

TAPS Working Group
Internet-Draft
Intended status: Standards Track
Expires: May 7, 2020

T. Pauly, Ed.
Apple Inc.
B. Trammell, Ed.
Google
A. Brunstrom
Karlstad University
G. Fairhurst
University of Aberdeen
C. Perkins
University of Glasgow
P. Tiesel
TU Berlin
C. Wood
Apple Inc.
November 04, 2019

An Architecture for Transport Services
draft-ietf-taps-arch-05

Abstract

This document provides an overview of the architecture of Transport Services, a model for exposing transport protocol features to applications for network communication. In contrast to what is provided by most existing Application Programming Interfaces (APIs), Transport Services is based on an asynchronous, event-driven interaction pattern; it uses messages for representing data transfer to applications; and it assumes an implementation that can use multiple IP addresses, multiple protocols, and multiple paths, and provide multiple application streams. This document further defines the common set of terminology and concepts to be used in definitions of Transport Services APIs and implementations.

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 May 7, 2020.

Copyright Notice

Copyright (c) 2019 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. Background	3
1.2. Overview	4
1.3. Specification of Requirements	5
2. API Model	5
2.1. Event-Driven API	6
2.2. Data Transfer Using Messages	7
2.3. Flexible Implementation	8
3. Design Principles	8
3.1. Common APIs for Common Features	9
3.2. Access to Specialized Features	9
3.3. Scope for API and Implementation Definitions	10
4. Transport Services Architecture and Concepts	11
4.1. Transport Services API Concepts	12
4.1.1. Connection Objects	14
4.1.2. Pre-Establishment	15
4.1.3. Establishment Actions	16
4.1.4. Data Transfer Objects and Actions	17
4.1.5. Event Handling	18
4.1.6. Termination Actions	18
4.2. Transport System Implementation Concepts	18
4.2.1. Candidate Gathering	20
4.2.2. Candidate Racing	20
4.2.3. Protocol Stack Equivalence	20
4.2.4. Separating Connection Groups	22
5. IANA Considerations	22
6. Security Considerations	23
7. Acknowledgements	23
8. References	24

8.1. Normative References	24
8.2. Informative References	24
Authors' Addresses	25

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. 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 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, without requiring changes to the applications. This flexibility enables faster deployment of new features and protocols. It can also support applications by offering racing and fallback mechanisms, which otherwise need to be implemented in each application separately.

This document is 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 [I-D.ietf-taps-minset]. 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 [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 or Optimizing Features), or else can be handled automatically by a system implementing Transport Services (referred to as Automatable Features). Among the 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:

- o Section 2 describes how the API model of Transport Services differs from traditional socket-based APIs. Specifically, it offers asynchronous event-driven interaction, the use of messages for data transfer, and the ability to easily adopt different transport protocols.
- o Section 3 explains the design principles that guide the Transport Services API. These principles are intended to make sure that transport protocols can continue to be enhanced and evolve without requiring too many changes by application developers.
- o Section 4 presents the Transport Services architecture diagram and defines the concepts that are used by both the API and implementation documents. The Preconnection allows applications to configure connection properties, and 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:

- o Applications create connections and transfer data using the socket API.
- o The socket API provides the interface to the implementations of TCP and UDP (typically implemented in the system's kernel).
- o TCP and UDP in the kernel send and receive data over the available network layer interfaces.

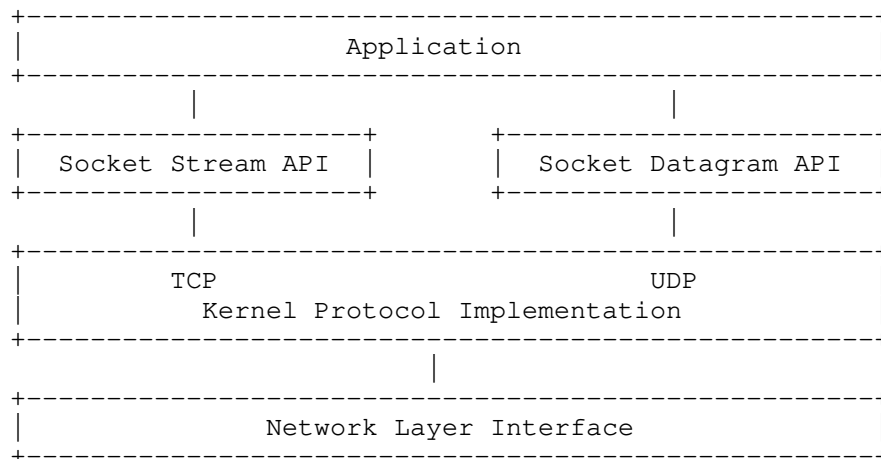


Figure 1: socket() API Model

The Transport Services architecture maintains this general model of interaction, but aims to both modernize the API surface exposed for transport protocols and enrich the capabilities of the transport system implementation.

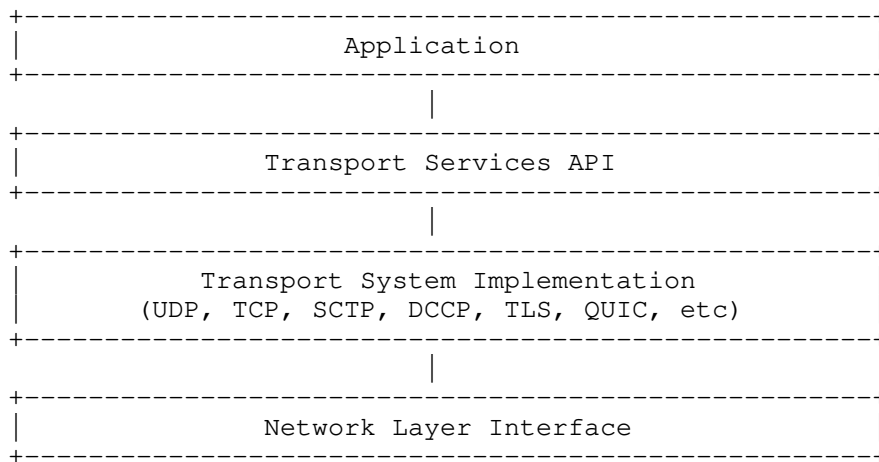


Figure 2: Transport Services API Model

The Transport Services API [I-D.ietf-taps-interface] defines the mechanism for an application to create network connections and transfer data. The implementation [I-D.ietf-taps-impl] is responsible for mapping the API to the various available transport protocols and managing the available network interfaces and paths.

There are key differences between the architecture of the Transport Services system and the architecture of the sockets API: it presents an asynchronous, event-driven API; it uses messages for representing data transfer to applications; and it assumes an implementation that can use multiple IP addresses, multiple protocols, multiple paths, and provide multiple application streams.

2.1. Event-Driven API

Originally, sockets presented a blocking interface for establishing connections and transferring data. However, most modern applications interact with the network asynchronously. When sockets are presented as an asynchronous interface, they generally use 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.

All interaction with a Transport Services system is expected to be asynchronous, and use an event-driven model unlike sockets Section 4.1.5. For example, if the application wants to read, its call to read will not fail, but will deliver an event containing the received data once it is available.

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

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

2.2. Data Transfer Using Messages

Sockets provide a message interface for datagram protocols like UDP, but provide an unstructured stream abstraction for TCP. While TCP does indeed provide the ability to send and receive data as streams, most applications need to interpret structure within these streams. For example, HTTP/1.1 uses character delimiters to segment messages over a 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. Messages seamlessly work with transport protocols that support datagrams or records, but can also be used over a stream by defining an application-layer framer Section 4.1.4. When framing protocols are placed on top of unstructured streams, the messages used in the API represent the framed messages within the stream. In the absence of a framer, protocols that deal only in byte streams, such as TCP, represent their data in each direction as a single, long message.

Providing a message-based abstraction provides many benefits, such as:

- o the ability to associate deadlines with messages, for applications that care about timing;
- o the ability to provide control of reliability, choosing which 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 the transport system could decide to not deliver a message, either following packet loss or because it has missed a deadline. In particular, this can avoid (re-)sending data that relies on a previous transmission that was never received.

- o 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, as 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.

2.3. Flexible Implementation

Sockets, for protocols like TCP, are generally limited to connecting to a single address over a single interface. They also present a single stream to the application. Software layers built upon sockets 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; and support multipath and multistreaming protocols.

Transport Services implementations are meant to be flexible at connection establishment time, considering many different options and trying to select the most optimal combinations (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.

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. 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 is intended to 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 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 API for the system remains the same over time, new protocols and features could be added to the system's implementation without requiring changes in applications for adoption.

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 ought to 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 is REQUIRED to allow access to the distilled minimal set of features offered by transport protocols [I-D.ietf-taps-minset].

3.2. 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. A Transport Services system therefore SHOULD also permit more specialized protocol features to be used. The interface for these specialized options ought to be exposed differently from the common options to ensure flexibility.

A specialized feature could 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 could 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 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 for constrained checksum usage, communication would not fail when a protocol such as TCP is selected, which uses a checksum covering the entire payload.

Other specialized features, however, could 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 a transport security function 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 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. 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 how it can be exposed to 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. All implementations are REQUIRED to offer an API that is sufficient 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. However, some features provided by this API will not be functional in certain implementations. For example, it is possible that some very constrained devices might not have a full TCP implementation beneath the API.

To preserve flexibility and compatibility with future protocols, top-level features in the Transport Services API SHOULD avoid referencing particular transport protocols. The mappings of these API features to specific implementations of each feature is explained in the [I-D.ietf-taps-impl] along with the implications of the feature on existing protocols. It is expected that [I-D.ietf-taps-interface] will be updated and supplemented as new protocols and protocol features are developed.

It is important to note that neither the Transport Services API [I-D.ietf-taps-interface] nor the Implementation document [I-D.ietf-taps-impl] define new protocols or protocol capabilities that affect what is communicated across the network. The Transport Services system MUST be deployable on one side only. A Transport Services system acting as a connection initiator can communicate with any existing system that implements the transport protocol(s) selected by the Transport Services system. Similarly, a Transport Services system acting as a listener can receive connections for any protocol that is supported by the system, from existing initiators.

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 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 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 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.

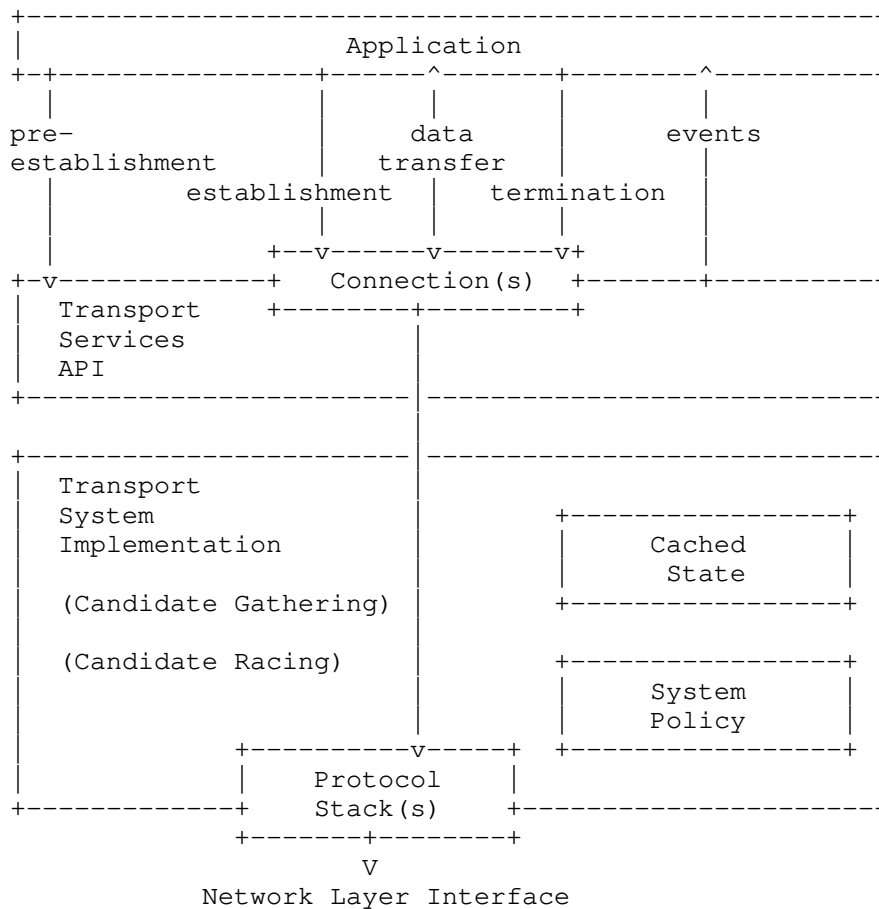


Figure 3: Concepts and Relationships in the Transport Services Architecture

4.1. Transport Services API Concepts

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

Beyond the connection objects, there are several high-level groups of actions that any Transport Services API implementing this specification **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 be defined to apply to multiple transport protocols. 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.

- o Establishment (Section 4.1.3) focuses on the actions that an application takes on the connection objects to prepare for data transfer.
- o Data Transfer (Section 4.1.4) 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.
- 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 MAY 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 and events during the lifetime of a connection. Note that some actions are alternatives (e.g., whether to initiate a connection or to listen for incoming connections), others are optional (e.g., setting Connection and Message Properties in Pre-Establishment), or have been omitted for brevity.

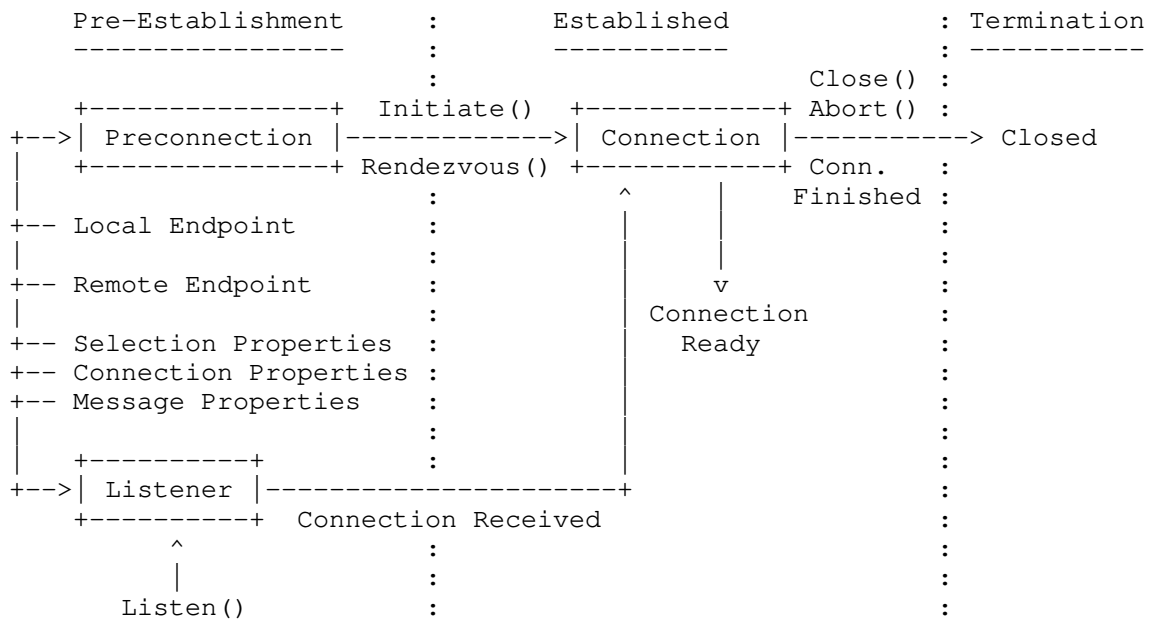


Figure 4: The lifetime of a connection

4.1.1. Connection 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 (Section 4.1.2) to which it will connect, and Selection Properties (Section 4.1.2) that influence the paths and protocols a Connection will use. A Preconnection can be fully specified such that it represents a single possible Connection, or it can be partially specified such that it represents a family of possible Connections. The Local Endpoint (Section 4.1.2) **MUST** be specified if the Preconnection is used to Listen for incoming connections. The Local Endpoint is **OPTIONAL** if it is used to Initiate connections. The Remote Endpoint **MUST** be specified in the Preconnection that is used to Initiate connections. The Remote Endpoint 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 **Transport Properties:** Transport Properties can be specified as part of a Preconnection to allow the application to configure the

Transport System and express their requirements, prohibitions, and preferences. There are three kinds of Transport Properties:

- * Selection Properties (Section 4.1.2)
 - * Connection Properties (Section 4.1.2)
 - * and Message Properties (Section 4.1.4); note that Message Properties can also be specified during data transfer to affect specific Messages.
- o Connection: A Connection object represents one or more active transport protocol instances that can send and/or receive Messages between local and remote systems. It holds state pertaining to the underlying transport protocol instances and any ongoing data transfers. 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. It can also represent a pool of transport protocol instance, e.g., a set of TCP and QUIC connections to equivalent endpoints, or a stream of a multi-streaming transport protocol instance.
 - o Listener: A Listener object accepts incoming transport protocol connections from remote systems 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 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, 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 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.
- o 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.
- o Selection Properties: The Selection Properties consist of the options that an application can set to influence the selection of

paths between the local and remote systems, to influence the selection of transport protocols, or to configure the behavior of generic transport protocol features. These options can take the form of requirements, prohibitions, or preferences. Examples of options that influence path selection include the interface type (such as a Wi-Fi Ethernet connection, or a Cellular LTE connection), requirements around the Maximum Transmission Unit (MTU) or path MTU (PMTU), or preferences for throughput and latency properties. Examples of options that influence protocol selection and configuration of transport protocol features include reliability, service class, multipath support, and fast open support.

- o **Connection Properties:** The Connection Properties are used to configure protocol-specific options and control per-connection behavior of the Transport System. For example, a protocol-specific Connection Property can express that if UDP is used, the implementation ought to use checksums. 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 MAY be used in this way by an implementation during connection establishment. Connection Properties SHOULD be specified on a Preconnection prior to Connection establishment, but MAY be modified later. Changes made to Connection Properties after establishment take effect on a best-effort basis. Such changes do not affect protocol or path selection, but only modify the manner in which a connection sends and receives data.

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 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. 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 systems 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. Boundaries of a Message might or might not be understood or transmitted by transport protocols. Specifically, what one application considers to be two Messages sent on a stream-based transport can be treated as a single Message by the application on the other side.
- o **Message Properties:** Message Properties can be used to annotate specific Messages. These properties might only apply to how 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. Message Properties MAY be set on a Preconnection to define defaults properties for sending. When receiving Messages, Message Properties can contain per-protocol properties for properties that are sent between the endpoints.
- o **Send:** The action to transmit a Message or partial Message over a Connection to the remote system. The interface to Send MAY include Message Properties specific to how the Message content is to be sent. The status of the Send operation can be delivered back to the sending 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 system, while the Message content itself will be delivered in an event (Section 4.1.5). The interface to Receive MAY include Message Properties specific to the Message that is to be delivered to the application.
- o **Framer:** A Framer is a data translation layer that can be added to a Connection to define how application-level Messages are transmitted over a transport protocol. This is particularly relevant for protocols that otherwise present unstructured streams, such as TCP.

4.1.5. Event Handling

This section provides the top-level categories of events that can be delivered to an application. This list is not exhaustive.

- 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. The event SHOULD deliver a reason or 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 might indicate a failure 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 might influence how and where data is sent and/or received.

4.1.6. Termination Actions

- o 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 system if the transport protocol allows this.
- o Abort: The action the application takes on a Connection to indicate a Close and also indicate that the transport system SHOULD NOT attempt to deliver any outstanding data.

4.2. Transport System Implementation Concepts

This section defines the set of objects 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.

- o **Connection Group:** A set of Connections that share properties and caches. For multiplexing transport protocols, only Connections within the same Connection Group are allowed to be multiplexed together. An application can explicitly define Connection Groups to control caching boundaries, as discussed in Section 4.2.4.
- o **Path:** Represents an available set of properties that a local system can use to communicate with a remote system, such as routes, addresses, and physical and virtual network interfaces.
- o **Protocol Instance:** A single instance of one protocol, including any state 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 can 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 Selection Properties and System Policy. Candidate Paths are identified during the gathering phase (Section 4.2.1) and can be used during the racing phase (Section 4.2.2).
- o **Candidate Protocol Stack:** One protocol stack that can be used by an application for a connection, of which there can be several. Candidate Protocol Stacks are identified during the gathering phase (Section 4.2.1) and are 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 either 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 Selection Properties provided by the application, the policies and heuristics of a Transport Services system.
- 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 Transport Properties provided by the application, and the heuristics or policies within the Transport Services system.

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. 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.
- o 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 IP addresses resolved from a DNS hostname.

4.2.3. Protocol Stack Equivalence

The Transport Services architecture defines a mechanism that allows applications to easily use different network paths and Protocol Stacks. In some cases, changing which Protocol Stacks or network paths are used will require updating the preferences expressed by the application that uses the Transport Services system. For example, an application can 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 by racing multiple candidate Protocol Stacks during connection establishment. This functionality in the API can be a powerful driver of new protocol adoption, but needs to 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 need to 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 system, whereas TCP does not preserve message boundaries on its own.
2. Both stacks MUST offer the transport services that are 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 reliability could be regarded as an equivalent Protocol Stack as long providing this would not conflict with any other application-requested properties.
3. Both stacks MUST offer the same security properties. The inclusion of transport security protocols [I-D.ietf-taps-transport-security] in a Protocol Stack adds additional 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 ought not to 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 transport system would 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 are considered equivalent, whereas TLS 1.2 and TLS 1.3 [RFC8446] are not considered equivalent.

4.2.4. Separating Connection Groups

By default, all stored properties of the implementation are shared within a process, such as cached protocol state, cached path state, and heuristics. This provides efficiency and convenience for the application, since the Transport System implementation can automatically optimize behavior.

There are several reasons, however, that an application might want to isolate some Connections within a single process. These reasons include:

- o Privacy concerns about re-using cached protocol state that can lead to linkability. Sensitive state may include TLS session state [RFC8446] and HTTP cookies [RFC6265].
- o 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 (for example, when Connections are multiplexed HTTP/2 or QUIC streams).
- o Performance concerns about Connections introducing head-of-line blocking due to multiplexing or needing to share state on a single thread.

The Transport Services API SHOULD allow applications to explicitly define Connection Groups that force separation of Cached State and Protocol Stacks. For example, a web browser application might use Connection Groups with separate caches for different tabs in the browser to decrease linkability.

The interface to specify these groups MAY expose fine-grained tuning for which properties and cached state is allowed to be shared with other Connections. For example, an application might want to allow sharing TCP Fast Open cookies across groups, but not TLS session state.

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 translates to 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 [I-D.ietf-taps-transport-security].

As described above in Section 4.2.3, if a Transport Services system races between two different Protocol Stacks, both MUST use the same security protocols and options.

Clients need to ensure that security APIs are used appropriately. In cases where clients 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, clients ought not to use PSK material created for the Encapsulating Security Protocol (ESP, part of IPsec) [RFC4303] with QUIC, and clients ought not to use private keys intended for server authentication as a keys for client authentication.

Moreover, Transport Services systems MUST NOT automatically fall back from secure protocols to insecure protocols, or to weaker versions of secure protocols. For example, if a client requests TLS, but the desired version of TLS is not available, its connection will fail. Clients are thus responsible for implementing security protocol fallback or version fallback by creating multiple Transport Services Connections, if so desired.

7. 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.

8. References

8.1. Normative References

- [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", draft-ietf-taps-interface-04 (work in progress), July 2019.
- [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>>.

8.2. 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-04 (work in progress), July 2019.
- [I-D.ietf-taps-minset]
Welzl, M. and S. Gjessing, "A Minimal Set of Transport Services for End Systems", draft-ietf-taps-minset-11 (work in progress), September 2018.
- [I-D.ietf-taps-transport-security]
Wood, C., Enghardt, T., Pauly, T., Perkins, C., and K. Rose, "A Survey of Transport Security Protocols", draft-ietf-taps-transport-security-09 (work in progress), September 2019.
- [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>>.

- [RFC4303] Kent, S., "IP Encapsulating Security Payload (ESP)", RFC 4303, DOI 10.17487/RFC4303, December 2005, <<https://www.rfc-editor.org/info/rfc4303>>.
- [RFC6265] Barth, A., "HTTP State Management Mechanism", RFC 6265, DOI 10.17487/RFC6265, April 2011, <<https://www.rfc-editor.org/info/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/info/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/info/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/info/rfc8095>>.
- [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>>.

Authors' Addresses

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

Email: tpauly@apple.com

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

Email: ietf@trammell.ch

Anna Brunstrom
Karlstad University
Universitetsgatan 2
651 88 Karlstad
Sweden

Email: anna.brunstrom@kau.se

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

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

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

Email: csp@csp Perkins.org

Philipp S. Tiesel
TU Berlin
Einsteinufer 25
10587 Berlin
Germany

Email: philipp@tiesel.net

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: Informational
Expires: May 7, 2020

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
November 04, 2019

Implementing Interfaces to Transport Services
draft-ietf-taps-impl-05

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 May 7, 2020.

Copyright Notice

Copyright (c) 2019 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 Connection Objects	4
3. Implementing Pre-Establishment	4
3.1. Configuration-time errors	5
3.2. Role of system policy	6
4. Implementing Connection Establishment	6
4.1. Candidate Gathering	8
4.1.1. Gathering Endpoint Candidates	8
4.1.2. Structuring Options as a Tree	9
4.1.3. Branch Types	11
4.2. Branching Order-of-Operations	13
4.3. Sorting Branches	14
4.4. Candidate Racing	15
4.4.1. Delayed	16
4.4.2. Failover	17
4.5. Completing Establishment	17
4.5.1. Determining Successful Establishment	18
4.6. Establishing multiplexed connections	18
4.7. Handling racing with "unconnected" protocols	19
4.8. Implementing listeners	19
4.8.1. Implementing listeners for Connected Protocols	20
4.8.2. Implementing listeners for Unconnected Protocols	20
4.8.3. Implementing listeners for Multiplexed Protocols	20
5. Implementing Sending and Receiving Data	21
5.1. Sending Messages	21
5.1.1. Message Properties	21
5.1.2. Send Completion	23
5.1.3. Batching Sends	23
5.2. Receiving Messages	23
5.3. Handling of data for fast-open protocols	24
6. Implementing Message Framers	24

6.1.	Defining Message Framers	25
6.2.	Sender-side Message Framing	26
6.3.	Receiver-side Message Framing	26
7.	Implementing Connection Management	27
7.1.	Pooled Connection	28
7.2.	Handling Path Changes	28
8.	Implementing Connection Termination	29
9.	Cached State	30
9.1.	Protocol state caches	30
9.2.	Performance caches	31
10.	Specific Transport Protocol Considerations	32
10.1.	TCP	33
10.2.	UDP	34
10.3.	TLS	35
10.4.	DTLS	37
10.5.	HTTP	37
10.6.	QUIC	38
10.7.	HTTP/2 transport	39
10.8.	SCTP	39
11.	IANA Considerations	42
12.	Security Considerations	42
12.1.	Considerations for Candidate Gathering	42
12.2.	Considerations for Candidate Racing	42
13.	Acknowledgements	42
14.	References	43
14.1.	Normative References	43
14.2.	Informative References	44
Appendix A.	Additional Properties	45
A.1.	Properties Affecting Sorting of Branches	45
Appendix B.	Reasons for errors	45
Appendix C.	Existing Implementations	46
Authors' Addresses	47

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 Connection Objects

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

- o the Preconnection, the bundle of properties that describes the application constraints on the transport;
- o the Connection, the basic object that represents a flow of data in either direction between the Local and Remote Endpoints;
- o 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 via Selection Properties and, if desired, also Connection Properties. Generally,

Connection Properties should be configured as early as possible, as they may serve as input to decisions that are made by the implementation (the Capacity Profile may guide usage of a protocol offering scavenger-type congestion control, for example). In the remainder of this document, we only refer to Selection Properties because they are the more typical case and have to be handled by all implementations.

The implementation stores these objects and properties as 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 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 have 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 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. 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 so that the transport system can select which set to use. This document structures candidates for racing as a tree.

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 Selection Properties.

4.1.1. Gathering Endpoint Candidates

Both Local and Remote Endpoint Candidates must be discovered during connection establishment. To support ICE, or similar protocols, that involve out-of-band indirect signalling 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.

4.1.1.1. Local Endpoint candidates

The set of possible Local Endpoints is gathered. In the simple case, this merely enumerates the local interfaces and protocols, 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 Local Endpoints is primarily a 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.

4.1.1.2. 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 of this connection.

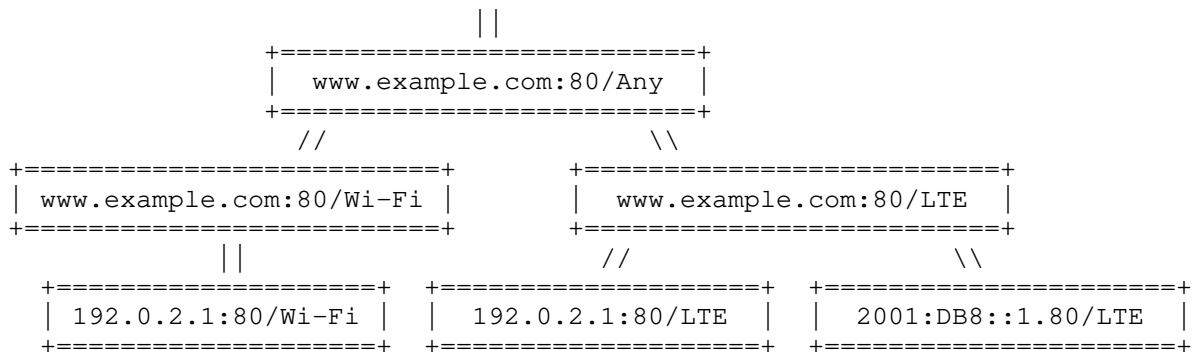
How this is done will depend on the type of the Remote Endpoint, and can also be specific to each Local Endpoint. A common case is when the Remote Endpoint is a DNS name, in which case it is resolved to give a set of IPv4 and IPv6 addresses representing that name. Some types of remote might require more 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.

4.1.2. 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]

4.1.3. 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.

4.1.3.1. 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]

4.1.3.2. 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.

4.1.3.3. 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 9.2). 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.

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 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 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]

4.3. 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 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:

- o "Interface Instance or 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 9.2:
 - * Scavenger: Prefer paths with the highest expected available bandwidth, based on observed maximum throughput
 - * Low Latency/Interactive: Prefer paths with the lowest expected Round Trip Time
 - * Constant-Rate Streaming: Prefer paths that can satisfy the requested Stream Send or Stream Receive Bitrate, based on observed maximum throughput

Implementations should process 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 branches according to preferred properties, and finally use avoided properties as a tiebreaker.

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

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 as part of a Pooled Connection, see Section 7.1, or in future connections. In both cases, keeping additional connections is generally 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 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 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 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 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) 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. 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

- 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 **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.

- 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 Final: when this is true, it means that a transport connection can be closed immediately after its transmission.
- 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 Transmission Profile: TBD - because it's not final in the API yet. Old text follows: 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]).
- o Singular Transmission: when this is true, the application requests to avoid transport-layer segmentation or network-layer fragmentation. Some transports implement network-layer fragmentation avoidance (Path MTU Discovery) without exposing this functionality to the application; in this case, only transport-layer segmentation should be avoided, by fitting the message into a single transport-layer segment or otherwise failing. Otherwise, network-layer fragmentation should be avoided--e.g. by requesting the IP Don't Fragment bit to be set in case of UDP(-Lite) and IPv4 (SET_DF in [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 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.

5.1.3. 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.

5.2. 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.

5.3. 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 [RFC8446]. 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 Message Framers

Message Framers are pieces of code 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 interface 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 can serve the purpose of framing data over TCP, but is exposed as a protocol natively supported by the Transport Services interface.

Most Message Framers fall into one of two categories:

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

Common Message Framers can be provided by the 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 interface for defining Message Framers as an example.

6.1. Defining Message Framers

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

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)
```

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)
```

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 like STARTTLS, that need to add TLS conditionally, to modify the Protocol Stack based on a handshake result.

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

6.2. Sender-side Message Framing

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

```
MessageFramer -> NewSentMessage<Connection, MessageData, MessageContext, IsEndOf  
Message>
```

Upon receiving this event, a framer implementation is responsible for performing any necessary transformations and sending the resulting data 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 system 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 System should store the properties in a generic storage, 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 System 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 implementation 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 protocols that employ request/response pairs and do not require in-order delivery of the responses, like HTTP, the transport implementation may distribute interactions across several underlying transport connections. For these kinds of protocols, implementations may hide the connection management and only expose a single Connection object and the individual requests/responses as messages. These Pooled Connections can use multiple connections or multiple streams of multi-streaming connections between endpoints, as long as all of these satisfy the requirements, and prohibitions specified in the Selection Properties of the Pooled Connection. This enables implementations to realize transparent connection coalescing, connection migration, and to perform per-message endpoint and path selection by choosing among these underlying connections.

7.2. 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 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. Pooled Connections Section 7.1 may add or remove underlying transport connections in a similar manner. 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.

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 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 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].

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). On the other hand, 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:

- 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.

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:

- 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.

10. Specific Transport Protocol Considerations

Each protocol that can run as part of a Transport Services implementation defines both its API mapping as well as implementation details. API mappings for a protocol apply most to Connections in which the 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. If HTTP/2 is used on top of TCP, the HTTP/2 mappings take precedence.

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

- o Unconnected. Unconnected protocols do not establish explicit state between endpoints, and do not perform a handshake during Connection establishment.
- o 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.
- o 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:

- o Byte-stream. Byte-stream protocols do not define any Message boundaries of their own apart from the end of a stream in each direction.
- o Datagram. Datagram protocols define Message boundaries at the same level of transmission, such that only complete (not partial) Messages are supported.
- o 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, primitives in the style of "CATEGORY.[SUBCATEGORY].PRIMITIVE_NAME.PROTOCOL" (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, which refers back the specifications for each protocol. This back-tracking method applies to all elements of [I-D.ietf-taps-minset] (see appendix D of [I-D.ietf-taps-interface]): they are listed in appendix A of [I-D.ietf-taps-minset] with an implementation hint in the same style, pointing back to section 4 of [RFC8303].

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 idempotent 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; or hitting a TCP retransmission timeout.

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. The two Connections are otherwise independent.

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.

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 or ABORT-EVENT.TCP) from the peer.

Close: Calling "Close" on a TCP Connection indicates that the Connection should be gracefully closed (CLOSE.TCP) by sending a FIN to the peer and waiting for a FIN-ACK before delivering the "Closed" event.

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. UDP

Connectedness: Unconnected

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".

10.3. TLS

The mapping of a TLS stream abstraction into the application is equivalent to the contract provided by TCP (see Section 10.1), and builds upon many of the actions of TCP connections.

Connectedness: Connected

Data Unit: Byte-stream

Connection Object: Connection objects represent a single TLS connection running over a TCP connection between two hosts.

Initiate: Calling "Initiate" on a TLS Connection causes it to first initiate a TCP connection. Once the TCP protocol is Ready, the

TLS handshake will be performed as a client (starting by sending a "client_hello", and so on).

InitiateWithSend: Early idempotent data is supported by TLS 1.3, and sends encrypted application data in the first TLS message when performing session resumption. For older versions of TLS, or if a session is not being resumed, the initial data will be delayed until the TLS handshake is complete. TCP Fast Option can also be enabled automatically.

Ready: A TLS Connection is ready once the underlying TCP connection is Ready, and TLS handshake is also complete and keys have been established to encrypt application data.

InitiateError: In addition to TCP initiation errors, TLS can generate errors during its handshake. Examples of error include a failure of the peer to successfully authenticate, the peer rejecting the local authentication, or a failure to match versions or algorithms.

ConnectionError: TLS connections will generate TCP errors, or errors due to failures to rekey or decrypt received messages.

Listen: Calling "Listen" for TLS listens on TCP, and sets up received connections to perform server-side TLS handshakes.

ConnectionReceived: TLS Listeners will deliver new connections once they have successfully completed both TCP and TLS handshakes.

Clone: As with TCP, calling "Clone" on a TLS Connection creates a new Connection with equivalent parameters. The two Connections are otherwise independent.

Send: Like TCP, TLS does not preserve message boundaries. Although application data is framed natively in TLS, there is not a general guarantee that these TLS messages represent semantically meaningful application stream boundaries. Rather, sending data on a TLS Connection only guarantees that the application data will be transmitted in an encrypted form. Marking Messages as Final causes a "close_notify" to be generated once the data has been written.

Receive: Like TCP, TLS delivers a stream of bytes without any Message delineation. The data is decrypted prior to being delivered to the application. If a "close_notify" is received, the stream-wide Message will be delivered with EndOfMessage set.

Close: Calling "Close" on a TLS Connection indicates that the Connection should be gracefully closed by sending a "close_notify" to the peer and waiting for a corresponding "close_notify" before delivering the "Closed" event.

Abort: Calling "Abort" on a TCP Connection indicates that the Connection should be immediately closed by sending a "close_notify", optionally preceded by "user_canceled", to the peer. Implementations do not need to wait to receive "close_notify" before delivering the "Closed" event.

10.4. DTLS

DTLS follows the same behavior as TLS (Section 10.3), with the notable exception of not inheriting behavior directly from TCP. Differences from TLS are detailed below, and all cases not explicitly mentioned should be considered the same as TLS.

Connectedness: Connected

Data Unit: Datagram

Connection Object: Connection objects represent a single DTLS connection running over a set of UDP ports between two hosts.

Initiate: Calling "Initiate" on a DTLS Connection causes it reserve a UDP local port, and begin sending handshake messages to the peer over UDP. These messages are reliable, and will be automatically retransmitted.

Ready: A DTLS Connection is ready once the TLS handshake is complete and keys have been established to encrypt application data.

Send: Sending over DTLS does preserve message boundaries in the same way that UDP datagrams do. Marking a Message as Final does send a "close_notify" like TLS.

Receive: Receiving over DTLS delivers one decrypted Message for each received DTLS datagram. If a "close_notify" is received, a Message will be delivered that is marked as Final.

10.5. 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.

Connectedness: Multiplexing Connected

Data Unit: Message

Connection Object: Connection objects represent a flow of HTTP messages between a client and a server, which may be an HTTP/1.1 connection over TCP, or a single stream in an HTTP/2 connection.

Initiate: Calling "Initiate" on an HTTP connection initiates a TCP or TLS connection as a client.

Clone: Calling "Clone" on an HTTP Connection opens a new stream on an existing HTTP/2 connection when possible. If the underlying version does not support multiplexed streams, calling "Clone" simply creates a new parallel connection.

Send: When an application sends an HTTP Message, it is expected to provide HTTP header values as a MessageContext in a canonical form, along with any associated HTTP message body as the Message data. The HTTP header values are encoded in the specific version format upon sending.

Receive: HTTP Connections deliver Messages in which HTTP header values attached to MessageContexts, and HTTP bodies in Message data.

Close: Calling "Close" on an HTTP Connection will only close the underlying TLS or TCP connection if the HTTP version does not support multiplexing. For HTTP/2, for example, closing the connection only closes a specific stream.

10.6. 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 (often after some timeout), or after an individual stream Connection sends an Abort.

Connectedness: Multiplexing Connected

Data Unit: Stream

Connection Object: Connection objects represent a single QUIC stream on a QUIC connection.

10.7. HTTP/2 transport

Similar to QUIC (Section 10.6), 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 10.1).

Connectedness: Multiplexing Connected

Data Unit: Stream

Connection Object: Connection objects represent a single HTTP/2 stream on a HTTP/2 connection.

10.8. SCTP

Connectedness: Connected

Data Unit: Message

API mappings for SCTP are as follows:

Connection Object: Connection objects represent a flow of SCTP messages between a client and a server, which may be an SCTP association or a stream in a SCTP association. How to map Connection objects to streams is described in [NEAT-flow-mapping]; in the following, a similar method is described. To map Connection objects to SCTP streams without head-of-line blocking on the sender side, both the sending and receiving SCTP

implementation must support message interleaving [RFC8260]. Both SCTP implementations must also support stream reconfiguration. Finally, both communicating endpoints must be aware of this intended multiplexing; [NEAT-flow-mapping] describes a way for a Transport System to negotiate the stream mapping capability using SCTP's adaptation layer indication, such that this functionality would only take effect if both ends sides are aware of it. The first flow, for which the SCTP association has been created, will always use stream id zero. All additional flows are assigned to unused stream ids in growing order. To avoid a conflict when both endpoints map new flows simultaneously, the peer which initiated the transport connection will use even stream numbers whereas the remote side will map its flows to odd stream numbers. Both sides maintain a status map of the assigned stream numbers. Generally, new streams must consume the lowest available (even or odd, depending on the side) stream number; this rule is relevant when lower numbers become available because Connection objects associated to the streams are closed.

Initiate: If this is the only Connection object that is assigned to the SCTP association or stream mapping has not been negotiated, `CONNECT.SCTP` is called. Else, a new stream is used: if there are enough streams available, "Initiate" is just a local operation that assigns a new stream number 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 has not been negotiated, `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, and stream mapping has been negotiated 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 has been negotiated, "Listen" just expects to receive a new message on a new stream id (chosen in accordance with the stream number 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 number in accordance with the stream number 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. 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 system that the stream number can now be re-used by the next "Initiate", "InitiateWithSend" or "Listen" calls.

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

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).

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.

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-04 (work in progress), July 2019.

[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", draft-ietf-taps-interface-04 (work in progress), July 2019.

[I-D.ietf-taps-minset]

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

[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>>.
- [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>>.

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-23 (work in progress), September 2019.
- [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..

14.3. URIs

- [1] <https://developer.apple.com/documentation/network>
- [2] <https://github.com/NEAT-project/neat>
- [3] <https://www.neat-project.org>
- [4] <https://github.com/fg-inet/python-asyncio-taps>

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 **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.
- 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.

Appendix B. 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.

- o **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.
- o **NoCandidates:** The configuration is valid, but none of the available transport protocols can satisfy the transport properties provided by the application.

- o `ResolutionFailed`: The remote or local specifier provided by the application can not be resolved.
- o `EstablishmentFailed`: The TAPS system was unable to establish a transport-layer connection to the remote endpoint specified by the application.
- o `PolicyProhibited`: The system policy prevents the transport system from performing the action requested by the application.
- o `NotCloneable`: The protocol stack is not capable of being cloned.
- o `MessageTooLarge`: The message size is too big for the transport system to handle.
- o `ProtocolFailed`: The underlying protocol stack failed.
- o `InvalidMessageProperties`: The message properties are either contradictory to the transport properties or they can not be satisfied by the transport system.
- o `DeframingFailed`: The data that was received by the underlying protocol stack could not be deframed.
- o `ConnectionAborted`: The connection was aborted by the peer.
- o `Timeout`: Delivery of a message was not possible after a timeout.

Appendix C. 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.

- o `Apple's Network.framework`:
 - * [A very brief introduction should be added]
 - * Documentation: <https://developer.apple.com/documentation/network> [1]
- o `NEAT`:
 - * 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> [2]
- * NEAT project: <https://www.neat-project.org> [3]
- o 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> [4]

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 Enhardt
TU Berlin
Marchstrasse 23
10587 Berlin
Germany

Email: theresa@inet.tu-berlin.de

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

Email: karl-johan.grinnemo@kau.se

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

Email: tom@erg.abdn.ac.uk

Philipp S. Tiesel
TU Berlin
Einsteinufer 25
10587 Berlin
Germany

Email: philipp@tiesel.net

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: Standards Track
Expires: May 7, 2020

B. Trammell, Ed.
Google
M. Welzl, Ed.
University of Oslo
T. Enghardt
TU Berlin
G. Fairhurst
University of Aberdeen
M. Kuehlewind
ETH Zurich
C. Perkins
University of Glasgow
P. Tiesel
TU Berlin
C. Wood
T. Pauly
Apple Inc.
November 04, 2019

An Abstract Application Layer Interface to Transport Services
draft-ietf-taps-interface-05

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 May 7, 2020.

Copyright Notice

Copyright (c) 2019 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
4.1. Usage Examples	8
4.1.1. Server Example	8
4.1.2. Client Example	9
4.1.3. Peer Example	10
4.2. Transport Properties	11
4.2.1. Transport Property Names	12
4.2.2. Transport Property Types	13
4.3. Scope of the Interface Definition	13
5. Pre-Establishment Phase	14
5.1. Specifying Endpoints	15
5.2. Specifying Transport Properties	16
5.2.1. Reliable Data Transfer (Connection)	18
5.2.2. Preservation of Message Boundaries	18
5.2.3. Configure Per-Message Reliability	18
5.2.4. Preservation of Data Ordering	18
5.2.5. Use 0-RTT Session Establishment with an Idempotent Message	19
5.2.6. Multistream Connections in Group	19
5.2.7. Full Checksum Coverage on Sending	19
5.2.8. Full Checksum Coverage on Receiving	19
5.2.9. Congestion control	19
5.2.10. Interface Instance or Type	20
5.2.11. Provisioning Domain Instance or Type	21
5.2.12. Parallel Use of Multiple Paths	21
5.2.13. Direction of communication	22

5.2.14. Notification of excessive retransmissions	22
5.2.15. Notification of ICMP soft error message arrival	22
5.3. Specifying Security Parameters and Callbacks	22
5.3.1. Pre-Connection Parameters	23
5.3.2. Connection Establishment Callbacks	24
6. Establishing Connections	24
6.1. Active Open: Initiate	24
6.2. Passive Open: Listen	26
6.3. Peer-to-Peer Establishment: Rendezvous	27
6.4. Connection Groups	28
7. Sending Data	29
7.1. Basic Sending	30
7.2. Sending Replies	30
7.3. Send Events	31
7.3.1. Sent	31
7.3.2. Expired	31
7.3.3. SendError	32
7.4. Message Properties	32
7.4.1. Lifetime	33
7.4.2. Priority	33
7.4.3. Ordered	34
7.4.4. Idempotent	34
7.4.5. Final	34
7.4.6. Corruption Protection Length	35
7.4.7. Reliable Data Transfer (Message)	35
7.4.8. Message Capacity Profile Override	36
7.4.9. Singular Transmission	36
7.5. Partial Sends	37
7.6. Batching Sends	37
7.7. Send on Active Open: InitiateWithSend	38
8. Receiving Data	38
8.1. Enqueuing Receives	38
8.2. Receive Events	39
8.2.1. Received	39
8.2.2. ReceivedPartial	40
8.2.3. ReceiveError	40
8.3. Receive Message Properties	41
8.3.1. ECN	41
8.3.2. Early Data	41
8.3.3. Receiving Final Messages	41
9. Message Contexts	42
10. Message Framers	42
10.1. Adding Message Framers to Connections	43
10.2. Framing Meta-Data	43
11. Managing Connections	44
11.1. Generic Connection Properties	45
11.1.1. Retransmission Threshold Before Excessive Retransmission Notification	46

11.1.2.	Required Minimum Corruption Protection Coverage for Receiving	46
11.1.3.	Priority (Connection)	46
11.1.4.	Timeout for Aborting Connection	46
11.1.5.	Connection Group Transmission Scheduler	47
11.1.6.	Maximum Message Size Concurrent with Connection Establishment	47
11.1.7.	Maximum Message Size Before Fragmentation or Segmentation	47
11.1.8.	Maximum Message Size on Send	47
11.1.9.	Maximum Message Size on Receive	48
11.1.10.	Capacity Profile	48
11.1.11.	Bounds on Send or Receive Rate	49
11.1.12.	TCP-specific Property: User Timeout	50
11.2.	Soft Errors	50
11.3.	Excessive retransmissions	51
12.	Connection Termination	51
13.	Connection State and Ordering of Operations and Events	51
14.	IANA Considerations	52
15.	Security Considerations	53
16.	Acknowledgements	53
17.	References	53
17.1.	Normative References	53
17.2.	Informative References	54
Appendix A.	Convenience Functions	56
A.1.	Adding Preference Properties	56
A.2.	Transport Property Profiles	56
A.2.1.	reliable-inorder-stream	57
A.2.2.	reliable-message	57
A.2.3.	unreliable-datagram	57
Appendix B.	Additional Properties	58
B.1.	Experimental Transport Properties	58
B.1.1.	Cost Preferences	59
Appendix C.	Sample API definition in Go	59
Appendix D.	Relationship to the Minimal Set of Transport Services for End Systems	59
Authors' Addresses	62

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 [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.

This document builds a modern abstract programming interface atop the high-level architecture for transport services defined in [I-D.ietf-taps-arch]. It derives specific path and protocol selection properties and supported transport features from the analysis provided in [RFC8095], [I-D.ietf-taps-minset], and [I-D.ietf-taps-transport-security].

2. Terminology and Notation

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

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

- o An Action creates an Object:

Object := Action()

- o An Action creates an array of Objects:

[]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 and Event parameters whose names are suffixed with a question mark are optional.

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

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

How these abstract concepts map into concrete implementations of this API in a given language on a given platform is largely dependent on the features of the language and the platform. Actions could be implemented as functions or method calls, for instance, and Events could be implemented via callbacks, communicating sequential processes, or other asynchronous calling conventions. The method for dispatching and handling Events is 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 that can be immediately detected, such as inconsistency in Transport Properties. Errors can provide an optional reason to give the application further details as to why the error occurred.

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 Message-orientation, as opposed to stream-orientation, using application-assisted framing and deframing where the underlying transport does not provide these;
- o Asynchronous Connection establishment, transmission, and reception, allowing concurrent operations during establishment and supporting event-driven application interactions with the transport layer, in line with developments in modern platforms and programming languages;
- 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; and
- 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.

4. API Summary

The Transport Services Interface is the basic common abstract application programming interface to the Transport Services Architecture defined in the TAPS Architecture [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 Message Framing which finds message boundaries in a stream. Messages are received asynchronously through a callback registered by the application. Errors and other notifications also happen asynchronously on the Connection.

Section 5, Section 6, Section 7, Section 8, and Section 12 describe the details of application interaction with Objects through Actions and Events in each phase of a Connection, following the phases described in [I-D.ietf-taps-arch].

4.1. Usage Examples

The following usage examples illustrate how an application might use a Transport Services Interface to:

- o Act as a server, by listening for incoming connections, receiving requests, and sending responses, see Section 4.1.1.
- o Act as a client, by connecting to a remote endpoint using Initiate, sending requests, and receiving responses, see Section 4.1.2.
- o 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 4.1.3.

The examples in this section presume that a transport protocol is available between the 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 may receive this byte stream as partial Messages and transform it into application-layer Messages. Alternatively, an application may provide a Message Framer, which can transform a byte stream into a sequence of Messages (Section 10).

4.1.1. Server Example

This is an example of how an application might listen for incoming Connections using the Transport Services Interface, 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.AddIdentity(identity)
SecurityParameters.AddPrivateKey(privateKey, publicKey)

// Specifying a remote endpoint is optional when using Listen()
Preconnection := NewPreconnection(LocalSpecifier,
                                   None,
                                   TransportProperties,
                                   SecurityParameters)

Listener := Preconnection.Listen()

Listener -> ConnectionReceived<Connection>

// Only receive complete messages
Connection.Receive()

Connection -> Received(messageDataRequest, messageContext)

Connection.Send(messageDataResponse)

Connection.Close()

// Stop listening for incoming Connections
Listener.Stop()
```

4.1.2. Client Example

This is an example of how an application might connect to a remote application using the Transport Services Interface, send a request, and receive a response.

```
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 := New Callback({
    // Verify identity of the remote endpoint, return the result
})
SecurityParameters.SetTrustVerificationCallback(TrustCallback)

// Specifying a local endpoint is optional when using Initiate()
Preconnection := NewPreconnection(None,
                                   RemoteSpecifier,
                                   TransportPreperties,
                                   SecurityParameters)

Connection := Preconnection.Initiate()

Connection -> Ready<>

Connection.Send(messageDataRequest)

// Only receive complete messages
Connection.Receive()

Connection -> Received(messageDataResponse, messageContext)

Connection.Close()
```

4.1.3. 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`.

```
LocalSpecifier := NewLocalEndpoint()
LocalSpecifier.WithPort(9876)

RemoteSpecifier := NewRemoteEndpoint()
RemoteSpecifier.WithHostname("example.com")
RemoteSpecifier.WithPort(9877)

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

SecurityParameters := NewSecurityParameters()
SecurityParameters.AddIdentity(identity)
SecurityParameters.AddPrivateKey(privateKey, publicKey)

TrustCallback := New Callback({
    // Verify identity of the remote endpoint, return the result
})
SecurityParameters.SetTrustVerificationCallback(trustCallback)

// Both local and remote endpoint must be specified
Preconnection := NewPreconnection(LocalSpecifier,
                                   RemoteSpecifier,
                                   TransportPreperties,
                                   SecurityParameters)

Preconnection.Rendezvous()

Preconnection -> RendezvousDone<Connection>

Connection.Send(messageDataRequest)

// Only receive complete messages
Connection.Receive()

Connection -> Received(messageDataResponse, messageContext)

Connection.Close()
```

4.2. Transport Properties

Each application using the Transport Services Interface declares its preferences for how the transport service should operate using properties at each stage of the lifetime of a connection using Transport Properties, as defined in [I-D.ietf-taps-arch].

Transport Properties are divided into Selection, Connection, and Message Properties. During pre-establishment, Selection Properties

(see Section 5.2) are used to specify which paths and protocol stacks can be used and are preferred by the application, and Connection Properties (see Section 11.1) can be used to influence decisions made during establishment and to fine-tune the eventually established connection. These Connection Properties can also be used later, to monitor and fine-tune established connections. The behavior of the selected protocol stack(s) when sending Messages is controlled by Message Properties (see Section 7.4).

All Transport Properties, regardless of the phase in which they are used, are organized within a single namespace. This enables setting them as defaults in earlier stages and querying them in later stages:

- o Connection Properties can be set on Preconnections
- o Message Properties can be set on Preconnections and Connections
- o The effect of Selection Properties can be queried on Connections and Messages

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, see Section 4.2.1, should not be used as an input to the selection process.

4.2.1. Transport Property Names

Transport Properties are referred to by property names. These names are lower-case strings whereby words are separated by hyphens. These names serve two purposes:

- o Allow different components of a TAPS 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.
- o Make code of different TAPS implementations look similar.

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

- o The Namespace part is empty for well known, generic properties, i.e., for properties that are not specific to a protocol and are defined in an RFC.

- o Protocol Specific Properties must use the protocol acronym as Namespace, e.g., "tcp" for TCP specific Transport Properties. For IETF protocols, property names under these namespaces should be defined in an RFC.
- o Vendor or implementation specific properties must use a string identifying the vendor or implementation as Namespace.

4.2.2. Transport Property Types

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

- o Boolean: can take the values "true" and "false"; representation is implementation-dependent.
- o Integer: can take positive or negative numeric integer values; range and representation is implementation-dependent.
- o Numeric: can take positive or negative numeric values; range and representation is implementation-dependent.
- o Enumeration: can take one value of a finite set of values, dependent on the property itself. The representation is implementation dependent; however, implementations MUST provide a method for the application to determine the entire set of possible values for each property.
- o Preference: can take one of five values (Prohibit, Avoid, Ignore, Prefer, Require) for the level of preference of a given property during protocol selection; see Section 5.2.

4.3. Scope of the Interface Definition

This document defines a language- and platform-independent interface to a Transport Services system. Given the wide variety of languages and language conventions used to write applications that use the transport layer to connect to other applications over the Internet, this independence makes this interface necessarily abstract.

There is no interoperability benefit in tightly defining how the interface is presented to application programmers across diverse platforms. However, maintaining the "shape" of the abstract interface across these platforms reduces the effort for programmers who learn the transport services interface to then apply their knowledge across multiple platforms.

We therefore make the following recommendations:

- o Actions, Events, and Errors in implementations of this interface 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.
- o Implementations of this interface SHOULD implement each Selection Property, Connection Property, and Message Context Property specified in this document, exclusive of appendices. Each interface SHOULD be implemented even when this 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.
- o 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.

5. Pre-Establishment Phase

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

A Preconnection Object represents a potential Connection. It has state that describes properties of a Connection that might exist in the future. This state comprises Local Endpoint and Remote Endpoint Objects that denote the endpoints of the potential Connection (see Section 5.1), the Selection Properties (see Section 5.2), any preconfigured Connection Properties (Section 11.1), and the security parameters (see Section 5.3):

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

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

Message Framers (see Section 10), if required, should be added to the Preconnection during pre-establishment.

5.1. Specifying Endpoints

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

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

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

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

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

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

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

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

The transport services API resolves names internally, when the `Initiate()`, `Listen()`, or `Rendezvous()` method is called establish a Connection. The API explicitly does not require the application to resolve names, though there is a tradeoff between early and late binding of addresses to names. Early binding allows the API implementation to reduce connection setup latency, at the cost of potentially limited scope for alternate path discovery during Connection establishment, as well as potential additional information

leakage about application interest when used with a resolution method (such as DNS without TLS) which does not protect query confidentiality.

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

5.2. Specifying Transport Properties

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

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

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

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

In addition, the pseudo-level "Default" can be used to reset the property to the default level used by the implementation. This level

will never show up when queuing the value of a preference - the effective preference must be returned instead.

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

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 added to the TransportProperties Object:

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

As preference typed selection properties may be used quite frequently, implementations should provide additional convenience functions as outlined in Appendix A.1. In addition, implementations should provide a mechanism to create TransportProperties objects that are preconfigured for common use cases as outlined in Appendix A.2.

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

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

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

Section 11.1 provides a list of Connection Properties, while Selection Properties are listed in the subsections below. Note that many properties are only considered during establishment, and can not be changed after a Connection is established; however, they can be queried. Querying a Selection Property after establishment yields

the value Required for properties of the selected protocol and path, Avoid for properties avoided during selection, and Ignore for all other properties.

An implementation of this interface must provide sensible defaults for Selection Properties. The defaults given for each property below represent a configuration that can be implemented over TCP. An alternate set of default Protocol Selection Properties would represent a configuration that can be implemented over UDP.

5.2.1. Reliable Data Transfer (Connection)

Name: reliability

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

5.2.2. Preservation of Message Boundaries

Name: preserve-msg-boundaries

This property specifies whether the application needs or prefers to use a transport protocol that preserves message boundaries. The default is to Prefer Preservation of Message Boundaries.

5.2.3. Configure Per-Message Reliability

Name: per-msg-reliability

This property specifies whether an application considers it useful to indicate its reliability requirements on a per-Message basis. This property applies to Connections and Connection Groups. The default is to Ignore this option.

5.2.4. Preservation of Data Ordering

Name: preserve-order

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

5.2.5. Use 0-RTT Session Establishment with an Idempotent Message

Name: zero-rtt-msg

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 (i.e., multiple copies of the message data may be passed to the Remote Endpoint). See also Section 7.4.4. The default is to Ignore this option. Note that disabling this property has no effect for protocols that are not connection-oriented and do not protect against duplicated messages, e.g., UDP.

5.2.6. Multistream Connections in Group

Name: multistreaming

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 Prefer this option.

5.2.7. Full Checksum Coverage on Sending

Name: per-msg-checksum-len-send

This property specifies whether the application desires protection against corruption for all data transmitted on this Connection. Disabling this property may enable to control checksum coverage later (see Section 7.4.6). The default is to Require this option.

5.2.8. Full Checksum Coverage on Receiving

Name: per-msg-checksum-len-recv

This property specifies whether the application desires protection against corruption for all data received on this Connection. The default is to Require this option.

5.2.9. Congestion control

Name: congestion-control

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

with [RFC2914]. Also note that reliability is usually combined with congestion control in protocol implementations, rendering "reliable but not congestion controlled" a request that is unlikely to succeed. The recommended default is to Require that the Connection is congestion controlled.

5.2.10. Interface Instance or Type

Name: interface

Type: Set (Preference, Enumeration)

This property allows the application to select which specific network interfaces or categories of interfaces it wants to "Require", "Prohibit", "Prefer", or "Avoid". Note that marking a specific interface as "Required" strictly limits path selection to a single interface, and may often lead 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, there may be "Wi-Fi" and "Wired Ethernet" interface types available. An implementation should provide all types that are supported on the local system to all remote systems, 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 may want to always "Prohibit Cellular". Note that marking a specific interface type as "Required" limits path selection to a small set of interfaces, and leads to less flexible and resilient connection establishment.

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

Interface types should not be treated as a proxy for properties of interfaces such as metered or unmetered network access. If an application needs to prohibit metered interfaces, this should be specified via Provisioning Domain attributes (see Section 5.2.11) or another specific property.

5.2.11. Provisioning Domain Instance or Type

Name: pvd

Type: Set (Preference, Enumeration)

Similar to interface instances and types (see Section 5.2.10), this property allows the application to control path selection by selecting which specific Provisioning Domains or categories of Provisioning Domains it wants to "Require", "Prohibit", "Prefer", or "Avoid". Provisioning Domains define consistent sets of network properties that may be more specific than network interfaces [RFC7556].

As with interface instances and types, this property is 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 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 broader than requiring specific PvD instances or interface instances, and should be preferred over these options.

5.2.12. Parallel Use of Multiple Paths

Name: multipath

This property specifies whether an application considers it useful to transfer data across multiple paths between the same end hosts. Generally, in most cases, this will improve performance (e.g., achieve greater throughput). One possible side-effect is increased jitter, which may be problematic for delay-sensitive applications. The recommended default is to Ignore this option.

5.2.13. Direction of communication

Name: direction

Type: Enumeration

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

The default is bidirectional. 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.

5.2.14. Notification of excessive retransmissions

Name: :retransmit-notify

This property specifies whether an application considers it useful to be informed in case sent data was retransmitted more often than a certain threshold. The default is to Ignore this option.

5.2.15. Notification of ICMP soft error message arrival

Name: :soft-error-notify

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. The default is to Ignore this option.

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

Once Initiate is called, the candidate Protocol Stack(s) may cause one or more candidate transport-layer connections to be created to the specified remote endpoint. The caller may immediately begin sending Messages on the Connection (see Section 7) after calling 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<reason?>

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

See also Section 7.7 to combine Connection establishment and transmission of the first message in a single action.

6.2. Passive Open: Listen

Passive open is the Action of waiting for Connections from remote Endpoints, commonly used by servers in client-server interactions. Passive open is supported by this interface through the Listen Action 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, properties added to the Preconnection have no effect on the Listener and the Preconnection can be disposed of or reused.

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

```
Listener.Stop()
```

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

```
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 -> ListenError<reason?>

A ListenError occurs either when the Properties of 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.

Listener -> Stopped<>

A Stopped Event occurs after the Listener 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<messageContext, reason?>

An RendezvousError occurs either when the Preconnection cannot be fulfilled for listening, when the Local Endpoint or Remote Endpoint cannot be resolved, when no transport-layer connection can be

established to the Remote Endpoint, or when the application is prohibited from rendezvous by policy.

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

```
[ ]Preconnection := Preconnection.Resolve()
```

The `Resolve()` call returns a list of `Preconnection` Objects, that represent the concrete addresses, local and server reflexive, on which a `Rendezvous()` for the `Preconnection` will listen for incoming `Connections`. These resolved `Preconnections` will share all other `Properties` with the `Preconnection` from which they are derived, though some `Properties` may be made more-specific by the resolution process. This list can be passed to a peer via a signalling protocol, such as SIP [RFC3261] or WebRTC [RFC7478], to configure the remote.

6.4. Connection Groups

Entangled `Connections` can be created using the `Clone` Action:

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

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

In addition, incoming entangled `Connections` can be received by creating a `Listener` on an existing connection:

```
Listener := Connection.Listen()
```

Changing one of these `Protocol Properties` on one `Connection` in the group changes it for all others. `Per-Message Protocol Properties`, however, are not entangled. For example, changing "Timeout for aborting `Connection`" (see Section 11.1.4) on one `Connection` in a group will automatically change this `Protocol Property` for all `Connections` in the group in the same way. However, changing

"Lifetime" (see Section 7.4.1) of a Message will only affect a single Message on a single Connection, entangled or not.

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

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

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

The Protocol Property "Priority" operates on entangled Connections as in Section 7.4.2: 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 with lower Priority values. An ideal transport system implementation would assign each Connection the capacity share $(M-N) \times C / M$, where N is the Connection's Priority value, M is the maximum Priority value used by all Connections in the group and C is the total available capacity. However, the Priority setting is purely advisory, and no guarantees are given about the way capacity is shared. Each implementation is free to implement a way to share capacity that it sees fit.

7. Sending Data

Once a Connection has been established, it can be used for sending data. Data is sent as 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 7.3). Sending partial Messages for streaming large data is also supported (see Section 7.5).

Messages are sent on a Connection using the Send action:

```
Connection.Send(messageData, messageContext?, endOfMessage?)
```

where messageData is the data object to send.

The optional messageContext parameter supports per-message properties and is described in Section 7.4. It can be used to identify send

events (see Section 7.3) related to a specific message or to inspect meta-data related to the message sent (see Section 9).

The optional `endOfMessage` parameter supports partial sending and is described in Section 7.5.

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 transferred as an array of bytes, and the resulting object contains both the byte array and the length of the array.

```
messageData := "hello".bytes()  
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 11.1.8) 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 7.3.3). For example, it is invalid to send a Message over a UDP connection that is larger than the available datagram sending size.

7.2. Sending Replies

When a message is sent in response to a message received, the application may use the Message Context of the received Message to construct a Message Context for the reply.

```
replyMessageContext := requestMessageContext.reply()
```

By using the "replyMessageContext", the transport system is informed that the message to be sent is a response and can map the response to the same underlying transport connection or stream the request was received from. The concept of Message Contexts is described in Section 9.

7.3. 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. Exactly one Event (Sent, Expired, or SendError) will be delivered in response to each call to Send. These Events can be implemented as callbacks that allow the specific Event to be associated with the call to Send.

Note that if partial Sends are used (Section 7.5), 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.

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

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

7.3.2. 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 7.4.1) expired. This is separate from SendError, as it is an expected behavior for partially reliable transports. The Expired Event contains an implementation-specific reference to the Message to which it applies.

7.3.3. SendError

Connection -> SendError<messageContext, reason?>

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.

7.4. Message Properties

Applications may need to annotate the Messages they send with extra information to control how data is scheduled and processed by the transport protocols in the Connection. Therefore a message context containing these properties can be passed to the Send Action. For other uses of the message context, see Section 9.

Note that message properties are per-Message, not per-Send if partial Messages are sent (Section 7.5). 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 Messages to be sent or received.

```
messageData := "hello".bytes()
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 to it.

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

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

Message Properties may be inconsistent with the properties of the Protocol Stacks underlying the Connection on which a given Message is sent. For example, a Connection must provide reliability to allow setting an 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 Properties are supported:

7.4.1. Lifetime

Name: msg-lifetime

Type: Integer

Default: infinite

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 system - 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 7.4.7). The type and units of Lifetime are implementation-specific.

7.4.2. Priority

Name: msg-prio

Type: Integer (non-negative)

Default: 100

This property represents a hierarchy of priorities. It can specify 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 11.1.3. Both Priority properties may interact, but can be used independently and be realized by different mechanisms.

7.4.3. Ordered

Name: msg-ordered

Type: Boolean

Default: true

If true, it specifies that the receiver-side transport protocol stack only deliver the Message to the receiving application after the previous ordered Message which was passed to the same Connection via the Send Action, when such a Message exists. If false, the Message may be delivered to the receiving application out of order. This property is used for protocols that support preservation of data ordering, see Section 5.2.4, but allow out-of-order delivery for certain messages.

7.4.4. Idempotent

Name: idempotent

Type: Boolean

Default: false

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

Note that for protocols that do not protect against duplicated messages, e.g., UDP, all messages MUST be marked as Idempotent. In order to enable protocol selection to choose such a protocol, Idempotent MUST be added to the TransportProperties passed to the Preconnection. If such a protocol was chosen, disabling Idempotent on individual messages MUST result in a SendError.

7.4.5. Final

Type: Boolean

Name: final

Default: false

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 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.

7.4.6. Corruption Protection Length

Name: `msg-checksum-len`

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

Default: full coverage

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

7.4.7. Reliable Data Transfer (Message)

Name: `msg-reliable`

Type: Boolean

Default: true

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 with the Selection Property 'Reliable Data Transfer (Connection)' enabled. When this is not the case, changing it will

generate an error. Disabling this property indicates that the transport system may disable retransmissions or other reliability mechanisms for this particular Message, but such disabling is not guaranteed.

7.4.8. Message Capacity Profile Override

Name: msg-capacity-profile

Type: Enumeration

This enumerated property specifies the application's preferred tradeoffs for sending this Message; it is a per-Message override of the Capacity Profile protocol and path selection property (see Section 11.1.10).

The following values are valid for Transmission Profile:

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

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

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

7.4.9. Singular Transmission

Name: singular-transmission

Type: Boolean

Default: false

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

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

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

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

7.6. Batching Sends

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)
)
```

7.7. 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 idempotent. This allows the transport 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 `InitiateError` signaling the failure of Connection establishment.

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 (see Section 8.2.2 and Section 10 for more information on how this is accomplished). If this value is set to some smaller value, the associated receive event will be triggered only when at least that many bytes are available, or the Message is complete with fewer bytes, or the system needs to free up memory. Applications should always check the length of the data delivered to the receive event and not assume it will be as long as `minIncompleteLength` in the case of shorter complete Messages or memory issues.

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

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

8.2. Receive Events

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

8.2.1. Received

Connection -> Received<messageData, messageContext>

A Received event indicates the delivery of a complete Message. It contains two objects, the received bytes as `messageData`, and the metadata and properties of the received Message as `messageContext`.

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, e.g., to send replies and map responses to their requests. See Section 9 for details.

See Section 10 for handling Message framing in situations where the Protocol Stack only provides a byte-stream transport.

8.2.2. ReceivedPartial

Connection -> ReceivedPartial<messageData, messageContext, endOfMessage>

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

Multiple invocations of ReceivedPartial deliver data for the same Message by passing the same MessageContext, until the endOfMessage flag is delivered or a ReceiveError occurs. All partial blocks of a single Message are delivered in order without gaps. This event does not support delivering 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 Message Frammer (see Section 10) 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 Message Frammer was supplied by the application

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

8.2.3. ReceiveError

Connection -> ReceiveError<messageContext, reason?>

A ReceiveError occurs when data is received by the underlying Protocol Stack that cannot be fully retrieved or parsed, 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 12).

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. 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) passed by the receive event. The following metadata values are supported:

8.3.1. ECN

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

8.3.2. Early Data

In some cases it may be valuable to know whether data was read as part of early data transfer (before connection establishment has finished). This is useful if applications need to treat early data separately, e.g., if early data has different security properties than data sent after connection establishment. In the case of TLS 1.3, client early data can be replayed maliciously (see [RFC8446]). 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 Message Context can indicate whether or not this Message is the Final Message on a Connection. For any Message that is marked as Final, the application can assume that there will be no more Messages received on the Connection once the Message has been completely

delivered. This corresponds to the Final property that may be marked on a sent Message Section 7.4.5.

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

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

9. Message Contexts

Using the MessageContext object, the application can set and retrieve meta-data of the message, including Message Properties (see Section 7.4) and framing meta-data (see Section 10.2). Therefore, a MessageContext object can be passed to the Send action and is returned by each Send and Receive related events.

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

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

To get or set Message Properties, the optional scope parameter is left empty, for framing meta-data, the framer is passed.

For MessageContexts returned by send events (see Section 7.3) and receive events (see Section 8.2), the application can query information about the local and remote endpoint:

```
RemoteEndpoint := MessageContext.GetRemoteEndpoint()
LocalEndpoint := MessageContext.GetLocalEndpoint()
```

Message Contexts can also be used to send messages that are flagged as a reply to other messages, see Section 7.2 for details. If the message received was sent by the remote endpoint as a reply to an earlier message and the transports provides this information, the MessageContext of the original request can be accessed using the Message Context of the reply:

```
RequestMessageContext := MessageContext.GetOriginalRequest()
```

10. 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 out 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 facility 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.

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 Framers can vary depending on the implementation; guidance on implementing Message Framers can be found in [I-D.ietf-taps-impl].

10.1. Adding Message Framers to Connections

The Message Framers object can be added to one or more Preconnections to run on top of transport protocols. Multiple Framers may be added. If multiple Framers are added, the last one added runs first when framing outbound messages, and last when parsing inbound data.

The following example adds a basic HTTP Message Framers to a Preconnection:

```
framer := NewHTTPMessageFramer()  
Preconnection.AddFramer(framer)
```

10.2. Framing Meta-Data

When sending Messages, applications can add specific Message values to a MessageContext (Section 9) that is intended for a Framers. This 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 Framers.

```
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")
```

11. 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 11.1 below. Specific Protocol Properties are defined in a transport- and implementation-specific way, and must not be assumed to apply across different protocols. Attempts to set Specific Protocol Properties on a protocol stack not containing that specific protocol are simply ignored, and do not raise an error; however, too much reliance by an application on Specific Protocol Properties may significantly reduce the flexibility of a transport services implementation.

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

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

At any point, the application can query Connection Properties.

```
ConnectionProperties := Connection.GetProperties()
```

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

- o The connection state, 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 "Direction of Communication" set to "unidirectional receive" or if a Message marked as "Final" was sent over this connection, see Section 7.4.5.
- 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 "Direction of Communication" set to "unidirectional send" or if a Message marked as "Final" was received, see Section 8.3.3. The latter is only supported by certain transport protocols, e.g., by TCP as half-closed connection.
- o For Connections that are Establishing: Transport Properties that the application specified on the Preconnection, see Section 5.2.
- o For Connections that are Established, Closing, or Closed: Selection (Section 5.2) and Connection Properties (Section 11.1) of the actual protocols that were selected and instantiated. Selection Properties indicate whether or not the Connection has or offers a certain Selection Property. Note that the actually instantiated protocol stack may not match all Protocol Selection Properties that the application specified on the Preconnection. For example, a certain Protocol Selection Property that an application specified as Preferred may not actually be present in the chosen protocol stack because none of the currently available transport protocols had this feature.
- o For Connections that are Established, additional properties of the path(s) in use. These properties can be derived from the local provisioning domain [RFC7556], measurements by the Protocol Stack, or other sources.

11.1. Generic Connection Properties

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

Note that many protocol properties have a corresponding selection property, which prefers protocols providing a specific transport

feature that controlled by that protocol property. [EDITOR'S NOTE:
todo: add these cross-references up to Section 5.2]

11.1.1. Retransmission Threshold Before Excessive Retransmission Notification

Name: retransmit-notify-threshold

Type: Integer

Default: -1

This property specifies after how many retransmissions to inform the application about "Excessive Retransmissions". The special value -1 means that this notification is disabled.

11.1.2. Required Minimum Corruption Protection Coverage for Receiving

Name: recv-checksum-len

Type: Integer

Default: -1

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 the special value -1 indicates full checksum coverage.

11.1.3. Priority (Connection)

Name: conn-prio

Type: Integer

Default: 100

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

11.1.4. Timeout for Aborting Connection

Name: conn-timeout

Type: Numeric

Default: -1

This property specifies how long to wait before deciding that a Connection has failed when trying to reliably deliver data to the destination. Adjusting this Property will only take effect when the underlying stack supports reliability. The special value -1 means that this timeout is not scheduled to happen.

11.1.5. Connection Group Transmission Scheduler

Name: conn-scheduler

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 6.4. The set of schedulers can be taken from [RFC8260].

11.1.6. Maximum Message Size Concurrent with Connection Establishment

Name: zero-rtt-msg-max-len

Type: Integer (read only)

This property represents the maximum Message size that can be sent before or during Connection establishment, see also Section 7.4.4. It is given in Bytes.

11.1.7. Maximum Message Size Before Fragmentation or Segmentation

Name: singular-transmission-msg-max-len

Type: Integer (read only)

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

11.1.8. Maximum Message Size on Send

Name: send-msg-max-len

Type: Integer (read only)

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

11.1.9. Maximum Message Size on Receive

Name: recv-msg-max-len

Type: Integer (read only)

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

11.1.10. Capacity Profile

Name: conn-capacity-profile

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

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

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

Low Latency/Interactive: The application is interactive, and prefers loss to latency. Response time should be optimized at the expense of bandwidth efficiency and delay variation when sending on this connection. This can be used by the system to disable the coalescing of multiple small Messages into larger packets (Nagle's algorithm); to prefer immediate acknowledgment from the peer endpoint when supported by the underlying transport; and so on.

Transport system implementations that map the requested capacity profile onto per-connection DSCP signaling without multiplexing SHOULD assign the DSCP Expedited Forwarding [RFC3246] PHB; when the Connection is multiplexed, the guidelines in Section 6 of [RFC7657] apply.

Low Latency/Non-Interactive: The application prefers loss to latency but is not interactive. Response time should be optimized at the expense of bandwidth efficiency and delay variation when sending on this connection. Transport system implementations that map the requested capacity profile onto per-connection DSCP signaling without multiplexing SHOULD assign a DSCP Assured Forwarding (AF21,AF22,AF23,AF24) [RFC2597] PHB; when the Connection is multiplexed, the guidelines in Section 6 of [RFC7657] apply.

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

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

The Capacity Profile for a selected protocol stack may be modified on a per-Message basis using the Transmission Profile Message Property; see Section 7.4.8.

11.1.11. Bounds on Send or Receive Rate

Name: max-send-rate / max-recv-rate

Type: Numeric / Numeric

Default: -1 / -1 (unlimited, for both values)

This property specifies 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 a data transfer useful. It is given in bits per second. The special value -1 indicates that no bound is specified.

11.1.12. TCP-specific Property: User Timeout

This property specifies, for the case TCP becomes the chosen transport protocol:

Advertised User Timeout (name: tcp.user-timeout-value, type: Integer):

a time value (default: the TCP default) to be advertised via the User Timeout Option (UTO) for the TCP at the remote endpoint to adapt its own "Timeout for aborting Connection" (see Section 11.1.4) value accordingly.

User Timeout Enabled (name: tcp.user-timeout, type: Boolean): a boolean (default false) to control whether the UTO option is enabled for a connection. This applies to both sending and receiving.

Changeable (name: tcp.user-timeout-recv, type: Boolean): a boolean (default true) which controls whether the "Timeout for aborting Connection" (see Section 11.1.4) may be changed based on a UTO option received from the remote peer. This boolean becomes false when "Timeout for aborting Connection" (see Section 11.1.4) is used.

All of the above parameters 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).

11.2. Soft Errors

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

Connection -> SoftError<>

11.3. Excessive retransmissions

This event notifies the application of excessive retransmissions, based on a configured threshold (see Section 11.1.1). This will only happen if the underlying protocol stack supports reliability and, with it, such notifications.

Connection -> ExcessiveRetransmission<>

12. Connection Termination

Close terminates a Connection after satisfying all the requirements that were specified regarding the delivery of Messages that the application has already given to the transport system. For example, if reliable delivery was requested for a Message handed over before calling Close, the transport system will ensure that this Message is indeed delivered. If the Remote Endpoint still has data to send, it cannot be received after this call.

Connection.Close()

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

Connection -> Closed<>

Abort terminates a Connection without delivering remaining data:

Connection.Abort()

A ConnectionError informs the application that data to could not be delivered after a timeout, or the other side has aborted the Connection; however, there is no guarantee that an Abort will indeed be signaled.

Connection -> ConnectionError<reason?>

13. Connection State and Ordering of Operations and Events

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

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

- o Ready<> occurs when a Connection created with Initiate() or InitiateWithSend() transitions to Established state.
- o ConnectionReceived<> occurs when a Connection created with Listen() transitions to Established state.
- o RendezvousDone<> occurs when a Connection created with Rendezvous() transitions to Established state.
- o Closed<> occurs when a Connection transitions to Closed state without error.
- o InitiateError<> occurs when a Connection created with Initiate() transitions from Establishing state to Closed state due to an error.
- o ConnectionError<> occurs when a Connection transitions to Closed state due to an error in all other circumstances.

The interface provides the following guarantees about the ordering of operations:

- 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).
- o 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.
- o No events will occur on a Connection after it is Closed; i.e., after a Closed<> event, an InitiateError<> or ConnectionError<> on that connection. To ensure this ordering, Closed<> will not occur on a Connection while other events on the Connection are still locally outstanding (i.e., known to the interface and waiting to be dealt with by the application). ConnectionError<> may occur after Closed<>, but the interface must gracefully handle all cases where application ignores these errors.

14. 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 4.2.1).

15. 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.

16. 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.

17. References

17.1. 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-04 (work in progress), July 2019.

[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.

- [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>>.
- [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>>.

17.2. 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-04 (work in progress), July 2019.
- [I-D.ietf-taps-minset]
Welzl, M. and S. Gjessing, "A Minimal Set of Transport Services for End Systems", draft-ietf-taps-minset-11 (work in progress), September 2018.
- [I-D.ietf-taps-transport-security]
Wood, C., Enghardt, T., Pauly, T., Perkins, C., and K. Rose, "A Survey of Transport Security Protocols", draft-ietf-taps-transport-security-09 (work in progress), September 2019.
- [LE-PHB] Bless, R., "A Lower Effort Per-Hop Behavior (LE PHB) for Differentiated Services", draft-ietf-tsvwg-le-phb-10 (work in progress), March 2019.
- [PROTOCOL-WARS]
Computer History Museum, ., "Protocol Wars (Revolution - The First 2000 Years of Computing)", 2019, <<https://www.computerhistory.org/revolution/networking/19/376>>.

- [RFC0793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, DOI 10.17487/RFC0793, September 1981, <<https://www.rfc-editor.org/info/rfc793>>.
- [RFC2474] Nichols, K., Blake, S., Baker, F., and D. Black, "Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers", RFC 2474, DOI 10.17487/RFC2474, December 1998, <<https://www.rfc-editor.org/info/rfc2474>>.
- [RFC2597] Heinanen, J., Baker, F., Weiss, W., and J. Wroclawski, "Assured Forwarding PHB Group", RFC 2597, DOI 10.17487/RFC2597, June 1999, <<https://www.rfc-editor.org/info/rfc2597>>.
- [RFC2914] Floyd, S., "Congestion Control Principles", BCP 41, RFC 2914, DOI 10.17487/RFC2914, September 2000, <<https://www.rfc-editor.org/info/rfc2914>>.
- [RFC3246] Davie, B., Charny, A., Bennet, J., Benson, K., Le Boudec, J., Courtney, W., Davari, S., Firoiu, V., and D. Stiliadis, "An Expedited Forwarding PHB (Per-Hop Behavior)", RFC 3246, DOI 10.17487/RFC3246, March 2002, <<https://www.rfc-editor.org/info/rfc3246>>.
- [RFC3261] Rosenberg, J., Schulzrinne, H., Camarillo, G., Johnston, A., Peterson, J., Sparks, R., Handley, M., and E. Schooler, "SIP: Session Initiation Protocol", RFC 3261, DOI 10.17487/RFC3261, June 2002, <<https://www.rfc-editor.org/info/rfc3261>>.
- [RFC4594] Babiarz, J., Chan, K., and F. Baker, "Configuration Guidelines for DiffServ Service Classes", RFC 4594, DOI 10.17487/RFC4594, August 2006, <<https://www.rfc-editor.org/info/rfc4594>>.
- [RFC5245] Rosenberg, J., "Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols", RFC 5245, DOI 10.17487/RFC5245, April 2010, <<https://www.rfc-editor.org/info/rfc5245>>.
- [RFC7478] Holmberg, C., Hakansson, S., and G. Eriksson, "Web Real-Time Communication Use Cases and Requirements", RFC 7478, DOI 10.17487/RFC7478, March 2015, <<https://www.rfc-editor.org/info/rfc7478>>.

- [RFC7556] Anipko, D., Ed., "Multiple Provisioning Domain Architecture", RFC 7556, DOI 10.17487/RFC7556, June 2015, <<https://www.rfc-editor.org/info/rfc7556>>.
- [RFC7657] Black, D., Ed. and P. Jones, "Differentiated Services (Diffserv) and Real-Time Communication", RFC 7657, DOI 10.17487/RFC7657, November 2015, <<https://www.rfc-editor.org/info/rfc7657>>.
- [RFC8084] Fairhurst, G., "Network Transport Circuit Breakers", BCP 208, RFC 8084, DOI 10.17487/RFC8084, March 2017, <<https://www.rfc-editor.org/info/rfc8084>>.
- [RFC8095] Fairhurst, G., Ed., Trammell, B., Ed., and M. Kuehlewind, Ed., "Services Provided by IETF Transport Protocols and Congestion Control Mechanisms", RFC 8095, DOI 10.17487/RFC8095, March 2017, <<https://www.rfc-editor.org/info/rfc8095>>.
- [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>>.

Appendix A. Convenience Functions

A.1. Adding Preference Properties

As Selection Properties of type Preference will be added to a TransportProperties object quite frequently, implementations should provide special actions for adding each preference level i.e, "TransportProperties.Add(some_property, avoid)" is equivalent to "TransportProperties.Avoid(some_property)":

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

A.2. Transport Property Profiles

To ease the use of the interface specified by this document, implementations should provide a mechanism to create Transport Property objects (see Section 5.2) that are pre-configured with

frequently used sets of properties. Implementations should at least short-hands to specify the following property profiles:

A.2.1. reliable-inorder-stream

This profile provides reliable, in-order transport service with congestion control. An example of a protocol that provides this service is TCP. It should consist of the following properties:

Property	Value
reliability	require
preserve-order	require
congestion-control	require
preserve-msg-boundaries	ignore

A.2.2. reliable-message

This profile provides message-preserving, reliable, in-order transport service with congestion control. An example of a protocol that provides this service is SCTP. It should consist of the following properties:

Property	Value
reliability	require
preserve-order	require
congestion-control	require
preserve-msg-boundaries	require

A.2.3. unreliable-datagram

This profile provides unreliable datagram transport service. An example of a protocol that provides this service is UDP. It should consist of the following properties:

Property	Value
reliability	ignore
preserve-order	ignore
congestion-control	ignore
preserve-msg-boundaries	require
idempotent	true

Applications that choose this Transport Property Profile for latency reasons should also consider setting the Capacity Profile Property, see Section 11.1.10 accordingly and my benefit from controlling checksum coverage, see Section 5.2.7 and Section 5.2.8.

Appendix B. 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.

B.1. Experimental Transport Properties

The following Transport Properties might be made available in addition to those specified in Section 5.2, Section 11.1, and Section 7.4.

B.1.1.1. Cost Preferences

[EDITOR'S NOTE: At IETF 103, opinions were that this property should stay, but it was also said that this is maybe not "on the right level". If / when moving it to the main text, note that this is meant to be applicable to a Preconnection or a Message.]

Name: cost-preferences

Type: Enumeration

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

Possible values are:

No Expense: Avoid transports associated with monetary cost

Optimize Cost: Prefer inexpensive transports and accept service degradation

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

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

The default is "Balance Cost".

Appendix C. Sample API definition in Go

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

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

[I-D.ietf-taps-minset] identifies a minimal set of transport services that end systems should offer. These services make all 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 [I-D.ietf-taps-minset], updated according to the discussion in Section 5 of [I-D.ietf-taps-minset]. This list is a subset of the transport features in Appendix A of [I-D.ietf-taps-minset], 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.

[EDITOR'S NOTE: This is early text. In the future, this section will contain backward references, which we currently avoid because things are still being moved around and names / categories etc. are changing.]

- o Connect: "Initiate" Action.
- o Listen: "Listen" Action.
- o Specify number of attempts and/or timeout for the first establishment message: "timeout" parameter of "Initiate" or "InitiateWithSend" Action.
- o Disable MPTCP: "Parallel Use of Multiple Paths" Property.
- o Hand over a message to reliably transfer (possibly multiple times) before connection establishment: "InitiateWithSend" Action.
- o Change timeout for aborting connection (using retransmit limit or time value): "Timeout for Aborting Connection" property, using a time value.
- o Timeout event when data could not be delivered for too long: "ConnectionError" Event.
- o Suggest timeout to the peer: "TCP-specific Property: User Timeout".
- o Notification of Excessive Retransmissions (early warning below abortion threshold): "Notification of excessive retransmissions" property.
- o Notification of ICMP error message arrival: "Notification of ICMP soft error message arrival" property.
- o Choose a scheduler to operate between streams of an association: "Connection Group Transmission Scheduler" property.

- o Configure priority or weight for a scheduler: "Priority (Connection)" property.
- o "Specify checksum coverage used by the sender" and "Disable checksum when sending": "Corruption Protection Length" property and "Full Checksum Coverage on Sending" property.
- o "Specify minimum checksum coverage required by receiver" and "Disable checksum requirement when receiving": "Required Minimum Corruption Protection Coverage for Receiving" property and "Full Checksum Coverage on Receiving" property.
- o "Specify DF" field and "Request not to bundle messages:" The "Singular Transmission" Message property combines both of these requests, i.e. if a request not to bundle messages is made, this also turns off DF in case of protocols that allow this (only UDP and UDP-Lite, which cannot bundle messages anyway).
- o Get max. transport-message size that may be sent using a non-fragmented IP packet from the configured interface: "Maximum Message Size Before Fragmentation or Segmentation" property.
- o Get max. transport-message size that may be received from the configured interface: "Maximum Message Size on Receive" property.
- o Obtain ECN field: "ECN" is a defined read-only Message Property of the MessageContext object.
- o "Specify DSCP field", "Disable Nagle algorithm", "Enable and configure a 'Low Extra Delay Background Transfer'": As suggested in Section 5.5 of [I-D.ietf-taps-minset], these transport features are collectively offered via the "Capacity Profile" property.
- o Close after reliably delivering all remaining data, causing an event informing the application on the other side: This is offered by the "Close" Action with slightly changed semantics in line with the discussion in Section 5.2 of [I-D.ietf-taps-minset].
- o "Abort without delivering remaining data, causing an event informing the application on the other side" and "Abort without delivering remaining data, not causing an event informing the application on the other side": This is offered by the "Abort" action without promising that this is signaled to the other side. If it is, a "ConnectionError" Event will fire at the peer.
- o "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.

Reliability is controlled via the "Reliable Data Transfer (Message)" Message property. Transmitting data as a message or without delimiters is controlled via Message Framers. The choice of congestion control is provided via the "Congestion control" property.

- o Configurable Message Reliability: The "Lifetime" Message Property implements a time-based way to configure message reliability.
- o "Ordered message delivery (potentially slower than unordered)" and "Unordered message delivery (potentially faster than ordered)": The two transport features are controlled via the Message Property "Ordered".
- o Request not to delay the acknowledgement (SACK) of a message: Should the protocol support it, this is one of the transport features the transport system can use when an application uses the "Capacity Profile" Property with value "Low Latency/Interactive".
- o Receive data (with no message delimiting): "Received" Event without using a Message Framers.
- o Receive a message: "Received" Event, using Message Framers.
- o Information about partial message arrival: "ReceivedPartial" Event.
- o Notification of send failures: "Expired" and "SendError" Events.
- o Notification that the stack has no more user data to send: Applications can obtain this information via the "Sent" Event.
- o Notification to a receiver that a partial message delivery has been aborted: "ReceiveError" Event.

Authors' Addresses

Brian Trammell (editor)
Google
Gustav-Gull-Platz 1
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
TU Berlin
Marchstrasse 23
10587 Berlin
Germany

Email: theresa@inet.tu-berlin.de

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

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

Mirja Kuehlewind
ETH Zurich
Gloriastrasse 35
8092 Zurich
Switzerland

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

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

Email: csp@csp Perkins.org

Philipp S. Tiesel
TU Berlin
Einsteinufer 25
10587 Berlin
Germany

Email: philipp@tiesel.net

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

Email: cawood@apple.com

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: March 31, 2020

C. Wood, Ed.
Apple Inc.
T. Enghardt
TU Berlin
T. Pauly
Apple Inc.
C. Perkins
University of Glasgow
K. Rose
Akamai Technologies, Inc.
September 28, 2019

A Survey of Transport Security Protocols
draft-ietf-taps-transport-security-09

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. Moreover, this document defines a minimal set of security features that a secure transport system should provide.

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 March 31, 2020.

Copyright Notice

Copyright (c) 2019 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	6
4. Transport Security Protocol Descriptions	7
4.1. TLS	7
4.1.1. Protocol Description	8
4.1.2. Security Features	9
4.1.3. Protocol Dependencies	9
4.2. DTLS	9
4.2.1. Protocol Description	10
4.2.2. Security Features	10
4.2.3. Protocol Dependencies	10
4.3. QUIC with TLS	11
4.3.1. Protocol Description	11
4.3.2. Security Features	12
4.3.3. Protocol Dependencies	12
4.3.4. Variant: Google QUIC	12
4.4. IKEv2 with ESP	12
4.4.1. IKEv2 Protocol Description	12
4.4.2. ESP Protocol Description	13
4.4.3. IKEv2 Security Features	14
4.4.4. ESP Security Features	14
4.4.5. IKEv2 Protocol Dependencies	14
4.4.6. ESP Protocol Dependencies	15
4.5. Secure RTP (with DTLS)	15
4.5.1. Protocol description	15
4.5.2. Security Features	16
4.5.3. Protocol Dependencies	16
4.5.4. Variant: ZRTP for Media Path Key Agreement	17
4.6. tcpcrypt	17
4.6.1. Protocol Description	17

4.6.2. Security Features	18
4.6.3. Protocol Dependencies	18
4.7. WireGuard	18
4.7.1. Protocol description	19
4.7.2. Security Features	19
4.7.3. Protocol Dependencies	20
4.8. CurveCP	20
4.8.1. Protocol Description	20
4.8.2. Protocol Features	21
4.8.3. Protocol Dependencies	21
4.9. MinimalT	22
4.9.1. Protocol Description	22
4.9.2. Protocol Features	23
4.9.3. Protocol Dependencies	23
4.10. OpenVPN	23
4.10.1. Protocol Description	23
4.10.2. Protocol Features	24
4.10.3. Protocol Dependencies	25
5. Security Features and Application Dependencies	25
5.1. Mandatory Features	25
5.2. Optional Features	26
5.3. Optional Feature Availability	27
6. Transport Security Protocol Interfaces	29
6.1. Pre-Connection Interfaces	29
6.2. Connection Interfaces	30
6.3. Post-Connection Interfaces	30
7. IANA Considerations	31
8. Security Considerations	31
9. Privacy Considerations	31
10. Acknowledgments	31
11. Informative References	31
Authors' Addresses	36

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 services and features a Transport Services system (a system that provides a transport API) needs to provide in order to add transport security. It examines Transport Layer Security (TLS), Datagram Transport Layer Security (DTLS), QUIC + TLS, tcpcrypt, Internet Key Exchange with Encapsulating Security Protocol (IKEv2 + ESP), SRTP (with DTLS), WireGuard, CurveCP, and MinimalT. 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. The document groups

these security features into a minimal set of features, which every secure transport system should provide in addition to the transport features described in [I-D.ietf-taps-minset], and additional optional features, which may not be available in every secure transport system. Finally, the document distills the application and transport interfaces provided by the transport security protocols.

Selected protocols 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 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.

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. Such protocols are thus omitted from this 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. Security Features extend the set of Transport Features described in [RFC8095] and provided by Transport Services implementations.
- 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: Framed protocol messages.
- 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 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 (and privacy) 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 session key establishment: Establishing cryptographic keys with forward-secure properties.
- o Cryptographic algorithm negotiation: Negotiating support of protocol algorithms, including algorithms for encryption, hashing, MAC (PRF), and digital signatures.
- o Session caching and management: Managing session state caches used for subsequent connections, with the aim of amortizing connection establishment costs.
- o Peer authentication: Authenticating peers using generic or protocol-specific mechanisms, such as certificates, raw public keys, pre-shared keys, or EAP methods.
- o Unilateral responder authentication: Requiring authentication for the responder of a connection.
- o Mutual authentication: Establishing connections in which both endpoints are authenticated.
- o Application authentication delegation: Delegating to applications out-of-band to perform peer authentication.
- o Record (channel or datagram) confidentiality and integrity: Encrypting and authenticating application plaintext bytes sent between peers over a channel or in individual datagrams.
- o Partial record confidentiality: Encrypting some portion of records.
- o Optional record integrity: Optionally authenticating certain records.
- o Record replay prevention: Detecting and defending against record replays, which can be due to in-network retransmissions.
- o Early data support: Transmitting application data prior to secure connection establishment via a handshake. For TLS, this support begins with TLS 1.3.

- o Connection mobility: Allowing a connection to be multihomed or resilient across network interface or address changes, such as NAT rebindings that occur without an endpoint's knowledge. Mobility allows cryptographic key material and other state information to be reused in the event of a connection change.
- o Application-layer feature negotiation: Securely negotiating application-specific functionality. Such features may be necessary for further application processing, such as the TLS parent connection protocol type via ALPN [RFC7301] or desired application identity via SNI [RFC6066].
- o Configuration extensions: Adding protocol features via extensions or configuration options. TLS extensions are a primary example of this feature.
- o Out-of-order record receipt: Processing of records received out-of-order.
- o Source validation (cookie or puzzle based): Validating peers and mitigating denial-of-service (DoS) attacks via explicit proof of origin (cookies) or work mechanisms (puzzles).
- o Length-hiding padding: Adding padding to records in order to hide plaintext message length and mitigate 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) [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 (possibly encrypted) data from one peer to the other. This data may contain handshake messages or raw application data.

4.1.1.1. Protocol Description

TLS is the composition of a handshake and record protocol [RFC8446]. 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.

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] and record size limit [RFC8449].

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. Security Features

- o Forward-secure session key establishment.
- o Cryptographic algorithm negotiation.
- o Stateful and stateless cross-connection session resumption.
- o Session caching and management.
- o Peer authentication (Certificate, raw public key, and pre-shared key).
- o Unilateral responder authentication.
- o Mutual authentication.
- o Application authentication delegation.
- o Record (channel) confidentiality and integrity.
- o Record replay prevention.
- o Application-layer feature negotiation.
- o Configuration extensions.
- o Early data support (starting with TLS 1.3).
- o Optional record-layer padding (starting with TLS 1.3).

4.1.3. Protocol Dependencies

- o In-order, reliable bytestream 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 unreliable datagram protocols like UDP instead of TCP. DTLS modifies the protocol to make sure it can still provide the same security guarantees as TLS even without reliability from the transport. 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.

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, or a similar mechanism in other datagram transports. 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. Security 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 DTLS relies on an unreliable datagram transport.

- 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 Uniqueness of the session within the transport flow (only one DTLS connection on a UDP 4-tuple, for example); or else support for the connection identifier extension to enable demultiplexing.
- o Path MTU discovery.
- o For the handshake: Reliable, in-order transport. DTLS provides its own reliability.

4.3. QUIC with TLS

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 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 of 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. QUIC implements a record protocol for TLS handshake messages to establish a connection. These messages are sent in CRYPTO frames [I-D.ietf-quic-transport] in Initial and Handshake packets. Initial packets are encrypted using fixed keys derived from the QUIC version and public packet information (Connection ID). Handshake packets are encrypted using TLS handshake secrets. Once TLS completes, QUIC uses the resulting traffic secrets to for the

QUIC connection to protect the rest of the frames. QUIC supports 0-RTT data using previously negotiated connection secrets. Early data is sent in 0-RTT packets, which may be included in the same datagram as the Initial and Handshake packets.

4.3.2. Security Features

- o DoS mitigation (cookie-based).

See also the properties of TLS.

4.3.3. Protocol Dependencies

- o QUIC transport assumes an unreliable transport, e.g., UDP.
- o QUIC transport relies on TLS 1.3 for key exchange, peer authentication, and shared secret derivation.
- o For the handshake: Reliable, in-order transport. QUIC provides its own reliability.

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 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. IKEv2 Protocol Description

IKEv2 is a control protocol that runs on UDP ports 500 or 4500 and TCP port 4500. 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 4.4.2. 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. IKE then performs a phase of authentication in which both peers present blobs signed by a shared secret or private key that authenticates the entire IKE exchange and the IKE identities. IKE then derives further sets of keys on demand, which together with traffic policies are 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 specifies an encryption and authentication algorithm, or an AEAD 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 re-keyed before 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 outer IP addresses and interfaces [RFC4555].

When UDP is not available or well-supported on a network, IKEv2 may be encapsulated in TCP [RFC8229].

4.4.2. ESP Protocol Description

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].

4.4.3. IKEv2 Security Features

- o Forward-secure session key establishment.
- o Cryptographic algorithm negotiation.
- o Peer authentication (certificate, raw public key, pre-shared key, and EAP).
- o Unilateral responder authentication.
- o Mutual authentication.
- o Record (datagram) confidentiality and integrity.
- o Session resumption.
- o Connection mobility.
- o DoS mitigation (cookie-based).

4.4.4. ESP Security Features

- o Record confidentiality and integrity.
- o Record replay protection.

4.4.5. IKEv2 Protocol Dependencies

- 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.

4.4.6. ESP Protocol Dependencies

- 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 signaling exchange (e.g., in SIP or WebRTC), and used to bind the DTLS key exchange in the media plane to the signaling plane. The combination of a mutually authenticated DTLS key exchange on the media path and a fingerprint sent in the signaling channel protects against active attacks on the media, provided the signaling can be trusted. Signaling needs to be protected as described in, for example, SIP [RFC3261] Authenticated Identity Management [RFC8224] or the WebRTC security architecture [I-D.ietf-rtcweb-security-arch], to provide complete system security.

4.5.2. Security Features

- o Forward-secure session 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 Secure RTP can run over UDP or TCP.
- o External key derivation and management protocol, e.g., DTLS [RFC5763].
- o External identity management protocol, e.g., SIP Authenticated Identity Management [RFC8224], 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 [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.

4.6.1. Protocol Description

Tcpcrypt extends TCP to enable opportunistic encryption between the two ends of a TCP connection [RFC8548]. 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) [RFC8547]. 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.

4.6.2. Security Features

- o Forward-secure session key establishment.
- o Record (channel) confidentiality and integrity.
- o Stateful cross-connection session resumption.
- o Session caching and management.
- o Application authentication delegation.

4.6.3. Protocol Dependencies

- o TCP for in-order, reliable transport.
- o TCP Encryption Negotiation Option (ENO).

4.7. WireGuard

WireGuard is a layer 3 protocol designed as an alternative to IPsec [WireGuard] for certain use cases. It uses UDP to encapsulate IP datagrams between peers. 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 [RFC8439] 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.

If a WireGuard receiver is under heavy load and cannot process a packet, e.g., cannot spare CPU cycles for expensive public key cryptographic operations, 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. Security Features

- o Forward-secure session key establishment.
- o Peer authentication (public-key and PSK).
- o Mutual authentication.
- o Record replay prevention (Stateful, timestamp-based).
- o Connection mobility.
- o DoS mitigation (cookie-based).

4.7.3. Protocol Dependencies

- o Datagram transport.
- o Out-of-band key distribution and management.

4.8. 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.

4.8.1. 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 public-key encryption and authentication, or "boxing", simplifies problems that come with symmetric key management and nonce 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.

Servers use cookies for source validation. After receiving a client's initial packet, encrypted under the server's long-term public key, a 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 client authentication by allowing clients 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, connection ID, and per-message nonce in the clear. All other data is encrypted.

4.8.2. Protocol Features

- o Datagram confidentiality and integrity (via public key encryption).
- o Peer authentication (public-key).
- o Unilateral responder authentication.
- o Mutual authentication.
- o Connection mobility (based on a client-chosen ephemeral identifier).
- o Optional length-hiding and anti-amplification padding.
- o Source validation (cookie-based)

4.8.3. Protocol Dependencies

- o An unreliable transport protocol such as UDP.

4.9. 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.

4.9.1. 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) publish 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 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.

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.

4.9.2. Protocol Features

- o Record or datagram confidentiality and integrity.
- o Forward-secure session key establishment.
- o Peer authentication (public-key).
- o Unilateral responder authentication.
- o DoS mitigation (puzzle-based).
- o Out-of-order receipt record.
- o Connection mobility (based on tunnel identifiers).

4.9.3. Protocol Dependencies

- o An unreliable transport protocol such as UDP.
- o A DNS-like resolution service to obtain location information (an IP address) and ephemeral keys.
- o A PKI trust store for certificate validation.

4.10. 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 UDP or TCP.

4.10.1. Protocol Description

OpenVPN facilitates authentication using either a pre-shared static key or using X.509 certificates and TLS. In pre-shared key mode, OpenVPN derives keys for encryption and authentication directly from one or multiple symmetric keys. In TLS mode, OpenVPN encapsulates a TLS handshake, in which both peers must present a certificate for authentication. After the handshake, both sides contribute random source material to derive keys for encryption and authentication using the TLS pseudo random function (PRF). OpenVPN provides the possibility to authenticate and encrypt the TLS handshake itself using a pre-shared key or passphrase. Furthermore, it supports re-keying using TLS.

After authentication and key exchange, OpenVPN encrypts payload data, i.e., IP packets or Ethernet frames, and authenticates the payload using HMAC. Applications can select an arbitrary encryption algorithm (cipher) and key size, as well hash function for HMAC. The default cipher and hash functions are AES-GCM and SHA1, respectively. Recent versions of the protocol support cipher negotiation.

OpenVPN can run over TCP or UDP. When running over UDP, OpenVPN provides a simple reliability layer for control packets such as the TLS handshake and key exchange. It assigns sequence numbers to packets, acknowledges packets it receives, and retransmits packets it deems lost. Similar to DTLS, this reliability layer is not used for data packets, which prevents the problem of two reliability mechanisms being encapsulated within each other. When running over TCP, OpenVPN includes the packet length in the header, which allows the peer to deframe the TCP stream into messages.

For replay protection, OpenVPN assigns an identifier to each outgoing packet, which is unique for the packet and the currently used key. In pre-shared key mode or with a CFB or OFB mode cipher, OpenVPN combines a timestamp with an incrementing sequence number into a 64-bit identifier. In TLS mode with CBC cipher mode, OpenVPN omits the timestamp, so identifiers are only 32-bit. This is sufficient since OpenVPN can guarantee the uniqueness of this identifier for each key, as it can trigger re-keying if needed.

OpenVPN supports connection mobility by allowing a peer to change its IP address during an ongoing session. When configured accordingly, a host will accept authenticated packets for a session from any IP address.

4.10.2. Protocol Features

- o Peer authentication using certificates or pre-shared key.
- o Mandatory mutual authentication.
- o Connection mobility.
- o Out-of-order record receipt.
- o Length-hiding padding.

See also the properties of TLS.

4.10.3. Protocol Dependencies

- o For control packets such as handshake and key exchange: Reliable, in-order transport. Reliability is provided either by TCP, or by OpenVPN's own reliability layer when using UDP.

5. Security Features and Application 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 Transport Services implementation. They were selected on the basis that they are either (a) required for any secure transport protocol or (b) nearly ubiquitous amongst common secure transport protocols.

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 or application dependency is unavailable.

In this context, an application dependency is an aspect of the security feature which can be exposed to the application. An application dependency may be required for the security feature to function, or it may provide additional information and control to the application. For example, an application may need to provide information such as keying material or authentication credentials, or it may want to restrict which cryptographic algorithms to allow for negotiation.

5.1. Mandatory Features

Mandatory features must be supported regardless of transport and application services available. Note that not all mandatory features are provided by each surveyed protocol above. For example, `tcpcrypt` does not provide responder authentication and `CurveCP` does not provide forward-secure session key establishment.

- o Record or datagram confidentiality and integrity.
 - * Application dependency: None.
- o Forward-secure session key establishment.
 - * Application dependency: None.

- o Unilateral responder authentication.
 - * (Optional) Application dependency: Application-provided trust information. System trust stores may also be used to authenticate responders.

5.2. Optional Features

In this section we list optional features along with their necessary application dependencies, if any.

- o Pre-shared key support (PSK):
 - * Application dependency: Application provisioning and distribution of pre-shared keys.
- o Mutual authentication (MA):
 - * Application dependency: Mutual authentication credentials required.
- o Cryptographic algorithm negotiation (AN):
 - * Application dependency: Application awareness of supported or desired algorithms.
- o Application authentication delegation (AD):
 - * Application dependency: Application opt-in and policy for endpoint authentication.
- o DoS mitigation (DM):
 - * Application dependency: None.
- o Connection mobility (CM):
 - * Application dependency: None.
- o Source validation (SV):
 - * Application dependency: None.
- o Application-layer feature negotiation (AFN):
 - * Application dependency: Specification of application-layer features or functionality.

- o Configuration extensions (CX):
 - * Application dependency: Specification of application-specific extensions.
- o Session caching and management (SC):
 - * Application dependency: None.
- o Length-hiding padding (LHP): (Optional) Application dependency: Knowledge of desired padding policies. Some protocols, such as IKE, can negotiate application-agnostic padding policies.
- o Early data support (ED):
 - * Application dependency: Anti-replay protections or hints of data idempotency.
- o Record replay prevention (RP):
 - * Application dependency: None.
- o Out-of-order receipt record (OO):
 - * Application dependency: None.

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.

Pro toc ol	P S K	A N	A D	M A	D M	C M	S V	A F N	CX	SC	LH P	ED	RP	OO
TLS	S	S	S	S	S	U *	M	S	S	S	S	S	U	U
DTLS	S	S	S	S	S	S	M	S	S	S	S	U	M	M
QUIC	S	S	S	S	S	S	M	S	S	S	S	S	M	M
IKE v2+ ESP	S	S	S	M	S	S	M	S	S	S	S	U	M	M
SRT P+D TLS	S	S	S	S	S	U	M	S	S	S	U	U	M	M
tcp crypt	U	S	M	U	U * *	U *	M	U	U	S	U	U	U	U
WireGuard	S	U	S	M	S	U	M	U	U	U	S+	U	M	M
MinimalT	U	U	U	M	S	M	M	U	U	U	S	U	U	U
CurveCP	U	U	U	S	S	M	M	U	U	U	S	U	M	M

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, following conventions in [I-D.ietf-taps-interface] and [I-D.ietf-taps-arch].

6.1. Pre-Connection Interfaces

Configuration interfaces are used to configure the security protocols before a handshake begins or the keys are negotiated.

- o Identities 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 Extensions (Application-Layer Protocol Negotiation): The application enables or configures extensions that are to be negotiated by the security protocol, such as ALPN [RFC7301]. Protocols: TLS, DTLS, QUIC + TLS
- 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, tcpcrypt, IKEv2, MinimalT
- 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, tcpcrypt, IKEv2, MinimalT
- 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 Mobility Events 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).
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. Privacy Considerations

Analysis of how features improve or degrade privacy is intentionally omitted from this survey. All security protocols surveyed generally improve privacy by reducing information leakage via encryption. 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.

10. 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.

11. 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/>>.

- [BLAKE2] Aumasson, J., Neves, S., Wilcox-O'Hearn, Z., and C. Winnerlein, "BLAKE2 -- simpler, smaller, fast as MD5", <<https://blake2.net/blake2.pdf>>.
- [Curve25519] Bernstein, D., "Curve25519 - new Diffie-Hellman speed records", <<https://cr.yp.to/ecdh/curve25519-20060209.pdf>>.
- [CurveCP] Bernstein, D., "CurveCP -- Usable security for the Internet", <<http://curvecp.org>>.
- [I-D.ietf-quic-tls] Thomson, M. and S. Turner, "Using TLS to Secure QUIC", draft-ietf-quic-tls-23 (work in progress), September 2019.
- [I-D.ietf-quic-transport] Iyengar, J. and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport", draft-ietf-quic-transport-23 (work in progress), September 2019.
- [I-D.ietf-rtcweb-security-arch] Rescorla, E., "WebRTC Security Architecture", draft-ietf-rtcweb-security-arch-20 (work in progress), July 2019.
- [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-04 (work in progress), July 2019.
- [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", draft-ietf-taps-interface-04 (work in progress), July 2019.
- [I-D.ietf-taps-minset] Welzl, M. and S. Gjessing, "A Minimal Set of Transport Services for End Systems", draft-ietf-taps-minset-11 (work in progress), September 2018.
- [I-D.ietf-tls-dtls-connection-id] Rescorla, E., Tschofenig, H., and T. Fossati, "Connection Identifiers for DTLS 1.2", draft-ietf-tls-dtls-connection-id-06 (work in progress), July 2019.

- [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-dtls13-32 (work in progress), July 2019.
- [MinimalT]
Petullo, W., Zhang, X., Solworth, J., Bernstein, D., and T. Lange, "MinimalT -- Minimal-latency Networking Through Better Security",
<<http://dl.acm.org/citation.cfm?id=2516737>>.
- [Noise]
Perrin, T., "The Noise Protocol Framework",
<<http://noiseprotocol.org/noise.pdf>>.
- [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>>.
- [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]
Baughen, 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>>.
- [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>>.
- [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>>.
- [RFC8224] Peterson, J., Jennings, C., Rescorla, E., and C. Wendt, "Authenticated Identity Management in the Session Initiation Protocol (SIP)", RFC 8224, DOI 10.17487/RFC8224, February 2018, <<https://www.rfc-editor.org/info/rfc8224>>.
- [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>>.

- [RFC8439] Nir, Y. and A. Langley, "ChaCha20 and Poly1305 for IETF Protocols", RFC 8439, DOI 10.17487/RFC8439, June 2018, <<https://www.rfc-editor.org/info/rfc8439>>.
- [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>>.
- [RFC8449] Thomson, M., "Record Size Limit Extension for TLS", RFC 8449, DOI 10.17487/RFC8449, August 2018, <<https://www.rfc-editor.org/info/rfc8449>>.
- [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>>.
- [SIGMA] Krawczyk, H., "SIGMA -- The 'SIGn-and-MAC' Approach to Authenticated Diffie-Hellman and Its Use in the IKE-Protocols", <<http://www.iacr.org/cryptodb/archive/2003/CRYPTO/1495/1495.pdf>>.
- [WireGuard] Donenfeld, J., "WireGuard -- Next Generation Kernel Network Tunnel", <<https://www.wireguard.com/papers/wireguard.pdf>>.

Authors' Addresses

Christopher A. Wood (editor)
Apple Inc.
One Apple Park Way
Cupertino, California 95014
United States of America

Email: cawood@apple.com

Theresa Enhardt
TU Berlin
Marchstr. 23
10587 Berlin
Germany

Email: theresa@inet.tu-berlin.de

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

Email: tpauly@apple.com

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

Email: csp@csp Perkins.org

Kyle Rose
Akamai Technologies, Inc.
150 Broadway
Cambridge, MA 02144
United States of America

Email: krose@krose.org