TAPS Working Group
Internet-Draft
Intended status: Standards Track
Expires: April 25, 2019

B. Trammell, Ed.
ETH Zurich
M. Welzl, Ed.
University of Oslo
T. Enghardt
TU Berlin
G. Fairhurst
University of Aberdeen
M. Kuehlewind
ETH Zurich
C. Perkins
University of Glasgow
P. Tiesel
TU Berlin
C. Wood
Apple Inc.
October 22, 2018

An Abstract Application Layer Interface to Transport Services
draft-ietf-taps-interface-02

Abstract

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

Status of This Memo

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

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

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

This Internet-Draft will expire on April 25, 2019.

Copyright Notice

Table of Contents

1. Introduction

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

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

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

provided in [RFC8095], [I-D.ietf-taps-minset], and [I-D.ietf-taps-transport-security].

2. Terminology and Notation

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

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

- An Action creates an Object:

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

- An Action creates an array of Objects:

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

- An Action 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(parameter, parameter, ...) / Event<parameter, parameter, ...>
```

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

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

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

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

3. Interface Design Principles

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

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

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

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

- Explicit support for multistreaming and multipath transport protocols, and the grouping of related Connections into Connection Groups through cloning of Connections, to allow applications to

take full advantage of new transport protocols supporting these features.

4. API Summary

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

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

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

Section 5, Section 6, Section 7, Section 8, and Section 10 describe the details of application interaction with Objects through Actions and Events in each phase of a Connection, following the phases described in [I-D.ietf-taps-arch].

4.1. Transport Properties

Each application using the Transport Services Interface declares its preferences for how the transport service should operate using properties at each stage of the lifetime of a connection. During pre-establishment, Selection Properties Section 5.2 are used to specify which paths and protocol stacks can be used and are preferred by the application, and Connection Properties Section 9.1 can be used to fine-tune the eventually established connection. These Connection Properties can also be used to monitor and fine-tune established connections. The behavior of the selected protocol stack(s) when sending Messages is controlled by Message Properties Section 7.3.

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

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

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

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

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

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

4.2. Scope of the Interface Definition

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

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

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

5. Pre-Establishment Phase

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

A Preconnection Object represents a potential Connection. It has state that describes properties of a Connection that might exist in the future. This state comprises Local Endpoint and Remote Endpoint Objects that denote the endpoints of the potential Connection (see Section 5.1), the Selection Properties (see Section 5.2), any preconfigured Connection Properties (Section 9.1), and the security parameters (see Section 5.3):

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

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

Framers (see Section 7.7) and deframers (see Section 8.4), if necessary, should be bound to the Preconnection during pre-establishment.

5.1. Specifying Endpoints

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

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

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

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

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

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

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

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

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

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

5.2. Specifying Transport Properties

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

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

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

Preference	Effect
Require	Select only protocols/paths providing the property, fail otherwise
Prefer	Prefer protocols/paths providing the property, proceed otherwise
Ignore	Cancel any system default preference for this property
Avoid	Prefer protocols/paths not providing the property, proceed otherwise
Prohibit	Select only protocols/paths not providing the property, fail otherwise

Internally, the transport system will first exclude all protocols and paths that match a Prohibit, then exclude all protocols and paths that do not match a Require, then sort candidates according to Preferred properties, and then use Avoided properties as a tiebreaker. Selection Properties which select paths take preference over those which select protocols. For example, if an application indicates a preference for a specific path by specifying an interface, but also a preference for a protocol not available on this

path, the transport system will try the path first, ignoring the preference.

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

```
TransportProperties := NewTransportProperties()
```

Individual properties are then added to the TransportProperties Object:

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

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

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

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

```
TransportProperties := Connection.GetTransportProperties()
```

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

Section 9.1 provides a list of Connection Properties, while Selection Properties are listed in the subsections below. Note that many properties are only considered during establishment, and can not be changed after a Connection is established; however, they can be queried. Querying a Selection Property after establishment yields the value Required for properties of the selected protocol and path, Avoid for properties avoided during selection, and Ignore for all other properties.

An implementation of this interface must provide sensible defaults for Selection Properties. The recommended defaults given for each

property below represent a configuration that can be implemented over TCP. An alternate set of default Protocol Selection Properties would represent a configuration that can be implemented over UDP.

5.2.1. Reliable Data Transfer (Connection)

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

5.2.2. Configure per-Message reliability

This property specifies whether an application considers it useful to indicate its reliability requirements on a per-Message basis. This property applies to Connections and Connection Groups. The recommended default is to not have this option.

5.2.3. Preservation of data ordering

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

5.2.4. Use 0-RTT session establishment with an idempotent Message

This property specifies whether an application would like to supply a Message to the transport protocol before Connection establishment, which will then be reliably transferred to the other side before or during Connection establishment, potentially multiple times. See also Section 7.3.4. The recommended default is to not have this option.

5.2.5. Multistream Connections in Group

This property specifies that the application would prefer multiple Connections within a Connection Group to be provided by streams of a single underlying transport connection where possible. The recommended default is to not have this option.

5.2.6. Control checksum coverage on sending or receiving

This property specifies whether the application considers it useful to enable, disable, or configure a checksum when sending a Message, or configure whether to require a checksum or not when receiving.

The recommended default is full checksum coverage without the option to configure it, and requiring a checksum when receiving.

5.2.7. Congestion control

This property specifies whether the application would like the Connection to be congestion controlled or not. Note that if a Connection is not congestion controlled, an application using such a Connection should itself perform congestion control in accordance with [RFC2914]. Also note that reliability is usually combined with congestion control in protocol implementations, rendering "reliable but not congestion controlled" a request that is unlikely to succeed. The recommended default is that the Connection is congestion controlled.

5.2.8. Interface Instance or Type

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

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

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

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

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

Interface types should not be treated as a proxy for properties of interfaces such as metered or unmetered network access. If an application needs to prohibit metered interfaces, this should be specified via Provisioning Domain attributes (see Section 5.2.9) or another specific property.

5.2.9. Provisioning Domain Instance or Type

Similar to interface instances and types (see Section 5.2.8), this property allows the application to control path selection by selecting which specific Provisioning Domains or categories of Provisioning Domains it wants to "Require", "Prohibit", "Prefer", or "Avoid". Provisioning Domains define consistent sets of network properties that may be more specific than network interfaces [RFC7556].

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

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

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

5.3. Specifying Security Parameters and Callbacks

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

5.3.1. Pre-Connection Parameters

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

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

Security configuration parameters and sample usage follow:

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

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

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

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

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

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

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

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

5.3.2. Connection Establishment Callbacks

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

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

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

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

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

6. Establishing Connections

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

6.1. Active Open: Initiate

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

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

Before calling Initiate, the caller must have populated a Preconnection Object with a Remote Endpoint specifier, optionally a

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

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

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

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

Connection -> Ready<>

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

Connection -> InitiateError<>

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

See also Section 7.6 to combine Connection establishment and transmission of the first message in a single action.

6.2. Passive Open: Listen

Passive open is the Action of waiting for Connections from remote endpoints, commonly used by servers in client-server interactions.

Passive open is supported by this interface through the Listen Action:

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

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

Preconnection.Stop()

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

Preconnection -> ConnectionReceived<Connection>

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

Preconnection -> ListenError<>

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

Preconnection -> Stopped<>

A Stopped event occurs after the Preconnection has stopped listening.

6.3. Peer-to-Peer Establishment: Rendezvous

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

Preconnection.Rendezvous()

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

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

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

Preconnection -> RendezvousDone<Connection>

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

Preconnection -> RendezvousError<msgRef, error>

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

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

[]Preconnection := Preconnection.Resolve()

The Resolve() call returns a list of Preconnection Objects, that represent the concrete addresses, local and server reflexive, on which a Rendezvous() for the Preconnection will listen for incoming

Connections. These resolved Preconnections will share all other Properties with the Preconnection from which they are derived, though some Properties may be made more-specific by the resolution process. This list can be passed to a peer via a signalling protocol, such as SIP [RFC3261] or WebRTC [RFC7478], to configure the remote.

6.4. Connection Groups

Groups of Connections can be created using the Clone Action:

```
Connection := Connection.Clone()
```

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

Changing one of these Protocol Properties on one Connection in the group changes it for all others. Per-Message Protocol Properties, however, are not entangled. For example, changing "Timeout for aborting Connection" (see Section 9.1.6) on one Connection in a group will automatically change this Protocol Property for all Connections in the group in the same way. However, changing "Lifetime" (see Section 7.3.1) of a Message will only affect a single Message on a single Connection, entangled or not.

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

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

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

The Protocol Property "Niceness" operates on entangled Connections as in Section 7.3.2: when allocating available network capacity among Connections in a Connection Group, sends on Connections with higher Niceness values will be prioritized over sends on Connections with lower Niceness values. An ideal transport system implementation would assign each Connection the capacity share (M-N) x C / M, where N is the Connection's Niceness value, M is the maximum Niceness value

used by all Connections in the group and C is the total available capacity. However, the Niceness setting is purely advisory, and no guarantees are given about the way capacity is shared. Each implementation is free to implement a way to share capacity that it sees fit.

7. Sending Data

Once a Connection has been established, it can be used for sending data. Data is sent in terms of Messages, which allow the application to communicate the boundaries of the data being transferred. By default, Send enqueues a complete Message, and takes optional per-Message properties (see Section 7.1). All Send actions are asynchronous, and deliver events (see Section 7.2). Sending partial Messages for streaming large data is also supported (see Section 7.4).

7.1. Basic Sending

The most basic form of sending on a connection involves enqueuing a single Data block as a complete Message, with default Message Properties. Message data is created as an array of octets, and the resulting object contains both the byte array and the length of the array.

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

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

Some transport protocols can deliver arbitrarily sized Messages, but other protocols constrain the maximum Message size. Applications can query the protocol property Maximum Message Size on Send to determine the maximum size allowed for a single Message. If a Message is too large to fit in the Maximum Message Size for the Connection, the Send will fail with a SendError event (Section 7.2.3). For example, it is invalid to send a Message over a UDP connection that is larger than the available datagram sending size.

7.2. Send Events

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

Note that if partial Sends are used (Section 7.4), there will still be exactly one Send Event delivered for each call to Send. For example, if a Message expired while two requests to Send data for that Message are outstanding, there will be two Expired events delivered.

7.2.1. Sent

Connection -> Sent<msgRef>

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 implementation of this interface. The exact disposition of the Message (i.e., whether it has actually been transmitted, moved into a buffer on the network interface, moved into a kernel buffer, and so on) when the Sent Event occurs is implementation-specific. The Sent Event contains an implementation-specific reference to the Message to which it applies.

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

7.2.2. Expired

Connection -> Expired<msgRef>

The Expired Event occurs when a previous Send Action expired before completion; i.e. when the Message was not sent before its Lifetime (see Section 7.3.1) expired. This is separate from SendError, as it is an expected behavior for partially reliable transports. The Expired Event contains an implementation-specific reference to the Message to which it applies.

7.2.3. SendError

Connection -> SendError<msgRef>

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

7.3. Message Properties

Applications may need to annotate the Messages they send with extra information to control how data is scheduled and processed by the transport protocols in the Connection. A MessageContext object contains properties for sending Messages, and can be passed to the Send Action. Note that these properties are per-Message, not per-Send if partial Messages are sent (Section 7.4). All data blocks associated with a single Message share properties. 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.

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

The simpler form of Send that does not take any messageContext is equivalent to passing a default MessageContext with not values added.

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

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

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

The following Message Properties are supported:

7.3.1. Lifetime

Type: Integer

Lifetime specifies how long a particular Message can wait to be sent to the remote endpoint before it is irrelevant and no longer needs to be (re-)transmitted. When a Message's Lifetime is infinite, it must be transmitted reliably. The type and units of Lifetime are implementation-specific.

7.3.2. Niceness

Type: Integer (non-negative)

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

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

Note that this property is not a per-message override of the connection Niceness - see Section 9.1.5. Both Niceness properties may interact, but can be used independently and be realized by different mechanisms.

7.3.3. Ordered

Type: Boolean

If true, it specifies that the receiver-side transport protocol stack only deliver the Message to the receiving application after the previous ordered Message which was passed to the same Connection via the Send Action, when such a Message exists. If false, the Message may be delivered to the receiving application out of order. This property is used for protocols that support preservation of data ordering, see Section 5.2.3, but allow out-of-order delivery for certain messages.

7.3.4. Idempotent

Type: Boolean

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

7.3.5. Final

Type: Boolean

If true, this Message is the last one that the application will send on a Connection. This allows underlying protocols to indicate to the

Remote Endpoint that the Connection has been effectively closed in the sending direction. For example, TCP-based Connections can send a FIN once a Message marked as Final has been completely sent, indicated by marking endOfMessage. Protocols that do not support signalling the end of a Connection in a given direction will ignore this property.

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

7.3.6. Corruption Protection Length

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

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

7.3.7. Reliable Data Transfer (Message)

Type: Boolean

This property specifies that a message should be sent in such a way that the transport protocol ensures all data is received on the other side without corruption. Changing the 'Reliable Data Transfer' property on Messages is only possible if the Connection supports reliability. When this is not the case, changing it will generate an error.

7.3.8. Transmission Profile

Type: Enumeration

This enumerated property specifies the application's preferred tradeoffs for sending this Message; it is a per-Message override of the Capacity Profile protocol and path selection property (see Section 9.1.12).

The following values are valid for Transmission Profile:

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

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

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

7.3.9. Singular Transmission

Type: Boolean

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

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

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

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

7.5. Batching Sends

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

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

7.6. Send on Active Open: InitiateWithIdempotentSend

For application-layer protocols where the Connection initiator also sends the first message, the InitiateWithIdempotentSend() action combines Connection initiation with a first Message sent, provided that message is idempotent.

Without a message context (as in Section 7.1):

```
Connection := Preconnection.InitiateWithIdempotentSend(messageData)
```

With a message context (as in Section 7.3):

```
Connection := Preconnection.InitiateWithIdempotentSend(messageData,
messageContext)
```

The message passed to InitiateWithIdempotentSend() is, as suggested by the name, considered to be idempotent (see Section 7.3.4)

regardless of declared message properties or defaults. If protocol stacks supporting 0-RTT establishment with idempotent data are available on the Preconnection, then 0-RTT establishment may be used with the given message when establishing candidate connections. For a non-idemponent initial message, or when the selected stack(s) do not support 0-RTT establishment, InitiateWithIdempotentSend is identical to Initiate() followed by Send().

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

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

7.7. Sender-side Framing

Sender-side framing allows a caller to provide the interface with a function that takes a Message of an appropriate application-layer type and returns an array of octets, the on-the-wire representation of the Message to be handed down to the Protocol Stack. It consists of a Framer Object with a single Action, Frame. Since the Framer depends on the protocol used at the application layer, it is bound to the Preconnection during the pre-establishment phase:

```
Preconnection.FrameWith(Framer)

OctetArray := Framer.Frame(messageData)
```

Sender-side framing is a convenience feature of the interface, for parity with receiver-side framing (see Section 8.4).

8. Receiving Data

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

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

8.1. Enqueuing Receives

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

Connection.Receive(minIncompleteLength, maxLength)

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

The application can set a minIncompleteLength value to indicates the smallest partial Message data size in bytes that should be delivered in response to this Receive. By default, this value is infinite, which means that only complete Messages should be delivered (see Section 8.2.2 and Section 8.4 for more information on how this is accomplished). If this value is set to some smaller value, the associated receive event will be triggered only when at least that many bytes are available, or the Message is complete with fewer bytes, or the system needs to free up memory. Applications should always check the length of the data delivered to the receive event and not assume it will be as long as minIncompleteLength in the case of shorter complete Messages or memory issues.

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

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

8.2. Receive Events

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

8.2.1. Received

Connection -> Received<messageData, messageContext>

A Received event indicates the delivery of a complete Message. It contains two objects, the received bytes as messageData, and the metadata and properties of the received Message as messageContext. See {#receive-context} for details about the received context.

The messageData object provides access to the bytes that were received for this Message, along with the length of the byte array.

See Section 8.4 for handling Message framing in situations where the Protocol Stack provides octet-stream transport only.

8.2.2. ReceivedPartial

Connection -> ReceivedPartial<messageData, messageContext, endOfMessage>

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

Multiple invocations of ReceivedPartial deliver data for the same Message by passing the same MessageContext, until the endOfMessage flag is delivered or a ReceiveError occurs. All partial blocks of a single Message are delivered in order without gaps. This event does not support delivering discontiguous partial Messages.

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

- 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 deframer (see Section 8.4) 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 deframer was supplied by the application

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

8.2.3. ReceiveError

Connection -> ReceiveError<messageContext>

A ReceiveError occurs when data is received by the underlying Protocol Stack that cannot be fully retrieved or deframed, or when some other indication is received that reception has failed. Such conditions that irrevocably lead to the termination of the Connection are signaled using ConnectionError instead (see Section 10).

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

8.3. Message Receive Context

Each Received Message Context may contain metadata from protocols in the Protocol Stack; which metadata is available is Protocol Stack dependent. The following metadata values are supported:

8.3.1. ECN

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

8.3.2. Early Data

In some cases it may be valuable to know whether data was read as part of early data transfer (before connection establishment has finished). This is useful if applications need to treat early data separately, e.g., if early data has different security properties than data sent after connection establishment. In the case of TLS 1.3, client early data can be replayed maliciously (see [I-D.ietf-tls-tls13]). Thus, receivers may wish to perform additional checks for early data to ensure it is idempotent or not replayed. If TLS 1.3 is available and the recipient Message was sent as part of early data, the corresponding metadata carries a flag indicating as such. If early data is enabled, applications should check this metadata field for Messages received during connection establishment and respond accordingly.

8.3.3. Receiving Final Messages

The Received Message Context can indicate whether or not this Message is the Final Message on a Connection. For any Message that is marked as Final, the application can assume that there will be no more Messages received on the Connection once the Message has been completely delivered. This corresponds to the Final property that may be marked on a sent Message Section 7.3.5.

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

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

8.4. Receiver-side De-framing over Stream Protocols

The Receive Event is intended to be fired once per application-layer Message sent by the remote endpoint; i.e., it is a desired property of this interface that a Send at one end of a Connection maps to exactly one Receive on the other end. This is possible with Protocol Stacks that provide message boundary preservation, but is not the case over Protocol Stacks that provide a simple octet stream transport.

For preserving message boundaries over stream transports, this interface provides receiver-side de-framing. This facility is based on the observation that, since many of our 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. A Deframer allows an application to push this de-framing down into the interface, in order to transform an octet stream into a sequence of Messages.

Concretely, receiver-side de-framing allows a caller to provide the interface with a function that takes an octet stream, as provided by the underlying Protocol Stack, reads and returns a single Message of an appropriate type for the application and platform, and leaves the octet stream at the start of the next Message to deframe. It consists of a Deframer Object with a single Action, Deframe. Since the Deframer depends on the protocol used at the application layer, it is bound to the Preconnection during the pre-establishment phase:

```
Preconnection.DeframeWith(Deframer)

{messageData} := Deframer.Deframe(OctetStream, ...)
```

9. Managing Connections

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

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

At any point, the application can query Connection Properties. ~~~ ConnectionProperties := Connection.GetProperties() ~~~

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

- 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 of Communication" set to "unidirectional receive" or if a Message marked as "Final" was sent over this connection, see Section 7.3.5.

- Whether the connection can be used to receive data. A connection can not be used for reading if the connection was created with the Selection Property "Direction of Communication" set to

"unidirectional send" or if a Message marked as "Final" was received, see Section 8.3.3. The latter is only supported by certain transport protocols, e.g., by TCP as half-closed connection.

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

- For Connections that are Established, Closing, or Closed: Selection (Section 5.2) and Connection Properties (Section 9.1) of the actual protocols that were selected and instantiated. Selection Properties indicate whether or not the Connection has or offers a certain Selection Property. Note that the actually instantiated protocol stack may not match all Protocol Selection Properties that the application specified on the Preconnection. For example, a certain Protocol Selection Property that an application specified as Preferred may not actually be present in the chosen protocol stack because none of the currently available transport protocols had this feature.

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

9.1. Generic Connection Properties

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

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

9.1.1. Notification of excessive retransmissions

Type: Boolean

This property specifies whether an application considers it useful to be informed in case sent data was retransmitted more often than a certain threshold. When set to true, the effect is twofold: The application may receive events in case excessive retransmissions. In addition, the transport system considers this as a preference to use transports stacks that can provide this notification. This is not a strict requirement. If set to false, no notification of excessive retransmissions will be sent and this transport feature is ignored for protocol selection.

The recommended default is to have this option.

9.1.2. Retransmission threshold before excessive retransmission notification

Type: Integer

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

9.1.3. Notification of ICMP soft error message arrival

Type: Boolean

This property specifies whether an application considers it useful to be informed when an ICMP error message arrives that does not force termination of a connection. When set to true, received ICMP errors will be available as SoftErrors. Note that even if a protocol supporting this property is selected, not all ICMP errors will necessarily be delivered, so applications cannot rely on receiving them. Setting this option also implies a preference to prefer transports stacks that can provide this notification. If not set, no events will be sent for ICMP soft error message and this transport feature is ignored for protocol selection.

This property applies to Connections and Connection Groups. The recommended default is not to have this option.

9.1.4. Required minimum coverage of the checksum for receiving

Type: Integer

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

9.1.5. Niceness (Connection)

Type: Integer

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

9.1.6. Timeout for aborting Connection

Type: Integer

This property specifies how long to wait before aborting a Connection during establishment, or before deciding that a Connection has failed after establishment. It is given in seconds.

9.1.7. Connection group transmission scheduler

Type: Enum

This property specifies which scheduler should be used among Connections within a Connection Group, see Section 6.4. The set of schedulers can be taken from [I-D.ietf-tsvwg-sctp-ndata].

9.1.8. Maximum message size concurrent with Connection establishment

Type: Integer (read only)

This property represents the maximum Message size that can be sent before or during Connection establishment, see also Section 7.3.4. It is given in Bytes.

9.1.9. Maximum Message size before fragmentation or segmentation

Type: Integer (read only)

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

9.1.10. Maximum Message size on send

Type: Integer (read only)

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

9.1.11. Maximum Message size on receive

Type: Integer (read only)

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

9.1.12. Capacity Profile

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

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

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

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

Low Latency/Non-Interactive: The application prefers loss to latency but is not interactive. Response time should be optimized at the expense of bandwidth efficiency and delay variation when sending on this connection.Transport system implementations that map the requested capacity profile onto per-connection DSCP signaling

without multiplexing SHOULD assign a DSCP Assured Forwarding (AF21,AF22,AF23,AF24) [RFC2597] PHB; when the Connection is multiplexed, the guidelines in section 6 of [RFC7657] apply.

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

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

The Capacity Profile for a selected protocol stack may be modified on a per-Message basis using the Transmission Profile Message Property; see Section 7.3.8.

9.2. Soft Errors

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

Connection -> SoftError<>

10. Connection Termination

Close terminates a Connection after satisfying all the requirements that were specified regarding the delivery of Messages that the application has already given to the transport system. For example, if reliable delivery was requested for a Message handed over before calling Close, the transport system will ensure that this Message is

indeed delivered. If the Remote Endpoint still has data to send, it cannot be received after this call.

Connection.Close()

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

Connection -> Closed<>

Abort terminates a Connection without delivering remaining data:

Connection.Abort()

A ConnectionError can inform the application that the other side has aborted the Connection; however, there is no guarantee that an Abort will indeed be signaled.

Connection -> ConnectionError<>

11. Connection State and Ordering of Operations and Events

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

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

- Ready<> occurs when a Connection created with Initiate() or InitiateWithIdempotentData() 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.
- InitiateError<> 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 interface 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 InitiateError<> or ConnectionError<> on that connection. To ensure this ordering, Closed<> will not occur on a Connection while other events on the Connection are still locally outstanding (i.e., known to the interface and waiting to be dealt with by the application). ConnectionError<> may occur after Closed<>, but the interface must gracefully handle all cases where application ignores these errors.

12. IANA Considerations

RFC-EDITOR: Please remove this section before publication.

This document has no Actions for IANA.

13. Security Considerations

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

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.

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.

15. References

15.1. Normative References

[I-D.ietf-taps-arch]
Pauly, T., Trammell, B., Brunstrom, A., Fairhurst, G., Perkins, C., Tiesel, P., and C. Wood, "An Architecture for Transport Services", draft-ietf-taps-arch-01 (work in progress), July 2018.

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

[I-D.ietf-tls-tls13]
Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", draft-ietf-tls-tls13-28 (work in progress), March 2018.

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

[I-D.ietf-tsvwg-sctp-ndata]
Stewart, R., Tuexen, M., Loreto, S., and R. Seggelmann, "Stream Schedulers and User Message Interleaving for the Stream Control Transmission Protocol", draft-ietf-tsvwg-sctp-ndata-13 (work in progress), September 2017.

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

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

15.2. Informative References

[I-D.ietf-taps-transport-security]
Pauly, T., Perkins, C., Rose, K., and C. Wood, "A Survey of Transport Security Protocols", draft-ietf-taps-transport-security-02 (work in progress), June 2018.

[LE-PHB] Bless, R., "A Lower Effort Per-Hop Behavior (LE PHB)", draft-ietf-tsvwg-le-phb-06 (work in progress), October 2018.

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

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

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

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

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

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

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

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

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

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

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

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

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

Appendix A. Additional Properties

The interface specified by this document represents the minimal common interface to an endpoint in the transport services architecture [I-D.ietf-taps-arch], based upon that architecture and on the minimal set of transport service features elaborated in [I-D.ietf-taps-minset]. However, the interface has been designed with extension points to allow the implementation of features beyond those in the minimal common interface: Protocol Selection Properties, Path Selection Properties, and Message Properties are open sets. Implementations of the interface are free to extend these sets to provide additional expressiveness to applications written on top of them.

This appendix enumerates a few additional properties that could be used to enhance transport protocol and/or path selection, or the transmission of messages given a Protocol Stack that implements them. These are not part of the interface, and may be removed from the

final document, but are presented here to support discussion within the TAPS working group as to whether they should be added to a future revision of the base specification.

A.1. Experimental Transport Properties

The following Transport Properties might be made available in addition to those specified in Section 5.2, Section 9.1, and Section 7.3.

A.1.1. Direction of communication

Classification: Selection Property, Control Property [TODO: Discuss]

Type: Enumeration

Applicability: Preconnection, Connection (read only)

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

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

unidirectional send: The connection must support sending data.

unidirectional receive: The connection must support receiving data

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

A.1.2. Suggest a timeout to the Remote Endpoint

Classification: Selection Property

Type: Preference

Applicability: Preconnection

This property specifies whether an application considers it useful to propose a timeout until the Connection is assumed to be lost. The default is to have this option.

[EDITOR'S NOTE: For discussion of this option, see https://github.com/taps-api/drafts/issues/109]

A.1.3. Abort timeout to suggest to the Remote Endpoint

Classification: Protocol Property

Type: Integer

Applicability: Preconnection, Connection

This numeric property specifies the timeout to propose to the Remote Endpoint. It is given in seconds.

[EDITOR'S NOTE: For discussion of this property, see https://github.com/taps-api/drafts/issues/109]

A.1.4. Traffic Category

Classification: Intent

Type: Enumeration

Applicability: Preconnection

This property specifies what the application expects the dominating traffic pattern to be. Possible values are:

Query: Single request / response style workload, latency bound

Control: Long lasting low bandwidth control channel, not bandwidth bound

Stream: Stream of data with steady data rate

Bulk: Bulk transfer of large Messages, presumably bandwidth bound

The default is to not assume any particular traffic pattern. Most categories suggest the use of other intents to further describe the traffic pattern anticipated, e.g., the bulk category suggesting the use of the Message Size intents or the stream category suggesting the Stream Bitrate and Duration intents.

A.1.5. Size to be Sent or Received

Classification: Intent

Type: Integer

Applicability: Preconnection, Message

This property specifies how many bytes the application expects to send (Size to be Sent) or how many bytes the application expects to receive in response (Size to be Received).

A.1.6. Duration

Classification: Intent

Type: Integer

Applicability: Preconnection

This Intent specifies what the application expects the lifetime of a Connection to be. It is given in milliseconds.

A.1.7. Send or Receive Bit-rate

Classification: Intent

Type: Integer

Applicability: Preconnection, Message

This Intent specifies what the application expects the bit-rate of a transfer to be. It is given in Bytes per second.

On a Message, this property specifies at what bitrate the application wishes the Message to be sent. A transport system supporting this feature will not exceed the requested Send Bitrate even if flow-control and congestion control allow higher bitrates. This helps to avoid a bursty traffic pattern on busy streaming video servers.

A.1.8. Cost Preferences

Classification: Intent

Type: Enumeration

Applicability: Preconnection, Message

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

Possible values are:

No Expense: Avoid transports associated with monetary cost

Optimize Cost: Prefer inexpensive transports and accept service degradation

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

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

The default is "Balance Cost".

Appendix B. Sample API definition in Go

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

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

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

[EDITOR'S NOTE: This is early text. In the future, this section will contain backward references, which we currently avoid because things are still being moved around and names / categories etc. are changing. Also, clearly, the intention is for the full minset to be reflected by the API at some point.]

- Connect:
  "Initiate" Action.

- Listen:
  "Listen" Action.

- Specify number of attempts and/or timeout for the first establishment message:
  TODO.

- Disable MPTCP:
  TODO.

- Hand over a message to reliably transfer (possibly multiple times) before connection establishment:
  "InitiateWithIdempotentSend" Action.

- Hand over a message to reliably transfer during connection establishment:
  TODO.

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

- Timeout event when data could not be delivered for too long:
  TODO: this should probably be covered by the "ConnectionError" Event, but the text above it currently reads: "...can inform the application that the other side has aborted the Connection". In this case, it is the local side.

- Suggest timeout to the peer:
  "Suggest a timeout to the Remote Endpoint" and "Abort timeout to suggest to the Remote Endpoint" Selection property. [EDITOR'S NOTE: For discussion of this option, see https://github.com/taps-api/drafts/issues/109].

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

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

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

- Configure priority or weight for a scheduler:
  "Niceness (Connection)" property.

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

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

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

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

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

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

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

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

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

- "Reliably transfer data, with congestion control", "Reliably transfer a message, with congestion control" and "Unreliably transfer a message":

Reliability is controlled via the "Reliable Data Transfer (Message)" Message property. Transmitting data without delimiters is done by not using a Framer. The choice of congestion control is provided via the "Congestion control" property.

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

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

- Request not to delay the acknowledgement (SACK) of a message:
  Should the protocol support it, this is one of the transport features the transport system can use when an application uses the Capacity Profile Property with value "Low Latency/Interactive".

- Receive data (with no message delimiting):
  "Received" Event without using a Deframer.

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

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

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

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

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

Authors' Addresses

Brian Trammell (editor)
ETH Zurich
Gloriastrasse 35
8092 Zurich
Switzerland

Email: ietf@trammell.ch

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

Email: michawe@ifi.uio.no

Theresa Enghardt
TU Berlin
Marchstrasse 23
10587 Berlin
Germany

Email: theresa@inet.tu-berlin.de

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

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

Mirja Kuehlewind
ETH Zurich
Gloriastrasse 35
8092 Zurich
Switzerland

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

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

Email: csp@csperkins.org

Philipp S. Tiesel
TU Berlin
Marchstrasse 23
10587 Berlin
Germany

Email: philipp@inet.tu-berlin.de

Chris Wood
Apple Inc.
1 Infinite Loop
Cupertino, California 95014
United States of America

Email: cawood@apple.com