TAPS Working Group
Internet-Draft
Intended status: Informational
Expires: November 1, 2018

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.
April 30, 2018

An Abstract Application Layer Interface to Transport Services
draft-ietf-taps-interface-00

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 November 1, 2018.

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.pauly-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 parameters.

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.pauly-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;

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

- 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.pauly-taps-arch]. An application primarily interacts with this interface through two Objects, Preconnections and Connections. A Preconnection represents a set of parameters 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 received from a remote endpoint. 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.pauly-taps-arch].

5. Pre-Establishment Phase

The pre-establishment phase allows applications to specify parameters 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 parameters 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 parameters (see Section 5.2), and the security parameters (see Section 5.3):

```
Preconnection := NewPreconnection(LocalEndpoint,
                                  RemoteEndpoint,
                                  TransportParams,
                                  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 7.3) and deframers (see Section 8.1), 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 NAT traversal protocols (see Section 6.3).

5.2. Specifying Transport Parameters

A Preconnection Object holds parameters reflecting the application's requirements and preferences for the transport. These include protocol and path selection parameters, as well as Generic and Specific Protocol Properties for configuration of the detailed operation of the selected Protocol Stacks.

All Transport Parameters are organized within a single namespace shared with Send Parameters (see Section 7.1). These transport parameters take values of parameter-specific types.

Note that it is possible for a set of specified transport parameters 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 a preference would break the application's functionality (e.g. the Reliable Data Transfer preference, which is a core assumption of many application protocols). Implementations of this interface should also raise any detected errors in configuration as early as possible, to help ensure these inconsistencies are caught early in the development process.

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.

The type of most Protocol and Path Selection properties is "preference" and has with 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

Internally, the transport system will first exclude all protocols and paths that match a Prohibit, then only keep candidates that 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.

An implementation of this interface must provide sensible defaults for protocol and path 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.

All transport parameters used in the pre-establishment phase are collected in a TransportParameters Object that is passed to the Preconnection Object.

```
TransportParameters := NewTransportParameters()
```

The Individual parameters are then added to the TransportParameters Object. While Protocol Properties use the "add" call, Transport Preferences use special calls for the levels defined in Section 5.2.

```
TransportParameters.Add(parameter, value)

TransportParameters.Require(preference)
TransportParameters.Prefer(preference)
TransportParameters.Ignore(preference)
TransportParameters.Avoid(preference)
TransportParameters.Prohibit(preference)
```

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

```
TransportParameters := Connection.GetTransportParameters()
```

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 Parameters either by being explicitly configured via a Preconnection, or by inheriting them from an antecedent via cloning; see Section 6.4 for more.

In addition to protocol and path selection properties, the transport parameters may also contain Generic and/or Specific Protocol Properties (see Section 9.1). These properties will be passed to the selected candidate Protocol Stack(s) to configure them before candidate Connection establishment.

The following properties can be used during Protocol and Path selection:

5.2.1. Reliable Data Transfer

Type: Preference

This property specifies whether the application wishes to use a transport protocol that that provides mechanisms to help ensure that all data is received and without corruption on the other side. This also entails being notified when a Connection is closed or aborted. This property applies to Connections and Connection Groups. This is a strict requirement. The default is to enable Reliable Data Transfer.

5.2.2. Preservation of data ordering

Type: Preference

This property specifies whether the application wishes to use a transport protocol that provides mechanisms to ensure that data is

received by the application on the other end in the same order as it was sent. This property applies to Connections and Connection Groups. This is a strict requirement. The default is to preserve data ordering.

5.2.3. Configure reliability on a per-Message basis

Type: Preference

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. This is not a strict requirement. The default is to not have this option.

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

Type: Preference

This property specifies whether an application would like to supply a Message to the transport protocol before Connection establishment, which will then be reliably transferred to the other side before or during Connection establishment, potentially multiple times. See also Section 7.1.4. This is a strict requirement. The default is to not have this option.

5.2.5. Multistream Connections in Group

Type: Preference

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. This is not a strict requirement. The default is to not have this option.

5.2.6. 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.

This property applies to Connections and Connection Groups. The default is to have this option.

5.2.7. 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 default is not to have this option.

5.2.8. Control checksum coverage on sending or receiving

Type: Preference

This property specifies whether the application considers it useful to enable / disable / configure a checksum when sending data, or decide whether to require a checksum or not when receiving data. This property applies to Connections and Connection Groups. This is not a strict requirement, as it signifies a reduction in reliability. The default is full checksum coverage without being able to change it, and requiring a checksum when receiving.

5.2.9. Interface Type

Type: Tuple (Enumeration, Preference)

This property specifies which kind of access network interface, e.g., WiFi, Ethernet, or LTE, to prefer over others for this Connection, in case they are available. In general, Interface Types should be used only with the "Prefer" and "Prohibit" preference level. Specifically, using the "Require" preference level for Interface Type may limit path selection in a way that is detrimental to connectivity. The default is to use the default interface configured in the system policy. The valid values for the access network interface kinds are implementation specific.

5.2.10. Capacity Profile

Type: Enumeration

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 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 be used to select protocol stacks with scavenger transmission control, to signal a preference for less-than-best-effort treatment, and so on.

5.3. Specifying Security Parameters and Callbacks

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.pauly-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(22)    // secp256k1
SecurityParameters.AddCiphersuite(0xCCA9)   // TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY
SecurityParameters.AddSignatureAlgorithm(7) // ed25519
```

- Session cache: Used to tune cache capacity, lifetime, re-use, and eviction policies, e.g., LRU or FIFO.

```
SecurityParameters.SetSessionCacheCapacity(1024)     // 1024 elements
SecurityParameters.SetSessionCacheLifetime(24*60*60) // 24 hours
SecurityParameters.SetSessionCacheReuse(1)           // One-time use
```

- Pre-shared keying material: 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)
```

Security decisions, especially pertaining to trust, are not static. Thus, once configured, parameters must also be supplied during live handshakes. These are best handled as client-provided callbacks. Security handshake callbacks 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)
```

Like transport parameters, security parameters are inherited during cloning (see Section 6.4).

6. Establishing Connections

Before a Connection can be used for data transfer, it must be established. Establishment ends the pre-establishment phase; all transport and cryptographic parameter specification must be complete before establishment, as these parameters 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 parameters necessary for candidate selection. After calling Initiate, no further parameters may be bound to the Connection. 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 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 parameters 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 parameters may be bound to the Preconnection, and no subsequent establishment call may be made on the Preconnection.

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.

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 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 parameters may be bound 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., ICE [RFC5245], it is expected that the Local Endpoint will be configured with some method of discovering NAT bindings, e.g., a 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 or WebRTC, 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 7.1.2: 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 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 capacity allocation and each implementation is free to implement exact capacity allocation as it sees fit.

7. Sending Data

Once a Connection has been established, it can be used for sending data. Data is sent by passing a Message Object and additional parameters Section 7.1 to the Send Action on an established Connection:

Connection.Send(Message, sendParameters)

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. It may itself contain an array of octets to be transmitted in the transport protocol payload, or be transformable to an array of octets by a sender-side framer (see Section 7.3).

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.

There may also be system and Protocol Stack dependent limits on the size of a Message which can be transmitted atomically. For that reason, the Message object passed to the Send action may also be a partial Message, either representing the whole data object and information about the range of bytes to send from it, or an object referring back to the larger whole Message. The details of partial Message sending are implementation-dependent.

If Send is called on a Connection which has not yet been established, an Initiate Action will be implicitly performed simultaneously with the Send. Used together with the Idempotent property (see Section 7.1.4), this can be used to send data during establishment for 0-RTT session resumption on Protocol Stacks that support it.

Like all Actions in this interface, the Send Action is asynchronous.

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 when the Sent Event occurs is specific to the implementation and the constraints on the Protocol Stacks implied by the Connection's transport parameters. 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.

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 7.1.1) 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.

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 send parameters not consistent with the Connection's transport parameters. The SendError contains an implementation-specific reference to the Message to which it applies.

7.1. Send Parameters

The Send Action takes per-Message send parameters which control how the contents will be sent down to the underlying Protocol Stack and transmitted.

If Send Parameters should be overridden for a specific Message, an empty sent parameter Object can be acquired and all desired Send Parameters can be added to that Object. A sendParameters Object can be reused for sending multiple contents with the same properties.

```
SendParameters := NewSendParameters()
SendParameters.Add(parameter, value)
```

The Send Parameters share a single namespace with the Transport Parameters (see Section 5.2). This allows the specification of Protocol Properties that can be overridden on a per-Message basis.

Send Parameters may be inconsistent with the properties of the Protocol Stacks underlying the Connection on which a given Message is sent. For example, infinite Lifetime is not possible on a Message over a Connection not providing reliability. Sending a Message with Send Properties inconsistent with the Transport Preferences on the Connection yields an error.

The following send parameters are supported:

7.1.1. Lifetime

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.

7.1.2. Niceness

Niceness 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). It is most naturally represented as a non-negative integer. 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 inversion of normal schemes for expressing priority has a convenient property: priority increases as both Niceness and Lifetime decrease.

7.1.3. Ordered

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.

7.1.4. Idempotent

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.

7.1.5. Corruption Protection Length

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.

7.1.6. Transmission Profile

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 5.2.10).

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 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: Delay and delay variation should be minimized at the expense of bandwidth efficiency.

Scavenger/Bulk: This Message may be sent at the system's leisure. This can be used to signal a preference for less-than-best-effort treatment, to delay sending until lower-cost paths are available, and so on.

7.2. 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(Message, sendParameters)
    Connection.Send(Message, sendParameters)
)
```

7.3. 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(Message)
```

Sender-side framing is a convenience feature of the interface, for parity with receiver-side framing (see Section 8.1).

8. Receiving Data

Once a Connection is established, Messages may be received on it. The application can indicate that it is ready to receive Messages by calling Receive() on the Connection.

```
Connection.Receive(ReceiveHandler, maxLength)
```

Receive takes a ReceiveHandler, which can handle the Received Event and the ReceiveError error. Each call to Receive will result in at most one Received event being sent to the handler, though implementations may provide convenience functions to indicate readiness to receive a larger but finite number of Messages with a single call. This allows an application to provide backpressure to the transport stack when it is temporarily not ready to receive messages.

Receive also takes an optional maxLength argument, the maximum size (in bytes of data) Message 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 received as a partial Message. Note that maxLength does not guarantee that the application will receive that many bytes if they are available; the interface may return partial Messages smaller than maxLength according to implementation constraints.

Connection -> Received<Message>

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. The Message may also contain metadata from protocols in the Protocol Stack; which metadata is available is Protocol Stack dependent. In particular, when this information is available, the value of the Explicit Congestion Notification (ECN) field is contained in such metadata. 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.

The Message Object must provide some method to retrieve an octet array containing application data, corresponding to a single message within the underlying Protocol Stack's framing. See Section 8.1 for handling framing in situations where the Protocol Stack provides octet-stream transport only.

The Message Object passed to Received is complete and atomic, unless one of the following conditions holds:

- 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 8.1) 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

The Message Object passed to Received will indicate one of the following:

1. this is a complete message;

2. this is a partial message containing a section of a message with a known message boundary (made partial for local buffering reasons, either by the underlying Protocol Stack or the deframer). In this case, the Message Object passed to Received may contain the byte offset of the data in the partial Message within the full Message, an indication whether this is the last (highest-offset) partial Message in the full Message, and an optional reference to the full Message it belongs to; or

3. this is a partial message containing data with no definite message boundary, i.e. the only known message boundary is given by termination of the Connection

Note that in the absence of message boundary preservation and without deframing, the entire Connection is represented as one large message of indeterminate length.

Connection -> ReceiveError<>

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).

8.1. 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)

Message := Deframer.Deframe(OctetStream, ...)
```

9. Setting and Querying of Connection Properties

At any point, the application can set and query the properties of a Connection. Depending on the phase the Connection is in, the Connection properties will include different information.

```
ConnectionProperties := Connection.GetProperties()

Connection.SetProperties()
```

Connection properties include:

- The status of the Connection, which can be one of the following: Establishing, Established, Closing, or Closed.

- Transport Features of the protocols that conform to the Required and Prohibited Transport Preferences, which might be selected by the transport system during Establishment. These features correspond to the properties given in Section 5.2 and can only be queried.

- Transport Features of the Protocol Stacks that were selected and instantiated, once the Connection has been established. These features correspond to the properties given in Section 5.2 and can only be queried. Instead of preference levels, these features have boolean values indicating whether or not they were selected. Note that these transport features may not fully reflect the specified parameters given in the pre-establishment phase. For example, a certain Protocol Selection Property that an application specified as Preferred may not actually be present in the chosen Protocol Stack Instances because none of the currently available transport protocols had this feature.

- Protocol Properties of the Protocol Stack in use (see Section 9.1 below). These can be set or queried. Certain specific procotol queries may be read-only, on a protocol- and property-specific basis.

- Path Properties of the path(s) in use, once the Connection has been established. These properties can be derived from the local provisioning domain, measurements by the Protocol Stack, or other sources. They can only be queried.

9.1. 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. The default settings of these properties will vary based on the specific protocols being used and the system's configuration.

Note that Protocol Properties are also set during pre-establishment, as transport parameters, to preconfigure Protocol Stacks during establishment.

Generic Protocol Properties include:

- Relative niceness: This numeric property is similar to the Niceness send property (see Section 7.1.2), 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.

- Timeout for aborting Connection: This numeric 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.

- Retransmission threshold before excessive retransmission notification: This numeric property specifies after how many retransmissions to inform the application about "Excessive Retransmissions".

- Required minimum coverage of the checksum for receiving: This numeric 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.

- Connection group transmission scheduler: This enumerated property specifies which scheduler should be used among Connections within a Connection Group. It applies to Connection Groups; the set of schedulers can be taken from [I-D.ietf-tsvwg-sctp-ndata].

- Maximum message size concurrent with Connection establishment: This numeric property represents the maximum Message size that can be sent before or during Connection establishment, see also Section 7.1.4. It is given in Bytes. This property is read-only.

- Maximum Message size before fragmentation or segmentation: This numeric 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.

- Maximum Message size on send: This numeric property represents the maximum Message size that can be sent. This property is read-only.

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

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.

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. IANA Considerations

RFC-EDITOR: Please remove this section before publication.

This document has no Actions for IANA.

12. 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.

13. 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.

14. References

14.1. Normative References

[I-D.ietf-taps-minset]
Welzl, M. and S. Gjessing, "A Minimal Set of Transport Services for TAPS Systems", draft-ietf-taps-minset-03 (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.

[I-D.pauly-taps-arch]
Pauly, T., Trammell, B., Brunstrom, A., Fairhurst, G., Perkins, C., Tiesel, P., and C. Wood, "An Architecture for Transport Services", draft-pauly-taps-arch-00 (work in progress), February 2018.

[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>.

14.2. Informative References

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

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

[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>.

[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.pauly-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 options on Message send 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 parameters and 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. Protocol and Path Selection Properties

The following protocol and path selection properties might be made available in addition to those specified in Section 5.2:

- Suggest a timeout to the Remote Endpoint: This boolean property specifies whether an application considers it useful to propose a timeout until the Connection is assumed to be lost. This property applies to Connections and Connection Groups. This is not a strict requirement. The default is to have this option.

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

- Request not to delay acknowledgment of Message: This boolean property specifies whether an application considers it useful to request for Message that its acknowledgment be sent out as early as possible instead of potentially being bundled with other acknowledgments. This property applies to Connections and Connection groups. This is not a strict requirement. 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.1.1. Application Intents

Application Intents are a group of transport properties expressing what an application wants to achieve, knows, assumes or prefers regarding its communication. They are not strict requirements. In particular, they should not be used to express any Quality of Service expectations that an application might have. Instead, an application should express its intentions and its expected traffic characteristics in order to help the transport system make decisions that best match it, but on a best-effort basis. Even though Application Intents do not represent Quality of Service requirements, a transport system may use them to determine a DSCP value, e.g. similar to Table 1 in [I-D.ietf-tsvwg-rtcweb-qos].

Application Intents can influence protocol selection, protocol configuration, path selection, and endpoint selection. For example, setting the "Timeliness" Intent to "Interactive" may lead the transport system to disable the Nagle algorithm for a Connection, while setting the "Timeliness" to "Background" may lead it to setting the DSCP value to "scavenger". If the "Size to be Sent" Intent is set on an individual Message, it may influence path selection.

Specifying Application Intents is not mandatory. An application can specify any combination of Application Intents. If specified, Application Intents are defined as parameters passed to the Preconnection Object, and may influence the Connection established from that Preconnection. If a Connection is cloned to form a Connection Group, and associated Application Intents are cloned along with the other transport parameters. Some Intents have also corresponding Message Properties, similar to the properties in Section 7.1.

Application Intents can be added to this interface as Transport Preferences with the "Prefer" preference level.

A.1.1.1. Traffic Category

This Intent specifies what the application expect the dominating traffic pattern to be.

Possible Category 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.1.1.2. Size to be Sent / Received

This Intent specifies what the application expects the size of a transfer to be. It is a numeric property and given in Bytes.

A.1.1.3. Duration

This Intent specifies what the application expects the lifetime of a transfer to be. It is a numeric property and given in milliseconds.

A.1.1.4. Send / Receive Bit-rate

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

A.1.1.5. Cost Preferences

This Intent 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. This Intent can also apply to an individual Messages.

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.2. Protocol Properties

The following protocol properties might be made available in addition to those in Section 9.1:

- Abort timeout to suggest to the Remote Endpoint: 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.3. Send Parameters

The following send parameters might be made available in addition to those specified in Section 7.1:

- Immediate: Immediate is a boolean property. If true, 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.

- Send Bitrate: This numeric property in Bytes per second specifies at what bitrate the application wishes the Message to be sent. A transport supporting this feature will not exceed the requested Send Bitrate even if flow-control and congestion control allow higher bitrates. This helps to avid bursty traffic pattern on busy video streaming servers.

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