large to fit in the Maximum Message Size for the Connection, the Send will fail with a SendError event (Section 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 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 Sends 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 which Send Action resulted in a particular Send 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 Message Context 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 which is too large for the system and Protocol Stack to handle, some failure of the underlying Protocol Stack, or a set of Message Properties not consistent with the Connection's transport properties. The SendError contains a reference to the Message Context 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 may be too large for the application to hold in memory at one time, or the length of the Message may be unknown or unbounded.

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

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

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

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

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

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

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?, timeou
```

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(s) do not support transmitting data upon connection establishment, InitiateWithSend is identical to Initiate() followed by Send().

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

The Events that may be sent after InitiateWithSend() are equivalent to those that would be sent by an invocation of Initiate() followed immediately by an invocation of Send(), with the caveat that a send failure that occurs because the Connection could not be established will not result in a SendError separate from the 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 Section 9.1.3.2 and connPriority 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 layer, including those which 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. However, the Transport Services system will seek to ensure that performance of relatively-prioritized connections and messages is not worse with respect to those connections and messages than an equivalent configuration in which all prioritization properties are left at their defaults.

The Transport Services API does order connPriority over msgPriority. In the absense 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 that should be delivered in response to this Receive. By default, this value is infinite, which means that only complete Messages should be delivered (see Section 9.3.2.2 and Section 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 when at least that many bytes are available, or the Message is complete with fewer bytes, or the system needs to free up memory. Applications should always check the length of the data delivered to the receive event and not assume it will be as long as minIncompleteLength in the case of shorter complete Messages or memory issues.

The maxLength argument indicates the maximum size of a Message in bytes 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, which can be a success or an error. This allows an application to provide backpressure to the transport stack when it is temporarily not ready to receive messages.

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

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

See Section 9.1.2 for handling 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 endOfMessage flag is delivered or a ReceiveError occurs. All partial blocks of a single Message are delivered in order without gaps. This event does not support delivering discontiguous partial Messages. If, for example, Message A is divided into three pieces (A1, A2, A3) and Message B is divided into three pieces (B1, B2, B3), and preserveOrder is not Required, the ReceivedPartial may deliver them in a sequence like this: A1, B1, B2, A2, A3, B3, 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 may 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 first 1000 bytes, message is incomplete
Connection -> ReceivedPartial<messageData(1000 bytes), messageContext, false>

Connection.Receive(1, 1000)

// Receive 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 when 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) 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 may want 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 signalled 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 Message Context may 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 following metadata values are supported:

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

When available, Message metadata carries the value of the Explicit Congestion Notification (ECN) field. This information can be used for logging and debugging, and for 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, and hence 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 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 Message Context can indicate whether or not this Message is the Final Message on a Connection. For any Message that is marked as Final, the application can assume that there will be no more Messages received on the Connection once the Message has been completely delivered. This corresponds to the final property that may be marked on a sent Message, see Section 9.1.3.5.

Some transport protocols and peers do not support signaling of the final property. Applications therefore 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 i) by the Local Endpoint (i.e., the application calls the Close, CloseGroup, Abort or AbortGroup Action), ii) by the Remote Endpoint (i.e., the remote application calls the Close, CloseGroup, Abort or AbortGroup Action), or iii) because of an error (e.g., a timeout). A local call of the Close Action will cause the Connection to either send a Closed Event or a ConnectionError Event, and a local call of the CloseGroup Action will cause all of the Connections in the group to either send 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, and 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 either send a Closed Event or a ConnectionError Event), but, different from 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 should therefore not rely on the invocation of such Events due to termination calls from the Remote Endpoint, but instead 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.

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 details of how exactly actions are handled, and how events are dispatched, are implementation dependent.

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

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

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

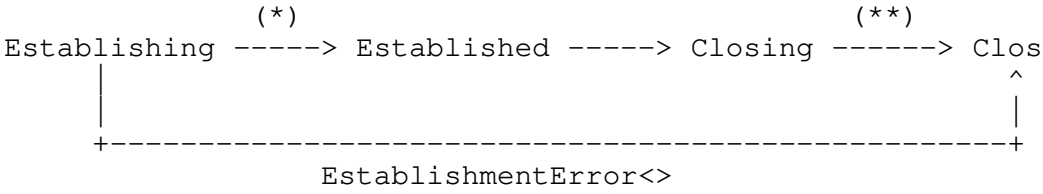
* RendezvousDone<> occurs when a Connection created with Rendezvous() transitions to Established state.

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

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

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

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



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

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 implementation).

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

* No events will occur on a Connection after it is Closed; i.e., after a Closed<> event, an EstablishmentError<> or ConnectionError<> will not occur on that connection. To ensure this ordering, Closed<> will not occur on a Connection while other events on the Connection are still locally outstanding (i.e., known to the Transport Services API and waiting to be dealt with by the application).

12. IANA Considerations

RFC-EDITOR: Please remove this section before publication.

This document has no Actions for IANA. Later versions of this document may create IANA registries for generic transport property names and transport property namespaces (see Section 4.1).

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.

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

In either case, the Transport Services API is an internal interface that is used to change 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 are useful to configure and select protocols and paths and are 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. In most cases, information provided for protocol and path selection should 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 communication attempts can lead to more than one connection establishment. This is the case, for example, when the Transport Services system also executes name resolution, when support mechanisms such as TURN or ICE are used to establish connectivity, if protocols or paths are raised, 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 may 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 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 tradeoff between early and late binding of addresses to names. Early binding allows the API implementation to reduce connection setup latency, at the cost of potentially limited scope for alternate path discovery during Connection establishment, as well as potential additional information leakage about application interest when used with a resolution method (such as DNS without TLS) which does not protect query confidentiality.

These communication activities are not different from what is used today. 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 for functional or security purposes. Further, a Transport Services system should provide an interface to poll information about which protocol and path is currently in use as well as provide logging about the communication events of each connection.

14. Acknowledgements

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

This work has been supported by Leibniz Prize project funds of DFG - German Research Foundation: Gottfried Wilhelm Leibniz-Preis 2011 (FKZ FE 570/4-1).

This work has been supported by the UK Engineering and Physical Sciences Research Council under grant EP/R04144X/1.

This work has been supported by 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.

15. References

15.1. Normative References

[I-D.ietf-taps-arch]
Pauly, T., Trammell, B., Brunstrom, A., Fairhurst, G., and C. Perkins, "An Architecture for Transport Services", Work in Progress, Internet-Draft, draft-ietf-taps-arch-16, 9 March 2023, <https://datatracker.ietf.org/doc/html/draft-ietf-taps-arch-16>.

[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/rfc/rfc2119>.

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

[RFC8084] Fairhurst, G., "Network Transport Circuit Breakers", BCP 208, RFC 8084, DOI 10.17487/RFC8084, March 2017, <https://www.rfc-editor.org/rfc/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/rfc/rfc8085>.

[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/rfc/rfc8174>.

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

[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/rfc/rfc8981>.

15.2. Informative References

[I-D.ietf-taps-impl] Brunstrom, A., Pauly, T., Enghardt, R., Tiesel, P. S., and M. Welzl, "Implementing Interfaces to Transport Services", Work in Progress, Internet-Draft, draft-ietf-taps-impl-15, 9 March 2023, <https://datatracker.ietf.org/doc/html/draft-ietf-taps-impl-15>.

[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/rfc/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/rfc/rfc2597>.

[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/rfc/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/rfc/rfc3261>.

[RFC4594] Babiarz, J., Chan, K., and F. Baker, "Configuration Guidelines for DiffServ Service Classes", RFC 4594, DOI 10.17487/RFC4594, August 2006, <https://www.rfc-editor.org/rfc/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/rfc/rfc5280>.

[RFC5482] Eggert, L. and F. Gont, "TCP User Timeout Option", RFC 5482, DOI 10.17487/RFC5482, March 2009, <https://www.rfc-editor.org/rfc/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/rfc/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/rfc/rfc7478>.

[RFC7556] Anipko, D., Ed., "Multiple Provisioning Domain Architecture", RFC 7556, DOI 10.17487/RFC7556, June 2015, <https://www.rfc-editor.org/rfc/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/rfc/rfc7657>.

[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/rfc/rfc8095>.

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

[RFC8260] Stewart, R., Tuexen, M., Loreto, S., and R. Seggelmann, "Stream Schedulers and User Message Interleaving for the Stream Control Transmission Protocol", RFC 8260, DOI 10.17487/RFC8260, November 2017, <https://www.rfc-editor.org/rfc/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/rfc/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/rfc/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/rfc/rfc8445>.

[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/rfc/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/rfc/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/rfc/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/rfc/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/rfc/rfc8699>.

[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/rfc/rfc8838>.

[RFC8899] Fairhurst, G., Jones, T., TÃ¼xen, M., RÃ¼ngeler, I., and T. VÃ¶lker, "Packetization Layer Path MTU Discovery for Datagram Transports", RFC 8899, DOI 10.17487/RFC8899, September 2020, <https://www.rfc-editor.org/rfc/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/rfc/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/rfc/rfc8923>.

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

[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 Global Internet Symposium (GI) workshop (GI 2018) , 2018.

Appendix A. Implementation Mapping

The way the concepts from this abstract API map into 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 functions or method calls, for instance, and Events could be implemented via event queues, handler functions or classes, communicating sequential processes, or other asynchronous calling conventions.

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:

* An Integer can typically be represented in C by an int or long, 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. In Python, by contrast, a Tuple may be represented natively as a tuple, a sequence of dynamically-typed elements.

* A Collection 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 similarly 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, according to the error handling idioms of the implementation platform, where they can be immediately detected, such as by generating 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, number of milliseconds, or a struct timeval in C or a user-defined Duration class in C++.

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 toTransportProperties.Avoid(some_property)‘:

```
TransportProperties.Require(property)
TransportProperties.Prefer(property)
TransportProperties.Ignore(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 pre-configured with frequently used sets of properties; the following are in common use in current applications:

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	ignore

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	ignore
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 Section 6.2.7 and Section 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, MPTCP, UDP, UDP-Lite, SCTP and LEDBAT available that 1) require interaction with the application, and 2) do not get in the way of a possible implementation over TCP (or, with limitations, UDP). The following text explains how this minimal set is reflected in the present API. For brevity, it is based on the list in Section 4.1 of [RFC8923], 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" (Section 4) of [RFC8303] for further details on the implementation with TCP, MPTCP, UDP, UDP-Lite, SCTP and LEDBAT.

* 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 max. transport-message size that may be sent using a non-fragmented IP packet from the configured interface: singularTransmissionMsgMaxLen property (Section 8.1.11.2).

* Get max. transport-message size that may be received from the configured interface: recvMsgMaxLen property (Section 8.1.11.4).

* Obtain ECN field: This is a read-only Message Property of the MessageContext object (see "UDP(-Lite)-specific Property: ECN" Section 9.3.3.1).

* "Specify DSCP field", "Disable Nagle algorithm", "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] (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": this is offered by the Abort action without promising that this is signaled to the other side. If it is, 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) and Received Event (Section 9.3.2.1).

* Receive a message: Receive Action (Section 9.3) 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) 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).

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

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

Godred Fairhurst
University of Aberdeen
Fraser Noble Building
Aberdeen, AB24 3UE
Email: gorry@erg.abdn.ac.uk
URI: http://www.erg.abdn.ac.uk/

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

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

Philipp S. Tiesel
SAP SE
Konrad-Zuse-Ring 10
14469 Potsdam
Germany
Email: philipp@tiesel.net

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