TAPS Working Group
Internet-Draft
Intended status: Informational
Expires: January 3, 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.

July 02, 2018

An Abstract Application Layer Interface to Transport Services
draft-ietf-taps-interface-01

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

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] and [I-D.ietf-taps-minset].

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 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 callback passing or other asynchronous calling conventions. The method for registering callbacks and handlers is left as an implementation detail, with the caveat that the interface for receiving Messages must require the application to invoke the Connection.Receive() Action once per Message to be received (see Section 8).

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

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

3. Interface Design Principles

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

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

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

o Asynchronous Connection establishment, transmission, and reception, allowing most application interactions with the transport layer to be Event-driven, in line with developments in modern platforms and programming languages;

- 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; and

- Atomic transmission of data, using application-assisted framing and deframing where the underlying transport does not provide these.

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.

In the following sections, we 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].

5. Pre-Establishment Phase

The pre-establishment phase allows applications to specify properties for the Connections they’re 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 transport properties (see Section 12), 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 in 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 7.6) 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 RemoteEndoint. 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 will resolve names internally, when the Initiate(), Listen(), or Rendezvous() method is called establish a Connection. The API does not need the application to resolve names, and premature name resolution can damage performance by limiting the scope for alternate path discovery during Connection establishment. The Resolve() method is, however, provided to resolve a Local Endpoint or a Remote Endpoint in cases where this is 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 (Protocol and Path Selection Properties), as well as Generic and Specific Protocol Properties for configuration of the detailed operation of the selected Protocol Stacks.

The protocol(s) and path(s) selected as candidates during Connection establishment are determined by a set of 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.

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. In case of conflicts between Protocol and Path Selection Properties, Path Selection Properties take precedence. For example, if an application indicates a preference for a specific path, but also a preference for a protocol not available on this path, the transport system will try the path first, so the Protocol Selection Property might not have an effect.

All Transport Properties used in the pre-establishment phase are collected in a TransportProperties Object that is passed to the Preconnection Object.

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

The Individual properties are then added to the TransportProperties Object.

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

Transport Properties of Preference Type, see Section 12.1.4, can use special calls to add a Property with a specific preference level, i.e, "TransportProperties.Add('some preference', avoid)" is equivalent to "TransportProperties.Avoid('some preference')"

```
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()
```

Section 12 provides a list of Transport Properties.

Note that most properties are only considered for Connection establishment and can not be changed after a Connection is established; however, they can be queried. See Section 9.

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

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 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:

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

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

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

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

6. Establishing Connections

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

6.1. Active Open: Initiate

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

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

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. After calling Initiate, no further properties may be added to the Preconnection. The Initiate() call consumes the Preconnection and creates a Connection Object. A Preconnection can only be initiated once.

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 cryptographic 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).

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() method on the Preconnection can be used to discover these bindings:

```
PreconnectionBindings := 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. 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 clone 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 group, and so on. Connections in a Connection Group share all their properties, and changing the properties on one Connection in the group changes the property for all others.

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<>
```

There is only one Protocol Property that is not entangled: niceness is kept as a separate per-Connection Property for individual Connections in the group. Niceness works as in Section 12.3.21: 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 the 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 it shares 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.

If Send is called on a Connection which has not yet been established, an Initiate Action will be implicitly performed simultaneously with the Send. Together with the Idempotent property (see Section 12.3.9), this can be used to send data during establishment for 0-RTT session resumption on Protocol Stacks that support it.

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 only issues a Send after this Event fires.

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 12.3.28) 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 Context Parameters

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 parameters for sending Messages, and can be passed to the

Send Action. Some of these parameters are properties as defined in Section 12. 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.

Message Properties share a single namespace with Transport Properties (see Section 12). This allows the specification of per-Connection Protocol Properties that can be overridden on a per-Message basis.

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.

Parameters 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 parameters 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 Context Parameters are supported:

[TODO: De-Duplicate with Properties in Section 12, find consensus on which Section to put them]

7.3.1. Lifetime

[TODO: De-Duplicate with Section 12.3.28]

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

[TODO: De-Duplicate with Section 12.3.21]

Niceness is a numeric (non-negative) value that represents an unbounded hierarchy of priorities of Messages, relative to other Messages sent over the same Connection and/or Connection Group (see Section 6.4). 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.

This encoding of the priority has a convenient property that the priority increases as both Niceness and Lifetime decrease.

7.3.3. Ordered

[TODO: De-Duplicate with Section 12.3.6]

Ordered is a boolean property. If true, this Message should be delivered after the last Message passed to the same Connection via the Send Action; if false, this Message may be delivered out of order.

7.3.4. Idempotent

[TODO: De-Duplicate with Section 12.3.9]

Idempotent is a boolean property. If true, the application-layer entity in the 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

[TODO: De-Duplicate with Section 12.3.1]

Final is a boolean property. 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 new Message will report an error.

7.3.6. Corruption Protection Length

[TODO: De-Duplicate with Section 12.3.15]

This numeric property specifies the length of the section of the Message, starting from byte 0, that the application assumes will be received 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 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. Transmission Profile

[TODO: De-Duplicate with Section 12.3.19]

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 12.3.19).

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.

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 messageData 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. 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. 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. All partial blocks of a single Message are delivered in order without gaps. This event does not support delivering discontiguous partial Messages.

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

- o the underlying Protocol Stack supports message boundary preservation, and the size of the Message is larger than the buffers available for a single message;
- o the underlying Protocol Stack does not support message boundary preservation, and the deframer (see Section 8.4) cannot determine the end of the message using the buffer space it has available; or
- o the underlying Protocol Stack does not support message boundary preservation, and no 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 the 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 and 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 streams. 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 12.3.1.

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. Setting and Querying Connection Properties

At any point, the application can query Connection Properties. It can also set per-connection Protocol Properties.

```
ConnectionProperties := Connection.GetProperties()

Connection.SetProperty(property, value)
```

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

- The status of the connection, 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 "Unidirectional Receive" or if a Message marked as "Final" was sent over this connection, see Section 12.3.1.

- 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 "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. Selection Properties of a Connection can only be queried, not set.

- For Connections that are Established, Closing, or Closed (TODO: double-check if closed belongs here): Transport Properties of the actual protocols that were selected and instantiated. These features correspond to the properties given in Section 12 and include Selection Properties and Protocol Properties.

  * 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. Selection Properties of a Connection can only be queried.

  * Protocol Properties of the protocol stack in use (see Section 12.2.2 below). These can be queried and set. Certain specific Procotol Properties may be read-only, on a protocol- and property-specific basis.

- For Connections that are Established, 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. They can only be queried.

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 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 be signaled:

```
Connection -> ConnectionError<>
```

A SoftError can inform the application about the receipt of an ICMP error message that does not force termination of the connection, if the underlying protocol stack supports access to soft errors; however, even if the underlying stack supports it, there is no guarantee that a soft error will be signaled.

```
Connection -> SoftError<>
```

11. Ordering of Operations and Events

As this interface is designed to be independent of concurrency model, the details of how exactly actions are handled, and on which threads/callbacks events are dispatched, are implementation dependent. However, the interface does provide the following guarantees about the ordering of operations:

- Received<> will never occur on a Connection before a Ready<> event on that Connection, or a ConnectionReceived<> or RendezvousDone<> containing that Connection.

- No events will occur on a Connection 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 the application ignoring these errors.

- 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).

12. Transport Properties

Transport Properties allow an application to control and introspect most aspects of the transport system and transport protocols.

Properties are structured in two ways:

- By how they influence the transport system, which leads to a classification into "Selection Properties", "Protocol Properties", "Control Properties" and "Intents".

- By the object they can be applied to: Preconnections, see Section 5.2, Connections, see Section 9, and Messages, see Section 7.3.

Because some properties can be applied or queried on multiple objects, all Transport Properties are organized within a single namespace.

Note that it is possible for a set of specified Transport Properties to be internally inconsistent, or to be inconsistent with the later use of the API by the application. Application developers can reduce inconsistency by only using the most stringent preference levels when failure to meet the property would break the application’s functionality. For example, it can set the Selection Property "Reliable Data Transfer", which is a core assumption of many application protocols, as Required. Implementations of this interface should also raise any detected errors in configuration as early as possible, to help ensure that inconsistencies are caught early in the development process.

12.1. Transport Property Types

Each Transport Property takes a value of a property-specific type.

12.1.1. Boolean

A boolean is a data type that can be either "true" or "false". Boolean transport properties should only be used for properties that can not be used in an optional way or to query the state of the transport implementation. For optional features, especially in Selection Properties, the usage of the Preference type (see Section 12.1.4) is preferred.

12.1.2. Enumeration

Enumeration types are used for transport properties that can take one value out of a limited set of choices. The representation is implementation dependent.

12.1.3. Integer

Integer types are used to represent integer numbers. The representation is implementation dependent.

12.1.4. Preference

The Preference type is used in most Selection properties on a Preconnection object to constrain Path Selection and Protocol Selection. It is a specific instance of the "Enum" type and has five different 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 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

When used on a Connection, this type becomes a (read-only) Boolean representing whether the selected transport supports the requested feature.

12.2. Transport Property Classification

Note: This section is subject to WG discussion on IETF-102.

Transport Properties - whether they apply to connections, preconnections, or messages - differ in the way they affect the transport system and protocols exposed through the transport system. The classification proposed below emphasizes two aspects of how properties affect the transport system, so applications know what to expect:

- o Whether properties affect protocols exposed through the transport system (Protocol Properties) or the transport system itself (Control Properties)

- o Whether properties have a clearly defined behavior that is likely to be invariant across implementations and environments (Protocol Properties and Control Properties) or whether the properties are interpreted by the transport system to provide a best effort service that matches the applications needs as well as possible (Intents).

Note: in I-D.ietf-taps-interface-00, we had a classification into Connection Properties and Message Properties, whereby Connection Properties where itself were sub-classified in Protocol-Selection, Path-Selection and Protocol properties.

The classification in this version of the draft emphasizes the way the property affects the transport system and protocols. It treats the aspect of whether properties are used on a connection, preconnection or message as an orthogonal dimension of classification.

The "Message Properties" from I-D.ietf-taps-interface-00 therefore have been split into "Protocol Properties" - emphasizing that they affect the protocol configurations - and "Control Properties" - emphasizing that they control the local transport system itself.

12.2.1. Selection Properties

Selection Properties influence protocol and path selection. Their value usually is or includes a Preference that constrains (in case of Require or Prohibit) or influences (Prefer, Ignore, Avoid) the selection of transport protocols and paths used.

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

Protocol Selection Properties can only be set on Preconnections, see Section 5.2. Path Selection Properties are usually used on Preconnections, but might also be used on messages to assist per-message path selection for multipath aware protocols.

12.2.2. Protocol Properties

Protocol Properties represent the configuration of the selected Protocol Stacks backing a Connection. Some properties apply generically across multiple transport protocols, while other properties only apply to specific protocols. Generic properties will be passed to the selected candidate Protocol Stack(s) to configure them before candidate Connection establishment. The default settings of these properties will vary based on the specific protocols being used and the system's configuration.

Most Protocol Properties can be set on a Preconnection during pre-establishment to preconfigure Protocol Stacks during establishment.

In order to specify Specific Protocol Properties, Transport System implementations may offer applications to attach a set of options to the Preconnection Object, associated with a specific protocol. For example, an application could specify a set of TCP Options to use if and only if TCP is selected by the system. Such properties 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.

Note that many protocol properties have a corresponding selection property which asks for a protocol providing a specific transport feature that is controlled by the protocol property.

12.2.3. Control Properties

[TODO: Discuss]

Control properties manage the local transport system behavior or request state changes in the local transport system. Depending on the protocols used, setting these properties might also influence the protocol state machine. See Section 12.3.1 for an example.

12.2.4. Intents

[TODO: Discuss]

Intents are hints to the transport system that do not directly map to a single protocol/transport feature or behavior of the transport system, but express a presumed application behavior or generic application needs.

The application can expect the transport system to take appropriate actions involving protocol selection, path selection and, setting of protocol flags. For example, if an application sets the "Capacity

Profile" to "bulk" on a Preconnection, this will likely influence path selection, DSCP flags in the IP header as well as niceness for multi-streaming connections. When using Intents, the application must not expect consistent behavior across different environments, implementations or versions of the same implementation.

12.3. Mandatory Transport Properties

The following properties are mandatory to implement in a transport system:

12.3.1. Final

See Section 7.3.5.

[TODO: Decide whether this is a property or a parameter]

12.3.2. Reliable Data Transfer (Connection)

Classification: Selection Property

Type: Preference

Applicability: Preconnection, Connection (read only)

This property specifies whether the application wishes 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 default is to enable Reliable Data Transfer.

12.3.3. Configure per-Message reliability

Classification: Selection Property

Type: Preference

Applicability: Preconnection, Connection (read only)

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 default is to not have this option.

12.3.4. Reliable Data Transfer (Message)

Classification: Protocol Property (Generic)

Type: Boolean

Applicability: Message

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 transport protocol supports partial reliability (see Section 12.3.3). Therefore, for protocols that always transfer data reliably, this property is always true and for protocols that always transfer data unreliably, this flag is always false. Changing it may generate an error.

12.3.5. Preservation of data ordering

Classification: Selection Property

Type: Preference

Applicability: Preconnection, Connection (read only)

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

12.3.6. Ordered

Classification: Protocol Property (Generic)

Type: Boolean

Applicability: Message

This property specifies that a Message should be delivered to the other side after the previous Message which was passed to the same Connection via the Send Action. It us used for protocols that support preservation of data ordering, see Section 12.3.5, but allow out-of-order delivery for certain messages.

12.3.7. 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.

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

Classification: Selection Property

Type: Preference

Applicability: Preconnection, Connection (read only)

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 12.3.9. The default is to not have this option.

12.3.9. Idempotent

Classification: Control Property

Type: Boolean

Applicability: Message

This property 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.

The application can query the maximum size of a message that can be sent idempotent, see Section 12.3.24.

12.3.10. Multistream Connections in Group

Classification: Selection Property

Type: Preference

Applicability: Preconnection, Connection (read only)

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 default is to not have this option.

12.3.11. Notification of excessive retransmissions

Classification: Control Property [TODO: Discuss]

Type: Boolean

Applicability: Preconnection, Connection

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 default is to have this option.

12.3.12. Retransmission threshold before excessive retransmission notification

Classification: Control Property [TODO: Discuss]

Type: Integer

Applicability: Preconnection, Connection

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

12.3.13. Notification of ICMP soft error message arrival

Classification: Control Property [TODO: Discuss]

Type: Boolean

Applicability: Preconnection, Connection

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 default is not to have this option.

12.3.14. Control checksum coverage on sending or receiving

Classification: Selection Property

Type: Preference

Applicability: Preconnection, Connection (read only)

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

12.3.15. Corruption Protection Length

Classification: Protocol Property (Generic)

Type: Integer

Applicability: Message

This numeric property specifies the length of the section of the Message, starting from byte 0, that the application assumes will be received 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 the 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.

12.3.16. Required minimum coverage of the checksum for receiving

Classification: Protocol Property (Generic)

Type: Integer

Applicability: Connection

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.

12.3.17. Interface Instance or Type

Classification: Selection Property

Type: Tuple (Enumeration, Preference)

Applicability: Preconnection, Connection (read only)

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

If a system supports discovery of specific interface identifiers, such as "en0" or "eth0" on Unix-style systems, an implemention should allow using these identifiers to define path preferences. 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 Section 12.3.18 or another specific property.

12.3.18. Provisioning Domain Instance or Type

Classification: Selection Property

Type: Tuple (Enumeration, Preference)

Applicability: Preconnection, Connection (read only)

Similar to interface instances and types Section 12.3.17, 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].

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 more broad than requiring specific PvD instances or interface instances, and should be preferred over those options.

12.3.19. Capacity Profile

Classification: Intent [TODO: Discuss]

Type: Enumeration

Applicability: Preconnection, Connection, Message

This property specifies the application's expectation of the dominating traffic pattern for this Connection. This implies that the transport system should optimize for the capacity profile specified. This can influence path and protocol selection. 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 stacks.

Low Latency: Response time (latency) should be optimized at the expense of bandwidth efficiency and delay variation 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.

Constant Rate: 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.

Scavenger/Bulk: The application is not interactive. It expects to send/receive a large amount of data, without any urgency. This can, for example, be used to select protocol stacks with scavenger transmission control, to signal a preference for less-than-best-effort treatment, or to assign the traffic to a lower-effort service.

12.3.20. Congestion control

Classification: Selection Property

Type: Preference

Applicability: Preconnection, Connection (read only)

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 default is that the Connection is congestion controlled.

12.3.21. Niceness

Classification: Protocol Property (Generic)

Type: Integer

Applicability: Connection, Message

This property is a numeric (non-negative) value that represents an unbounded hierarchy of priorities. It can specify the priority of a Message, relative to other Messages sent over the same Connection and/or Connection Group (see Section 6.4), or the priority of a Connection, relative to other Connections in the same Connection Group.

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.

This encoding of the priority has a convenient property that the priority increases as both Niceness and Lifetime decrease.

As noted in Section 6.4, when set on a Connection, this property is not entangled when Connections are cloned.

12.3.22. Timeout for aborting Connection

Classification: Control Property [TODO: Discuss]

Type: Integer

Applicability: Preconnection, Connection

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.

12.3.23. Connection group transmission scheduler

Classification: Protocol Property (Generic) / Control Property [TODO: Discuss]

Type: Enum

Applicability: Preconnection, Connection

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].

12.3.24. Maximum message size concurrent with Connection establishment

Classification: Protocol Property (Generic)

Type: Integer

Applicability: Connection (read only)

This property represents the maximum Message size that can be sent before or during Connection establishment, see also Section 12.3.9. It is given in Bytes. This property is read-only.

12.3.25. Maximum Message size before fragmentation or segmentation

Classification: Protocol Property (Generic)

Type: Integer

Applicability: Connection (read only)

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

12.3.26. Maximum Message size on send

Classification: Protocol Property (Generic)

Type: Integer

Applicability: Connection (read only)

This property represents the maximum Message size that can be sent. This property is read-only.

12.3.27. Maximum Message size on receive

Classification: Protocol Property (Generic)

Type: Integer

Applicability: Connection (read only)

This numeric property represents the maximum Message size that can be received. This property is read-only.

12.3.28. Lifetime

Classification: Protocol Property (Generic)

Type: Integer

Applicability: Message

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.

12.4. Optional Transport Properties

TODO: Maybe move some of the above properties here.

12.5. Experimental Transport Properties

TODO: Move Appendix A here.

13. IANA Considerations

RFC-EDITOR: Please remove this section before publication.

This document has no Actions for IANA.

14. Security Considerations

This document describes a generic API for interacting with a transport services (TAPS) system. Part of this API includes configuration details for transport security protocols, as discussed in Section 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.

15. 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.

16. References

16.1. Normative References

[I-D.ietf-taps-arch]
Pauly, T., Trammell, B., Brunstrom, A., Fairhurst, G., Perkins, C., Tiesel, P., and C. Wood, "An Architecture for Transport Services", draft-ietf-taps-arch-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-04 (work in progress), June 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>.

16.2. 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.

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

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

[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>.

[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>.

[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 12:

A.1.1. 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.1.2. 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.1.3. Request not to delay acknowledgment of Message

Classification: Selection Property

Type: Preference

Applicability: Preconnection

This property specifies whether an application considers it useful to be able to request for a Message that its acknowledgment be sent out as early as possible instead of potentially being bundled with other acknowledgments. The default is to not have this option.

[EDITOR’S NOTE: For discussion of this option, see https://github.com/taps-api/drafts/issues/90]

A.1.4. Traffic Category

Classification: Intent

Type: Enumeration

Applicability: Preconnection

This property specifies what the application expect 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 bursty traffic pattern on busy video streaming 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".

A.1.9. Immediate

Classification: Protocol Property (Generic)

Type: Boolean

Applicability: Message

This property specifies whether the caller prefers immediacy to efficient capacity usage for this Message. For example, this means that the Message should not be bundled with other Message into the same transmission by the underlying Protocol Stack.

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.

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