

TAPS Working Group  
Internet-Draft  
Intended status: Informational  
Expires: January 2, 2019

T. Pauly, Ed.  
Apple Inc.  
B. Trammell, Ed.  
ETH Zurich  
A. Brunstrom  
Karlstad University  
G. Fairhurst  
University of Aberdeen  
C. Perkins  
University of Glasgow  
P. Tiesel  
TU Berlin  
C. Wood  
Apple Inc.  
July 01, 2018

An Architecture for Transport Services  
draft-ietf-taps-arch-01

Abstract

This document provides an overview of the architecture of Transport Services, a system for exposing the features of transport protocols to applications. This architecture serves as a basis for Application Programming Interfaces (APIs) and implementations that provide flexible transport networking services. It defines the common set of terminology and concepts to be used in more detailed discussion of Transport Services.

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 January 2, 2019.

## Copyright Notice

Copyright (c) 2018 IETF Trust and the persons identified as the document authors. All rights reserved.

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

## Table of Contents

1. Introduction . . . . .	2
2. Background . . . . .	3
3. Design Principles . . . . .	4
3.1. Common APIs for Common Features . . . . .	4
3.2. Access to Specialized Features . . . . .	4
3.3. Scope for API and Implementation Definitions . . . . .	5
4. Transport Services Architecture and Concepts . . . . .	6
4.1. Transport Services API Concepts . . . . .	7
4.1.1. Basic Objects . . . . .	9
4.1.2. Pre-Establishment . . . . .	10
4.1.3. Establishment Actions . . . . .	11
4.1.4. Data Transfer Objects and Actions . . . . .	11
4.1.5. Event Handling . . . . .	12
4.1.6. Termination Actions . . . . .	13
4.2. Transport System Implementation Concepts . . . . .	13
4.2.1. Candidate Gathering . . . . .	14
4.2.2. Candidate Racing . . . . .	14
4.3. Protocol Stack Equivalence . . . . .	15
4.3.1. Transport Security Equivalence . . . . .	16
4.4. Message Framing, Parsing, and Serialization . . . . .	16
5. IANA Considerations . . . . .	17
6. Security Considerations . . . . .	17
7. Acknowledgements . . . . .	18
8. Informative References . . . . .	18
Authors' Addresses . . . . .	19

## 1. Introduction

Many APIs to perform transport networking have been deployed, perhaps the most widely known and imitated being the BSD `socket()` [POSIX] interface. The names and functions between these APIs are not

consistent, and vary depending on the protocol being used. For example, sending and receiving on a stream of data is conceptually the same between operating on an unencrypted Transmission Control Protocol (TCP) stream and operating on an encrypted Transport Layer Security (TLS) [I-D.ietf-tls-tls13] stream over TCP, but applications cannot use the same socket `send()` and `recv()` calls on top of both kinds of connections. Similarly, terminology for the implementation of protocols offering transport services vary based on the context of the protocols themselves. This variety can lead to confusion when trying to understand the similarities and differences between protocols, and how applications can use them effectively.

The goal of the Transport Services architecture is to provide a common, flexible, and reusable interface for transport protocols. As applications adopt this interface, they will benefit from a wide set of transport features that can evolve over time, and ensure that the system providing the interface can optimize its behavior based on the application requirements and network conditions.

This document is developed in parallel with the specification of the Transport Services API [I-D.ietf-taps-interface] and Implementation [I-D.ietf-taps-impl] documents.

## 2. Background

The Transport Services architecture is based on the survey of Services Provided by IETF Transport Protocols and Congestion Control Mechanisms [RFC8095], and the distilled minimal set of the features offered by transport protocols [I-D.ietf-taps-minset]. This work has identified common features and patterns across all transport protocols developed thus far in the IETF.

Since transport security is an increasingly relevant aspect of using transport protocols on the Internet, this architecture also considers the impact of transport security protocols on the feature set exposed by transport services [I-D.ietf-taps-transport-security].

One of the key insights to come from identifying the minimal set of features provided by transport protocols [I-D.ietf-taps-minset] was that features either require application interaction and guidance (referred to as Functional Features), or else can be handled automatically by a system implementing Transport Services (referred to as Automatable Features). Among the Functional Features, some were common across all or nearly all transport protocols, while others could be seen as features that, if specified, would only be useful with a subset of protocols, or perhaps even a single transport protocol, but would not harm the functionality of other protocols. For example, some protocols can deliver messages faster for

applications that do not require them to arrive in the order in which they were sent. However, this functionality must be explicitly allowed by the application, since reordering messages would be undesirable in many cases.

### 3. Design Principles

The goal of the Transport Services architecture is to redefine the interface between applications and transports in a way that allows the transport layer to evolve and improve without fundamentally changing the contract with the application. This requires a careful consideration of how to expose the capabilities of protocols.

There are several degrees in which a Transport Services system can offer flexibility to an application: it can provide access to multiple sets of protocols and protocol features, it can use these protocols across multiple paths that may have different performance and functional characteristics, and it can communicate with different Remote Endpoints to optimize performance. Beyond these, if the API for the system remains the same over time, new protocols and features may be added to the system's implementation without requiring changes in applications for adoption.

The following considerations were used in the design of this architecture.

#### 3.1. Common APIs for Common Features

Functionality that is common across multiple transport protocols should be accessible through a unified set of API calls. An application should be able to implement logic for its basic use of transport networking (establishing the transport, and sending and receiving data) once, and expect that implementation to continue to function as the transports change.

Any Transport Services API must allow access to the distilled minimal set of features offered by transport protocols [I-D.ietf-taps-minset].

#### 3.2. Access to Specialized Features

Since applications will often need to control fine-grained details of transport protocols to optimize their behavior and ensure compatibility with remote peers, a Transport Services system also needs to allow more specialized protocol features to be used. The interface for these specialized options should be exposed differently from the common options to ensure flexibility.

A specialized feature may be required by an application only when using a specific protocol, and not when using others. For example, if an application is using UDP, it may require control over the checksum or fragmentation behavior for UDP; if it used a protocol to frame its data over a byte stream like TCP, it would not need these options. In such cases, the API should expose the features in such a way that they take effect when a particular protocol is selected, but do not imply that only that protocol may be used if there are equivalent options.

Other specialized features, however, may be strictly required by an application and thus constrain the set of protocols that can be used. For example, if an application requires encryption of its transport data, only protocol stacks that include some transport security protocol are eligible to be used. A Transport Services API must allow applications to define such requirements and constrain the system's options. Since such options are not part of the core/common features, it should be simple for an application to modify its set of constraints and change the set of allowable protocol features without changing the core implementation.

### 3.3. Scope for API and Implementation Definitions

The Transport Services API is envisioned as the abstract model for a family of APIs that share a common way to expose transport features and encourage flexibility. The abstract API definition [I-D.ietf-taps-interface] describes this interface and is aimed at application developers.

Implementations that provide the Transport Services API [I-D.ietf-taps-impl] will vary due to system-specific support and the needs of the deployment scenario. It is expected that all implementations of Transport Services will offer the entire mandatory API, but that some features will not be functional in certain implementations. All implementations must offer sufficient APIs to use the distilled minimal set of features offered by transport protocols [I-D.ietf-taps-minset], including API support for TCP and UDP transport, but it is possible that some very constrained devices might not have, for example, a full TCP implementation.

In order to preserve flexibility and compatibility with future protocols, top-level features in the Transport Services API should avoid referencing particular transport protocols. Mappings of these API features in the Implementation document, on the other hand, must explain the ramifications of each feature on existing protocols. It is expected that the Implementation document will be updated and supplemented as new protocols and protocol features are developed.

It is important to note that neither the Transport Services API nor the Implementation document defines new protocols that require any changes on remote hosts. The Transport Services system must be deployable on one side only, as a way to allow an application to make better use of available capabilities on a system and protocol features that may be supported by peers across the network.

#### 4. Transport Services Architecture and Concepts

The concepts defined in this document are intended primarily for use in the documents and specifications that describe the Transport Services architecture and API. While the specific terminology may be used in some implementations, it is expected that there will remain a variety of terms used by running code.

The architecture divides the concepts for Transport Services into two categories:

1. API concepts, which are meant to be exposed to applications; and
2. System-implementation concepts, which are meant to be internally used when building systems that implement Transport Services.

The following diagram summarizes the top-level concepts in the architecture and how they relate to one another.

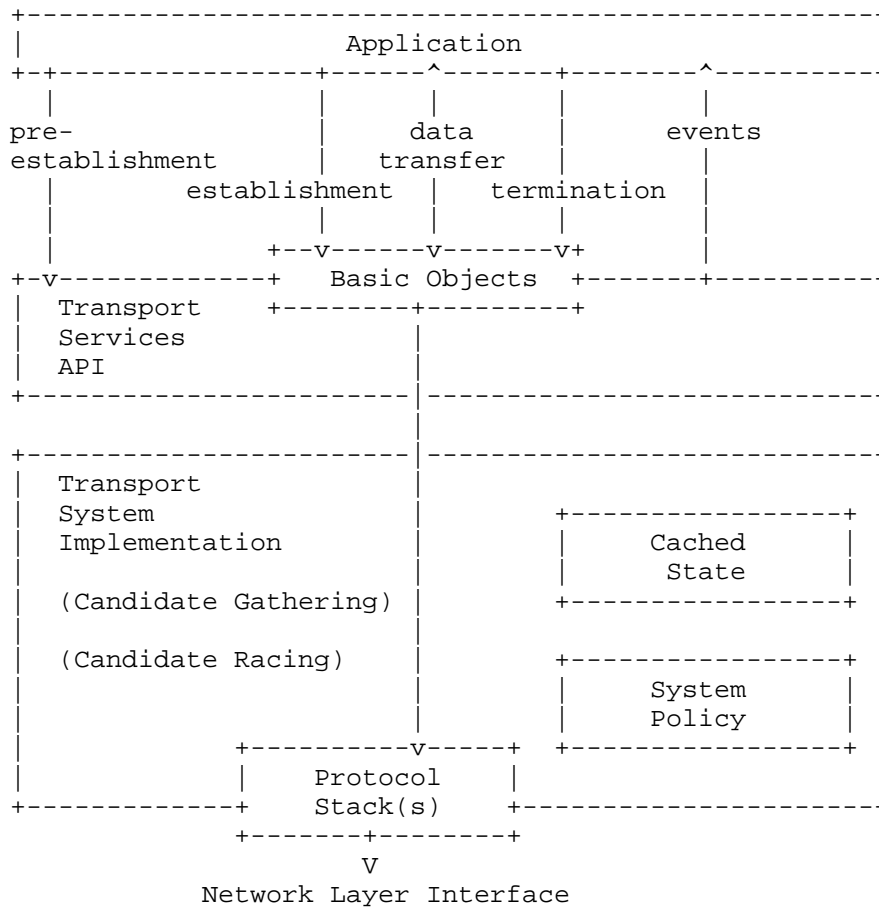


Figure 1: Concepts and Relationships in the Transport Services Architecture

#### 4.1. Transport Services API Concepts

Fundamentally, a Transport Services API needs to provide basic objects (Section 4.1.1) that allow applications to establish communication and send and receive data. These may be exposed as handles or referenced objects, depending on the language.

Beyond the basic objects, there are several high-level groups of actions that any Transport Services API must provide:

- o Pre-Establishment (Section 4.1.2) encompasses the properties that an application can pass to describe its intent, requirements, prohibitions, and preferences for its networking operations. For

any system that provides generic Transport Services, these properties should primarily be defined to apply to multiple transports. Properties may have a large impact on the rest of the aspects of the interface: they can modify how establishment occurs, they can influence the expectations around data transfer, and they determine the set of events that will be supported.

- o Establishment (Section 4.1.3) focuses on the actions that an application takes on the basic objects to prepare for data transfer.
- o Data Transfer (Section 4.1.4) consists of how an application represents data to be sent and received, the functions required to send and receive that data, and how the application is notified of the status of its data transfer.
- o Event Handling (Section 4.1.5) defines the set of properties about which an application can receive notifications during the lifetime of transport objects. Events can also provide opportunities for the application to interact with the underlying transport by querying state or updating maintenance options.
- o Termination (Section 4.1.6) focuses on the methods by which data transmission is stopped, and state is torn down in the transport.

The diagram below provides a high-level view of the actions taken during the lifetime of a connection.



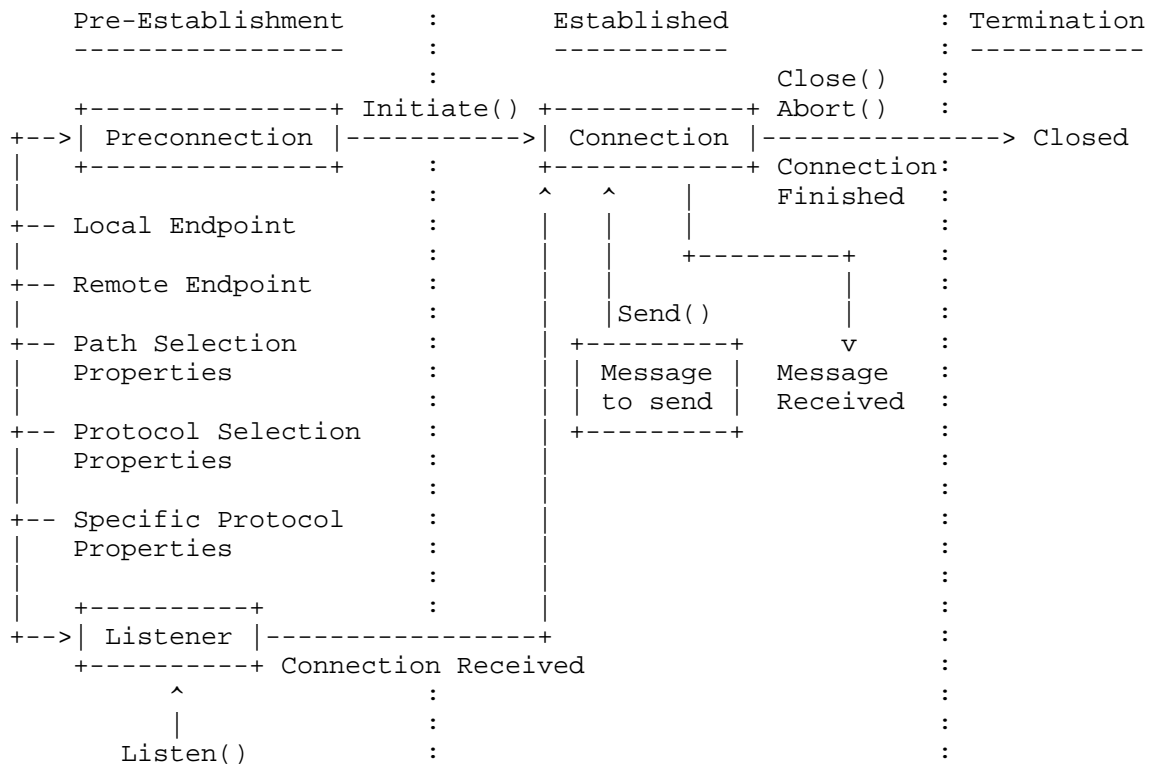


Figure 2: The lifetime of a connection

## 4.1.1.1. Basic Objects

- o Preconnection: A Preconnection object is a representation of a potential connection. It has state that describes parameters of a Connection that might exist in the future: the Local Endpoint from which that Connection will be established, the Remote Endpoint to which it will connect, and Path Selection Properties, Protocol Selection Properties, and Specific Protocol Properties that influence the choice of transport that a Connection will use. A Preconnection can be fully specified and represent a single possible Connection, or it can be partially specified such that it represents a family of possible Connections. 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.

- o **Connection:** A Connection object represents an active transport protocol instance that can send and/or receive Messages between a Local Endpoint and a Remote Endpoint. It holds state pertaining to the underlying transport protocol instance and any ongoing data transfer. This represents, for example, an active connection in a connection-oriented protocol such as TCP, or a fully-specified 5-tuple for a connectionless protocol such as UDP.
- o **Listener:** A Listener object accepts incoming transport protocol connections from Remote Endpoints and generates corresponding Connection objects. It is created from a Preconnection object that specifies the type of incoming connections it will accept.

#### 4.1.2. Pre-Establishment

- o **Endpoint:** An Endpoint represents one side of a transport connection. Endpoints can be Local Endpoints or Remote Endpoints, and respectively represent an identity that the application uses for the source or destination of a connection. An Endpoint may be specified at various levels, and an Endpoint with wider scope (such as a hostname) can be resolved to more concrete identities (such as IP addresses).
- o **Remote Endpoint:** The Remote Endpoint represents the application's name for a peer that can participate in a transport connection. For example, the combination of a DNS name for the peer and a service name/port.
- o **Local Endpoint:** The Local Endpoint represents the application's name for itself that it uses for transport connections. For example, a local IP address and port.
- o **Path Selection Properties:** The Path Selection Properties consist of the options that an application may set to influence the selection of paths between the Local Endpoint and the Remote Endpoint. These options can take the form of requirements, prohibitions, or preferences. Examples of options that may influence path selection include the interface type (such as a Wi-Fi Ethernet connection, or a Cellular LTE connection), characteristics of the path that are locally known like Maximum Transmission Unit (MTU) or discovered like Path MTU (PMTU), or predicted based on cached information like expected throughput or latency.
- o **Protocol Selection Properties:** The Protocol Selection Properties consist of the options that an application may set to influence the selection of transport protocol, or to configure the behavior of generic transport protocol features. These options can take

the form of requirements, prohibitions, and preferences. Examples include reliability, service class, multipath support, and fast open support.

- o **Specific Protocol Properties:** The Specific Protocol Properties refer to the subset of Protocol Properties options that apply to a single protocol (transport protocol, IP, or security protocol). The presence of such Properties does not necessarily require that a specific protocol must be used when a Connection is established, but that if this protocol is employed, a particular set of options should then be used..

#### 4.1.3. Establishment Actions

- o **Initiate:** The primary action that an application can take to create a Connection to a Remote Endpoint, and prepare any required local or remote state to be able to send and/or receive Messages. For some protocols, this may initiate a client-to-server style handshake; for other protocols, this may just establish local state. The process of identifying options for connecting, such as resolution of the Remote Endpoint, occurs in response the Initiate call.
- o **Listen:** The action of marking a Listener as willing to accept incoming Connections. The Listener will then create Connection objects as incoming connections are accepted (Section 4.1.5).
- o **Rendezvous:** The action of establishing a peer-to-peer connection with a Remote Endpoint. It simultaneously attempts to initiate a connection to a Remote Endpoint whilst listening for an incoming connection from that endpoint. This corresponds, for example, to a TCP simultaneous open [RFC0793]. The process of identifying options for the connection, such as resolution of the Remote Endpoint, occurs during the Rendezvous call. If successful, the rendezvous call returns a Connection object to represent the established peer-to-peer connection.

#### 4.1.4. Data Transfer Objects and Actions

- o **Message:** A Message object is a unit of data that can be represented as bytes that can be transferred between two endpoints over a transport connection. The bytes within a Message are assumed to be ordered within the Message. If an application does not care about the order in which a peer receives two distinct spans of bytes, those spans of bytes are considered independent Messages. If a received Message is incomplete or corrupted, it may or may not be usable by certain applications. Boundaries of a Message may or may not be understood or transmitted by transport

protocols. Specifically, what one application considers to be two Messages sent on a stream-based transport may be treated as a single Message by the application on the other side.

- o Send: The action to transmit a Message or partial Message over a Connection to a Remote Endpoint. The interface to Send may include options specific to how the Message's content is to be sent. Status of the Send operation may be delivered back to the application in an event (Section 4.1.5).
- o Receive: An action that indicates that the application is ready to asynchronously accept a Message over a Connection from a Remote Endpoint, while the Message content itself will be delivered in an event (Section 4.1.5). The interface to Receive may include options specific to the Message that is to be delivered to the application.

#### 4.1.5. Event Handling

This list of events that can be delivered to an application is not exhaustive, but gives the top-level categories of events. The API may expand this list.

- o Connection Ready: Signals to an application that a given Connection is ready to send and/or receive Messages. If the Connection relies on handshakes to establish state between peers, then it is assumed that these steps have been taken.
- o Connection Finished: Signals to an application that a given Connection is no longer usable for sending or receiving Messages. This should deliver an error to the application that describes the nature of the termination.
- o Connection Received: Signals to an application that a given Listener has passively received a Connection.
- o Message Received: Delivers received Message content to the application, based on a Receive action. This may include an error if the Receive action cannot be satisfied due to the Connection being closed.
- o Message Sent: Notifies the application of the status of its Send action. This may be an error if the Message cannot be sent, or an indication that Message has been processed by the protocol stack.
- o Path Properties Changed: Notifies the application that some property of the Connection has changed that may influence how and where data is sent and/or received.

#### 4.1.6. Termination Actions

- o Close: The action an application may take on a Connection to indicate that it no longer intends to send data, is no longer willing to receive data, and that the protocol should signal this state to the remote endpoint if applicable.
- o Abort: The action the application may take on a Connection to indicate a Close, but with the additional indication that the transport system should not attempt to deliver any outstanding data.

#### 4.2. Transport System Implementation Concepts

The Transport System Implementation Concepts define the set of objects used internally to a system or library to provide the functionality required to provide a transport service across a network, as required by the abstract interface.

- o Connection Group: A set of Connections that share properties. For multiplexing transport protocols, the Connection Group defines the set of Connections that can be multiplexed together.
- o Path: Represents an available set of properties that a Local Endpoint may use to send or receive packets with a Remote Endpoint.
- o Protocol Instance: A single instance of one protocol, including any state it has necessary to establish connectivity or send and receive Messages.
- o Protocol Stack: A set of Protocol Instances (including relevant application, security, transport, or Internet protocols) that are used together to establish connectivity or send and receive Messages. A single stack may be simple (a single transport protocol instance over IP), or complex (multiple application protocol streams going through a single security and transport protocol, over IP; or, a multi-path transport protocol over multiple transport sub-flows).
- o Candidate Path: One path that is available to an application and conforms to the Path Selection Properties and System Policy. Candidate Paths are identified during the gathering phase (Section 4.2.1) and may be used during the racing phase (Section 4.2.2).
- o Candidate Protocol Stack: One protocol stack that may be used by an application for a connection, of which there may be several.

Candidate Protocol Stacks are identified during the gathering phase (Section 4.2.1) and may be started during the racing phase (Section 4.2.2).

- o System Policy: Represents the input from an operating system or other global preferences that can constrain or influence how an implementation will gather candidate paths and protocol stacks (Section 4.2.1) and race the candidates during establishment (Section 4.2.2). Specific aspects of the System Policy may apply to all Connections, or only certain ones depending on the runtime context and properties of the Connection.
- o Cached State: The state and history that the implementation keeps for each set of associated endpoints that have been used previously. This can include DNS results, TLS session state, previous success and quality of transport protocols over certain paths.

#### 4.2.1. Candidate Gathering

- o Path Selection: Path Selection represents the act of choosing one or more paths that are available to use based on the Path Selection Properties provided by the application, and a Transport Services system's policies and heuristics.
- o Protocol Selection: Protocol Selection represents the act of choosing one or more sets of protocol options that are available to use based on the Protocol Properties provided by the application, and a Transport Services system's policies and heuristics.

#### 4.2.2. Candidate Racing

- o Protocol Option Racing: Protocol Racing is the act of attempting to establish, or scheduling attempts to establish, multiple Protocol Stacks that differ based on the composition of protocols or the options used for protocols.
- o Path Racing: Path Racing is the act of attempting to establish, or scheduling attempts to establish, multiple Protocol Stacks that differ based on a selection from the available Paths.
- o Endpoint Racing: Endpoint Racing is the act of attempting to establish, or scheduling attempts to establish, multiple Protocol Stacks that differ based on the specific representation of the Remote Endpoint and the Local Endpoint, such as IP addresses resolved from a DNS hostname.

#### 4.3. Protocol Stack Equivalence

The Transport Services architecture defines a mechanism that allows applications to easily use different network paths and Protocol Stacks. Transitioning between different Protocol Stacks may in some cases be controlled by properties that only change when application code is updated. For example, an application may enable the use of a multipath or multistreaming transport protocol by modifying the properties in its Pre-Connection configuration. In some cases, however, the Transport Services system will be able to automatically change Protocol Stacks without an update to the application, either by selecting a new stack entirely, or racing multiple candidate Protocol Stacks during connection establishment. This functionality can be a powerful driver of new protocol adoption, but must be constrained carefully to avoid unexpected behavior that can lead to functional or security problems.

If two different Protocol Stacks can be safely swapped, or raced in parallel (see Section 4.2.2), then they are considered to be "equivalent". Equivalent Protocol Stacks must meet the following criteria:

1. Both stacks must offer the same interface to the application for connection establishment and data transmission. For example, if one Protocol Stack has UDP as the top-level interface to the application, then it is not equivalent to a Protocol Stack that runs TCP as the top-level interface. Among other differences, the UDP stack would allow an application to read out message boundaries based on datagrams sent from the Remote Endpoint, whereas TCP does not preserve message boundaries on its own.
2. Both stacks must offer the same transport services, as required by the application. For example, if an application specifies that it requires reliable transmission of data, then a Protocol Stack using UDP without any reliability layer on top would not be allowed to replace a Protocol Stack using TCP. However, if the application does not require reliability, then a Protocol Stack that adds unnecessary reliability might be allowed as an equivalent Protocol Stack as long as it does not conflict with any other application-requested properties.
3. Both stacks must offer the same security properties. See the security protocol equivalence section below for further discussion (Section 4.3.1).

#### 4.3.1. Transport Security Equivalence

The inclusion of transport security protocols [I-D.ietf-taps-transport-security] in a Protocol Stack adds extra restrictions to Protocol Stack equivalence. Security features and properties, such as cryptographic algorithms, peer authentication, and identity privacy vary across security protocols, and across versions of security protocols. Protocol equivalence should not be assumed for different protocols or protocol versions, even if they offer similar application configuration options.

To ensure that security protocols are not incorrectly swapped, Transport Services systems should only automatically generate equivalent Protocol Stacks when the transport security protocols within the stacks are identical. Specifically, a system should consider protocols identical only if they are of the same type and version. For example, the same version of TLS running over two different transport protocol stacks may be considered equivalent, whereas TLS 1.2 and TLS 1.3 [I-D.ietf-tls-tls13] should not be considered equivalent.

#### 4.4. Message Framing, Parsing, and Serialization

While some transports expose a byte stream abstraction, most higher level protocols impose some structure onto that byte stream. That is, the higher level protocol operates in terms of messages, protocol data units (PDUs), rather than using unstructured sequences of bytes, with each message being processed in turn. Protocols are specified in terms of state machines acting on semantic messages, with parsing the byte stream into messages being a necessary annoyance, rather than a semantic concern. Accordingly, the Transport Services architecture exposes messages as the primary abstraction. Protocols that deal only in byte streams, such as TCP, represent their data in each direction as a single, long message. When framing protocols are placed on top of byte streams, the messages used in the API represent the framed messages within the stream.

Providing a message-based abstraction also provides:

- o the ability to associate deadlines with messages, for transports that care about timing;
- o the ability to provide control of reliability, choosing what messages to retransmit in the event of packet loss, and how best to make use of the data that arrived;
- o the ability to manage dependencies between messages, when some messages may not be delivered due to either packet loss or missing



a deadline, in particular the ability to avoid (re-)sending data that relies on a previous transmission that was never received.

All require explicit message boundaries, and application-level framing of messages, to be effective. Once a message is passed to the transport, it can not be cancelled or paused, but prioritization as well as lifetime and retransmission management will provide the protocol stack with all needed information to send the messages as quickly as possible without blocking transmission unnecessarily. The transport services architecture facilitates this by handling messages, with known identity (sequence numbers, in the simple case), lifetimes, niceness, and antecedents.

Transport protocols such as SCTP provide a message-oriented API that has similar features to those we describe. Other transports, such as TCP, do not. To support a message oriented API, while still being compatible with stream-based transport protocols, implementations of the transport services architecture should provide APIs for framing and de-framing messages. That is, we push message framing down into the transport services API, allowing applications to send and receive complete messages. This is backwards compatible with existing protocols and APIs, since the wire format of messages does not change, but gives the protocol stack additional information to allow it to make better use of modern transport services.

## 5. IANA Considerations

RFC-EDITOR: Please remove this section before publication.

This document has no actions for IANA.

## 6. Security Considerations

The Transport Services architecture does not recommend use of specific security protocols or algorithms. Its goal is to offer ease of use for existing protocols by providing a generic security-related interface. Each provided interface mimics an existing protocol-specific interface provided by supported security protocols. For example, trust verification callbacks are common parts of TLS APIs. Transport Services APIs will expose similar functionality.

Clients must take care to use security APIs appropriately. In cases where clients use said interface to provide sensitive keying material, e.g., access to private keys or copies of pre-shared keys (PSKs), key use must be validated. For example, clients should not use PSK material created for the Encapsulating Security Protocol (ESP, part of IPsec) [RFC4303] with QUIC, and clients must not use private keys intended for server authentication as a keys for client

authentication. Moreover, unlike certain transport features such as TCP Fast Open (TFO) [RFC7413] or Explicit Congestion Notification (ECN) [RFC3168] which can fall back to standard configurations, Transport Services systems must not permit fallback for security protocols. For example, if a client requests TLS, yet TLS or the desired version are not available, its connection must fail. Clients are responsible for implementing protocol or version fallback using a Transport Services API if so desired.

## 7. Acknowledgements

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

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.

## 8. Informative References

### [I-D.ietf-taps-impl]

Brunstrom, A., Pauly, T., Enghardt, T., Grinnemo, K., Jones, T., Tiesel, P., Perkins, C., and M. Welzl, "Implementing Interfaces to Transport Services", draft-ietf-taps-impl-00 (work in progress), May 2018.

### [I-D.ietf-taps-interface]

Trammell, B., Welzl, M., Enghardt, T., Fairhurst, G., Kuehlewind, M., Perkins, C., Tiesel, P., and C. Wood, "An Abstract Application Layer Interface to Transport Services", draft-ietf-taps-interface-00 (work in progress), April 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-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.
- [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.
- [POSIX] "IEEE Std. 1003.1-2008 Standard for Information Technology -- Portable Operating System Interface (POSIX). Open group Technical Standard: Base Specifications, Issue 7", n.d..
- [RFC0793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, DOI 10.17487/RFC0793, September 1981, <<https://www.rfc-editor.org/info/rfc793>>.
- [RFC3168] Ramakrishnan, K., Floyd, S., and D. Black, "The Addition of Explicit Congestion Notification (ECN) to IP", RFC 3168, DOI 10.17487/RFC3168, September 2001, <<https://www.rfc-editor.org/info/rfc3168>>.
- [RFC4303] Kent, S., "IP Encapsulating Security Payload (ESP)", RFC 4303, DOI 10.17487/RFC4303, December 2005, <<https://www.rfc-editor.org/info/rfc4303>>.
- [RFC7413] Cheng, Y., Chu, J., Radhakrishnan, S., and A. Jain, "TCP Fast Open", RFC 7413, DOI 10.17487/RFC7413, December 2014, <<https://www.rfc-editor.org/info/rfc7413>>.
- [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>>.

## Authors' Addresses

Tommy Pauly (editor)  
Apple Inc.  
One Apple Park Way  
Cupertino, California 95014  
United States of America  
  
Email: [tpauly@apple.com](mailto:tpauly@apple.com)

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

Email: [ietf@trammell.ch](mailto:ietf@trammell.ch)

Anna Brunstrom  
Karlstad University  
Universitetsgatan 2  
651 88 Karlstad  
Sweden

Email: [anna.brunstrom@kau.se](mailto:anna.brunstrom@kau.se)

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

Email: [gorry@erg.abdn.ac.uk](mailto:gorry@erg.abdn.ac.uk)  
URI: <http://www.erg.abdn.ac.uk/>

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

Email: [csp@csp Perkins.org](mailto:csp@csp Perkins.org)

Philipp S. Tiesel  
TU Berlin  
Marchstrasse 23  
10587 Berlin  
Germany

Email: [philipp@inet.tu-berlin.de](mailto:philipp@inet.tu-berlin.de)

Chris Wood  
Apple Inc.  
One Apple Park Way  
Cupertino, California 95014  
United States of America

Email: cawood@apple.com

TAPS Working Group  
Internet-Draft  
Intended status: Standards Track  
Expires: 7 July 2022

T. Pauly, Ed.  
Apple Inc.  
B. Trammell, Ed.  
Google Switzerland GmbH  
A. Brunstrom  
Karlstad University  
G. Fairhurst  
University of Aberdeen  
C. Perkins  
University of Glasgow  
3 January 2022

An Architecture for Transport Services  
draft-ietf-taps-arch-12

Abstract

This document describes an architecture for exposing transport protocol features to applications for network communication, a Transport Services system. The Transport Services Application Programming Interface (API) is based on an asynchronous, event-driven interaction pattern. This API uses messages for representing data transfer to applications, and describes how implementations can use multiple IP addresses, multiple protocols, and multiple paths, and provide multiple application streams. This document further defines common terminology and concepts to be used in definitions of a Transport Service API and a Transport Services implementation.

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 7 July 2022.

## Copyright Notice

Copyright (c) 2022 IETF Trust and the persons identified as the document authors. All rights reserved.

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

## Table of Contents

1. Introduction . . . . .	3
1.1. Background . . . . .	4
1.2. Overview . . . . .	4
1.3. Specification of Requirements . . . . .	5
2. API Model . . . . .	5
2.1. Event-Driven API . . . . .	7
2.2. Data Transfer Using Messages . . . . .	8
2.3. Flexible Implementation . . . . .	8
3. API and Implementation Requirements . . . . .	9
3.1. Provide Common APIs for Common Features . . . . .	10
3.2. Allow Access to Specialized Features . . . . .	11
3.3. Select Equivalent Protocol Stacks . . . . .	12
3.4. Maintain Interoperability . . . . .	13
4. Transport Services Architecture and Concepts . . . . .	13
4.1. Transport Services API Concepts . . . . .	14
4.1.1. Endpoint Objects . . . . .	16
4.1.2. Connections and Related Objects . . . . .	16
4.1.3. Pre-Establishment . . . . .	18
4.1.4. Establishment Actions . . . . .	18
4.1.5. Data Transfer Objects and Actions . . . . .	19
4.1.6. Event Handling . . . . .	20
4.1.7. Termination Actions . . . . .	21
4.1.8. Connection Groups . . . . .	21
4.2. Transport Services Implementation . . . . .	22
4.2.1. Candidate Gathering . . . . .	23
4.2.2. Candidate Racing . . . . .	23
4.2.3. Separating Connection Contexts . . . . .	24
5. IANA Considerations . . . . .	24
6. Security and Privacy Considerations . . . . .	25
7. Acknowledgements . . . . .	26
8. References . . . . .	26
8.1. Normative References . . . . .	26

8.2. Informative References . . . . .	26
Authors' Addresses . . . . .	28

## 1. Introduction

Many application programming interfaces (APIs) to perform transport networking have been deployed, perhaps the most widely known and imitated being the BSD Socket [POSIX] interface (Socket API). The naming of objects and functions across these APIs is not consistent, and varies depending on the protocol being used. For example, sending and receiving streams of data is conceptually the same for both an unencrypted Transmission Control Protocol (TCP) stream and operating on an encrypted Transport Layer Security (TLS) [RFC8446] stream over TCP, but applications cannot use the same socket send() and recv() calls on top of both kinds of connections. Similarly, terminology for the implementation of transport protocols varies based on the context of the protocols themselves: terms such as "flow", "stream", "message", and "connection" can take on many different meanings. This variety can lead to confusion when trying to understand the similarities and differences between protocols, and how applications can use them effectively.

The goal of the Transport Services architecture is to provide a flexible and reusable architecture that provides a common interface for transport protocols. As applications adopt this interface, they will benefit from a wide set of transport features that can evolve over time, and ensure that the system providing the interface can optimize its behavior based on the application requirements and network conditions, without requiring changes to the applications. This flexibility enables faster deployment of new features and protocols. It can also support applications by offering racing mechanisms (attempting multiple IP addresses, protocols, or network paths in parallel), which otherwise need to be implemented in each application separately (see Section 4.2.2).

This document was developed in parallel with the specification of the Transport Services API [I-D.ietf-taps-interface] and implementation guidelines [I-D.ietf-taps-impl]. Although following the Transport Services architecture does not require that all APIs and implementations are identical, a common minimal set of features represented in a consistent fashion will enable applications to be easily ported from one system to another.



### 1.1. Background

The Transport Services architecture is based on the survey of services provided by IETF transport protocols and congestion control mechanisms [RFC8095], and the distilled minimal set of the features offered by transport protocols [RFC8923]. These documents identified common features and patterns across all transport protocols developed thus far in the IETF.

Since transport security is an increasingly relevant aspect of using transport protocols on the Internet, this architecture also considers the impact of transport security protocols on the feature-set exposed by Transport Services [RFC8922].

One of the key insights to come from identifying the minimal set of features provided by transport protocols [RFC8923] was that features either require application interaction and guidance (referred to in that document as Functional or Optimizing Features), or else can be handled automatically by a system implementing Transport Services (referred to as Automatable Features). Among the identified Functional and Optimizing Features, some were common across all or nearly all transport protocols, while others could be seen as features that, if specified, would only be useful with a subset of protocols, but would not harm the functionality of other protocols. For example, some protocols can deliver messages faster for applications that do not require messages to arrive in the order in which they were sent. However, this functionality needs to be explicitly allowed by the application, since reordering messages would be undesirable in many cases.

### 1.2. Overview

This document describes the Transport Services architecture in three sections:

- \* Section 2 describes how the API model of Transport Services architecture differs from traditional socket-based APIs. Specifically, it offers asynchronous event-driven interaction, the use of messages for data transfer, and the flexibility to use different transport protocols and paths without requiring major changes to the application.
- \* Section 3 explains the fundamental requirements for a Transport Services system. These principles are intended to make sure that transport protocols can continue to be enhanced and evolve without requiring significant changes by application developers.

- \* Section 4 presents a diagram showing the Transport Services architecture and defines the concepts that are used by both the API [I-D.ietf-taps-interface] and implementation guidelines [I-D.ietf-taps-impl]. The Preconnection allows applications to configure Connection Properties.
- \* Section 4 also presents how an abstract Connection is used to select a transport protocol instance such as TCP, UDP, or another transport. The Connection represents an object that can be used to send and receive messages.

### 1.3. Specification of Requirements

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

## 2. API Model

The traditional model of using sockets for networking can be represented as follows:

- \* Applications create connections and transfer data using the Socket API.
- \* The Socket API provides the interface to the implementations of TCP and UDP (typically implemented in the system's kernel).
- \* TCP and UDP in the kernel send and receive data over the available network-layer interfaces.
- \* Sockets are bound directly to transport-layer and network-layer addresses, obtained via a separate resolution step, usually performed by a system-provided stub resolver.

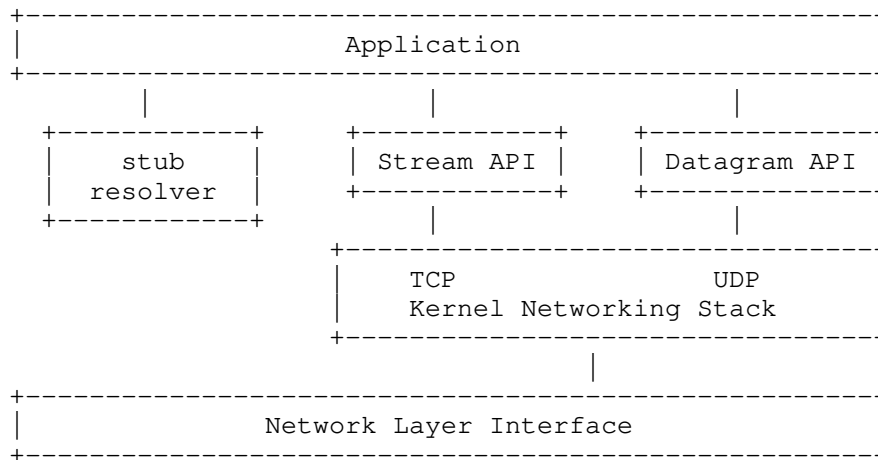


Figure 1: Socket API Model

The Transport Services architecture evolves this general model of interaction, to both modernize the API surface presented to applications by the transport layer and to enrich the capabilities of the implementation below the API.

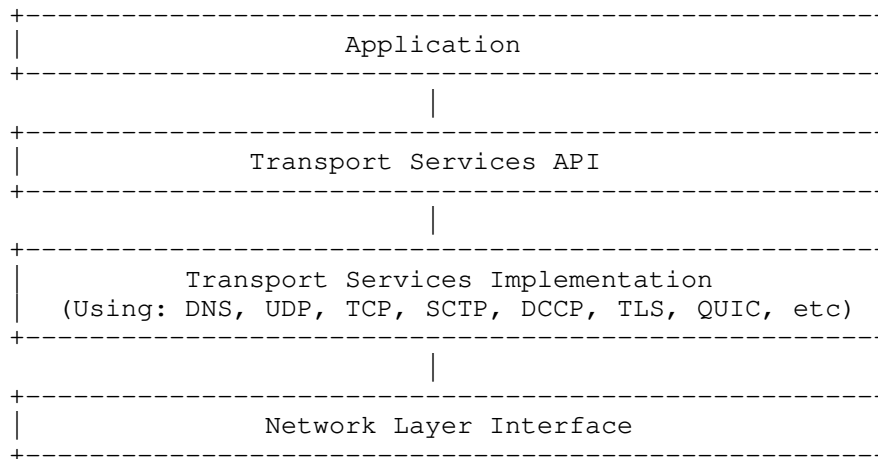


Figure 2: Transport Services API Model

The Transport Services API [I-D.ietf-taps-interface] defines the interface for an application to create Connections and transfer data. It combines interfaces for multiple interaction patterns into a unified whole. By combining name resolution with connection establishment and data transfer in a single API, it allows for more flexible implementations to provide path and transport protocol agility on the application's behalf.

The Transport Services implementation [I-D.ietf-taps-impl] implements the transport layer protocols and other functions needed to send and receive data. It is responsible for mapping the API to a specific available transport protocol stack and managing the available network interfaces and paths.

There are key differences between the Transport Services architecture and the architecture of the Socket API: the API of the Transport Services architecture is asynchronous and event-driven; it uses messages for representing data transfer to applications; and it describes how implementations can use multiple IP addresses, multiple protocols, multiple paths, and provide multiple application streams.

## 2.1. Event-Driven API

Originally, the Socket API presented a blocking interface for establishing connections and transferring data. However, most modern applications interact with the network asynchronously. Emulation of an asynchronous interface using the Socket API generally uses a try-and-fail model. If the application wants to read, but data has not yet been received from the peer, the call to read will fail. The application then waits and can try again later. In contrast to the Socket API, all interaction using the Transport Services API is expected to be asynchronous and use an event-driven model (see Section 4.1.6). For example, an application first issues a call to receive new data from the connection. When delivered data becomes available, this data is delivered to the application using asynchronous events that contain the data. Error handling is also asynchronous; a failure to send data results in an asynchronous error event.

This API also delivers events regarding the lifetime of a connection and changes in the available network links, which were not previously made explicit in the Socket API.

Using asynchronous events allows for a more natural interaction model when establishing connections and transferring data. Events in time more closely reflect the nature of interactions over networks, as opposed to how the Socket API represent network resources as file system objects that may be temporarily unavailable.

Separate from events, callbacks are also provided for asynchronous interactions with the API not directly related to events on the network or network interfaces.

## 2.2. Data Transfer Using Messages

The Socket API provides a message interface for datagram protocols like UDP, but provides an unstructured stream abstraction for TCP. While TCP has the ability to send and receive data as a byte-stream, most applications need to interpret structure within this byte-stream. For example, HTTP/1.1 uses character delimiters to segment messages over a byte-stream [RFC7230]; TLS record headers carry a version, content type, and length [RFC8446]; and HTTP/2 uses frames to segment its headers and bodies [RFC7540].

The Transport Services API represents data as messages, so that it more closely matches the way applications use the network. Providing a message-based abstraction provides many benefits, such as:

- \* the ability to associate deadlines with messages, for applications that care about timing;
- \* the ability control reliability, which messages to retransmit when there is packet loss, and how best to make use of the data that arrived;
- \* the ability to automatically assign messages and connections to underlying transport connections to utilize multi-streaming and pooled connections.

Allowing applications to interact with messages is backwards-compatible with existing protocols and APIs because it does not change the wire format of any protocol. Instead, it gives the protocol stack additional information to allow it to make better use of modern transport services, while simplifying the application's role in parsing data. For protocols which natively use a streaming abstraction, framers (Section 4.1.5) bridge the gap between the two abstractions.

## 2.3. Flexible Implementation

The Socket API for protocols like TCP is generally limited to connecting to a single address over a single interface. It also presents a single stream to the application. Software layers built upon this API often propagate this limitation of a single-address single-stream model. The Transport Services architecture is designed:

- \* to handle multiple candidate endpoints, protocols, and paths;
- \* to support candidate protocol racing to select the most optimal stack in each situation;
- \* to support multipath and multistreaming protocols;
- \* to provide state caching and application control over it.

A Transport Services implementation is intended to be flexible at connection establishment time, considering many different options and trying to select the most optimal combinations by racing them and measuring the results (see Section 4.2.1 and Section 4.2.2). This requires applications to provide higher-level endpoints than IP addresses, such as hostnames and URLs, which are used by a Transport Services implementation for resolution, path selection, and racing. An implementation can further implement fallback mechanisms if connection establishment of one protocol fails or performance is detected to be unsatisfactory.

Information used in connection establishment (e.g. cryptographic resumption tokens, information about usability of certain protocols on the path, results of racing in previous connections) are cached in the Transport Services implementation. Applications have control over whether this information is used for a specific establishment, in order to allow tradeoffs between efficiency and linkability.

Flexibility after connection establishment is also important. Transport protocols that can migrate between multiple network-layer interfaces need to be able to process and react to interface changes. Protocols that support multiple application-layer streams need to support initiating and receiving new streams using existing connections.

### 3. API and Implementation Requirements

A goal of the Transport Services architecture is to redefine the interface between applications and transports in a way that allows the transport layer to evolve and improve without fundamentally changing the contract with the application. This requires a careful consideration of how to expose the capabilities of protocols. This architecture also encompasses system policies that can influence and inform how transport protocols use a network path or interface.

There are several ways the Transport Services system can offer flexibility to an application: it can provide access to transport protocols and protocol features; it can use these protocols across multiple paths that could have different performance and functional

characteristics; and it can communicate with different remote systems to optimize performance, robustness to failure, or some other metric. Beyond these, if the Transport Services API remains the same over time, new protocols and features can be added to the Transport Services implementation without requiring changes in applications for adoption. Similarly, this can provide a common basis for utilizing information about a network path or interface, enabling evolution below the transport layer.

The normative requirements described in this section allow Transport Services APIs and Transport Services implementation to provide this functionality without causing incompatibility or introducing security vulnerabilities.

### 3.1. Provide Common APIs for Common Features

Any functionality that is common across multiple transport protocols SHOULD be made accessible through a unified set of calls using the Transport Services API. As a baseline, any Transport Services API SHOULD allow access to the minimal set of features offered by transport protocols [RFC8923].

An application can specify constraints and preferences for the protocols, features, and network interfaces it will use via Properties. Properties are used by an application to declare its preferences for how the transport service should operate at each stage in the lifetime of a connection. Transport Properties are subdivided into Selection Properties, which specify which paths and protocol stacks can be used and are preferred by the application; Connection Properties, which inform decisions made during connection establishment and fine-tune the established connection; and Message Properties, set on individual Messages.

It is RECOMMENDED that the Transport Services API offers properties that are common to multiple transport protocols. This enables a Transport Services implementation to appropriately select between protocols that offer equivalent features. Similarly, it is RECOMMENDED that the Properties offered by the Transport Services API are applicable to a variety of network layer interfaces and paths, which permits racing of different network paths without affecting the applications using the API. Each is expected to have a default value.

It is RECOMMENDED that the default values for Properties are selected to ensure correctness for the widest set of applications, while providing the widest set of options for selection. For example, since both applications that require reliability and those that do not require reliability can function correctly when a protocol

provides reliability, reliability ought to be enabled by default. As another example, the default value for a Property regarding the selection of network interfaces ought to permit as many interfaces as possible.

Applications using the Transport Services API are REQUIRED to be robust to the automated selection provided by the Transport Services implementation. This automated selection is constrained by the properties and preferences expressed by the application and requires applications to explicitly set properties that define any necessary constraints on protocol, path, and interface selection.

### 3.2. Allow Access to Specialized Features

There are applications that will need to control fine-grained details of transport protocols to optimize their behavior and ensure compatibility with remote systems. It is therefore RECOMMENDED that the Transport Services API and the Transport Services implementation permit more specialized protocol features to be used.

A specialized feature could be needed by an application only when using a specific protocol, and not when using others. For example, if an application is using TCP, it could require control over the User Timeout Option for TCP; these options would not take effect for other transport protocols. In such cases, the API ought to expose the features in such a way that they take effect when a particular protocol is selected, but do not imply that only that protocol could be used. For example, if the API allows an application to specify a preference to use the User Timeout Option, communication would not fail when a protocol such as QUIC is selected.

Other specialized features, however, can be strictly required by an application and thus constrain the set of protocols that can be used. For example, if an application requires support for automatic handover or failover for a connection, only protocol stacks that provide this feature are eligible to be used, e.g., protocol stacks that include a multipath protocol or a protocol that supports connection migration. A Transport Services API needs to allow applications to define such requirements and constrain the options available to a Transport Services implementation. Since such options are not part of the core/common features, it will generally be simple for an application to modify its set of constraints and change the set of allowable protocol features without changing the core implementation.



### 3.3. Select Equivalent Protocol Stacks

A Transport Services implementation can select Protocol Stacks based on the Selection and Connection Properties communicated by the application, along with any security parameters. If two different Protocol Stacks can be safely swapped, or raced in parallel (see Section 4.2.2), then they are considered to be "equivalent". Equivalent Protocol Stacks are defined as stacks that can provide the same Transport Properties and interface expectations as requested by the application.

The following two examples show non-equivalent Protocol Stacks:

- \* If the application requires preservation of message boundaries, a Protocol Stack that runs UDP as the top-level interface to the application is not equivalent to a Protocol Stack that runs TCP as the top-level interface. A UDP stack would allow an application to read out message boundaries based on datagrams sent from the remote system, whereas TCP does not preserve message boundaries on its own, but needs a framing protocol on top to determine message boundaries.
- \* If the application specifies that it requires reliable transmission of data, then a Protocol Stack using UDP without any reliability layer on top would not be allowed to replace a Protocol Stack using TCP.

The following example shows equivalent Protocol Stacks:

- \* If the application does not require reliable transmission of data, then a Protocol Stack that adds reliability could be regarded as an equivalent Protocol Stack as long as providing this would not conflict with any other application-requested properties.

To ensure that security protocols are not incorrectly swapped, a Transport Services implementation MUST only select Protocol Stacks that meet application requirements ([RFC8922]). A Transport Services implementation SHOULD only race Protocol Stacks where the transport security protocols within the stacks are identical. A Transport Services implementation MUST NOT automatically fall back from secure protocols to insecure protocols, or to weaker versions of secure protocols. A Transport Services implementation MAY allow applications to explicitly specify that fallback to a specific other version of a protocol \, e.g., to allow fallback to TLS 1.2 if TLS 1.3 is not available.

### 3.4. Maintain Interoperability

It is important to note that neither the Transport Services API [I-D.ietf-taps-interface] nor the guidelines for the Transport Service implementation [I-D.ietf-taps-impl] define new protocols or protocol capabilities that affect what is communicated across the network. A Transport Services system **MUST NOT** require that a peer on the other side of a connection uses the same API or implementation. A Transport Services implementation acting as a connection initiator is able to communicate with any existing endpoint that implements the transport protocol(s) and all the required properties selected. Similarly, a Transport Services implementation acting as a listener can receive connections for any protocol that is supported from an existing initiator that implements the protocol, independent of whether the initiator uses the Transport Services architecture or not.

A Transport Services system makes decisions that select protocols and interfaces. In normal use, a given version of a Transport Services system **SHOULD** result in consistent protocol and interface selection decisions for the same network conditions given the same set of Properties. This is intended to provide predictable outcomes to the application using the API.

## 4. Transport Services Architecture and Concepts

This section and the remainder of this document describe the architecture non-normatively. The concepts defined in this document are intended primarily for use in the documents and specifications that describe the Transport Services system. This includes the architecture, the Transport Services API and the associated Transport Services implementation. While the specific terminology can be used in some implementations, it is expected that there will remain a variety of terms used by running code.

The architecture divides the concepts for Transport Services system into two categories:

1. API concepts, which are intended to be exposed to applications; and
2. System-implementation concepts, which are intended to be internally used by a Transport Services implementation.

The following diagram summarizes the top-level concepts in the architecture and how they relate to one another.

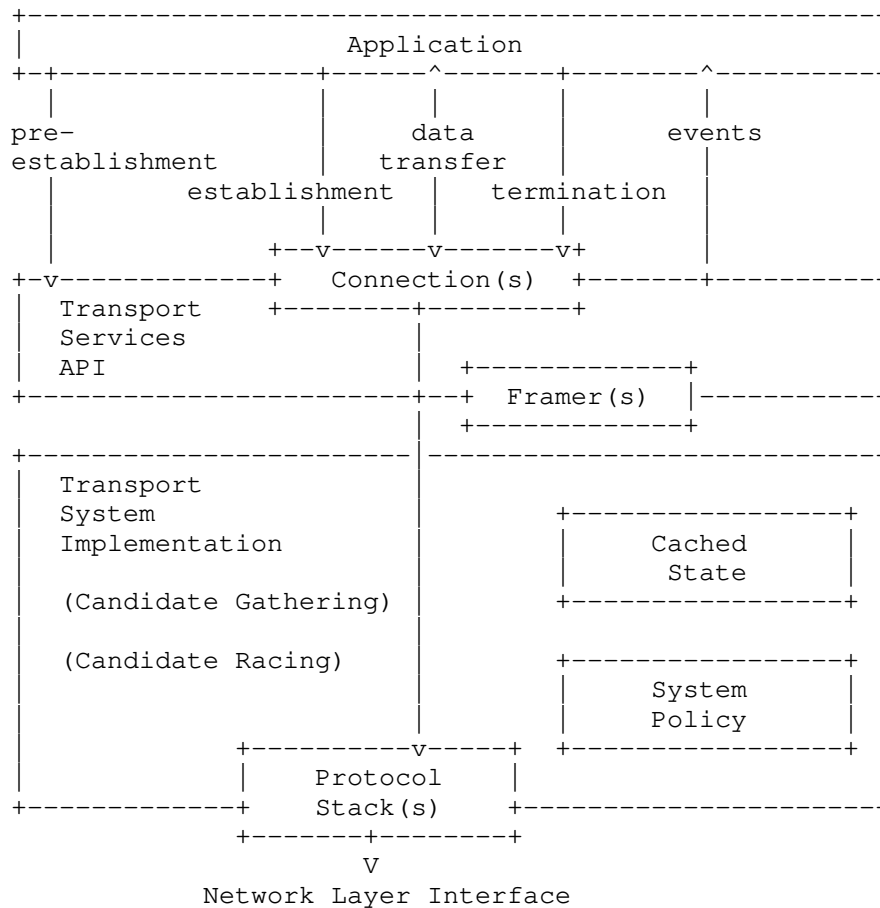


Figure 3: Concepts and Relationships in the Transport Services Architecture

#### 4.1. Transport Services API Concepts

Fundamentally, a Transport Services API needs to provide connection objects (Section 4.1.2) that allow applications to establish communication, and then send and receive data. These could be exposed as handles or referenced objects, depending on the chosen programming language.

Beyond the connection objects, there are several high-level groups of actions that any Transport Services API needs to provide:

- \* Pre-Establishment (Section 4.1.3) encompasses the properties that an application can pass to describe its intent, requirements, prohibitions, and preferences for its networking operations. These properties apply to multiple transport protocols, unless otherwise specified. Properties specified during Pre-Establishment can have a large impact on the rest of the interface: they modify how establishment occurs, they influence the expectations around data transfer, and they determine the set of events that will be supported.
- \* Establishment (Section 4.1.4) focuses on the actions that an application takes on the connection objects to prepare for data transfer.
- \* Data Transfer (Section 4.1.5) consists of how an application represents the data to be sent and received, the functions required to send and receive that data, and how the application is notified of the status of its data transfer.
- \* Event Handling (Section 4.1.6) defines categories of notifications which an application can receive during the lifetime of transport objects. Events also provide opportunities for the application to interact with the underlying transport by querying state or updating maintenance options.
- \* Termination (Section 4.1.7) focuses on the methods by which data transmission is stopped, and state is torn down in the transport.

The diagram below provides a high-level view of the actions and events during the lifetime of a Connection object. Note that some actions are alternatives (e.g., whether to initiate a connection or to listen for incoming connections), while others are optional (e.g., setting Connection and Message Properties in Pre-Establishment) or have been omitted for brevity and simplicity.

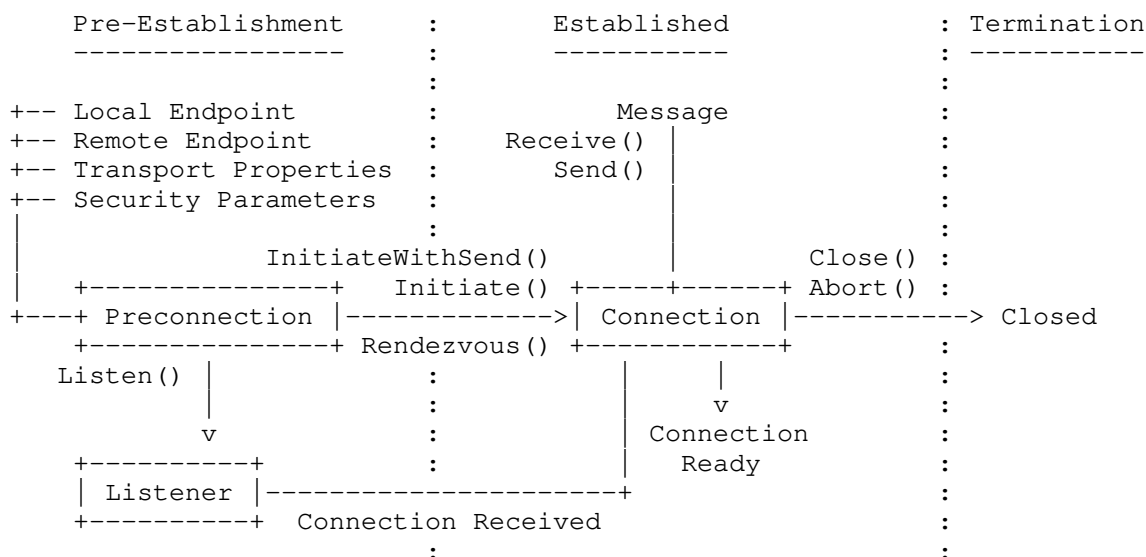


Figure 4: The lifetime of a Connection object

#### 4.1.1. Endpoint Objects

- \* **Endpoint:** An Endpoint represents an identifier for one side of a transport connection. Endpoints can be Local Endpoints or Remote Endpoints, and respectively represent an identity that the application uses for the source or destination of a connection. An Endpoint can be specified at various levels of abstraction. An Endpoint at a higher level of abstraction (such as a hostname) can be resolved to more concrete identities (such as IP addresses). An endpoint may also represent a multicast group, in which case it selects a multicast transport for communication.
- \* **Remote Endpoint:** The Remote Endpoint represents the application's identifier for a peer that can participate in a transport connection; for example, the combination of a DNS name for the peer and a service name/port.
- \* **Local Endpoint:** The Local Endpoint represents the application's identifier for itself that it uses for transport connections; for example, a local IP address and port.

#### 4.1.2. Connections and Related Objects

- \* **Preconnection:** A Preconnection object is a representation of a Connection that has not yet been established. It has state that describes parameters of the Connection: the Local Endpoint from

which that Connection will be established, the Remote Endpoint (Section 4.1.3) to which it will connect, and Transport Properties that influence the paths and protocols a Connection will use. A Preconnection can be either fully specified (representing a single possible Connection), or it can be partially specified (representing a family of possible Connections). The Local Endpoint (Section 4.1.3) is required for a Preconnection used to Listen for incoming Connections, but optional if it is used to Initiate a Connection. The Remote Endpoint is required in a Preconnection that used to Initiate a Connection, but is optional if it is used to Listen for incoming Connections. The Local Endpoint and the Remote Endpoint are both required if a peer-to-peer Rendezvous is to occur based on the Preconnection.

- \* **Transport Properties:** Transport Properties allow the application to express their requirements, prohibitions, and preferences and configure a Transport Services system. There are three kinds of Transport Properties:
  - **Selection Properties (Section 4.1.3):** Selection Properties can only be specified on a Preconnection.
  - **Connection Properties (Section 4.1.3):** Connection Properties can be specified on a Preconnection and changed on the Connection.
  - **Message Properties (Section 4.1.5):** Message Properties can be specified as defaults on a Preconnection or a Connection, and can also be specified during data transfer to affect specific Messages.
- \* **Connection:** A Connection object represents one or more active transport protocol instances that can send and/or receive Messages between Local and Remote Endpoints. It is an abstraction that represents the communication. The Connection object holds state pertaining to the underlying transport protocol instances and any ongoing data transfers. For example, an active Connection can represent a connection-oriented protocol such as TCP, or can represent a fully-specified 5-tuple for a connectionless protocol such as UDP, where the Connection remains an abstraction at the end points. It can also represent a pool of transport protocol instances, e.g., a set of TCP and QUIC connections to equivalent endpoints, or a stream of a multi-streaming transport protocol instance. Connections can be created from a Preconnection or by a Listener.

- \* **Listener:** A Listener object accepts incoming transport protocol connections from Remote Endpoints and generates corresponding Connection objects. It is created from a Preconnection object that specifies the type of incoming Connections it will accept.

#### 4.1.3. Pre-Establishment

- \* **Selection Properties:** The Selection Properties consist of the properties that an application can set to influence the selection of paths between the Local and Remote Endpoints, to influence the selection of transport protocols, or to configure the behavior of generic transport protocol features. These properties can take the form of requirements, prohibitions, or preferences. Examples of properties that influence path selection include the interface type (such as a Wi-Fi connection, or a Cellular LTE connection), requirements around the largest Message that can be sent, or preferences for throughput and latency. Examples of properties that influence protocol selection and configuration of transport protocol features include reliability, multipath support, and fast open support.
- \* **Connection Properties:** The Connection Properties are used to configure protocol-specific options and control per-connection behavior of a Transport Services implementation; for example, a protocol-specific Connection Property can express that if TCP is used, the implementation ought to use the User Timeout Option. Note that the presence of such a property does not require that a specific protocol will be used. In general, these properties do not explicitly determine the selection of paths or protocols, but can be used by an implementation during connection establishment. Connection Properties are specified on a Preconnection prior to Connection establishment, and can be modified on the Connection later. Changes made to Connection Properties after Connection establishment take effect on a best-effort basis.
- \* **Security Parameters:** Security Parameters define an application's requirements for authentication and encryption on a Connection. They are used by Transport Security protocols (such as those described in [RFC8922]) to establish secure Connections. Examples of parameters that can be set include local identities, private keys, supported cryptographic algorithms, and requirements for validating trust of remote identities. Security Parameters are primarily associated with a Preconnection object, but properties related to identities can be associated directly with endpoints.

#### 4.1.4. Establishment Actions

- \* **Initiate:** The primary action that an application can take to create a Connection to a Remote Endpoint, and prepare any required local or remote state to enable the transmission of Messages. For some protocols, this will initiate a client-to-server style handshake; for other protocols, this will just establish local state (e.g., with connectionless protocols such as UDP). The process of identifying options for connecting, such as resolution of the Remote Endpoint, occurs in response to the Initiate call.
- \* **Listen:** Enables a listener to accept incoming Connections. The Listener will then create Connection objects as incoming connections are accepted (Section 4.1.6). Listeners by default register with multiple paths, protocols, and Local Endpoints, unless constrained by Selection Properties and/or the specified Local Endpoint(s). Connections can be accepted on any of the available paths or endpoints.
- \* **Rendezvous:** The action of establishing a peer-to-peer connection with a Remote Endpoint. It simultaneously attempts to initiate a connection to a Remote Endpoint while listening for an incoming connection from that endpoint. The process of identifying options for the connection, such as resolution of the Remote Endpoint, occurs in response to the Rendezvous call. As with Listeners, the set of local paths and endpoints is constrained by Selection Properties. If successful, the Rendezvous call returns a Connection object to represent the established peer-to-peer connection. The processes by which connections are initiated during a Rendezvous action will depend on the set of Local and Remote Endpoints configured on the Preconnection. For example, if the Local and Remote Endpoints are TCP host candidates, then a TCP simultaneous open [RFC0793] will be performed. However, if the set of Local Endpoints includes server reflexive candidates, such as those provided by STUN, a Rendezvous action will race candidates in the style of the ICE algorithm [RFC8445] to perform NAT binding discovery and initiate a peer-to-peer connection.

#### 4.1.5. Data Transfer Objects and Actions

- \* **Message:** A Message object is a unit of data that can be represented as bytes that can be transferred between two endpoints over a transport connection. The bytes within a Message are assumed to be ordered. If an application does not care about the order in which a peer receives two distinct spans of bytes, those spans of bytes are considered independent Messages.
- \* **Message Properties:** Message Properties are used to specify details about Message transmission. They can be specified directly on individual Messages, or can be set on a Preconnection or



Connection as defaults. These properties might only apply to how a Message is sent (such as how the transport will treat prioritization and reliability), but can also include properties that specific protocols encode and communicate to the Remote Endpoint. When receiving Messages, Message Properties can contain information about the received Message, such as metadata generated at the receiver and information signalled by the Remote Endpoint. For example, a Message can be marked with a Message Property indicating that it is the final message on a connection.

- \* **Send:** The action to transmit a Message over a Connection to the Remote Endpoint. The interface to Send can accept Message Properties specific to how the Message content is to be sent. The status of the Send operation is delivered back to the sending application in an Event (Section 4.1.6).
- \* **Receive:** An action that indicates that the application is ready to asynchronously accept a Message over a Connection from a Remote Endpoint, while the Message content itself will be delivered in an Event (Section 4.1.6). The interface to Receive can include Message Properties specific to the Message that is to be delivered to the application.
- \* **Framer:** A Framer is a data translation layer that can be added to a Connection to define how application-layer Messages are transmitted over a transport stack. This is particularly relevant when using a protocol that otherwise presents unstructured streams, such as TCP.

#### 4.1.6. Event Handling

The following categories of events can be delivered to an application:

- \* **Connection Ready:** Signals to an application that a given Connection is ready to send and/or receive Messages. If the Connection relies on handshakes to establish state between peers, then it is assumed that these steps have been taken.
- \* **Connection Closed:** Signals to an application that a given Connection is no longer usable for sending or receiving Messages. The event delivers a reason or error to the application that describes the nature of the termination.
- \* **Connection Received:** Signals to an application that a given Listener has received a Connection.

- \* **Message Received:** Delivers received Message content to the application, based on a Receive action. This can include an error if the Receive action cannot be satisfied due to the Connection being closed.
- \* **Message Sent:** Notifies the application of the status of its Send action. This might indicate a failure if the Message cannot be sent, or an indication that the Message has been processed by the Transport Services system.
- \* **Path Properties Changed:** Notifies the application that a property of the Connection has changed that might influence how and where data is sent and/or received.

#### 4.1.7. Termination Actions

- \* **Close:** The action an application takes on a Connection to indicate that it no longer intends to send data, is no longer willing to receive data, and that the protocol should signal this state to the Remote Endpoint if the transport protocol allows this. (Note that this is distinct from the concept of "half-closing" a bidirectional connection, such as when a FIN is sent in one direction of a TCP connection. The end of a stream can also be indicated using Message Properties when sending.)
- \* **Abort:** The action the application takes on a Connection to indicate a Close and also indicate that a Transport Services system should not attempt to deliver any outstanding data, and immediately drop the connection. This is intended for immediate, usually abnormal, termination of a connection.

#### 4.1.8. Connection Groups

A Connection Group is a set of Connections that shares properties and caches. A Connection Group represents state for managing Connections within a single application, and does not require end-to-end protocol signaling. For multiplexing transport protocols, only Connections within the same Connection Group are allowed to be multiplexed together.

When the API clones an existing Connection, this adds a new Connection to the Connection Group. A change to one of the Connection Properties on any Connection in the Connection Group automatically changes the Connection Property for all others. All Connections in a Connection Group share the same set of Connection Properties except for the Connection Priority. These Connection Properties are said to be entangled.

For multiplexing transport protocols, only Connections within the same Connection Group are allowed to be multiplexed together. Passive Connections can also be added to a Connection Group, e.g., when a Listener receives a new Connection that is just a new stream of an already active multi-streaming protocol instance.

While Connection Groups are managed by the Transport Services system, an application can define Connection Contexts to control caching boundaries, as discussed in Section 4.2.3.

#### 4.2. Transport Services Implementation

This section defines the key concepts of the Transport Services architecture.

- \* Transport Service implementaion: This consists of all objects and protocol instances used internally to a system or library to implement the functionality needed to provide a transport service across a network, as required by the abstract interface.
- \* Transport Service system: This consists of the Transport Service implementaion and the Transport Services API.
- \* Path: Represents an available set of properties that a local endpoint can use to communicate with a Remote Endpoint, such as routes, addresses, and physical and virtual network interfaces.
- \* Protocol Instance: A single instance of one protocol, including any state necessary to establish connectivity or send and receive Messages.
- \* Protocol Stack: A set of Protocol Instances (including relevant application, security, transport, or Internet protocols) that are used together to establish connectivity or send and receive Messages. A single stack can be simple (a single transport protocol instance over IP), or it can be complex (multiple application protocol streams going through a single security and transport protocol, over IP; or, a multi-path transport protocol over multiple transport sub-flows).
- \* Candidate Path: One path that is available to an application and conforms to the Selection Properties and System Policy, of which there can be several. Candidate Paths are identified during the gathering phase (Section 4.2.1) and can be used during the racing phase (Section 4.2.2).

- \* **Candidate Protocol Stack:** One Protocol Stack that can be used by an application for a Connection, which there can be several candidates. Candidate Protocol Stacks are identified during the gathering phase (Section 4.2.1) and are started during the racing phase (Section 4.2.2).
- \* **System Policy:** Represents the input from an operating system or other global preferences that can constrain or influence how an implementation will gather candidate paths and Protocol Stacks (Section 4.2.1) and race the candidates during establishment (Section 4.2.2). Specific aspects of the System Policy either apply to all Connections or only certain ones, depending on the runtime context and properties of the Connection.
- \* **Cached State:** The state and history that the implementation keeps for each set of associated Endpoints that have been used previously. This can include DNS results, TLS session state, previous success and quality of transport protocols over certain paths, as well as other information.

#### 4.2.1. Candidate Gathering

- \* **Candidate Path Selection:** Candidate Path Selection represents the act of choosing one or more paths that are available to use based on the Selection Properties and any available Local and Remote Endpoints provided by the application, as well as the policies and heuristics of a Transport Services implementation.
- \* **Candidate Protocol Selection:** Candidate Protocol Selection represents the act of choosing one or more sets of Protocol Stacks that are available to use based on the Transport Properties provided by the application, and the heuristics or policies within the Transport Services implementation.

#### 4.2.2. Candidate Racing

Connection establishment attempts for a set of candidates may be performed simultaneously, synchronously, serially, or using some combination of all of these. We refer to this process as racing, borrowing terminology from Happy Eyeballs [RFC8305].

- \* **Protocol Option Racing:** Protocol Option Racing is the act of attempting to establish, or scheduling attempts to establish, multiple Protocol Stacks that differ based on the composition of protocols or the options used for protocols.

- \* **Path Racing:** Path Racing is the act of attempting to establish, or scheduling attempts to establish, multiple Protocol Stacks that differ based on a selection from the available Paths. Since different Paths will have distinct configurations for local addresses and DNS servers, attempts across different Paths will perform separate DNS resolution steps, which can lead to further racing of the resolved Remote Endpoints.
- \* **Remote Endpoint Racing:** Remote Endpoint Racing is the act of attempting to establish, or scheduling attempts to establish, multiple Protocol Stacks that differ based on the specific representation of the Remote Endpoint, such as a particular IP address that was resolved from a DNS hostname.

#### 4.2.3. Separating Connection Contexts

By default, stored properties of the implementation, such as cached protocol state, cached path state, and heuristics, may be shared (e.g. across multiple connections in an application). This provides efficiency and convenience for the application, since the Transport Services system can automatically optimize behavior.

The Transport Services API can allow applications to explicitly define Connection Contexts that force separation of Cached State and Protocol Stacks. For example, a web browser application could use Connection Contexts with separate caches when implementing different tabs. Possible reasons to isolate Connections using separate Connection Contexts include:

- \* Privacy concerns about re-using cached protocol state that can lead to linkability. Sensitive state could include TLS session state [RFC8446] and HTTP cookies [RFC6265]. These concerns could be addressed using Connection Contexts with separate caches, such as for different browser tabs.
- \* Privacy concerns about allowing Connections to multiplex together, which can tell a Remote Endpoint that all of the Connections are coming from the same application. Using Connection Contexts avoids the Connections being multiplexed in a HTTP/2 or QUIC stream.

#### 5. IANA Considerations

RFC-EDITOR: Please remove this section before publication.

This document has no actions for IANA.

## 6. Security and Privacy Considerations

The Transport Services architecture does not recommend use of specific security protocols or algorithms. Its goal is to offer ease of use for existing protocols by providing a generic security-related interface. Each provided interface translates to an existing protocol-specific interface provided by supported security protocols. For example, trust verification callbacks are common parts of TLS APIs; a Transport Services API exposes similar functionality [RFC8922].

As described above in Section 3.3, if a Transport Services implementation races between two different Protocol Stacks, both need to use the same security protocols and options. However, a Transport Services implementation can race different security protocols, e.g., if the application explicitly specifies that it considers them equivalent.

The application controls whether information from previous racing attempts, or other information about past communications that was cached by the Transport Services system is used during establishment. This allows applications to make tradeoffs between efficiency (through racing) and privacy (via information that might leak from the cache toward an on-path observer). Some applications have native concepts (e.g. "incognito mode") that align with this functionality.

Applications need to ensure that they use security APIs appropriately. In cases where applications use an interface to provide sensitive keying material, e.g., access to private keys or copies of pre-shared keys (PSKs), key use needs to be validated. For example, applications ought not to use PSK material created for the Encapsulating Security Protocol (ESP, part of IPsec) [RFC4303] with QUIC, and applications ought not to use private keys intended for server authentication as keys for client authentication.

A Transport Services system must not automatically fall back from secure protocols to insecure protocols, or to weaker versions of secure protocols (see Section 3.3). For example, if an application requests a specific version of TLS, but the desired version of TLS is not available, its connection will fail. Applications are thus responsible for implementing security protocol fallback or version fallback by creating multiple Connections, if so desired. Alternatively, the Transport Services API MAY allow applications to specify that fallback to a specific other version of a protocol is allowed by the Transport Services system.

## 7. Acknowledgements

This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreements No. 644334 (NEAT), No. 688421 (MAMI) and No 815178 (5GENESIS).

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 Theresa Enhardt, Max Franke, Mirja Kuehlewind, Jonathan Lennox, and Michael Welzl for the discussions and feedback that helped shape the architecture described here. Particular thanks is also due to Philipp S. Tiesel and Christopher A. Wood, who were both co-authors of this architecture specification as it progressed through the TAPS working group. Thanks as well to Stuart Cheshire, Josh Graessley, David Schinazi, and Eric Kinnear for their implementation and design efforts, including Happy Eyeballs, that heavily influenced this work.

## 8. References

### 8.1. Normative References

- [I-D.ietf-taps-interface]  
Trammell, B., Welzl, M., Enhardt, T., Fairhurst, G., Kuehlewind, M., Perkins, C., Tiesel, P. S., Wood, C. A., Pauly, T., and K. Rose, "An Abstract Application Layer Interface to Transport Services", Work in Progress, Internet-Draft, draft-ietf-taps-interface-14, 3 January 2022, <<https://datatracker.ietf.org/doc/html/draft-ietf-taps-interface-14>>.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/rfc/rfc2119>>.
- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/rfc/rfc8174>>.

### 8.2. Informative References

- [I-D.ietf-taps-impl]  
Brunstrom, A., Pauly, T., Enghardt, T., Grinnemo, K., Jones, T., Tiesel, P. S., Perkins, C., and M. Welzl, "Implementing Interfaces to Transport Services", Work in Progress, Internet-Draft, draft-ietf-taps-impl-10, 12 July 2021, <<https://datatracker.ietf.org/doc/html/draft-ietf-taps-impl-10>>.
- [POSIX] "IEEE Std. 1003.1-2008 Standard for Information Technology -- Portable Operating System Interface (POSIX). Open group Technical Standard: Base Specifications, Issue 7", 2008.
- [RFC0793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, DOI 10.17487/RFC0793, September 1981, <<https://www.rfc-editor.org/rfc/rfc793>>.
- [RFC4303] Kent, S., "IP Encapsulating Security Payload (ESP)", RFC 4303, DOI 10.17487/RFC4303, December 2005, <<https://www.rfc-editor.org/rfc/rfc4303>>.
- [RFC6265] Barth, A., "HTTP State Management Mechanism", RFC 6265, DOI 10.17487/RFC6265, April 2011, <<https://www.rfc-editor.org/rfc/rfc6265>>.
- [RFC7230] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, DOI 10.17487/RFC7230, June 2014, <<https://www.rfc-editor.org/rfc/rfc7230>>.
- [RFC7540] Belshe, M., Peon, R., and M. Thomson, Ed., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, DOI 10.17487/RFC7540, May 2015, <<https://www.rfc-editor.org/rfc/rfc7540>>.
- [RFC8095] Fairhurst, G., Ed., Trammell, B., Ed., and M. Kuehlewind, Ed., "Services Provided by IETF Transport Protocols and Congestion Control Mechanisms", RFC 8095, DOI 10.17487/RFC8095, March 2017, <<https://www.rfc-editor.org/rfc/rfc8095>>.
- [RFC8305] Schinazi, D. and T. Pauly, "Happy Eyeballs Version 2: Better Connectivity Using Concurrency", RFC 8305, DOI 10.17487/RFC8305, December 2017, <<https://www.rfc-editor.org/rfc/rfc8305>>.



- [RFC8445] Keranen, A., Holmberg, C., and J. Rosenberg, "Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal", RFC 8445, DOI 10.17487/RFC8445, July 2018, <<https://www.rfc-editor.org/rfc/rfc8445>>.
- [RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <<https://www.rfc-editor.org/rfc/rfc8446>>.
- [RFC8922] Enghardt, T., Pauly, T., Perkins, C., Rose, K., and C. Wood, "A Survey of the Interaction between Security Protocols and Transport Services", RFC 8922, DOI 10.17487/RFC8922, October 2020, <<https://www.rfc-editor.org/rfc/rfc8922>>.
- [RFC8923] Welzl, M. and S. Gjessing, "A Minimal Set of Transport Services for End Systems", RFC 8923, DOI 10.17487/RFC8923, October 2020, <<https://www.rfc-editor.org/rfc/rfc8923>>.

#### Authors' Addresses

Tommy Pauly (editor)  
Apple Inc.  
One Apple Park Way  
Cupertino, California 95014,  
United States of America

Email: [tpauly@apple.com](mailto:tpauly@apple.com)

Brian Trammell (editor)  
Google Switzerland GmbH  
Gustav-Gull-Platz 1  
CH- 8004 Zurich  
Switzerland

Email: [ietf@trammell.ch](mailto:ietf@trammell.ch)

Anna Brunstrom  
Karlstad University  
Universitetsgatan 2  
651 88 Karlstad  
Sweden

Email: [anna.brunstrom@kau.se](mailto:anna.brunstrom@kau.se)

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

Email: [gorry@erg.abdn.ac.uk](mailto:gorry@erg.abdn.ac.uk)  
URI: <http://www.erg.abdn.ac.uk/>

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

Email: [csp@csp Perkins.org](mailto:csp@csp Perkins.org)

TAPS Working Group  
Internet-Draft  
Intended status: Informational  
Expires: January 2, 2019

A. Brunstrom, Ed.  
Karlstad University  
T. Pauly, Ed.  
Apple Inc.  
T. Enghardt  
TU Berlin  
K-J. Grinnemo  
Karlstad University  
T. Jones  
University of Aberdeen  
P. Tiesel  
TU Berlin  
C. Perkins  
University of Glasgow  
M. Welzl  
University of Oslo  
July 01, 2018

Implementing Interfaces to Transport Services  
draft-ietf-taps-impl-01

Abstract

The Transport Services architecture [I-D.ietf-taps-arch] defines a system that allows applications to use transport networking protocols flexibly. This document serves as a guide to implementation on how to build such a system.

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 January 2, 2019.

## Copyright Notice

Copyright (c) 2018 IETF Trust and the persons identified as the document authors. All rights reserved.

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

## Table of Contents

1. Introduction . . . . .	3
2. Implementing Basic Objects . . . . .	3
3. Implementing Pre-Establishment . . . . .	4
3.1. Configuration-time errors . . . . .	4
3.2. Role of system policy . . . . .	5
4. Implementing Connection Establishment . . . . .	6
4.1. Candidate Gathering . . . . .	7
4.1.1. Structuring Options as a Tree . . . . .	7
4.1.2. Branch Types . . . . .	9
4.2. Branching Order-of-Operations . . . . .	11
4.3. Sorting Branches . . . . .	12
4.4. Candidate Racing . . . . .	13
4.4.1. Delayed Racing . . . . .	14
4.4.2. Failover . . . . .	15
4.5. Completing Establishment . . . . .	15
4.5.1. Determining Successful Establishment . . . . .	16
4.6. Establishing multiplexed connections . . . . .	17
4.7. Handling racing with "unconnected" protocols . . . . .	17
4.8. Implementing listeners . . . . .	18
4.8.1. Implementing listeners for Connected Protocols . . . . .	18
4.8.2. Implementing listeners for Unconnected Protocols . . . . .	18
4.8.3. Implementing listeners for Multiplexed Protocols . . . . .	19
5. Implementing Data Transfer . . . . .	19
5.1. Data transfer for streams, datagrams, and frames . . . . .	19
5.1.1. Sending Messages . . . . .	19
5.1.2. Receiving Messages . . . . .	21
5.2. Handling of data for fast-open protocols . . . . .	22
6. Implementing Maintenance . . . . .	23
6.1. Changing Protocol Properties . . . . .	23
6.2. Handling Path Changes . . . . .	24
7. Implementing Termination . . . . .	24

8.	Cached State . . . . .	25
8.1.	Protocol state caches . . . . .	25
8.2.	Performance caches . . . . .	26
9.	Specific Transport Protocol Considerations . . . . .	27
9.1.	TCP . . . . .	27
9.2.	UDP . . . . .	27
9.3.	SCTP . . . . .	28
9.4.	TLS . . . . .	28
9.5.	HTTP . . . . .	29
9.6.	QUIC . . . . .	29
9.7.	HTTP/2 transport . . . . .	29
10.	Rendezvous and Environment Discovery . . . . .	30
11.	IANA Considerations . . . . .	32
12.	Security Considerations . . . . .	32
12.1.	Considerations for Candidate Gathering . . . . .	32
12.2.	Considerations for Candidate Racing . . . . .	32
13.	Acknowledgements . . . . .	32
14.	References . . . . .	33
14.1.	Normative References . . . . .	33
14.2.	Informative References . . . . .	34
Appendix A.	Additional Properties . . . . .	34
A.1.	Properties Affecting Sorting of Branches . . . . .	35
A.2.	Send Parameters . . . . .	35
Authors' Addresses	. . . . .	36

## 1. Introduction

The Transport Services architecture [I-D.ietf-taps-arch] defines a system that allows applications to use transport networking protocols flexibly. The interface such a system exposes to applications is defined as the Transport Services API [I-D.ietf-taps-interface]. This API is designed to be generic across multiple transport protocols and sets of protocols features.

This document serves as a guide to implementation on how to build a system that provides a Transport Services API. It is the job of an implementation of a Transport Services system to turn the requests of an application into decisions on how to establish connections, and how to transfer data over those connections once established. The terminology used in this document is based on the Architecture [I-D.ietf-taps-arch].

## 2. Implementing Basic Objects

The basic objects that are exposed to applications for Transport Services are the Preconnection, the bundle of properties that describes the application constraints on the transport; the Connection, the basic object that represents a flow of data in either

direction between the Local and Remote Endpoints; and the Listener, a passive waiting object that delivers new Connections.

Preconnection objects should be implemented as bundles of properties that an application can both read and write. Once a Preconnection has been used to create an outbound Connection or a Listener, the implementation should ensure that the copy of the properties held by the Connection or Listener is immutable. This may involve performing a deep-copy if the application is still able to modify properties on the original Preconnection object.

Connection objects represent the interface between the application and the implementation to manage transport state, and conduct data transfer. During the process of establishment (Section 4), the Connection will be unbound to a specific transport flow, since there may be multiple candidate Protocol Stacks being raced. Once the Connection is established, the object should be considered mapped to a specific Protocol Stack. The notion of a Connection maps to many different protocols, depending on the Protocol Stack. For example, the Connection may ultimately represent the interface into a TCP connection, a TLS session over TCP, a UDP flow with fully-specified local and remote endpoints, a DTLS session, a SCTP stream, a QUIC stream, or an HTTP/2 stream.

Listener objects are created with a Preconnection, at which point their configuration should be considered immutable by the implementation. The process of listening is described in Section 4.8.

### 3. Implementing Pre-Establishment

During pre-establishment the application specifies the Endpoints to be used for communication as well as its preferences regarding Protocol and Path Selection. The implementation stores these objects and properties as part of the Preconnection object for use during connection establishment. For Protocol and Path Selection Properties that are not provided by the application, the implementation must use the default values specified in the Transport Services API ([I-D.ietf-taps-interface]).

#### 3.1. Configuration-time errors

The transport system should have a list of supported protocols available, which each have transport features reflecting the capabilities of the protocol. Once an application specifies its Transport Parameters, the transport system should match the required and prohibited properties against the transport features of the available protocols.

In the following cases, failure should be detected during pre-establishment:

- o The application requested Protocol Properties that include requirements or prohibitions that cannot be satisfied by any of the available protocols. For example, if an application requires "Configure Reliability per Message", but no such protocol is available on the host running the transport system, e.g., because SCTP is not supported by the operating system, this should result in an error.
- o The application requested Protocol Properties that are in conflict with each other, i.e., the required and prohibited properties cannot be satisfied by the same protocol. For example, if an application prohibits "Reliable Data Transfer" but then requires "Configure Reliability per Message", this mismatch should result in an error.

It is important to fail as early as possible in such cases in order to avoid allocating resources, e.g., to endpoint resolution, only to find out later that there is no protocol that satisfies the requirements.

### 3.2. Role of system policy

The properties specified during pre-establishment has a close connection to system policy. The implementation is responsible for combining and reconciling several different sources of preferences when establishing Connections. These include, but are not limited to:

1. Application preferences, i.e., preferences specified during the pre-establishment such as Local Endpoint, Remote Endpoint, Path Selection Properties, and Protocol Selection Properties.
2. Dynamic system policy, i.e., policy compiled from internally and externally acquired information about available network interfaces, supported transport protocols, and current/previous Connections. Examples of ways to externally retrieve policy-support information are through OS-specific statistics/measurement tools and tools that reside on middleboxes and routers.
3. Default implementation policy, i.e., predefined policy by OS or application.

In general, any protocol or path used for a connection must conform to all three sources of constraints. Any violation of any of the

layers should cause a protocol or path to be considered ineligible for use. For an example of application preferences leading to constraints, an application may prohibit the use of metered network interfaces for a given Connection to avoid user cost. Similarly, the system policy at a given time may prohibit the use of such a metered network interface from the application's process. Lastly, the implementation itself may default to disallowing certain network interfaces unless explicitly requested by the application and allowed by the system.

It is expected that the database of system policies and the method of looking up these policies will vary across various platforms. An implementation should attempt to look up the relevant policies for the system in a dynamic way to make sure it is reflecting an accurate version of the system policy, since the system's policy regarding the application's traffic may change over time due to user or administrative changes.

#### 4. Implementing Connection Establishment

The process of establishing a network connection begins when an application expresses intent to communicate with a remote endpoint by calling `Initiate`. (At this point, any constraints or requirements the application may have on the connection are available from pre-establishment.) The process can be considered complete once there is at least one Protocol Stack that has completed any required setup to the point that it can transmit and receive the application's data.

Connection establishment is divided into two top-level steps: `Candidate Gathering`, to identify the paths, protocols, and endpoints to use, and `Candidate Racing`, in which the necessary protocol handshakes are conducted in order to select which set to use.

The most simple example of this process might involve identifying the single IP address to which the implementation wishes to connect, using the system's current default interface or path, and starting a TCP handshake to establish a stream to the specified IP address. However, each step may also vary depending on the requirements of the connection: if the endpoint is defined as a hostname and port, then there may be multiple resolved addresses that are available; there may also be multiple interfaces or paths available, other than the default system interface; and some protocols may not need any transport handshake to be considered "established" (such as UDP), while other connections may utilize layered protocol handshakes, such as TLS over TCP.

Whenever an implementation has multiple options for connection establishment, it can view the set of all individual connection



establishment options as a single, aggregate connection establishment. The aggregate set conceptually includes every valid combination of endpoints, paths, and protocols. As an example, consider an implementation that initiates a TCP connection to a hostname + port endpoint, and has two valid interfaces available (Wi-Fi and LTE). The hostname resolves to a single IPv4 address on the Wi-Fi network, and resolves to the same IPv4 address on the LTE network, as well as a single IPv6 address. The aggregate set of connection establishment options can be viewed as follows:

```
Aggregate [Endpoint: www.example.com:80] [Interface: Any] [Protocol: TCP]
|-> [Endpoint: 192.0.2.1:80] [Interface: Wi-Fi] [Protocol: TCP]
|-> [Endpoint: 192.0.2.1:80] [Interface: LTE] [Protocol: TCP]
|-> [Endpoint: 2001:DB8::1.80] [Interface: LTE] [Protocol: TCP]
```

Any one of these sub-entries on the aggregate connection attempt would satisfy the original application intent. The concern of this section is the algorithm defining which of these options to try, when, and in what order.

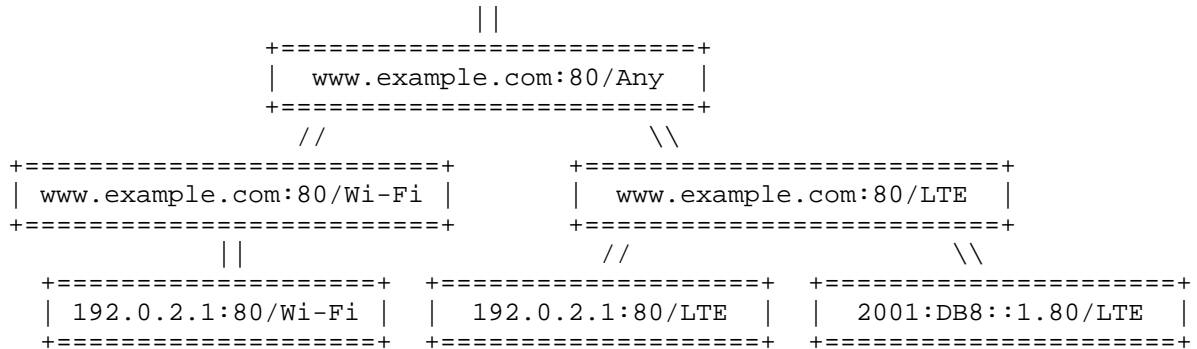
#### 4.1. Candidate Gathering

The step of gathering candidates involves identifying which paths, protocols, and endpoints may be used for a given Connection. This list is determined by the requirements, prohibitions, and preferences of the application as specified in the Path Selection Properties and Protocol Selection Properties.

##### 4.1.1. Structuring Options as a Tree

When an implementation responsible for connection establishment needs to consider multiple options, it should logically structure these options as a hierarchical tree. Each leaf node of the tree represents a single, coherent connection attempt, with an Endpoint, a Path, and a set of protocols that can directly negotiate and send data on the network. Each node in the tree that is not a leaf represents a connection attempt that is either underspecified, or else includes multiple distinct options. For example, when connecting on an IP network, a connection attempt to a hostname and port is underspecified, because the connection attempt requires a resolved IP address as its remote endpoint. In this case, the node represented by the connection attempt to the hostname is a parent node, with child nodes for each IP address. Similarly, an implementation that is allowed to connect using multiple interfaces will have a parent node of the tree for the decision between the paths, with a branch for each interface.

The example aggregate connection attempt above can be drawn as a tree by grouping the addresses resolved on the same interface into branches:



The rest of this section will use a notation scheme to represent this tree. The parent (or trunk) node of the tree will be represented by a single integer, such as "1". Each child of that node will have an integer that identifies it, from 1 to the number of children. That child node will be uniquely identified by concatenating its integer to it's parents identifier with a dot in between, such as "1.1" and "1.2". Each node will be summarized by a tuple of three elements: Endpoint, Path, and Protocol. The above example can now be written more succinctly as:

```

1 [www.example.com:80, Any, TCP]
  1.1 [www.example.com:80, Wi-Fi, TCP]
    1.1.1 [192.0.2.1:80, Wi-Fi, TCP]
  1.2 [www.example.com:80, LTE, TCP]
    1.2.1 [192.0.2.1:80, LTE, TCP]
    1.2.2 [2001:DB8::1.80, LTE, TCP]

```

When an implementation views this aggregate set of connection attempts as a single connection establishment, it only will use one of the leaf nodes to transfer data. Thus, when a single leaf node becomes ready to use, then the entire connection attempt is ready to use by the application. Another way to represent this is that every leaf node updates the state of its parent node when it becomes ready, until the trunk node of the tree is ready, which then notifies the application that the connection as a whole is ready to use.

A connection establishment tree may be degenerate, and only have a single leaf node, such as a connection attempt to an IP address over a single interface with a single protocol.

```

1 [192.0.2.1:80, Wi-Fi, TCP]

```

A parent node may also only have one child (or leaf) node, such as a when a hostname resolves to only a single IP address.

```
1 [www.example.com:80, Wi-Fi, TCP]
  1.1 [192.0.2.1:80, Wi-Fi, TCP]
```

#### 4.1.2. Branch Types

There are three types of branching from a parent node into one or more child nodes. Any parent node of the tree must only use one type of branching.

##### 4.1.2.1. Derived Endpoints

If a connection originally targets a single endpoint, there may be multiple endpoints of different types that can be derived from the original. The connection library should order the derived endpoints according to application preference, system policy and expected performance.

DNS hostname-to-address resolution is the most common method of endpoint derivation. When trying to connect to a hostname endpoint on a traditional IP network, the implementation should send DNS queries for both A (IPv4) and AAAA (IPv6) records if both are supported on the local link. The algorithm for ordering and racing these addresses should follow the recommendations in Happy Eyeballs [RFC8305].

```
1 [www.example.com:80, Wi-Fi, TCP]
  1.1 [2001:DB8::1.80, Wi-Fi, TCP]
  1.2 [192.0.2.1:80, Wi-Fi, TCP]
  1.3 [2001:DB8::2.80, Wi-Fi, TCP]
  1.4 [2001:DB8::3.80, Wi-Fi, TCP]
```

DNS-Based Service Discovery can also provide an endpoint derivation step. When trying to connect to a named service, the client may discover one or more hostname and port pairs on the local network using multicast DNS. These hostnames should each be treated as a branch which can be attempted independently from other hostnames. Each of these hostnames may also resolve to one or more addresses, thus creating multiple layers of branching.

```
1 [term-printer._ipp._tcp.meeting.ietf.org, Wi-Fi, TCP]
  1.1 [term-printer.meeting.ietf.org:631, Wi-Fi, TCP]
    1.1.1 [31.133.160.18.631, Wi-Fi, TCP]
```

#### 4.1.2.2. Alternate Paths

If a client has multiple network interfaces available to it, such as mobile client with both Wi-Fi and Cellular connectivity, it can attempt a connection over either interface. This represents a branch point in the connection establishment. Like with derived endpoints, the interfaces should be ranked based on preference, system policy, and performance. Attempts should be started on one interface, and then on other interfaces successively after delays based on expected round-trip-time or other available metrics.

```
1 [192.0.2.1:80, Any, TCP]
  1.1 [192.0.2.1:80, Wi-Fi, TCP]
  1.2 [192.0.2.1:80, LTE, TCP]
```

This same approach applies to any situation in which the client is aware of multiple links or views of the network. Multiple Paths, each with a coherent set of addresses, routes, DNS server, and more, may share a single interface. A path may also represent a virtual interface service such as a Virtual Private Network (VPN).

The list of available paths should be constrained by any requirements or prohibitions the application sets, as well as system policy.

#### 4.1.2.3. Protocol Options

Differences in possible protocol compositions and options can also provide a branching point in connection establishment. This allows clients to be resilient to situations in which a certain protocol is not functioning on a server or network.

This approach is commonly used for connections with optional proxy server configurations. A single connection may be allowed to use an HTTP-based proxy, a SOCKS-based proxy, or connect directly. These options should be ranked and attempted in succession.

```
1 [www.example.com:80, Any, HTTP/TCP]
  1.1 [192.0.2.8:80, Any, HTTP/HTTP Proxy/TCP]
  1.2 [192.0.2.7:10234, Any, HTTP/SOCKS/TCP]
  1.3 [www.example.com:80, Any, HTTP/TCP]
    1.3.1 [192.0.2.1:80, Any, HTTP/TCP]
```

This approach also allows a client to attempt different sets of application and transport protocols that may provide preferable characteristics when available. For example, the protocol options could involve QUIC [I-D.ietf-quic-transport] over UDP on one branch, and HTTP/2 [RFC7540] over TLS over TCP on the other:

- 1 [www.example.com:443, Any, Any HTTP]
  - 1.1 [www.example.com:443, Any, QUIC/UDP]
    - 1.1.1 [192.0.2.1:443, Any, QUIC/UDP]
  - 1.2 [www.example.com:443, Any, HTTP2/TLS/TCP]
    - 1.2.1 [192.0.2.1:443, Any, HTTP2/TLS/TCP]

Another example is racing SCTP with TCP:

- 1 [www.example.com:80, Any, Any Stream]
  - 1.1 [www.example.com:80, Any, SCTP]
    - 1.1.1 [192.0.2.1:80, Any, SCTP]
  - 1.2 [www.example.com:80, Any, TCP]
    - 1.2.1 [192.0.2.1:80, Any, TCP]

Implementations that support racing protocols and protocol options should maintain a history of which protocols and protocol options successfully established, on a per-network basis (see Section 8.2). This information can influence future racing decisions to prioritize or prune branches.

#### 4.2. Branching Order-of-Operations

Branch types must occur in a specific order relative to one another to avoid creating leaf nodes with invalid or incompatible settings. In the example above, it would be invalid to branch for derived endpoints (the DNS results for www.example.com) before branching between interface paths, since usable DNS results on one network may not necessarily be the same as DNS results on another network due to local network entities, supported address families, or enterprise network configurations. Implementations must be careful to branch in an order that results in usable leaf nodes whenever there are multiple branch types that could be used from a single node.

The order of operations for branching, where lower numbers are acted upon first, should be:

1. Alternate Paths
2. Protocol Options
3. Derived Endpoints

Branching between paths is the first in the list because results across multiple interfaces are likely not related to one another: endpoint resolution may return different results, especially when using locally resolved host and service names, and which protocols are supported and preferred may differ across interfaces. Thus, if

multiple paths are attempted, the overall connection can be seen as a race between the available paths or interfaces.

Protocol options are checked next in order. Whether or not a set of protocol, or protocol-specific options, can successfully connect is generally not dependent on which specific IP address is used. Furthermore, the protocol stacks being attempted may influence or altogether change the endpoints being used. Adding a proxy to a connection's branch will change the endpoint to the proxy's IP address or hostname. Choosing an alternate protocol may also modify the ports that should be selected.

Branching for derived endpoints is the final step, and may have multiple layers of derivation or resolution, such as DNS service resolution and DNS hostname resolution.

#### 4.3. Sorting Branches

Implementations should sort the branches of the tree of connection options in order of their preference rank. Leaf nodes on branches with higher rankings represent connection attempts that will be raced first. Implementations should order the branches to reflect the preferences expressed by the application for its new connection, including Protocol and Path Selection Properties, which are specified in [I-D.ietf-taps-interface].

In addition to the properties provided by the application, an implementation may include additional criteria such as cached performance estimates, see Section 8.2, or system policy, see Section 3.2, in the ranking. Two examples of how the Protocol and Path Selection Properties may be used to sort branches are provided below:

- o Interface Type: If the application specifies an interface type to be preferred or avoided, implementations should rank paths accordingly. If the application specifies an interface type to be required or prohibited, we expect an implementation to not include the non-conforming paths into the three.
- o Capacity Profile: An implementation may use the Capacity Profile to prefer paths optimized for the application's expected traffic pattern according to cached performance estimates, see Section 8.2:
  - \* Interactive/Low Latency: Prefer paths with the lowest expected Round Trip Time

- \* Constant Rate: Prefer paths that can satisfy the requested Stream Send or Stream Receive Bitrate, based on observed maximum throughput
- \* Scavenger/Bulk: Prefer paths with the highest expected available bandwidth, based on observed maximum throughput

[Note: See Appendix A.1 for additional examples related to Properties under discussion.]

Implementations should process properties in the following order: Prohibit, Require, Prefer, Avoid. If Protocol or Path Selection Properties contain any prohibited properties, the implementation should first purge branches containing nodes with these properties. For required properties, it should only keep branches that satisfy these requirements. Finally, it should order branches according to preferred properties, and finally use avoided properties as a tiebreaker.

For Require and Avoid, Path Selection Properties take precedence over Protocol Selection Properties. For example, if the application has indicated both a preference for WiFi over LTE and for a feature only available in SCTP, branches will be first sorted accord to the Path Selection Property, with WiFi at the top. Then, branches with SCTP will be sorted to the top within their subtree according to the Protocol Selection Property. However, if the implementation has cached the information that SCTP is not available on the path over WiFi, there is no SCTP node in the WiFi subtree. Here, the path over WiFi will be tried first, and, if connection establishment succeeds, TCP will be used. So the Path Selection Property of preferring WiFi takes precedence over the Protocol Selection Property of preferring SCTP.

1. [www.example.com:80, Any, Any Stream]
  - 1.1 [192.0.2.1:80, Wi-Fi, Any Stream]
    - 1.1.1 [192.0.2.1:80, Wi-Fi, TCP]
  - 1.2 [192.0.3.1:80, LTE, Any Stream]
    - 1.2.1 [192.0.3.1:80, LTE, SCTP]
    - 1.2.2 [192.0.3.1:80, LTE, TCP]

#### 4.4. Candidate Racing

The primary goal of the Candidate Racing process is to successfully negotiate a protocol stack to an endpoint over an interface--to connect a single leaf node of the tree--with as little delay and as few unnecessary connections attempts as possible. Optimizing these two factors improves the user experience, while minimizing network load.

This section covers the dynamic aspect of connection establishment. While the tree described above is a useful conceptual and architectural model, an implementation does not know what the full tree may become up front, nor will many of the possible branches be used in the common case.

There are three different approaches to racing the attempts for different nodes of the connection establishment tree:

1. Immediate
2. Delayed
3. Failover

Each approach is appropriate in different use-cases and branch types. However, to avoid consuming unnecessary network resources, implementations should not use immediate racing as a default approach.

The timing algorithms for racing should remain independent across branches of the tree. Any timers or racing logic is isolated to a given parent node, and is not ordered precisely with regards to other children of other nodes.

#### 4.4.1. Delayed Racing

Delayed racing can be used whenever a single node of the tree has multiple child nodes. Based on the order determined when building the tree, the first child node will be initiated immediately, followed by the next child node after some delay. Once that second child node is initiated, the third child node (if present) will begin after another delay, and so on until all child nodes have been initiated, or one of the child nodes successfully completes its negotiation.

Delayed racing attempts occur in parallel. Implementations should not terminate an earlier child connection attempt upon starting a secondary child.

The delay between starting child nodes should be based on the properties of the previously started child node. For example, if the first child represents an IP address with a known route, and the second child represents another IP address, the delay between starting the first and second IP addresses can be based on the expected retransmission cadence for the first child's connection (derived from historical round-trip-time). Alternatively, if the first child represents a branch on a Wi-Fi interface, and the second



child represents a branch on an LTE interface, the delay should be based on the expected time in which the branch for the first interface would be able to establish a connection, based on link quality and historical round-trip-time.

Any delay should have a defined minimum and maximum value based on the branch type. Generally, branches between paths and protocols should have longer delays than branches between derived endpoints. The maximum delay should be considered with regards to how long a user is expected to wait for the connection to complete.

If a child node fails to connect before the delay timer has fired for the next child, the next child should be started immediately.

#### 4.4.2. Failover

If an implementation or application has a strong preference for one branch over another, the branching node may choose to wait until one child has failed before starting the next. Failure of a leaf node is determined by its protocol negotiation failing or timing out; failure of a parent branching node is determined by all of its children failing.

An example in which failover is recommended is a race between a protocol stack that uses a proxy and a protocol stack that bypasses the proxy. Failover is useful in case the proxy is down or misconfigured, but any more aggressive type of racing may end up unnecessarily avoiding a proxy that was preferred by policy.

#### 4.5. Completing Establishment

The process of connection establishment completes when one leaf node of the tree has completed negotiation with the remote endpoint successfully, or else all nodes of the tree have failed to connect. The first leaf node to complete its connection is then used by the application to send and receive data.

It is useful to process success and failure throughout the tree by child nodes reporting to their parent nodes (towards the trunk of the tree). For example, in the following case, if 1.1.1 fails to connect, it reports the failure to 1.1. Since 1.1 has no other child nodes, it also has failed and reports that failure to 1. Because 1.2 has not yet failed, 1 is not considered to have failed. Since 1.2 has not yet started, it is started and the process continues. Similarly, if 1.1.1 successfully connects, then it marks 1.1 as connected, which propagates to the trunk node 1. At this point, the connection as a whole is considered to be successfully connected and ready to process application data

```
1 [www.example.com:80, Any, TCP]
  1.1 [www.example.com:80, Wi-Fi, TCP]
    1.1.1 [192.0.2.1:80, Wi-Fi, TCP]
  1.2 [www.example.com:80, LTE, TCP]
...
```

If a leaf node has successfully completed its connection, all other attempts should be made ineligible for use by the application for the original request. New connection attempts that involve transmitting data on the network should not be started after another leaf node has completed successfully, as the connection as a whole has been established. An implementation may choose to let certain handshakes and negotiations complete in order to gather metrics to influence future connections. Similarly, an implementation may choose to hold onto fully established leaf nodes that were not the first to establish for use in future connections, but this approach is not recommended since those attempts were slower to connect and may exhibit less desirable properties.

#### 4.5.1. Determining Successful Establishment

Implementations may select the criteria by which a leaf node is considered to be successfully connected differently on a per-protocol basis. If the only protocol being used is a transport protocol with a clear handshake, like TCP, then the obvious choice is to declare that node "connected" when the last packet of the three-way handshake has been received. If the only protocol being used is an "unconnected" protocol, like UDP, the implementation may consider the node fully "connected" the moment it determines a route is present, before sending any packets on the network, see further Section 4.7.

For protocol stacks with multiple handshakes, the decision becomes more nuanced. If the protocol stack involves both TLS and TCP, an implementation could determine that a leaf node is connected after the TCP handshake is complete, or it can wait for the TLS handshake to complete as well. The benefit of declaring completion when the TCP handshake finishes, and thus stopping the race for other branches of the tree, is that there will be less burden on the network from other connection attempts. On the other hand, by waiting until the TLS handshake is complete, an implementation avoids the scenario in which a TCP handshake completes quickly, but TLS negotiation is either very slow or fails altogether in particular network conditions or to a particular endpoint. To avoid the issue of TLS possibly failing, the implementation should not generate a Ready event for the Connection until TLS is established.

If all of the leaf nodes fail to connect during racing, i.e. none of the configurations that satisfy all requirements given in the

Transport Parameters actually work over the available paths, then the transport system should notify the application with an `InitiateError` event. An `InitiateError` event should also be generated in case the transport system finds no usable candidates to race.

#### 4.6. Establishing multiplexed connections

Multiplexing several Connections over a single underlying transport connection requires that the Connections to be multiplexed belong to the same Connection Group (as is indicated by the application using the `Clone` call). When the underlying transport connection supports multi-streaming, the Transport System can map each Connection in the Connection Group to a different stream. Thus, when the Connections that are offered to an application by the Transport System are multiplexed, the Transport System may implement the establishment of a new Connection by simply beginning to use a new stream of an already established transport connection and there is no need for a connection establishment procedure. This, then, also means that there may not be any "establishment" message (like a TCP SYN), but the application can simply start sending or receiving. Therefore, when the `Initiate` action of a Transport System is called without Messages being handed over, it cannot be guaranteed that the other endpoint will have any way to know about this, and hence a passive endpoint's `ConnectionReceived` event may not be called upon an active endpoint's `Initiate`. Instead, calling the `ConnectionReceived` event may be delayed until the first Message arrives.

#### 4.7. Handling racing with "unconnected" protocols

While protocols that use an explicit handshake to validate a Connection to a peer can be used for racing multiple establishment attempts in parallel, "unconnected" protocols such as raw UDP do not offer a way to validate the presence of a peer or the usability of a Connection without application feedback. An implementation should consider such a protocol stack to be established as soon as a local route to the peer endpoint is confirmed.

However, if a peer is not reachable over the network using the unconnected protocol, or data cannot be exchanged for any other reason, the application may want to attempt using another candidate Protocol Stack. The implementation should maintain the list of other candidate Protocol Stacks that were eligible to use. In the case that the application signals that the initial Protocol Stack is failing for some reason and that another option should be attempted, the Connection can be updated to point to the next candidate Protocol Stack. This can be viewed as an application-driven form of Protocol Stack racing.

#### 4.8. Implementing listeners

When an implementation is asked to Listen, it registers with the system to wait for incoming traffic to the Local Endpoint. If no Local Endpoint is specified, the implementation should either use an ephemeral port or generate an error.

If the Path Selection Properties do not require a single network interface or path, but allow the use of multiple paths, the Listener object should register for incoming traffic on all of the network interfaces or paths that conform to the Path Selection Properties. The set of available paths can change over time, so the implementation should monitor network path changes and register and de-register the Listener across all usable paths. When using multiple paths, the Listener is generally expected to use the same port for listening on each.

If the Protocol Selection Properties allow multiple protocols to be used for listening, and the implementation supports it, the Listener object should register across the eligible protocols for each path. This means that inbound Connections delivered by the implementation may have heterogeneous protocol stacks.

##### 4.8.1. Implementing listeners for Connected Protocols

Connected protocols such as TCP and TLS-over-TCP have a strong mapping between the Local and Remote Endpoints (five-tuple) and their protocol connection state. These map well into Connection objects. Whenever a new inbound handshake is being started, the Listener should generate a new Connection object and pass it to the application.

##### 4.8.2. Implementing listeners for Unconnected Protocols

Unconnected protocols such as UDP and UDP-lite generally do not provide the same mechanisms that connected protocols do to offer Connection objects. Implementations should wait for incoming packets for unconnected protocols on a listening port and should perform five-tuple matching of packets to either existing Connection objects or the creation of new Connection objects. On platforms with facilities to create a "virtual connection" for unconnected protocols implementations should use these mechanisms to minimise the handling of datagrams intended for already created Connection objects.

#### 4.8.3. Implementing listeners for Multiplexed Protocols

Protocols that provide multiplexing of streams into a single five-tuple can listen both for entirely new connections (a new HTTP/2 stream on a new TCP connection, for example) and for new sub-connections (a new HTTP/2 stream on an existing connection). If the abstraction of Connection presented to the application is mapped to the multiplexed stream, then the Listener should deliver new Connection objects in the same way for either case. The implementation should allow the application to introspect the Connection Group marked on the Connections to determine the grouping of the multiplexing.

### 5. Implementing Data Transfer

#### 5.1. Data transfer for streams, datagrams, and frames

The most basic mapping for sending a Message is an abstraction of datagrams, in which the transport protocol naturally deals in discrete packets. Each Message here corresponds to a single datagram. Generally, these will be short enough that sending and receiving will always use a complete Message.

For protocols that expose byte-streams, the only delineation provided by the protocol is the end of the stream in a given direction. Each Message in this case corresponds to the entire stream of bytes in a direction. These Messages may be quite long, in which case they can be sent in multiple parts.

Protocols that provide the framing (such as length-value protocols, or protocols that use delimiters) provide data boundaries that may be longer than a traditional packet datagram. Each Message for framing protocols corresponds to a single frame, which may be sent either as a complete Message, or in multiple parts.

##### 5.1.1. Sending Messages

The effect of the application sending a Message is determined by the top-level protocol in the established Protocol Stack. That is, if the top-level protocol provides an abstraction of framed messages over a connection, the receiving application will be able to obtain multiple Messages on that connection, even if the framing protocol is built on a byte-stream protocol like TCP.

## 5.1.1.1. Send Parameters

- o Lifetime: this should be implemented by removing the Message from its queue of pending Messages after the Lifetime has expired. A queue of pending Messages within the transport system implementation that have yet to be handed to the Protocol Stack can always support this property, but once a Message has been sent into the send buffer of a protocol, only certain protocols may support de-queueing a message. For example, TCP cannot remove bytes from its send buffer, while in case of SCTP, such control over the SCTP send buffer can be exercised using the partial reliability extension [RFC8303]. When there is no standing queue of Messages within the system, and the Protocol Stack does not support removing a Message from its buffer, this property may be ignored.
- o Niceness: this represents the ability to de-prioritize a Message in favor of other Messages. This can be implemented by the system re-ordering Messages that have yet to be handed to the Protocol Stack, or by giving relative priority hints to protocols that support priorities per Message. For example, an implementation of HTTP/2 could choose to send Messages of different niceness on streams of different priority.
- o Ordered: when this is false, it disables the requirement of in-order-delivery for protocols that support configurable ordering.
- o Idempotent: when this is true, it means that the Message can be used by mechanisms that might transfer it multiple times - e.g., as a result of racing multiple transports or as part of TCP Fast Open.
- o Corruption Protection Length: when this is set to any value other than -1, it limits the required checksum in protocols that allow limiting the checksum length (e.g. UDP-Lite).
- o Immediate Acknowledgement: this informs the implementation that the sender intends to execute tight control over the send buffer, and therefore wants to avoid delayed acknowledgements. In case of SCTP, a request to immediately send acknowledgements can be implemented using the "sack-immediately flag" described in Section 4.2 of [RFC8303] for the SEND.SCTP primitive.
- o Instantaneous Capacity Profile: when this is set to "Interactive/Low Latency", the Message should be sent immediately, even when this comes at the cost of using the network capacity less efficiently. For example, small messages can sometimes be bundled to fit into a single data packet for the sake of reducing header

overhead; such bundling should not be used. For example, in case of TCP, the Nagle algorithm should be disabled when Interactive/Low Latency is selected as the capacity profile. Scavenger/Bulk can translate into usage of a congestion control mechanism such as LEDBAT, and/or the capacity profile can lead to a choice of a DSCP value as described in [I-D.ietf-taps-minset]).

[Note: See also Appendix A.2 for additional Send Parameters under discussion.]

#### 5.1.1.2. Send Completion

The application should be notified whenever a Message or partial Message has been consumed by the Protocol Stack, or has failed to send. The meaning of the Message being consumed by the stack may vary depending on the protocol. For a basic datagram protocol like UDP, this may correspond to the time when the packet is sent into the interface driver. For a protocol that buffers data in queues, like TCP, this may correspond to when the data has entered the send buffer.

#### 5.1.1.3. Batching Sends

Since sending a Message may involve a context switch between the application and the transport system, sending patterns that involve multiple small Messages can incur high overhead if each needs to be enqueued separately. To avoid this, the application should have a way to indicate a batch of Send actions, during which time the implementation will hold off on processing Messages until the batch is complete. This can also help context switches when enqueueing data in the interface driver if the operation can be batched.

#### 5.1.2. Receiving Messages

Similar to sending, Receiving a Message is determined by the top-level protocol in the established Protocol Stack. The main difference with Receiving is that the size and boundaries of the Message are not known beforehand. The application can communicate in its Receive action the parameters for the Message, which can help the implementation know how much data to deliver and when. For example, if the application only wants to receive a complete Message, the implementation should wait until an entire Message (datagram, stream, or frame) is read before delivering any Message content to the application. This requires the implementation to understand where messages end, either via a supplied deframer or because the top-level protocol in the established Protocol Stack preserves message boundaries; if, on the other hand, the top-level protocol only supports a byte-stream and no deframers were supported, the

application must specify the minimum number of bytes of Message content it wants to receive (which may be just a single byte) to control the flow of received data.

If a Connection becomes finished before a requested Receive action can be satisfied, the implementation should deliver any partial Message content outstanding, or if none is available, an indication that there will be no more received Messages.

## 5.2. Handling of data for fast-open protocols

Several protocols allow sending higher-level protocol or application data within the first packet of their protocol establishment, such as TCP Fast Open [RFC7413] and TLS 1.3 [I-D.ietf-tls-tls13]. This approach is referred to as sending Zero-RTT (0-RTT) data. This is a desirable property, but poses challenges to an implementation that uses racing during connection establishment.

If the application has 0-RTT data to send in any protocol handshakes, it needs to provide this data before the handshakes have begun. When racing, this means that the data should be provided before the process of connection establishment has begun. If the application wants to send 0-RTT data, it must indicate this to the implementation by setting the Idempotent send parameter to true when sending the data. In general, 0-RTT data may be replayed (for example, if a TCP SYN contains data, and the SYN is retransmitted, the data will be retransmitted as well), but racing means that different leaf nodes have the opportunity to send the same data independently. If data is truly idempotent, this should be permissible.

Once the application has provided its 0-RTT data, an implementation should keep a copy of this data and provide it to each new leaf node that is started and for which a 0-RTT protocol is being used.

It is also possible that protocol stacks within a particular leaf node use 0-RTT handshakes without any idempotent application data. For example, TCP Fast Open could use a Client Hello from TLS as its 0-RTT data, shortening the cumulative handshake time.

0-RTT handshakes often rely on previous state, such as TCP Fast Open cookies, previously established TLS tickets, or out-of-band distributed pre-shared keys (PSKs). Implementations should be aware of security concerns around using these tokens across multiple addresses or paths when racing. In the case of TLS, any given ticket or PSK should only be used on one leaf node. If implementations have multiple tickets available from a previous connection, each leaf node attempt must use a different ticket. In effect, each leaf node will



send the same early application data, yet encoded (encrypted) differently on the wire.

## 6. Implementing Maintenance

Maintenance encompasses changes that the application can request to a Connection, or that a Connection can react to based on system and network changes.

### 6.1. Changing Protocol Properties

Appendix A.1 of [I-D.ietf-taps-minset] explains, using primitives that are described in [RFC8303] and [RFC8304], how to implement changing the following protocol properties of an established connection with TCP and UDP. Below, we amend this description for other protocols (if applicable):

- o Relative niceness: for SCTP, this can be done using the primitive `CONFIGURE_STREAM_SCHEDULER.SCTP` described in section 4 of [RFC8303].
- o Timeout for aborting Connection: for SCTP, this can be done using the primitive `CHANGE_TIMEOUT.SCTP` described in section 4 of [RFC8303].
- o Abort timeout to suggest to the Remote Endpoint: for TCP, this can be done using the primitive `CHANGE_TIMEOUT.TCP` described in section 4 of [RFC8303].
- o Retransmission threshold before excessive retransmission notification: for TCP, this can be done using `ERROR.TCP` described in section 4 of [RFC8303].
- o Required minimum coverage of the checksum for receiving: for UDP-Lite, this can be done using the primitive `SET_MIN_CHECKSUM_COVERAGE.UDP-Lite` described in section 4 of [RFC8303].
- o Connection group transmission scheduler: for SCTP, this can be done using the primitive `SET_STREAM_SCHEDULER.SCTP` described in section 4 of [RFC8303].

It may happen that the application attempts to set a Protocol Property which does not apply to the actually chosen protocol. In this case, the implementation should fail gracefully, i.e., it may give a warning to the application, but it should not terminate the Connection.

## 6.2. Handling Path Changes

When a path change occurs, the Transport Services implementation is responsible for notifying Protocol Instances in the Protocol Stack. If the Protocol Stack includes a transport protocol that supports multipath connectivity, an update to the available paths should inform the Protocol Instance of the new set of paths that are permissible based on the Path Selection Properties passed by the application. A multipath protocol can establish new subflows over new paths, and should tear down subflows over paths that are no longer available. If the Protocol Stack includes a transport protocol that does not support multipath, but support migrating between paths, the update to available paths can be used as the trigger to migrating the connection. For protocols that do not support multipath or migration, the Protocol Instances may be informed of the path change, but should not be forcibly disconnected if the previously used path becomes unavailable. An exception to this case is if the System Policy changes to prohibit traffic from the Connection based on its properties, in which case the Protocol Stack should be disconnected.

## 7. Implementing Termination

With TCP, when an application closes a connection, this means that it has no more data to send (but expects all data that has been handed over to be reliably delivered). However, with TCP only, "close" does not mean that the application will stop receiving data. This is related to TCP's ability to support half-closed connections.

SCTP is an example of a protocol that does not support such half-closed connections. Hence, with SCTP, the meaning of "close" is stricter: an application has no more data to send (but expects all data that has been handed over to be reliably delivered), and will also not receive any more data.

Implementing a protocol independent transport system means that the exposed semantics must be the strictest subset of the semantics of all supported protocols. Hence, as is common with all reliable transport protocols, after a Close action, the application can expect to have its reliability requirements honored regarding the data it has given to the Transport System, but it cannot expect to be able to read any more data after calling Close.

Abort differs from Close only in that no guarantees are given regarding data that the application has handed over to the Transport System before calling Abort.

As explained in section Section 4.6, when a new stream is multiplexed on an already existing connection of a Transport Protocol Instance, there is no need for a connection establishment procedure. Because the Connections that are offered by the Transport System can be implemented as streams that are multiplexed on a transport protocol's connection, it can therefore not be guaranteed that one Endpoint's Initiate action provokes a ConnectionReceived event at its peer.

For Close (provoking a Finished event) and Abort (provoking a ConnectionError event), the same logic applies: while it is desirable to be informed when a peer closes or aborts a Connection, whether this is possible depends on the underlying protocol, and no guarantees can be given. With SCTP, the transport system can use the stream reset procedure to cause a Finish event upon a Close action from the peer [NEAT-flow-mapping].

## 8. Cached State

Beyond a single Connection's lifetime, it is useful for an implementation to keep state and history. This cached state can help improve future Connection establishment due to re-using results and credentials, and favoring paths and protocols that performed well in the past.

Cached state may be associated with different Endpoints for the same Connection, depending on the protocol generating the cached content. For example, session tickets for TLS are associated with specific endpoints, and thus should be cached based on a Connection's hostname Endpoint (if applicable). On the other hand, performance characteristics of a path are more likely tied to the IP address and subnet being used.

### 8.1. Protocol state caches

Some protocols will have long-term state to be cached in association with Endpoints. This state often has some time after which it is expired, so the implementation should allow each protocol to specify an expiration for cached content.

Examples of cached protocol state include:

- o The DNS protocol can cache resolution answers (A and AAAA queries, for example), associated with a Time To Live (TTL) to be used for future hostname resolutions without requiring asking the DNS resolver again.
- o TLS caches session state and tickets based on a hostname, which can be used for resuming sessions with a server.

- o TCP can cache cookies for use in TCP Fast Open.

Cached protocol state is primarily used during Connection establishment for a single Protocol Stack, but may be used to influence an implementation's preference between several candidate Protocol Stacks. For example, if two IP address Endpoints are otherwise equally preferred, an implementation may choose to attempt a connection to an address for which it has a TCP Fast Open cookie.

Applications must have a way to flush protocol cache state if desired. This may be necessary, for example, if application-layer identifiers rotate and clients wish to avoid linkability via trackable TLS tickets or TFO cookies.

## 8.2. Performance caches

In addition to protocol state, Protocol Instances should provide data into a performance-oriented cache to help guide future protocol and path selection. Some performance information can be gathered generically across several protocols to allow predictive comparisons between protocols on given paths:

- o Observed Round Trip Time
- o Connection Establishment latency
- o Connection Establishment success rate

These items can be cached on a per-address and per-subnet granularity, and averaged between different values. The information should be cached on a per-network basis, since it is expected that different network attachments will have different performance characteristics. Besides Protocol Instances, other system entities may also provide data into performance-oriented caches. This could for instance be signal strength information reported by radio modems like Wi-Fi and mobile broadband or information about the battery-level of the device. Furthermore, the system may cache the observed maximum throughput on a path as an estimate of the available bandwidth.

An implementation should use this information, when possible, to determine preference between candidate paths, endpoints, and protocol options. Eligible options that historically had significantly better performance than others should be selected first when gathering candidates (see Section 4.1) to ensure better performance for the application.

The reasonable lifetime for cached performance values will vary depending on the nature of the value. Certain information, like the connection establishment success rate to a Remote Endpoint using a given protocol stack, can be stored for a long period of time (hours or longer), since it is expected that the capabilities of the Remote Endpoint are not changing very quickly. On the other hand, Round Trip Time observed by TCP over a particular network path may vary over a relatively short time interval. For such values, the implementation should remove them from the cache more quickly, or treat older values with less confidence/weight.

## 9. Specific Transport Protocol Considerations

### 9.1. TCP

Connection lifetime for TCP translates fairly simply into the the abstraction presented to an application. When the TCP three-way handshake is complete, its layer of the Protocol Stack can be considered Ready (established). This event will cause racing of Protocol Stack options to complete if TCP is the top-level protocol, at which point the application can be notified that the Connection is Ready to send and receive.

If the application sends a Close, that can translate to a graceful termination of the TCP connection, which is performed by sending a FIN to the remote endpoint. If the application sends an Abort, then the TCP state can be closed abruptly, leading to a RST being sent to the peer.

Without a layer of framing (a top-level protocol in the established Protocol Stack that preserves message boundaries, or an application-supplied deframer) on top of TCP, the receiver side of the transport system implementation can only treat the incoming stream of bytes as a single Message, terminated by a FIN when the Remote Endpoint closes the Connection.

### 9.2. UDP

UDP as a direct transport does not provide any handshake or connectivity state, so the notion of the transport protocol becoming Ready or established is degenerate. Once the system has validated that there is a route on which to send and receive UDP datagrams, the protocol is considered Ready. Similarly, a Close or Abort has no meaning to the on-the-wire protocol, but simply leads to the local state being torn down.

When sending and receiving messages over UDP, each Message should correspond to a single UDP datagram. The Message can contain

metadata about the packet, such as the ECN bits applied to the packet.

### 9.3. SCTP

To support sender-side stream schedulers (which are implemented on the sender side), a receiver-side Transport System should always support message interleaving [RFC8260].

SCTP messages can be very large. To allow the reception of large messages in pieces, a "partial flag" can be used to inform a (native SCTP) receiving application that a message is incomplete. After receiving the "partial flag", this application would know that the next receive calls will only deliver remaining parts of the same message (i.e., no messages or partial messages will arrive on other streams until the message is complete) (see Section 8.1.20 in [RFC6458]). The "partial flag" can therefore facilitate the implementation of the receiver buffer in the receiving application, at the cost of limiting multiplexing and temporarily creating head-of-line blocking delay at the receiver.

When a Transport System transfers a Message, it seems natural to map the Message object to SCTP messages in order to support properties such as "Ordered" or "Lifetime" (which maps onto partially reliable delivery with a SCTP\_PR\_SCTP\_TTL policy [RFC6458]). However, since multiplexing of Connections onto SCTP streams may happen, and would be hidden from the application, the Transport System requires a per-stream receiver buffer anyway, so this potential benefit is lost and the "partial flag" becomes unnecessary for the system.

The problem of long messages either requiring large receiver-side buffers or getting in the way of multiplexing is addressed by message interleaving [RFC8260], which is yet another reason why a receivers-side transport system supporting SCTP should implement this mechanism.

### 9.4. TLS

The mapping of a TLS stream abstraction into the application is equivalent to the contract provided by TCP (see Section 9.1). The Ready state should be determined by the completion of the TLS handshake, which involves potentially several more round trips beyond the TCP handshake. The application should not be notified that the Connection is Ready until TLS is established.

### 9.5. HTTP

HTTP requests and responses map naturally into Messages, since they are delineated chunks of data with metadata that can be sent over a transport. To that end, HTTP can be seen as the most prevalent framing protocol that runs on top of streams like TCP, TLS, etc.

In order to use a transport Connection that provides HTTP Message support, the establishment and closing of the connection can be treated as it would without the framing protocol. Sending and receiving of Messages, however, changes to treat each Message as a well-delineated HTTP request or response, with the content of the Message representing the body, and the Headers being provided in Message metadata.

### 9.6. QUIC

QUIC provides a multi-streaming interface to an encrypted transport. Each stream can be viewed as equivalent to a TLS stream over TCP, so a natural mapping is to present each QUIC stream as an individual Connection. The protocol for the stream will be considered Ready whenever the underlying QUIC connection is established to the point that this stream's data can be sent. For streams after the first stream, this will likely be an immediate operation.

Closing a single QUIC stream, presented to the application as a Connection, does not imply closing the underlying QUIC connection itself. Rather, the implementation may choose to close the QUIC connection once all streams have been closed (possibly after some timeout), or after an individual stream Connection sends an Abort.

Messages over a direct QUIC stream should be represented similarly to the TCP stream (one Message per direction, see Section 9.1), unless a framing mapping is used on top of QUIC.

### 9.7. HTTP/2 transport

Similar to QUIC (Section 9.6), HTTP/2 provides a multi-streaming interface. This will generally use HTTP as the unit of Messages over the streams, in which each stream can be represented as a transport Connection. The lifetime of streams and the HTTP/2 connection should be managed as described for QUIC.

It is possible to treat each HTTP/2 stream as a raw byte-stream instead of a carrier for HTTP messages, in which case the Messages over the streams can be represented similarly to the TCP stream (one Message per direction, see Section 9.1).

## 10. Rendezvous and Environment Discovery

The connection establishment process outlined in Section 4 is appropriate for client-server connections, but needs to be expanded in peer-to-peer Rendezvous scenarios, as follows:

### o Gathering Local Endpoint candidates

The set of possible Local Endpoints is gathered. In the simple case, this merely enumerates the local interfaces and protocols, allocates ephemeral source ports. For example, a system that has WiFi and Ethernet and supports IPv4 and IPv6 might gather four candidate locals (IPv4 on Ethernet, IPv6 on Ethernet, IPv4 on WiFi, and IPv6 on WiFi) that can form the source for a transient.

If NAT traversal is required, the process of gathering Local Endpoints becomes broadly equivalent to the ICE candidate gathering phase [RFC5245]. The endpoint determines its server reflexive Local Endpoints (i.e., the translated address of a local, on the other side of a NAT) and relayed locals (e.g., via a TURN server or other relay), for each interface and network protocol. These are added to the set of candidate Local Endpoints for this connection.

Gathering locals is primarily an endpoint local operation, although it might involve exchanges with a STUN server to derive server reflexive locals, or with a TURN server or other relay to derive relayed locals. It does not involve communication with the Remote Endpoint.

### o Gathering Remote Endpoint Candidates

The Remote Endpoint is typically a name that needs to be resolved into a set of possible addresses that can be used for communication. Resolving the Remote Endpoint is the process of recursively performing such name lookups, until fully resolved, to return the set of candidates for the remote of this connection.

How this is done will depend on the type of the Remote Endpoint, and can also be specific to each Local Endpoint. A common case is when the Remote Endpoint is a DNS name, in which case it is resolved to give a set of IPv4 and IPv6 addresses representing that name. Some types of remote might require more complex resolution. Resolving the Remote Endpoint for a peer-to-peer connection might involve communication with a rendezvous server, which in turn contacts the peer to gain consent to communicate and retrieve its set of candidate locals, which are returned and form the candidate remote addresses for contacting that peer.



Resolving the remote is not a local operation. It will involve a directory service, and can require communication with the remote to rendezvous and exchange peer addresses. This can expose some or all of the candidate locals to the remote.

- o Establishing Connections

The set of candidate Local Endpoints and the set of candidate Remote Endpoints are paired, to derive a priority ordered set of Candidate Paths that can potentially be used to establish a Connection.

Then, communication is attempted over each candidate path, in priority order. If there are multiple candidates with the same priority, then connection establishment proceeds simultaneously and uses the transient that wins the race to be established. Otherwise, connection establishment is sequential, paced at a rate that should not congest the network. Depending on the chosen transport, this phase might involve racing TCP connections to a server over IPv4 and IPv6 [RFC8305], or it could involve a STUN exchange to establish peer-to-peer UDP connectivity [RFC5245], or some other means.

- o Confirming and Maintaining Connections

Once connectivity has been established, unused resources can be released and the chosen path can be confirmed. This is primarily required when establishing peer-to-peer connectivity, where connections supporting relayed locals that were not required can be closed, and where an associated signalling operation might be needed to inform middleboxes and proxies of the chosen path. Keep-alive messages may also be sent, as appropriate, to ensure NAT and firewall state is maintained, so the Connection remains operational.

To support ICE, or similar protocols, that involve an out-of-band indirect signalling exchange to exchange candidates with the Remote Endpoint, it's important to be able to query the set of candidate Local Endpoints, and give the protocol stack a set of candidate Remote Endpoints, before it attempts to establish connections.

(TO-DO: It is expected that a single abstract algorithm can be identified that supports both the peer-to-peer and client-server connection racing, allowing this text to be merged with Section 4)

## 11. IANA Considerations

RFC-EDITOR: Please remove this section before publication.

This document has no actions for IANA.

## 12. Security Considerations

### 12.1. Considerations for Candidate Gathering

Implementations should avoid downgrade attacks that allow network interference to cause the implementation to select less secure, or entirely insecure, combinations of paths and protocols.

### 12.2. Considerations for Candidate Racing

See Section 5.2 for security considerations around racing with 0-RTT data.

An attacker that knows a particular device is racing several options during connection establishment may be able to block packets for the first connection attempt, thus inducing the device to fall back to a secondary attempt. This is a problem if the secondary attempts have worse security properties that enable further attacks. Implementations should ensure that all options have equivalent security properties to avoid incentivizing attacks.

Since results from the network can determine how a connection attempt tree is built, such as when DNS returns a list of resolved endpoints, it is possible for the network to cause an implementation to consume significant on-device resources. Implementations should limit the maximum amount of state allowed for any given node, including the number of child nodes, especially when the state is based on results from the network.

## 13. Acknowledgements

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

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.

## 14. References

### 14.1. 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-00 (work in progress), April 2018.

#### [I-D.ietf-taps-interface]

Trammell, B., Welzl, M., Enghardt, T., Fairhurst, G., Kuehlewind, M., Perkins, C., Tiesel, P., and C. Wood, "An Abstract Application Layer Interface to Transport Services", draft-ietf-taps-interface-00 (work in progress), April 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.

[RFC6458] Stewart, R., Tuexen, M., Poon, K., Lei, P., and V. Yasevich, "Sockets API Extensions for the Stream Control Transmission Protocol (SCTP)", RFC 6458, DOI 10.17487/RFC6458, December 2011, <<https://www.rfc-editor.org/info/rfc6458>>.

[RFC7413] Cheng, Y., Chu, J., Radhakrishnan, S., and A. Jain, "TCP Fast Open", RFC 7413, DOI 10.17487/RFC7413, December 2014, <<https://www.rfc-editor.org/info/rfc7413>>.

[RFC7540] Belshe, M., Peon, R., and M. Thomson, Ed., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, DOI 10.17487/RFC7540, May 2015, <<https://www.rfc-editor.org/info/rfc7540>>.

[RFC8260] Stewart, R., Tuexen, M., Loreto, S., and R. Seggelmann, "Stream Schedulers and User Message Interleaving for the Stream Control Transmission Protocol", RFC 8260, DOI 10.17487/RFC8260, November 2017, <<https://www.rfc-editor.org/info/rfc8260>>.

- [RFC8303] Welzl, M., Tuexen, M., and N. Khademi, "On the Usage of Transport Features Provided by IETF Transport Protocols", RFC 8303, DOI 10.17487/RFC8303, February 2018, <<https://www.rfc-editor.org/info/rfc8303>>.
- [RFC8304] Fairhurst, G. and T. Jones, "Transport Features of the User Datagram Protocol (UDP) and Lightweight UDP (UDP-Lite)", RFC 8304, DOI 10.17487/RFC8304, February 2018, <<https://www.rfc-editor.org/info/rfc8304>>.
- [RFC8305] Schinazi, D. and T. Pauly, "Happy Eyeballs Version 2: Better Connectivity Using Concurrency", RFC 8305, DOI 10.17487/RFC8305, December 2017, <<https://www.rfc-editor.org/info/rfc8305>>.

#### 14.2. Informative References

- [I-D.ietf-quic-transport]  
Iyengar, J. and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport", draft-ietf-quic-transport-13 (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.
- [NEAT-flow-mapping]  
"Transparent Flow Mapping for NEAT (in Workshop on Future of Internet Transport (FIT 2017))", n.d..
- [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>>.
- [Trickle] "Trickle - Rate Limiting YouTube Video Streaming (ATC 2012)", n.d..

#### Appendix A. Additional Properties

This appendix discusses implementation considerations for 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. Properties Affecting Sorting of Branches

In addition to the Protocol and Path Selection Properties discussed in Section 4.3, the following properties under discussion can influence branch sorting:

- o Size to be Sent or Received: An implementation may use the Size to be Sent or Received in combination with cached performance estimates, see Section 8.2, e.g. the observed Round Trip Time and the observed maximum throughput, to compute an estimate of the completion time of a transfer over different available paths. It may then prefer the path with the shorter expected completion time. This property may be used instead of the Capacity profile, as the application does not always know whether its transfer will be latency-bound or bandwidth-bound, and thus may not be able to specify a Capacity Profile. However, the application may know the Size to be Sent or Received from metadata, e.g., in adaptive HTTP streaming such as MPEG-DASH, or in operating system upgrades. A related paper is currently under submission.
- o Send / Receive Bitrate: If the application indicates an expected send or receive bitrate, an implementation may prefer a path that can likely provide the desired bandwidth, based on cached maximum throughput, see Section 8.2. The application may know the Send or Receive Bitrate from metadata in adaptive HTTP streaming, such as MPEG-DASH.
- o Cost Preferences: If the application indicates a preference to avoid expensive paths, and some paths are associated with a monetary cost, an implementation should decrease the ranking of such paths. If the application indicates that it prohibits using expensive paths, paths that are associated with a cost should be purged from the decision tree.

#### A.2. Send Parameters

In addition to the Send Parameters listed in Section 5.1.1.1, the following Send Parameters are under discussion:

- o Send Bitrate: If an application indicates a certain bitrate it wants to send on the connection, the implementation may limit the bitrate of the outgoing communication to that rate, for example by setting an upper bound for the TCP congestion window of a connection calculated from the Send Bitrate and the Round Trip

Time. This helps to avoid bursty traffic patterns on video streaming servers, see [Trickle].

#### Authors' Addresses

Anna Brunstrom (editor)  
Karlstad University  
Universitetsgatan 2  
651 88 Karlstad  
Sweden

Email: [anna.brunstrom@kau.se](mailto:anna.brunstrom@kau.se)

Tommy Pauly (editor)  
Apple Inc.  
One Apple Park Way  
Cupertino, California 95014  
United States of America

Email: [tpauly@apple.com](mailto:tpauly@apple.com)

Theresa Enghardt  
TU Berlin  
Marchstrasse 23  
10587 Berlin  
Germany

Email: [theresa@inet.tu-berlin.de](mailto:theresa@inet.tu-berlin.de)

Karl-Johan Grinnemo  
Karlstad University  
Universitetsgatan 2  
651 88 Karlstad  
Sweden

Email: [karl-johan.grinnemo@kau.se](mailto:karl-johan.grinnemo@kau.se)

Tom Jones  
University of Aberdeen  
Fraser Noble Building  
Aberdeen, AB24 3UE  
UK

Email: [tom@erg.abdn.ac.uk](mailto:tom@erg.abdn.ac.uk)

Philipp S. Tiesel  
TU Berlin  
Marchstrasse 23  
10587 Berlin  
Germany

Email: philipp@inet.tu-berlin.de

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

Email: csp@csp Perkins.org

Michael Welzl  
University of Oslo  
PO Box 1080 Blindern  
0316 Oslo  
Norway

Email: michawe@ifi.uio.no

TAPS Working Group  
Internet-Draft  
Intended status: Informational  
Expires: 8 September 2022

A. Brunstrom, Ed.  
Karlstad University  
T. Pauly, Ed.  
Apple Inc.  
T. Enghardt  
Netflix  
P. Tiesel  
SAP SE  
M. Welzl  
University of Oslo  
7 March 2022

Implementing Interfaces to Transport Services  
draft-ietf-taps-impl-12

Abstract

The Transport Services system enables applications to use transport protocols flexibly for network communication and defines a protocol-independent Transport Services Application Programming Interface (API) that is based on an asynchronous, event-driven interaction pattern. This document serves as a guide to implementation on how to build such a system.

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 8 September 2022.

Copyright Notice

Copyright (c) 2022 IETF Trust and the persons identified as the document authors. All rights reserved.



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

## Table of Contents

1. Introduction . . . . .	3
2. Implementing Connection Objects . . . . .	4
3. Implementing Pre-Establishment . . . . .	5
3.1. Configuration-time errors . . . . .	5
3.2. Role of system policy . . . . .	6
4. Implementing Connection Establishment . . . . .	7
4.1. Structuring Candidates as a Tree . . . . .	8
4.1.1. Branch Types . . . . .	10
4.1.2. Branching Order-of-Operations . . . . .	12
4.1.3. Sorting Branches . . . . .	14
4.2. Candidate Gathering . . . . .	15
4.2.1. Gathering Endpoint Candidates . . . . .	15
4.3. Candidate Racing . . . . .	17
4.3.1. Simultaneous . . . . .	17
4.3.2. Staggered . . . . .	18
4.3.3. Failover . . . . .	19
4.4. Completing Establishment . . . . .	19
4.4.1. Determining Successful Establishment . . . . .	20
4.5. Establishing multiplexed connections . . . . .	21
4.6. Handling connectionless protocols . . . . .	21
4.7. Implementing listeners . . . . .	21
4.7.1. Implementing listeners for Connected Protocols . . . . .	22
4.7.2. Implementing listeners for Connectionless Protocols . . . . .	22
4.7.3. Implementing listeners for Multiplexed Protocols . . . . .	22
5. Implementing Sending and Receiving Data . . . . .	23
5.1. Sending Messages . . . . .	23
5.1.1. Message Properties . . . . .	23
5.1.2. Send Completion . . . . .	25
5.1.3. Batching Sends . . . . .	25
5.2. Receiving Messages . . . . .	25
5.3. Handling of data for fast-open protocols . . . . .	26
6. Implementing Message Framers . . . . .	27
6.1. Defining Message Framers . . . . .	28
6.2. Sender-side Message Framing . . . . .	29
6.3. Receiver-side Message Framing . . . . .	30
7. Implementing Connection Management . . . . .	31

7.1.	Pooled Connection . . . . .	31
7.2.	Handling Path Changes . . . . .	32
8.	Implementing Connection Termination . . . . .	33
9.	Cached State . . . . .	34
9.1.	Protocol state caches . . . . .	34
9.2.	Performance caches . . . . .	35
10.	Specific Transport Protocol Considerations . . . . .	36
10.1.	TCP . . . . .	37
10.2.	MPTCP . . . . .	39
10.3.	UDP . . . . .	39
10.4.	UDP-Lite . . . . .	40
10.5.	UDP Multicast Receive . . . . .	40
10.6.	SCTP . . . . .	42
11.	IANA Considerations . . . . .	45
12.	Security Considerations . . . . .	45
12.1.	Considerations for Candidate Gathering . . . . .	45
12.2.	Considerations for Candidate Racing . . . . .	45
13.	Acknowledgements . . . . .	46
14.	References . . . . .	46
14.1.	Normative References . . . . .	46
14.2.	Informative References . . . . .	47
Appendix A.	API Mapping Template . . . . .	49
Appendix B.	Additional Properties . . . . .	50
B.1.	Properties Affecting Sorting of Branches . . . . .	50
Appendix C.	Reasons for errors . . . . .	51
Appendix D.	Existing Implementations . . . . .	52
Authors' Addresses	. . . . .	52

## 1. Introduction

The Transport Services architecture [I-D.ietf-taps-arch] defines a system that allows applications to flexibly use transport networking protocols. The API that such a system exposes to applications is defined as the Transport Services API [I-D.ietf-taps-interface]. This API is designed to be generic across multiple transport protocols and sets of protocols features.

This document serves as a guide to implementation on how to build a system that provides a Transport Services API. It is the job of an implementation of a Transport Services system to turn the requests of an application into decisions on how to establish connections, and how to transfer data over those connections once established. The terminology used in this document is based on the Architecture [I-D.ietf-taps-arch].

## 2. Implementing Connection Objects

The connection objects that are exposed to applications for Transport Services are:

- \* the Preconnection, the bundle of Properties that describes the application constraints on, and preferences for, the transport;
- \* the Connection, the basic object that represents a flow of data as Messages in either direction between the Local and Remote Endpoints;
- \* and the Listener, a passive waiting object that delivers new Connections.

Preconnection objects should be implemented as bundles of properties that an application can both read and write. A Preconnection object influences a Connection only at one point in time: when the Connection is created. Connection objects represent the interface between the application and the implementation to manage transport state, and conduct data transfer. During the process of establishment (Section 4), the Connection will not be bound to a specific transport protocol instance, since multiple candidate Protocol Stacks might be raced.

Once a Preconnection has been used to create an outbound Connection or a Listener, the implementation should ensure that the copy of the properties held by the Connection or Listener cannot be mutated by the application making changes to the original Preconnection object. This may involve the implementation performing a deep-copy, copying the object with all the objects that it references.

Once the Connection is established, Transport Services implementation maps actions and events to the details of the chosen Protocol Stack. For example, the same Connection object may ultimately represent a single instance of one transport protocol (e.g., a TCP connection, a TLS session over TCP, a UDP flow with fully-specified Local and Remote Endpoints, a DTLS session, a SCTP stream, a QUIC stream, or an HTTP/2 stream). The properties held by a Connection or Listener is independent of other connections that are not part of the same Connection Group.

Connection establishment is only a local operation for a Datagram transport (e.g., UDP(-Lite)), which serves to simplify the local send/receive functions and to filter the traffic for the specified addresses and ports [RFC8085].

Once Initiate has been called, the Selection Properties and Endpoint information are immutable (i.e., an application is not able to later modify Selection Properties on the original Preconnection object). Listener objects are created with a Preconnection, at which point their configuration should be considered immutable by the implementation. The process of listening is described in Section 4.7.

### 3. Implementing Pre-Establishment

During pre-establishment the application specifies one or more Endpoints to be used for communication as well as protocol preferences and constraints via Selection Properties and, if desired, also Connection Properties. Generally, Connection Properties should be configured as early as possible, because they can serve as input to decisions that are made by the implementation (e.g., the Capacity Profile can guide usage of a protocol offering scavenger-type congestion control).

The implementation stores these properties as a part of the Preconnection object for use during connection establishment. For Selection Properties that are not provided by the application, the implementation must use the default values specified in the Transport Services API ([I-D.ietf-taps-interface]).

#### 3.1. Configuration-time errors

The Transport Services system should have a list of supported protocols available, which each have transport features reflecting the capabilities of the protocol. Once an application specifies its Transport Properties, the transport system matches the required and prohibited properties against the transport features of the available protocols.

In the following cases, failure should be detected during pre-establishment:

- \* A request by an application for Protocol Properties that cannot be satisfied by any of the available protocols. For example, if an application requires "Configure Reliability per Message", but no such feature is available in any protocol the host running the transport system on the host running the transport system this should result in an error, e.g., when SCTP is not supported by the operating system.
- \* A request by an application for Protocol Properties that are in conflict with each other, i.e., the required and prohibited properties cannot be satisfied by the same protocol. For example,

if an application prohibits "Reliable Data Transfer" but then requires "Configure Reliability per Message", this mismatch should result in an error.

To avoid allocating resources that are not finally needed, it is important that configuration-time errors fail as early as possible.

### 3.2. Role of system policy

The properties specified during pre-establishment have a close relationship to system policy. The implementation is responsible for combining and reconciling several different sources of preferences when establishing Connections. These include, but are not limited to:

1. Application preferences, i.e., preferences specified during the pre-establishment via Selection Properties.
2. Dynamic system policy, i.e., policy compiled from internally and externally acquired information about available network interfaces, supported transport protocols, and current/previous Connections. Examples of ways to externally retrieve policy-support information are through OS-specific statistics/measurement tools and tools that reside on middleboxes and routers.
3. Default implementation policy, i.e., predefined policy by OS or application.

In general, any protocol or path used for a connection must conform to all three sources of constraints. A violation that occurs at any of the policy layers should cause a protocol or path to be considered ineligible for use. For an example of application preferences leading to constraints, an application may prohibit the use of metered network interfaces for a given Connection to avoid user cost. Similarly, the system policy at a given time may prohibit the use of such a metered network interface from the application's process. Lastly, the implementation itself may default to disallowing certain network interfaces unless explicitly requested by the application and allowed by the system.

It is expected that the database of system policies and the method of looking up these policies will vary across various platforms. An implementation should attempt to look up the relevant policies for the system in a dynamic way to make sure it is reflecting an accurate version of the system policy, since the system's policy regarding the application's traffic may change over time due to user or administrative changes.

#### 4. Implementing Connection Establishment

The process of establishing a network connection begins when an application expresses intent to communicate with a Remote Endpoint by calling Initiate. (At this point, any constraints or requirements the application may have on the connection are available from pre-establishment.) The process can be considered complete once there is at least one Protocol Stack that has completed any required setup to the point that it can transmit and receive the application's data.

Connection establishment is divided into two top-level steps: Candidate Gathering, to identify the paths, protocols, and endpoints to use, and Candidate Racing (see Section 4.2.2 of [I-D.ietf-taps-arch]), in which the necessary protocol handshakes are conducted so that the transport system can select which set to use.

This document structures the candidates for racing as a tree as terminological convention. While a tree structure is not the only way in which racing can be implemented, it does ease the illustration of how racing works.

The most simple example of this process might involve identifying the single IP address to which the implementation wishes to connect, using the system's current default path (i.e., using the default interface), and starting a TCP handshake to establish a stream to the specified IP address. However, each step may also differ depending on the requirements of the connection: if the endpoint is defined as a hostname and port, then there may be multiple resolved addresses that are available; there may also be multiple paths available, (in this case using an interface other than the default system interface); and some protocols may not need any transport handshake to be considered "established" (such as UDP), while other connections may utilize layered protocol handshakes, such as TLS over TCP.

Whenever an implementation has multiple options for connection establishment, it can view the set of all individual connection establishment options as a single, aggregate connection establishment. The aggregate set conceptually includes every valid combination of endpoints, paths, and protocols. As an example, consider an implementation that initiates a TCP connection to a hostname + port endpoint, and has two valid interfaces available (Wi-Fi and LTE). The hostname resolves to a single IPv4 address on the Wi-Fi network, and resolves to the same IPv4 address on the LTE network, as well as a single IPv6 address. The aggregate set of connection establishment options can be viewed as follows:

```
Aggregate [Endpoint: www.example.com:80] [Interface: Any] [Protocol: TCP]
|-> [Endpoint: 192.0.2.1:80] [Interface: Wi-Fi] [Protocol: TCP]
|-> [Endpoint: 192.0.2.1:80] [Interface: LTE] [Protocol: TCP]
|-> [Endpoint: 2001:DB8::1:80] [Interface: LTE] [Protocol: TCP]
```

Any one of these sub-entries on the aggregate connection attempt would satisfy the original application intent. The concern of this section is the algorithm defining which of these options to try, when, and in what order.

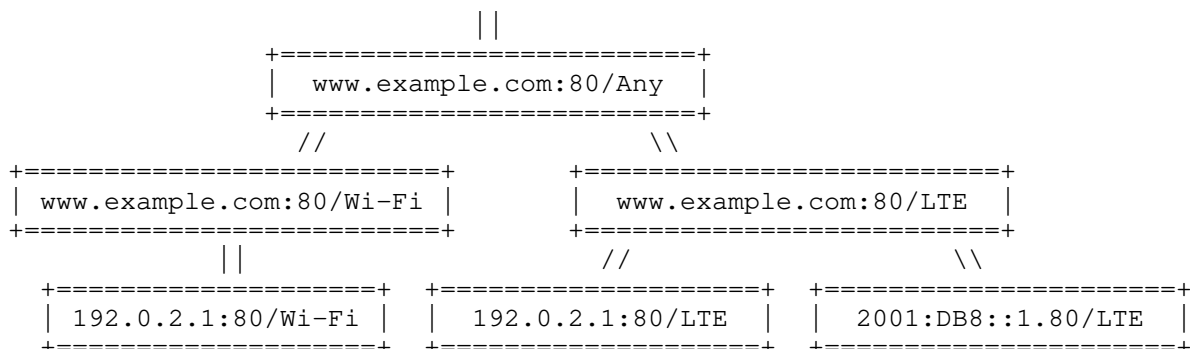
During Candidate Gathering, an implementation first excludes all protocols and paths that match a Prohibit or do not match all Require properties. Then, the implementation will sort branches according to Preferred properties, Avoided properties, and possibly other criteria.

#### 4.1. Structuring Candidates as a Tree

As noted above, the consideration of multiple candidates in a gathering and racing process can be conceptually structured as a tree; this terminological convention is used throughout this document.

Each leaf node of the tree represents a single, coherent connection attempt, with an endpoint, a network path, and a set of protocols that can directly negotiate and send data on the network. Each node in the tree that is not a leaf represents a connection attempt that is either underspecified, or else includes multiple distinct options. For example, when connecting on an IP network, a connection attempt to a hostname and port is underspecified, because the connection attempt requires a resolved IP address as its Remote Endpoint. In this case, the node represented by the connection attempt to the hostname is a parent node, with child nodes for each IP address. Similarly, an implementation that is allowed to connect using multiple interfaces will have a parent node of the tree for the decision between the network paths, with a branch for each interface.

The example aggregate connection attempt above can be drawn as a tree by grouping the addresses resolved on the same interface into branches:



The rest of this section will use a notation scheme to represent this tree. The parent (or trunk) node of the tree will be represented by a single integer, such as "1". Each child of that node will have an integer that identifies it, from 1 to the number of children. That child node will be uniquely identified by concatenating its integer to it's parents identifier with a dot in between, such as "1.1" and "1.2". Each node will be summarized by a tuple of three elements: endpoint, path (labeled here by interface), and protocol. The above example can now be written more succinctly as:

```

1 [www.example.com:80, Any, TCP]
  1.1 [www.example.com:80, Wi-Fi, TCP]
    1.1.1 [192.0.2.1:80, Wi-Fi, TCP]
  1.2 [www.example.com:80, LTE, TCP]
    1.2.1 [192.0.2.1:80, LTE, TCP]
    1.2.2 [2001:DB8::1.80, LTE, TCP]

```

When an implementation views this aggregate set of connection attempts as a single connection establishment, it only will use one of the leaf nodes to transfer data. Thus, when a single leaf node becomes ready to use, then the entire connection attempt is ready to use by the application. Another way to represent this is that every leaf node updates the state of its parent node when it becomes ready, until the trunk node of the tree is ready, which then notifies the application that the connection as a whole is ready to use.

A connection establishment tree may be degenerate, and only have a single leaf node, such as a connection attempt to an IP address over a single interface with a single protocol.

```

1 [192.0.2.1:80, Wi-Fi, TCP]

```

A parent node may also only have one child (or leaf) node, such as a when a hostname resolves to only a single IP address.



```
1 [www.example.com:80, Wi-Fi, TCP]
  1.1 [192.0.2.1:80, Wi-Fi, TCP]
```

#### 4.1.1. Branch Types

There are three types of branching from a parent node into one or more child nodes. Any parent node of the tree must only use one type of branching.

##### 4.1.1.1. Derived Endpoints

If a connection originally targets a single endpoint, there may be multiple endpoints of different types that can be derived from the original. This creates an ordered list of the derived endpoints according to application preference, system policy and expected performance.

DNS hostname-to-address resolution is the most common method of endpoint derivation. When trying to connect to a hostname endpoint on a traditional IP network, the implementation should send DNS queries for both A (IPv4) and AAAA (IPv6) records if both are supported on the local interface. The algorithm for ordering and racing these addresses should follow the recommendations in Happy Eyeballs [RFC8305].

```
1 [www.example.com:80, Wi-Fi, TCP]
  1.1 [2001:DB8::1.80, Wi-Fi, TCP]
  1.2 [192.0.2.1:80, Wi-Fi, TCP]
  1.3 [2001:DB8::2.80, Wi-Fi, TCP]
  1.4 [2001:DB8::3.80, Wi-Fi, TCP]
```

DNS-Based Service Discovery [RFC6763] can also provide an endpoint derivation step. When trying to connect to a named service, the client may discover one or more hostname and port pairs on the local network using multicast DNS [RFC6762]. These hostnames should each be treated as a branch that can be attempted independently from other hostnames. Each of these hostnames might resolve to one or more addresses, which would create multiple layers of branching.

```
1 [term-printer._ipp._tcp.meeting.ietf.org, Wi-Fi, TCP]
  1.1 [term-printer.meeting.ietf.org:631, Wi-Fi, TCP]
    1.1.1 [31.133.160.18.631, Wi-Fi, TCP]
```

Applications can influence which derived endpoints are allowed and preferred via Selection Properties set on the Preconnection. For example, setting a preference for useTemporaryLocalAddress would prefer the use of IPv6 over IPv4, and requiring useTemporaryLocalAddress would eliminate IPv4 options, since IPv4 does not support temporary addresses.

#### 4.1.1.2. Alternate Paths

If a client has multiple network paths available to it, e.g., a mobile client with interfaces for both Wi-Fi and Cellular connectivity, it can attempt a connection over any of the paths. This represents a branch point in the connection establishment. Similar to a derived endpoint, the paths should be ranked based on preference, system policy, and performance. Attempts should be started on one path (e.g., a specific interface), and then successively on other paths (or interfaces) after delays based on expected path round-trip-time or other available metrics.

```
1 [192.0.2.1:80, Any, TCP]
  1.1 [192.0.2.1:80, Wi-Fi, TCP]
  1.2 [192.0.2.1:80, LTE, TCP]
```

This same approach applies to any situation in which the client is aware of multiple links or views of the network. A single interface may be shared by multiple network paths, each with a coherent set of addresses, routes, DNS server, and more. A path may also represent a virtual interface service such as a Virtual Private Network (VPN).

The list of available paths should be constrained by any requirements the application sets, as well as by the system policy.

#### 4.1.1.3. Protocol Options

Differences in possible protocol compositions and options can also provide a branching point in connection establishment. This allows clients to be resilient to situations in which a certain protocol is not functioning on a server or network.

This approach is commonly used for connections with optional proxy server configurations. A single connection might have several options available: an HTTP-based proxy, a SOCKS-based proxy, or no proxy. These options should be ranked and attempted in succession.

- 1 [www.example.com:80, Any, HTTP/TCP]
  - 1.1 [192.0.2.8:80, Any, HTTP/HTTP Proxy/TCP]
  - 1.2 [192.0.2.7:10234, Any, HTTP/SOCKS/TCP]
  - 1.3 [www.example.com:80, Any, HTTP/TCP]
    - 1.3.1 [192.0.2.1:80, Any, HTTP/TCP]

This approach also allows a client to attempt different sets of application and transport protocols that, when available, could provide preferable features. For example, the protocol options could involve QUIC [I-D.ietf-quic-transport] over UDP on one branch, and HTTP/2 [RFC7540] over TLS over TCP on the other:

- 1 [www.example.com:443, Any, Any HTTP]
  - 1.1 [www.example.com:443, Any, QUIC/UDP]
    - 1.1.1 [192.0.2.1:443, Any, QUIC/UDP]
  - 1.2 [www.example.com:443, Any, HTTP2/TLS/TCP]
    - 1.2.1 [192.0.2.1:443, Any, HTTP2/TLS/TCP]

Another example is racing SCTP with TCP:

- 1 [www.example.com:80, Any, Any Stream]
  - 1.1 [www.example.com:80, Any, SCTP]
    - 1.1.1 [192.0.2.1:80, Any, SCTP]
  - 1.2 [www.example.com:80, Any, TCP]
    - 1.2.1 [192.0.2.1:80, Any, TCP]

Implementations that support racing protocols and protocol options should maintain a history of which protocols and protocol options successfully established, on a per-network and per-endpoint basis (see Section 9.2). This information can influence future racing decisions to prioritize or prune branches.

#### 4.1.2. Branching Order-of-Operations

Branch types must occur in a specific order relative to one another to avoid creating leaf nodes with invalid or incompatible settings. In the example above, it would be invalid to branch for derived endpoints (the DNS results for www.example.com) before branching between interface paths, since there are situations when the results will be different across networks due to private names or different supported IP versions. Implementations must be careful to branch in an order that results in usable leaf nodes whenever there are multiple branch types that could be used from a single node.

The order of operations for branching should be:

1. Alternate Paths

## 2. Protocol Options

## 3. Derived Endpoints

where a lower number indicates higher precedence and therefore higher placement in the tree. Branching between paths is the first in the list because results across multiple interfaces are likely not related to one another: endpoint resolution may return different results, especially when using locally resolved host and service names, and which protocols are supported and preferred may differ across interfaces. Thus, if multiple paths are attempted, the overall connection can be seen as a race between the available paths or interfaces.

Protocol options are next checked in order. Whether or not a set of protocol, or protocol-specific options, can successfully connect is generally not dependent on which specific IP address is used. Furthermore, the protocol stacks being attempted may influence or altogether change the endpoints being used. Adding a proxy to a connection's branch will change the endpoint to the proxy's IP address or hostname. Choosing an alternate protocol may also modify the ports that should be selected.

Branching for derived endpoints is the final step, and may have multiple layers of derivation or resolution, such as DNS service resolution and DNS hostname resolution.

For example, if the application has indicated both a preference for WiFi over LTE and for a feature only available in SCTP, branches will be first sorted accord to path selection, with WiFi at the top. Then, branches with SCTP will be sorted to the top within their subtree according to the properties influencing protocol selection. However, if the implementation has current cache information that SCTP is not available on the path over WiFi, there is no SCTP node in the WiFi subtree. Here, the path over WiFi will be tried first, and, if connection establishment succeeds, TCP will be used. So the Selection Property of preferring WiFi takes precedence over the Property that led to a preference for SCTP.

```
1. [www.example.com:80, Any, Any Stream]
1.1 [192.0.2.1:80, Wi-Fi, Any Stream]
1.1.1 [192.0.2.1:80, Wi-Fi, TCP]
1.2 [192.0.3.1:80, LTE, Any Stream]
1.2.1 [192.0.3.1:80, LTE, SCTP]
1.2.2 [192.0.3.1:80, LTE, TCP]
```

#### 4.1.3. Sorting Branches

Implementations should sort the branches of the tree of connection options in order of their preference rank, from most preferred to least preferred. Leaf nodes on branches with higher rankings represent connection attempts that will be raced first. Implementations should order the branches to reflect the preferences expressed by the application for its new connection, including Selection Properties, which are specified in [I-D.ietf-taps-interface].

In addition to the properties provided by the application, an implementation may include additional criteria such as cached performance estimates, see Section 9.2, or system policy, see Section 3.2, in the ranking. Two examples of how Selection and Connection Properties may be used to sort branches are provided below:

- \* "Interface Instance or Type": If the application specifies an interface type to be preferred or avoided, implementations should accordingly rank the paths. If the application specifies an interface type to be required or prohibited, an implementation is expected to not include the non-conforming paths.
- \* "Capacity Profile": An implementation can use the Capacity Profile to prefer paths that match an application's expected traffic pattern. This match will use cached performance estimates, see Section 9.2:
  - Scavenger: Prefer paths with the highest expected available capacity, but minimising impact on other traffic, based on the observed maximum throughput;
  - Low Latency/Interactive: Prefer paths with the lowest expected Round Trip Time, based on observed round trip time estimates;
  - Low Latency/Non-Interactive: Prefer paths with a low expected Round Trip Time, but can tolerate delay variation;
  - Constant-Rate Streaming: Prefer paths that are expected to satisfy the requested Stream Send or Stream Receive Bitrate, based on the observed maximum throughput;
  - Capacity-Seeking: Prefer adapting to paths to determine the highest available capacity, based on the observed maximum throughput.

Implementations process the Properties in the following order: Prohibit, Require, Prefer, Avoid. If Selection Properties contain any prohibited properties, the implementation should first purge branches containing nodes with these properties. For required properties, it should only keep branches that satisfy these requirements. Finally, it should order the branches according to the preferred properties, and finally use any avoided properties as a tiebreaker. When ordering branches, an implementation can give more weight to properties that the application has explicitly set, than to the properties that are default.

The available protocols and paths on a specific system and in a specific context can change; therefore, the result of sorting and the outcome of racing may vary, even when using the same Selection and Connection Properties. However, an implementation ought to provide a consistent outcome to applications, e.g., by preferring protocols and paths that are already used by existing Connections that specified similar Properties.

#### 4.2. Candidate Gathering

The step of gathering candidates involves identifying which paths, protocols, and endpoints may be used for a given Connection. This list is determined by the requirements, prohibitions, and preferences of the application as specified in the Selection Properties.

##### 4.2.1. Gathering Endpoint Candidates

Both Local and Remote Endpoint Candidates must be discovered during connection establishment. To support Interactive Connectivity Establishment (ICE) [RFC8445], or similar protocols that involve out-of-band indirect signalling to exchange candidates with the Remote Endpoint, it is important to query the set of candidate Local Endpoints, and provide the protocol stack with a set of candidate Remote Endpoints, before the Local Endpoint attempts to establish connections.

###### 4.2.1.1. Local Endpoint candidates

The set of possible Local Endpoints is gathered. In the simple case, this merely enumerates the local interfaces and protocols, and allocates ephemeral source ports. For example, a system that has WiFi and Ethernet and supports IPv4 and IPv6 might gather four candidate Local Endpoints (IPv4 on Ethernet, IPv6 on Ethernet, IPv4 on WiFi, and IPv6 on WiFi) that can form the source for a transient.

If NAT traversal is required, the process of gathering Local Endpoints becomes broadly equivalent to the ICE candidate gathering phase (see Section 5.1.1. of [RFC8445]). The endpoint determines its server reflexive Local Endpoints (i.e., the translated address of a Local Endpoint, on the other side of a NAT, e.g via a STUN sever [RFC5389]) and relayed Local Endpoints (e.g., via a TURN server [RFC5766] or other relay), for each interface and network protocol. These are added to the set of candidate Local Endpoints for this connection.

Gathering Local Endpoints is primarily a local operation, although it might involve exchanges with a STUN server to derive server reflexive Local Endpoints, or with a TURN server or other relay to derive relayed Local Endpoints. However, it does not involve communication with the Remote Endpoint.

#### 4.2.1.2. Remote Endpoint Candidates

The Remote Endpoint is typically a name that needs to be resolved into a set of possible addresses that can be used for communication. Resolving the Remote Endpoint is the process of recursively performing such name lookups, until fully resolved, to return the set of candidates for the Remote Endpoint of this connection.

How this resolution is done will depend on the type of the Remote Endpoint, and can also be specific to each Local Endpoint. A common case is when the Remote Endpoint is a DNS name, in which case it is resolved to give a set of IPv4 and IPv6 addresses representing that name. Some types of Remote Endpoint might require more complex resolution. Resolving the Remote Endpoint for a peer-to-peer connection might involve communication with a rendezvous server, which in turn contacts the peer to gain consent to communicate and retrieve its set of candidate Local Endpoints, which are returned and form the candidate remote addresses for contacting that peer.

Resolving the Remote Endpoint is not a local operation. It will involve a directory service, and can require communication with the Remote Endpoint to rendezvous and exchange peer addresses. This can expose some or all of the candidate Local Endpoints to the Remote Endpoint.

#### 4.3. Candidate Racing

The primary goal of the Candidate Racing process is to successfully negotiate a protocol stack to an endpoint over an interface to connect a single leaf node of the tree with as little delay and as few unnecessary connections attempts as possible. Optimizing these two factors improves the user experience, while minimizing network load.

This section covers the dynamic aspect of connection establishment. The tree described above is a useful conceptual and architectural model. However, an implementation is unable to know the full tree before it is formed and many of the possible branches ultimately might not be used.

There are three different approaches to racing the attempts for different nodes of the connection establishment tree:

1. Simultaneous
2. Staggered
3. Failover

Each approach is appropriate in different use-cases and branch types. However, to avoid consuming unnecessary network resources, implementations should not use simultaneous racing as a default approach.

The timing algorithms for racing should remain independent across branches of the tree. Any timers or racing logic is isolated to a given parent node, and is not ordered precisely with regards to other children of other nodes.

##### 4.3.1. Simultaneous

Simultaneous racing is when multiple alternate branches are started without waiting for any one branch to make progress before starting the next alternative. This means the attempts are effectively simultaneous. Simultaneous racing should be avoided by implementations, since it consumes extra network resources and establishes state that might not be used.



#### 4.3.2. Staggered

Staggered racing can be used whenever a single node of the tree has multiple child nodes. Based on the order determined when building the tree, the first child node will be initiated immediately, followed by the next child node after some delay. Once that second child node is initiated, the third child node (if present) will begin after another delay, and so on until all child nodes have been initiated, or one of the child nodes successfully completes its negotiation.

Staggered racing attempts can proceed in parallel. Implementations should not terminate an earlier child connection attempt upon starting a secondary child.

If a child node fails to establish connectivity (as in Section 4.4.1) before the delay time has expired for the next child, the next child should be started immediately.

Staggered racing between IP addresses for a generic Connection should follow the Happy Eyeballs algorithm described in [RFC8305]. [RFC8421] provides guidance for racing when performing Interactive Connectivity Establishment (ICE).

Generally, the delay before starting a given child node ought to be based on the length of time the previously started child node is expected to take before it succeeds or makes progress in connection establishment. Algorithms like Happy Eyeballs choose a delay based on how long the transport connection handshake is expected to take. When performing staggered races in multiple branch types (such as racing between network interfaces, and then racing between IP addresses), a longer delay may be chosen for some branch types. For example, when racing between network interfaces, the delay should also take into account the amount of time it takes to prepare the network interface (such as radio association) and name resolution over that interface, in addition to the delay that would be added for a single transport connection handshake.

Since the staggered delay can be chosen based on dynamic information, such as predicted round-trip time, implementations should define upper and lower bounds for delay times. These bounds are implementation-specific, and may differ based on which branch type is being used.

#### 4.3.3. Failover

If an implementation or application has a strong preference for one branch over another, the branching node may choose to wait until one child has failed before starting the next. Failure of a leaf node is determined by its protocol negotiation failing or timing out; failure of a parent branching node is determined by all of its children failing.

An example in which failover is recommended is a race between a protocol stack that uses a proxy and a protocol stack that bypasses the proxy. Failover is useful in case the proxy is down or misconfigured, but any more aggressive type of racing may end up unnecessarily avoiding a proxy that was preferred by policy.

#### 4.4. Completing Establishment

The process of connection establishment completes when one leaf node of the tree has successfully completed negotiation with the Remote Endpoint, or else all nodes of the tree have failed to connect. The first leaf node to complete its connection is then used by the application to send and receive data.

Successes and failures of a given attempt should be reported up to parent nodes (towards the trunk of the tree). For example, in the following case, if 1.1.1 fails to connect, it reports the failure to 1.1. Since 1.1 has no other child nodes, it also has failed and reports that failure to 1. Because 1.2 has not yet failed, 1 is not considered to have failed. Since 1.2 has not yet started, it is started and the process continues. Similarly, if 1.1.1 successfully connects, then it marks 1.1 as connected, which propagates to the trunk node 1. At this point, the connection as a whole is considered to be successfully connected and ready to process application data.

```
1 [www.example.com:80, Any, TCP]
  1.1 [www.example.com:80, Wi-Fi, TCP]
    1.1.1 [192.0.2.1:80, Wi-Fi, TCP]
    1.2 [www.example.com:80, LTE, TCP]
  ...
```

If a leaf node has successfully completed its connection, all other attempts should be made ineligible for use by the application for the original request. New connection attempts that involve transmitting data on the network ought not to be started after another leaf node has already successfully completed, because the connection as a whole has now been established. An implementation may choose to let certain handshakes and negotiations complete in order to gather metrics to influence future connections. Keeping additional connections is generally not recommended since those attempts were slower to connect and may exhibit less desirable properties.

#### 4.4.1. Determining Successful Establishment

Implementations may select the criteria by which a leaf node is considered to be successfully connected differently on a per-protocol basis. If the only protocol being used is a transport protocol with a clear handshake, like TCP, then the obvious choice is to declare that node "connected" when the last packet of the three-way handshake has been received. If the only protocol being used is an connectionless protocol, like UDP, the implementation may consider the node fully "connected" the moment it determines a route is present, before sending any packets on the network, see further Section 4.6.

For protocol stacks with multiple handshakes, the decision becomes more nuanced. If the protocol stack involves both TLS and TCP, an implementation could determine that a leaf node is connected after the TCP handshake is complete, or it can wait for the TLS handshake to complete as well. The benefit of declaring completion when the TCP handshake finishes, and thus stopping the race for other branches of the tree, is reduced burden on the network and Remote Endpoints from further connection attempts that are likely to be abandoned. On the other hand, by waiting until the TLS handshake is complete, an implementation avoids the scenario in which a TCP handshake completes quickly, but TLS negotiation is either very slow or fails altogether in particular network conditions or to a particular endpoint. To avoid the issue of TLS possibly failing, the implementation should not generate a Ready event for the Connection until TLS is established.

If all of the leaf nodes fail to connect during racing, i.e. none of the configurations that satisfy all requirements given in the Transport Properties actually work over the available paths, then the transport system should notify the application with an `InitiateError` event. An `InitiateError` event should also be generated in case the transport system finds no usable candidates to race.

#### 4.5. Establishing multiplexed connections

Multiplexing several Connections over a single underlying transport connection requires that the Connections to be multiplexed belong to the same Connection Group (as is indicated by the application using the Clone call). When the underlying transport connection supports multi-streaming, the Transport Services System can map each Connection in the Connection Group to a different stream. Thus, when the Connections that are offered to an application by the Transport Services API are multiplexed, the Transport Services implementation can establish a new Connection by simply beginning to use a new stream of an already established transport Connection and there is no need for a connection establishment procedure. This, then, also means that there may not be any "establishment" message (like a TCP SYN), but the application can simply start sending or receiving. Therefore, when the Initiate action of a Transport Services API is called without Messages being handed over, it cannot be guaranteed that the Remote Endpoint will have any way to know about this, and hence a passive endpoint's ConnectionReceived event might not be called until data is received. Instead, calling the ConnectionReceived event could be delayed until the first Message arrives.

#### 4.6. Handling connectionless protocols

While protocols that use an explicit handshake to validate a Connection to a peer can be used for racing multiple establishment attempts in parallel, connectionless protocols such as raw UDP do not offer a way to validate the presence of a peer or the usability of a Connection without application feedback. An implementation should consider such a protocol stack to be established as soon as the Transport Services system has selected a path on which to send data.

However, if a peer is not reachable over the network using the connectionless protocol, or data cannot be exchanged for any other reason, the application may want to attempt using another candidate Protocol Stack. The implementation should maintain the list of other candidate Protocol Stacks that were eligible to use.

#### 4.7. Implementing listeners

When an implementation is asked to Listen, it registers with the system to wait for incoming traffic to the Local Endpoint. If no Local Endpoint is specified, the implementation should use an ephemeral port.

If the Selection Properties do not require a single network interface or path, but allow the use of multiple paths, the Listener object should register for incoming traffic on all of the network interfaces or paths that conform to the Properties. The set of available paths can change over time, so the implementation should monitor network path changes, and change the registration of the Listener across all usable paths as appropriate. When using multiple paths, the Listener is generally expected to use the same port for listening on each.

If the Selection Properties allow multiple protocols to be used for listening, and the implementation supports it, the Listener object should support receiving inbound connections for each eligible protocol on each eligible path.

#### 4.7.1. Implementing listeners for Connected Protocols

Connected protocols such as TCP and TLS-over-TCP have a strong mapping between the Local and Remote Endpoints (four-tuple) and their protocol connection state. These map into Connection objects. Whenever a new inbound handshake is being started, the Listener should generate a new Connection object and pass it to the application.

#### 4.7.2. Implementing listeners for Connectionless Protocols

Connectionless protocols such as UDP and UDP-lite generally do not provide the same mechanisms that connected protocols do to offer Connection objects. Implementations should wait for incoming packets for connectionless protocols on a listening port and should perform four-tuple matching of packets to either existing Connection objects or the creation of new Connection objects. On platforms with facilities to create a "virtual connection" for connectionless protocols implementations should use these mechanisms to minimise the handling of datagrams intended for already created Connection objects.

#### 4.7.3. Implementing listeners for Multiplexed Protocols

Protocols that provide multiplexing of streams into a single four-tuple can listen both for entirely new connections (a new HTTP/2 stream on a new TCP connection, for example) and for new sub-connections (a new HTTP/2 stream on an existing connection). If the abstraction of Connection presented to the application is mapped to the multiplexed stream, then the Listener should deliver new Connection objects in the same way for either case. The implementation should allow the application to introspect the Connection Group marked on the Connections to determine the grouping of the multiplexing.

## 5. Implementing Sending and Receiving Data

The most basic mapping for sending a Message is an abstraction of datagrams, in which the transport protocol naturally deals in discrete packets. Each Message here corresponds to a single datagram. Generally, these will be short enough that sending and receiving will always use a complete Message.

For protocols that expose byte-streams, the only delineation provided by the protocol is the end of the stream in a given direction. Each Message in this case corresponds to the entire stream of bytes in a direction. These Messages may be quite long, in which case they can be sent in multiple parts.

Protocols that provide the framing (such as length-value protocols, or protocols that use delimiters) may support Message sizes that do not fit within a single datagram. Each Message for framing protocols corresponds to a single frame, which may be sent either as a complete Message in the underlying protocol, or in multiple parts.

### 5.1. Sending Messages

The effect of the application sending a Message is determined by the top-level protocol in the established Protocol Stack. That is, if the top-level protocol provides an abstraction of framed messages over a connection, the receiving application will be able to obtain multiple Messages on that connection, even if the framing protocol is built on a byte-stream protocol like TCP.

#### 5.1.1. Message Properties

- \* **Lifetime:** this should be implemented by removing the Message from the queue of pending Messages after the Lifetime has expired. A queue of pending Messages within the transport system implementation that have yet to be handed to the Protocol Stack can always support this property, but once a Message has been sent into the send buffer of a protocol, only certain protocols may support removing a message. For example, an implementation cannot remove bytes from a TCP send buffer, while it can remove data from a SCTP send buffer using the partial reliability extension [RFC8303]. When there is no standing queue of Messages within the system, and the Protocol Stack does not support the removal of a Message from the stack's send buffer, this property may be ignored.
- \* **Priority:** this represents the ability to prioritize a Message over other Messages. This can be implemented by the system re-ordering Messages that have yet to be handed to the Protocol Stack, or by

giving relative priority hints to protocols that support priorities per Message. For example, an implementation of HTTP/2 could choose to send Messages of different Priority on streams of different priority.

- \* **Ordered:** when this is false, this disables the requirement of in-order-delivery for protocols that support configurable ordering. When the protocol stack does not support configurable ordering, this property may be ignored.
- \* **Safely Replayable:** when this is true, this means that the Message can be used by a transport mechanism that might transfer it multiple times -- e.g., as a result of racing multiple transports or as part of TCP Fast Open. Also, protocols that do not protect against duplicated messages, such as UDP (when used directly, without a protocol layered atop), can only be used with Messages that are Safely Replayable. When a transport system is permitted to replay messages, replay protection could be provided by the application.
- \* **Final:** when this is true, this means that the sender will not send any further messages. The Connection need not be closed (in case the Protocol Stack supports half-close operation, like TCP). Any messages sent after a Final message will result in a SendError.
- \* **Corruption Protection Length:** when this is set to any value other than Full Coverage, it sets the minimum protection in protocols that allow limiting the checksum length (e.g. UDP-Lite). If the protocol stack does not support checksum length limitation, this property may be ignored.
- \* **Reliable Data Transfer (Message):** When true, the property specifies that the Message must be reliably transmitted. When false, and if unreliable transmission is supported by the underlying protocol, then the Message should be unreliably transmitted. If the underlying protocol does not support unreliable transmission, the Message should be reliably transmitted.
- \* **Message Capacity Profile Override:** When true, this expresses a wish to override the Generic Connection Property Capacity Profile for this Message. Depending on the value, this can, for example, be implemented by changing the DSCP value of the associated packet (note that the guidelines in Section 6 of [RFC7657] apply; e.g., the DSCP value should not be changed for different packets within a reliable transport protocol session or DCCP connection).

- \* No Fragmentation: When set, this property limits the message size to the Maximum Message Size Before Fragmentation or Segmentation (see Section 10.1.7 of [I-D.ietf-taps-interface]). Messages larger than this size generate an error. Setting this avoids transport-layer segmentation or network-layer fragmentation. When used with transports running over IP version 4 the Don't Fragment bit will be set to avoid on-path IP fragmentation ([RFC8304]).

#### 5.1.2. Send Completion

The application should be notified whenever a Message or partial Message has been consumed by the Protocol Stack, or has failed to send. The time at which a Message is considered to have been consumed by the Protocol Stack may vary depending on the protocol. For example, for a basic datagram protocol like UDP, this may correspond to the time when the packet is sent into the interface driver. For a protocol that buffers data in queues, like TCP, this may correspond to when the data has entered the send buffer. The time at which a message failed to send is when Transport Services implementation (including the Protocol Stack) has not successfully sent the entire Message content or partial Message content on any open candidate connection; this can depend on protocol-specific timeouts.

#### 5.1.3. Batching Sends

Since sending a Message may involve a context switch between the application and the Transport Services system, sending patterns that involve multiple small Messages can incur high overhead if each needs to be enqueued separately. To avoid this, the application can indicate a batch of Send actions through the API. When this is used, the implementation can defer the processing of Messages until the batch is complete.

#### 5.2. Receiving Messages

Similar to sending, Receiving a Message is determined by the top-level protocol in the established Protocol Stack. The main difference with Receiving is that the size and boundaries of the Message are not known beforehand. The application can communicate in its Receive action the parameters for the Message, which can help the Transport Services implementation know how much data to deliver and when. For example, if the application only wants to receive a complete Message, the implementation should wait until an entire Message (datagram, stream, or frame) is read before delivering any Message content to the application. This requires the implementation to understand where messages end, either via a supplied deframer or because the top-level protocol in the established Protocol Stack



preserves message boundaries. If the top-level protocol only supports a byte-stream and no framers were supported, the application can control the flow of received data by specifying the minimum number of bytes of Message content it wants to receive at one time.

If a Connection finishes before a requested Receive action can be satisfied, the Transport Services API should deliver any partial Message content outstanding, or if none is available, an indication that there will be no more received Messages.

### 5.3. Handling of data for fast-open protocols

Several protocols allow sending higher-level protocol or application data during their protocol establishment, such as TCP Fast Open [RFC7413] and TLS 1.3 [RFC8446]. This approach is referred to as sending Zero-RTT (0-RTT) data. This is a desirable feature, but poses challenges to an implementation that uses racing during connection establishment.

The amount of data that can be sent as 0-RTT data varies by protocol and can be queried by the application using the Maximum Message Size Concurrent with Connection Establishment Connection Property. An implementation can set this property according to the protocols that it will race based on the given Selection Properties when the application requests to establish a connection.

If the application has 0-RTT data to send in any protocol handshakes, it needs to provide this data before the handshakes have begun. When racing, this means that the data should be provided before the process of connection establishment has begun. If the application wants to send 0-RTT data, it must indicate this to the implementation by setting the Safely Replayable send parameter to true when sending the data. In general, 0-RTT data may be replayed (for example, if a TCP SYN contains data, and the SYN is retransmitted, the data will be retransmitted as well but may be considered as a new connection instead of a retransmission). Also, when racing connections, different leaf nodes have the opportunity to send the same data independently. If data is truly safely replayable, this should be permissible.

Once the application has provided its 0-RTT data, a Transport Services implementation should keep a copy of this data and provide it to each new leaf node that is started and for which a 0-RTT protocol is being used.

It is also possible that protocol stacks within a particular leaf node use 0-RTT handshakes without any safely replayable application data. For example, TCP Fast Open could use a Client Hello from TLS as its 0-RTT data, shortening the cumulative handshake time.

0-RTT handshakes often rely on previous state, such as TCP Fast Open cookies, previously established TLS tickets, or out-of-band distributed pre-shared keys (PSKs). Implementations should be aware of security concerns around using these tokens across multiple addresses or paths when racing. In the case of TLS, any given ticket or PSK should only be used on one leaf node, since servers will likely reject duplicate tickets in order to prevent replays (see section-8.1 [RFC8446]). If implementations have multiple tickets available from a previous connection, each leaf node attempt can use a different ticket. In effect, each leaf node will send the same early application data, yet encoded (encrypted) differently on the wire.

## 6. Implementing Message Framers

Message Framers are functions that define simple transformations between application Message data and raw transport protocol data. A Framers can encapsulate or encode outbound Messages, and decapsulate or decode inbound data into Messages.

While many protocols can be represented as Message Framers, for the purposes of the Transport Services API, these are ways for applications or application frameworks to define their own Message parsing to be included within a Connection's Protocol Stack. As an example, TLS is exposed as a protocol natively supported by the Transport Services API, even though it could also serve the purpose of framing data over TCP.

Most Message Framers fall into one of two categories:

- \* Header-prefixed record formats, such as a basic Type-Length-Value (TLV) structure
- \* Delimiter-separated formats, such as HTTP/1.1.

Common Message Framers can be provided by a Transport Services implementation, but an implementation ought to allow custom Message Framers to be defined by the application or some other piece of software. This section describes one possible API for defining Message Framers as an example.

## 6.1. Defining Message Framers

A Message Framer is primarily defined by the code that handles events for a framer implementation, specifically how it handles inbound and outbound data parsing. The function that implements custom framing logic will be referred to as the "framer implementation", which may be provided by a Transport Services implementation or the application itself. The Message Framer refers to the object or function within the main Connection implementation that delivers events to the custom framer implementation whenever data is ready to be parsed or framed.

The Transport Services implementation needs to ensure that all of the events and actions taken on a Message Framer are synchronized to ensure consistent behavior. For example, some of the actions defined below (such as `PrependFramer` and `StartPassthrough`) modify how data flows in a protocol stack, and require synchronization with sending and parsing data in the Message Framer.

When a Connection establishment attempt begins, an event can be delivered to notify the framer implementation that a new Connection is being created. Similarly, a stop event can be delivered when a Connection is being torn down. The framer implementation can use the Connection object to look up specific properties of the Connection or the network being used that may influence how to frame Messages.

```
MessageFramer -> Start(Connection)
MessageFramer -> Stop(Connection)
```

When a Message Framer generates a Start event, the framer implementation has the opportunity to start writing some data prior to the Connection delivering its Ready event. This allows the implementation to communicate control data to the Remote Endpoint that can be used to parse Messages.

```
MessageFramer.MakeConnectionReady(Connection)
```

Similarly, when a Message Framer generates a Stop event, the framer implementation has the opportunity to write some final data or clear up its local state before the Closed event is delivered to the Application. The framer implementation can indicate that it has finished with this.

```
MessageFramer.MakeConnectionClosed(Connection)
```

At any time if the implementation encounters a fatal error, it can also cause the Connection to fail and provide an error.

```
MessageFramer.FailConnection(Connection, Error)
```

Should the framer implementation deem the candidate selected during racing unsuitable, it can signal this to the Transport Services API by failing the Connection prior to marking it as ready. If there are no other candidates available, the Connection will fail. Otherwise, the Connection will select a different candidate and the Message Framer will generate a new Start event.

Before an implementation marks a Message Framer as ready, it can also dynamically add a protocol or framer above it in the stack. This allows protocols that need to add TLS conditionally, like STARTTLS [RFC3207], to modify the Protocol Stack based on a handshake result.

```
otherFramer := NewMessageFramer()
MessageFramer.PrependFramer(Connection, otherFramer)
```

A Message Framer might also choose to go into a passthrough mode once an initial exchange or handshake has been completed, such as the STARTTLS case mentioned above. This can also be useful for proxy protocols like SOCKS [RFC1928] or HTTP CONNECT [RFC7230]. In such cases, a Message Framer implementation can intercept sending and receiving of messages at first, but then indicate that no more processing is needed.

```
MessageFramer.StartPassthrough()
```

## 6.2. Sender-side Message Framing

Message Framers generate an event whenever a Connection sends a new Message.

```
MessageFramer -> NewSentMessage<Connection, MessageData, MessageContext, IsEndOfMessage>
```

Upon receiving this event, a framer implementation is responsible for performing any necessary transformations and sending the resulting data back to the Message Framer, which will in turn send it to the next protocol. Implementations SHOULD ensure that there is a way to pass the original data through without copying to improve performance.

```
MessageFramer.Send(Connection, Data)
```

To provide an example, a simple protocol that adds a length as a header would receive the NewSentMessage event, create a data representation of the length of the Message data, and then send a block of data that is the concatenation of the length header and the original Message data.

### 6.3. Receiver-side Message Framing

In order to parse a received flow of data into Messages, the Message Framer notifies the framer implementation whenever new data is available to parse.

MessageFramer -> HandleReceivedData<Connection>

Upon receiving this event, the framer implementation can inspect the inbound data. The data is parsed from a particular cursor representing the unprocessed data. The application requests a specific amount of data it needs to have available in order to parse. If the data is not available, the parse fails.

MessageFramer.Parse(Connection, MinimumIncompleteLength, MaximumLength) -> (Data, MessageContext, IsEndOfMessage)

The framer implementation can directly advance the receive cursor once it has parsed data to effectively discard data (for example, discard a header once the content has been parsed).

To deliver a Message to the application, the framer implementation can either directly deliver data that it has allocated, or deliver a range of data directly from the underlying transport and simultaneously advance the receive cursor.

MessageFramer.AdvanceReceiveCursor(Connection, Length)

MessageFramer.DeliverAndAdvanceReceiveCursor(Connection, MessageContext, Length, IsEndOfMessage)

MessageFramer.Deliver(Connection, MessageContext, Data, IsEndOfMessage)

Note that MessageFramer.DeliverAndAdvanceReceiveCursor allows the framer implementation to earmark bytes as part of a Message even before they are received by the transport. This allows the delivery of very large Messages without requiring the implementation to directly inspect all of the bytes.

To provide an example, a simple protocol that parses a length as a header value would receive the HandleReceivedData event, and call Parse with a minimum and maximum set to the length of the header field. Once the parse succeeded, it would call AdvanceReceiveCursor with the length of the header field, and then call DeliverAndAdvanceReceiveCursor with the length of the body that was parsed from the header, marking the new Message as complete.

## 7. Implementing Connection Management

Once a Connection is established, the Transport Services API allows applications to interact with the Connection by modifying or inspecting Connection Properties. A Connection can also generate events in the form of Soft Errors.

The set of Connection Properties that are supported for setting and getting on a Connection are described in [I-D.ietf-taps-interface]. For any properties that are generic, and thus could apply to all protocols being used by a Connection, the Transport Services implementation should store the properties in storage common to all protocols, and notify all protocol instances in the Protocol Stack whenever the properties have been modified by the application. For protocol-specific properties, such as the User Timeout that applies to TCP, the Transport Services implementation only needs to update the relevant protocol instance.

If an error is encountered in setting a property (for example, if the application tries to set a TCP-specific property on a Connection that is not using TCP), the action should fail gracefully. The application may be informed of the error, but the Connection itself should not be terminated.

The Transport Services API should allow protocol instances in the Protocol Stack to pass up arbitrary generic or protocol-specific errors that can be delivered to the application as Soft Errors. These allow the application to be informed of ICMP errors, and other similar events.

### 7.1. Pooled Connection

For applications that do not need in-order delivery of Messages, the Transport Services implementation may distribute Messages of a single Connection across several underlying transport connections or multiple streams of multi-streaming connections between endpoints, as long as all of these satisfy the Selection Properties. The Transport Services implementation will then hide this connection management and only expose a single Connection object, which we here call a "Pooled Connection". This is in contrast to Connection Groups, which explicitly expose combined treatment of Connections, giving the application control over multiplexing, for example.

Pooled Connections can be useful when the application using the Transport Services system implements a protocol such as HTTP, which employs request/response pairs and does not require in-order delivery of responses. This enables implementations of Transport Services systems to realize transparent connection coalescing, connection migration, and to perform per-message endpoint and path selection by choosing among multiple underlying connections.

## 7.2. Handling Path Changes

When a path change occurs, e.g., when the IP address of an interface changes or a new interface becomes available, the Transport Services implementation is responsible for notifying the Protocol Instance of the change. The path change may interrupt connectivity on a path for an active connection or provide an opportunity for a transport that supports multipath or migration to adapt to the new paths. Note that, in the model of the Transport Services API, migration is considered a part of multipath connectivity; it is just a limiting policy on multipath usage. If the multipath Selection Property is set to Disabled, migration is disallowed.

For protocols that do not support multipath or migration, the Protocol Instances should be informed of the path change, but should not be forcibly disconnected if the previously used path becomes unavailable. There are many common user scenarios that can lead to a path becoming temporarily unavailable, and then recovering before the transport protocol reaches a timeout error. These are particularly common using mobile devices. Examples include: an Ethernet cable becoming unplugged and then plugged back in; a device losing a Wi-Fi signal while a user is in an elevator, and reattaching when the user leaves the elevator; and a user losing the radio signal while riding a train through a tunnel. If the device is able to rejoin a network with the same IP address, a stateful transport connection can generally resume. Thus, while it is useful for a Protocol Instance to be aware of a temporary loss of connectivity, the Transport Services implementation should not aggressively close connections in these scenarios.

If the Protocol Stack includes a transport protocol that supports multipath connectivity, the Transport Services implementation should also inform the Protocol Instance of potentially new paths that become permissible based on the multipath Selection Property and the multipath-policy Connection Property choices made by the application. A protocol can then establish new subflows over new paths while an active path is still available or, if migration is supported, also after a break has been detected, and should attempt to tear down subflows over paths that are no longer used. The Connection Property multipath-policy of the Transport Services API allows an application

to indicate when and how different paths should be used. However, detailed handling of these policies is still implementation-specific. For example, if the multipath Selection Property is set to active, the decision about when to create a new path or to announce a new path or set of paths to the Remote Endpoint, e.g., in the form of additional IP addresses, is implementation-specific. If the Protocol Stack includes a transport protocol that does not support multipath, but does support migrating between paths, the update to the set of available paths can trigger the connection to be migrated.

In case of Pooled Connections Section 7.1, the Transport Services implementation may add connections over new paths to the pool if permissible based on the multipath policy and Selection Properties. In case a previously used path becomes unavailable, the transport system may disconnect all connections that require this path, but should not disconnect the pooled connection object exposed to the application. The strategy to do so is implementation-specific, but should be consistent with the behavior of multipath transports.

## 8. Implementing Connection Termination

With TCP, when an application closes a connection, this means that it has no more data to send (but expects all data that has been handed over to be reliably delivered). However, with TCP only, "close" does not mean that the application will stop receiving data. This is related to TCP's ability to support half-closed connections.

SCTP is an example of a protocol that does not support such half-closed connections. Hence, with SCTP, the meaning of "close" is stricter: an application has no more data to send (but expects all data that has been handed over to be reliably delivered), and will also not receive any more data.

Implementing a protocol independent transport system means that the exposed semantics must be the strictest subset of the semantics of all supported protocols. Hence, as is common with all reliable transport protocols, after a Close action, the application can expect to have its reliability requirements honored regarding the data provided to the Transport Services API, but it cannot expect to be able to read any more data after calling Close.

Abort differs from Close only in that no guarantees are given regarding any data that the application sent to the Transport Services API before calling Abort.

As explained in Section 4.5, when a new stream is multiplexed on an already existing connection of a Transport Protocol Instance, there is no need for a connection establishment procedure. Because the



Connections that are offered by a Transport Services implementation can be implemented as streams that are multiplexed on a transport protocol's connection, it can therefore not be guaranteed an Initiate action from one endpoint provokes a ConnectionReceived event at its peer.

For Close (provoking a Finished event) and Abort (provoking a ConnectionError event), the same logic applies: while it is desirable to be informed when a peer closes or aborts a Connection, whether this is possible depends on the underlying protocol, and no guarantees can be given. With SCTP, the transport system can use the stream reset procedure to cause a Finish event upon a Close action from the peer [NEAT-flow-mapping].

## 9. Cached State

Beyond a single Connection's lifetime, it is useful for an implementation to keep state and history. This cached state can help improve future Connection establishment due to re-using results and credentials, and favoring paths and protocols that performed well in the past.

Cached state may be associated with different endpoints for the same Connection, depending on the protocol generating the cached content. For example, session tickets for TLS are associated with specific endpoints, and thus should be cached based on a Connection's hostname endpoint (if applicable). However, performance characteristics of a path are more likely tied to the IP address and subnet being used.

### 9.1. Protocol state caches

Some protocols will have long-term state to be cached in association with endpoints. This state often has some time after which it is expired, so the implementation should allow each protocol to specify an expiration for cached content.

Examples of cached protocol state include:

- \* The DNS protocol can cache resolution answers (A and AAAA queries, for example), associated with a Time To Live (TTL) to be used for future hostname resolutions without requiring asking the DNS resolver again.
- \* TLS caches session state and tickets based on a hostname, which can be used for resuming sessions with a server.
- \* TCP can cache cookies for use in TCP Fast Open.

Cached protocol state is primarily used during Connection establishment for a single Protocol Stack, but may be used to influence an implementation's preference between several candidate Protocol Stacks. For example, if two IP address endpoints are otherwise equally preferred, an implementation may choose to attempt a connection to an address for which it has a TCP Fast Open cookie.

Applications can use the Transport Services API to request that a Connection Group maintain a separate cache for protocol state. Connections in the group will not use cached state from connections outside the group, and connections outside the group will not use state cached from connections inside the group. This may be necessary, for example, if application-layer identifiers rotate and clients wish to avoid linkability via trackable TLS tickets or TFO cookies.

## 9.2. Performance caches

In addition to protocol state, Protocol Instances should provide data into a performance-oriented cache to help guide future protocol and path selection. Some performance information can be gathered generically across several protocols to allow predictive comparisons between protocols on given paths:

- \* Observed Round Trip Time
- \* Connection Establishment latency
- \* Connection Establishment success rate

These items can be cached on a per-address and per-subnet granularity, and averaged between different values. The information should be cached on a per-network basis, since it is expected that different network attachments will have different performance characteristics. Besides Protocol Instances, other system entities may also provide data into performance-oriented caches. This could for instance be signal strength information reported by radio modems like Wi-Fi and mobile broadband or information about the battery-level of the device. Furthermore, the system may cache the observed maximum throughput on a path as an estimate of the available bandwidth.

An implementation should use this information, when possible, to influence preference between candidate paths, endpoints, and protocol options. Eligible options that historically had significantly better performance than others should be selected first when gathering candidates (see Section 4.2) to ensure better performance for the application.

The reasonable lifetime for cached performance values will vary depending on the nature of the value. Certain information, like the connection establishment success rate to a Remote Endpoint using a given protocol stack, can be stored for a long period of time (hours or longer), since it is expected that the capabilities of the Remote Endpoint are not changing very quickly. On the other hand, the Round Trip Time observed by TCP over a particular network path may vary over a relatively short time interval. For such values, the implementation should remove them from the cache more quickly, or treat older values with less confidence/weight.

[I-D.ietf-tcpm-2140bis] provides guidance about sharing of TCP Control Block information between connections on initialization.

#### 10. Specific Transport Protocol Considerations

Each protocol that is supported by a Transport Services implementation should have a well-defined API mapping. API mappings for a protocol are important for Connections in which a given protocol is the "top" of the Protocol Stack. For example, the mapping of the Send function for TCP applies to Connections in which the application directly sends over TCP.

Each protocol has a notion of Connectedness. Possible values for Connectedness are:

- \* Connectionless. Connectionless protocols do not establish explicit state between endpoints, and do not perform a handshake during Connection establishment.
- \* Connected. Connected protocols establish state between endpoints, and perform a handshake during Connection establishment. The handshake may be 0-RTT to send data or resume a session, but bidirectional traffic is required to confirm connectedness.
- \* Multiplexing Connected. Multiplexing Connected protocols share properties with Connected protocols, but also explicitly support opening multiple application-level flows. This means that they can support cloning new Connection objects without a new explicit handshake.

Protocols also define a notion of Data Unit. Possible values for Data Unit are:

- \* Byte-stream. Byte-stream protocols do not define any Message boundaries of their own apart from the end of a stream in each direction.

- \* Datagram. Datagram protocols define Message boundaries at the same level of transmission, such that only complete (not partial) Messages are supported.
- \* Message. Message protocols support Message boundaries that can be sent and received either as complete or partial Messages. Maximum Message lengths can be defined, and Messages can be partially reliable.

Below, terms in capitals with a dot (e.g., "CONNECT.SCTP") refer to the primitives with the same name in section 4 of [RFC8303]. For further implementation details, the description of these primitives in [RFC8303] points to section 3 of [RFC8303] and section 3 of [RFC8304], which refers back to the relevant specifications for each protocol. This back-tracking method applies to all elements of [RFC8923] (see appendix D of [I-D.ietf-taps-interface]): they are listed in appendix A of [RFC8923] with an implementation hint in the same style, pointing back to section 4 of [RFC8303].

This document defines the API mappings for protocols defined in [RFC8923]. Other protocol mappings can be provided as separate documents, following the mapping template Appendix A.

#### 10.1. TCP

Connectedness: Connected

Data Unit: Byte-stream

API mappings for TCP are as follows:

Connection Object: TCP connections between two hosts map directly to Connection objects.

Initiate: CONNECT.TCP. Calling Initiate on a TCP Connection causes it to reserve a local port, and send a SYN to the Remote Endpoint.

InitiateWithSend: CONNECT.TCP with parameter user message. Early safely replayable data is sent on a TCP Connection in the SYN, as TCP Fast Open data.

Ready: A TCP Connection is ready once the three-way handshake is complete.

InitiateError: Failure of CONNECT.TCP. TCP can throw various errors during connection setup. Specifically, it is important to handle a RST being sent by the peer during the handshake.

**ConnectionError:** Once established, TCP throws errors whenever the connection is disconnected, such as due to receiving a RST from the peer.

**Listen:** LISTEN.TCP. Calling Listen for TCP binds a local port and prepares it to receive inbound SYN packets from peers.

**ConnectionReceived:** TCP Listeners will deliver new connections once they have replied to an inbound SYN with a SYN-ACK.

**Clone:** Calling Clone on a TCP Connection creates a new Connection with equivalent parameters. These Connections, and Connections generated via later calls to Clone on an Established Connection, form a Connection Group. To realize entanglement for these Connections, with the exception of Connection Priority, changing a Connection Property on one of them must affect the Connection Properties of the others too. No guarantees of honoring the Connection Property Connection Priority are given, and thus it is safe for an implementation of a transport system to ignore this property. When it is reasonable to assume that Connections traverse the same path (e.g., when they share the same encapsulation), support for it can also experimentally be implemented using a congestion control coupling mechanism (see for example [TCP-COUPLING] or [RFC3124]).

**Send:** SEND.TCP. TCP does not on its own preserve Message boundaries. Calling Send on a TCP connection lays out the bytes on the TCP send stream without any other delineation. Any Message marked as Final will cause TCP to send a FIN once the Message has been completely written, by calling CLOSE.TCP immediately upon successful termination of SEND.TCP. Note that transmitting a Message marked as Final should not cause the Closed event to be delivered to the application, as it will still be possible to receive data until the peer closes or aborts the TCP connection.

**Receive:** With RECEIVE.TCP, TCP delivers a stream of bytes without any Message delineation. All data delivered in the Received or ReceivedPartial event will be part of a single stream-wide Message that is marked Final (unless a Message Framer is used). EndOfMessage will be delivered when the TCP Connection has received a FIN (CLOSE-EVENT.TCP) from the peer. Note that reception of a FIN should not cause the Closed event to be delivered to the application, as it will still be possible for the application to send data.

**Close:** Calling Close on a TCP Connection indicates that the

Connection should be gracefully closed (CLOSE.TCP) by sending a FIN to the peer. It will then still be possible to receive data until the peer closes or aborts the TCP connection. The Closed event will be issued upon reception of a FIN.

Abort: Calling Abort on a TCP Connection indicates that the Connection should be immediately closed by sending a RST to the peer (ABORT.TCP).

## 10.2. MPTCP

Connectedness: Connected

Data Unit: Byte-stream

the Transport Services API mappings for MPTCP are identical to TCP. MPTCP adds support for multipath properties, such as "Multipath Transport" and "Policy for using Multipath Transports".

## 10.3. UDP

Connectedness: Connectionless

Data Unit: Datagram

API mappings for UDP are as follows:

Connection Object: UDP connections represent a pair of specific IP addresses and ports on two hosts.

Initiate: CONNECT.UDP. Calling Initiate on a UDP Connection causes it to reserve a local port, but does not generate any traffic.

InitiateWithSend: Early data on a UDP Connection does not have any special meaning. The data is sent whenever the Connection is Ready.

Ready: A UDP Connection is ready once the system has reserved a local port and has a path to send to the Remote Endpoint.

InitiateError: UDP Connections can only generate errors on initiation due to port conflicts on the local system.

ConnectionError: Once in use, UDP throws "soft errors" (ERROR.UDP(-Lite)) upon receiving ICMP notifications indicating failures in the network.

Listen: LISTEN.UDP. Calling Listen for UDP binds a local port and

prepares it to receive inbound UDP datagrams from peers.

**ConnectionReceived:** UDP Listeners will deliver new connections once they have received traffic from a new Remote Endpoint.

**Clone:** Calling Clone on a UDP Connection creates a new Connection with equivalent parameters. The two Connections are otherwise independent.

**Send:** SEND.UDP(-Lite). Calling Send on a UDP connection sends the data as the payload of a complete UDP datagram. Marking Messages as Final does not change anything in the datagram's contents. Upon sending a UDP datagram, some relevant fields and flags in the IP header can be controlled: DSCP (SET\_DSCP.UDP(-Lite)), DF in IPv4 (SET\_DF.UDP(-Lite)) and ECN flag (SET\_ECN.UDP(-Lite)).

**Receive:** RECEIVE.UDP(-Lite). UDP only delivers complete Messages to Received, each of which represents a single datagram received in a UDP packet. Upon receiving a UDP datagram, the ECN flag from the IP header can be obtained (GET\_ECN.UDP(-Lite)).

**Close:** Calling Close on a UDP Connection (ABORT.UDP(-Lite)) releases the local port reservation.

**Abort:** Calling Abort on a UDP Connection (ABORT.UDP(-Lite)) is identical to calling Close.

#### 10.4. UDP-Lite

**Connectedness:** Connectionless

**Data Unit:** Datagram

The Transport Services API mappings for UDP-Lite are identical to UDP. Properties that require checksum coverage are not supported by UDP-Lite, such as "Corruption Protection Length", "Full Checksum Coverage on Sending", "Required Minimum Corruption Protection Coverage for Receiving", and "Full Checksum Coverage on Receiving".

#### 10.5. UDP Multicast Receive

**Connectedness:** Connectionless

**Data Unit:** Datagram

API mappings for Receiving Multicast UDP are as follows:

**Connection Object:** Established UDP Multicast Receive connections

represent a pair of specific IP addresses and ports. The "unidirectional receive" transport property is required, and the Local Endpoint must be configured with a group IP address and a port.

**Initiate:** Calling `Initiate` on a UDP Multicast Receive Connection causes an immediate `InitiateError`. This is an unsupported operation.

**InitiateWithSend:** Calling `InitiateWithSend` on a UDP Multicast Receive Connection causes an immediate `InitiateError`. This is an unsupported operation.

**Ready:** A UDP Multicast Receive Connection is ready once the system has received traffic for the appropriate group and port.

**InitiateError:** UDP Multicast Receive Connections generate an `InitiateError` if `Initiate` is called.

**ConnectionError:** Once in use, UDP throws "soft errors" (`ERROR.UDP(-Lite)`) upon receiving ICMP notifications indicating failures in the network.

**Listen:** `LISTEN.UDP`. Calling `Listen` for UDP Multicast Receive binds a local port, prepares it to receive inbound UDP datagrams from peers, and issues a multicast host join. If a Remote Endpoint with an address is supplied, the join is Source-specific Multicast, and the path selection is based on the route to the Remote Endpoint. If a Remote Endpoint is not supplied, the join is Any-source Multicast, and the path selection is based on the outbound route to the group supplied in the Local Endpoint.

There are cases where it is required to open multiple connections for the same address(es). For example, one Connection might be opened for a multicast group to for a multicast control bus, and another application later opens a separate Connection to the same group to send signals to and/or receive signals from the common bus. In such cases, the Transport Services system needs to explicitly enable re-use of the same set of addresses (equivalent to setting `SO_REUSEADDR` in the socket API).

**ConnectionReceived:** UDP Multicast Receive Listeners will deliver new connections once they have received traffic from a new Remote Endpoint.

**Clone:** Calling `Clone` on a UDP Multicast Receive Connection creates a new Connection with equivalent parameters. The two Connections are otherwise independent.



Send: `SEND.UDP(-Lite)`. Calling Send on a UDP Multicast Receive connection causes an immediate `SendError`. This is an unsupported operation.

Receive: `RECEIVE.UDP(-Lite)`. The Receive operation in a UDP Multicast Receive connection only delivers complete Messages to Received, each of which represents a single datagram received in a UDP packet. Upon receiving a UDP datagram, the ECN flag from the IP header can be obtained (`GET_ECN.UDP(-Lite)`).

Close: Calling Close on a UDP Multicast Receive Connection (`ABORT.UDP(-Lite)`) releases the local port reservation and leaves the group.

Abort: Calling Abort on a UDP Multicast Receive Connection (`ABORT.UDP(-Lite)`) is identical to calling Close.

#### 10.6. SCTP

Connectedness: Connected

Data Unit: Message

API mappings for SCTP are as follows:

Connection Object: Connection objects can be mapped to an SCTP association or a stream in an SCTP association. Mapping Connection objects to SCTP streams is called "stream mapping" and has additional requirements as follows. The following explanation assumes a client-server communication model.

Stream mapping requires an association to already be in place between the client and the server, and it requires the server to understand that a new incoming stream should be represented as a new Connection Object by the Transport Services system. A new SCTP stream is created by sending an SCTP message with a new stream id. Thus, to implement stream mapping, the Transport Services API MUST provide a newly created Connection Object to the application upon the reception of such a message. The necessary semantics to implement a Transport Services system Close and Abort primitives are provided by the stream reconfiguration (reset) procedure described in [RFC6525]. This also allows to re-use a stream id after resetting ("closing") the stream. To implement this functionality, SCTP stream reconfiguration [RFC6525] MUST be supported by both the client and the server side.

To avoid head-of-line blocking, stream mapping SHOULD only be implemented when both sides support message interleaving [RFC8260]. This allows a sender to schedule transmissions between multiple streams without risking that transmission of a large message on one stream might block transmissions on other streams for a long time.

To avoid conflicts between stream ids, the following procedure is recommended: the first Connection, for which the SCTP association has been created, MUST always use stream id zero. All additional Connections are assigned to unused stream ids in growing order. To avoid a conflict when both endpoints map new Connections simultaneously, the peer which initiated association MUST use even stream ids whereas the remote side MUST map its Connections to odd stream ids. Both sides maintain a status map of the assigned stream ids. Generally, new streams SHOULD consume the lowest available (even or odd, depending on the side) stream id; this rule is relevant when lower ids become available because Connection objects associated with the streams are closed.

SCTP stream mapping as described here has been implemented in a research prototype; a description of this implementation is given in [NEAT-flow-mapping].

**Initiate:** If this is the only Connection object that is assigned to the SCTP Association or stream mapping is not used, CONNECT.SCTP is called. Else, unless the Selection Property `activeReadBeforeSend` is Preferred or Required, a new stream is used: if there are enough streams available, Initiate is a local operation that assigns a new stream id to the Connection object. The number of streams is negotiated as a parameter of the prior CONNECT.SCTP call, and it represents a trade-off between local resource usage and the number of Connection objects that can be mapped without requiring a reconfiguration signal. When running out of streams, ADD\_STREAM.SCTP must be called.

**InitiateWithSend:** If this is the only Connection object that is assigned to the SCTP association or stream mapping is not used, CONNECT.SCTP is called with the "user message" parameter. Else, a new stream is used (see Initiate for how to handle running out of streams), and this just sends the first message on a new stream.

**Ready:** Initiate or InitiateWithSend returns without an error, i.e. SCTP's four-way handshake has completed. If an association with the peer already exists, stream mapping is used and enough streams are available, a Connection Object instantly becomes Ready after calling Initiate or InitiateWithSend.

**InitiateError:** Failure of CONNECT.SCTP.

ConnectionError: TIMEOUT.SCTP or ABORT-EVENT.SCTP.

Listen: LISTEN.SCTP. If an association with the peer already exists and stream mapping is used, Listen just expects to receive a new message with a new stream id (chosen in accordance with the stream id assignment procedure described above).

ConnectionReceived: LISTEN.SCTP returns without an error (a result of successful CONNECT.SCTP from the peer), or, in case of stream mapping, the first message has arrived on a new stream (in this case, Receive is also invoked).

Clone: Calling Clone on an SCTP association creates a new Connection object and assigns it a new stream id in accordance with the stream id assignment procedure described above. If there are not enough streams available, ADD\_STREAM.SCTP must be called.

Priority (Connection): When this value is changed, or a Message with Message Property Priority is sent, and there are multiple Connection objects assigned to the same SCTP association, CONFIGURE\_STREAM\_SCHEDULER.SCTP is called to adjust the priorities of streams in the SCTP association.

Send: SEND.SCTP. Message Properties such as Lifetime and Ordered map to parameters of this primitive.

Receive: RECEIVE.SCTP. The "partial flag" of RECEIVE.SCTP invokes a ReceivedPartial event.

Close: If this is the only Connection object that is assigned to the SCTP association, CLOSE.SCTP is called, and the Closed event will be delivered to the application upon the ensuing CLOSE-EVENT.SCTP. Else, the Connection object is one out of several Connection objects that are assigned to the same SCTP association, and RESET\_STREAM.SCTP must be called, which informs the peer that the stream will no longer be used for mapping and can be used by future Initiate, InitiateWithSend or Listen calls. At the peer, the event RESET\_STREAM-EVENT.SCTP will fire, which the peer must answer by issuing RESET\_STREAM.SCTP too. The resulting local RESET\_STREAM-EVENT.SCTP informs the Transport Services system that the stream id can now be re-used by the next Initiate, InitiateWithSend or Listen calls, and invokes a Closed event towards the application.

Abort: If this is the only Connection object that is assigned to the SCTP association, ABORT.SCTP is called. Else, the Connection object is one out of several Connection objects that are assigned to the same SCTP association, and shutdown proceeds as described under Close.

## 11. IANA Considerations

RFC-EDITOR: Please remove this section before publication.

This document has no actions for IANA.

## 12. Security Considerations

[I-D.ietf-taps-arch] outlines general security consideration and requirements for any system that implements the Transport Services architecture. [I-D.ietf-taps-interface] provides further discussion on security and privacy implications of the Transport Services API. This document provides additional guidance on implementation specifics for the Transport Services API and as such the security considerations in both of these documents apply. The next two subsections discuss further considerations that are specific to mechanisms specified in this document.

### 12.1. Considerations for Candidate Gathering

Implementations should avoid downgrade attacks that allow network interference to cause the implementation to select less secure, or entirely insecure, combinations of paths and protocols.

### 12.2. Considerations for Candidate Racing

See Section 5.3 for security considerations around racing with 0-RTT data.

An attacker that knows a particular device is racing several options during connection establishment may be able to block packets for the first connection attempt, thus inducing the device to fall back to a secondary attempt. This is a problem if the secondary attempts have worse security properties that enable further attacks. Implementations should ensure that all options have equivalent security properties to avoid incentivizing attacks.

Since results from the network can determine how a connection attempt tree is built, such as when DNS returns a list of resolved endpoints, it is possible for the network to cause an implementation to consume significant on-device resources. Implementations should limit the maximum amount of state allowed for any given node, including the number of child nodes, especially when the state is based on results from the network.

### 13. Acknowledgements

This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 644334 (NEAT) and No. 815178 (5GENESIS).

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

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

This work has been supported by the Research Council of Norway under its "Toppforsk" programme through the "OCARINA" project.

Thanks to Colin Perkins, Tom Jones, Karl-Johan Grinnemo, Gorrry Fairhurst, for their contributions to the design of this specification. Thanks also to Stuart Cheshire, Josh Graessley, David Schinazi, and Eric Kinnear for their implementation and design efforts, including Happy Eyeballs, that heavily influenced this work.

### 14. References

#### 14.1. Normative References

[I-D.ietf-taps-arch]

Pauly, T., Trammell, B., Brunstrom, A., Fairhurst, G., and C. Perkins, "An Architecture for Transport Services", Work in Progress, Internet-Draft, draft-ietf-taps-arch-12, 3 January 2022, <<https://datatracker.ietf.org/doc/html/draft-ietf-taps-arch-12>>.

[I-D.ietf-taps-interface]

Trammell, B., Welzl, M., Enghardt, T., Fairhurst, G., Kuehlewind, M., Perkins, C., Tiesel, P. S., Wood, C. A., Pauly, T., and K. Rose, "An Abstract Application Layer Interface to Transport Services", Work in Progress, Internet-Draft, draft-ietf-taps-interface-14, 3 January 2022, <<https://datatracker.ietf.org/doc/html/draft-ietf-taps-interface-14>>.

[RFC7413] Cheng, Y., Chu, J., Radhakrishnan, S., and A. Jain, "TCP Fast Open", RFC 7413, DOI 10.17487/RFC7413, December 2014, <<https://www.rfc-editor.org/rfc/rfc7413>>.

- [RFC7540] Belshé, M., Peon, R., and M. Thomson, Ed., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, DOI 10.17487/RFC7540, May 2015, <<https://www.rfc-editor.org/rfc/rfc7540>>.
- [RFC8303] Welzl, M., Tuexen, M., and N. Khademi, "On the Usage of Transport Features Provided by IETF Transport Protocols", RFC 8303, DOI 10.17487/RFC8303, February 2018, <<https://www.rfc-editor.org/rfc/rfc8303>>.
- [RFC8304] Fairhurst, G. and T. Jones, "Transport Features of the User Datagram Protocol (UDP) and Lightweight UDP (UDP-Lite)", RFC 8304, DOI 10.17487/RFC8304, February 2018, <<https://www.rfc-editor.org/rfc/rfc8304>>.
- [RFC8305] Schinazi, D. and T. Pauly, "Happy Eyeballs Version 2: Better Connectivity Using Concurrency", RFC 8305, DOI 10.17487/RFC8305, December 2017, <<https://www.rfc-editor.org/rfc/rfc8305>>.
- [RFC8421] Martinsen, P., Reddy, T., and P. Patil, "Guidelines for Multihomed and IPv4/IPv6 Dual-Stack Interactive Connectivity Establishment (ICE)", BCP 217, RFC 8421, DOI 10.17487/RFC8421, July 2018, <<https://www.rfc-editor.org/rfc/rfc8421>>.
- [RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <<https://www.rfc-editor.org/rfc/rfc8446>>.
- [RFC8923] Welzl, M. and S. Gjessing, "A Minimal Set of Transport Services for End Systems", RFC 8923, DOI 10.17487/RFC8923, October 2020, <<https://www.rfc-editor.org/rfc/rfc8923>>.

#### 14.2. Informative References

- [I-D.ietf-quic-transport] Iyengar, J. and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport", Work in Progress, Internet-Draft, draft-ietf-quic-transport-34, 14 January 2021, <<https://datatracker.ietf.org/doc/html/draft-ietf-quic-transport-34>>.

- [I-D.ietf-tcpm-2140bis]  
Touch, J., Welzl, M., and S. Islam, "TCP Control Block Interdependence", Work in Progress, Internet-Draft, draft-ietf-tcpm-2140bis-11, 12 April 2021, <<https://datatracker.ietf.org/doc/html/draft-ietf-tcpm-2140bis-11>>.
- [NEAT-flow-mapping]  
"Transparent Flow Mapping for NEAT", IFIP NETWORKING 2017 Workshop on Future of Internet Transport (FIT 2017) , 2017.
- [RFC1928] Leech, M., Ganis, M., Lee, Y., Kuris, R., Koblas, D., and L. Jones, "SOCKS Protocol Version 5", RFC 1928, DOI 10.17487/RFC1928, March 1996, <<https://www.rfc-editor.org/rfc/rfc1928>>.
- [RFC3124] Balakrishnan, H. and S. Seshan, "The Congestion Manager", RFC 3124, DOI 10.17487/RFC3124, June 2001, <<https://www.rfc-editor.org/rfc/rfc3124>>.
- [RFC3207] Hoffman, P., "SMTP Service Extension for Secure SMTP over Transport Layer Security", RFC 3207, DOI 10.17487/RFC3207, February 2002, <<https://www.rfc-editor.org/rfc/rfc3207>>.
- [RFC5389] Rosenberg, J., Mahy, R., Matthews, P., and D. Wing, "Session Traversal Utilities for NAT (STUN)", RFC 5389, DOI 10.17487/RFC5389, October 2008, <<https://www.rfc-editor.org/rfc/rfc5389>>.
- [RFC5766] Mahy, R., Matthews, P., and J. Rosenberg, "Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN)", RFC 5766, DOI 10.17487/RFC5766, April 2010, <<https://www.rfc-editor.org/rfc/rfc5766>>.
- [RFC6525] Stewart, R., Tuexen, M., and P. Lei, "Stream Control Transmission Protocol (SCTP) Stream Reconfiguration", RFC 6525, DOI 10.17487/RFC6525, February 2012, <<https://www.rfc-editor.org/rfc/rfc6525>>.
- [RFC6762] Cheshire, S. and M. Krochmal, "Multicast DNS", RFC 6762, DOI 10.17487/RFC6762, February 2013, <<https://www.rfc-editor.org/rfc/rfc6762>>.
- [RFC6763] Cheshire, S. and M. Krochmal, "DNS-Based Service Discovery", RFC 6763, DOI 10.17487/RFC6763, February 2013, <<https://www.rfc-editor.org/rfc/rfc6763>>.

- [RFC7230] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, DOI 10.17487/RFC7230, June 2014, <<https://www.rfc-editor.org/rfc/rfc7230>>.
- [RFC7657] Black, D., Ed. and P. Jones, "Differentiated Services (Diffserv) and Real-Time Communication", RFC 7657, DOI 10.17487/RFC7657, November 2015, <<https://www.rfc-editor.org/rfc/rfc7657>>.
- [RFC8085] Eggert, L., Fairhurst, G., and G. Shepherd, "UDP Usage Guidelines", BCP 145, RFC 8085, DOI 10.17487/RFC8085, March 2017, <<https://www.rfc-editor.org/rfc/rfc8085>>.
- [RFC8260] Stewart, R., Tuexen, M., Loreto, S., and R. Seggellmann, "Stream Schedulers and User Message Interleaving for the Stream Control Transmission Protocol", RFC 8260, DOI 10.17487/RFC8260, November 2017, <<https://www.rfc-editor.org/rfc/rfc8260>>.
- [RFC8445] Keranen, A., Holmberg, C., and J. Rosenberg, "Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal", RFC 8445, DOI 10.17487/RFC8445, July 2018, <<https://www.rfc-editor.org/rfc/rfc8445>>.
- [TCP-COUPLING]  
"ctrlTCP: Reducing Latency through Coupled, Heterogeneous Multi-Flow TCP Congestion Control", IEEE INFOCOM Global Internet Symposium (GI) workshop (GI 2018) , n.d..

#### Appendix A. API Mapping Template

Any protocol mapping for the Transport Services API should follow a common template.

Connectedness: (Connectionless/Connected/Multiplexing Connected)

Data Unit: (Byte-stream/Datagram/Message)

Connection Object:

Initiate:

InitiateWithSend:

Ready:



InitiateError:  
ConnectionError:  
Listen:  
ConnectionReceived:  
Clone:  
Send:  
Receive:  
Close:  
Abort:

## Appendix B. Additional Properties

This appendix discusses implementation considerations for 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.

### B.1. Properties Affecting Sorting of Branches

In addition to the Protocol and Path Selection Properties discussed in Section 4.1.3, the following properties under discussion can influence branch sorting:

- \* **Bounds on Send or Receive Rate:** If the application indicates a bound on the expected Send or Receive bitrate, an implementation may prefer a path that can likely provide the desired bandwidth, based on cached maximum throughput, see Section 9.2. The application may know the Send or Receive Bitrate from metadata in adaptive HTTP streaming, such as MPEG-DASH.
- \* **Cost Preferences:** If the application indicates a preference to avoid expensive paths, and some paths are associated with a monetary cost, an implementation should decrease the ranking of such paths. If the application indicates that it prohibits using expensive paths, paths that are associated with a cost should be purged from the decision tree.

## Appendix C. Reasons for errors

The Transport Services API [I-D.ietf-taps-interface] allows for the several generic error types to specify a more detailed reason as to why an error occurred. This appendix lists some of the possible reasons.

- \* **InvalidConfiguration:** The transport properties and endpoints provided by the application are either contradictory or incomplete. Examples include the lack of a Remote Endpoint on an active open or using a multicast group address while not requesting a unidirectional receive.
- \* **NoCandidates:** The configuration is valid, but none of the available transport protocols can satisfy the transport properties provided by the application.
- \* **ResolutionFailed:** The remote or local specifier provided by the application can not be resolved.
- \* **EstablishmentFailed:** The Transport Services system was unable to establish a transport-layer connection to the Remote Endpoint specified by the application.
- \* **PolicyProhibited:** The system policy prevents the transport system from performing the action requested by the application.
- \* **NotCloneable:** The protocol stack is not capable of being cloned.
- \* **MessageTooLarge:** The message size is too big for the transport system to handle.
- \* **ProtocolFailed:** The underlying protocol stack failed.
- \* **InvalidMessageProperties:** The message properties are either contradictory to the transport properties or they can not be satisfied by the transport system.
- \* **DeframingFailed:** The data that was received by the underlying protocol stack could not be deframed.
- \* **ConnectionAborted:** The connection was aborted by the peer.
- \* **Timeout:** Delivery of a message was not possible after a timeout.

## Appendix D. Existing Implementations

This appendix gives an overview of existing implementations, at the time of writing, of transport systems that are (to some degree) in line with this document.

### \* Apple's Network.framework:

- Network.framework is a transport-level API built for C, Objective-C, and Swift. It is a connect-by-name API that supports transport security protocols. It provides userspace implementations of TCP, UDP, TLS, DTLS, proxy protocols, and allows extension via custom framers.
- Documentation: <https://developer.apple.com/documentation/network> (<https://developer.apple.com/documentation/network>)

### \* NEAT and NEATPy:

- NEAT is the output of the European H2020 research project "NEAT"; it is a user-space library for protocol-independent communication on top of TCP, UDP and SCTP, with many more features such as a policy manager.
- Code: <https://github.com/NEAT-project/neat> (<https://github.com/NEAT-project/neat>)
- NEAT project: <https://www.neat-project.org> (<https://www.neat-project.org>)
- NEATPy is a Python shim over NEAT which updates the NEAT API to be in line with version 6 of the Transport Services API draft.
- Code: <https://github.com/theagilepadawan/NEATPy> (<https://github.com/theagilepadawan/NEATPy>)

### \* PyTAPS:

- A TAPS implementation based on Python asyncio, offering protocol-independent communication to applications on top of TCP, UDP and TLS, with support for multicast.
- Code: <https://github.com/fg-inet/python-asyncio-taps> (<https://github.com/fg-inet/python-asyncio-taps>)

## Authors' Addresses

Anna Brunstrom (editor)  
Karlstad University  
Universitetsgatan 2  
651 88 Karlstad  
Sweden  
Email: anna.brunstrom@kau.se

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

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

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

Michael Welzl  
University of Oslo  
PO Box 1080 Blindern  
0316 Oslo  
Norway  
Email: michawe@ifi.uio.no

TAPS Working Group  
Internet-Draft  
Intended status: Informational  
Expires: January 3, 2019

B. Trammell, Ed.  
ETH Zurich  
M. Welzl, Ed.  
University of Oslo  
T. Enhardt  
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

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 January 3, 2019.

#### Copyright Notice

Copyright (c) 2018 IETF Trust and the persons identified as the document authors. All rights reserved.

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

#### Table of Contents

1. Introduction . . . . .	4
2. Terminology and Notation . . . . .	5
3. Interface Design Principles . . . . .	6
4. API Summary . . . . .	7
5. Pre-Establishment Phase . . . . .	7
5.1. Specifying Endpoints . . . . .	8
5.2. Specifying Transport Properties . . . . .	9
5.3. Specifying Security Parameters and Callbacks . . . . .	10
5.3.1. Pre-Connection Parameters . . . . .	11
5.3.2. Connection Establishment Callbacks . . . . .	12
6. Establishing Connections . . . . .	12
6.1. Active Open: Initiate . . . . .	12
6.2. Passive Open: Listen . . . . .	13
6.3. Peer-to-Peer Establishment: Rendezvous . . . . .	14
6.4. Connection Groups . . . . .	16
7. Sending Data . . . . .	16
7.1. Basic Sending . . . . .	17
7.2. Send Events . . . . .	17
7.2.1. Sent . . . . .	18
7.2.2. Expired . . . . .	18
7.2.3. SendError . . . . .	18
7.3. Message Context Parameters . . . . .	18
7.3.1. Lifetime . . . . .	19
7.3.2. Niceness . . . . .	20
7.3.3. Ordered . . . . .	20
7.3.4. Idempotent . . . . .	20
7.3.5. Final . . . . .	20
7.3.6. Corruption Protection Length . . . . .	21
7.3.7. Transmission Profile . . . . .	21

7.4.	Partial Sends . . . . .	22
7.5.	Batching Sends . . . . .	22
7.6.	Sender-side Framing . . . . .	23
8.	Receiving Data . . . . .	23
8.1.	Enqueuing Receives . . . . .	23
8.2.	Receive Events . . . . .	24
8.2.1.	Received . . . . .	24
8.2.2.	ReceivedPartial . . . . .	24
8.2.3.	ReceiveError . . . . .	25
8.3.	Message Receive Context . . . . .	25
8.3.1.	ECN . . . . .	26
8.3.2.	Early Data . . . . .	26
8.3.3.	Receiving Final Messages . . . . .	26
8.4.	Receiver-side De-framing over Stream Protocols . . . . .	26
9.	Setting and Querying Connection Properties . . . . .	27
10.	Connection Termination . . . . .	28
11.	Ordering of Operations and Events . . . . .	29
12.	Transport Properties . . . . .	30
12.1.	Transport Property Types . . . . .	30
12.1.1.	Boolean . . . . .	30
12.1.2.	Enumeration . . . . .	31
12.1.3.	Integer . . . . .	31
12.1.4.	Preference . . . . .	31
12.2.	Transport Property Classification . . . . .	31
12.2.1.	Selection Properties . . . . .	32
12.2.2.	Protocol Properties . . . . .	33
12.2.3.	Control Properties . . . . .	33
12.2.4.	Intents . . . . .	33
12.3.	Mandatory Transport Properties . . . . .	34
12.3.1.	Final . . . . .	34
12.3.2.	Reliable Data Transfer (Connection) . . . . .	34
12.3.3.	Configure per-Message reliability . . . . .	34
12.3.4.	Reliable Data Transfer (Message) . . . . .	35
12.3.5.	Preservation of data ordering . . . . .	35
12.3.6.	Ordered . . . . .	35
12.3.7.	Direction of communication . . . . .	36
12.3.8.	Use 0-RTT session establishment with an idempotent Message . . . . .	36
12.3.9.	Idempotent . . . . .	36
12.3.10.	Multistream Connections in Group . . . . .	37
12.3.11.	Notification of excessive retransmissions . . . . .	37
12.3.12.	Retransmission threshold before excessive retransmission notification . . . . .	37
12.3.13.	Notification of ICMP soft error message arrival . . . . .	38
12.3.14.	Control checksum coverage on sending or receiving . . . . .	38
12.3.15.	Corruption Protection Length . . . . .	38
12.3.16.	Required minimum coverage of the checksum for receiving . . . . .	39

12.3.17. Interface Instance or Type . . . . .	39
12.3.18. Provisioning Domain Instance or Type . . . . .	40
12.3.19. Capacity Profile . . . . .	41
12.3.20. Congestion control . . . . .	41
12.3.21. Niceness . . . . .	42
12.3.22. Timeout for aborting Connection . . . . .	42
12.3.23. Connection group transmission scheduler . . . . .	43
12.3.24. Maximum message size concurrent with Connection establishment . . . . .	43
12.3.25. Maximum Message size before fragmentation or segmentation . . . . .	43
12.3.26. Maximum Message size on send . . . . .	43
12.3.27. Maximum Message size on receive . . . . .	44
12.3.28. Lifetime . . . . .	44
12.4. Optional Transport Properties . . . . .	44
12.5. Experimental Transport Properties . . . . .	44
13. IANA Considerations . . . . .	44
14. Security Considerations . . . . .	45
15. Acknowledgements . . . . .	45
16. References . . . . .	45
16.1. Normative References . . . . .	45
16.2. Informative References . . . . .	46
Appendix A. Additional Properties . . . . .	47
A.1. Experimental Transport Properties . . . . .	47
A.1.1. Suggest a timeout to the Remote Endpoint . . . . .	48
A.1.2. Abort timeout to suggest to the Remote Endpoint . . . . .	48
A.1.3. Request not to delay acknowledgment of Message . . . . .	48
A.1.4. Traffic Category . . . . .	49
A.1.5. Size to be Sent or Received . . . . .	49
A.1.6. Duration . . . . .	49
A.1.7. Send or Receive Bit-rate . . . . .	50
A.1.8. Cost Preferences . . . . .	50
A.1.9. Immediate . . . . .	51
Appendix B. Sample API definition in Go . . . . .	51
Authors' Addresses . . . . .	51

## 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:

- o An Action creates an Object:

Object := Action()

- o An Action is performed on an Object:

Object.Action()

- o An Object sends an Event:

Object -> Event<>

- o 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 principles, 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;

- o 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
- o 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 `RemoteEndpoint`. 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:

- o 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)
```

- o 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)
```

- o 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)
```

- o 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 `Initiate()`; 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  $(M-N) \times C / M$ , 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 enqueueing 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 infinite 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. Enqueueing 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 indicate the smallest partial Message data size in bytes that should be delivered in response to this Receive. By default, this value is infinite, which means that only complete Messages should be delivered. 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 discontinuous 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 to the termination of the Connection are signaled using `ConnectionError` instead (see Section 10).

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

#### 8.3. Message Receive Context

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

#### 8.3.1. ECN

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

#### 8.3.2. Early Data

In some cases it may be valuable to know whether data was read as part of early data 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:

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

- o 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.
- o 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 Protocol Properties may be read-only, on a protocol- and property-specific basis.
- o 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:

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

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

## 12.1.1.2. 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.

## 12.1.1.3. Integer

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

## 12.1.1.4. 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 is 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 implementation 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 identifier. 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 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.1.1.9. 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](mailto:ietf@trammell.ch)

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

Email: [michawe@ifi.uio.no](mailto:michawe@ifi.uio.no)

Theresa Enghardt  
TU Berlin  
Marchstrasse 23  
10587 Berlin  
Germany

Email: [theresa@inet.tu-berlin.de](mailto:theresa@inet.tu-berlin.de)

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

Email: [gorry@erg.abdn.ac.uk](mailto: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](mailto:mirja.kuehlewind@tik.ee.ethz.ch)

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

Email: [csp@csp Perkins.org](mailto:csp@csp Perkins.org)

Philipp S. Tiesel  
TU Berlin  
Marchstrasse 23  
10587 Berlin  
Germany

Email: [philipp@inet.tu-berlin.de](mailto:philipp@inet.tu-berlin.de)



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

Email: cawood@apple.com

TAPS Working Group  
Internet-Draft  
Intended status: Standards Track  
Expires: 8 September 2022

B. Trammell, Ed.  
Google Switzerland GmbH  
M. Welzl, Ed.  
University of Oslo  
T. Enghardt  
Netflix  
G. Fairhurst  
University of Aberdeen  
M. Kuehlewind  
Ericsson  
C. Perkins  
University of Glasgow  
P. Tiesel  
SAP SE  
T. Pauly  
Apple Inc.  
7 March 2022

An Abstract Application Layer Interface to Transport Services  
draft-ietf-taps-interface-15

Abstract

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

Status of This Memo

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

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

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

This Internet-Draft will expire on 8 September 2022.

#### Copyright Notice

Copyright (c) 2022 IETF Trust and the persons identified as the document authors. All rights reserved.

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

#### Table of Contents

1. Introduction . . . . .	4
1.1. Terminology and Notation . . . . .	5
1.2. Specification of Requirements . . . . .	7
2. Overview of the API Design . . . . .	7
3. API Summary . . . . .	8
3.1. Usage Examples . . . . .	9
3.1.1. Server Example . . . . .	9
3.1.2. Client Example . . . . .	10
3.1.3. Peer Example . . . . .	12
4. Transport Properties . . . . .	13
4.1. Transport Property Names . . . . .	14
4.2. Transport Property Types . . . . .	15
5. Scope of the API Definition . . . . .	15
6. Pre-Establishment Phase . . . . .	16
6.1. Specifying Endpoints . . . . .	17
6.1.1. Using Multicast Endpoints . . . . .	19
6.1.2. Constraining Interfaces for Endpoints . . . . .	19
6.1.3. Endpoint Aliases . . . . .	20
6.1.4. Endpoint Examples . . . . .	20
6.1.5. Multicast Examples . . . . .	21
6.2. Specifying Transport Properties . . . . .	23
6.2.1. Reliable Data Transfer (Connection) . . . . .	26
6.2.2. Preservation of Message Boundaries . . . . .	27
6.2.3. Configure Per-Message Reliability . . . . .	27
6.2.4. Preservation of Data Ordering . . . . .	27

6.2.5.	Use 0-RTT Session Establishment with a Safely Replayable Message . . . . .	27
6.2.6.	Multistream Connections in Group . . . . .	28
6.2.7.	Full Checksum Coverage on Sending . . . . .	28
6.2.8.	Full Checksum Coverage on Receiving . . . . .	28
6.2.9.	Congestion control . . . . .	29
6.2.10.	Keep alive . . . . .	29
6.2.11.	Interface Instance or Type . . . . .	29
6.2.12.	Provisioning Domain Instance or Type . . . . .	30
6.2.13.	Use Temporary Local Address . . . . .	31
6.2.14.	Multipath Transport . . . . .	32
6.2.15.	Advertisement of Alternative Addresses . . . . .	33
6.2.16.	Direction of communication . . . . .	33
6.2.17.	Notification of ICMP soft error message arrival . . .	34
6.2.18.	Initiating side is not the first to write . . . . .	34
6.3.	Specifying Security Parameters and Callbacks . . . . .	35
6.3.1.	Specifying Security Parameters on a Pre-Connection .	35
6.3.2.	Connection Establishment Callbacks . . . . .	37
7.	Establishing Connections . . . . .	37
7.1.	Active Open: Initiate . . . . .	38
7.2.	Passive Open: Listen . . . . .	39
7.3.	Peer-to-Peer Establishment: Rendezvous . . . . .	40
7.4.	Connection Groups . . . . .	42
7.5.	Adding and Removing Endpoints on a Connection . . . .	44
8.	Managing Connections . . . . .	44
8.1.	Generic Connection Properties . . . . .	46
8.1.1.	Required Minimum Corruption Protection Coverage for Receiving . . . . .	46
8.1.2.	Connection Priority . . . . .	47
8.1.3.	Timeout for Aborting Connection . . . . .	47
8.1.4.	Timeout for keep alive packets . . . . .	47
8.1.5.	Connection Group Transmission Scheduler . . . . .	48
8.1.6.	Capacity Profile . . . . .	48
8.1.7.	Policy for using Multipath Transports . . . . .	50
8.1.8.	Bounds on Send or Receive Rate . . . . .	51
8.1.9.	Group Connection Limit . . . . .	51
8.1.10.	Isolate Session . . . . .	51
8.1.11.	Read-only Connection Properties . . . . .	52
8.2.	TCP-specific Properties: User Timeout Option (UTO) . .	53
8.2.1.	Advertised User Timeout . . . . .	53
8.2.2.	User Timeout Enabled . . . . .	53
8.2.3.	Timeout Changeable . . . . .	54
8.3.	Connection Lifecycle Events . . . . .	54
8.3.1.	Soft Errors . . . . .	54
8.3.2.	Path change . . . . .	54
9.	Data Transfer . . . . .	54
9.1.	Messages and Framers . . . . .	55
9.1.1.	Message Contexts . . . . .	55

9.1.2.	Message Framers . . . . .	55
9.1.3.	Message Properties . . . . .	58
9.2.	Sending Data . . . . .	64
9.2.1.	Basic Sending . . . . .	64
9.2.2.	Send Events . . . . .	65
9.2.3.	Partial Sends . . . . .	66
9.2.4.	Batching Sends . . . . .	66
9.2.5.	Send on Active Open: InitiateWithSend . . . . .	67
9.2.6.	Priority and the Transport Services API . . . . .	67
9.3.	Receiving Data . . . . .	68
9.3.1.	Enqueueing Receives . . . . .	68
9.3.2.	Receive Events . . . . .	69
9.3.3.	Receive Message Properties . . . . .	71
10.	Connection Termination . . . . .	73
11.	Connection State and Ordering of Operations and Events . . . . .	74
12.	IANA Considerations . . . . .	76
13.	Privacy and Security Considerations . . . . .	76
14.	Acknowledgements . . . . .	78
15.	References . . . . .	78
15.1.	Normative References . . . . .	78
15.2.	Informative References . . . . .	79
Appendix A.	Implementation Mapping . . . . .	83
A.1.	Types . . . . .	83
A.2.	Events and Errors . . . . .	84
A.3.	Time Duration . . . . .	84
Appendix B.	Convenience Functions . . . . .	84
B.1.	Adding Preference Properties . . . . .	84
B.2.	Transport Property Profiles . . . . .	84
B.2.1.	reliable-inorder-stream . . . . .	84
B.2.2.	reliable-message . . . . .	85
B.2.3.	unreliable-datagram . . . . .	85
Appendix C.	Relationship to the Minimal Set of Transport Services for End Systems . . . . .	86
Authors' Addresses	. . . . .	89

## 1. Introduction

This document specifies an abstract application programming interface (API) that specifies the interface component of the high-level Transport Services architecture defined in [I-D.ietf-taps-arch]. A Transport Services system supports asynchronous, atomic transmission of messages over transport protocols and network paths dynamically selected at runtime, in environments where an endpoint selects from multiple interfaces and potential transport protocols.

Applications that adopt this API will benefit from a wide set of transport features that can evolve over time. This protocol-independent API ensures that the system providing the API can

optimize its behavior based on the application requirements and network conditions, without requiring changes to the applications. This flexibility enables faster deployment of new features and protocols, and can support applications by offering racing and fallback mechanisms, which otherwise need to be separately implemented in each application.

The Transport Services system derives specific path and protocol selection properties and supported transport features from the analysis provided in [RFC8095], [RFC8923], and [RFC8922]. The Transport Services API enables an implementation to dynamically choose a transport protocol rather than statically binding applications to a protocol at compile time. The Transport Services API also provides applications with a way to override transport selection and instantiate a specific stack, e.g., to support servers wishing to listen to a specific protocol. However, forcing a choice to use a specific transport stack is discouraged for general use, because it can reduce portability.

### 1.1. Terminology and Notation

The Transport Services API is described in terms of

- \* Objects with which an application can interact;
- \* Actions the application can perform on these Objects;
- \* Events, which an Object can send to an application to be processed asynchronously; and
- \* Parameters associated with these Actions and Events.

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

- \* An Action that creates an Object:

Object := Action()

- \* An Action that creates an array of Objects:

[]Object := Action()

- \* An Action that is performed on an Object:

Object.Action()

- \* An Object sends an Event:

Object -> Event<>

- \* An Action takes a set of Parameters; an Event contains a set of Parameters. Action and Event parameters whose names are suffixed with a question mark are optional.

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

Objects that are passed as parameters to Actions use call-by-value behavior. Actions associated with no Object are Actions on the API; they are equivalent to Actions on a per-application global context.

Events are sent to the application or application-supplied code (e.g. framers, see Section 9.1.2) for processing; the details of event processing are platform- and implementation-specific.

We also make use of the following basic types:

- \* Boolean: Instances take the value true or false.
- \* Integer: Instances take positive or negative integer values.
- \* Numeric: Instances take positive or negative real number values.
- \* Enumeration: A family of types in which each instance takes one of a fixed, predefined set of values specific to a given enumerated type.
- \* Tuple: An ordered grouping of multiple value types, represented as a comma-separated list in parentheses, e.g., (Enumeration, Preference). Instances take a sequence of values each valid for the corresponding value type.
- \* Array: Denoted []Type, an instance takes a value for each of zero or more elements in a sequence of the given Type. An array may be of fixed or variable length.
- \* Collection: An unordered grouping of one or more values of the same type.

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

## 1.2. Specification of Requirements

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

## 2. Overview of the API Design

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

- \* A Transport Services system can offer a variety of transport protocols, independent of the Protocol Stacks that will be used at runtime. All common features of these protocol stacks are made available to the application in a transport-independent way to the degree possible. This enables applications written to a single API to make use of transport protocols in terms of the features they provide.
- \* A unified API to datagram and stream-oriented transports, allowing use of a common API for connection establishment and closing.
- \* Message-orientation, as opposed to stream-orientation, using application-assisted framing and deframing where the underlying transport does not provide these.
- \* Asynchronous Connection establishment, transmission, and reception. This allows concurrent operations during establishment and event-driven application interactions with the transport layer;
- \* Selection between alternate network paths, using additional information about the networks over which a connection can operate (e.g. Provisioning Domain (PvD) information [RFC7556]) where available.
- \* Explicit support for transport-specific features to be applied, should that particular transport be part of a chosen Protocol Stack.
- \* Explicit support for security properties as first-order transport features.



- \* Explicit support for configuration of cryptographic identities and transport security parameters persistent across multiple Connections.
- \* Explicit support for multistreaming and multipath transport protocols, and the grouping of related Connections into Connection Groups through "cloning" of Connections (see Section 7.4). This function allows applications to take full advantage of new transport protocols supporting these features.

### 3. API Summary

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

- \* by initiating the Preconnection (i.e., actively opening, as in a client; Section 7.1),
- \* through listening on the Preconnection (i.e., passively opening, as in a server Section 7.2),
- \* or rendezvousing on the Preconnection (i.e., peer to peer establishment; Section 7.3).

Once a Connection is established, data can be sent and received on it in the form of Messages. The API supports the preservation of message boundaries both via explicit Protocol Stack support, and via application support through a Message Framing that finds message boundaries in a stream. Messages are received asynchronously through event handlers registered by the application. Errors and other notifications also happen asynchronously on the Connection. It is not necessary for an application to handle all Events; some Events may have implementation-specific default handlers. The application should not assume that ignoring Events (e.g., Errors) is always safe.

### 3.1. Usage Examples

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

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

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

If none of the available transport protocols provides Preservation of Message Boundaries, but there is a transport protocol that provides a reliable ordered byte stream, an application could receive this byte stream as partial Messages and transform it into application-layer Messages. Alternatively, an application might provide a Message Framing, which can transform a sequence of Messages into a byte stream and vice versa (Section 9.1.2).

#### 3.1.1. Server Example

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

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

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

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

// Specifying a Remote Endpoint is optional when using Listen()
Preconnection := NewPreconnection(LocalSpecifier,
                                   TransportProperties,
                                   SecurityParameters)

Listener := Preconnection.Listen()

Listener -> ConnectionReceived<Connection>

// Only receive complete messages in a Conn.Received handler
Connection.Receive()

Connection -> Received<messageDataRequest, messageContext>

//---- Receive event handler begin ----
Connection.Send(messageDataResponse)
Connection.Close()

// Stop listening for incoming Connections
// (this example supports only one Connection)
Listener.Stop()
//---- Receive event handler end ----
```

### 3.1.2. Client Example

This is an example of how an application might open two Connections to a remote application using the Transport Services API, and send a request as well as receive a response on each of them.

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

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

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

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

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

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

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

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

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

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

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

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

### 3.1.3. Peer Example

This is an example of how an application might establish a connection with a peer using `Rendezvous()`, send a `Message`, and receive a `Message`.

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

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

LocalCandidates = [HostCandidate, StunCandidate]

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

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

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

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

Preconnection.AddRemote(RemoteCandidates)
Preconnection.Rendezvous()

Preconnection -> RendezvousDone<Connection>

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

Connection -> Received<messageDataResponse, messageContext>

// If new remote endpoint candidates are received from the peer over
```

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

// On a PathChange<> events, resolve the local endpoints to see if a
// new local endpoint has become available and, if so, send to the peer
// as a new candidate and add to the connection:
Connection -> PathChange<>

//---- PathChange event handler begin ----
ResolvedLocal, ResolvedRemote = Preconnection.Resolve()
if ResolvedLocal has changed:
    // ...Send the ResolvedLocal list to peer via signalling channel
    // Add the new local endpoints to the connection:
    Connection.AddLocal(ResolvedLocal)
//---- PathChange event handler end ----

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

#### 4. Transport Properties

Each application using the Transport Services API declares its preferences for how the Transport Services system should operate. This is done by using Transport Properties, as defined in [I-D.ietf-taps-arch], at each stage of the lifetime of a connection.

Transport Properties are divided into Selection, Connection, and Message Properties. Selection Properties (see Section 6.2) can only be set during pre-establishment. They are only used to specify which paths and protocol stacks can be used and are preferred by the application. Although Connection Properties (see Section 8.1) can be set during pre-establishment, they may be changed later. They are used to inform decisions made during establishment and to fine-tune the established connection. Calling Initiate on a Preconnection creates an outbound Connection or a Listener, and the Selection Properties remain readable from the Connection or Listener, but become immutable.

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

Selection Properties can be set on Preconnections, and the effect of Selection Properties can be queried on Connections and Messages. Connection Properties can be set on Connections and Preconnections; when set on Preconnections, they act as an initial default for the resulting Connections. Message Properties can be set on Messages, Connections, and Preconnections; when set on the latter two, they act as an initial default for the Messages sent over those Connections,

Note that configuring Connection Properties and Message Properties on Preconnections is preferred over setting them later. Early specification of Connection Properties allows their use as additional input to the selection process. Protocol Specific Properties, which enable configuration of specialized features of a specific protocol, see Section 3.2 of [I-D.ietf-taps-arch], are not used as an input to the selection process, but only support configuration if the respective protocol has been selected.

#### 4.1. Transport Property Names

Transport Properties are referred to by property names. For the purposes of this document, these names are alphanumeric strings in which words may be separated by hyphens. Specifically, the following characters are allowed: lowercase letters a-z, uppercase letters A-Z, digits 0-9, the hyphen -, and the underscore \_. These names serve two purposes:

- \* Allowing different components of a Transport Services implementation to pass Transport Properties, e.g., between a language frontend and a policy manager, or as a representation of properties retrieved from a file or other storage.
- \* Making the code of different Transport Services implementations look similar. While individual programming languages may preclude strict adherence to the aforementioned naming convention (for instance, by prohibiting the use of hyphens in symbols), users interacting with multiple implementations will still benefit from the consistency resulting from the use of visually similar symbols.

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

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

- \* Protocol Specific Properties MUST use the protocol acronym as the Namespace, e.g., tcp for TCP specific Transport Properties. For IETF protocols, property names under these namespaces SHOULD be defined in an RFC.
- \* Vendor or implementation specific properties MUST use a string identifying the vendor or implementation as the Namespace.

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

#### 4.2. Transport Property Types

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

Most Selection Properties (see Section 6.2) are of the Enumeration type, and use the Preference Enumeration, which takes one of five possible values (Prohibit, Avoid, Ignore, Prefer, or Require) denoting the level of preference for a given property during protocol selection.

#### 5. Scope of the API Definition

This document defines a language- and platform-independent API of a Transport Services system. Given the wide variety of languages and language conventions used to write applications that use the transport layer to connect to other applications over the Internet, this independence makes this API necessarily abstract.

There is no interoperability benefit in tightly defining how the API is presented to application programmers across diverse platforms. However, maintaining the "shape" of the abstract API across different platforms reduces the effort for programmers who learn to use the Transport Services API to then apply their knowledge to another platform.

We therefore make the following recommendations:



- \* Actions, Events, and Errors in implementations of the Transport Services API SHOULD use the names given for them in the document, subject to capitalization, punctuation, and other typographic conventions in the language of the implementation, unless the implementation itself uses different names for substantially equivalent objects for networking by convention.
- \* Transport Services systems SHOULD implement each Selection Property, Connection Property, and Message Context Property specified in this document. The Transport Services API SHOULD be implemented even when in a specific implementation/platform it will always result in no operation, e.g. there is no action when the API specifies a Property that is not available in a transport protocol implemented on a specific platform. For example, if TCP is the only underlying transport protocol, the Message Property `msgOrdered` can be implemented (trivially, as a no-op) as disabling the requirement for ordering will not have any effect on delivery order for Connections over TCP. Similarly, the `msg-lifetime` Message Property can be implemented but ignored, as the description of this Property states that "it is not guaranteed that a Message will not be sent when its Lifetime has expired".
- \* Implementations may use other representations for Transport Property Names, e.g., by providing constants, but should provide a straight-forward mapping between their representation and the property names specified here.

## 6. Pre-Establishment Phase

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

A Preconnection Object represents a potential Connection. It is a passive Object (a data structure) that merely maintains the state that describes the properties of a Connection that might exist in the future. This state comprises Local Endpoint and Remote Endpoint Objects that denote the endpoints of the potential Connection (see Section 6.1), the Selection Properties (see Section 6.2), any preconfigured Connection Properties (Section 8.1), and the security parameters (see Section 6.3):

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

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

If more than one Local Endpoint is specified on a Preconnection, then all the Local Endpoints on the Preconnection MUST represent the same host. For example, they might correspond to different interfaces on a multi-homed host, or they might correspond to local interfaces and a STUN server that can be resolved to a server reflexive address for a Preconnection used to make a peer-to-peer Rendezvous().

If more than one Remote Endpoint is specified on the Preconnection, then all the Remote Endpoints on the Preconnection SHOULD represent the same service. For example, the Remote Endpoints might represent various network interfaces of a host, or a server reflexive address that can be used to reach a host, or a set of hosts that provide equivalent local balanced service.

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

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

## 6.1. Specifying Endpoints

The transport services API uses the Local Endpoint and Remote Endpoint Objects to refer to the endpoints of a transport connection. Endpoints can be created as either Remote or Local:

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

A single Endpoint Object represents the identity of a network host. That endpoint can be more or less specific depending on which identifiers are set. For example, an Endpoint that only specifies a hostname may in fact end up corresponding to several different IP addresses on different hosts.

An Endpoint Object can be configured with the following identifiers:

- \* Hostname (string):

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

- \* Port (a 16-bit integer):

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

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

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

- \* IP address (IPv4 or IPv6 address):

```
RemoteSpecifier.WithIPv4Address(192.0.2.21)
```

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

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

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

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

An Endpoint cannot have multiple identifiers of a same type set. That is, an endpoint cannot have two IP addresses specified. Two separate IP addresses are represented as two Endpoint Objects. If a Preconnection specifies a Remote Endpoint with a specific IP address set, it will only establish Connections to that IP address. If, on the other hand, the Remote Endpoint specifies a hostname but no addresses, the Connection can perform name resolution and attempt using any address derived from the original hostname of the Remote Endpoint. Note that multiple Remote Endpoints can be added to a Preconnection, as discussed in Section 7.5.

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

The `Resolve()` action on a Preconnection can be used by the application to force early binding when required, for example with some Network Address Translator (NAT) traversal protocols (see Section 7.3).

#### 6.1.1. Using Multicast Endpoints

Specifying a multicast group address on a Local Endpoint will indicate to the Transport Services system that the resulting connection will be used to receive multicast messages. The Remote Endpoint can be used to filter incoming multicast from specific senders. Such a Preconnection will only support calling `Listen()`, not `Initiate()`. Calling `Listen()` will cause the Transport Services system to register for receiving multicast, such as issuing an IGMP join [RFC3376] or using MLD for IPV6 [RFC4604]. Any Connections that are accepted from this Listener are receive-only.

Similarly, specifying a multicast group address on the Remote Endpoint will indicate that the resulting connection will be used to send multicast messages, and that the Preconnection will support `Initiate()` but not `Listen()`. Any Connections created this way are send-only.

A `Rendezvous()` call on Preconnections containing group addresses results in an `EstablishmentError` as described in Section 7.3.

See Section 6.1.5 for more examples.

#### 6.1.2. Constraining Interfaces for Endpoints

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

- \* Specifying an interface on a `RemoteEndpoint` qualifies the scope of the remote endpoint, e.g., for link-local addresses.
- \* Specifying an interface on a `LocalEndpoint` explicitly binds all candidates derived from this endpoint to use the specified interface.

- \* Specifying an interface using the interface Selection Property (Section 6.2.11) or indirectly via the pvd Selection Property (Section 6.2.12) influences the selection among the available candidates.

While specifying an interface on an endpoint restricts the candidates available for connection establishment in the Pre-Establishment Phase, the Selection Properties prioritize and constrain the connection establishment.

#### 6.1.3. Endpoint Aliases

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

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

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

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

```
RemoteSpecifier.WithProtocol(QUIC)
```

The following example shows a case where "example.com" has a server running on port 443, with an alternate port of 8443 for QUIC.

```
RemoteSpecifier := NewRemoteEndpoint()  
RemoteSpecifier.WithHostname("example.com")  
RemoteSpecifier.WithPort(443)  
  
QUICRemoteSpecifier := NewRemoteEndpoint()  
QUICRemoteSpecifier.WithHostname("example.com")  
QUICRemoteSpecifier.WithPort(8443)  
QUICRemoteSpecifier.WithProtocol(QUIC)  
  
RemoteSpecifier.AddAlias(QUICRemoteSpecifier)
```

#### 6.1.4. Endpoint Examples

The following examples of Endpoints show common usage patterns.

Specify a Remote Endpoint using a hostname and service name:

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

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

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

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

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

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

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

As an alternative to specifying an interface name for the Local Endpoint, an application can express more fine-grained preferences using the Interface Instance or Type Selection Property, see Section 6.2.11. However, if the application specifies Selection Properties that are inconsistent with the Local Endpoint, this will result in an Error once the application attempts to open a Connection.

Specify a Local Endpoint using a STUN server:

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

#### 6.1.5. Multicast Examples

Specify a Local Endpoint using an Any-Source Multicast group to join on a named local interface:

```
LocalSpecifier := NewLocalEndpoint()  
LocalSpecifier.WithIPv4Address(233.252.0.0)  
LocalSpecifier.WithInterface("en0")
```

Source-Specific Multicast requires setting both a Local and Remote Endpoint:

```
LocalSpecifier := NewLocalEndpoint()  
LocalSpecifier.WithIPv4Address(232.1.1.1)  
LocalSpecifier.WithInterface("en0")  
  
RemoteSpecifier := NewRemoteEndpoint()  
RemoteSpecifier.WithIPv4Address(192.0.2.22)
```

One common pattern for multicast is to both send and receive multicast. For such cases, an application can set up both a Listener and a Connection. The Listener is only used to accept Connections that receive inbound multicast. The initiated Connection is only used to send multicast.

```
// Prepare multicast Listener
LocalMulticastSpecifier := NewLocalEndpoint()
LocalMulticastSpecifier.WithIPv4Address(233.252.0.0)
LocalMulticastSpecifier.WithPort(5353)
LocalMulticastSpecifier.WithInterface("en0")

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

// Specifying a Remote Endpoint is optional when using Listen()
Preconnection := NewPreconnection(LocalMulticastSpecifier,
                                   TransportProperties,
                                   SecurityParameters)

MulticastListener := Preconnection.Listen()

// Handle inbound messages sent to the multicast group
MulticastListener -> ConnectionReceived<MulticastReceiverConnection>
MulticastReceiverConnection.Receive()
MulticastReceiverConnection -> Received<messageDataRequest, messageContext>

// Prepare Connection to send multicast
LocalSpecifier := NewLocalEndpoint()
LocalSpecifier.WithPort(5353)
LocalSpecifier.WithInterface("en0")
RemoteMulticastSpecifier := NewRemoteEndpoint()
RemoteMulticastSpecifier.WithIPv4Address(233.252.0.0)
RemoteMulticastSpecifier.WithPort(5353)
RemoteMulticastSpecifier.WithInterface("en0")

Preconnection2 := NewPreconnection(LocalSpecifier,
                                   RemoteMulticastSpecifier,
                                   TransportProperties,
                                   SecurityParameters)

// Send outbound messages to the multicast group
MulticastSenderConnection := Preconnection.Initiate()
MulticastSenderConnection.Send(messageData)
```

## 6.2. Specifying Transport Properties

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



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

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

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

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

Table 1: Selection Property Preference Levels

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

Note that application preferences can conflict with each other. For example, if an application indicates a preference for a specific path by specifying an interface, but also a preference for a protocol, a situation might occur in which the preferred protocol is not available on the preferred path. In such cases, applications can expect properties that determine path selection to be prioritized over properties that determine protocol selection. The transport system **SHOULD** determine the preferred path first, regardless of protocol preferences. This ordering is chosen to provide consistency across implementations, based on the fact that it is more common for the use of a given network path to determine cost to the user (i.e., an interface type preference might be based on a user's preference to avoid being charged more for a cellular data plan).

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

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

Individual properties are then set on the TransportProperties Object. Setting a Transport Property to a value overrides the previous value of this Transport Property.

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

To aid readability, implementations **MAY** provide additional convenience functions to simplify use of Selection Properties: see Appendix B.1 for examples. In addition, implementations **MAY** provide a mechanism to create TransportProperties objects that are preconfigured for common use cases as outlined in Appendix B.2.

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

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

Section 8.1 provides a list of Connection Properties, while Selection Properties are listed in the subsections below. Many properties are only considered during establishment, and can not be changed after a Connection is established; however, they can still be queried. The return type of a queried Selection Property is Boolean, where true means that the Selection Property has been applied and false means that the Selection Property has not been applied. Note that true does not mean that a request has been honored. For example, if Congestion control was requested with preference level Prefer, but congestion control could not be supported, querying the congestionControl property yields the value false. If the preference level Avoid was used for Congestion control, and, as requested, the Connection is not congestion controlled, querying the congestionControl property also yields the value false.

An implementation of the Transport Services API must provide sensible defaults for Selection Properties. The default values for each property below represent a configuration that can be implemented over TCP. If these default values are used and TCP is not supported by a Transport Services system, then an application using the default set of Properties might not succeed in establishing a connection. Using the same default values for independent Transport Services implementations can be beneficial when applications are ported between different implementations/platforms, even if this default could lead to a connection failure when TCP is not available. If default values other than those suggested below are used, it is RECOMMENDED to clearly document any differences.

#### 6.2.1. Reliable Data Transfer (Connection)

Name: reliability

Type: Preference

Default: Require

This property specifies whether the application needs to use a transport protocol that ensures that all data is received at the Remote Endpoint without corruption. When reliable data transfer is enabled, this also entails being notified when a Connection is closed or aborted.

#### 6.2.2. Preservation of Message Boundaries

Name: preserveMsgBoundaries

Type: Preference

Default: Ignore

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

#### 6.2.3. Configure Per-Message Reliability

Name: perMsgReliability

Type: Preference

Default: Ignore

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

#### 6.2.4. Preservation of Data Ordering

Name: preserveOrder

Type: Preference

Default: Require

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

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

Name: zeroRttMsg

Type: Preference

Default: Ignore

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

#### 6.2.6. Multistream Connections in Group

Name: multistreaming

Type: Preference

Default: Prefer

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

#### 6.2.7. Full Checksum Coverage on Sending

Name: fullChecksumSend

Type: Preference

Default: Require

This property specifies the application's need for protection against corruption for all data transmitted on this Connection. Disabling this property could enable later control of the sender checksum coverage (see Section 9.1.3.6).

#### 6.2.8. Full Checksum Coverage on Receiving

Name: fullChecksumRecv

Type: Preference

Default: Require

This property specifies the application's need for protection against corruption for all data received on this Connection. Disabling this property could enable later control of the required minimum receiver checksum coverage (see Section 8.1.1).

#### 6.2.9. Congestion control

Name: congestionControl

Type: Preference

Default: Require

This property specifies whether the application would like the Connection to be congestion controlled or not. Note that if a Connection is not congestion controlled, an application using such a Connection SHOULD itself perform congestion control in accordance with [RFC2914] or use a circuit breaker in accordance with [RFC8084], whichever is appropriate. Also note that reliability is usually combined with congestion control in protocol implementations, rendering "reliable but not congestion controlled" a request that is unlikely to succeed. If the Connection is congestion controlled, performing additional congestion control in the application can have negative performance implications.

#### 6.2.10. Keep alive

Name: keepAlive

Type: Preference

Default: Ignore

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

#### 6.2.11. Interface Instance or Type

Name: interface

Type: Collection of (Preference, Enumeration)

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

This property allows the application to select any specific network interfaces or categories of interfaces it wants to Require, Prohibit, Prefer, or Avoid. Note that marking a specific interface as Require strictly limits path selection to that single interface, and often leads to less flexible and resilient connection establishment.

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

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

The set of interface types is expected to change over time as new access technologies become available. The taxonomy of interface types on a given Transport Services system is implementation-specific.

Interface types should not be treated as a proxy for properties of interfaces such as metered or unmetered network access. If an application needs to prohibit metered interfaces, this should be specified via Provisioning Domain attributes (see Section 6.2.12) or another specific property.

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

#### 6.2.12. Provisioning Domain Instance or Type

Name: pvd

Type: Collection of (Preference, Enumeration)

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

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

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

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

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

#### 6.2.13. Use Temporary Local Address

Name: useTemporaryLocalAddress

Type: Preference

Default: Avoid for Listeners and Rendezvous Connections. Prefer for other Connections.



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

#### 6.2.14. Multipath Transport

Name: multipath

Type: Enumeration

Default: Disabled for connections created through initiate and rendezvous, Passive for listeners

This property specifies whether and how applications want to take advantage of transferring data across multiple paths between the same end hosts. Using multiple paths allows connections to migrate between interfaces or aggregate bandwidth as availability and performance properties change. Possible values are:

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

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

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

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

Setting this property to "Active", can have privacy implications: It enables the transport to establish connectivity using alternate paths that might result in users being linkable across the multiple paths, even if the Advertisement of Alternative Addresses property (see Section 6.2.15 below) is set to false.

Note that Multipath Transport has no corresponding Selection Property of type Preference. Enumeration values other than "Disabled" are interpreted as a preference for choosing protocols that can make use of multiple paths. The "Disabled" value implies a requirement not to use multiple paths in parallel but does not prevent choosing a protocol that is capable of using multiple paths, e.g., it does not prevent choosing TCP, but prevents sending the MP\_CAPABLE option in the TCP handshake.

#### 6.2.15. Advertisement of Alternative Addresses

Name: `advertises-altaddr`

Type: Boolean

Default: False

This property specifies whether alternative addresses, e.g., of other interfaces, should be advertised to the peer endpoint by the protocol stack. Advertising these addresses enables the peer-endpoint to establish additional connectivity, e.g., for connection migration or using multiple paths.

Note that this can have privacy implications because it might result in users being linkable across the multiple paths. Also, note that setting this to false does not prevent the local Transport Services system from establishing connectivity using alternate paths (see Section 6.2.14 above); it only prevents proactive advertisement of addresses.

#### 6.2.16. Direction of communication

Name: `direction`

Type: Enumeration

Default: Bidirectional

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

Bidirectional: The connection must support sending and receiving

data

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

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

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

#### 6.2.17. Notification of ICMP soft error message arrival

Name: `softErrorNotify`

Type: Preference

Default: Ignore

This property specifies whether an application considers it useful to be informed when an ICMP error message arrives that does not force termination of a connection. When set to true, received ICMP errors are available as `SoftErrors`, see Section 8.3.1. Note that even if a protocol supporting this property is selected, not all ICMP errors will necessarily be delivered, so applications cannot rely upon receiving them [RFC8085].

#### 6.2.18. Initiating side is not the first to write

Name: `activeReadBeforeSend`

Type: Preference

Default: Ignore

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

### 6.3. Specifying Security Parameters and Callbacks

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

#### 6.3.1. Specifying Security Parameters on a Pre-Connection

Common security parameters such as TLS ciphersuites are known to implementations. Clients should use common safe defaults for these values whenever possible. However, as discussed in [RFC8922], many transport security protocols require specific security parameters and constraints from the client at the time of configuration and actively during a handshake. These configuration parameters need to be specified in the pre-connection phase and are created as follows:

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

Security configuration parameters and sample usage follow:

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

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

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

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

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

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

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

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

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

```
SecurityParameters := NewDisabledSecurityParameters()
```

```
SecurityParameters := NewOpportunisticSecurityParameters()
```

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

### 6.3.2. Connection Establishment Callbacks

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

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

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

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

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

## 7. Establishing Connections

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

### 7.1. Active Open: Initiate

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

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

The timeout parameter specifies how long to wait before aborting Active open. Before calling Initiate, the caller must have populated a Preconnection Object with a Remote Endpoint specifier, optionally a Local Endpoint specifier (if not specified, the system will attempt to determine a suitable Local Endpoint), as well as all properties necessary for candidate selection.

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

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

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

```
Connection -> Ready<>
```

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

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

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

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

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

## 7.2. Passive Open: Listen

Passive open is the Action of waiting for Connections from Remote Endpoints, commonly used by servers in client-server interactions. Passive open is supported by the Transport Services API through the Listen Action and returns a Listener object:

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

Before calling Listen, the caller must have initialized the Preconnection during the pre-establishment phase with a Local Endpoint specifier, as well as all properties necessary for Protocol Stack selection. A Remote Endpoint may optionally be specified, to constrain what Connections are accepted.

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

```
Listener.Stop()
```

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

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

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

```
Listener.SetNewConnectionLimit(value)
```



If the caller wants to rate-limit the number of inbound Connections that will be delivered, it can set a cap using `SetNewConnectionLimit()`. This mechanism allows a server to protect itself from being drained of resources. Each time a new Connection is delivered by the `ConnectionReceived` Event, the value is automatically decremented. Once the value reaches zero, no further Connections will be delivered until the caller sets the limit to a higher value. By default, this value is Infinite. The caller is also able to reset the value to Infinite at any point.

Listener -> EstablishmentError<reason?>

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

Listener -> Stopped<>

A Stopped Event occurs after the Listener has stopped listening.

### 7.3. Peer-to-Peer Establishment: Rendezvous

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

`Preconnection.Rendezvous()`

A Preconnection Object used in a `Rendezvous()` MUST have both the Local Endpoint candidates and the Remote Endpoint candidates specified, along with the transport properties and security parameters needed for Protocol Stack selection, before the `Rendezvous()` Action is initiated.

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

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

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

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

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

Note that the set of LocalEndpoints returned by `Resolve()` might or might not contain information about all possible local interfaces; it is valid only for a `Rendezvous` happening at the same time as the resolution. Care should be taken in using these values in any other context.

An application that uses `Rendezvous()` to establish a peer-to-peer connection in the presence of NATs will configure the Preconnection object with at least one a Local Endpoint that supports NAT binding discovery. It will then `Resolve()` the Preconnection, and pass the resulting list of Local Endpoint candidates to the peer via a signalling protocol, for example as part of an ICE [RFC5245] exchange within SIP [RFC3261] or WebRTC [RFC7478]. The peer will then, via the same signalling channel, return the Remote Endpoint candidates. The set of Remote Endpoint candidates are then configured onto the Preconnection:

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

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

If successful, the `Rendezvous()` Action returns a Connection object via a `RendezvousDone<>` Event:

```
Preconnection -> RendezvousDone<Connection>
```

The `RendezvousDone<>` Event occurs when a Connection is established with the Remote Endpoint. For Connection-oriented transports, this occurs when the transport-layer connection is established; for Connectionless transports, it occurs when the first Message is

received from the Remote Endpoint. The resulting Connection is contained within the RendezvousDone<> Event, and is ready to use as soon as it is passed to the application via the Event. Changes made to a Preconnection after Rendezvous() has been called do not have any effect on existing Connections.

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

Preconnection -> EstablishmentError<reason?>

#### 7.4. Connection Groups

Connection Groups can be created using the Clone Action:

Connection := Connection.Clone(framer?)

Calling Clone on a Connection yields a Connection Group containing two Connections: the parent Connection on which Clone was called, and a resulting cloned Connection. The new Connection is actively opened, and it will send a Ready Event or an EstablishmentError Event. Calling Clone on any of these Connections adds another Connection to the Connection Group. Connections in a Connection Group share all Connection Properties except Connection Priority (see Section 8.1.2), and these Connection Properties are entangled: Changing one of the Connection Properties on one Connection in the Connection Group automatically changes the Connection Property for all others. For example, changing Timeout for aborting Connection (see Section 8.1.3) on one Connection in a Connection Group will automatically make the same change to this Connection Property for all other Connections in the Connection Group. Like all other Properties, Connection Priority is copied to the new Connection when calling Clone(), but in this case, a later change to the Connection Priority on one Connection does not change it on the other Connections in the same Connection Group.

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

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

It is also possible to check which Connections belong to the same Connection Group. Calling GroupedConnections() on a specific Connection returns a set of all Connections in the same group.

```
[]Connection := Connection.GroupedConnections()
```

Connections will belong to the same group if the application previously called Clone. Passive Connections can also be added to the same group - e.g., when a Listener receives a new Connection that is just a new stream of an already active multi-streaming protocol instance.

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

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

Attempts to clone a Connection can result in a CloneError:

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

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

### 7.5. Adding and Removing Endpoints on a Connection

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

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

The AddRemote() action can be used to add one or more new remote endpoints to a Connection:

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

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

Similarly, the RemoveRemote() action can be used to tell a connection to stop using one or more remote endpoints:

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

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

Similarly, the AddLocal() and RemoveLocal() actions can be used to add and remove local endpoints to/from a Connection.

## 8. Managing Connections

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

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

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

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

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

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

Note that changing one of the Connection Properties on one Connection in a Connection Group will also change it for all other Connections of that group; see further Section 7.4.

At any point, the application can query Connection Properties.

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

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

- \* The connection state, which can be one of the following:  
Establishing, Established, Closing, or Closed.
- \* Whether the connection can be used to send data. A connection can not be used for sending if the connection was created with the Selection Property Direction of Communication set to unidirectional receive or if a Message marked as Final was sent over this connection. See also Section 9.1.3.5.

- \* Whether the connection can be used to receive data. A connection cannot be used for reading if the connection was created with the Selection Property Direction of Communication set to unidirectional send or if a Message marked as Final was received. See Section 9.3.3.3. The latter is only supported by certain transport protocols, e.g., by TCP as half-closed connection.
- \* For Connections that are Established, Closing, or Closed: Connection Properties (Section 8.1) of the actual protocols that were selected and instantiated, and Selection Properties that the application specified on the Preconnection. Selection Properties of type Preference will be exposed as boolean values indicating whether or not the property applies to the selected transport. Note that the instantiated protocol stack might not match all Protocol Selection Properties that the application specified on the Preconnection.
- \* For Connections that are Established: information concerning the path(s) used by the Protocol Stack. This can be derived from local PVD information, measurements by the Protocol Stack, or other sources. For example, a TAPS system that is configured to receive and process PVD information [RFC7556] could also provide network configuration information for the chosen path(s).

### 8.1. Generic Connection Properties

Generic Connection Properties are defined independent of the chosen protocol stack and therefore available on all Connections.

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

#### 8.1.1. Required Minimum Corruption Protection Coverage for Receiving

Name: `recvChecksumLen`

Type: Integer or Full Coverage

Default: Full Coverage

If this property is an Integer, it specifies the minimum number of bytes in a received message that need to be covered by a checksum. A receiving endpoint will not forward messages that have less coverage to the application. The application is responsible for handling any corruption within the non-protected part of the message [RFC8085]. A special value of 0 means that a received packet may also have a zero checksum field.

#### 8.1.2. Connection Priority

Name: connPriority

Type: Integer (non-negative)

Default: 100

This Property is a non-negative integer representing the priority of this Connection relative to other Connections in the same Connection Group. A higher value reflects a higher priority. It has no effect on Connections not part of a Connection Group. As noted in Section 7.4, this property is not entangled when Connections are cloned, i.e., changing the Priority on one Connection in a Connection Group does not change it on the other Connections in the same Connection Group. No guarantees of a specific behavior regarding Connection Priority are given; a Transport Services system may ignore this property. See Section 9.2.6 for more details.

#### 8.1.3. Timeout for Aborting Connection

Name: connTimeout

Type: Numeric or Disabled

Default: Disabled

If this property is Numeric, it specifies how long to wait before deciding that an active Connection has failed when trying to reliably deliver data to the Remote Endpoint. Adjusting this Property will only take effect when the underlying stack supports reliability. If this property has the enumerated value Disabled, it means that no timeout is scheduled.

#### 8.1.4. Timeout for keep alive packets

Name: keepAliveTimeout

Type: Numeric or Disabled

Default: Implementation-defined

A Transport Services API can request a protocol that supports sending keep alive packets Section 6.2.10. If this property is an Integer, it specifies the maximum length of time an idle connection (one for which no transport packets have been sent) should wait before the Local Endpoint sends a keep-alive packet to the Remote Endpoint. Adjusting this Property will only take effect when the underlying



stack supports sending keep-alive packets. Guidance on setting this value for datagram transports is provided in [RFC8085]. A value greater than the connection timeout (Section 8.1.3) or the enumerated value Disabled will disable the sending of keep-alive packets.

#### 8.1.5. Connection Group Transmission Scheduler

Name: connScheduler

Type: Enumeration

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

This property specifies which scheduler should be used among Connections within a Connection Group, see Section 7.4. The set of schedulers can be taken from [RFC8260].

#### 8.1.6. Capacity Profile

Name: connCapacityProfile

Type: Enumeration

Default: Default Profile (Best Effort)

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

The following values are valid for the Capacity Profile:

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

Scavenger: The application is not interactive. It expects to send

and/or receive data without any urgency. This can, for example, be used to select protocol stacks with scavenger transmission control and/or to assign the traffic to a lower-effort service. Transport Services implementations that map the requested capacity profile onto per-connection DSCP signaling SHOULD assign the DSCP Less than Best Effort [RFC8622] PHB.

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

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

**Constant-Rate Streaming:** The application expects to send/receive data at a constant rate after Connection establishment. Delay and delay variation should be minimized at the expense of efficient use of the available capacity. This implies that the Connection might fail if the Path is unable to maintain the desired rate. A transport can interpret this capacity profile as preferring a circuit breaker [RFC8084] to a rate-adaptive congestion controller. Transport system implementations that map the requested capacity profile onto per-connection DSCP signaling without multiplexing SHOULD assign a DSCP Assured Forwarding (AF31,AF32,AF33,AF34) [RFC2597] PHB.

**Capacity-Seeking:** The application expects to send/receive data at

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

The Capacity Profile for a selected protocol stack may be modified on a per-Message basis using the Transmission Profile Message Property; see Section 9.1.3.8.

#### 8.1.7. Policy for using Multipath Transports

Name: multipath-policy

Type: Enumeration

Default: Handover

This property specifies the local policy for transferring data across multiple paths between the same end hosts if Multipath Transport is not set to Disabled (see Section 6.2.14). Possible values are:

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

Interactive: The connection ought only to attempt to minimize the latency for interactive traffic patterns by transmitting data across multiple paths when this is beneficial. The goal of minimizing the latency will be balanced against the cost of each of these paths. Depending on the cost of the lower-latency path, the scheduling might choose to use a higher-latency path. Traffic can be scheduled such that data may be transmitted on multiple paths in parallel to achieve a lower latency. The specific scheduling algorithm is implementation-specific.

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

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

#### 8.1.8. Bounds on Send or Receive Rate

Name: minSendRate / minRecvRate / maxSendRate / maxRecvRate

Type: Numeric or Unlimited / Numeric or Unlimited / Numeric or Unlimited / Numeric or Unlimited

Default: Unlimited / Unlimited / Unlimited / Unlimited

Integer values of this property specify an upper-bound rate that a transfer is not expected to exceed (even if flow control and congestion control allow higher rates), and/or a lower-bound rate below which the application does not deem it will be useful. These are specified in bits per second. The enumerated value Unlimited indicates that no bound is specified.

#### 8.1.9. Group Connection Limit

Name: groupConnLimit

Type: Numeric or Unlimited

Default: Unlimited

If this property is an Integer, it controls the number of Connections that can be accepted from a peer as new members of the Connection's group. Similar to SetNewConnectionLimit(), this limits the number of ConnectionReceived Events that will occur, but constrained to the group of the Connection associated with this property. For a multi-streaming transport, this limits the number of allowed streams.

#### 8.1.10. Isolate Session

Name: isolateSession

Type: Boolean

Default: false

When set to true, this property will initiate new Connections using as little cached information (such as session tickets or cookies) as possible from previous connections that are not in the same Connection Group. Any state generated by this Connection will only be shared with Connections in the same Connection Group. Cloned Connections will use saved state from within the Connection Group. This is used for separating Connection Contexts as specified in [I-D.ietf-taps-arch].

Note that this does not guarantee no leakage of information, as implementations may not be able to fully isolate all caches (e.g. RTT estimates). Note that this property may degrade connection performance.

#### 8.1.11. Read-only Connection Properties

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

##### 8.1.11.1. Maximum Message Size Concurrent with Connection Establishment

Name: zeroRttMsgMaxLen

Type: Integer

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

##### 8.1.11.2. Maximum Message Size Before Fragmentation or Segmentation

Name: singularTransmissionMsgMaxLen

Type: Integer

This property, if applicable, represents the maximum Message size that can be sent without incurring network-layer fragmentation at the sender. It is specified as the number of bytes. It exposes a value to the application based on the Maximum Packet Size (MPS) as described in Datagram PLPMTUD [RFC8899]. This can allow a sending stack to avoid unwanted fragmentation at the network-layer or segmentation by the transport layer.

##### 8.1.11.3. Maximum Message Size on Send

Name: sendMsgMaxLen

Type: Integer

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

##### 8.1.11.4. Maximum Message Size on Receive

Name: recvMsgMaxLen

Type: Integer

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

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

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

These TCP-specific properties are included here because the feature Suggest timeout to the peer is part of the minimal set of transport services [RFC8923], where this feature was categorized as "functional". This means that when an Transport Services implementation offers this feature, the Transport Services API has to expose an interface to the application. Otherwise, the implementation might violate assumptions by the application, which could cause the application to fail.

All of the below properties are optional (e.g., it is possible to specify User Timeout Enabled as true, but not specify an Advertised User Timeout value; in this case, the TCP default will be used). These properties reflect the API extension specified in Section 3 of [RFC5482].

### 8.2.1. Advertised User Timeout

Name: tcp.userTimeoutValue

Type: Integer

Default: the TCP default

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

### 8.2.2. User Timeout Enabled

Name: tcp.userTimeoutEnabled

Type: Boolean

Default: false

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

### 8.2.3. Timeout Changeable

Name: tcp.userTimeoutChangeable

Type: Boolean

Default: true

This property controls whether the Timeout for aborting Connection (see Section 8.1.3) may be changed based on a UTO option received from the remote peer. This boolean becomes false when Timeout for aborting Connection (see Section 8.1.3) is used.

### 8.3. Connection Lifecycle Events

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

#### 8.3.1. Soft Errors

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

Connection -> SoftError<>

#### 8.3.2. Path change

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

Connection -> PathChange<>

### 9. Data Transfer

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

## 9.1. Messages and Framers

Each Message has an optional Message Context, which allows to add Message Properties, identify Send Events related to a specific Message or to inspect meta-data related to the Message sent. Framers can be used to extend or modify the message data with additional information that can be processed at the receiver to detect message boundaries.

### 9.1.1. Message Contexts

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

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

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

These Message Properties may be generic properties or Protocol Specific Properties.

For MessageContexts returned by send Events (see Section 9.2.2) and receive Events (see Section 9.3.2), the application can query information about the Local and Remote Endpoint:

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

### 9.1.2. Message Framers

Although most applications communicate over a network using well-formed Messages, the boundaries and metadata of the Messages are often not directly communicated by the transport protocol itself. For example, HTTP applications send and receive HTTP messages over a byte-stream transport, requiring that the boundaries of HTTP messages be parsed from the stream of bytes.

Message Framers allow extending a Connection's Protocol Stack to define how to encapsulate or encode outbound Messages, and how to decapsulate or decode inbound data into Messages. Message Framers allow message boundaries to be preserved when using a Connection object, even when using byte-stream transports. This is designed based on the fact that many of the current application protocols evolved over TCP, which does not provide message boundary



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

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

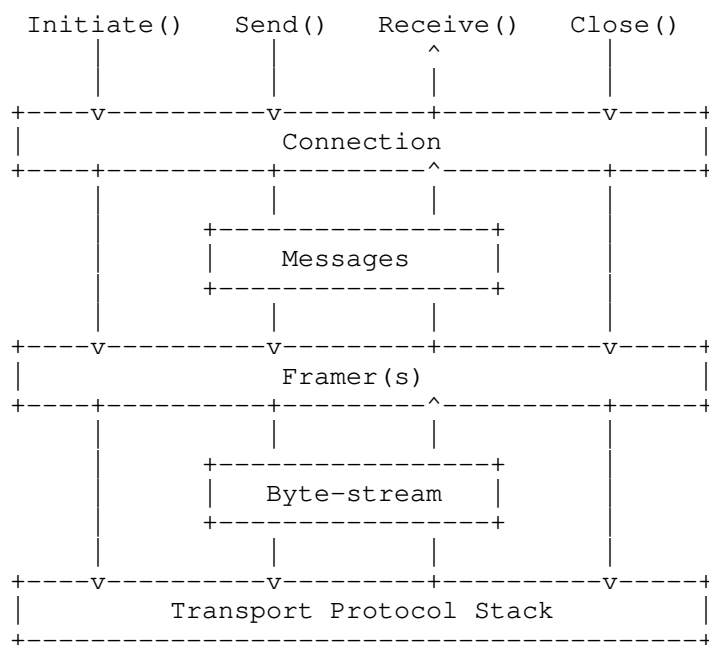


Figure 1: Protocol Stack showing a Message Framer

Note that while Message Framers add the most value when placed above a protocol that otherwise does not preserve message boundaries, they can also be used with datagram- or message-based protocols. In these cases, they add an additional transformation to further encode or encapsulate, and can potentially support packing multiple application-layer Messages into individual transport datagrams.

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

#### 9.1.2.1. Adding Message Framers to Pre-Connections

The Message Framer object can be added to one or more Preconnections to run on top of transport protocols. Multiple Framers may be added to a Preconnection; in this case, the Framers operate as a framing stack, i.e. the last one added runs first when framing outbound messages, and last when parsing inbound data.

The following example adds a basic HTTP Message Framer to a Preconnection:

```
framer := NewHTTPMessageFramer()  
Preconnection.AddFramer(framer)
```

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

#### 9.1.2.2. Framing Meta-Data

When sending Messages, applications can add Framer-specific properties to a MessageContext (Section 9.1.1). In order to set these properties, the add and get actions on the MessageContext. To avoid naming conflicts, the property names SHOULD be prefixed with a namespace referencing the framer implementation or the protocol it implements as described in Section 4.1.

This mechanism can be used, for example, to set the type of a Message for a TLV format. The namespace of values is custom for each unique Message Framer.

```
messageContext := NewMessageContext()  
messageContext.add(framer, key, value)  
Connection.Send(messageData, messageContext)
```

When an application receives a MessageContext in a Receive event, it can also look to see if a value was set by a specific Message Framer.

```
messageContext.get(framer, key) -> value
```

For example, if an HTTP Message Framer is used, the values could correspond to HTTP headers:

```
httpFramer := NewHTTPMessageFramer()  
...  
messageContext := NewMessageContext()  
messageContext.add(httpFramer, "accept", "text/html")
```

### 9.1.3. Message Properties

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

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

A MessageContext object contains metadata for the Messages to be sent or received.

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

The simpler form of Send, which does not take any messageContext, is equivalent to passing a default MessageContext without adding any Message Properties.

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

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

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

If a Message Property contradicts a Connection Property, and if this per-Message behavior can be supported, it overrides the Connection Property for the specific Message. For example, if Reliable Data Transfer (Connection) is set to Require and a protocol with configurable per-Message reliability is used, setting Reliable Data Transfer (Message) to false for a particular Message will allow this Message to be sent without any reliability guarantees. Changing the Reliable Data Transfer property on Messages is only possible for Connections that were established enabling the Selection Property Configure Per-Message Reliability.

The following Message Properties are supported:

#### 9.1.3.1. Lifetime

Name: msgLifetime

Type: Numeric

Default: infinite

The 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. This is a hint to the Transport Services implementation - it is not guaranteed that a Message will not be sent when its Lifetime has expired.

Setting a Message's Lifetime to infinite indicates that the application does not wish to apply a time constraint on the transmission of the Message, but it does not express a need for reliable delivery; reliability is adjustable per Message via the Reliable Data Transfer (Message) property (see Section 9.1.3.7). The type and units of Lifetime are implementation-specific.

#### 9.1.3.2. Priority

Name: msgPriority

Type: Integer (non-negative)

Default: 100

This property specifies the priority of a Message, relative to other Messages sent over the same Connection.

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

Note that this property is not a per-message override of the Connection Priority – see Section 8.1.2. The Priority properties may interact, but can be used independently and be realized by different mechanisms; see Section 9.2.6.

#### 9.1.3.3. Ordered

Name: msgOrdered

Type: Boolean

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

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

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

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

#### 9.1.3.4. Safely Replayable

Name: safelyReplayable

Type: Boolean

Default: false

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

For protocols that do not protect against duplicated messages, e.g., UDP, all messages need to be marked as Safely Replayable. To enable protocol selection to choose such a protocol, Safely Replayable needs to be added to the TransportProperties passed to the Preconnection. If such a protocol was chosen, disabling Safely Replayable on individual messages MUST result in a SendError.

#### 9.1.3.5. Final

Name: final

Type: Boolean

Default: false

If true, this indicates a Message is the last that the application will send on a Connection. This allows underlying protocols to indicate to the Remote Endpoint that the Connection has been effectively closed in the sending direction. For example, TCP-based Connections can send a FIN once a Message marked as Final has been completely sent, indicated by marking endOfMessage. Protocols that do not support signalling the end of a Connection in a given direction will ignore this property.

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

#### 9.1.3.6. Sending Corruption Protection Length

Name: msgChecksumLen

Type: Integer or Full Coverage

Default: Full Coverage

If this property is an Integer, it specifies the minimum length of the section of a sent Message, starting from byte 0, that the application requires to be delivered without corruption due to lower layer errors. It is used to specify options for simple integrity protection via checksums. A value of 0 means that no checksum needs to be calculated, and the enumerated value Full Coverage means that the entire Message needs to be protected by a checksum. Only Full Coverage is guaranteed, any other requests are advisory, which may result in Full Coverage being applied.

#### 9.1.3.7. Reliable Data Transfer (Message)

Name: msgReliable

Type: Boolean

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

When true, this property specifies that a Message should be sent in such a way that the transport protocol ensures all data is received on the other side without corruption. Changing the Reliable Data Transfer property on Messages is only possible for Connections that were established enabling the Selection Property Configure Per-Message Reliability. When this is not the case, changing msgReliable will generate an error.

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

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

#### 9.1.3.8. Message Capacity Profile Override

Name: msgCapacityProfile

Type: Enumeration

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

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

#### 9.1.3.9. No Network-Layer Fragmentation

Name: noFragmentation

Type: Boolean

Default: false

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

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

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

Note: `noSegmentation` should be used when it is desired to only send a message within a single network packet.

#### 9.1.3.10. No Segmentation

Name: `noSegmentation`

Type: Boolean

Default: false

When set to true, this property requests the transport layer to not provide segmentation of messages larger than the maximum size permitted by the network layer, and also to avoid network-layer source fragmentation of messages. When running over IPv4, setting this property to true can result in a sending endpoint setting the Don't Fragment bit in the IPv4 header of packets generated by the transport layer. An attempt to send a message that results in a size greater than the transport's current estimate of its maximum packet size (`singularTransmissionMsgMaxLen`) will result in a `SendError`. This only takes effect when the transport and network layer support this functionality.



## 9.2. Sending Data

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

Messages are sent on a Connection using the Send action:

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

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

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

### 9.2.1. Basic Sending

The most basic form of sending on a connection involves enqueueing a single Data block as a complete Message with default Message Properties.

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

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

Some transport protocols can deliver arbitrarily sized Messages, but other protocols constrain the maximum Message size. Applications can query the Connection Property "Maximum Message size on send" (Section 8.1.11.3) to determine the maximum size allowed for a single Message. If a Message is too large to fit in the Maximum Message Size for the Connection, the Send will fail with a SendError event (Section 9.2.2.3). For example, it is invalid to send a Message over a UDP connection that is larger than the available datagram sending size.

### 9.2.2. Send Events

Like all Actions in Transport Services API, the Send Action is asynchronous. There are several Events that can be delivered in response to Sending a Message. Exactly one Event (Sent, Expired, or SendError) will be delivered in response to each call to Send.

Note that if partial Sends are used (Section 9.2.3), there will still be exactly one Send Event delivered for each call to Send. For example, if a Message expired while two requests to Send data for that Message are outstanding, there will be two Expired events delivered.

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

#### 9.2.2.1. Sent

Connection -> Sent<messageContext>

The Sent Event occurs when a previous Send Action has completed, i.e., when the data derived from the Message has been passed down or through the underlying Protocol Stack and is no longer the responsibility of the Transport Services API. The exact disposition of the Message (i.e., whether it has actually been transmitted, moved into a buffer on the network interface, moved into a kernel buffer, and so on) when the Sent Event occurs is implementation-specific. The Sent Event contains a reference to the Message Context of the Message to which it applies.

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

#### 9.2.2.2. Expired

Connection -> Expired<messageContext>

The Expired Event occurs when a previous Send Action expired before completion; i.e. when the Message was not sent before its Lifetime (see Section 9.1.3.1) expired. This is separate from SendError, as it is an expected behavior for partially reliable transports. The Expired Event contains a reference to the Message Context of the Message to which it applies.

#### 9.2.2.3. SendError

Connection -> SendError<messageContext, reason?>

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

#### 9.2.3. Partial Sends

It is not always possible for an application to send all data associated with a Message in a single Send Action. The Message data may be too large for the application to hold in memory at one time, or the length of the Message may be unknown or unbounded.

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

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

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

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

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

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

#### 9.2.4. Batching Sends

To reduce the overhead of sending multiple small Messages on a Connection, the application could batch several Send Actions together. This provides a hint to the system that the sending of these Messages ought to be coalesced when possible, and that sending any of the batched Messages can be delayed until the last Message in the batch is enqueued.

The semantics for starting and ending a batch can be implementation-specific, but need to allow multiple Send Actions to be enqueued.

```
Connection.StartBatch()  
Connection.Send(messageData)  
Connection.Send(messageData)  
Connection.EndBatch()
```

#### 9.2.5. Send on Active Open: InitiateWithSend

For application-layer protocols where the Connection initiator also sends the first message, the InitiateWithSend() action combines Connection initiation with a first Message sent:

```
Connection := Preconnection.InitiateWithSend(messageData, messageContext?, timeout?)
```

Whenever possible, a messageContext should be provided to declare the Message passed to InitiateWithSend as Safely Replayable. This allows the Transport Services system to make use of 0-RTT establishment in case this is supported by the available protocol stacks. When the selected stack(s) do not support transmitting data upon connection establishment, InitiateWithSend is identical to Initiate() followed by Send().

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

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

#### 9.2.6. Priority and the Transport Services API

The Transport Services API provides two properties to allow a sender to signal the relative priority of data transmission: the Priority Message Property Section 9.1.3.2, and the Connection Priority Connection Property Section 8.1.2. These properties are designed to allow the expression and implementation of a wide variety of approaches to transmission priority in the transport and application layer, including those which do not appear on the wire (affecting only sender-side transmission scheduling) as well as those that do (e.g. [I-D.ietf-httpbis-priority]).

A Transport Services system gives no guarantees about how its expression of relative priorities will be realized. However, the Transport Services system will seek to ensure that performance of relatively-prioritized connections and messages is not worse with respect to those connections and messages than an equivalent configuration in which all prioritization properties are left at their defaults.

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

### 9.3. Receiving Data

Once a Connection is established, it can be used for receiving data (unless the Direction of Communication property is set to unidirectional send). As with sending, the data is received in Messages. Receiving is an asynchronous operation, in which each call to Receive enqueues a request to receive new data from the connection. Once data has been received, or an error is encountered, an event will be delivered to complete any pending Receive requests (see Section 9.3.2). If Messages arrive at the Transport Services system before Receive requests are issued, ensuing Receive requests will first operate on these Messages before awaiting any further Messages.

#### 9.3.1. Enqueueing Receives

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

`Connection.Receive(minIncompleteLength?, maxLength?)`

By default, Receive will try to deliver complete Messages in a single event (Section 9.3.2.1).

The application can set a `minIncompleteLength` value to indicate the smallest partial Message data size in bytes that should be delivered in response to this Receive. By default, this value is infinite, which means that only complete Messages should be delivered (see Section 9.3.2.2 and Section 9.1.2 for more information on how this is accomplished). If this value is set to some smaller value, the associated receive event will be triggered only when at least that many bytes are available, or the Message is complete with fewer bytes, or the system needs to free up memory. Applications should

always check the length of the data delivered to the receive event and not assume it will be as long as `minIncompleteLength` in the case of shorter complete Messages or memory issues.

The `maxLength` argument indicates the maximum size of a Message in bytes that the application is currently prepared to receive. The default value for `maxLength` is infinite. If an incoming Message is larger than the minimum of this size and the maximum Message size on receive for the Connection's Protocol Stack, it will be delivered via `ReceivedPartial` events (Section 9.3.2.2).

Note that `maxLength` does not guarantee that the application will receive that many bytes if they are available; the Transport Services API could return `ReceivedPartial` events with less data than `maxLength` according to implementation constraints. Note also that `maxLength` and `minIncompleteLength` are intended only to manage buffering, and are not interpreted as a receiver preference for message reordering.

### 9.3.2. Receive Events

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

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

#### 9.3.2.1. Received

Connection -> `Received<messageData, messageContext>`

A `Received` event indicates the delivery of a complete Message. It contains two objects, the received bytes as `messageData`, and the metadata and properties of the received Message as `messageContext`.

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

See Section 9.1.2 for handling Message framing in situations where the Protocol Stack only provides a byte-stream transport.

#### 9.3.2.2. ReceivedPartial

Connection -> ReceivedPartial<messageData, messageContext, endOfMessage>

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

Multiple invocations of ReceivedPartial deliver data for the same Message by passing the same MessageContext, until the endOfMessage flag is delivered or a ReceiveError occurs. All partial blocks of a single Message are delivered in order without gaps. This event does not support delivering discontinuous partial Messages. If, for example, Message A is divided into three pieces (A1, A2, A3) and Message B is divided into three pieces (B1, B2, B3), and preserveOrder is not Required, the ReceivedPartial may deliver them in a sequence like this: A1, B1, B2, A2, A3, B3, because the messageContext allows the application to identify the pieces as belonging to Message A and B, respectively. However, a sequence like: A1, A3 will never occur.

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

- \* the underlying Protocol Stack supports message boundary preservation, and the size of the Message is larger than the buffers available for a single message;
- \* the underlying Protocol Stack does not support message boundary preservation, and the Message Framers (see Section 9.1.2) cannot determine the end of the message using the buffer space it has available; or
- \* the underlying Protocol Stack does not support message boundary preservation, and no Message Framers were supplied by the application

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

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

```
Connection.Receive(1, 1000)

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

Connection.Receive(1, 1000)

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

#### 9.3.2.3. ReceiveError

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

A `ReceiveError` occurs when data is received by the underlying Protocol Stack that cannot be fully retrieved or parsed, and when it is useful for the application to be notified of such errors. For example, a `ReceiveError` can indicate that a `Message` (identified via the `MessageContext`) that was being partially received previously, but had not completed, encountered an error and will not be completed. This can be useful for an application, which may want to use this error as a hint to remove previously received `Message` parts from memory. As another example, if an incoming `Message` does not fulfill the Required Minimum Corruption Protection Coverage for Receiving property (see Section 8.1.1), an application can use this error as a hint to inform the peer application to adjust the Sending Corruption Protection Length property (see Section 9.1.3.6).

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

#### 9.3.3. Receive Message Properties

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



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

When available, Message metadata carries the value of the Explicit Congestion Notification (ECN) field. This information can be used for logging and debugging, and for building applications that need access to information about the transport internals for their own operation. This property is specific to UDP and UDP-Lite because these protocols do not implement congestion control, and hence expose this functionality to the application (see [RFC8293], following the guidance in [RFC8085])

#### 9.3.3.2. Early Data

In some cases it can be valuable to know whether data was read as part of early data transfer (before connection establishment has finished). This is useful if applications need to treat early data separately, e.g., if early data has different security properties than data sent after connection establishment. In the case of TLS 1.3, client early data can be replayed maliciously (see [RFC8446]). Thus, receivers might wish to perform additional checks for early data to ensure it is safely replayable. If TLS 1.3 is available and the recipient Message was sent as part of early data, the corresponding metadata carries a flag indicating as such. If early data is enabled, applications should check this metadata field for Messages received during connection establishment and respond accordingly.

#### 9.3.3.3. Receiving Final Messages

The Message Context can indicate whether or not this Message is the Final Message on a Connection. For any Message that is marked as Final, the application can assume that there will be no more Messages received on the Connection once the Message has been completely delivered. This corresponds to the Final property that may be marked on a sent Message, see Section 9.1.3.5.

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

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

## 10. Connection Termination

A Connection can be terminated i) by the Local Endpoint (i.e., the application calls the Close, CloseGroup, Abort or AbortGroup Action), ii) by the Remote Endpoint (i.e., the remote application calls the Close, CloseGroup, Abort or AbortGroup Action), or iii) because of an error (e.g., a timeout). A local call of the Close Action will cause the Connection to either send a Closed Event or a ConnectionError Event, and a local call of the CloseGroup Action will cause all of the Connections in the group to either send a Closed Event or a ConnectionError Event. A local call of the Abort Action will cause the Connection to send a ConnectionError Event, indicating local Abort as a reason, and a local call of the AbortGroup Action will cause all of the Connections in the group to send a ConnectionError Event, indicating local Abort as a reason.

Remote Action calls map to Events similar to local calls (e.g., a remote Close causes the Connection to either send a Closed Event or a ConnectionError Event), but, different from local Action calls, it is not guaranteed that such Events will indeed be invoked. When an application needs to free resources associated with a Connection, it should therefore not rely on the invocation of such Events due to termination calls from the Remote Endpoint, but instead use the local termination Actions.

Close terminates a Connection after satisfying all the requirements that were specified regarding the delivery of Messages that the application has already given to the Transport Services system. Upon successfully satisfying all these requirements, the Connection will send the Closed Event. For example, if reliable delivery was requested for a Message handed over before calling Close, the Closed Event will signify that this Message has indeed been delivered. This Action does not affect any other Connection in the same Connection Group.

Connection.Close()

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

Connection -> Closed<>

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

Connection.Abort()

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

Connection.CloseGroup()

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

Connection.AbortGroup()

A ConnectionError informs the application that: 1) data could not be delivered to the peer after a timeout, or 2) the Connection has been aborted (e.g., because the peer has called Abort). There is no guarantee that an Abort from the peer will be signaled.

Connection -> ConnectionError<reason?>

## 11. Connection State and Ordering of Operations and Events

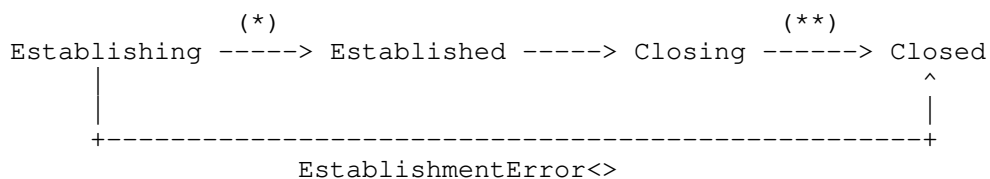
This Transport Services API is designed to be independent of an implementation's concurrency model. The details of how exactly actions are handled, and how events are dispatched, are implementation dependent.

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

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

- \* ConnectionReceived<> occurs when a Connection created with Listen() transitions to Established state.
- \* RendezvousDone<> occurs when a Connection created with Rendezvous() transitions to Established state.
- \* Closed<> occurs when a Connection transitions to Closed state without error.
- \* EstablishmentError<> occurs when a Connection created with Initiate() transitions from Establishing state to Closed state due to an error.
- \* ConnectionError<> occurs when a Connection transitions to Closed state due to an error in all other circumstances.

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



(\*) Ready<>, ConnectionReceived<>, RendezvousDone<>

(\*\*) Closed<>, ConnectionError<>

Figure 2: Connection State Diagram

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

- \* Sent<> events will occur on a Connection in the order in which the Messages were sent (i.e., delivered to the kernel or to the network interface, depending on implementation).
- \* Received<> will never occur on a Connection before it is Established; i.e. before a Ready<> event on that Connection, or a ConnectionReceived<> or RendezvousDone<> containing that Connection.

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

## 12. IANA Considerations

RFC-EDITOR: Please remove this section before publication.

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

## 13. Privacy and Security Considerations

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

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

In either case, the Transport Services API is an internal interface that is used to change information locally between two systems. However, as the Transport Services system is responsible for network communication, it is in the position to potentially share any information provided by the application with the network or another communication peer. Most of the information provided over the Transport Services API are useful to configure and select protocols and paths and are not necessarily privacy sensitive. Still, some information could be privacy sensitive because it might reveal usage characteristics and habits of the user of an application.

Of course any communication over a network reveals usage characteristics, as all packets, as well as their timing and size, are part of the network-visible wire image [RFC8546]. However, the selection of a protocol and its configuration also impacts which information is visible, potentially in clear text, and which other entities can access it. In most cases, information provided for protocol and path selection should not directly translate to information that can be observed by network devices on the path. However, there might be specific configuration information that is intended for path exposure, e.g., a DiffServ codepoint setting, that is either provided directly by the application or indirectly configured for a traffic profile.

Applications should be aware that communication attempts can lead to more than one connection establishment. This is the case, for example, when the Transport Services system also executes name resolution, when support mechanisms such as TURN or ICE are used to establish connectivity, if protocols or paths are raised, or if a path fails and fallback or re-establishment is supported in the Transport Services system.

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

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

These communication activities are not different from what is used today. However, the goal of a Transport Services system is to support such mechanisms as a generic service within the transport layer. This enables applications to more dynamically benefit from innovations and new protocols in the transport, although it reduces transparency of the underlying communication actions to the application itself. The Transport Services API is designed such that protocol and path selection can be limited to a small and controlled set if required by the application for functional or security purposes. Further, A Transport Services system should provide an

interface to poll information about which protocol and path is currently in use as well as provide logging about the communication events of each connection.

#### 14. Acknowledgements

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

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

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

This work has been supported by the Research Council of Norway under its "Toppforsk" programme through the "OCARINA" project.

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

#### 15. References

##### 15.1. Normative References

[I-D.ietf-taps-arch]

Pauly, T., Trammell, B., Brunstrom, A., Fairhurst, G., and C. Perkins, "An Architecture for Transport Services", Work in Progress, Internet-Draft, draft-ietf-taps-arch-12, 3 January 2022, <<https://www.ietf.org/archive/id/draft-ietf-taps-arch-12.txt>>.

[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>>.

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

- [RFC8084] Fairhurst, G., "Network Transport Circuit Breakers", BCP 208, RFC 8084, DOI 10.17487/RFC8084, March 2017, <<https://www.rfc-editor.org/info/rfc8084>>.
- [RFC8085] Eggert, L., Fairhurst, G., and G. Shepherd, "UDP Usage Guidelines", BCP 145, RFC 8085, DOI 10.17487/RFC8085, March 2017, <<https://www.rfc-editor.org/info/rfc8085>>.
- [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>>.
- [RFC8303] Welzl, M., Tuexen, M., and N. Khademi, "On the Usage of Transport Features Provided by IETF Transport Protocols", RFC 8303, DOI 10.17487/RFC8303, February 2018, <<https://www.rfc-editor.org/info/rfc8303>>.
- [RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <<https://www.rfc-editor.org/info/rfc8446>>.
- [RFC8981] Gont, F., Krishnan, S., Narten, T., and R. Draves, "Temporary Address Extensions for Stateless Address Autoconfiguration in IPv6", RFC 8981, DOI 10.17487/RFC8981, February 2021, <<https://www.rfc-editor.org/info/rfc8981>>.

## 15.2. Informative References

- [I-D.ietf-httpbis-priority] Oku, K. and L. Pardue, "Extensible Prioritization Scheme for HTTP", Work in Progress, Internet-Draft, draft-ietf-httpbis-priority-12, 17 January 2022, <<https://www.ietf.org/archive/id/draft-ietf-httpbis-priority-12.txt>>.
- [I-D.ietf-taps-impl] Brunstrom, A., Pauly, T., Enghardt, T., Tiesel, P. S., and M. Welzl, "Implementing Interfaces to Transport Services", Work in Progress, Internet-Draft, draft-ietf-taps-impl-11, 9 January 2022, <<https://www.ietf.org/archive/id/draft-ietf-taps-impl-11.txt>>.
- [RFC2474] Nichols, K., Blake, S., Baker, F., and D. Black, "Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers", RFC 2474, DOI 10.17487/RFC2474, December 1998, <<https://www.rfc-editor.org/info/rfc2474>>.



- [RFC2597] Heinanen, J., Baker, F., Weiss, W., and J. Wroclawski, "Assured Forwarding PHB Group", RFC 2597, DOI 10.17487/RFC2597, June 1999, <<https://www.rfc-editor.org/info/rfc2597>>.
- [RFC3246] Davie, B., Charny, A., Bennet, J.C.R., Benson, K., Le Boudec, J.Y., Courtney, W., Davari, S., Firoiu, V., and D. Stiliadis, "An Expedited Forwarding PHB (Per-Hop Behavior)", RFC 3246, DOI 10.17487/RFC3246, March 2002, <<https://www.rfc-editor.org/info/rfc3246>>.
- [RFC3261] Rosenberg, J., Schulzrinne, H., Camarillo, G., Johnston, A., Peterson, J., Sparks, R., Handley, M., and E. Schooler, "SIP: Session Initiation Protocol", RFC 3261, DOI 10.17487/RFC3261, June 2002, <<https://www.rfc-editor.org/info/rfc3261>>.
- [RFC3376] Cain, B., Deering, S., Kouvelas, I., Fenner, B., and A. Thyagarajan, "Internet Group Management Protocol, Version 3", RFC 3376, DOI 10.17487/RFC3376, October 2002, <<https://www.rfc-editor.org/info/rfc3376>>.
- [RFC4594] Babiarz, J., Chan, K., and F. Baker, "Configuration Guidelines for DiffServ Service Classes", RFC 4594, DOI 10.17487/RFC4594, August 2006, <<https://www.rfc-editor.org/info/rfc4594>>.
- [RFC4604] Holbrook, H., Cain, B., and B. Haberman, "Using Internet Group Management Protocol Version 3 (IGMPv3) and Multicast Listener Discovery Protocol Version 2 (MLDv2) for Source-Specific Multicast", RFC 4604, DOI 10.17487/RFC4604, August 2006, <<https://www.rfc-editor.org/info/rfc4604>>.
- [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>>.
- [RFC5280] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, DOI 10.17487/RFC5280, May 2008, <<https://www.rfc-editor.org/info/rfc5280>>.
- [RFC5482] Eggert, L. and F. Gont, "TCP User Timeout Option", RFC 5482, DOI 10.17487/RFC5482, March 2009, <<https://www.rfc-editor.org/info/rfc5482>>.

- [RFC5865] Baker, F., Polk, J., and M. Dolly, "A Differentiated Services Code Point (DSCP) for Capacity-Admitted Traffic", RFC 5865, DOI 10.17487/RFC5865, May 2010, <<https://www.rfc-editor.org/info/rfc5865>>.
- [RFC7478] Holmberg, C., Hakansson, S., and G. Eriksson, "Web Real-Time Communication Use Cases and Requirements", RFC 7478, DOI 10.17487/RFC7478, March 2015, <<https://www.rfc-editor.org/info/rfc7478>>.
- [RFC7556] Anipko, D., Ed., "Multiple Provisioning Domain Architecture", RFC 7556, DOI 10.17487/RFC7556, June 2015, <<https://www.rfc-editor.org/info/rfc7556>>.
- [RFC7657] Black, D., Ed. and P. Jones, "Differentiated Services (Diffserv) and Real-Time Communication", RFC 7657, DOI 10.17487/RFC7657, November 2015, <<https://www.rfc-editor.org/info/rfc7657>>.
- [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>>.
- [RFC8229] Pauly, T., Touati, S., and R. Mantha, "TCP Encapsulation of IKE and IPsec Packets", RFC 8229, DOI 10.17487/RFC8229, August 2017, <<https://www.rfc-editor.org/info/rfc8229>>.
- [RFC8260] Stewart, R., Tuexen, M., Loreto, S., and R. Seggelmann, "Stream Schedulers and User Message Interleaving for the Stream Control Transmission Protocol", RFC 8260, DOI 10.17487/RFC8260, November 2017, <<https://www.rfc-editor.org/info/rfc8260>>.
- [RFC8293] Ghanwani, A., Dunbar, L., McBride, M., Bannai, V., and R. Krishnan, "A Framework for Multicast in Network Virtualization over Layer 3", RFC 8293, DOI 10.17487/RFC8293, January 2018, <<https://www.rfc-editor.org/info/rfc8293>>.
- [RFC8445] Keranen, A., Holmberg, C., and J. Rosenberg, "Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal", RFC 8445, DOI 10.17487/RFC8445, July 2018, <<https://www.rfc-editor.org/info/rfc8445>>.

- [RFC8489] Petit-Huguenin, M., Salgueiro, G., Rosenberg, J., Wing, D., Mahy, R., and P. Matthews, "Session Traversal Utilities for NAT (STUN)", RFC 8489, DOI 10.17487/RFC8489, February 2020, <<https://www.rfc-editor.org/info/rfc8489>>.
- [RFC8546] Trammell, B. and M. Kuehlewind, "The Wire Image of a Network Protocol", RFC 8546, DOI 10.17487/RFC8546, April 2019, <<https://www.rfc-editor.org/info/rfc8546>>.
- [RFC8622] Bless, R., "A Lower-Effort Per-Hop Behavior (LE PHB) for Differentiated Services", RFC 8622, DOI 10.17487/RFC8622, June 2019, <<https://www.rfc-editor.org/info/rfc8622>>.
- [RFC8656] Reddy, T., Ed., Johnston, A., Ed., Matthews, P., and J. Rosenberg, "Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN)", RFC 8656, DOI 10.17487/RFC8656, February 2020, <<https://www.rfc-editor.org/info/rfc8656>>.
- [RFC8699] Islam, S., Welzl, M., and S. Gjessing, "Coupled Congestion Control for RTP Media", RFC 8699, DOI 10.17487/RFC8699, January 2020, <<https://www.rfc-editor.org/info/rfc8699>>.
- [RFC8838] Ivov, E., Uberti, J., and P. Saint-Andre, "Trickle ICE: Incremental Provisioning of Candidates for the Interactive Connectivity Establishment (ICE) Protocol", RFC 8838, DOI 10.17487/RFC8838, January 2021, <<https://www.rfc-editor.org/info/rfc8838>>.
- [RFC8899] Fairhurst, G., Jones, T., Tüxen, M., Rüngeler, I., and T. Völker, "Packetization Layer Path MTU Discovery for Datagram Transports", RFC 8899, DOI 10.17487/RFC8899, September 2020, <<https://www.rfc-editor.org/info/rfc8899>>.
- [RFC8922] Enghardt, T., Pauly, T., Perkins, C., Rose, K., and C. Wood, "A Survey of the Interaction between Security Protocols and Transport Services", RFC 8922, DOI 10.17487/RFC8922, October 2020, <<https://www.rfc-editor.org/info/rfc8922>>.
- [RFC8923] Welzl, M. and S. Gjessing, "A Minimal Set of Transport Services for End Systems", RFC 8923, DOI 10.17487/RFC8923, October 2020, <<https://www.rfc-editor.org/info/rfc8923>>.

[TCP-COUPLING]

Islam, S., Welzl, M., Hiorth, K., Hayes, D., Armitage, G., and S. Gjessing, "ctrlTCP: Reducing Latency through Coupled, Heterogeneous Multi-Flow TCP Congestion Control", IEEE INFOCOM Global Internet Symposium (GI) workshop (GI 2018) , 2018.

## Appendix A. Implementation Mapping

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

### A.1. Types

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

- \* An Integer can typically be represented in C by an int or long, subject to the underlying platform's ranges for each.
- \* In C, a Tuple may be represented as a struct with one member for each of the value types in the ordered grouping. In Python, by contrast, a Tuple may be represented natively as a tuple, a sequence of dynamically-typed elements.
- \* A Collection may be represented as a std::set in C++ or as a set in Python. In C, it may be represented as an array or as a higher-level data structure with appropriate accessors defined.

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

## A.2. Events and Errors

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

## A.3. Time Duration

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

## Appendix B. Convenience Functions

### B.1. Adding Preference Properties

As Selection Properties of type Preference will be set on a TransportProperties object quite frequently, implementations can provide special actions for adding each preference level i.e, TransportProperties.Set(some\_property, avoid) is equivalent to TransportProperties.Avoid(some\_property):

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

### B.2. Transport Property Profiles

To ease the use of the Transport Services API specified by this document, implementations can provide a mechanism to create Transport Property objects (see Section 6.2) that are pre-configured with frequently used sets of properties; the following are in common use in current applications:

#### B.2.1. reliable-inorder-stream

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

Property	Value
reliability	require
preserveOrder	require
congestionControl	require
preserveMsgBoundaries	ignore

Table 2: reliable-inorder-stream preferences

#### B.2.2. reliable-message

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

Property	Value
reliability	require
preserveOrder	require
congestionControl	require
preserveMsgBoundaries	require

Table 3: reliable-message preferences

#### B.2.3. unreliable-datagram

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

Property	Value
reliability	avoid
preserveOrder	avoid
congestionControl	ignore
preserveMsgBoundaries	require
safely replayable	true

Table 4: unreliable-datagram  
preferences

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

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

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

[RFC8923] identifies a minimal set of transport services that end systems should offer. These services make all non-security-related transport features of TCP, MPTCP, UDP, UDP-Lite, SCTP and LEDBAT available that 1) require interaction with the application, and 2) do not get in the way of a possible implementation over TCP (or, with limitations, UDP). The following text explains how this minimal set is reflected in the present API. For brevity, it is based on the list in Section 4.1 of [RFC8923], updated according to the discussion in Section 5 of [RFC8923]. The present API covers all elements of this section. This list is a subset of the transport features in Appendix A of [RFC8923], which refers to the primitives in "pass 2" (Section 4) of [RFC8303] for further details on the implementation with TCP, MPTCP, UDP, UDP-Lite, SCTP and LEDBAT.

\* Connect: Initiate Action (Section 7.1).

\* Listen: Listen Action (Section 7.2).

- \* Specify number of attempts and/or timeout for the first establishment message: timeout parameter of Initiate (Section 7.1) or InitiateWithSend Action (Section 9.2.5).
- \* Disable MPTCP: multipath Property (Section 6.2.14).
- \* Hand over a message to reliably transfer (possibly multiple times) before connection establishment: InitiateWithSend Action (Section 9.2.5).
- \* Change timeout for aborting connection (using retransmit limit or time value): connTimeout property, using a time value (Section 8.1.3).
- \* Timeout event when data could not be delivered for too long: ConnectionError Event (Section 10).
- \* Suggest timeout to the peer: See "TCP-specific Properties: User Timeout Option (UTO)" (Section 8.2).
- \* Notification of ICMP error message arrival: softErrorNotify (Section 6.2.17) and SoftError Event (Section 8.3.1).
- \* Choose a scheduler to operate between streams of an association: connScheduler property (Section 8.1.5).
- \* Configure priority or weight for a scheduler: connPriority property (Section 8.1.2).
- \* "Specify checksum coverage used by the sender" and "Disable checksum when sending": msgChecksumLen property (Section 9.1.3.6) and fullChecksumSend property (Section 6.2.7).
- \* "Specify minimum checksum coverage required by receiver" and "Disable checksum requirement when receiving": recvChecksumLen property (Section 8.1.1) and fullChecksumRecv property (Section 6.2.8).
- \* "Specify DF field": noFragmentation property (Section 9.1.3.9).
- \* Get max. transport-message size that may be sent using a non-fragmented IP packet from the configured interface: singularTransmissionMsgMaxLen property (Section 8.1.11.2).
- \* Get max. transport-message size that may be received from the configured interface: recvMsgMaxLen property (Section 8.1.11.4).



- \* Obtain ECN field: This is a read-only Message Property of the MessageContext object (see "UDP(-Lite)-specific Property: ECN" Section 9.3.3.1).
- \* "Specify DSCP field", "Disable Nagle algorithm", "Enable and configure a Low Extra Delay Background Transfer": as suggested in Section 5.5 of [RFC8923], these transport features are collectively offered via the connCapacityProfile property (Section 8.1.6). Per-Message control ("Request not to bundle messages") is offered via the msgCapacityProfile property (Section 9.1.3.8).
- \* Close after reliably delivering all remaining data, causing an event informing the application on the other side: this is offered by the Close Action with slightly changed semantics in line with the discussion in Section 5.2 of [RFC8923] (Section 10).
- \* "Abort without delivering remaining data, causing an event informing the application on the other side" and "Abort without delivering remaining data, not causing an event informing the application on the other side": this is offered by the Abort action without promising that this is signaled to the other side. If it is, a ConnectionError Event will be invoked at the peer (Section 10).
- \* "Reliably transfer data, with congestion control", "Reliably transfer a message, with congestion control" and "Unreliably transfer a message": data is transferred via the Send action (Section 9.2). Reliability is controlled via the reliability (Section 6.2.1) property and the msgReliable Message Property (Section 9.1.3.7). Transmitting data as a message or without delimiters is controlled via Message Framers (Section 9.1.2). The choice of congestion control is provided via the congestionControl property (Section 6.2.9).
- \* Configurable Message Reliability: the msgLifetime Message Property implements a time-based way to configure message reliability (Section 9.1.3.1).
- \* "Ordered message delivery (potentially slower than unordered)" and "Unordered message delivery (potentially faster than ordered)": these two transport features are controlled via the Message Property msgOrdered (Section 9.1.3.3).

- \* Request not to delay the acknowledgement (SACK) of a message: should the protocol support it, this is one of the transport features the Transport Services system can apply when an application uses the connCapacityProfile Property (Section 8.1.6) or the msgCapacityProfile Message Property (Section 9.1.3.8) with value Low Latency/Interactive.
- \* Receive data (with no message delimiting): Receive Action (Section 9.3) and Received Event (Section 9.3.2.1).
- \* Receive a message: Receive Action (Section 9.3) and Received Event (Section 9.3.2.1), using Message Framers (Section 9.1.2).
- \* Information about partial message arrival: Receive Action (Section 9.3) and ReceivedPartial Event (Section 9.3.2.2).
- \* Notification of send failures: Expired Event (Section 9.2.2.2) and SendError Event (Section 9.2.2.3).
- \* Notification that the stack has no more user data to send: applications can obtain this information via the Sent Event (Section 9.2.2.1).
- \* Notification to a receiver that a partial message delivery has been aborted: ReceiveError Event (Section 9.3.2.3).
- \* Notification of Excessive Retransmissions (early warning below abortion threshold): SoftError Event (Section 8.3.1).

#### Authors' Addresses

Brian Trammell (editor)  
Google Switzerland GmbH  
Gustav-Gull-Platz 1  
CH- 8004 Zurich  
Switzerland  
Email: ietf@trammell.ch

Michael Welzl (editor)  
University of Oslo  
PO Box 1080 Blindern  
0316 Oslo  
Norway  
Email: michawe@ifi.uio.no

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

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

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

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

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

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

Network Working Group  
Internet-Draft  
Intended status: Informational  
Expires: January 1, 2019

T. Pauly  
Apple Inc.  
C. Perkins  
University of Glasgow  
K. Rose  
Akamai Technologies, Inc.  
C. Wood  
Apple Inc.  
June 30, 2018

A Survey of Transport Security Protocols  
draft-ietf-taps-transport-security-02

Abstract

This document provides a survey of commonly used or notable network security protocols, with a focus on how they interact and integrate with applications and transport protocols. Its goal is to supplement efforts to define and catalog transport services [RFC8095] by describing the interfaces required to add security protocols. It examines Transport Layer Security (TLS), Datagram Transport Layer Security (DTLS), Quick UDP Internet Connections with TLS (QUIC + TLS), MinimalT, CurveCP, tcpcrypt, Internet Key Exchange with Encapsulating Security Protocol (IKEv2 + ESP), SRTP (with DTLS), and WireGuard. This survey is not limited to protocols developed within the scope or context of the IETF, and those included represent a superset of features a TAPS system may need to support.

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 January 1, 2019.

## Copyright Notice

Copyright (c) 2018 IETF Trust and the persons identified as the document authors. All rights reserved.

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

## Table of Contents

1. Introduction . . . . .	3
2. Terminology . . . . .	4
3. Security Features . . . . .	5
4. Transport Security Protocol Descriptions . . . . .	7
4.1. TLS . . . . .	7
4.1.1. Protocol Description . . . . .	7
4.1.2. Protocol Features . . . . .	8
4.1.3. Protocol Dependencies . . . . .	9
4.2. DTLS . . . . .	9
4.2.1. Protocol Description . . . . .	9
4.2.2. Protocol Features . . . . .	10
4.2.3. Protocol Dependencies . . . . .	10
4.3. (IETF) QUIC with TLS . . . . .	10
4.3.1. Protocol Description . . . . .	10
4.3.2. Protocol Features . . . . .	11
4.3.3. Protocol Dependencies . . . . .	11
4.3.4. Variant: Google QUIC . . . . .	11
4.4. IKEv2 with ESP . . . . .	12
4.4.1. Protocol descriptions . . . . .	12
4.4.2. Protocol features . . . . .	13
4.4.3. Protocol dependencies . . . . .	14
4.5. Secure RTP (with DTLS) . . . . .	14
4.5.1. Protocol description . . . . .	14
4.5.2. Protocol features . . . . .	15
4.5.3. Protocol dependencies . . . . .	16
4.5.4. Variant: ZRTP for Media Path Key Agreement . . . . .	16
4.6. tcpcrypt . . . . .	16
4.6.1. Protocol Description . . . . .	16
4.6.2. Protocol Features . . . . .	17
4.6.3. Protocol Dependencies . . . . .	18
4.7. WireGuard . . . . .	18

4.7.1.	Protocol description . . . . .	18
4.7.2.	Protocol features . . . . .	19
4.7.3.	Protocol dependencies . . . . .	19
4.8.	MinimalT . . . . .	19
4.8.1.	Protocol Description . . . . .	19
4.8.2.	Protocol Features . . . . .	20
4.8.3.	Protocol Dependencies . . . . .	20
4.9.	CurveCP . . . . .	20
4.9.1.	Protocol Description . . . . .	20
4.9.2.	Protocol Features . . . . .	22
4.9.3.	Protocol Dependencies . . . . .	22
5.	Security Features and Transport Dependencies . . . . .	22
5.1.	Mandatory Features . . . . .	22
5.2.	Optional Features . . . . .	23
5.3.	Optional Feature Availability . . . . .	25
6.	Transport Security Protocol Interfaces . . . . .	25
6.1.	Pre-Connection Interfaces . . . . .	26
6.2.	Connection Interfaces . . . . .	27
6.3.	Post-Connection Interfaces . . . . .	27
7.	IANA Considerations . . . . .	28
8.	Security Considerations . . . . .	28
9.	Acknowledgments . . . . .	28
10.	Normative References . . . . .	28
	Authors' Addresses . . . . .	32

## 1. Introduction

This document provides a survey of commonly used or notable network security protocols, with a focus on how they interact and integrate with applications and transport protocols. Its goal is to supplement efforts to define and catalog transport services [RFC8095] by describing the interfaces required to add security protocols. It examines Transport Layer Security (TLS), Datagram Transport Layer Security (DTLS), Quick UDP Internet Connections with TLS (QUIC + TLS), MinimalT, CurveCP, tcpcrypt, Internet Key Exchange with Encapsulating Security Protocol (IKEv2 + ESP), SRTP (with DTLS), and WireGuard. For each protocol, this document provides a brief description, the security features it provides, and the dependencies it has on the underlying transport. This is followed by defining the set of transport security features shared by these protocols. Finally, we distill the application and transport interfaces provided by the transport security protocols.

Selected protocols represent a superset of functionality and features a TAPS system may need to support, both internally and externally - via an API - for applications [I-D.ietf-taps-arch]. Ubiquitous IETF protocols such as (D)TLS, as well as non-standard protocols such as Google QUIC, are both included despite overlapping features. As

such, this survey is not limited to protocols developed within the scope or context of the IETF. Outside of this candidate set, protocols that do not offer new features are omitted. For example, newer protocols such as WireGuard make unique design choices that have important implications on applications, such as how to best configure peer public keys and to delegate algorithm selection to the system. In contrast, protocols such as ALTS [ALTS] are omitted since they do not represent features deemed unique.

Also, authentication-only protocols such as TCP-AO [RFC5925] and IPsec AH [RFC4302] are excluded from this survey. TCP-AO adds authenticity protections to long-lived TCP connections, e.g., replay protection with per-packet Message Authentication Codes. (This protocol obsoletes TCP MD5 "signature" options specified in [RFC2385].) One prime use case of TCP-AO is for protecting BGP connections. Similarly, AH adds per-datagram authenticity and adds similar replay protection. Despite these improvements, neither protocol sees general use and both lack critical properties important for emergent transport security protocols: confidentiality, privacy protections, and agility. Thus, we omit these and related protocols from our survey.

## 2. Terminology

The following terms are used throughout this document to describe the roles and interactions of transport security protocols:

- o Transport Feature: a specific end-to-end feature that the transport layer provides to an application. Examples include confidentiality, reliable delivery, ordered delivery, message-versus-stream orientation, etc.
- o Transport Service: a set of Transport Features, without an association to any given framing protocol, which provides functionality to an application.
- o Transport Protocol: an implementation that provides one or more different transport services using a specific framing and header format on the wire. A Transport Protocol services an application.
- o Application: an entity that uses a transport protocol for end-to-end delivery of data across the network. This may also be an upper layer protocol or tunnel encapsulation.
- o Security Feature: a feature that a network security layer provides to applications. Examples include authentication, encryption, key generation, session resumption, and privacy. Features may be Mandatory or Optional for an application's implementation.

- o Security Protocol: a defined network protocol that implements one or more security features. Security protocols may be used alongside transport protocols, and in combination with other security protocols when appropriate.
- o Handshake Protocol: a protocol that enables peers to validate each other and to securely establish shared cryptographic context.
- o Record Protocol: a security protocol that allows data to be divided into manageable blocks and protected using shared cryptographic context.
- o Session: an ephemeral security association between applications.
- o Cryptographic context: a set of cryptographic parameters, including but not necessarily limited to keys for encryption, authentication, and session resumption, enabling authorized parties to a session to communicate securely.
- o Connection: the shared state of two or more endpoints that persists across messages that are transmitted between these endpoints. A connection is a transient participant of a session, and a session generally lasts between connection instances.
- o Connection Mobility: a property of a connection that allows it to be multihomed or resilient across network interface or address changes.
- o Peer: an endpoint application party to a session.
- o Client: the peer responsible for initiating a session.
- o Server: the peer responsible for responding to a session initiation.

### 3. Security Features

In this section, we enumerate Security Features exposed by protocols discussed in the remainder of this document. Protocol security properties that are unrelated to the API surface exposed by such protocols, such as client or server identity hiding, are not listed here as features.

- o Forward-secure key establishment: Cryptographic key establishment with forward secure properties.



- o Cryptographic algorithm negotiation: Negotiate support of protocol algorithms, including: encryption, hash, MAC (PRF), and digital signature algorithms.
- o Stateful and stateless cross-connection session resumption: Connection establishment without needing to complete an entirely new handshake.
- o Peer authentication (certificate, raw public key, pre-shared key, or EAP-based): Peer authentication using select or protocol-specific mechanisms.
- o Mutual authentication: Connection establishment wherein both endpoints are authenticated.
- o Record (channel or datagram) confidentiality and integrity: Encryption and authentication of application plaintext bytes sent between peers over a channel or in individual datagrams.
- o Partial record confidentiality: Encryption of some portion of records.
- o Optional record integrity: Optional authentication of certain records.
- o Record replay prevention: Protocol detection and defense against record replays, e.g., due to in-network retransmissions.
- o Application-layer feature negotiation: Securely negotiate application-specific functionality.
- o Early data support (starting with TLS 1.3): Transmission of application data prior to connection (handshake) establishment.
- o Connection mobility: Connection continuation in the presence of 5-tuple changes beneath the secure transport protocol, e.g., due to NAT rebindings.
- o Application-layer authentication delegation: Out-of-band peer authentication performed by applications outside of the connection establishment.
- o Out-of-order record receipt: Processing of records received out-of-order.
- o DoS mitigation (cookie or puzzle based): Peer DoS mitigation via explicit proof of origin (cookie) or work mechanisms (puzzles).

- o Connection re-keying: In-band cryptographic handshake re-keying.
- o Optional length-hiding and anti-amplification padding: Protocol-drive record padding aimed at hiding plaintext message length and mitigating amplification attack vectors.

#### 4. Transport Security Protocol Descriptions

This section contains descriptions of security protocols currently used to protect data being sent over a network.

For each protocol, we describe its provided features and dependencies on other protocols.

##### 4.1. TLS

TLS (Transport Layer Security) [RFC5246] is a common protocol used to establish a secure session between two endpoints. Communication over this session "prevents eavesdropping, tampering, and message forgery." TLS consists of a tightly coupled handshake and record protocol. The handshake protocol is used to authenticate peers, negotiate protocol options, such as cryptographic algorithms, and derive session-specific keying material. The record protocol is used to marshal (possibly encrypted) data from one peer to the other. This data may contain handshake messages or raw application data.

###### 4.1.1. Protocol Description

TLS is the composition of a handshake and record protocol [I-D.ietf-tls-tls13]. The record protocol is designed to marshal an arbitrary, in-order stream of bytes from one endpoint to the other. It handles segmenting, compressing (when enabled), and encrypting data into discrete records. When configured to use an authenticated encryption with associated data (AEAD) algorithm, it also handles nonce generation and encoding for each record. The record protocol is hidden from the client behind a bytestream-oriented API.

The handshake protocol serves several purposes, including: peer authentication, protocol option (key exchange algorithm and ciphersuite) negotiation, and key derivation. Peer authentication may be mutual; however, commonly, only the server is authenticated. X.509 certificates are commonly used in this authentication step, though other mechanisms, such as raw public keys [RFC7250], exist. The client is not authenticated unless explicitly requested by the server with a CertificateRequest handshake message. Assuming strong cryptography, an infrastructure for trust establishment, correctly-functioning endpoints, and communication patterns free from side

channels, server authentication is sufficient to establish a channel resistant to eavesdroppers.

The handshake protocol is also extensible. It allows for a variety of extensions to be included by either the client or server. These extensions are used to specify client preferences, e.g., the application-layer protocol to be driven with the TLS connection [RFC7301], or signals to the server to aid operation, e.g., Server Name Indication (SNI) [RFC6066]. Various extensions also exist to tune the parameters of the record protocol, e.g., the maximum fragment length [RFC6066].

Alerts are used to convey errors and other atypical events to the endpoints. There are two classes of alerts: closure and error alerts. A closure alert is used to signal to the other peer that the sender wishes to terminate the connection. The sender typically follows a close alert with a TCP FIN segment to close the connection. Error alerts are used to indicate problems with the handshake or individual records. Most errors are fatal and are followed by connection termination. However, warning alerts may be handled at the discretion of the implementation.

Once a session is disconnected all session keying material must be destroyed, with the exception of secrets previously established expressly for purposes of session resumption. TLS supports stateful and stateless resumption. (Here, "state" refers to bookkeeping on a per-session basis by the server. It is assumed that the client must always store some state information in order to resume a session.)

#### 4.1.2. Protocol Features

- o Forward-secure key establishment.
- o Cryptographic algorithm negotiation.
- o Stateful and stateless cross-connection session resumption.
- o Peer authentication (Certificate, raw public key, and pre-shared key).
- o Mandatory server authentication, optional client authentication.
- o Record (channel) confidentiality and integrity.
- o Record replay prevention.
- o Application-layer feature negotiation.

- o Early data support (starting with TLS 1.3).
- o Optional length-hiding padding (starting with TLS 1.3).

#### 4.1.3. Protocol Dependencies

- o TCP for in-order, reliable transport.
- o (Optionally) A PKI trust store for certificate validation.

#### 4.2. DTLS

DTLS (Datagram Transport Layer Security) [RFC6347] is based on TLS, but differs in that it is designed to run over UDP instead of TCP. Since UDP does not guarantee datagram ordering or reliability, DTLS modifies the protocol to make sure it can still provide the same security guarantees as TLS. DTLS was designed to be as close to TLS as possible, so this document will assume that all properties from TLS are carried over except where specified.

##### 4.2.1. Protocol Description

DTLS is modified from TLS to operate with the possibility of packet loss, reordering, and duplication that may occur when operating over UDP. To enable out-of-order delivery of application data, the DTLS record protocol itself has no inter-record dependencies. However, as the handshake requires reliability, each handshake message is assigned an explicit sequence number to enable retransmissions of lost packets and in-order processing by the receiver. Handshake message loss is remedied by sender retransmission after a configurable period in which the expected response has not yet been received.

As the DTLS handshake protocol runs atop the record protocol, to account for long handshake messages that cannot fit within a single record, DTLS supports fragmentation and subsequent reconstruction of handshake messages across records. The receiver must reassemble records before processing.

DTLS relies on unique UDP 4-tuples to identify connections. Since all application-layer data is encrypted, demultiplexing over the same 4-tuple requires the use of a connection identifier extension [I-D.ietf-tls-dtls-connection-id] to permit identification of the correct connection-specific cryptographic context without the use of trial decryption. (Note that this extension is only supported in DTLS 1.2 and 1.3 {[I-D.ietf-tls-dtls13]}.)

Since datagrams can be replayed, DTLS provides optional anti-replay detection based on a window of acceptable sequence numbers [RFC6347].

#### 4.2.2. Protocol Features

- o Record replay protection.
- o Record (datagram) confidentiality and integrity.
- o Out-of-order record receipt.
- o DoS mitigation (cookie-based).

See also the features from TLS.

#### 4.2.3. Protocol Dependencies

- o Since DTLS runs over an unreliable, unordered datagram transport, it does not require any reliability features.
- o The DTLS record protocol explicitly encodes record lengths, so although it runs over a datagram transport, it does not rely on the transport protocol's framing beyond requiring transport-level reconstruction of datagrams fragmented over packets. (Note: DTLS 1.3 short header records omit the explicit length field.)
- o UDP 4-tuple uniqueness, or the connection identifier extension, for demultiplexing.
- o Path MTU discovery.

### 4.3. (IETF) QUIC with TLS

QUIC (Quick UDP Internet Connections) is a new standards-track transport protocol that runs over UDP, loosely based on Google's original proprietary gQUIC protocol [I-D.ietf-quic-transport]. (See Section 4.3.4 for more details.) The QUIC transport layer itself provides support for data confidentiality and integrity. This requires keys to be derived with a separate handshake protocol. A mapping for QUIC over TLS 1.3 [I-D.ietf-quic-tls] has been specified to provide this handshake.

#### 4.3.1. Protocol Description

As QUIC relies on TLS to secure its transport functions, it creates specific integration points between its security and transport functions:

- o Starting the handshake to generate keys and provide authentication (and providing the transport for the handshake).
- o Client address validation.
- o Key ready events from TLS to notify the QUIC transport.
- o Exporting secrets from TLS to the QUIC transport.

The QUIC transport layer support multiple streams over a single connection. The first stream is reserved specifically for a TLS connection. The TLS handshake, along with further records, are sent over this stream. This TLS connection follows the TLS standards and inherits the security properties of TLS. The handshake generates keys, which are then exported to the rest of the QUIC connection, and are used to protect the rest of the streams.

Initial QUIC messages (packets) are encrypted using "fixed" keys derived from the QUIC version and public packet information (Connection ID). Packets are later encrypted using keys derived from the TLS traffic secret upon handshake completion. The TLS 1.3 handshake for QUIC is used in either a single-RTT mode or a fast-open zero-RTT mode. When zero-RTT handshakes are possible, the encryption first transitions to use the zero-RTT keys before using single-RTT handshake keys after the next TLS flight.

#### 4.3.2. Protocol Features

- o DoS mitigation (cookie-based).

See also the properties of TLS.

#### 4.3.3. Protocol Dependencies

- o QUIC transport relies on UDP.
- o QUIC transport relies on TLS 1.3 for peer authentication and initial key derivation.
- o TLS within QUIC relies on QUIC for reliable handshake record transmission and receipt.

#### 4.3.4. Variant: Google QUIC

Google QUIC (gQUIC) is a UDP-based multiplexed streaming protocol designed and deployed by Google following experience from deploying SPDY, the proprietary predecessor to HTTP/2. gQUIC was originally known as "QUIC": this document uses gQUIC to unambiguously

distinguish it from the standards-track IETF QUIC. The proprietary technical forebear of IETF QUIC, gQUIC was originally designed with tightly-integrated security and application data transport protocols.

#### 4.4. IKEv2 with ESP

IKEv2 [RFC7296] and ESP [RFC4303] together form the modern IPsec protocol suite that encrypts and authenticates IP packets, either as for creating tunnels (tunnel-mode) or for direct transport connections (transport-mode). This suite of protocols separates out the key generation protocol (IKEv2) from the transport encryption protocol (ESP). Each protocol can be used independently, but this document considers them together, since that is the most common pattern.

##### 4.4.1. Protocol descriptions

###### 4.4.1.1. IKEv2

IKEv2 is a control protocol that runs on UDP port 500. Its primary goal is to generate keys for Security Associations (SAs). An SA contains shared (cryptographic) information used for establishing other SAs or keying ESP; See Section 4.4.1.2. IKEv2 first uses a Diffie-Hellman key exchange to generate keys for the "IKE SA", which is a set of keys used to encrypt further IKEv2 messages. It then goes through a phase of authentication in which both peers present blobs signed by a shared secret or private key, after which another set of keys is derived, referred to as the "Child SA". These Child SA keys are used by ESP.

IKEv2 negotiates which protocols are acceptable to each peer for both the IKE and Child SAs using "Proposals". Each proposal may contain an encryption algorithm, an authentication algorithm, a Diffie-Hellman group, and (for IKE SAs only) a pseudorandom function algorithm. Each peer may support multiple proposals, and the most preferred mutually supported proposal is chosen during the handshake.

The authentication phase of IKEv2 may use Shared Secrets, Certificates, Digital Signatures, or an EAP (Extensible Authentication Protocol) method. At a minimum, IKEv2 takes two round trips to set up both an IKE SA and a Child SA. If EAP is used, this exchange may be expanded.

Any SA used by IKEv2 can be rekeyed upon expiration, which is usually based either on time or number of bytes encrypted.

There is an extension to IKEv2 that allows session resumption [RFC5723].

MOBIKE is a Mobility and Multihoming extension to IKEv2 that allows a set of Security Associations to migrate over different addresses and interfaces [RFC4555].

When UDP is not available or well-supported on a network, IKEv2 may be encapsulated in TCP [RFC8229].

#### 4.4.1.2. ESP

ESP is a protocol that encrypts and authenticates IPv4 and IPv6 packets. The keys used for both encryption and authentication can be derived from an IKEv2 exchange. ESP Security Associations come as pairs, one for each direction between two peers. Each SA is identified by a Security Parameter Index (SPI), which is marked on each encrypted ESP packet.

ESP packets include the SPI, a sequence number, an optional Initialization Vector (IV), payload data, padding, a length and next header field, and an Integrity Check Value.

From [RFC4303], "ESP is used to provide confidentiality, data origin authentication, connectionless integrity, an anti-replay service (a form of partial sequence integrity), and limited traffic flow confidentiality."

Since ESP operates on IP packets, it is not directly tied to the transport protocols it encrypts. This means it requires little or no change from transports in order to provide security.

ESP packets may be sent directly over IP, but where network conditions warrant (e.g., when a NAT is present or when a firewall blocks such packets) they may be encapsulated in UDP [RFC3948] or TCP [RFC8229].

#### 4.4.2. Protocol features

##### 4.4.2.1. IKEv2

- o Forward-secure key establishment.
- o Cryptographic algorithm negotiation.
- o Peer authentication (Certificate, raw public key, pre-shared key, and EAP).
- o Mutual authentication.
- o Record (datagram) confidentiality and integrity.



- o Session resumption.
- o Connection mobility.
- o DoS mitigation (cookie-based).

#### 4.4.2.2. ESP

- o Record confidentiality and integrity.
- o Record replay protection.

#### 4.4.3. Protocol dependencies

##### 4.4.3.1. IKEv2

- o Availability of UDP to negotiate, or implementation support for TCP-encapsulation.
- o Some EAP authentication types require accessing a hardware device, such as a SIM card; or interacting with a user, such as password prompting.

##### 4.4.3.2. ESP

- o Since ESP is below transport protocols, it does not have any dependencies on the transports themselves, other than on UDP or TCP where encapsulation is employed.

#### 4.5. Secure RTP (with DTLS)

Secure RTP (SRTP) is a profile for RTP that provides confidentiality, message authentication, and replay protection for RTP data packets and RTP control protocol (RTCP) packets [RFC3711].

##### 4.5.1. Protocol description

SRTP adds confidentiality and optional integrity protection to RTP data packets, and adds confidentiality and mandatory integrity protection to RTCP packets. For RTP data packets, this is done by encrypting the payload section of the packet and optionally appending an authentication tag (MAC) as a packet trailer, with the RTP header authenticated but not encrypted (the RTP header was left unencrypted to enable RTP header compression [RFC2508] [RFC3545]). For RTCP packets, the first packet in the compound RTCP packet is partially encrypted, leaving the first eight octets of the header as clear-text to allow identification of the packet as RTCP, while the remainder of

the compound packet is fully encrypted. The entire RTCP packet is then authenticated by appending a MAC as packet trailer.

Packets are encrypted using session keys, which are ultimately derived from a master key and an additional master salt and session salt. SRTP packets carry a 2-byte sequence number to partially identify the unique packet index. SRTP peers maintain a separate roll-over counter (ROC) for RTP data packets that is incremented whenever the sequence number wraps. The sequence number and ROC together determine the packet index. RTCP packets have a similar, yet differently named, field called the RTCP index which serves the same purpose.

Numerous encryption modes are supported. For popular modes of operation, e.g., AES-CTR, the (unique) initialization vector (IV) used for each encryption mode is a function of the RTP SSRC (synchronization source), packet index, and session "salting key".

SRTP offers replay detection by keeping a replay list of already seen and processed packet indices. If a packet arrives with an index that matches one in the replay list, it is silently discarded.

DTLS [RFC5764] is commonly used to perform mutual authentication and key agreement for SRTP [RFC5763]. Peers use DTLS to perform mutual certificate-based authentication on the media path, and to generate the SRTP master key. Peer certificates can be issued and signed by a certificate authority. Alternatively, certificates used in the DTLS exchange can be self-signed. If they are self-signed, certificate fingerprints are included in the signalling exchange (e.g., in SIP or WebRTC), and used to bind the DTLS key exchange in the media plane to the signalling plane. The combination of a mutually authenticated DTLS key exchange on the media path and a fingerprint sent in the signalling channel protects against active attacks on the media, provided the signalling can be trusted. Signalling needs to be protected as described in, for example, SIP [RFC3261] Authenticated Identity Management [RFC4474] or the WebRTC security architecture [I-D.ietf-rtcweb-security-arch], to provide complete system security.

#### 4.5.2. Protocol features

- o Forward-secure key establishment.
- o Cryptographic algorithm negotiation.
- o Mutual authentication.
- o Partial datagram confidentiality. (Packet headers are not encrypted.)

- o Optional authentication of data packets.
- o Mandatory authentication of control packets.
- o Out-of-order record receipt.

#### 4.5.3. Protocol dependencies

- o External key derivation and management protocol, e.g., DTLS [RFC5763].
- o External identity management protocol, e.g., SIP Authenticated Identity Management [RFC4474], WebRTC Security Architecture [I-D.ietf-rtcweb-security-arch].

#### 4.5.4. Variant: ZRTP for Media Path Key Agreement

ZRTP [RFC6189] is an alternative key agreement protocol for SRTP. It uses standard SRTP to protect RTP data packets and RTCP packets, but provides alternative key agreement and identity management protocols.

Key agreement is performed using a Diffie-Hellman key exchange that runs on the media path. This generates a shared secret that is then used to generate the master key and salt for SRTP.

ZRTP does not rely on a PKI or external identity management system. Rather, it uses an ephemeral Diffie-Hellman key exchange with hash commitment to allow detection of man-in-the-middle attacks. This requires endpoints to display a short authentication string that the users must read and verbally compare to validate the hashes and ensure security. Endpoints cache some key material after the first call to use in subsequent calls; this is mixed in with the Diffie-Hellman shared secret, so the short authentication string need only be checked once for a given user. This gives key continuity properties analogous to the secure shell (ssh) [RFC4253].

#### 4.6. tcpcrypt

Tcpcrypt is a lightweight extension to the TCP protocol to enable opportunistic encryption with hooks available to the application layer for implementation of endpoint authentication.

##### 4.6.1. Protocol Description

Tcpcrypt extends TCP to enable opportunistic encryption between the two ends of a TCP connection [I-D.ietf-tcpinc-tcpcrypt]. It is a family of TCP encryption protocols (TEP), distinguished by key exchange algorithm. The use of a TEP is negotiated with a TCP option

during the initial TCP handshake via the mechanism described by TCP Encryption Negotiation Option (ENO) [I-D.ietf-tcpinc-tcpenc]. In the case of initial session establishment, once a tcpcrypt TEP has been negotiated the key exchange occurs within the data segments of the first few packets exchanged after the handshake completes. The initiator of a connection sends a list of supported AEAD algorithms, a random nonce, and an ephemeral public key share. The responder typically chooses a mutually-supported AEAD algorithm and replies with this choice, its own nonce, and ephemeral key share. An initial shared secret is derived from the ENO handshake, the tcpcrypt handshake, and the initial keying material resulting from the key exchange. The traffic encryption keys on the initial connection are derived from the shared secret. Connections can be re-keyed before the natural AEAD limit for a single set of traffic encryption keys is reached.

Each tcpcrypt session is associated with a ladder of resumption IDs, each derived from the respective entry in a ladder of shared secrets. These resumption IDs can be used to negotiate a stateful resumption of the session in a subsequent connection, resulting in use of a new shared secret and traffic encryption keys without requiring a new key exchange. Willingness to resume a session is signaled via the ENO option during the TCP handshake. Given the length constraints imposed by TCP options, unlike stateless resumption mechanisms (such as that provided by session tickets in TLS) resumption in tcpcrypt requires the maintenance of state on the server, and so successful resumption across a pool of servers implies shared state.

Owing to middlebox ossification issues, tcpcrypt only protects the payload portion of a TCP packet. It does not encrypt any header information, such as the TCP sequence number.

Tcpcrypt exposes a universally-unique connection-specific session ID to the application, suitable for application-level endpoint authentication either in-band or out-of-band.

#### 4.6.2. Protocol Features

- o Forward-secure key establishment.
- o Record (channel) confidentiality and integrity.
- o Stateful cross-connection session resumption.
- o Connection re-keying.
- o Application authentication delegation.

#### 4.6.3. Protocol Dependencies

- o TCP
- o TCP Encryption Negotiation Option (ENO)

#### 4.7. WireGuard

WireGuard is a layer 3 protocol designed to complement or replace IPsec [WireGuard]. Unlike most transport security protocols, which rely on PKI for peer authentication, WireGuard authenticates peers using pre-shared public keys delivered out-of-band, each of which is bound to one or more IP addresses. Moreover, as a protocol suited for VPNs, WireGuard offers no extensibility, negotiation, or cryptographic agility.

##### 4.7.1. Protocol description

WireGuard is a simple VPN protocol that binds a pre-shared public key to one or more IP addresses. Users configure WireGuard by associating peer public keys with IP addresses. These mappings are stored in a CryptoKey Routing Table. (See Section 2 of [WireGuard] for more details and sample configurations.) These keys are used upon WireGuard packet transmission and reception. For example, upon receipt of a Handshake Initiation message, receivers use the static public key in their CryptoKey routing table to perform necessary cryptographic computations.

WireGuard builds on Noise [Noise] for 1-RTT key exchange with identity hiding. The handshake hides peer identities as per the SIGMA construction [SIGMA]. As a consequence of using Noise, WireGuard comes with a fixed set of cryptographic algorithms:

- o x25519 [Curve25519] and HKDF [RFC5869] for ECDH and key derivation.
- o ChaCha20+Poly1305 [RFC7539] for packet authenticated encryption.
- o BLAKE2s [BLAKE2] for hashing.

There is no cryptographic agility. If weaknesses are found in any of these algorithms, new message types using new algorithms must be introduced.

WireGuard is designed to be entirely stateless, modulo the CryptoKey routing table, which has size linear with the number of trusted peers. If a WireGuard receiver is under heavy load and cannot process a packet, e.g., cannot spare CPU cycles for point

multiplication, it can reply with a cookie similar to DTLS and IKEv2. This cookie only proves IP address ownership. Any rate limiting scheme can be applied to packets coming from non-spoofed addresses.

#### 4.7.2. Protocol features

- o Forward-secure key establishment.
- o Peer authentication (Public-key and PSK).
- o Mutual authentication.
- o Record replay prevention (Stateful, timestamp-based).
- o DoS mitigation (cookie-based).

#### 4.7.3. Protocol dependencies

- o Datagram transport.
- o Out-of-band key distribution and management.

#### 4.8. MinimalT

MinimalT is a UDP-based transport security protocol designed to offer confidentiality, mutual authentication, DoS prevention, and connection mobility [MinimalT]. One major goal of the protocol is to leverage existing protocols to obtain server-side configuration information used to more quickly bootstrap a connection. MinimalT uses a variant of TCP's congestion control algorithm.

##### 4.8.1. Protocol Description

MinimalT is a secure transport protocol built on top of a widespread directory service. Clients and servers interact with local directory services to (a) resolve server information and (b) public ephemeral state information, respectively. Clients connect to a local resolver once at boot time. Through this resolver they recover the IP address(es) and public key(s) of each server to which they want to connect.

Connections are instances of user-authenticated, mobile sessions between two endpoints. Connections run within tunnels between hosts. A tunnel is a server-authenticated container that multiplexes multiple connections between the same hosts. All connections in a tunnel share the same transport state machine and encryption. Each tunnel has a dedicated control connection used to configure and manage the tunnel over time. Moreover, since tunnels are independent

of the network address information, they may be reused as both ends of the tunnel move about the network. This does however imply that the connection establishment and packet encryption mechanisms are coupled.

Before a client connects to a remote service, it must first establish a tunnel to the host providing or offering the service. Tunnels are established in 1-RTT using an ephemeral key obtained from the directory service. Tunnel initiators provide their own ephemeral key and, optionally, a DoS puzzle solution such that the recipient (server) can verify the authenticity of the request and derive a shared secret. Within a tunnel, new connections to services may be established.

#### 4.8.2. Protocol Features

- o Forward-secure key establishment.
- o DoS mitigation (puzzle-based).
- o Connection mobility (tunnel-based).

Additional (orthogonal) transport features include: connection multiplexing between hosts across shared tunnels, and congestion control state is shared across connections between the same host pairs.

#### 4.8.3. Protocol Dependencies

- o A DNS-like resolution service to obtain location information (an IP address) and ephemeral keys.
- o A PKI trust store for certificate validation.

### 4.9. CurveCP

CurveCP [CurveCP] is a UDP-based transport security protocol from Daniel J. Bernstein. Unlike other transport security protocols, it is based entirely upon highly efficient public key algorithms. This removes many pitfalls associated with nonce reuse and key synchronization.

#### 4.9.1. Protocol Description

CurveCP is a UDP-based transport security protocol. It is built on three principal features: exclusive use of public key authenticated encryption of packets, server-chosen cookies to prohibit memory and

computation DoS at the server, and connection mobility with a client-chosen ephemeral identifier.

There are two rounds in CurveCP. In the first round, the client sends its first initialization packet to the server, carrying its (possibly fresh) ephemeral public key  $C'$ , with zero-padding encrypted under the server's long-term public key. The server replies with a cookie and its own ephemeral key  $S'$  and a cookie that is to be used by the client. Upon receipt, the client then generates its second initialization packet carrying: the ephemeral key  $C'$ , cookie, and an encryption of  $C'$ , the server's domain name, and, optionally, some message data. The server verifies the cookie and the encrypted payload and, if valid, proceeds to send data in return. At this point, the connection is established and the two parties can communicate.

The use of only public-key encryption and authentication, or "boxing", is done to simplify problems that come with symmetric key management and synchronization. For example, it allows the sender of a message to be in complete control of each message's nonce. It does not require either end to share secret keying material. Furthermore, it allows connections (or sessions) to be associated with unique ephemeral public keys as a mechanism for enabling forward secrecy given the risk of long-term private key compromise.

The client and server do not perform a standard key exchange. Instead, in the initial exchange of packets, each party provides its own ephemeral key to the other end. The client can choose a new ephemeral key for every new connection. However, the server must rotate these keys on a slower basis. Otherwise, it would be trivial for an attacker to force the server to create and store ephemeral keys with a fake client initialization packet.

Unlike TCP, the server employs cookies to enable source validation. After receiving the client's initial packet, encrypted under the server's long-term public key, the server generates and returns a stateless cookie that must be echoed back in the client's following message. This cookie is encrypted under the client's ephemeral public key. This stateless technique prevents attackers from hijacking client initialization packets to obtain cookie values to flood clients. (A client would detect the duplicate cookies and reject the flooded packets.) Similarly, replaying the client's second packet, carrying the cookie, will be detected by the server.

CurveCP supports a weak form of client authentication. Clients are permitted to send their long-term public keys in the second initialization packet. A server can verify this public key and, if untrusted, drop the connection and subsequent data.



Unlike some other protocols, CurveCP data packets leave only the ephemeral public key, the connection ID, and the per-message nonce in the clear. Everything else is encrypted.

#### 4.9.2. Protocol Features

- o Datagram confidentiality and integrity (via public key encryption).
- o 1-RTT session bootstrapping.
- o Connection mobility (based on a client-chosen ephemeral identifier).
- o Optional length-hiding and anti-amplification padding.

#### 4.9.3. Protocol Dependencies

- o An unreliable transport protocol such as UDP.

### 5. Security Features and Transport Dependencies

There exists a common set of features shared across the transport protocols surveyed in this document. Mandatory features constitute a baseline of functionality that an application may assume for any TAPS implementation. Optional features by contrast may vary from implementation to implementation, and so an application cannot simply assume they are available. Applications learn of and use optional features by querying for their presence and support. Optional features may not be implemented, or may be disabled if their presence impacts transport services or if a necessary transport service is unavailable.

#### 5.1. Mandatory Features

- o Segment or datagram encryption and authentication: Transit data must be protected with an authenticated encryption algorithm.
- o Forward-secure key establishment: Negotiated keying material must come from an authenticated, forward-secure key exchange protocol.
- o Public-key (raw or certificate) based authentication: Authentication based on public key signatures is commonplace for many transport security protocols.
- o Responder authentication: The endpoint (receiver) of a new connection must be authenticated before any data is sent to said party.

- o Pre-shared key support: A security protocol must be able to use a pre-shared key established out-of-band or from a prior session to encrypt individual messages, packets, or datagrams.

## 5.2. Optional Features

- o Cryptographic algorithm negotiation (AN): Transport security protocols should permit applications to configure supported or enabled cryptographic algorithms.
  - \* Transport dependency: None.
  - \* Application dependency: Application awareness of supported or desired algorithms.
- o Application authentication delegation (AD): Some protocols may completely defer endpoint authentication to applications, e.g., to reduce online protocol complexity.
  - \* Transport dependency: None.
  - \* Application dependency: Application opt-in and policy for endpoint authentication
- o Mutual authentication (MA): Transport security protocols should allow each endpoint to authenticate the other if required by the application.
  - \* Transport dependency: None.
  - \* Application dependency: Mutual authentication required for application support.
- o DoS mitigation (DM): Transport security protocols may need to support volumetric DoS prevention via, e.g., cookies or initiator-side puzzles.
  - \* Transport dependency: None.
  - \* Application dependency: None.
- o Connection mobility (CM): Sessions should not be bound to a network connection (or 5-tuple). This allows cryptographic key material and other state information to be reused in the event of a connection change. Examples of this include a NAT rebinding that occurs without a client's knowledge.

- \* Transport dependency: Connections are unreliable or can change due to unpredictable network events, e.g., NAT re-bindings.
- \* Application dependency: None.
- o Source validation (SV): Source validation must be provided to mitigate server-targeted DoS attacks. This can be done with puzzles or cookies.
  - \* Transport dependency: Packets may arrive as datagrams instead of streams from unauthenticated sources.
  - \* Application dependency: None.
- o Application-layer feature negotiation (AFN): The type of application using a transport security protocol often requires features configured at the connection establishment layer, e.g., ALPN [RFC7301]. Moreover, application-layer features may often be used to offload the session to another server which can better handle the request. (The TLS SNI is one example of such a feature.) As such, transport security protocols should provide a generic mechanism to allow for such application-specific features and options to be configured or otherwise negotiated.
  - \* Transport dependency: None.
  - \* Application dependency: Specification of application-layer features or functionality.
- o Configuration extensions (CX): The protocol negotiation should be extensible with addition of new configuration options.
  - \* Transport dependency: None.
  - \* Application dependency: Specification of application-specific extensions.
- o Session caching and management (SC): Sessions should be cacheable to enable reuse and amortize the cost of performing session establishment handshakes.
  - \* Transport dependency: None.
  - \* Application dependency: None.
- o Length-hiding padding (LHP): Applications may wish to defer traffic padding to the security protocol to deter traffic analysis attacks.

\* Transport dependency: None.

\* Application dependency: Knowledge of desired padding policies.

### 5.3. Optional Feature Availability

The following table lists the availability of the above-listed optional features in each of the analyzed protocols. "Mandatory" indicates that the feature is intrinsic to the protocol and cannot be disabled. "Supported" indicates that the feature is optionally provided natively or through a (standardized, where applicable) extension.

Protocol	AN	AD	MA	DM	CM	SV	AFN	CX	SC	LHP
TLS	S	S	S	S	U*	M	S	S	S	S
DTLS	S	S	S	S	S	M	S	S	S	S
IETF QUIC	S	S	S	S	S	M	S	S	S	S
IKEv2+ESP	S	S	M	S	S	M	S	S	S	S
SRTP+DTLS	S	S	S	S	U	M	S	S	S	U
tcpcrypt	S	M	U	U**	U*	M	U	U	S	U
WireGuard	U	S	M	S	U	M	U	U	U	S+
MinimalT	U	U	M	S	M	M	U	U	U	S
CurveCP	U	U	S	S	M	M	U	U	U	S

M=Mandatory

S=Supported but not required

U=Unsupported

\*=On TCP; MPTCP would provide this ability

\*\*=TCP provides SYN cookies natively, but these are not cryptographically strong

+ =For transport packets only

## 6. Transport Security Protocol Interfaces

This section describes the interface surface exposed by the security protocols described above. Note that not all protocols support each

interface. We partition these interfaces into pre-connection (configuration), connection, and post-connection interfaces.

#### 6.1. Pre-Connection Interfaces

Configuration interfaces are used to configure the security protocols before a handshake begins or the keys are negotiated.

- o Identity and Private Keys  
The application can provide its identities (certificates) and private keys, or mechanisms to access these, to the security protocol to use during handshakes.  
Protocols: TLS, DTLS, QUIC + TLS, MinimalT, CurveCP, IKEv2, WireGuard, SRTP
- o Supported Algorithms (Key Exchange, Signatures, and Ciphersuites)  
The application can choose the algorithms that are supported for key exchange, signatures, and ciphersuites.  
Protocols: TLS, DTLS, QUIC + TLS, MinimalT, tcpcrypt, IKEv2, SRTP
- o Session Cache Management  
The application provides the ability to save and retrieve session state (such as tickets, keying material, and server parameters) that may be used to resume the security session.  
Protocols: TLS, DTLS, QUIC + TLS, MinimalT
- o Authentication Delegation  
The application provides access to a separate module that will provide authentication, using EAP for example.  
Protocols: IKEv2, SRTP
- o Pre-Shared Key Import  
Either the handshake protocol or the application directly can supply pre-shared keys for the record protocol use for encryption/decryption and authentication. If the application can supply keys directly, this is considered explicit import; if the handshake protocol traditionally provides the keys directly, it is considered direct import; if the keys can only be shared by the handshake, they are considered non-importable.
  - \* Explicit import: QUIC, ESP
  - \* Direct import: TLS, DTLS, MinimalT, tcpcrypt, WireGuard
  - \* Non-importable: CurveCP

## 6.2. Connection Interfaces

- o Identity Validation  
During a handshake, the security protocol will conduct identity validation of the peer. This can call into the application to offload validation. Protocols: All (TLS, DTLS, QUIC + TLS, MinimalT, CurveCP, IKEv2, WireGuard, SRTP (DTLS))
- o Source Address Validation  
The handshake protocol may delegate validation of the remote peer that has sent data to the transport protocol or application. This involves sending a cookie exchange to avoid DoS attacks. Protocols: QUIC + TLS, DTLS, WireGuard

## 6.3. Post-Connection Interfaces

- o Connection Termination The security protocol may be instructed to tear down its connection and session information. This is needed by some protocols to prevent application data truncation attacks. Protocols: TLS, DTLS, QUIC + TLS, MinimalT, tcpcrypt, IKEv2
- o Key Update  
The handshake protocol may be instructed to update its keying material, either by the application directly or by the record protocol sending a key expiration event. Protocols: TLS, DTLS, QUIC + TLS, MinimalT, tcpcrypt, IKEv2
- o Pre-Shared Key Export The handshake protocol will generate one or more keys to be used for record encryption/decryption and authentication. These may be explicitly exportable to the application, traditionally limited to direct export to the record protocol, or inherently non-exportable because the keys must be used directly in conjunction with the record protocol.
  - \* Explicit export: TLS (for QUIC), tcpcrypt, IKEv2, DTLS (for SRTP)
  - \* Direct export: TLS, DTLS, MinimalT
  - \* Non-exportable: CurveCP
- o Key Expiration  
The record protocol can signal that its keys are expiring due to reaching a time-based deadline, or a use-based deadline (number of bytes that have been encrypted with the key). This interaction is often limited to signaling between the record layer and the handshake layer.

Protocols: ESP ((Editor's note: One may consider TLS/DTLS to also have this interface))

- o Connection mobility

The record protocol can be signaled that it is being migrated to another transport or interface due to connection mobility, which may reset address and state validation.

Protocols: QUIC, MinimalT, CurveCP, ESP, WireGuard (roaming)

## 7. IANA Considerations

This document has no request to IANA.

## 8. Security Considerations

This document summarizes existing transport security protocols and their interfaces. It does not propose changes to or recommend usage of reference protocols. Moreover, no claims of security and privacy properties beyond those guaranteed by the protocols discussed are made. For example, metadata leakage via timing side channels and traffic analysis may compromise any protocol discussed in this survey. Applications using Security Interfaces SHOULD take such limitations into consideration when using a particular protocol implementation.

## 9. Acknowledgments

The authors would like to thank Mirja Kuehlewind, Brian Trammell, Yannick Sierra, Frederic Jacobs, and Bob Bradley for their input and feedback on earlier versions of this draft.

## 10. Normative References

[ALTS] "Application Layer Transport Security", n.d..

[BLAKE2] "BLAKE2 -- simpler, smaller, fast as MD5", n.d..

[Curve25519]  
"Curve25519 - new Diffie-Hellman speed records", n.d..

[CurveCP] "CurveCP -- Usable security for the Internet", n.d..

[I-D.ietf-quic-tls]  
Thomson, M. and S. Turner, "Using Transport Layer Security (TLS) to Secure QUIC", draft-ietf-quic-tls-13 (work in progress), June 2018.

- [I-D.ietf-quic-transport]  
Iyengar, J. and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport", draft-ietf-quic-transport-13 (work in progress), June 2018.
- [I-D.ietf-rtcweb-security-arch]  
Rescorla, E., "WebRTC Security Architecture", draft-ietf-rtcweb-security-arch-14 (work in progress), March 2018.
- [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-00 (work in progress), April 2018.
- [I-D.ietf-tcpinc-tcpcrypt]  
Bittau, A., Giffin, D., Handley, M., Mazieres, D., Slack, Q., and E. Smith, "Cryptographic protection of TCP Streams (tcpcrypt)", draft-ietf-tcpinc-tcpcrypt-12 (work in progress), June 2018.
- [I-D.ietf-tcpinc-tcpno]  
Bittau, A., Giffin, D., Handley, M., Mazieres, D., and E. Smith, "TCP-ENO: Encryption Negotiation Option", draft-ietf-tcpinc-tcpno-19 (work in progress), June 2018.
- [I-D.ietf-tls-dtls-connection-id]  
Rescorla, E., Tschofenig, H., Fossati, T., and T. Gondrom, "The Datagram Transport Layer Security (DTLS) Connection Identifier", draft-ietf-tls-dtls-connection-id-00 (work in progress), December 2017.
- [I-D.ietf-tls-dtls13]  
Rescorla, E., Tschofenig, H., and N. Modadugu, "The Datagram Transport Layer Security (DTLS) Protocol Version 1.3", draft-ietf-tls-dtls13-26 (work in progress), March 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.
- [MinimalT]  
"MinimalT -- Minimal-latency Networking Through Better Security", n.d..
- [Noise]  
"The Noise Protocol Framework", n.d..



- [RFC2385] Heffernan, A., "Protection of BGP Sessions via the TCP MD5 Signature Option", RFC 2385, DOI 10.17487/RFC2385, August 1998, <<https://www.rfc-editor.org/info/rfc2385>>.
- [RFC2508] Casner, S. and V. Jacobson, "Compressing IP/UDP/RTP Headers for Low-Speed Serial Links", RFC 2508, DOI 10.17487/RFC2508, February 1999, <<https://www.rfc-editor.org/info/rfc2508>>.
- [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>>.
- [RFC3545] Koren, T., Casner, S., Geevarghese, J., Thompson, B., and P. Ruddy, "Enhanced Compressed RTP (CRTP) for Links with High Delay, Packet Loss and Reordering", RFC 3545, DOI 10.17487/RFC3545, July 2003, <<https://www.rfc-editor.org/info/rfc3545>>.
- [RFC3711] Baugher, M., McGrew, D., Naslund, M., Carrara, E., and K. Norrman, "The Secure Real-time Transport Protocol (SRTP)", RFC 3711, DOI 10.17487/RFC3711, March 2004, <<https://www.rfc-editor.org/info/rfc3711>>.
- [RFC3948] Huttunen, A., Swander, B., Volpe, V., DiBurro, L., and M. Stenberg, "UDP Encapsulation of IPsec ESP Packets", RFC 3948, DOI 10.17487/RFC3948, January 2005, <<https://www.rfc-editor.org/info/rfc3948>>.
- [RFC4253] Ylonen, T. and C. Lonvick, Ed., "The Secure Shell (SSH) Transport Layer Protocol", RFC 4253, DOI 10.17487/RFC4253, January 2006, <<https://www.rfc-editor.org/info/rfc4253>>.
- [RFC4302] Kent, S., "IP Authentication Header", RFC 4302, DOI 10.17487/RFC4302, December 2005, <<https://www.rfc-editor.org/info/rfc4302>>.
- [RFC4303] Kent, S., "IP Encapsulating Security Payload (ESP)", RFC 4303, DOI 10.17487/RFC4303, December 2005, <<https://www.rfc-editor.org/info/rfc4303>>.
- [RFC4474] Peterson, J. and C. Jennings, "Enhancements for Authenticated Identity Management in the Session Initiation Protocol (SIP)", RFC 4474, DOI 10.17487/RFC4474, August 2006, <<https://www.rfc-editor.org/info/rfc4474>>.

- [RFC4555] Eronen, P., "IKEv2 Mobility and Multihoming Protocol (MOBIKE)", RFC 4555, DOI 10.17487/RFC4555, June 2006, <<https://www.rfc-editor.org/info/rfc4555>>.
- [RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, DOI 10.17487/RFC5246, August 2008, <<https://www.rfc-editor.org/info/rfc5246>>.
- [RFC5723] Sheffer, Y. and H. Tschofenig, "Internet Key Exchange Protocol Version 2 (IKEv2) Session Resumption", RFC 5723, DOI 10.17487/RFC5723, January 2010, <<https://www.rfc-editor.org/info/rfc5723>>.
- [RFC5763] Fischl, J., Tschofenig, H., and E. Rescorla, "Framework for Establishing a Secure Real-time Transport Protocol (SRTP) Security Context Using Datagram Transport Layer Security (DTLS)", RFC 5763, DOI 10.17487/RFC5763, May 2010, <<https://www.rfc-editor.org/info/rfc5763>>.
- [RFC5764] McGrew, D. and E. Rescorla, "Datagram Transport Layer Security (DTLS) Extension to Establish Keys for the Secure Real-time Transport Protocol (SRTP)", RFC 5764, DOI 10.17487/RFC5764, May 2010, <<https://www.rfc-editor.org/info/rfc5764>>.
- [RFC5869] Krawczyk, H. and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", RFC 5869, DOI 10.17487/RFC5869, May 2010, <<https://www.rfc-editor.org/info/rfc5869>>.
- [RFC5925] Touch, J., Mankin, A., and R. Bonica, "The TCP Authentication Option", RFC 5925, DOI 10.17487/RFC5925, June 2010, <<https://www.rfc-editor.org/info/rfc5925>>.
- [RFC6066] Eastlake 3rd, D., "Transport Layer Security (TLS) Extensions: Extension Definitions", RFC 6066, DOI 10.17487/RFC6066, January 2011, <<https://www.rfc-editor.org/info/rfc6066>>.
- [RFC6189] Zimmermann, P., Johnston, A., Ed., and J. Callas, "ZRTP: Media Path Key Agreement for Unicast Secure RTP", RFC 6189, DOI 10.17487/RFC6189, April 2011, <<https://www.rfc-editor.org/info/rfc6189>>.
- [RFC6347] Rescorla, E. and N. Modadugu, "Datagram Transport Layer Security Version 1.2", RFC 6347, DOI 10.17487/RFC6347, January 2012, <<https://www.rfc-editor.org/info/rfc6347>>.

- [RFC7250] Wouters, P., Ed., Tschofenig, H., Ed., Gilmore, J., Weiler, S., and T. Kivinen, "Using Raw Public Keys in Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)", RFC 7250, DOI 10.17487/RFC7250, June 2014, <<https://www.rfc-editor.org/info/rfc7250>>.
- [RFC7296] Kaufman, C., Hoffman, P., Nir, Y., Eronen, P., and T. Kivinen, "Internet Key Exchange Protocol Version 2 (IKEv2)", STD 79, RFC 7296, DOI 10.17487/RFC7296, October 2014, <<https://www.rfc-editor.org/info/rfc7296>>.
- [RFC7301] Friedl, S., Popov, A., Langley, A., and E. Stephan, "Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension", RFC 7301, DOI 10.17487/RFC7301, July 2014, <<https://www.rfc-editor.org/info/rfc7301>>.
- [RFC7539] Nir, Y. and A. Langley, "ChaCha20 and Poly1305 for IETF Protocols", RFC 7539, DOI 10.17487/RFC7539, May 2015, <<https://www.rfc-editor.org/info/rfc7539>>.
- [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>>.
- [RFC8229] Pauly, T., Touati, S., and R. Mantha, "TCP Encapsulation of IKE and IPsec Packets", RFC 8229, DOI 10.17487/RFC8229, August 2017, <<https://www.rfc-editor.org/info/rfc8229>>.
- [SIGMA] "SIGMA -- The 'SIGN-and-MAC' Approach to Authenticated Diffie-Hellman and Its Use in the IKE-Protocols", n.d..
- [WireGuard] "WireGuard -- Next Generation Kernel Network Tunnel", n.d..

## Authors' Addresses

Tommy Pauly  
Apple Inc.  
One Apple Park Way  
Cupertino, California 95014  
United States of America

Email: [tpauly@apple.com](mailto:tpauly@apple.com)

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

Email: [csp@csperkins.org](mailto:csp@csperkins.org)

Kyle Rose  
Akamai Technologies, Inc.  
150 Broadway  
Cambridge, MA 02144  
United States of America

Email: [krose@krose.org](mailto:krose@krose.org)

Christopher A. Wood  
Apple Inc.  
One Apple Park Way  
Cupertino, California 95014  
United States of America

Email: [cawood@apple.com](mailto:cawood@apple.com)

Network Working Group  
Internet-Draft  
Intended status: Informational  
Expires: 25 October 2020

T. Enghardt  
TU Berlin  
T. Pauly  
Apple Inc.  
C. Perkins  
University of Glasgow  
K. Rose  
Akamai Technologies, Inc.  
C.A. Wood  
Cloudflare  
23 April 2020

A Survey of the Interaction Between Security Protocols and Transport  
Services  
draft-ietf-taps-transport-security-12

Abstract

This document provides a survey of commonly used or notable network security protocols, with a focus on how they interact and integrate with applications and transport protocols. Its goal is to supplement efforts to define and catalog transport services by describing the interfaces required to add security protocols. This survey is not limited to protocols developed within the scope or context of the IETF, and those included represent a superset of features a Transport Services system may need to support.

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 25 October 2020.

## Copyright Notice

Copyright (c) 2020 IETF Trust and the persons identified as the document authors. All rights reserved.

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

## Table of Contents

1. Introduction . . . . .	3
1.1. Goals . . . . .	4
1.2. Non-Goals . . . . .	4
2. Terminology . . . . .	5
3. Transport Security Protocol Descriptions . . . . .	6
3.1. Application Payload Security Protocols . . . . .	7
3.1.1. TLS . . . . .	7
3.1.2. DTLS . . . . .	7
3.2. Application-Specific Security Protocols . . . . .	8
3.2.1. Secure RTP . . . . .	8
3.3. Transport-Layer Security Protocols . . . . .	8
3.3.1. IETF QUIC . . . . .	8
3.3.2. Google QUIC . . . . .	9
3.3.3. tcpcrypt . . . . .	9
3.3.4. MinimaLT . . . . .	9
3.3.5. CurveCP . . . . .	9
3.4. Packet Security Protocols . . . . .	9
3.4.1. IPsec . . . . .	10
3.4.2. WireGuard . . . . .	10
3.4.3. OpenVPN . . . . .	10
4. Transport Dependencies . . . . .	10
4.1. Reliable Byte-Stream Transports . . . . .	10
4.2. Unreliable Datagram Transports . . . . .	11
4.2.1. Datagram Protocols with Defined Byte-Stream Mappings . . . . .	11
4.3. Transport-Specific Dependencies . . . . .	12
5. Application Interface . . . . .	12
5.1. Pre-Connection Interfaces . . . . .	12
5.2. Connection Interfaces . . . . .	15
5.3. Post-Connection Interfaces . . . . .	16
5.4. Summary of Interfaces Exposed by Protocols . . . . .	17
6. IANA Considerations . . . . .	18

7. Security Considerations . . . . .	18
8. Privacy Considerations . . . . .	19
9. Acknowledgments . . . . .	19
10. Informative References . . . . .	19
Authors' Addresses . . . . .	22

## 1. Introduction

Services and features provided by transport protocols have been cataloged in [RFC8095]. This document supplements that work by surveying commonly used and notable network security protocols, and identifying the interfaces between these protocols and both transport protocols and applications. It examines Transport Layer Security (TLS), Datagram Transport Layer Security (DTLS), IETF QUIC, Google QUIC (gQUIC), tcpcrypt, Internet Protocol Security (IPsec), Secure Real-time Transport Protocol (SRTP) with DTLS, WireGuard, CurveCP, and MinimalLT. For each protocol, this document provides a brief description. Then, it describes the interfaces between these protocols and transports in Section 4 and the interfaces between these protocols and applications in Section 5.

A Transport Services system exposes an interface for applications to access various (secure) transport protocol features. The security protocols included in this survey represent a superset of functionality and features a Transport Services system may need to support, both internally and externally (via an API) for applications [I-D.ietf-taps-arch]. Ubiquitous IETF protocols such as (D)TLS, as well as non-standard protocols such as gQUIC, are included despite overlapping features. As such, this survey is not limited to protocols developed within the scope or context of the IETF. Outside of this candidate set, protocols that do not offer new features are omitted. For example, newer protocols such as WireGuard make unique design choices that have implications for and limitations on application usage. In contrast, protocols such as SSH [RFC4253], GRE [RFC2890], L2TP [RFC5641], and ALTS [ALTS] are omitted since they do not provide interfaces deemed unique.

Authentication-only protocols such as TCP-AO [RFC5925] and IPsec Authentication Header (AH) [RFC4302] are excluded from this survey. TCP-AO adds authentication to long-lived TCP connections, e.g., replay protection with per-packet Message Authentication Codes. (TCP-AO obsoletes TCP MD5 "signature" options specified in [RFC2385].) One primary use case of TCP-AO is for protecting BGP connections. Similarly, AH adds per-datagram authentication and integrity, along with replay protection. Despite these improvements, neither protocol sees general use and both lack critical properties important for emergent transport security protocols, such as confidentiality and privacy protections. Such protocols are thus omitted from this survey.

This document only surveys point-to-point protocols; multicast protocols are out of scope.

### 1.1. Goals

This survey is intended to help identify the most common interface surfaces between security protocols and transport protocols, and between security protocols and applications.

One of the goals of the Transport Services effort is to define a common interface for using transport protocols that allows software using transport protocols to easily adopt new protocols that provide similar feature-sets. The survey of the dependencies security protocols have upon transport protocols can guide implementations in determining which transport protocols are appropriate to be able to use beneath a given security protocol. For example, a security protocol that expects to run over a reliable stream of bytes, like TLS, restricts the set of transport protocols that can be used to those that offer a reliable stream of bytes.

Defining the common interfaces that security protocols provide to applications also allows interfaces to be designed in a way that common functionality can use the same APIs. For example, many security protocols that provide authentication let the application be involved in peer identity validation. Any interface to use a secure transport protocol stack thus needs to allow applications to perform this action during connection establishment.

### 1.2. Non-Goals

While this survey provides similar analysis to that which was performed for transport protocols in [RFC8095], it is important to distinguish that the use of security protocols requires more consideration.



It is not a goal to allow software implementations to automatically switch between different security protocols, even where their interfaces to transport and applications are equivalent. Even between versions, security protocols have subtly different guarantees and vulnerabilities. Thus, any implementation needs to only use the set of protocols and algorithms that are requested by applications or by a system policy.

Different security protocols also can use incompatible notions of peer identity and authentication, and cryptographic options. It is not a goal to identify a common set of representations for these concepts.

The protocols surveyed in this document represent a superset of functionality and features a Transport Services system may need to support. It does not list all transport protocols that a Transport Services system may need to implement, nor does it mandate that a Transport Service system implement any particular protocol.

A Transport Services system may implement any secure transport protocol that provides the described features. In doing so, it may need to expose an interface to the application to configure these features.

## 2. Terminology

The following terms are used throughout this document to describe the roles and interactions of transport security protocols (some of which are also defined in [RFC8095]):

- \* **Transport Feature:** a specific end-to-end feature that the transport layer provides to an application. Examples include confidentiality, reliable delivery, ordered delivery, and message-versus-stream orientation.
- \* **Transport Service:** a set of Transport Features, without an association to any given framing protocol, which provides functionality to an application.
- \* **Transport Services system:** a software component that exposes an interface to different Transport Services to an application.
- \* **Transport Protocol:** an implementation that provides one or more different transport services using a specific framing and header format on the wire. A Transport Protocol services an application, whether directly or in conjunction with a security protocol.

- \* **Application:** an entity that uses a transport protocol for end-to-end delivery of data across the network. This may also be an upper layer protocol or tunnel encapsulation.
- \* **Security Protocol:** a defined network protocol that implements one or more security features, such as authentication, encryption, key generation, session resumption, and privacy. Security protocols may be used alongside transport protocols, and in combination with other security protocols when appropriate.
- \* **Handshake Protocol:** a protocol that enables peers to validate each other and to securely establish shared cryptographic context.
- \* **Record:** Framed protocol messages.
- \* **Record Protocol:** a security protocol that allows data to be divided into manageable blocks and protected using shared cryptographic context.
- \* **Session:** an ephemeral security association between applications.
- \* **Connection:** the shared state of two or more endpoints that persists across messages that are transmitted between these endpoints. A connection is a transient participant of a session, and a session generally lasts between connection instances.
- \* **Peer:** an endpoint application party to a session.
- \* **Client:** the peer responsible for initiating a session.
- \* **Server:** the peer responsible for responding to a session initiation.

### 3. Transport Security Protocol Descriptions

This section contains brief transport and security descriptions of various security protocols currently used to protect data being sent over a network. These protocols are grouped based on where in the protocol stack they are implemented, which influences which parts of a packet they protect: Generic application payload, application payload for specific application-layer protocols, both application payload and transport headers, or entire IP packets.

Note that not all security protocols can be easily categorized, e.g., as some protocols can be used in different ways or in combination with other protocols. One major reason for this is that channel security protocols often consist of two components:

- \* A handshake protocol, which is responsible for negotiating parameters, authenticating the endpoints, and establishing shared keys.
- \* A record protocol, which is used to encrypt traffic using keys and parameters provided by the handshake protocol.

For some protocols, such as tcpcrypt, these two components are tightly integrated. In contrast, for IPsec, these components are implemented in separate protocols: AH and ESP are record protocols, which can use keys supplied by the handshake protocol IKEv2, by other handshake protocols, or by manual configuration. Moreover, some protocols can be used in different ways: While the base TLS protocol as defined in [RFC8446] has an integrated handshake and record protocol, TLS or DTLS can also be used to negotiate keys for other protocols, as in DTLS-SRTP, or the handshake protocol can be used with a separate record layer, as in QUIC [I-D.ietf-quic-transport].

### 3.1. Application Payload Security Protocols

The following protocols provide security that protects application payloads sent over a transport. They do not specifically protect any headers used for transport-layer functionality.

#### 3.1.1. TLS

TLS (Transport Layer Security) [RFC8446] is a common protocol used to establish a secure session between two endpoints. Communication over this session "prevents eavesdropping, tampering, and message forgery." TLS consists of a tightly coupled handshake and record protocol. The handshake protocol is used to authenticate peers, negotiate protocol options, such as cryptographic algorithms, and derive session-specific keying material. The record protocol is used to marshal and, once the handshake has sufficiently progressed, encrypt, data from one peer to the other. This data may contain handshake messages or raw application data.

#### 3.1.2. DTLS

DTLS (Datagram Transport Layer Security) [RFC6347] [I-D.ietf-tls-dtls13] is based on TLS, but differs in that it is designed to run over unreliable datagram protocols like UDP instead of TCP. DTLS modifies the protocol to make sure it can still provide equivalent security guarantees to TLS with the exception of order protection/non-replayability. DTLS was designed to be as similar to TLS as possible, so this document assumes that all properties from TLS are carried over except where specified.

### 3.2. Application-Specific Security Protocols

The following protocols provide application-specific security by protecting application payloads used for specific use-cases. Unlike the protocols above, these are not intended for generic application use.

#### 3.2.1. Secure RTP

Secure RTP (SRTP) is a profile for RTP that provides confidentiality, message authentication, and replay protection for RTP data packets and RTP control protocol (RTCP) packets [RFC3711]. SRTP provides a record layer only, and requires a separate handshake protocol to provide key agreement and identity management.

The commonly used handshake protocol for SRTP is DTLS, in the form of DTLS-SRTP [RFC5764]. This is an extension to DTLS that negotiates the use of SRTP as the record layer, and describes how to export keys for use with SRTP.

ZRTP [RFC6189] is an alternative key agreement and identity management protocol for SRTP. ZRTP Key agreement is performed using a Diffie-Hellman key exchange that runs on the media path. This generates a shared secret that is then used to generate the master key and salt for SRTP.

### 3.3. Transport-Layer Security Protocols

The following security protocols provide protection for both application payloads and headers that are used for transport services.

#### 3.3.1. IETF QUIC

QUIC is a new standards-track transport protocol that runs over UDP, loosely based on Google's original proprietary gQUIC protocol [I-D.ietf-quic-transport] (See Section 3.3.2 for more details). The QUIC transport layer itself provides support for data confidentiality and integrity. This requires keys to be derived with a separate handshake protocol. A mapping for QUIC of TLS 1.3 [I-D.ietf-quic-tls] has been specified to provide this handshake.

### 3.3.2. Google QUIC

Google QUIC (gQUIC) is a UDP-based multiplexed streaming protocol designed and deployed by Google following experience from deploying SPDY, the proprietary predecessor to HTTP/2. gQUIC was originally known as "QUIC": this document uses gQUIC to unambiguously distinguish it from the standards-track IETF QUIC. The proprietary technical forebear of IETF QUIC, gQUIC was originally designed with tightly-integrated security and application data transport protocols.

### 3.3.3. tcpcrypt

Tcpcrypt [RFC8548] is a lightweight extension to the TCP protocol for opportunistic encryption. Applications may use tcpcrypt's unique session ID for further application-level authentication. Absent this authentication, tcpcrypt is vulnerable to active attacks.

### 3.3.4. MinimalT

MinimalT [MinimalT] is a UDP-based transport security protocol designed to offer confidentiality, mutual authentication, DoS prevention, and connection mobility. One major goal of the protocol is to leverage existing protocols to obtain server-side configuration information used to more quickly bootstrap a connection. MinimalT uses a variant of TCP's congestion control algorithm.

### 3.3.5. CurveCP

CurveCP [CurveCP] is a UDP-based transport security that, unlike many other security protocols, is based entirely upon public key algorithms. CurveCP provides its own reliability for application data as part of its protocol.

## 3.4. Packet Security Protocols

The following protocols provide protection for IP packets. These are generally used as tunnels, such as for Virtual Private Networks (VPNs). Often, applications will not interact directly with these protocols. However, applications that implement tunnels will interact directly with these protocols.

#### 3.4.1. IPsec

IKEv2 [RFC7296] and ESP [RFC4303] together form the modern IPsec protocol suite that encrypts and authenticates IP packets, either for creating tunnels (tunnel-mode) or for direct transport connections (transport-mode). This suite of protocols separates out the key generation protocol (IKEv2) from the transport encryption protocol (ESP). Each protocol can be used independently, but this document considers them together, since that is the most common pattern.

#### 3.4.2. WireGuard

WireGuard [WireGuard] is an IP-layer protocol designed as an alternative to IPsec for certain use cases. It uses UDP to encapsulate IP datagrams between peers. Unlike most transport security protocols, which rely on Public Key Infrastructure (PKI) for peer authentication, WireGuard authenticates peers using pre-shared public keys delivered out-of-band, each of which is bound to one or more IP addresses. Moreover, as a protocol suited for VPNs, WireGuard offers no extensibility, negotiation, or cryptographic agility.

#### 3.4.3. OpenVPN

OpenVPN [OpenVPN] is a commonly used protocol designed as an alternative to IPsec. A major goal of this protocol is to provide a VPN that is simple to configure and works over a variety of transports. OpenVPN encapsulates either IP packets or Ethernet frames within a secure tunnel and can run over either UDP or TCP. For key establishment, OpenVPN can either use TLS as a handshake protocol or use pre-shared keys.

### 4. Transport Dependencies

Across the different security protocols listed above, the primary dependency on transport protocols is the presentation of data: either an unbounded stream of bytes, or framed messages. Within protocols that rely on the transport for message framing, most are built to run over transports that inherently provide framing, like UDP, but some also define how their messages can be framed over byte-stream transports.

#### 4.1. Reliable Byte-Stream Transports

The following protocols all depend upon running on a transport protocol that provides a reliable, in-order stream of bytes. This is typically TCP.

Application Payload Security Protocols:

- \* TLS

Transport-Layer Security Protocols:

- \* tcpcrypt

#### 4.2. Unreliable Datagram Transports

The following protocols all depend on the transport protocol to provide message framing to encapsulate their data. These protocols are built to run using UDP, and thus do not have any requirement for reliability. Running these protocols over a protocol that does provide reliability will not break functionality, but may lead to multiple layers of reliability if the security protocol is encapsulating other transport protocol traffic.

Application Payload Security Protocols:

- \* DTLS
- \* ZRTP
- \* SRTP

Transport-Layer Security Protocols:

- \* QUIC
- \* MinimaLT
- \* CurveCP

Packet Security Protocols:

- \* IPsec
- \* WireGuard
- \* OpenVPN

##### 4.2.1. Datagram Protocols with Defined Byte-Stream Mappings

Of the protocols listed above that depend on the transport for message framing, some do have well-defined mappings for sending their messages over byte-stream transports like TCP.

#### Application Payload Security Protocols:

- \* DTLS when used as a handshake protocol for SRTP [RFC7850]
- \* ZRTP [RFC6189]
- \* SRTP [RFC4571][RFC3711]

#### Packet Security Protocols:

- \* IPsec [RFC8229]

### 4.3. Transport-Specific Dependencies

One protocol surveyed, `tcpcrypt`, has an direct dependency on a feature in the transport that is needed for its functionality. Specifically, `tcpcrypt` is designed to run on top of TCP, and uses the TCP Encryption Negotiation Option (ENO) [RFC8547] to negotiate its protocol support.

QUIC, CurveCP, and MinimaLT provide both transport functionality and security functionality. They depend on running over a framed protocol like UDP, but they add their own layers of reliability and other transport services. Thus, an application that uses one of these protocols cannot decouple the security from transport functionality.

## 5. Application Interface

This section describes the interface exposed by the security protocols described above. We partition these interfaces into pre-connection (configuration), connection, and post-connection interfaces, following conventions in [I-D.ietf-taps-interface] and [I-D.ietf-taps-arch].

Note that not all protocols support each interface. The table in Section 5.4 summarizes which protocol exposes which of the interfaces. In the following sections, we provide abbreviations of the interface names to use in the summary table.

### 5.1. Pre-Connection Interfaces

Configuration interfaces are used to configure the security protocols before a handshake begins or keys are negotiated.



- \* Identities and Private Keys (IPK): The application can provide its identity, credentials (e.g., certificates), and private keys, or mechanisms to access these, to the security protocol to use during handshakes.
  - TLS
  - DTLS
  - ZRTP
  - QUIC
  - MinimaLT
  - CurveCP
  - IPsec
  - WireGuard
  - OpenVPN
- \* Supported Algorithms (Key Exchange, Signatures, and Ciphersuites) (ALG): The application can choose the algorithms that are supported for key exchange, signatures, and ciphersuites.
  - TLS
  - DTLS
  - ZRTP
  - QUIC
  - tcpcrypt
  - MinimaLT
  - IPsec
  - OpenVPN
- \* Extensions (EXT): The application enables or configures extensions that are to be negotiated by the security protocol, such as Application-Layer Protocol Negotiation (ALPN) [RFC7301].
  - TLS

- DTLS
- QUIC
- \* Session Cache Management (CM): The application provides the ability to save and retrieve session state (such as tickets, keying material, and server parameters) that may be used to resume the security session.
  - TLS
  - DTLS
  - ZRTP
  - QUIC
  - tcpcrypt
  - MinimaLT
- \* Authentication Delegation (AD): The application provides access to a separate module that will provide authentication, using Extensible Authentication Protocol (EAP) [RFC3748] for example.
  - IPsec
  - tcpcrypt
- \* Pre-Shared Key Import (PSKI): Either the handshake protocol or the application directly can supply pre-shared keys for use in encrypting (and authenticating) communication with a peer.
  - TLS
  - DTLS
  - ZRTP
  - QUIC
  - tcpcrypt
  - MinimaLT
  - IPsec
  - WireGuard

- OpenVPN

## 5.2. Connection Interfaces

- \* Identity Validation (IV): During a handshake, the security protocol will conduct identity validation of the peer. This can offload validation or occur transparently to the application.
  - TLS
  - DTLS
  - ZRTP
  - QUIC
  - MinimalLT
  - CurveCP
  - IPsec
  - WireGuard
  - OpenVPN
- \* Source Address Validation (SAV): The handshake protocol may interact with the transport protocol or application to validate the address of the remote peer that has sent data. This involves sending a cookie exchange to avoid DoS attacks. (This list omits protocols which depend on TCP and therefore implicitly perform SAV.)
  - DTLS
  - QUIC
  - IPsec
  - WireGuard

### 5.3. Post-Connection Interfaces

- \* Connection Termination (CT): The security protocol may be instructed to tear down its connection and session information. This is needed by some protocols, e.g., to prevent application data truncation attacks in which an attacker terminates an underlying insecure connection-oriented protocol to terminate the session.
  - TLS
  - DTLS
  - ZRTP
  - QUIC
  - tcpcrypt
  - MinimaLT
  - IPsec
  - OpenVPN
- \* Key Update (KU): The handshake protocol may be instructed to update its keying material, either by the application directly or by the record protocol sending a key expiration event.
  - TLS
  - DTLS
  - QUIC
  - tcpcrypt
  - MinimaLT
  - IPsec
- \* Shared Secret Export (PSKE): The handshake protocol may provide an interface for producing shared secrets for application-specific uses.
  - TLS
  - DTLS

- tcpcrypt
- IPsec
- OpenVPN
- MinimaLT
- \* Key Expiration (KE): The record protocol can signal that its keys are expiring due to reaching a time-based deadline, or a use-based deadline (number of bytes that have been encrypted with the key). This interaction is often limited to signaling between the record layer and the handshake layer.
  - IPsec
- \* Mobility Events (ME): The record protocol can be signaled that it is being migrated to another transport or interface due to connection mobility, which may reset address and state validation and induce state changes such as use of a new Connection Identifier (CID).
  - DTLS (version 1.3 only [I-D.ietf-tls-dtls13])
  - QUIC
  - MinimaLT
  - CurveCP
  - IPsec [RFC4555]
  - WireGuard

#### 5.4. Summary of Interfaces Exposed by Protocols

The following table summarizes which protocol exposes which interface.

Protocol	IPK	ALG	EXT	CM	AD	PSKI	IV	SAV	CT	KU	PSKE	KE	ME
TLS	x	x	x	x		x	x		x	x	x		
DTLS	x	x	x	x		x	x	x	x	x	x		x
ZRTP	x	x		x		x	x		x				
QUIC	x	x	x	x		x	x	x	x	x			x
tcpcrypt		x		x	x	x			x	x	x		
MinimalT	x	x		x		x	x		x	x	x		x
CurveCP	x						x						x
IPsec	x	x			x	x	x	x	x	x	x	x	x
WireGuard	x					x	x	x					x
OpenVPN	x	x				x	x		x		x		

Table 1

x=Interface is exposed (blank)=Interface is not exposed

## 6. IANA Considerations

This document has no request to IANA.

## 7. Security Considerations

This document summarizes existing transport security protocols and their interfaces. It does not propose changes to or recommend usage of reference protocols. Moreover, no claims of security and privacy properties beyond those guaranteed by the protocols discussed are made. For example, metadata leakage via timing side channels and traffic analysis may compromise any protocol discussed in this survey. Applications using Security Interfaces should take such limitations into consideration when using a particular protocol implementation.

## 8. Privacy Considerations

Analysis of how features improve or degrade privacy is intentionally omitted from this survey. All security protocols surveyed generally improve privacy by using encryption to reduce information leakage. However, varying amounts of metadata remain in the clear across each protocol. For example, client and server certificates are sent in cleartext in TLS 1.2 [RFC5246], whereas they are encrypted in TLS 1.3 [RFC8446]. A survey of privacy features, or lack thereof, for various security protocols could be addressed in a separate document.

## 9. Acknowledgments

The authors would like to thank Bob Bradley, Frederic Jacobs, Mirja Kuehlewind, Yannick Sierra, Brian Trammell, and Magnus Westerlund for their input and feedback on this draft.

## 10. Informative References

- [ALTS] Ghali, C., Stubblefield, A., Knapp, E., Li, J., Schmidt, B., and J. Boeuf, "Application Layer Transport Security", <<https://cloud.google.com/security/encryption-in-transit/application-layer-transport-security/>>.
- [CurveCP] Bernstein, D.J., "CurveCP -- Usable security for the Internet", <<http://curvecp.org>>.
- [I-D.ietf-quic-tls] Thomson, M. and S. Turner, "Using TLS to Secure QUIC", Work in Progress, Internet-Draft, draft-ietf-quic-tls-27, 21 February 2020, <<http://www.ietf.org/internet-drafts/draft-ietf-quic-tls-27.txt>>.
- [I-D.ietf-quic-transport] Iyengar, J. and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport", Work in Progress, Internet-Draft, draft-ietf-quic-transport-27, 21 February 2020, <<http://www.ietf.org/internet-drafts/draft-ietf-quic-transport-27.txt>>.
- [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", Work in Progress, Internet-Draft, draft-ietf-taps-arch-07, 9 March 2020, <<http://www.ietf.org/internet-drafts/draft-ietf-taps-arch-07.txt>>.

- [I-D.ietf-taps-interface]  
Trammell, B., Welzl, M., Enghardt, T., Fairhurst, G., Kuehlewind, M., Perkins, C., Tiesel, P., Wood, C., and T. Pauly, "An Abstract Application Layer Interface to Transport Services", Work in Progress, Internet-Draft, draft-ietf-taps-interface-06, 9 March 2020, <<http://www.ietf.org/internet-drafts/draft-ietf-taps-interface-06.txt>>.
- [I-D.ietf-tls-dtls13]  
Rescorla, E., Tschofenig, H., and N. Modadugu, "The Datagram Transport Layer Security (DTLS) Protocol Version 1.3", Work in Progress, Internet-Draft, draft-ietf-tls-dtls13-37, 9 March 2020, <<http://www.ietf.org/internet-drafts/draft-ietf-tls-dtls13-37.txt>>.
- [MinimalT] Petullo, W.M., Zhang, X., Solworth, J.A., Bernstein, D.J., and T. Lange, "MinimalT -- Minimal-latency Networking Through Better Security", <<http://dl.acm.org/citation.cfm?id=2516737>>.
- [OpenVPN] "OpenVPN cryptographic layer", <<https://openvpn.net/community-resources/openvpn-cryptographic-layer/>>.
- [RFC2385] Heffernan, A., "Protection of BGP Sessions via the TCP MD5 Signature Option", RFC 2385, DOI 10.17487/RFC2385, August 1998, <<https://www.rfc-editor.org/info/rfc2385>>.
- [RFC2890] Dommetty, G., "Key and Sequence Number Extensions to GRE", RFC 2890, DOI 10.17487/RFC2890, September 2000, <<https://www.rfc-editor.org/info/rfc2890>>.
- [RFC3711] Baugher, M., McGrew, D., Naslund, M., Carrara, E., and K. Norrman, "The Secure Real-time Transport Protocol (SRTP)", RFC 3711, DOI 10.17487/RFC3711, March 2004, <<https://www.rfc-editor.org/info/rfc3711>>.
- [RFC3748] Aboba, B., Blunk, L., Vollbrecht, J., Carlson, J., and H. Levkowetz, Ed., "Extensible Authentication Protocol (EAP)", RFC 3748, DOI 10.17487/RFC3748, June 2004, <<https://www.rfc-editor.org/info/rfc3748>>.
- [RFC4253] Ylonen, T. and C. Lonvick, Ed., "The Secure Shell (SSH) Transport Layer Protocol", RFC 4253, DOI 10.17487/RFC4253, January 2006, <<https://www.rfc-editor.org/info/rfc4253>>.



- [RFC4302] Kent, S., "IP Authentication Header", RFC 4302, DOI 10.17487/RFC4302, December 2005, <<https://www.rfc-editor.org/info/rfc4302>>.
- [RFC4303] Kent, S., "IP Encapsulating Security Payload (ESP)", RFC 4303, DOI 10.17487/RFC4303, December 2005, <<https://www.rfc-editor.org/info/rfc4303>>.
- [RFC4555] Eronen, P., "IKEv2 Mobility and Multihoming Protocol (MOBIKE)", RFC 4555, DOI 10.17487/RFC4555, June 2006, <<https://www.rfc-editor.org/info/rfc4555>>.
- [RFC4571] Lazzaro, J., "Framing Real-time Transport Protocol (RTP) and RTP Control Protocol (RTCP) Packets over Connection-Oriented Transport", RFC 4571, DOI 10.17487/RFC4571, July 2006, <<https://www.rfc-editor.org/info/rfc4571>>.
- [RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, DOI 10.17487/RFC5246, August 2008, <<https://www.rfc-editor.org/info/rfc5246>>.
- [RFC5641] McGill, N. and C. Pignataro, "Layer 2 Tunneling Protocol Version 3 (L2TPv3) Extended Circuit Status Values", RFC 5641, DOI 10.17487/RFC5641, August 2009, <<https://www.rfc-editor.org/info/rfc5641>>.
- [RFC5764] McGrew, D. and E. Rescorla, "Datagram Transport Layer Security (DTLS) Extension to Establish Keys for the Secure Real-time Transport Protocol (SRTP)", RFC 5764, DOI 10.17487/RFC5764, May 2010, <<https://www.rfc-editor.org/info/rfc5764>>.
- [RFC5925] Touch, J., Mankin, A., and R. Bonica, "The TCP Authentication Option", RFC 5925, DOI 10.17487/RFC5925, June 2010, <<https://www.rfc-editor.org/info/rfc5925>>.
- [RFC6189] Zimmermann, P., Johnston, A., Ed., and J. Callas, "ZRTP: Media Path Key Agreement for Unicast Secure RTP", RFC 6189, DOI 10.17487/RFC6189, April 2011, <<https://www.rfc-editor.org/info/rfc6189>>.
- [RFC6347] Rescorla, E. and N. Modadugu, "Datagram Transport Layer Security Version 1.2", RFC 6347, DOI 10.17487/RFC6347, January 2012, <<https://www.rfc-editor.org/info/rfc6347>>.

- [RFC7296] Kaufman, C., Hoffman, P., Nir, Y., Eronen, P., and T. Kivinen, "Internet Key Exchange Protocol Version 2 (IKEv2)", STD 79, RFC 7296, DOI 10.17487/RFC7296, October 2014, <<https://www.rfc-editor.org/info/rfc7296>>.
- [RFC7301] Friedl, S., Popov, A., Langley, A., and E. Stephan, "Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension", RFC 7301, DOI 10.17487/RFC7301, July 2014, <<https://www.rfc-editor.org/info/rfc7301>>.
- [RFC7850] Nandakumar, S., "Registering Values of the SDP 'proto' Field for Transporting RTP Media over TCP under Various RTP Profiles", RFC 7850, DOI 10.17487/RFC7850, April 2016, <<https://www.rfc-editor.org/info/rfc7850>>.
- [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>>.
- [RFC8229] Pauly, T., Touati, S., and R. Mantha, "TCP Encapsulation of IKE and IPsec Packets", RFC 8229, DOI 10.17487/RFC8229, August 2017, <<https://www.rfc-editor.org/info/rfc8229>>.
- [RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <<https://www.rfc-editor.org/info/rfc8446>>.
- [RFC8547] Bittau, A., Giffin, D., Handley, M., Mazieres, D., and E. Smith, "TCP-ENO: Encryption Negotiation Option", RFC 8547, DOI 10.17487/RFC8547, May 2019, <<https://www.rfc-editor.org/info/rfc8547>>.
- [RFC8548] Bittau, A., Giffin, D., Handley, M., Mazieres, D., Slack, Q., and E. Smith, "Cryptographic Protection of TCP Streams (tcpcrypt)", RFC 8548, DOI 10.17487/RFC8548, May 2019, <<https://www.rfc-editor.org/info/rfc8548>>.
- [WireGuard] Donenfeld, J.A., "WireGuard -- Next Generation Kernel Network Tunnel", <<https://www.wireguard.com/papers/wireguard.pdf>>.

Authors' Addresses

Theresa Enghardt  
TU Berlin  
Marchstr. 23  
10587 Berlin  
Germany

Email: [ietf@tenghardt.net](mailto:ietf@tenghardt.net)

Tommy Pauly  
Apple Inc.  
One Apple Park Way  
Cupertino, California 95014,  
United States of America

Email: [tpauly@apple.com](mailto:tpauly@apple.com)

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

Email: [csp@csp Perkins.org](mailto:csp@csp Perkins.org)

Kyle Rose  
Akamai Technologies, Inc.  
150 Broadway  
Cambridge, MA 02144,  
United States of America

Email: [krose@krose.org](mailto:krose@krose.org)

Christopher A. Wood  
Cloudflare  
101 Townsend St  
San Francisco,  
United States of America

Email: [caw@heapingbits.net](mailto:caw@heapingbits.net)