TAPS Working Group
Internet-Draft
Intended status: Standards Track
Expires: 12 May 2024

T. Pauly, Ed.
Apple Inc.
B. Trammell, Ed.
Google Switzerland GmbH
A. Brunstrom
Karlstad University
G. Fairhurst
University of Aberdeen
C. Perkins
University of Glasgow
9 November 2023

Architecture and Requirements for Transport Services
draft-ietf-taps-arch-19

Abstract

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

Status of This Memo

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

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

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

This Internet-Draft will expire on 12 May 2024.

Table of Contents

1. Introduction

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

The goal of the Transport Services System architecture is to provide a flexible and reusable system with a common interface for transport protocols. An application uses the Transport Services System through an abstract Connection (we use capitalization to distinguish these from the underlying connections of, e.g., TCP). This provides flexible connection establishment allowing an application to request or require a set of properties.

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.

This architecture can also support applications by offering racing mechanisms (attempting multiple IP addresses, protocols, or network paths in parallel), which otherwise need to be implemented in each application separately (see Section 4.2.2). Racing selects one or more candidates each with equivalent protocol stacks that are used to identify an optimal combination of transport protocol instance such as TCP, UDP, or another transport, together with configuration of parameters and interfaces. A Connection represents an object that,

once established, can be used to send and receive messages. A Connection can also be created from another Connection, by cloning, and then forms a part of a Connection Group whose Connections share properties.

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

1.1. Background

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

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

One of the key insights to come from identifying the minimal set of features provided by transport protocols [RFC8923] was that features either require application interaction and guidance (referred to in that document as Functional or Optimizing Features), or else can be handled automatically by an implementation of the Transport Services System (referred to as Automatable Features). Among the identified Functional and Optimizing Features, some are common across all or nearly all transport protocols, while others present 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. 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 System in three sections:

* Section 2 describes how the Transport Services API model differs from that of traditional socket-based APIs. Specifically, it offers asynchronous event-driven interaction, the use of messages for data transfer, and the flexibility to use different transport protocols and paths without requiring major changes to the application.

* Section 3 explains the fundamental requirements for a Transport Services System. These principles are intended to make sure that transport protocols can continue to be enhanced and evolve without requiring significant changes by application developers.

* Section 4 presents the Transport Services Implementation and defines the concepts that are used by the API [I-D.ietf-taps-interface] and described in the implementation guidelines [I-D.ietf-taps-impl]. This introduces the Preconnection, which allows applications to configure Connection Properties.

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.

1.4. Glossary of Key Terms

This subsection provides a glossary of key terms related to the Transport Services architecture. It provides a short description of key terms that are later defined in this document.

* Application: An entity that uses the transport layer for end-to-end delivery of data across the network [RFC8095].

* Cached State: The state and history that the Transport Services Implementation keeps for each set of the associated Endpoints that have been used previously.

* Candidate Path: One path that is available to an application and conforms to the Selection Properties and System Policy during racing.

* Candidate Protocol Stack: One Protocol Stack that can be used by an application for a Connection during racing.

* Client: The peer responsible for initiating a Connection.

* Clone: A Connection that was created from another Connection, and forms a part of a Connection Group.

* Connection: Shared state of two or more Endpoints that persists across Messages that are transmitted and received between these Endpoints [RFC8303]. When this document (and other Transport Services documents) use the capitalized "Connection" term, it refers to a Connection object that is being offered by the Transport Services system, as opposed to more generic uses of the word "connection".

* Connection Context: A set of stored properties across Connections, such as cached protocol state, cached path state, and heuristics, which can include one or more Connection Groups.

* Connection Group: A set of Connections that share properties and caches.

* Connection Property: A Transport Property that controls per-Connection behavior of a Transport Services implementation.

* Endpoint: An entity that communicates with one or more other endpoints using a transport protocol.

* Endpoint Identifier: An identifier that specifies one side of a Connection (local or remote), such as a hostname or URL.

* Equivalent Protocol Stacks: Protocol Stacks that can be safely swapped or raced in parallel during establishment of a Connection.

* Event: A primitive that is invoked by an Endpoint [RFC8303].

* Framer: A data translation layer that can be added to a Connection to define how application-layer Messages are transmitted over a Protocol Stack.

* Local Endpoint: The local Endpoint.

* Local Endpoint Identifier: A representation of the application's identifier for itself that it uses for a Connection.

* Message: A unit of data that can be transferred between two Endpoints over a Connection.

* Message Property: A property that can be used to specify details about Message transmission, or obtain details about the transmission after receiving a Message.

* Parameter: A value passed between an application and a transport protocol by a primitive [RFC8303].

* Path: A representation of an available set of properties that a Local Endpoint can use to communicate with a Remote Endpoint.

* Peer: An Endpoint application party to a Connection.

* Preconnection: an object that represents a Connection that has not yet been established.

* Preference: A preference to prohibit, avoid, ignore, prefer, or require a specific Transport Feature.

* Primitive: A function call that is used to locally communicate between an application and an Endpoint, which is related to one or more Transport Features [RFC8303].

* Protocol Instance: A single instance of one protocol, including any state necessary to establish connectivity or send and receive Messages.

* Protocol Stack: A set of Protocol Instances that are used together to establish connectivity or send and receive Messages.

* Racing: The attempt to select between multiple Protocol Stacks based on the Selection and Connection Properties communicated by the application, along with any Security Parameters.

* Remote Endpoint: The peer that a local Endpoint can communicate with when a Connection is established.

* Remote Endpoint Identifier: A representation of the application's identifier for a peer that can participate in establishing a Connection.

* Rendezvous: The action of establishing a peer-to-peer Connection with a Remote Endpoint.

* Security Parameters: Parameters that define an application's requirements for authentication and encryption on a Connection.

* Server: The peer responsible for responding to a Connection initiation.

* Socket: The combination of a destination IP address and a destination port number [RFC8303].

* System Policy: 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 and race the candidates during establishment of a Connection.

* Selection Property: A Transport Property that can be set to influence the selection of paths between the Local and Remote Endpoints.

* Transport Feature: A specific end-to-end feature that the transport layer provides to an application.

* Transport Property: A property that expresses requirements, prohibitions and preferences [RFC8095].

* Transport Service: A set of transport features, without an association to any given framing protocol, that provides a complete service to an application.

* Transport Services Implementation: This consists of all objects and protocol instances used internally to a system or library to implement the functionality needed to provide a transport service across a network, as required by the abstract interface.

* Transport Services System: The Transport Services Implementation and the Transport Services API.

2. API Model

The traditional model of using sockets can be represented as follows (see figure 1):

* Applications create connections and transfer data using the Socket API.

* The Socket API provides the interface to the implementations of TCP and UDP (typically implemented in the system's kernel).

* TCP and UDP in the kernel send and receive data over the available network-layer interfaces.

* Sockets are bound directly to transport-layer and network-layer addresses, obtained via a separate resolution step, usually performed by a system-provided DNS stub resolver.

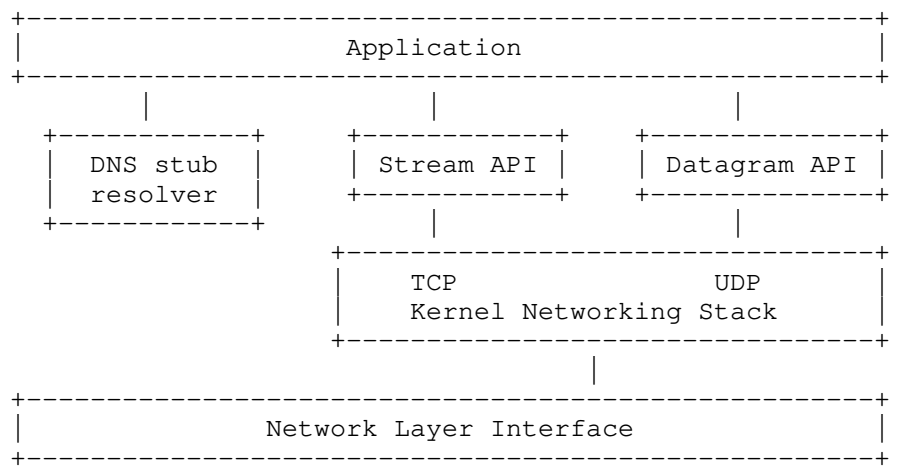


Figure 1: Socket API Model

The architecture of the Transport Services System is an evolution of this general model of interaction. It both modernizes the API presented to applications by the transport layer and enriches the capabilities of the Transport Services Implementation below this API.

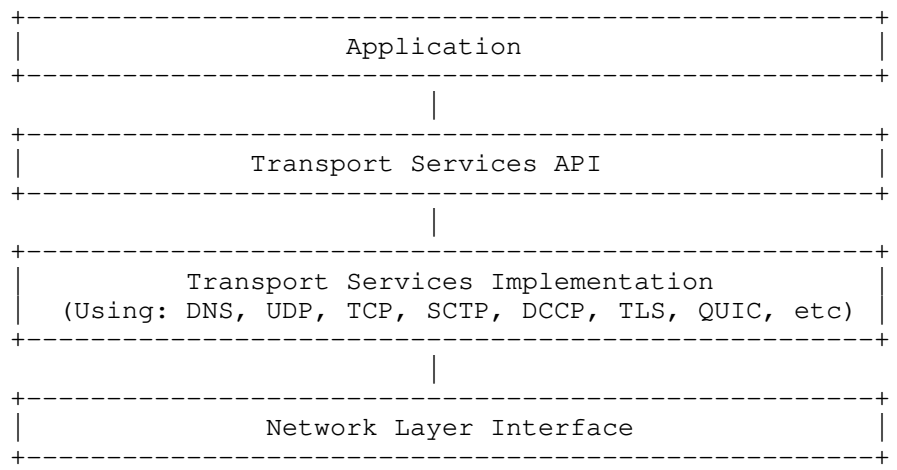


Figure 2: Transport Services API Model

The Transport Services API [I-D.ietf-taps-interface] defines the interface for an application to create Connections and transfer data. It combines interfaces for multiple interaction patterns into a unified whole (see figure 2). This offers generic functions and also the protocol-specific mappings for TCP, UDP, UDP-Lite, and other

protocol layers. These mapping are extensible. Future documents could define similar mappings for new layers and for other transport protocols, such as QUIC [RFC9000]. By combining name resolution with connection establishment and data transfer in a single API, it allows for more flexible implementations to provide path and transport protocol agility on the application's behalf.

The Transport Services Implementation [I-D.ietf-taps-impl] is the component of the Transport Services System that implements the transport layer protocols and other functions needed to send and receive data. It is responsible for mapping the API to a specific available transport Protocol Stack and managing the available network interfaces and paths.

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

2.1. Event-Driven API

Originally, the Socket API presented a blocking interface for establishing connections and transferring data. However, most modern applications interact with the network asynchronously. Emulation of an asynchronous interface using the Socket API can 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.

In contrast to the Socket API, all interactions using the Transport Services API are expected to be asynchronous. The API is defined around an event-driven model (see Section 4.1.6), which models this asynchronous interaction. Other forms of asynchronous communication could also be available to applications, depending on the platform implementing the interface.

For example, when an application that uses the Transport Services API wants to receive data, it issues an asynchronous call to receive new data from the Connection. When delivered data becomes available, this data is delivered to the application using asynchronous events that contain the data. Error handling is also asynchronous, resulting in asynchronous error events.

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

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

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

2.2. Data Transfer Using Messages

The Socket API provides a message interface for datagram protocols like UDP, but provides an unstructured stream abstraction for TCP. While TCP has the ability to send and receive data as a byte-stream, most applications need to interpret structure within this byte-stream. For example, HTTP/1.1 uses character delimiters to segment messages over a byte-stream [RFC9112]; TLS record headers carry a version, content type, and length [RFC8446]; and HTTP/2 uses frames to segment its headers and bodies [RFC9113].

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

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

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

2.3. Flexible Implementation

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

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

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

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

Flexibility after connection establishment is also important. Transport protocols that can migrate between multiple network-layer interfaces need to be able to process and react to interface changes.

Protocols that support multiple application-layer streams need to support initiating and receiving new streams using existing connections.

2.4. Coexistence

While the architecture of the Transport Services System is designed as an enhanced replacement for the Socket API, it need not replace it entirely on a system or platform; indeed, coexistence has been recommended for incremental deployability [RFC8170]. The architecture is therefore designed such that it can run alongside (or, indeed, on top of) an existing Socket API implementation; only applications built to the Transport Services API are managed by the system's Transport Services Implementation.

3. API and Implementation Requirements

One goal of the 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. The architecture also encompasses system policies that can influence and inform how transport protocols use a network path or interface.

There are several ways the Transport Services System can offer flexibility to an application: it can provide access to transport protocols and protocol features; it can use these protocols across multiple paths that could have different performance and functional characteristics; and it can communicate with different remote systems to optimize performance, robustness to failure, or some other metric. Beyond these, if the Transport Services API remains the same over time, new protocols and features can be added to the Transport Services Implementation without requiring changes in applications for adoption. Similarly, this can provide a common basis for utilizing information about a network path or interface, enabling evolution below the transport layer.

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

3.1. Provide Common APIs for Common Features

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

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

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

It is RECOMMENDED that the default values for Properties are selected to ensure correctness for the widest set of applications, while providing the widest set of options for selection. For example, since both applications that require reliability and those that do not require reliability can function correctly when a protocol provides reliability, reliability ought to be enabled by default. As another example, the default value for a Property regarding the selection of network interfaces ought to permit as many interfaces as possible.

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

3.2. Allow Access to Specialized Features

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

A specialized feature could be needed by an application only when using a specific protocol, and not when using others. For example, if an application is using TCP, it could require control over the User Timeout Option for TCP [RFC5482]; these options would not take effect for other transport protocols. In such cases, the API ought to expose the features in such a way that they take effect when a particular protocol is selected, but do not imply that only that protocol could be used. For example, if the API allows an application to specify a preference to use the User Timeout Option, communication would not fail when a protocol such as UDP is selected.

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

To control these specialized features, the application can declare its preference whether the presence of a specific feature is prohibited, should be avoided, can be ignored, is preferred, or is required in the pre-establishment phase. An implementation of a Transport Services API would honor this preference and allow the application to query the availability of each specialized feature after a successful establishment.

3.3. Select Between Equivalent Protocol Stacks

A Transport Services Implementation can attempt and select between multiple Protocol Stacks based on the Selection and Connection Properties communicated by the application, along with any Security Parameters. The implementation can only attempt to use multiple Protocol Stacks when they are "equivalent", which means that the stacks can provide the same Transport Properties and interface expectations as requested by the application. Equivalent Protocol Stacks can be safely swapped or raced in parallel (see Section 4.2.2) during connection establishment.

The following two examples show non-equivalent Protocol Stacks:

* If the application requires preservation of message boundaries, a Protocol Stack that runs UDP as the top-level interface to the application is not equivalent to a Protocol Stack that runs TCP as the top-level interface. A UDP stack would allow an application to read out message boundaries based on datagrams sent from the remote system, whereas TCP does not preserve message boundaries on its own, but needs a framing protocol on top to determine message boundaries.

* If the application specifies that it requires reliable transmission of data, then a Protocol Stack using UDP without any reliability layer on top would not be allowed to replace a Protocol Stack using TCP.

The following example shows Equivalent Protocol Stacks:

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

A Transport Services Implementation can race different security protocols, e.g., if the System Policy is explicitly configured to consider them equivalent. A Transport Services implementation SHOULD only race Protocol Stacks where the transport security protocols within the stacks are identical. To ensure that security protocols are not incorrectly swapped, a Transport Services Implementation MUST only select Protocol Stacks that meet application requirements ([RFC8922]). A Transport Services Implementation MUST NOT automatically fall back from secure protocols to insecure protocols, or to weaker versions of secure protocols. A Transport Services Implementation MAY allow applications to explicitly specify which versions of a protocol ought to be permitted, e.g., to allow a minimum version of TLS 1.2 in case TLS 1.3 is not available.

A Transport Services Implementation MAY specify security properties relating to how the system operates (e.g., requirements, prohibitions, and preferences for the use of DNS Security Extensions (DNSSEC) or DNS over HTTPS (DoH)).

3.4. Maintain Interoperability

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

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

3.5. Support Monitoring

The Transport Services API increases the layer of abstraction for applications, and it enables greater automation below the API. Such increased abstraction comes at the cost of increased complexity when application programmers, users or system administrators try to understand why any issues and failures may be happening. Transport Services systems should therefore offer monitoring functions that provide relevant debug and diagnostics information. For example, such monitoring functions could indicate the protocol(s) in use, the number of open connections per protocol, and any statistics that these protocols may offer.

4. Transport Services Architecture and Concepts

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

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

1. API concepts, which are intended to be exposed to applications; and

2. System-implementation concepts, which are intended to be internally used by a Transport Services Implementation.

The following diagram summarizes the top-level concepts in a Transport Services System and how they relate to one another.

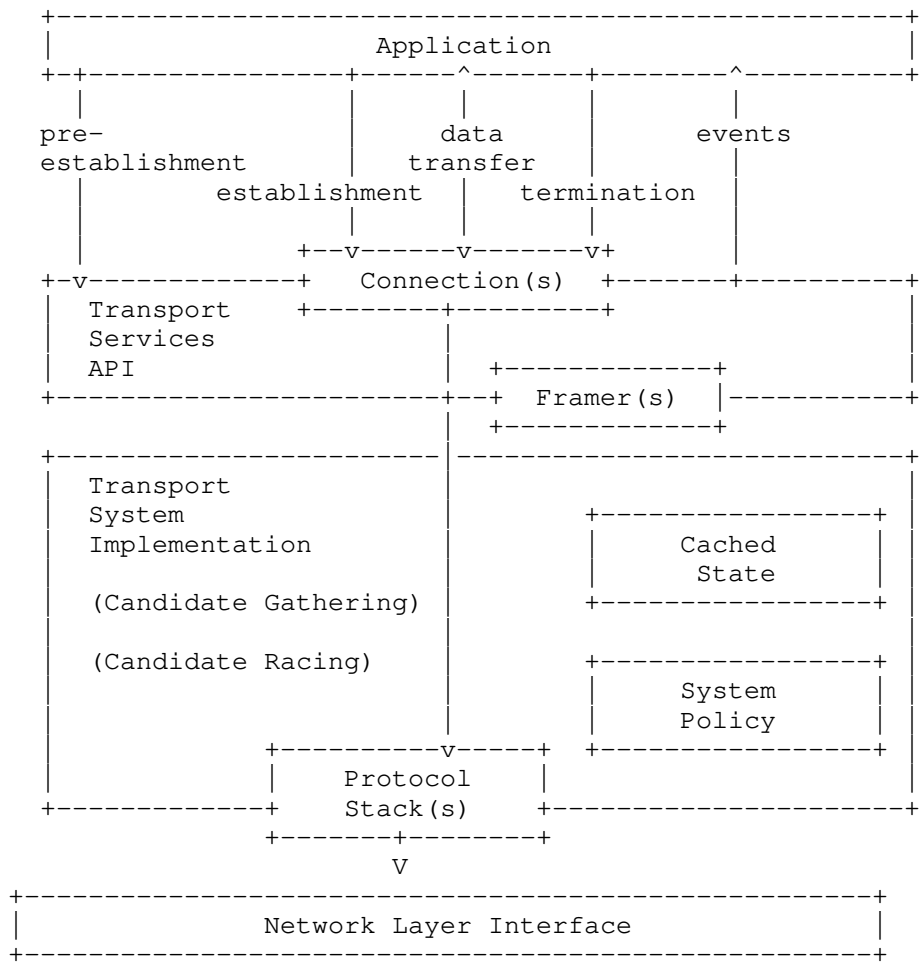


Figure 3: Concepts and Relationships in the Architecture of the Transport Services System

The Transport Services Implementation includes the Cached State and System Policy.

The System Policy provides input from an operating system or other global preferences that can constrain or influence how an implementation will gather Candidate Paths and Protocol Stacks and race the candidates when establishing a Connection. As the details of System Policy configuration and enforcement are largely platform- and implementation- dependent, and do not affect application-level interoperability, the Transport Services API [I-D.ietf-taps-interface] does not specify an interface for reading or writing System Policy.

The Cached State is the state and history that the Transport Services Implementation keeps for each set of associated Endpoints that have previously been used. An application ought to explicitly request any required or desired properties via the Transport Services API.

4.1. Transport Services API Concepts

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

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

* Pre-establishment (Section 4.1.3) encompasses the properties that an application can pass to describe its intent, requirements, prohibitions, and preferences for its networking operations. These properties apply to multiple transport protocols, unless otherwise specified. Properties specified during pre-establishment can have a large impact on the rest of the interface: they modify how establishment occurs, they influence the expectations around data transfer, and they determine the set of events that will be supported.

* Establishment (Section 4.1.4) focuses on the actions that an application takes on the Connection objects to prepare for data transfer.

* Data Transfer (Section 4.1.5) consists of how an application represents the data to be sent and received, the functions required to send and receive that data, and how the application is notified of the status of its data transfer.

* Event Handling (Section 4.1.6) defines categories of notifications that an application can receive during the lifetime of a Connection. Events also provide opportunities for the application to interact with the underlying transport by querying state or updating maintenance options.

* Termination (Section 4.1.7) focuses on the methods by which data transmission is stopped, and connection state is torn down.

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

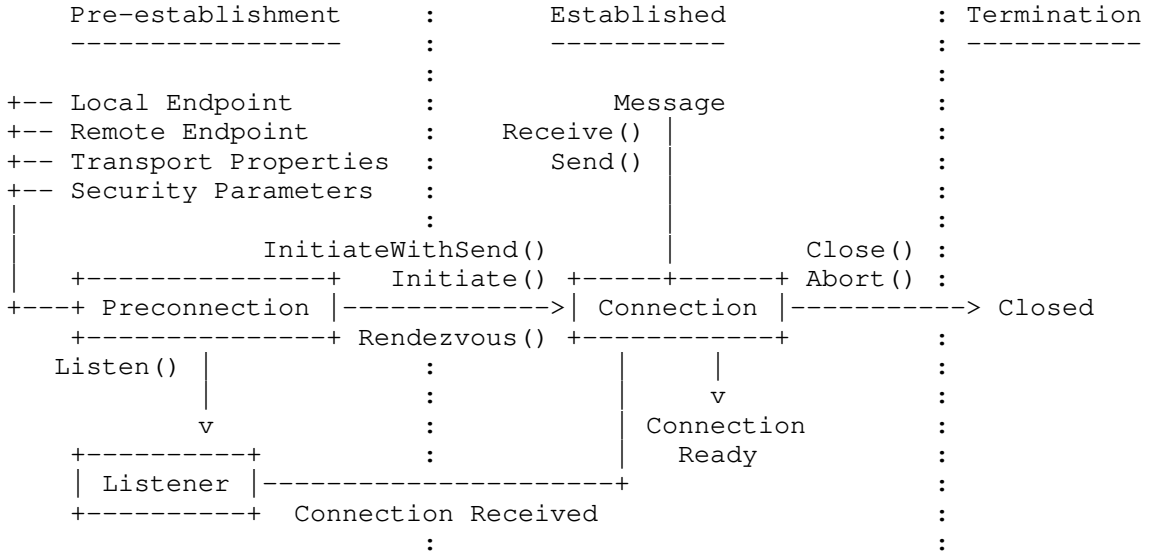


Figure 4: The lifetime of a Connection object

In this diagram, the lifetime of a Connection object is divided into three phases: pre-establishment, the Established state, and Termination.

Pre-establishment is based around a Preconnection object, that contains various sub-objects that describe the properties and parameters of desired Connections (Local and Remote Endpoints, Transport Properties, and Security Parameters). A Preconnection can be used to start listening for inbound connections, in which case a Listener object is created, or can be used to establish a new connection directly using Initiate (for outbound connections) or Rendezvous (for peer-to-peer connections).

Once a Connection is in the Established state, an application can send and receive Message objects, and receive state updates.

Closing or aborting a connection, either locally or from the peer, can terminate a connection.

4.1.1. Endpoint Objects

An Endpoint Identifier specifies one side of a transport connection. Endpoints can be Local Endpoints or Remote Endpoints, and the Endpoint Identifiers can respectively represent an identity that the application uses for the source or destination of a connection. An Endpoint Identifier can be specified at various levels of abstraction. An Endpoint Identifier at a higher level of abstraction (such as a hostname) can be resolved to more concrete identities (such as IP addresses). A Remote Endpoint Identifier can also represent a multicast group or anycast address. In the case of multicast, this selects a multicast transport for communication.

* Remote Endpoint Identifier: The Remote Endpoint Identifier represents the application's identifier for a peer that can participate in a transport connection; for example, the combination of a DNS name for the peer and a service name/port.

* Local Endpoint Identifier: The Local Endpoint Identifier represents the application's identifier for itself that it uses for transport connections; for example, a local IP address and port.

4.1.2. Connections and Related Objects

* Connection: A Connection object represents one or more active transport protocol instances that can send and/or receive Messages between Local and Remote Endpoints. It is an abstraction that represents the communication. The Connection object holds state pertaining to the underlying transport protocol instances and any ongoing data transfers. For example, an active Connection can represent a connection-oriented protocol such as TCP, or can represent a fully-specified 5-tuple for a connectionless protocol

such as UDP, where the Connection remains an abstraction at the endpoints. It can also represent a pool of transport protocol instances, e.g., a set of TCP and QUIC connections to equivalent endpoints, or a stream of a multi-streaming transport protocol instance. Connections can be created from a Preconnection or by a Listener.

* Preconnection: A Preconnection object is a representation of a Connection that has not yet been established. It has state that describes parameters of the Connection: the Local Endpoint Identifier from which that Connection will be established, the Remote Endpoint Identifier (Section 4.1.3) to which it will connect, and Transport Properties that influence the paths and protocols a Connection will use. A Preconnection can be either fully specified (representing a single possible Connection), or it can be partially specified (representing a family of possible Connections). The Local Endpoint (Section 4.1.3) is required for a Preconnection used to Listen for incoming Connections, but optional if it is used to Initiate a Connection. The Remote Endpoint Identifier is required in a Preconnection that used to Initiate a Connection, but is optional if it is used to Listen for incoming Connections. The Local Endpoint Identifier and the Remote Endpoint Identifier are both required if a peer-to-peer Rendezvous is to occur based on the Preconnection.

* Transport Properties: Transport Properties allow the application to express their requirements, prohibitions, and preferences and configure a Transport Services Implementation. There are three kinds of Transport Properties:

  - Selection Properties (Section 4.1.3): Selection Properties can only be specified on a Preconnection.

  - Connection Properties (Section 4.1.3): Connection Properties can be specified on a Preconnection and changed on the Connection.

  - Message Properties (Section 4.1.5): Message Properties can be specified as defaults on a Preconnection or a Connection, and can also be specified during data transfer to affect specific Messages.

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

4.1.3. Pre-establishment

* Selection Properties: The Selection Properties consist of the properties that an application can set to influence the selection of paths between the Local and Remote Endpoints, to influence the selection of transport protocols, or to configure the behavior of generic transport protocol features. These properties can take the form of requirements, prohibitions, or preferences. Examples of properties that influence path selection include the interface type (such as a Wi-Fi connection, or a Cellular LTE connection), requirements around the largest Message that can be sent, or preferences for throughput and latency. Examples of properties that influence protocol selection and configuration of transport protocol features include reliability, multipath support, and fast open support.

* Connection Properties: The Connection Properties are used to configure protocol-specific options and control per-connection behavior of a Transport Services Implementation; for example, a protocol-specific Connection Property can express that if TCP is used, the implementation ought to use the User Timeout Option. Note that the presence of such a property does not require that a specific protocol will be used. In general, these properties do not explicitly determine the selection of paths or protocols, but can be used by an implementation during connection establishment. Connection Properties are specified on a Preconnection prior to Connection establishment, and can be modified on the Connection later. Changes made to Connection Properties after Connection establishment take effect on a best-effort basis.

* Security Parameters: Security Parameters define an application's requirements for authentication and encryption on a Connection. They are used by Transport Security protocols (such as those described in [RFC8922]) to establish secure Connections. Examples of parameters that can be set include local identities, private keys, supported cryptographic algorithms, and requirements for validating trust of remote identities. Security Parameters are primarily associated with a Preconnection object, but properties related to identities can be associated directly with Endpoints.

4.1.4. Establishment Actions

* Initiate: The primary action that an application can take to create a Connection to a Remote Endpoint, and prepare any required local or remote state to enable the transmission of Messages. For some protocols, this will initiate a client-to-server style handshake; for other protocols, this will just establish local state (e.g., with connectionless protocols such as UDP). The

process of identifying options for connecting, such as resolution of the Remote Endpoint Identifier, occurs in response to the Initiate call.

* Listen: Enables a Listener to accept incoming connections. The Listener will then create Connection objects as incoming connections are accepted (Section 4.1.6). Listeners by default register with multiple paths, protocols, and Local Endpoints, unless constrained by Selection Properties and/or the specified Local Endpoint Identifier(s). Connections can be accepted on any of the available paths or endpoints.

* Rendezvous: The action of establishing a peer-to-peer connection with a Remote Endpoint. It simultaneously attempts to initiate a connection to a Remote Endpoint while listening for an incoming connection from that Endpoint. The process of identifying options for the connection, such as resolution of the Remote Endpoint Identifier(s), occurs in response to the Rendezvous call. As with Listeners, the set of local paths and endpoints is constrained by Selection Properties. If successful, the Rendezvous call generates and asynchronously returns a Connection object to represent the established peer-to-peer connection. The processes by which connections are initiated during a Rendezvous action will depend on the set of Local and Remote Endpoints configured on the Preconnection. For example, if the Local and Remote Endpoints are TCP host candidates, then a TCP simultaneous open [RFC9293] might be performed. However, if the set of Local Endpoints includes server reflexive candidates, such as those provided by STUN (Session Traversal Utilities for NAT) [RFC5389], a Rendezvous action will race candidates in the style of the ICE (Interactive Connection Establishment) algorithm [RFC8445] to perform NAT binding discovery and initiate a peer-to-peer connection.

4.1.5. Data Transfer Objects and Actions

* Message: A Message object is a unit of data that can be represented as bytes that can be transferred between two endpoints over a transport connection. The bytes within a Message are assumed to be ordered. If an application does not care about the order in which a peer receives two distinct spans of bytes, those spans of bytes are considered independent Messages. Messages are sent in the payload of IP packets. One packet can carry one or more Messages or parts of a Message.

* Message Properties: Message Properties are used to specify details about Message transmission. They can be specified directly on individual Messages, or can be set on a Preconnection or Connection as defaults. These properties might only apply to how

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

* Send: The action to transmit a Message over a Connection to the Remote Endpoint. The interface to Send can accept Message Properties specific to how the Message content is to be sent. The status of the Send operation is delivered back to the sending application in an event (Section 4.1.6).

* Receive: An action that indicates that the application is ready to asynchronously accept a Message over a Connection from a Remote Endpoint, while the Message content itself will be delivered in an event (Section 4.1.6). The interface to Receive can include Message Properties specific to the Message that is to be delivered to the application.

* Framer: A Framer is a data translation layer that can be added to a Connection. 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. In this way, message boundaries can be preserved when using a Connection object, even with a protocol that otherwise presents unstructured streams, such as TCP. This is designed based on the fact that many of the current application protocols evolved over TCP, which does not provide message boundary preservation, and since many of these protocols require message boundaries to function, each application layer protocol has defined its own framing. For example, when an HTTP application sends and receives HTTP messages over a byte-stream transport, it must parse the boundaries of HTTP messages from the stream of bytes.

4.1.6. Event Handling

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

* Connection Ready: Signals to an application that a given Connection is ready to send and/or receive Messages. If the Connection relies on handshakes to establish state between peers, then it is assumed that these steps have been taken.

* Connection Closed: Signals to an application that a given Connection is no longer usable for sending or receiving Messages. The event delivers a reason or error to the application that describes the nature of the termination.

* Connection Received: Signals to an application that a given Listener has received a Connection.

* Message Received: Delivers received Message content to the application, based on a Receive action. To allow an application to limit the occurrence of such events, each call to Receive will be paired with a single Receive event. This can include an error if the Receive action cannot be satisfied, e.g., due to the Connection being closed.

* Message Sent: Notifies the application of the status of its Send action. This might indicate a failure if the Message cannot be sent, or an indication that the Message has been processed by the Transport Services System.

* Path Properties Changed: Notifies the application that a property of the Connection has changed that might influence how and where data is sent and/or received.

4.1.7. Termination Actions

* Close: The action an application takes on a Connection to indicate that it no longer intends to send data, is no longer willing to receive data, and that the protocol should signal this state to the Remote Endpoint if the transport protocol allows this. (Note that this is distinct from the concept of "half-closing" a bidirectional connection, such as when a FIN is sent in one direction of a TCP connection [RFC9293]. The end of a stream can also be indicated using Message Properties when sending.)

* Abort: The action the application takes on a Connection to indicate a Close and also indicate that the Transport Services System should not attempt to deliver any outstanding data, and immediately drop the connection. This is intended for immediate, usually abnormal, termination of a connection.

4.1.8. Connection Groups

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

The API allows a Connection to be created from another Connection. This adds the new Connection to the Connection Group. A change to one of the Connection Properties on any Connection in the Connection Group automatically changes the Connection Property for all others. All Connections in a Connection Group share the same set of Connection Properties except for the Connection Priority. These Connection Properties are said to be entangled.

Passive Connections can also be added to a Connection Group, e.g., when a Listener receives a new Connection that is just a new stream of an already active multi-streaming protocol instance.

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

4.2. Transport Services Implementation

This section defines the key architectural concepts for the Transport Services Implementation within the Transport Services System.

The Transport Services System consists of the Transport Services Implementation and the Transport Services API. The Transport Services Implementation consists of all objects and protocol instances used internally to a system or library to implement the functionality needed to provide a transport service across a network, as required by the abstract interface.

* Path: Represents an available set of properties that a Local Endpoint can use to communicate with a Remote Endpoint, such as routes, addresses, and physical and virtual network interfaces.

* Protocol Instance: A single instance of one protocol, including any state necessary to establish connectivity or send and receive Messages.

* Protocol Stack: A set of Protocol Instances (including relevant application, security, transport, or Internet protocols) that are used together to establish connectivity or send and receive Messages. A single stack can be simple (a single transport protocol instance over IP), or it can be complex (multiple application protocol streams going through a single security and transport protocol, over IP; or, a multi-path transport protocol over multiple transport sub-flows).

* Candidate Path: One path that is available to an application and conforms to the Selection Properties and System Policy, of which there can be several. Candidate Paths are identified during the gathering phase (Section 4.2.1) and can be used during the racing phase (Section 4.2.2).

* Candidate Protocol Stack: One Protocol Stack that can be used by an application for a Connection, for which there can be several candidates. Candidate Protocol Stacks are identified during the gathering phase (Section 4.2.1) and are started during the racing phase (Section 4.2.2).

* System Policy: The input from an operating system or other global preferences that can constrain or influence how an implementation will gather candidate paths and Protocol Stacks (Section 4.2.1) and race the candidates during establishment (Section 4.2.2). Specific aspects of the System Policy either apply to all Connections or only certain ones, depending on the runtime context and properties of the Connection.

* Cached State: The state and history that the implementation keeps for each set of associated Endpoints that have been used previously. This can include DNS results, TLS session state, previous success and quality of transport protocols over certain paths, as well as other information. This caching does not imply that the same decisions are necessarily made for subsequent connections, rather, it means that cached state is used by a Transport Services Implementation to inform functions such as choosing the candidates to be raced, selecting appropriate transport parameters, etc. An application SHOULD NOT rely on specific caching behaviour, instead it ought to explicitly request any required or desired properties via the Transport Services API.

4.2.1. Candidate Gathering

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

* Candidate Protocol Selection: Candidate Protocol Selection represents the act of choosing one or more sets of Protocol Stacks that are available to use based on the Transport Properties provided by the application, and the heuristics or policies within the Transport Services Implementation.

4.2.2. Candidate Racing

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

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

* Path Racing: Path Racing is the act of attempting to establish, or scheduling attempts to establish, multiple Protocol Stacks that differ based on a selection from the available Paths. Since different Paths will have distinct configurations (see [RFC7556]) 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 Endpoint Identifiers.

* 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 Identifier, such as a particular IP address that was resolved from a DNS hostname.

4.2.3. Separating Connection Contexts

A Transport Services Implementation can by default share stored properties across Connections within an application, such as cached protocol state, cached path state, and heuristics. This provides efficiency and convenience for the application, since the Transport Services System can automatically optimize behavior.

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

* Privacy concerns about re-using cached protocol state that can lead to linkability. Sensitive state could include TLS session state [RFC8446] and HTTP cookies [RFC6265]. These concerns could be addressed using Connection Contexts with separate caches, such as for different browser tabs.

* Privacy concerns about allowing Connections to multiplex together, which can tell a Remote Endpoint that all of the Connections are coming from the same application. Using Connection Contexts avoids the Connections being multiplexed in a HTTP/2 or QUIC stream.

5. IANA Considerations

This document has no actions for IANA.

6. Security and Privacy Considerations

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

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

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

Applications need to ensure that they use security APIs appropriately. In cases where applications use an interface to provide sensitive keying material, e.g., access to private keys or copies of pre-shared keys (PSKs), key use needs to be validated and scoped to the intended protocols and roles. For example, if an application provides a certificate to only be used as client authentication for outbound TLS and QUIC connections, the Transport Services System MUST NOT use this automatically in other contexts (such as server authentication for inbound connections, or in other another security protocol handshake that is not equivalent to TLS).

A Transport Services System MUST NOT automatically fall back from secure protocols to insecure protocols, or to weaker versions of secure protocols (see Section 3.3). For example, if an application requests a specific version of TLS, but the desired version of TLS is not available, its connection will fail. As described in Section 3.3, the Transport Services API can allow applications to specify minimum versions that are allowed to be used by the Transport Services System.

7. Acknowledgements

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

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

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

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

8. References

8.1. Normative References

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

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

8.2. Informative References

[I-D.ietf-taps-impl] Brunstrom, A., Pauly, T., Enghardt, R., Tiesel, P. S., and M. Welzl, "Implementing Interfaces to Transport Services", Work in Progress, Internet-Draft, draft-ietf-taps-impl-16, 5 June 2023, <https://datatracker.ietf.org/doc/html/draft-ietf-taps-impl-16>.

[I-D.ietf-taps-interface] Trammell, B., Welzl, M., Enghardt, R., Fairhurst, G., Kühlewind, M., Perkins, C., Tiesel, P. S., and T. Pauly, "An Abstract Application Layer Interface to Transport Services", Work in Progress, Internet-Draft, draft-ietf-taps-interface-22, 6 July 2023, <https://datatracker.ietf.org/doc/html/draft-ietf-taps-interface-22>.

[POSIX] "IEEE Std. 1003.1-2008 Standard for Information Technology -- Portable Operating System Interface (POSIX). Open group Technical Standard: Base Specifications, Issue 7", 2008.

[RFC5389] Rosenberg, J., Mahy, R., Matthews, P., and D. Wing, "Session Traversal Utilities for NAT (STUN)", RFC 5389, DOI 10.17487/RFC5389, October 2008, <https://www.rfc-editor.org/rfc/rfc5389>.

[RFC5482] Eggert, L. and F. Gont, "TCP User Timeout Option", RFC 5482, DOI 10.17487/RFC5482, March 2009, <https://www.rfc-editor.org/rfc/rfc5482>.

[RFC6265] Barth, A., "HTTP State Management Mechanism", RFC 6265, DOI 10.17487/RFC6265, April 2011, <https://www.rfc-editor.org/rfc/rfc6265>.

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

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

[RFC8170] Thaler, D., Ed., "Planning for Protocol Adoption and Subsequent Transitions", RFC 8170, DOI 10.17487/RFC8170, May 2017, <https://www.rfc-editor.org/rfc/rfc8170>.

[RFC8303] Welzl, M., Tuexen, M., and N. Khademi, "On the Usage of Transport Features Provided by IETF Transport Protocols", RFC 8303, DOI 10.17487/RFC8303, February 2018, <https://www.rfc-editor.org/rfc/rfc8303>.

[RFC8305] Schinazi, D. and T. Pauly, "Happy Eyeballs Version 2: Better Connectivity Using Concurrency", RFC 8305, DOI 10.17487/RFC8305, December 2017, <https://www.rfc-editor.org/rfc/rfc8305>.

[RFC8445] Keranen, A., Holmberg, C., and J. Rosenberg, "Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal", RFC 8445, DOI 10.17487/RFC8445, July 2018, <https://www.rfc-editor.org/rfc/rfc8445>.

[RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <https://www.rfc-editor.org/rfc/rfc8446>.

[RFC8922] Enghardt, T., Pauly, T., Perkins, C., Rose, K., and C. Wood, "A Survey of the Interaction between Security Protocols and Transport Services", RFC 8922, DOI 10.17487/RFC8922, October 2020, <https://www.rfc-editor.org/rfc/rfc8922>.

[RFC8923] Welzl, M. and S. Gjessing, "A Minimal Set of Transport Services for End Systems", RFC 8923, DOI 10.17487/RFC8923, October 2020, <https://www.rfc-editor.org/rfc/rfc8923>.

[RFC9000] Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport", RFC 9000, DOI 10.17487/RFC9000, May 2021, <https://www.rfc-editor.org/rfc/rfc9000>.

[RFC9112] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "HTTP/1.1", STD 99, RFC 9112, DOI 10.17487/RFC9112, June 2022, <https://www.rfc-editor.org/rfc/rfc9112>.

[RFC9113] Thomson, M., Ed. and C. Benfield, Ed., "HTTP/2", RFC 9113, DOI 10.17487/RFC9113, June 2022, <https://www.rfc-editor.org/rfc/rfc9113>.

[RFC9293] Eddy, W., Ed., "Transmission Control Protocol (TCP)", STD 7, RFC 9293, DOI 10.17487/RFC9293, August 2022, <https://www.rfc-editor.org/rfc/rfc9293>.

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 Switzerland GmbH
Gustav-Gull-Platz 1
CH- 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
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@csperkins.org

TAPS Working Group
Internet-Draft
Intended status: Informational
Expires: 16 June 2024

A. Brunstrom, Ed.
Karlstad University
T. Pauly, Ed.
Apple Inc.
R. Enghardt
Netflix
P. Tiesel
SAP SE
M. Welzl
University of Oslo
14 December 2023

Implementing Interfaces to Transport Services
draft-ietf-taps-impl-18

Abstract

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

Table of Contents

1. Introduction

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

This document serves as a guide to implementing a system that provides a Transport Services API. This guide offers suggestions to developers, but it is not prescriptive: implementations are free to take any desired form as long as the API specification in [I-D.ietf-taps-interface] is honored. 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 Transport Services architecture [I-D.ietf-taps-arch].

2. Implementing Connection Objects

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

* the Preconnection, the bundle of properties that describes the application constraints on, and preferences for, the transport;

* the Connection, the basic object that represents a flow of data as Messages in either direction between the Local and Remote Endpoints;

* and the Listener, a passive waiting object that delivers new Connections.

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

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

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

Connection establishment is only a local operation for a connectionless protocols, which serves to simplify the local send/receive functions and to filter the traffic for the specified addresses and ports [RFC8085] (for example using UDP or UDP-Lite transport without a connection handshake procedure).

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

3. Implementing Pre-Establishment

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

During pre-establishment the application specifies one or more Endpoints to be used for communication as well as protocol preferences and constraints via Selection Properties and, if desired, also Connection Properties. Section 4 of [I-D.ietf-taps-interface] states that Connection Properties should preferably be configured during pre-establishment, because they can serve as input to decisions that are made by the implementation (e.g., the capacity profile can guide usage of a protocol offering scavenger-type congestion control).

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

3.1. Configuration-time errors

The Transport Services system should have a list of supported protocols available, which each have transport features reflecting the capabilities of the protocol. Once an application specifies its Transport Properties, the Transport Services system matches the required and prohibited properties against the transport features of the available protocols (see Section 6.2 of [I-D.ietf-taps-interface] for the definition of property preferences).

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

* A request by an application for properties that cannot be satisfied by any of the available protocols. For example, if an application requires perMsgReliability, but no such feature is available in any protocol on the host running the Transport Services system this should result in an error.

* A request by an application for properties that are in conflict with each other, such as specifying required and prohibited properties that cannot be satisfied by any protocol. For example, if an application prohibits reliability but then requires perMsgReliability, this mismatch should result in an error.

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

3.2. Role of system policy

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

1. Application preferences, i.e., preferences specified during the pre-establishment via Selection Properties.

2. Dynamic system policy, i.e., policy compiled from internally and externally acquired information about available network interfaces, supported transport protocols, and current/previous Connections. Examples of ways to externally retrieve policy-support information are through OS-specific statistics/measurement tools and tools that reside on middleboxes and routers.

3. Default implementation policy, i.e., predefined policy by OS or application.

In general, any protocol or path used for a Connection must conform to all three sources of constraints. A violation that occurs at any of the policy layers should cause a protocol or path to be considered ineligible for use. If such a violation prevents a Connection from being established, this should be communicated to the application, e.g. via the EstablishmentError event. 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.

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 which point the Preconnection object contains all constraints or requirements the application has configured. The establishment 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 (defined in Section 4.2.1 of [I-D.ietf-taps-arch]), to identify the paths, protocols, and endpoints to use (see Section 4.2); and Candidate Racing (defined in Section 4.2.2 of [I-D.ietf-taps-arch]), in which the necessary protocol handshakes are conducted so that the Transport Services system can select which set to use (see Section 4.3). Candidate Racing involves attempting multiple options for connection establishment, and choosing the first option to succeed as the Protocol Stack to use for the connection. These attempts are usually staggered, starting each next option after a delay, but they can also be performed in parallel or only after waiting for failures.

For ease of illustration, this document structures the candidates for racing as a tree (see Section 4.1). This is not meant to restrict implementations from structuring racing candidates differently.

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

Whenever an implementation has multiple options for connection establishment, it can view the set of all individual connection establishment options as a single, aggregate connection establishment. The aggregate set conceptually includes every valid combination of endpoints, paths, and protocols. As an example, consider an implementation that initiates a TCP connection to a hostname + port Endpoint Identifier, 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 Identifier: www.example.com:443] [Interface: Any]   [Protocol
: TCP]
|-> [Endpoint Identifier: [2001:db8:23::1]:443]        [Interface: Wi-Fi] [Protoc
ol: TCP]
|-> [Endpoint Identifier: 192.0.2.1:443]             [Interface: LTE]   [Protocol
: TCP]
|-> [Endpoint Identifier: [2001:db8:42::1]:443]        [Interface: LTE]   [Protoc
ol: TCP]
```

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

During Candidate Gathering (Section 4.2), an implementation prunes and sorts branches according to the Selection Property preferences (Section 6.2 of [I-D.ietf-taps-interface]. It first excludes all protocols and paths that match a Prohibit property or do not match all Require properties. Then it will sort branches according to Preferred properties, Avoided properties, and possibly other criteria.

4.1. Structuring Candidates as a Tree

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

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

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

The rest of this section will use a notation scheme to represent this tree. The root node (or parent node) of the tree will be represented by a single integer, such as "1". ("1" is used assuming that this is the first connection made by the system; future connections created by the application would allocate numbers in an increasing manner.) 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 its parent's identifier with a dot in between, such as "1.1" and "1.2". Each node will be summarized by a tuple of three elements: endpoint, path (labeled here

by interface), and protocol. In Protocol Stacks, the layers are separated by '/' and ordered with the protocol closest to the application first. The above example can now be written more succinctly as:

```
1 [www.example.com:443, any path, TCP]
  1.1 [www.example.com:443, Wi-Fi, TCP]
    1.1.1 [[2001:db8:23::1]:443, Wi-Fi, TCP]
  1.2 [www.example.com:443, LTE, TCP]
    1.2.1 [192.0.2.1:443, LTE, TCP]
    1.2.2 [[2001:db8.42::1]:443, LTE, TCP]
```

When an implementation is asked to establish a single connection, only one of the leaf nodes in the candidate set is needed to transfer data. Thus, once a single leaf node becomes ready to use, then the connection establishment tree is considered ready. One way to implement this is by having every leaf node update the state of its parent node when it becomes ready, until the root 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 consist of only a single node, such as a connection attempt to an IP address over a single interface with a single protocol.

```
1 [[2001:db8:23::1]:443, Wi-Fi, TCP]
```

A root 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:443, Wi-Fi, TCP]
  1.1 [[2001:db8:23::1]:443, Wi-Fi, TCP]
```

4.1.1. Branch Types

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

4.1.1.1. Derived Endpoints

If a connection originally targets a single Endpoint Identifer, there may be multiple endpoint candidates of different types that can be derived from the original. This creates an ordered list of the derived endpoint candidates 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 Identifer on a traditional IP network, the implementation should send all applicable DNS queries. Commonly, this will include both A (IPv4) and AAAA (IPv6) records if both address families are supported on the local interface. This can also include SRV records [RFC2782], SVCB and HTTPS records [I-D.ietf-dnsop-svcb-https], or other future record types. The algorithm for ordering and racing these addresses should follow the recommendations in Happy Eyeballs [RFC8305].

```
1 [www.example.com:443, Wi-Fi, TCP]
  1.1 [[2001:db8::1]:443, Wi-Fi, TCP]
  1.2 [192.0.2.1:443, Wi-Fi, TCP]
  1.3 [[2001:db8::2]:443, Wi-Fi, TCP]
  1.4 [[2001:db8::3]:443, Wi-Fi, TCP]
```

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

```
1 [term-printer._ipp._tcp.meeting.example.com, Wi-Fi, TCP]
  1.1 [term-printer.meeting.example.com:631, Wi-Fi, TCP]
    1.1.1 [31.133.160.18:631, Wi-Fi, TCP]
```

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

4.1.1.2. Network Paths

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

```
1 [192.0.2.1:443, any path, TCP]
  1.1 [192.0.2.1:443, Wi-Fi, TCP]
  1.2 [192.0.2.1:443, LTE, TCP]
```

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

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

4.1.1.3. Protocol Options

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

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

```
1 [www.example.com:443, any path, HTTP/TCP]
  1.1 [192.0.2.8:443, any path, HTTP/HTTP Proxy/TCP]
  1.2 [192.0.2.7:10234, any path, HTTP/SOCKS/TCP]
  1.3 [www.example.com:443, any path, HTTP/TCP]
    1.3.1 [192.0.2.1:443, any path, HTTP/TCP]
```

This approach also allows a client to attempt different sets of application and transport protocols that, when available, could provide preferable features. For example, the protocol options could involve QUIC [RFC9000] over UDP on one branch, and HTTP/2 [RFC7540] over TLS over TCP on the other:

```
1 [www.example.com:443, any path, HTTP]
  1.1 [www.example.com:443, any path, HTTP3/QUIC/UDP]
    1.1.1 [192.0.2.1:443, any path, HTTP3/QUIC/UDP]
  1.2 [www.example.com:443, any path, HTTP2/TLS/TCP]
    1.2.1 [192.0.2.1:443, any path, HTTP2/TLS/TCP]
```

Another example is racing SCTP with TCP:

```
1 [www.example.com:4740, any path, reliable-inorder-stream]
  1.1 [www.example.com:4740, any path, SCTP]
    1.1.1 [192.0.2.1:4740, any path, SCTP]
  1.2 [www.example.com:4740, any path, TCP]
    1.2.1 [192.0.2.1:4740, any path, TCP]
```

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

4.1.2. Branching Order-of-Operations

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

This document recommends the following order of operations for branching:

1. Network Paths

2. Protocol Options

3. Derived Endpoints

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

Protocol options are next checked in order. Whether or not a set of protocols, 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 Endpoint Identifers being used. Adding a proxy to a connection's branch will change the Endpoint Identifer 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 attempted first. Then, branches with SCTP will be attempted first within their subtree according to the properties influencing protocol selection. However, if the implementation has current cache information that SCTP is not available on the path over WiFi, there would be no SCTP node in the WiFi subtree. Here, the path over WiFi will be attempted first, and, if connection establishment succeeds, TCP will be used. Thus, the Selection Property preferring WiFi takes precedence over the Property that led to a preference for SCTP.

1. [www.example.com:80, any path, reliable-inorder-stream]
1.1 [192.0.2.1:443, Wi-Fi, reliable-inorder-stream]
1.1.1 [192.0.2.1:443, Wi-Fi, TCP]
1.2 [192.0.3.1:443, LTE, reliable-inorder-stream]
1.2.1 [192.0.3.1:443, LTE, SCTP]
1.2.2 [192.0.3.1:443, LTE, TCP]

4.1.3. Sorting Branches

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

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

* "Interface Instance or Type" (property name interface): If the application specifies an interface type to be preferred or avoided, implementations should accordingly rank the paths. If the application specifies an interface type to be required or prohibited, an implementation is expected to exclude the non-conforming paths.

* "Capacity Profile" (property name connCapacityProfile): An implementation can use the capacity profile to prefer paths that match an application's expected traffic profile. This match will use cached performance estimates, see Section 9.2. Some examples of path preferences based on capacity profiles include:

  - Low Latency/Interactive: Prefer paths with the lowest expected Round Trip Time, based on observed Round Trip Time estimates;

  - Low Latency/Non-Interactive: Prefer paths with a low expected Round Trip Time, but can tolerate delay variation;

  - Constant-Rate Streaming: Prefer paths that are expected to satisfy the requested stream send or receive bitrate, based on the observed maximum throughput;

  - Capacity-Seeking: Prefer adapting to paths to determine the highest available capacity, based on the observed maximum throughput.

As another example, branch sorting can also be influenced by bounds on the send or receive rate (Selection Properties minSendRate / minRecvRate / maxSendRate / maxRecvRate): 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.

Implementations process the Properties (Section 6.2 of [I-D.ietf-taps-interface]) in the following order: Prohibit, Require, Prefer, Avoid. If Selection Properties contain any prohibited properties, the implementation should first purge branches containing nodes with these properties. For required properties, it should only keep branches that satisfy these requirements. Finally, it should order the branches according to the preferred properties, and finally use any avoided properties as a tiebreaker. When ordering branches, an implementation can give more weight to properties that the application has explicitly set, than to the properties that are default.

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

4.2. Candidate Gathering

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

4.2.1. Gathering Endpoint Candidates

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

4.2.1.1. Local Endpoint candidates

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

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

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

4.2.1.2. Remote Endpoint Candidates

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

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

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

4.3. Candidate Racing

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

This section covers the dynamic aspect of connection establishment. The tree described above is a useful conceptual and architectural model. However, an implementation is unable to know all of the nodes that will be used until steps like name resolution have occurred, and many of the possible branches ultimately might not be attempted.

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

1. Simultaneous

2. Staggered

3. Failover

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

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

4.3.1. Simultaneous

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

4.3.2. Staggered

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

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

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

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

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

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

4.3.3. Failover

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

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

4.4. Completing Establishment

The process of connection establishment completes when one leaf node of the tree has successfully completed negotiation with the Remote Endpoint, or else all nodes of the tree have failed to connect. The first leaf node to complete its connection is then used by the application to send and receive data. This is signalled to the application using the Ready event in the API (Section 7.1 of

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

Successes and failures of a given attempt should be reported up to parent nodes (towards the root 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 root 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:443, Any, TCP]
  1.1 [www.example.com:443, Wi-Fi, TCP]
    1.1.1 [192.0.2.1:443, Wi-Fi, TCP]
  1.2 [www.example.com:443, LTE, TCP]
...
```

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

4.4.1. Determining Successful Establishment

On a per-protocol basis, implementations may select different criteria by which a leaf node is considered to be successfully connected. 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 three-way handshake has been completed. If the only protocol being used is an connectionless protocol, like UDP, the implementation may consider the node fully "connected" the moment it determines a route is present, before sending any packets on the network, see further Section 4.6.

When the Initiate action is called without any Messages being sent at the same time, depending on the protocols involved, it is not guaranteed that the Remote Endpoint will be notified of this, and hence a passive endpoint’s application may not receive a ConnectionReceived event until it receives the first Message on the new Connection.

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

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

4.5. Establishing multiplexed connections

Multiplexing several Connections over a single underlying transport connection requires that the Connections to be multiplexed belong to the same Connection Group (as is indicated by the application using the Clone action). When the underlying transport connection supports multi-streaming, the Transport Services System can map each Connection in the Connection Group to a different stream of this connection.

For such streams, there is often no explicit connection establishment procedure for the new stream prior to sending data on it (e.g., with SCTP). In this case, the same considerations apply to determining stream establishment as apply to establishing a UDP connection, as discussed in Section 4.4.1. This means that there might not be any "establishment" message (like a TCP SYN).

4.6. Handling connectionless protocols

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

However, this can cause a problem if a specific peer is not reachable over the network using the connectionless protocol, or data cannot be exchanged with the peer for any other reason. To handle the lack of an explicit handshake in the underlying protocol, an application can use a Message Framer (Section 6) on top of a connectionless protocol to only mark a specific connection attempt as ready when some data has been received, or after some application-level handshake has been performed by the Message Framer.

4.7. Implementing Listeners

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

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

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

4.7.1. Implementing Listeners for Connected Protocols

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

4.7.2. Implementing Listeners for Connectionless Protocols

Connectionless protocols such as UDP and UDP-lite generally do not provide the same mechanisms that connected protocols do to offer Connection objects. Implementations should wait for incoming packets for connectionless protocols on a listening port and should perform four-tuple matching of packets to existing Connection objects if possible. If a matching Connection object does not exist, an incoming packet from a connectionless protocol should cause a new Connection object to be created.

4.7.3. Implementing Listeners for Multiplexed Protocols

Protocols that provide multiplexing of streams can listen for entirely new connections as well as for new sub-connections (streams of an already existing connection). A new stream arrival on an existing connection is presented to the application as a new Connection. This new Connection is grouped with all other Connections that are multiplexed via the same protocol.

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 (such as UDP). Each Message here corresponds to a single datagram.

For protocols that expose byte-streams (such as TCP), 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 framing (such as length-value protocols, or protocols that use delimiters like HTTP/1.1) may support Message sizes that do not fit within a single datagram. Each Message for framing protocols corresponds to a single frame, which may be sent either as a complete Message in the underlying protocol, or in multiple parts.

Messages themselves generally consist of bytes passed in the messageData parameter intended to be processed at an application layer. However, Message objects presented through the API can carry associated Message Properties passed through the messageContext parameter. When these are Protocol Specific Properties, they can include metadata that exists separately from a byte encoding. For example, these Properties can include name-value pairs of information, like HTTP header fields. In such cases, Messages might

be "empty", insofar as they contain zero bytes in the messageData parameter, but can still include data in the messageContext that is interpreted by the Protocol Stack.

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

The API allows various properties to be associated with each Message, which should be implemented as discussed below.

* msgLifetime: this should be implemented by removing the Message from the queue of pending Messages after the Lifetime has expired. A queue of pending Messages within the Transport Services Implementation that have yet to be handed to the Protocol Stack can always support this property, but once a Message has been sent into the send buffer of a protocol, only certain protocols may support removing it from their send buffer. For example, a Transport Services Implementation cannot remove bytes from a TCP send buffer, while it can remove data from a SCTP send buffer using the partial reliability extension [RFC8303]. When there is no standing queue of Messages within the system, and the Protocol Stack does not support the removal of a Message from the stack's send buffer, this property may be ignored.

* msgPriority: this represents the ability to prioritize a Message over other Messages. This can be implemented by the Transport Services system by 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.

* msgOrdered: when this is false, this disables the requirement of in-order-delivery for protocols that support configurable ordering. When the Protocol Stack does not support configurable ordering, this property may be ignored.

* safelyReplayable: when this is true, this means that the Message can be used by a transport mechanism that might deliver it multiple times -- e.g., as a result of racing multiple transports

or as part of TCP Fast Open. Also, protocols that do not protect against duplicated Messages, such as UDP (when used directly, without a protocol layered atop), can only be used with Messages that are Safely Replayable. When a Transport Services system is permitted to replay Messages, replay protection could be provided by the application.

* final: when this is true, this means that the sender will not send any further Messages. The Connection need not be closed (in case the Protocol Stack supports half-close operation, like TCP). Any Messages sent after a Message marked final will result in a SendError.

* msgChecksumLen: when this is set to any value other than Full Coverage, it sets the minimum protection in protocols that allow limiting the checksum length (e.g. UDP-Lite). If the Protocol Stack does not support checksum length limitation, this property may be ignored.

* msgReliable: When true, the property specifies that the Message must be reliably transmitted. When false, and if unreliable transmission is supported by the underlying protocol, then the Message should be unreliably transmitted. If the underlying protocol does not support unreliable transmission, the Message should be reliably transmitted.

* msgCapacityProfile: When true, this expresses a wish to override the Generic Connection Property connCapacityProfile for this Message. Depending on the value, this can, for example, be implemented by changing the DSCP value of the associated packet (note that the guidelines in Section 6 of [RFC7657] apply; e.g., the DSCP value should not be changed for different packets within a reliable transport protocol session or DCCP connection).

* noFragmentation: Setting this avoids network-layer fragmentation. Messages exceeding the transports current estimate of its maximum packet size (the singularTransmissionMsgMaxLen Connection Property) can result in transport segmentation when permitted, or generate an error. When used with transports running over IP version 4, the Don’t Fragment bit should be set to avoid on-path IP fragmentation ([RFC8304]).

* noSegmentation: When set, this property limits the Message size to the transports current estimate of its maximum packet size (the singularTransmissionMsgMaxLen Connection Property). Messages larger than this size generate an error. Setting this avoids transport-layer segmentation and network-layer fragmentation. When used with transports running over IP version 4, the Don't Fragment bit should be set to avoid on-path IP fragmentation ([RFC8304]).

5.1.2. Send Completion

The application should be notified (using a Sent, Expired or SendError event) whenever a Message or partial Message has been consumed by the Protocol Stack, or has failed to send. The time at which a Message is considered to have been consumed by the Protocol Stack may vary depending on the protocol. For example, for a basic datagram protocol like UDP, this may correspond to the time when the packet is sent into the interface driver. For a protocol that buffers data in queues, like TCP, this may correspond to when the data has entered the send buffer. The time at which a Message failed to send is when the Transport Services Implementation (including the Protocol Stack) has experienced a failure related to sending; this can depend on protocol-specific timeouts.

5.1.3. Batching Sends

Sending multiple Messages can incur high overhead if each needs to be enqueued separately (e.g., each Message might involve a context switch between the application and the Transport Services System). To avoid this, the application can indicate a batch of Send actions through the API. When this is used, the implementation can defer the processing of Messages until the batch is complete.

5.2. Receiving Messages

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

the flow of received data by specifying the minimum and maximum number of bytes of Message content it wants to receive at one time.

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

5.3. Handling of data for fast-open protocols

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

The application can express its preference for sending messagess as 0-RTT data by using the zeroRttMsg Selection Property on the Preconnection. Then, the application can provide the message to send as 0-RTT data via the InitiateWithSend action. In order to be sent as 0-RTT data, the message needs to be marked with the safelyReplayable send paramteter. In general, 0-RTT data may be replayed (for example, if a TCP SYN contains data, and the SYN is retransmitted, the data will be retransmitted as well but may be considered as a new connection instead of a retransmission). When racing connections, different leaf nodes have the opportunity to send the same data independently. If data is truly safely replayable, this is permissible.

Once the application has provided its 0-RTT data, a Transport Services Implementation should keep a copy of this data and provide it to each new leaf node that is started and for which a protocol instance supporting 0-RTT is being used. Note that the amount of data that can actually be sent as 0-RTT data varies by protocol, so any given Protocol Stack might only consume part of the saved data prior to becoming established. The implementation needs to keep track of how much data a particular Protocol Stack has consumed, and ensure that any pending 0-RTT-eligible data from the application is handled before subsequent Messages.

It is also possible for Protocol Stacks within a particular leaf node to use a 0-RTT handshakes in a lower-level protocol without any safely replayable application data if a higher-level protocol in the stack has idempotent handshake data to send. For example, TCP Fast Open could use a Client Hello from TLS as its 0-RTT data, without any data being provided by the application.

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

6. Implementing Message Framers

Message Framers are functions that define simple transformations between application Message data and raw transport protocol data. Generally, a Message Framer implements a simple application protocol that can either be provided by the Transport Services implementation or by the application. It is optional for Transport Services system implementations to provide Message Framers: the specification [I-D.ietf-taps-interface] does not prescribe any particular Message Framers to be implemented. A Framer can encapsulate or encode outbound Messages, decapsulate or decode inbound data into Messages, and implement parts of protocols that do not directly map to application Messages (such as protocol handshakes or preludes before Message exchange).

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

Most Message Framers fall into one of two categories:

* Header-prefixed record formats, such as a basic Type-Length-Value (TLV) structure

* Delimiter-separated formats, such as HTTP/1.1

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

6.1. Defining Message Framers

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

The API examples in this section use the notation conventions for the Transport Services API defined in Section 1.1 of [I-D.ietf-taps-interface].

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

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

```
MessageFramer -> Start<connection>
MessageFramer -> Stop<connection>
```

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

Once the framer implementation has completed its setup or handshake, it can indicate to the application that it is ready for handling data with this call.

```
MessageFramer.MakeConnectionReady(connection)
```

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

```
MessageFramer.MakeConnectionClosed(connection)
```

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

```
MessageFramer.FailConnection(connection, error)
```

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

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

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

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

```
MessageFramer.StartPassthrough()
```

6.2. Sender-side Message Framing

Message Framers generate an event whenever a Connection sends a new Message. The parameters to the event align with the Send action in the API (Section 9.2 of [I-D.ietf-taps-interface]).


```
NewSentMessage<connection, messageData, messageContext, endOfMessage>
```

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

MessageFramer.Send(connection, messageData)

To provide an example, a simple protocol that adds the length of the Message data 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.

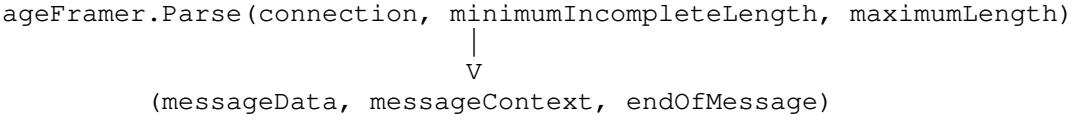
6.3. Receiver-side Message Framing

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.

The parameters to the events and calls for receiving data with a framer align with the Receive action in the API (Section 9.3 of [I-D.ietf-taps-interface]).

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.



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,
endOfMessage)
MessageFramer.Deliver(connection, messageContext, messageData, endOfMessage)
```

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

7. Implementing Connection Management

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

The set of Connection Properties that are supported for setting and getting on a Connection are described in [I-D.ietf-taps-interface]. For any properties that are generic, and thus could apply to all protocols being used by a Connection, the Transport Services Implementation should store the properties in storage common to all protocols, and notify the Protocol Stack as a whole whenever the properties have been modified by the application. [RFC8303] and [RFC8304] offer guidance on how to do this for TCP, MPTCP, SCTP, UDP and UDP-Lite; see Section 10 for a description of a back-tracking method to find the relevant protocol primitives using these documents. For Protocol-specific Properties, such as the User Timeout that applies to TCP, the Transport Services Implementation only needs to update the relevant protocol instance.

Some Connection Properties might apply to multiple protocols within a Protocol Stack. Depending on the specific property, it might be appropriate to apply the property across multiple protocols simultaneously, or else only apply it to one protocol. In general, the Transport Services Implementation should allow the protocol closest to the application to interpret Connection Properties, and potentially modify the set of Connection Properties passed down to the next protocol in the stack. For example, if the application has

requested to use keepalives with the keepAlive property, and the Protocol Stack contains both HTTP/2 and TCP, the HTTP/2 protocol can choose to enable its own keepalives to satisfy the application request, and disable TCP-level keepalives. For cases where the application needs to have fine-grained per-protocol control, the Transport Services Implementation can expose Protocol-specific Properties.

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 must fail gracefully. The application must be informed of the error, but the Connection itself must not be terminated.

When protocol instances in the Protocol Stack report generic or protocol-specific errors, the API will deliver them to the application as SoftError events. These allow the application to be informed of ICMP errors, and other similar events.

7.1. Pooled Connection

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

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

7.2. Handling Path Changes

When a path change occurs, e.g., when the IP address of an interface changes or a new interface becomes available, the Transport Services Implementation is responsible for notifying the Protocol Instance of the change. The path change may interrupt connectivity on a path for an active Connection or provide an opportunity for a transport that supports multipath or migration to adapt to the new paths. Note

that, in the model of the Transport Services API, migration is considered a part of multipath connectivity; it is just a limiting policy on multipath usage. If the multipath Selection Property is set to Disabled, migration is disallowed.

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

If the Protocol Stack includes a transport protocol that supports multipath connectivity, the Transport Services Implementation should also inform the Protocol Instance about potentially new paths that become permissible based on the multipath Selection Property and the multipathPolicy Connection Property choices made by the application. A protocol can then establish new subflows over new paths while an active path is still available or, if migration is supported, also after a break has been detected, and should attempt to tear down subflows over paths that are no longer used. The Connection Property multipathPolicy of the Transport Services API allows an application to indicate when and how different paths should be used. However, detailed handling of these policies is implementation-specific. For example, if the multipath Selection Property is set to active, the decision about when to create a new path or to announce a new path or set of paths to the Remote Endpoint, e.g., in the form of additional IP addresses, is implementation-specific. If the Protocol Stack includes a transport protocol that does not support multipath, but does support migrating between paths, the update to the set of available paths can trigger the connection to be migrated.

In the case of a Pooled Connection Section 7.1, the Transport Services Implementation may add connections over new paths to the pool if permissible based on the multipath policy and Selection Properties. In the case that a previously used path becomes unavailable, the Transport Services system may disconnect all connections that require this path, but should not disconnect the

pooled Connection object exposed to the application. The strategy to do so is implementation-specific, but should be consistent with the behavior of multipath transports.

8. Implementing Connection Termination

For Close (which leads to a Closed event) and Abort (which leads to a ConnectionError event), the application might find it useful 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. When an underlying transport connection supports multi-streaming (such as SCTP), the Transport Services system can use a 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 Identifer (if applicable). However, performance characteristics of a path are more likely tied to the IP address and subnet being used.

9.1. Protocol state caches

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

Examples of cached protocol state include:

* The DNS protocol can cache resolved addresses (such as those retrieved from A and AAAA queries), associated with a Time To Live (TTL) to be used for future hostname resolutions without requiring asking the DNS resolver again.

* TLS caches session state and tickets based on a hostname, which can be used for resuming sessions with a server.

* TCP can cache cookies for use in TCP Fast Open.

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

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

9.2. Performance caches

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

* Observed Round Trip Time

* Connection establishment latency

* Connection establishment success rate

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

An implementation should use this information, when possible, to influence preference between candidate paths, endpoints, and protocol options. Eligible options that historically had significantly better

performance than others should be selected first when gathering candidates (see Section 4.2) to ensure better performance for the application.

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

[RFC9040] provides guidance about sharing of TCP Control Block information between connections on initialization.

10. Specific Transport Protocol Considerations

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

Each protocol has a notion of Connectedness. Possible definitions of Connectedness for various types of protocols are:

* Connectionless. Connectionless protocols do not establish explicit state between endpoints, and do not perform a handshake during Connection establishment.

* Connected. Connected (also called "connection-oriented") protocols establish state between endpoints, and perform a handshake during connection establishment. The handshake may be 0-RTT to send data or resume a session, but bidirectional traffic is required to confirm connectedness.

* Multiplexing Connected. Multiplexing Connected protocols share properties with Connected protocols, but also explictly support opening multiple application-level flows. This means that they can support cloning new Connection objects without a new explicit handshake.

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

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

* Datagram. Datagram protocols define message boundaries at the same level of transmission, such that only complete (not partial) messages are supported.

* Message. Message protocols support message boundaries that can be sent and received either as complete or partial messages. Maximum message lengths can be defined, and messages can be partially reliable.

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

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

10.1. TCP

Connectedness: Connected

Data Unit: Byte-stream

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

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

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

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

EstablishmentError: Failure of CONNECT.TCP. TCP can throw various

errors during connection setup. Specifically, it is important to handle a RST being sent by the peer during the handshake.

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

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

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

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

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

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

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

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

CloseGroup: Calling CloseGroup on a TCP Connection (CLOSE.TCP) is identical to calling Close on this Connection and on all Connections in the same ConnectionGroup.

AbortGroup: Calling AbortGroup on a TCP Connection (ABORT.TCP) is identical to calling Abort on this Connection and on all Connections in the same ConnectionGroup.

10.2. MPTCP

Connectedness: Connected

Data Unit: Byte-stream

The Transport Services API mappings for MPTCP are identical to TCP. MPTCP adds support for multipath properties, such as multipath and multipathPolicy, and actions for managing paths, such as AddRemote and RemoveRemote.

10.3. UDP

Connectedness: Connectionless

Data Unit: Datagram

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.

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

ConnectionError: UDP Connections can only generate Connection errors in response to Abort calls. (Once in use, UDP Connections can also generate SoftError events (ERROR.UDP) 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. 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), DF in IPv4 (SET_DF.UDP) and ECN flag (SET_ECN.UDP).

Receive: RECEIVE.UDP. 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).

Close: Calling Close on a UDP Connection (ABORT.UDP) releases the local port reservation. The Connection then issues a Closed event.

Abort: Calling Abort on a UDP Connection (ABORT.UDP) is identical to calling Close, except that the Connection will send a ConnectionError event rather than a Closed event.

CloseGroup: Calling CloseGroup on a UDP Connection (ABORT.UDP) is identical to calling Close on this Connection and on all Connections in the same ConnectionGroup.

AbortGroup: Calling AbortGroup on a UDP Connection (ABORT.UDP) is identical to calling Close on this Connection and on all Connections in the same ConnectionGroup.

10.4. UDP-Lite

Connectedness: Connectionless

Data Unit: Datagram

The Transport Services API mappings for UDP-Lite are identical to UDP. In addition, UDP-Lite supports the msgChecksumLen and recvChecksumLen Properties that allow an application to specify the minimum number of bytes in a Message that need to be covered by a checksum.

This includes: CONNECT.UDP-Lite; LISTEN.UDP-Lite; SEND.UDP-Lite; RECEIVE.UDP-Lite; ABORT.UDP-Lite; ERROR.UDP-Lite; SET_DSCP.UDP-Lite; SET_DF.UDP-Lite; SET_ECN.UDP-Lite; GET_ECN.UDP-Lite.

10.5. UDP Multicast Receive

Connectedness: Connectionless

Data Unit: Datagram

Connection Object: Established UDP Multicast Receive connections represent a pair of specific IP addresses and ports. The direction Selection Property must be set to unidirectional receive, and the Local Endpoint must be configured with a group IP address and a port.

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

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

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

EstablishmentError: UDP Multicast Receive Connections generate an EstablishmentError indicating that joining a multicast group failed if Initiate is called.

ConnectionError: The only ConnectionError generated by a UDP Multicast Receive Connection is in response to an Abort call.

Listen: LISTEN.UDP. Calling Listen for UDP Multicast Receive binds

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

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

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

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

Send: SEND.UDP. Calling Send on a UDP Multicast Receive connection causes an immediate SendError. This is an unsupported operation.

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

Close: Calling Close on a UDP Multicast Receive Connection (ABORT.UDP) releases the local port reservation and leaves the group. The Connection then issues a Closed event.

Abort: Calling Abort on a UDP Multicast Receive Connection (ABORT.UDP) is identical to calling Close, except that the Connection will send a ConnectionError event rather than a Closed event.

CloseGroup: Calling CloseGroup on a UDP Multicast Receive Connection (ABORT.UDP) is identical to calling Close on this Connection and on all Connections in the same ConnectionGroup.

AbortGroup: Calling AbortGroup on a UDP Multicast Receive Connection

(ABORT.UDP) is identical to calling Close on this Connection and on all Connections in the same ConnectionGroup.

10.6. SCTP

Connectedness: Connected

Data Unit: Message

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

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

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

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

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

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

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

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

EstablishmentError: Failure of CONNECT.SCTP.

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

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

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

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

Send: SEND.SCTP. Message Properties such as msgLifetime and

msgOrdered map to parameters of this primitive.

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

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

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

CloseGroup: Calling CloseGroup calls CLOSE.SCTP, closing all Connections in the SCTP association.

AbortGroup: Calling AbortGroup calls ABORT.SCTP, immediately closing all Connections in the SCTP association.

In addition to the API mappings described above, when there are multiple Connection objects assigned to the same SCTP association, SCTP can support Connection properties such as connPriority and connScheduler where CONFIGURE_STREAM_SCHEDULER.SCTP can be called to adjust the priorities of streams in the SCTP association.

11. IANA Considerations

This document has no actions for IANA.

12. Security Considerations

[I-D.ietf-taps-arch] outlines general security consideration and requirements for any system that implements the Transport Services archtecture. [I-D.ietf-taps-interface] provides further discussion on security and privacy implications of the Transport Services API. This document provides additional guidance on implementation specifics for the Transport Services API and as such the security

considerations in both of these documents apply. The next two subsections discuss further considerations that are specific to mechanisms specified in this document.

12.1. Considerations for Candidate Gathering

The Security Considerations of the Transport Services Architecture [I-D.ietf-taps-arch] forbids gathering and racing with Protocol Stacks that do not have equivalent security properties. Therefore, implementations need to avoid downgrade attacks that allow network interference to cause the implementation to select less secure, or entirely insecure, combinations of paths and protocols.

12.2. Considerations for Candidate Racing

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

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

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

13. Acknowledgements

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

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

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

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

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

14. References

14.1. Normative References

[I-D.ietf-taps-arch]
Pauly, T., Trammell, B., Brunstrom, A., Fairhurst, G., and C. Perkins, "Architecture and Requirements for Transport Services", Work in Progress, Internet-Draft, draft-ietf-taps-arch-19, 9 November 2023, <https://datatracker.ietf.org/doc/html/draft-ietf-taps-arch-19>.

[I-D.ietf-taps-interface]
Trammell, B., Welzl, M., Enghardt, R., Fairhurst, G., Kühlewind, M., Perkins, C., Tiesel, P. S., and T. Pauly, "An Abstract Application Layer Interface to Transport Services", Work in Progress, Internet-Draft, draft-ietf-taps-interface-23, 14 November 2023, <https://datatracker.ietf.org/doc/html/draft-ietf-taps-interface-23>.

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

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

[RFC8303] Welzl, M., Tuexen, M., and N. Khademi, "On the Usage of Transport Features Provided by IETF Transport Protocols", RFC 8303, DOI 10.17487/RFC8303, February 2018, <https://www.rfc-editor.org/rfc/rfc8303>.

[RFC8304] Fairhurst, G. and T. Jones, "Transport Features of the User Datagram Protocol (UDP) and Lightweight UDP (UDP-Lite)", RFC 8304, DOI 10.17487/RFC8304, February 2018, <https://www.rfc-editor.org/rfc/rfc8304>.

[RFC8305] Schinazi, D. and T. Pauly, "Happy Eyeballs Version 2: Better Connectivity Using Concurrency", RFC 8305, DOI 10.17487/RFC8305, December 2017, <https://www.rfc-editor.org/rfc/rfc8305>.

[RFC8421] Martinsen, P., Reddy, T., and P. Patil, "Guidelines for Multihomed and IPv4/IPv6 Dual-Stack Interactive Connectivity Establishment (ICE)", BCP 217, RFC 8421, DOI 10.17487/RFC8421, July 2018, <https://www.rfc-editor.org/rfc/rfc8421>.

[RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <https://www.rfc-editor.org/rfc/rfc8446>.

[RFC8923] Welzl, M. and S. Gjessing, "A Minimal Set of Transport Services for End Systems", RFC 8923, DOI 10.17487/RFC8923, October 2020, <https://www.rfc-editor.org/rfc/rfc8923>.

14.2. Informative References

[I-D.ietf-dnsop-svcb-https] Schwartz, B. M., Bishop, M., and E. Nygren, "Service Binding and Parameter Specification via the DNS (SVCB and HTTPS Resource Records)", Work in Progress, Internet-Draft, draft-ietf-dnsop-svcb-https-12, 11 March 2023, <https://datatracker.ietf.org/doc/html/draft-ietf-dnsop-svcb-https-12>.

[NEAT-flow-mapping] "Transparent Flow Mapping for NEAT", IFIP NETWORKING 2017 Workshop on Future of Internet Transport (FIT 2017) , 2017.

[RFC1928] Leech, M., Ganis, M., Lee, Y., Kuris, R., Koblas, D., and L. Jones, "SOCKS Protocol Version 5", RFC 1928, DOI 10.17487/RFC1928, March 1996, <https://www.rfc-editor.org/rfc/rfc1928>.

[RFC2782] Gulbrandsen, A., Vixie, P., and L. Esibov, "A DNS RR for specifying the location of services (DNS SRV)", RFC 2782, DOI 10.17487/RFC2782, February 2000, <https://www.rfc-editor.org/rfc/rfc2782>.

[RFC3124] Balakrishnan, H. and S. Seshan, "The Congestion Manager", RFC 3124, DOI 10.17487/RFC3124, June 2001, <https://www.rfc-editor.org/rfc/rfc3124>.

[RFC3207] Hoffman, P., "SMTP Service Extension for Secure SMTP over Transport Layer Security", RFC 3207, DOI 10.17487/RFC3207, February 2002, <https://www.rfc-editor.org/rfc/rfc3207>.

[RFC5389] Rosenberg, J., Mahy, R., Matthews, P., and D. Wing, "Session Traversal Utilities for NAT (STUN)", RFC 5389, DOI 10.17487/RFC5389, October 2008, <https://www.rfc-editor.org/rfc/rfc5389>.

[RFC5766] Mahy, R., Matthews, P., and J. Rosenberg, "Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN)", RFC 5766, DOI 10.17487/RFC5766, April 2010, <https://www.rfc-editor.org/rfc/rfc5766>.

[RFC6525] Stewart, R., Tuexen, M., and P. Lei, "Stream Control Transmission Protocol (SCTP) Stream Reconfiguration", RFC 6525, DOI 10.17487/RFC6525, February 2012, <https://www.rfc-editor.org/rfc/rfc6525>.

[RFC6762] Cheshire, S. and M. Krochmal, "Multicast DNS", RFC 6762, DOI 10.17487/RFC6762, February 2013, <https://www.rfc-editor.org/rfc/rfc6762>.

[RFC6763] Cheshire, S. and M. Krochmal, "DNS-Based Service Discovery", RFC 6763, DOI 10.17487/RFC6763, February 2013, <https://www.rfc-editor.org/rfc/rfc6763>.

[RFC7230] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, DOI 10.17487/RFC7230, June 2014, <https://www.rfc-editor.org/rfc/rfc7230>.

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

[RFC8085] Eggert, L., Fairhurst, G., and G. Shepherd, "UDP Usage Guidelines", BCP 145, RFC 8085, DOI 10.17487/RFC8085, March 2017, <https://www.rfc-editor.org/rfc/rfc8085>.

[RFC8260] Stewart, R., Tuexen, M., Loreto, S., and R. 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/rfc/rfc8260>.

[RFC8445] Keranen, A., Holmberg, C., and J. Rosenberg, "Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal", RFC 8445, DOI 10.17487/RFC8445, July 2018, <https://www.rfc-editor.org/rfc/rfc8445>.

[RFC9000] Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport", RFC 9000, DOI 10.17487/RFC9000, May 2021, <https://www.rfc-editor.org/rfc/rfc9000>.

[RFC9040] Touch, J., Welzl, M., and S. Islam, "TCP Control Block Interdependence", RFC 9040, DOI 10.17487/RFC9040, July 2021, <https://www.rfc-editor.org/rfc/rfc9040>.

[TCP-COUPLING] "ctrlTCP: Reducing Latency through Coupled, Heterogeneous Multi-Flow TCP Congestion Control", IEEE INFOCOM Global Internet Symposium (GI) workshop (GI 2018) , n.d..

Appendix A. API Mapping Template

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

Connectedness: (Connectionless/Connected/Multiplexing Connected)

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

Connection Object:

Initiate:

InitiateWithSend:

Ready:

EstablishmentError:

ConnectionError:

Listen:

ConnectionReceived:

Clone:

Send:

Receive:

Close:

Abort:

CloseGroup:

AbortGroup:

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 about why an error occurred. This appendix lists some of the possible reasons.

* InvalidConfiguration: The transport properties and Endpoint Identifers provided by the application are either contradictory or incomplete. Examples include the lack of a Remote Endpoint Identifer on an active open or using a multicast group address while not requesting a unidirectional receive.

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

* ResolutionFailed: The remote or local specifier provided by the application can not be resolved.

* EstablishmentFailed: The Transport Services system was unable to establish a transport-layer connection to the Remote Endpoint specified by the application.

* PolicyProhibited: The system policy prevents the Transport Services system from performing the action requested by the application.

* NotCloneable: The Protocol Stack is not capable of being cloned.

* MessageTooLarge: The Message is too big for the Transport Services system to handle.

* ProtocolFailed: The underlying Protocol Stack failed.

* InvalidMessageProperties: The Message Properties either contradict the Transport Properties or they can not be satisfied by the Transport Services system.

* DeframingFailed: The data that was received by the underlying Protocol Stack could not be processed by the Message Framer.

* ConnectionAborted: The connection was aborted by the peer.

* 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 Services systems that are (to some degree) in line with this document.

* Apple's Network.framework:

  - Network.framework is a transport-level API built for C, Objective-C, and Swift. It a connect-by-name API that supports transport security protocols. It provides userspace implementations of TCP, UDP, TLS, DTLS, proxy protocols, and allows extension via custom framers.

  - Documentation: https://developer.apple.com/documentation/network (https://developer.apple.com/documentation/network)

* NEAT and NEATPy:

  - NEAT is the output of the European H2020 research project "NEAT"; it is a user-space library for protocol-independent communication on top of TCP, UDP and SCTP, with many more features, such as a policy manager.

  - Code: https://github.com/NEAT-project/neat (https://github.com/NEAT-project/neat)

  - Code at the Software Heritage Archive: https://archive.softwareheritage.org/swh:1:dir:737820840f83c4ec9493a8c0cc89b3159e2e1a57;origin=https://github.com/NEAT-project/neat;visit=swh:1:snp:bbb611b04e355439d47e426e8ad5d07cdbf647e0;anchor=swh:1:rev:652ee991043ce3560a6e5715fa2a5c211139d15c (https://archive.softwareheritage.org/swh:1:dir:737820840f83c4ec9493a8c0cc89b3159e2e1a57;origin=https://github.com/NEAT-project/neat;visit=swh:1:snp:bbb611b04e355439d47e426e8ad5d07cdbf647e0;anchor=swh:1:rev:652ee991043ce3560a6e5715fa2a5c211139d15c)

  - NEAT project: https://www.neat-project.org (https://www.neat-project.org)

- NEATPy is a Python shim over NEAT which updates the NEAT API to be in line with version 6 of the Transport Services API draft.

- Code: https://github.com/theagilepadawan/NEATPy (https://github.com/theagilepadawan/NEATPy)

- Code at the Software Heritage Archive: https://archive.softwareheritage.org/swh:1:dir:295ccd148cf918ccb9ed7ad14b5ae968a8d2c370;origin=https://github.com/theagilepadawan/NEATPy;visit=swh:1:snp:6e1a3a9dd4c532ba6c0f52c8f734c1256a06cedc;anchor=swh:1:rev:cd0788d7f7f34a0e9b8654516da7c002c44d2e95 (https://archive.softwareheritage.org/swh:1:dir:295ccd148cf918ccb9ed7ad14b5ae968a8d2c370;origin=https://github.com/theagilepadawan/NEATPy;visit=swh:1:snp:6e1a3a9dd4c532ba6c0f52c8f734c1256a06cedc;anchor=swh:1:rev:cd0788d7f7f34a0e9b8654516da7c002c44d2e95)

* PyTAPS:

  - A TAPS implementation based on Python asyncio, offering protocol-independent communication to applications on top of TCP, UDP and TLS, with support for multicast.

  - Code: https://github.com/fg-inet/python-asyncio-taps (https://github.com/fg-inet/python-asyncio-taps)

  - Code at the Software Heritage Archive: https://archive.softwareheritage.org/swh:1:dir:a7151096d91352b439b092ef116d04f38e52e556;origin=https://github.com/fg-inet/python-asyncio-taps;visit=swh:1:snp:4841e59b53b28bb385726e7d3a569bee0fea7fc4;anchor=swh:1:rev:63571fd7545da25142bc1a6371b8f13097cba38e (https://archive.softwareheritage.org/swh:1:dir:a7151096d91352b439b092ef116d04f38e52e556;origin=https://github.com/fg-inet/python-asyncio-taps;visit=swh:1:snp:4841e59b53b28bb385726e7d3a569bee0fea7fc4;anchor=swh:1:rev:63571fd7545da25142bc1a6371b8f13097cba38e)

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

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

Philipp S. Tiesel
SAP SE
George-Stephenson-Straße 7-13
10557 Berlin
Germany
Email: philipp@tiesel.net

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

TAPS Working Group
Internet-Draft
Intended status: Standards Track
Expires: 18 September 2024

B. Trammell, Ed.
Google Switzerland GmbH
M. Welzl, Ed.
University of Oslo
R. Enghardt
Netflix
G. Fairhurst
University of Aberdeen
M. Kuehlewind
Ericsson
C. Perkins
University of Glasgow
P. Tiesel
SAP SE
T. Pauly
Apple Inc.
17 March 2024

An Abstract Application Layer Interface to Transport Services
draft-ietf-taps-interface-26

Abstract

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

This Internet-Draft will expire on 18 September 2024.

Table of Contents

1. Introduction

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

Applications that adopt this API will benefit from a wide set of transport features that can evolve over time. This protocol-independent API ensures that the system providing the API can optimize its behavior based on the application requirements and network conditions, without requiring changes to the applications. This flexibility enables faster deployment of new features and protocols, and can support applications by offering racing and fallback mechanisms, which otherwise need to be separately implemented in each application. Transport Services Implementations are free to take any desired form as long as the API specification in this document is honored; a nonprescriptive guide to implementing a Transport Services system is available [I-D.ietf-taps-impl].

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

1.1. Terminology and Notation

The Transport Services API is described in terms of

* Objects with which an application can interact;
* Actions the application can perform on these objects;
* Events, which an object can send to an application to be processed asynchronously; and
* Parameters associated with these actions and events.

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

* An action that creates an object:

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

* An action that creates an array of objects:

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

* An action that is performed on an object:

  ```
  Object.Action()
  ```

* An object sends an event:

  ```
  Object -> Event<>
  ```

* An action takes a set of Parameters; an event contains a set of Parameters. action and event parameters whose names are suffixed with a question mark are optional.

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

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

Events are sent to the application or application-supplied code (e.g. framers, see Section 9.1.2) for processing; the details of event interfaces are platform- and implementation-specific, and can be implemented using other forms of asynchronous processing, as idiomatic for the implementing platform.

We also make use of the following basic types:

* Boolean: Instances take the value true or false.

* Integer: Instances take integer values.

* Numeric: Instances take real number values.

* String: Instances are represented in UTF-8.

* IP Address: An IPv4 [RFC791] or IPv6 [RFC4291] address.

* Enumeration: A family of types in which each instance takes one of a fixed, predefined set of values specific to a given enumerated type.

* Tuple: An ordered grouping of multiple value types, represented as a comma-separated list in parentheses, e.g., (Enumeration, Preference). Instances take a sequence of values each valid for the corresponding value type.

* Array: Denoted []Type, an instance takes a value for each of zero or more elements in a sequence of the given Type. An array can be of fixed or variable length.

* Set: An unordered grouping of one or more different values of the same type.

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

1.2. Specification of Requirements

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

2. Overview of the API Design

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

* A Transport Services system that can offer a variety of transport protocols, independent of the Protocol Stacks that will be used at runtime. To the degree possible, all common features of these protocol stacks are made available to the application in a transport-independent way. This enables applications written to a single API to make use of transport protocols in terms of the features they provide.

* A unified API to datagram and stream-oriented transports, allowing use of a common API for Connection establishment and closing.

* Message-orientation, as opposed to stream-orientation, using application-assisted framing and deframing where the underlying transport does not provide these.

* Asynchronous Connection establishment, transmission, and reception. This allows concurrent operations during establishment and event-driven application interactions with the transport layer.

* Selection between alternate network paths, using additional information about the networks over which a Connection can operate (e.g. Provisioning Domain (PvD) information [RFC7556]) where available.

* Explicit support for transport-specific features to be applied, when that particular transport is part of a chosen Protocol Stack.

* Explicit support for security properties as first-order transport features.

* Explicit support for configuration of cryptographic identities and transport security parameters persistent across multiple Connections.

* Explicit support for multistreaming and multipath transport protocols, and the grouping of related Connections into Connection Groups through "cloning" of Connections (see Section 7.4). This function allows applications to take full advantage of new transport protocols supporting these features.

3. API Summary

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

* by initiating the Preconnection (i.e., creating a Connection from the Preconnection, actively opening, as in a client; see Initiate() in Section 7.1),

* by listening on the Preconnection (i.e., creating a Listener based on the Preconnection, passively opening, as in a server; see Listen() in Section 7.2),

* or by a rendezvous for the Preconnection (i.e., peer to peer establishment; see Rendezvous() in Section 7.3).

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

The application SHOULD NOT assume that ignoring events (e.g., errors) is always safe.

3.1. Usage Examples

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

* Act as a server, by listening for incoming Connections, receiving requests, and sending responses, see Section 3.1.1.

* Act as a client, by connecting to a Remote Endpoint using Initiate, sending requests, and receiving responses, see Section 3.1.2.

* Act as a peer, by connecting to a Remote Endpoint using Rendezvous while simultaneously waiting for incoming Connections, sending Messages, and receiving Messages, see Section 3.1.3.

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

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

3.1.1. Server Example

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

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

TransportProperties := NewTransportProperties()
TransportProperties.Require(preserveMsgBoundaries)
// Reliable Data Transfer and Preserve Order are Required by default

SecurityParameters := NewSecurityParameters()
SecurityParameters.Set(serverCertificate, myCertificate)

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

Listener := Preconnection.Listen()

Listener -> ConnectionReceived<Connection>

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

Connection -> Received<messageDataRequest, messageContext>

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

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

3.1.2. Client Example

This is an example of how an application might open two Connections to a remote application using the Transport Services API, and send a request as well as receive a response on each of them. The code designated with comments as "Ready event handler" could, e.g., be implemented as a callback function, for example. This function would receive the Connection that it expects to operate on ("Connection" and "Connection2" in the example), handed over using the variable name "C".

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

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

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

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

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

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

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

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

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

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

A Preconnection serves as a template for creating a Connection via initiating, listening, or via rendezvous. Once a Connection has been created, changes made to the Preconnection that was used to create it do not affect this Connection. Preconnections are reusable after being used to create a Connection, whether this Connection was closed or not. Hence, in the above example, it would be correct for the client to initiate a third Connection to the example.com server by continuing as follows:

```
   //.. carry out adjustments to the Preconnection, if desired
   Connection3 := Preconnection.Initiate()
```

3.1.3. Peer Example

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

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

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

   LocalCandidates = [HostCandidate, StunCandidate]

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

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

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

   // Application-specific code goes here to send the ResolvedLocal
   // list to peer via some out-of-band signalling channel (e.g.,
   // in a SIP message)
   ...

   // Application-specific code goes here to receive RemoteCandidates
   // (type []RemoteEndpoint, a list of RemoteEndpoint objects) from
   // the peer via the signalling channel
   ...

   // Add remote candidates and initiate the rendezvous:
   Preconnection.AddRemote(RemoteCandidates)
   Preconnection.Rendezvous()

   Preconnection -> RendezvousDone<Connection>
```

```
//---- RendezvousDone event handler begin ----
Connection.Send(messageDataRequest)
Connection.Receive()
//---- RendezvousDone event handler end ----

Connection -> Received<messageDataResponse, messageContext>

// If new Remote Endpoint candidates are received from the
// peer over the signalling channel, for example if using
// Trickle ICE, then add them to the Connection:
Connection.AddRemote(NewRemoteCandidates)

// On a PathChange<> event, resolve the Local Endpoint Identifiers to
// see if a new Local Endpoint has become available and, if
// so, send to the peer as a new candidate and add to the
// Connection:
Connection -> PathChange<>

//---- PathChange event handler begin ----
ResolvedLocal, ResolvedRemote = Preconnection.Resolve()
if ResolvedLocal has changed:
  // Application-specific code goes here to send the
  // ResolvedLocal list to peer via signalling channel
  ...

  // Add the new Local Endpoints to the Connection:
  Connection.AddLocal(ResolvedLocal)
//---- PathChange event handler end ----


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

4. Transport Properties

Each application using the Transport Services API declares its preferences for how the Transport Services system is to operate. This is done by using Transport Properties, as defined in [I-D.ietf-taps-arch], at each stage of the lifetime of a Connection.

Transport Properties are divided into Selection, Connection, and Message Properties.

Selection Properties (see Section 6.2) can only be set during pre-establishment. They are only used to specify which paths and Protocol Stacks can be used and are preferred by the application. Calling Initiate on a Preconnection creates an outbound Connection, and the Selection Properties remain readable from the Connection, but

become immutable. Selection Properties can be set on Preconnections, and the effect of Selection Properties can be queried on Connections and Messages.

Connection Properties (see Section 8.1) are used to inform decisions made during establishment and to fine-tune the established Connection. They can be set during pre-establishment, and can be changed later. Connection Properties can be set on Connections and Preconnections; when set on Preconnections, they act as an initial default for the resulting Connections.

Message Properties (see Section 9.1.3) control the behavior of the selected protocol stack(s) when sending Messages. Message Properties can be set on Messages, Connections, and Preconnections; when set on the latter two, they act as an initial default for the Messages sent over those Connections.

Note that configuring Connection Properties and Message Properties on Preconnections is preferred over setting them later. Early specification of Connection Properties allows their use as additional input to the selection process. Protocol-specific Properties, which enable configuration of specialized features of a specific protocol (see Section 3.2 of [I-D.ietf-taps-arch]) are not used as an input to the selection process, but only support configuration if the respective protocol has been selected.

4.1. Transport Property Names

Transport Properties are referred to by property names, represented as case-insensitive strings. These names serve two purposes:

* Allowing different components of a Transport Services Implementation to pass Transport Properties, e.g., between a language frontend and a policy manager, or as a representation of properties retrieved from a file or other storage.

* Making the code of different Transport Services Implementations look similar. While individual programming languages might preclude strict adherence to the aforementioned naming convention (for instance, by prohibiting the use of hyphens in symbols), users interacting with multiple implementations will still benefit from the consistency resulting from the use of visually similar symbols.

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

* The optional Namespace component and its trailing character . MUST be omitted for well-known, generic properties, i.e., for properties that are not specific to a protocol.

* Protocol-specific Properties MUST use the protocol acronym as the Namespace (e.g., a Connection that uses TCP could support a TCP-specific Transport Property, such as the TCP user timeout value, in a Protocol-specific Property called tcp.userTimeoutValue (see Section 8.2)).

* Vendor or implementation specific properties MUST be placed in a Namespace starting with the underscore _ character and SHOULD use a string identifying the vendor or implementation.

* For IETF protocols, the name of a Protocol-specific Property MUST be specified in an IETF document published in the RFC Series after IETF review. An IETF protocol Namespace does not start with an underscore character.

Namespaces for each of the keywords provided in the IANA protocol numbers registry (see https://www.iana.org/assignments/protocol-numbers/protocol-numbers.xhtml) are reserved for Protocol-specific Properties and MUST NOT be used for vendor or implementation-specific properties. Terms listed as keywords as in the protocol numbers registry SHOULD be avoided as any part of a vendor- or implementation-specific property name.

Though Transport Property Names are case-insensitive, it is recommended to use camelCase to improve readability. Implementations may transpose Transport Property Names into snake_case or PascalCase to blend into the language environment.

4.2. Transport Property Types

Each Transport Property has one of the basic types described in Section 1.1.

Most Selection Properties (see Section 6.2) are of the Enumeration type, and use the Preference Enumeration, which takes one of five possible values (Prohibit, Avoid, No Preference, Prefer, or Require) denoting the level of preference for a given property during protocol selection.

5. Scope of the API Definition

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

There is no interoperability benefit in tightly defining how the API is presented to application programmers across diverse platforms. However, maintaining the "shape" of the abstract API across different platforms reduces the effort for programmers who learn to use the Transport Services API to then apply their knowledge to another platform. That said, implementations have significant freedom in presenting this API to programmers, balancing the conventions of the protocol with the shape of the API. We make the following recommendations:

* Actions, events, and errors in implementations of the Transport Services API SHOULD use the names given for them in the document, subject to capitalization, punctuation, and other typographic conventions in the language of the implementation, unless the implementation itself uses different names for substantially equivalent objects for networking by convention.

* Transport Services systems SHOULD implement each Selection Property, Connection Property, and Message Context Property specified in this document. These features SHOULD be implemented even when in a specific implementation it will always result in no operation, e.g. there is no action when the API specifies a Property that is not available in a transport protocol implemented on a specific platform. For example, if TCP is the only underlying transport protocol, the Message Property msgOrdered can be implemented (trivially, as a no-op) as disabling the requirement for ordering will not have any effect on delivery order for Connections over TCP. Similarly, the msgLifetime Message Property can be implemented but ignored, as the description of this Property states that "it is not guaranteed that a Message will not be sent when its Lifetime has expired".

* Implementations can use other representations for Transport Property Names, e.g., by providing constants, but should provide a straight-forward mapping between their representation and the property names specified here.

6. Pre-Establishment Phase

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

A Preconnection object represents a potential Connection. It is a passive object (a data structure) that merely maintains the state that describes the properties of a Connection that might exist in the future. This state comprises Local Endpoint and Remote Endpoint objects that denote the endpoints of the potential Connection (see Section 6.1), the Selection Properties (see Section 6.2), any preconfigured Connection Properties (Section 8.1), and the security parameters (see Section 6.3):

```
Preconnection := NewPreconnection([]LocalEndpoint,
                                  []RemoteEndpoint,
                                  TransportProperties,
                                  SecurityParameters)
```

At least one Local Endpoint MUST be specified if the Preconnection is used to Listen for incoming Connections, but the list of Local Endpoints MAY be empty if the Preconnection is used to Initiate connections. If no Local Endpoint is specified, the Transport Services system will assign an ephemeral local port to the Connection on the appropriate interface(s). At least one Remote Endpoint MUST be specified if the Preconnection is used to Initiate Connections, but the list of Remote Endpoints MAY be empty if the Preconnection is used to Listen for incoming Connections. At least one Local Endpoint and one Remote Endpoint MUST be specified if a peer-to-peer Rendezvous is to occur based on the Preconnection.

If more than one Local Endpoint is specified on a Preconnection, then the application is indicating that all of the Local Endpoints are eligible to be used for Connections. For example, their Endpoint Identifiers might correspond to different interfaces on a multi-homed host, or their Endpoint Identifiers might correspond to local interfaces and a STUN server that can be resolved to a server reflexive address for a Preconnection used to make a peer-to-peer Rendezvous.

If more than one Remote Endpoint is specified on the Preconnection, the application is indicating that it expects all of the Remote Endpoints to offer an equivalent service, and that the Transport Services system can choose any of them for a Connection. For example, a Remote Endpoint might represent various network interfaces of a host, or a server reflexive address that can be used to reach a host, or a set of hosts that provide equivalent local balanced service.

In most cases, it is expected that a single Remote Endpoint will be specified by name, and a later call to Initiate on the Preconnection (see Section 7.1) will internally resolve that name to a list of concrete Endpoint Identifers. Specifying multiple Remote Endpoints on a Preconnection allows applications to override this for more detailed control.

If Message Framers are used (see Section 9.1.2), they MUST be added to the Preconnection during pre-establishment.

6.1. Specifying Endpoints

The Transport Services API uses the Local Endpoint and Remote Endpoint objects to refer to the endpoints of a Connection. Endpoints can be created as either remote or local:

```
RemoteSpecifier := NewRemoteEndpoint()
LocalSpecifier := NewLocalEndpoint()
```

A single Endpoint object represents the identity of a network host. That endpoint can be more or less specific depending on which Endpoint Identifiers are set. For example, an Endpoint that only specifies a hostname can, in fact, finally correspond to several different IP addresses on different hosts.

An Endpoint object can be configured with the following identifiers:

* HostName (string):

```
RemoteSpecifier.WithHostName("example.com")
```

* Port (a 16-bit unsigned Integer):

```
RemoteSpecifier.WithPort(443)
```

* Service (an identifier string that maps to a port; either a service name associated with a port number, from https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml, or a DNS SRV service name to be resolved):

```
RemoteSpecifier.WithService("https")
```

* IP address (an IPv4 or IPv6 address type; note that the examples here show the human-readable form of the IP addresses, but the functions can take a binary encoding of the addresses):

```
RemoteSpecifier.WithIPAddress(192.0.2.21)

RemoteSpecifier.WithIPAddress(2001:db8:4920:e29d:a420:7461:7073:a)
```

* Interface identifier (which can be a string name or other platform-specific identifier), e.g., to qualify link-local addresses (see Section 6.1.2 for details):

```
LocalSpecifier.WithInterface("en0")
```

The Resolve action on a Preconnection can be used to obtain a list of available local interfaces.

Note that an IPv6 address specified with a scope zone ID (e.g. fe80::2001:db8%en0) is equivalent to WithIPAddress with an unscoped address and WithInterface together.

Applications creating Endpoint objects using WithHostName SHOULD provide fully-qualified domain names (FQDNs). Not providing an FQDN will result in the Transport Services Implementation needing to use DNS search domains for name resolution, which might lead to inconsistent or unpredictable behavior.

The design of the API MUST NOT permit an Endpoint object to be configured with multiple Endpoint Identifiers of the same type. For example, an Endpoint object cannot specify two IP addresses. Two separate IP addresses are represented as two Endpoint objects. If a Preconnection specifies a Remote Endpoint with a specific IP address set, it will only establish Connections to that IP address. If, on the other hand, a Remote Endpoint specifies a hostname but no addresses, the Transport Services Implementation can perform name resolution and attempt using any address derived from the original hostname of the Remote Endpoint. Note that multiple Remote Endpoints can be added to a Preconnection, as discussed in Section 7.5.

The Transport Services system resolves names internally, when the Initiate, Listen, or Rendezvous method is called to establish a Connection. Privacy considerations for the timing of this resolution are given in Section 13.

The Resolve action on a Preconnection can be used by the application to force early binding when required, for example with some Network Address Translator (NAT) traversal protocols (see Section 7.3).

6.1.1. Using Multicast Endpoints

To use multicast, a Preconnection is first created with the Local/Remote Endpoint Identifer specifying the any-source multicast (ASM) or source-specific multicast (SSM) multicast group and destination port number. This is then followed by a call to either Initiate, Listen, or Rendezvous depending on whether the resulting Connection is to be used to send messages to the multicast group, receive messages from the group, or, for an any-source multicast group, to both send and receive messages.

Note that the Transport Services API has separate specifier calls for multicast groups to avoid introducing filter properties for single-source multicast and seeks to avoid confusion that can be caused by overloading the unicast specifiers.

Calling Initiate on that Preconnection creates a Connection that can be used to send Messages to the multicast group. The Connection object that is created will support Send but not Receive. Any Connections created this way are send-only, and do not join the multicast group. The resulting Connection will have a Local Endpoint identifying the local interface to which the Connection is bound and a Remote Endpoint identifying the multicast group.

The following API calls can be used to configure a Preconnection before calling Initiate:

```
RemoteSpecifier.WithMulticastGroupIP(GroupAddress)
RemoteSpecifier.WithPort(PortNumber)
RemoteSpecifier.WithHopLimit(HopLimit)
```

Calling Listen on a Preconnection with a multicast group specified on the Remote Endpoint will join the multicast group to receive Messages. This Listener will create one Connection for each Remote Endpoint sending to the group, with the Local Endpoint Identifer specified as a group address. The set of Connection objects created forms a Connection Group. The receiving interface can be restricted by passing it as part of the LocalSpecifier or queried through the Message Context on the Messages received (see Section 9.1.1 for further details).

Specifying WithHopLimit sets the Time To Live (TTL) field in the header of IPv4 packets or the Hop Count field in the header of IPv6 packets.

The following API calls can be used to configure a Preconnection before calling Listen:

```
LocalSpecifier.WithSingleSourceMulticastGroupIP(GroupAddress,
                                                SourceAddress)
LocalSpecifier.WithAnySourceMulticastGroupIP(GroupAddress)
LocalSpecifier.WithPort(PortNumber)
```

Calling Rendezvous on a Preconnection with an any-source multicast group address as the Remote Endpoint Identifer will join the multicast group, and also indicates that the resulting Connection can be used to send Messages to the multicast group. The Rendezvous call will return both a Connection that can be used to send to the group, that acts the same as a Connection returned by calling Initiate with a multicast Remote Endpoint, and a Listener that acts as if Listen had been called with a multicast Remote Endpoint. Calling Rendezvous on a Preconnection with a source-specific multicast group address as the Local Endpoint Identifer results in an EstablishmentError.

The following API calls can be used to configure a Preconnection before calling Rendezvous:

```
RemoteSpecifier.WithMulticastGroupIP(GroupAddress)
RemoteSpecifier.WithPort(PortNumber)
RemoteSpecifier.WithHopLimit(HopLimit)
LocalSpecifier.WithAnySourceMulticastGroupIP(GroupAddress)
LocalSpecifier.WithPort(PortNumber)
LocalSpecifier.WithHopLimit(HopLimit)
```

See Section 6.1.5 for more examples.

6.1.2. Constraining Interfaces for Endpoints

Note that this API has multiple ways to constrain and prioritize endpoint candidates based on the network interface:

* Specifying an interface on a Remote Endpoint qualifies the scope zone of the Remote Endpoint, e.g., for link-local addresses.

* Specifying an interface on a Local Endpoint explicitly binds all candidates derived from this Endpoint to use the specified interface.

* Specifying an interface using the interface Selection Property (Section 6.2.11) or indirectly via the pvd Selection Property (Section 6.2.12) influences the selection among the available candidates.

While specifying an Interface on an Endpoint restricts the candidates available for Connection establishment in the Pre-Establishment Phase, the Selection Properties prioritize and constrain the Connection establishment.

6.1.3. Protocol-Specific Endpoints

An Endpoint can have an alternative definition when using different protocols. For example, a server that supports both TLS/TCP and QUIC could be accessible on two different port numbers depending on which protocol is used.

To scope an Endpoint to apply conditionally to a specific transport protocol (such as defining an alternate port to use when QUIC is selected, as opposed to TCP), an Endpoint can be associated with a protocol identifier. Protocol identifiers are objects or enumeration values provided by the Transport Services API, which will vary based on which protocols are implemented in a particular system.

```
AlternateRemoteSpecifier.WithProtocol(QUIC)
```

The following example shows a case where example.com has a server running on port 443, with an alternate port of 8443 for QUIC. Both endpoints can be passed when creating a Preconnection.

```
RemoteSpecifier := NewRemoteEndpoint()
RemoteSpecifier.WithHostName("example.com")
RemoteSpecifier.WithPort(443)

QUICRemoteSpecifier := NewRemoteEndpoint()
QUICRemoteSpecifier.WithHostName("example.com")
QUICRemoteSpecifier.WithPort(8443)
QUICRemoteSpecifier.WithProtocol(QUIC)

RemoteSpecifiers := [ RemoteSpecifier, QUICRemoteSpecifier ]
```

6.1.4. Endpoint Examples

The following examples of Endpoints show common usage patterns.

Specify a Remote Endpoint using a hostname "example.com" and a service name "https", which tells the system to use the default port for HTTPS (443):

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

Specify a Remote Endpoint using an IPv6 address and remote port:

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

Specify a Remote Endpoint using an IPv4 address and remote port:

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

Specify a Local Endpoint using a local interface name and no local port, to let the system assign an ephemeral local port:

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

Specify a Local Endpoint using a local interface name and local port:

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

As an alternative to specifying an interface name for the Local Endpoint, an application can express more fine-grained preferences using the Interface Instance or Type Selection Property, see Section 6.2.11. However, if the application specifies Selection Properties that are inconsistent with the Local Endpoint, this will result in an error once the application attempts to open a Connection.

Specify a Local Endpoint using a STUN server:

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

6.1.5. Multicast Examples

The following examples show how multicast groups can be used.

Join an Any-Source Multicast group in receive-only mode, bound to a known port on a named local interface:

```
RemoteSpecifier := NewRemoteEndpoint()

LocalSpecifier := NewLocalEndpoint()
LocalSpecifier.WithAnySourceMulticastGroupIP(233.252.0.0)
LocalSpecifier.WithPort(5353)
LocalSpecifier.WithInterface("en0")

TransportProperties := ...
SecurityParameters  := ...

Preconnection := NewPreconnection(LocalSpecifier,
                                  RemoteSpecifier,
                                  TransportProperties,
                                  SecurityProperties)
Listener := Preconnection.Listen()
```

Join a Source-Specific Multicast group in receive-only mode, bound to a known port on a named local interface:

```
RemoteSpecifier := NewRemoteEndpoint()

LocalSpecifier := NewLocalEndpoint()

LocalSpecifier.WithSingleSourceMulticastGroupIP(233.252.0.0,
                                                198.51.100.10)
LocalSpecifier.WithPort(5353)
LocalSpecifier.WithInterface("en0")

TransportProperties := ...
SecurityParameters  := ...

Preconnection := NewPreconnection(LocalSpecifier,
                                  RemoteSpecifier,
                                  TransportProperties,
                                  SecurityProperties)
Listener := Preconnection.Listen()
```

Create a Source-Specific Multicast group as a sender:

```
RemoteSpecifier := NewRemoteEndpoint()
RemoteSpecifier.WithMulticastGroupIP(233.251.240.1)
RemoteSpecifier.WithPort(5353)
RemoteSpecifier.WithHopLimit(8)

LocalSpecifier := NewLocalEndpoint()
LocalSpecifier.WithIPAddress(192.0.2.22)
LocalSpecifier.WithInterface("en0")

TransportProperties := ...
SecurityParameters  := ...

Preconnection := NewPreconnection(LocalSpecifier,
                                  RemoteSpecifier,
                                  TransportProperties,
                                  SecurityProperties)
Connection := Preconnection.Initiate()
```

Join an any-source multicast group as both a sender and a receiver:

```
RemoteSpecifier := NewRemoteEndpoint()
RemoteSpecifier.WithMulticastGroupIP(233.252.0.0)
RemoteSpecifier.WithPort(5353)
RemoteSpecifier.WithHopLimit(8)

LocalSpecifier := NewLocalEndpoint()
LocalSpecifier.WithAnySourceMulticastGroupIP(233.252.0.0)
LocalSpecifier.WithIPAddress(192.0.2.22)
LocalSpecifier.WithPort(5353)
LocalSpecifier.WithInterface("en0")


TransportProperties := ...
SecurityParameters  := ...

Preconnection := NewPreconnection(LocalSpecifier,
                                  RemoteSpecifier,
                                  TransportProperties,
                                  SecurityProperties)
Connection, Listener := Preconnection.Rendezvous()
```

6.2. Specifying Transport Properties

A Preconnection object holds properties reflecting the application's requirements and preferences for the transport. These include Selection Properties for selecting Protocol Stacks and paths, as well as Connection Properties and Message Properties for configuration of the detailed operation of the selected Protocol Stacks on a per-Connection and Message level.

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

When additional information (such as Provisioning Domain (PvD) information [RFC7556]) is available about the networks over which an endpoint can operate, this can inform the selection between alternate network paths. Path information can include PMTU, set of supported DSCPs, expected usage, cost, etc. The usage of this information by the Transport Services System is generally independent of the specific mechanism/protocol used to receive the information (e.g. zero-conf, DHCP, or IPv6 RA).

Most Selection Properties are represented as Preferences, which can take one of five values:

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

Table 1: Selection Property Preference Levels

The implementation MUST ensure an outcome that is consistent with all application requirements expressed using Require and Prohibit. While preferences expressed using Prefer and Avoid influence protocol and path selection as well, outcomes can vary given the same Selection Properties, because the available protocols and paths can differ across systems and contexts. However, implementations are RECOMMENDED to seek to provide a consistent outcome to an application, when provided with the same set of Selection Properties.

Note that application preferences can conflict with each other. For example, if an application indicates a preference for a specific path by specifying an interface, but also a preference for a protocol, a situation might occur in which the preferred protocol is not available on the preferred path. In such cases, applications can expect properties that determine path selection to be prioritized over properties that determine protocol selection. The transport system SHOULD determine the preferred path first, regardless of protocol preferences. This ordering is chosen to provide consistency across implementations, based on the fact that it is more common for the use of a given network path to determine cost to the user (i.e., an interface type preference might be based on a user’s preference to avoid being charged more for a cellular data plan).

Selection and Connection Properties, as well as defaults for Message Properties, can be added to a Preconnection to configure the selection process and to further configure the eventually selected Protocol Stack(s). They are collected into a TransportProperties object to be passed into a Preconnection object:

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

Individual properties are then set on the TransportProperties object. Setting a Transport Property to a value overrides the previous value of this Transport Property.

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

To aid readability, implementations MAY provide additional convenience functions to simplify use of Selection Properties: see Appendix B.1 for examples. In addition, implementations MAY provide a mechanism to create TransportProperties objects that are preconfigured for common use cases, as outlined in Appendix B.2.

Transport Properties for an established Connection can be queried via the Connection object, as outlined in Section 8.

A Connection gets its Transport Properties either by being explicitly configured via a Preconnection, by configuration after establishment, or by inheriting them from an antecedent via cloning; see Section 7.4 for more.

Section 8.1 provides a list of Connection Properties, while Selection Properties are listed in the subsections below. Selection Properties are only considered during establishment, and can not be changed after a Connection is established. After a Connection is established, Selection Properties can only be read to check the properties used by the Connection. Upon reading, the Preference type of a Selection Property changes into Boolean, where true means that the selected Protocol Stack supports the feature or uses the path associated with the Selection Property, and false means that the Protocol Stack does not support the feature or use the path. Implementations of Transport Services systems could alternatively use the two Preference values Require and Prohibit to represent true and false, respectively. Other types of Selection Properties remain unchanged when they are made available for reading after a Connection is established.

An implementation of the Transport Services API needs to provide sensible defaults for Selection Properties. The default values for each property below represent a configuration that can be implemented over TCP. If these default values are used and TCP is not supported

by a Transport Services system, then an application using the default set of Properties might not succeed in establishing a Connection. Using the same default values for independent Transport Services systems can be beneficial when applications are ported between different implementations/platforms, even if this default could lead to a Connection failure when TCP is not available. If default values other than those suggested below are used, it is RECOMMENDED to clearly document any differences.

6.2.1. Reliable Data Transfer (Connection)

Name: reliability

Type: Preference

Default: Require

This property specifies whether the application needs to use a transport protocol that ensures that all data is received at the Remote Endpoint in order without loss or duplication. When reliable data transfer is enabled, this also entails being notified when a Connection is closed or aborted.

6.2.2. Preservation of Message Boundaries

Name: preserveMsgBoundaries

Type: Preference

Default: No Preference

This property specifies whether the application needs or prefers to use a transport protocol that preserves message boundaries.

6.2.3. Configure Per-Message Reliability

Name: perMsgReliability

Type: Preference

Default: No Preference

This property specifies whether an application considers it useful to specify different reliability requirements for individual Messages in a Connection.

6.2.4. Preservation of Data Ordering

Name: preserveOrder

Type: Preference

Default: Require

This property specifies whether the application wishes to use a transport protocol that can ensure that data is received by the application at the Remote Endpoint in the same order as it was sent.

6.2.5. Use 0-RTT Session Establishment with a Safely Replayable Message

Name: zeroRttMsg

Type: Preference

Default: No Preference

This property specifies whether an application would like to supply a Message to the transport protocol before connection establishment that will then be reliably transferred to the Remote Endpoint before or during connection establishment. This Message can potentially be received multiple times (i.e., multiple copies of the Message data could be passed to the Remote Endpoint). See also Section 9.1.3.4.

6.2.6. Multistream Connections in Group

Name: multistreaming

Type: Preference

Default: Prefer

This property specifies whether the application would prefer multiple Connections within a Connection Group to be provided by streams of a single underlying transport connection where possible.

6.2.7. Full Checksum Coverage on Sending

Name: fullChecksumSend

Type: Preference

Default: Require

This property specifies the application's need for protection against corruption for all data transmitted on this Connection. Disabling this property could enable the application to influence the sender checksum coverage after Connection establishment (see Section 9.1.3.6).

6.2.8. Full Checksum Coverage on Receiving

Name: fullChecksumRecv

Type: Preference

Default: Require

This property specifies the application's need for protection against corruption for all data received on this Connection. Disabling this property could enable the application to influence the required minimum receiver checksum coverage after Connection establishment (see Section 8.1.1).

6.2.9. Congestion control

Name: congestionControl

Type: Preference

Default: Require

This property specifies whether the application would like the Connection to be congestion controlled or not. Note that if a Connection is not congestion controlled, an application using such a Connection SHOULD itself perform congestion control in accordance with [RFC2914] or use a circuit breaker in accordance with [RFC8084], whichever is appropriate. Also note that reliability is usually combined with congestion control in protocol implementations, rendering "reliable but not congestion controlled" a request that is unlikely to succeed. If the Connection is congestion controlled, performing additional congestion control in the application can have negative performance implications.

6.2.10. Keep alive

Name: keepAlive

Type: Preference

Default: No Preference

This property specifies whether the application would like the Connection to send keep-alive packets or not. Note that if a Connection determines that keep-alive packets are being sent, the application SHOULD itself avoid generating additional keep-alive messages. Note that when supported, the system will use the default period for generation of the keep alive-packets. (See also Section 8.1.4).

6.2.11. Interface Instance or Type

Name: interface

Type: Set of (Preference, Enumeration)

Default: Empty (not setting a preference for any interface)

This property allows the application to select any specific network interfaces or categories of interfaces it wants to Require, Prohibit, Prefer, or Avoid. Note that marking a specific interface as Require strictly limits path selection to that single interface, and often leads to less flexible and resilient connection establishment.

In contrast to other Selection Properties, this property is a set of tuples of (Enumerated) interface identifier and preference. It 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 could be Wi-Fi and Cellular interface types available; whereas on a desktop computer, Wi-Fi and Wired Ethernet interface types might be available. An implementation should provide all types that are supported on the local system, to allow applications to be written generically. For example, if a single implementation is used on both mobile devices and desktop devices, it ought to define the Cellular interface type for both systems, since an application might wish to always prohibit cellular.

The set of interface types is expected to change over time as new access technologies become available. The taxonomy of interface types on a given Transport Services system is implementation-specific.

Interface types SHOULD NOT be treated as a proxy for properties of interfaces such as metered or unmetered network access. If an application needs to prohibit metered interfaces, this should be specified via Provisioning Domain attributes (see Section 6.2.12) or another specific property.

Note that this property is not used to specify an interface scope zone for a particular Endpoint. Section 6.1.2 provides details about how to qualify endpoint candidates on a per-interface basis.

6.2.12. Provisioning Domain Instance or Type

Name: pvd

Type: Set of (Preference, Enumeration)

Default: Empty (not setting a preference for any PvD)

Similar to interface (see Section 6.2.11), this property allows the application to control path selection by selecting which specific Provisioning Domain (PvD) or categories of PVDs it wants to Require, Prohibit, Prefer, or Avoid. Provisioning Domains define consistent sets of network properties that might be more specific than network interfaces [RFC7556].

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

The identification of a specific PvD is implementation- and system-specific. [RFC8801] defines how to use an FQDN to identify a PvD when advertised by a network, but systems might also use other locally-relevant identifiers such as string names or Integers to identify PvDs. As with requiring specific interfaces, requiring a specific PvD strictly limits the path selection.

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

6.2.13. Use Temporary Local Address

Name: useTemporaryLocalAddress

Type: Preference

Default: Avoid for Listeners and Rendezvous Connections. Prefer for other Connections.

This property allows the application to express a preference for the use of temporary local addresses, sometimes called "privacy" addresses [RFC8981]. Temporary addresses are generally used to prevent linking connections over time when a stable address, sometimes called "permanent" address, is not needed. There are some caveats to note when specifying this property. First, if an application Requires the use of temporary addresses, the resulting Connection cannot use IPv4, because temporary addresses do not exist in IPv4. Second, temporary local addresses might involve trading off privacy for performance. For instance, temporary addresses (e.g., [RFC8981]) can interfere with resumption mechanisms that some protocols rely on to reduce initial latency.

6.2.14. Multipath Transport

Name: multipath

Type: Enumeration

Default: Disabled for Connections created through initiate and rendezvous, Passive for Listeners

This property specifies whether and how applications want to take advantage of transferring data across multiple paths between the same end hosts. Using multiple paths allows Connections to migrate between interfaces or aggregate bandwidth as availability and performance properties change. Possible values are:

Disabled: The Connection will not use multiple paths once established, even if the chosen transport supports using multiple paths.

Active: The Connection will negotiate the use of multiple paths if the chosen transport supports this.

Passive: The Connection will support the use of multiple paths if the Remote Endpoint requests it.

The policy for using multiple paths is specified using the separate multipathPolicy property, see Section 8.1.7 below. To enable the peer endpoint to initiate additional paths towards a local address other than the one initially used, it is necessary to set the advertisesAltaddr property (see Section 6.2.15 below).

Setting this property to Active can have privacy implications: It enables the transport to establish connectivity using alternate paths that might result in users being linkable across the multiple paths, even if the advertisesAltaddr property (see Section 6.2.15 below) is set to false.

Note that Multipath Transport has no corresponding Selection Property of type Preference. Enumeration values other than Disabled are interpreted as a preference for choosing protocols that can make use of multiple paths. The Disabled value implies a requirement not to use multiple paths in parallel but does not prevent choosing a protocol that is capable of using multiple paths, e.g., it does not prevent choosing TCP, but prevents sending the MP_CAPABLE option in the TCP handshake.

6.2.15. Advertisement of Alternative Addresses

Name: advertisesAltaddr

Type: Boolean

Default: false

This property specifies whether alternative addresses, e.g., of other interfaces, ought to be advertised to the peer endpoint by the Protocol Stack. Advertising these addresses enables the peer endpoint to establish additional connectivity, e.g., for Connection migration or using multiple paths.

Note that this can have privacy implications because it might result in users being linkable across the multiple paths. Also, note that setting this to false does not prevent the local Transport Services system from _establishing_ connectivity using alternate paths (see Section 6.2.14 above); it only prevents _proactive advertisement_ of addresses.

6.2.16. Direction of communication

Name: direction

Type: Enumeration

Default: Bidirectional

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

Bidirectional: The Connection must support sending and receiving data

Unidirectional send: The Connection must support sending data, and the application cannot use the Connection to receive any data

Unidirectional receive: The Connection must support receiving data, and the application cannot use the Connection to send any data

Since unidirectional communication can be supported by transports offering bidirectional communication, specifying unidirectional communication might cause a transport stack that supports bidirectional communication to be selected.

6.2.17. Notification of ICMP soft error message arrival

Name: softErrorNotify

Type: Preference

Default: No Preference

This property specifies whether an application considers it useful to be informed when an ICMP error message arrives that does not force termination of a connection. When set to true, received ICMP errors are available as SoftError events, see Section 8.3.1. Note that even if a protocol supporting this property is selected, not all ICMP errors will necessarily be delivered, so applications cannot rely upon receiving them [RFC8085].

6.2.18. Initiating side is not the first to write

Name: activeReadBeforeSend

Type: Preference

Default: No Preference

The most common client-server communication pattern involves the client actively opening a Connection, then sending data to the server. The server listens (passive open), reads, and then answers. This property specifies whether an application wants to diverge from this pattern -- either by actively opening with Initiate, immediately

followed by reading, or passively opening with Listen, immediately followed by writing. This property is ignored when establishing connections using Rendezvous. Requiring this property limits the choice of mappings to underlying protocols, which can reduce efficiency. For example, it prevents the Transport Services system from mapping Connections to SCTP streams, where the first transmitted data takes the role of an active open signal.

6.3. Specifying Security Parameters and Callbacks

Most security parameters, e.g., TLS ciphersuites, local identity and private key, etc., can be configured statically. Others are dynamically configured during Connection establishment. Security parameters and callbacks are partitioned based on their place in the lifetime of Connection establishment. Similar to Transport Properties, both parameters and callbacks are inherited during cloning (see Section 7.4).

This document specifies an abstract API, which could appear to conflict with the need for security parameters to be unambiguous. The Transport Services System SHOULD provide reasonable, secure defaults for each enumerated security parameter, such that users of the system only need to specify parameters required to establish a secure connection (e.g., serverCertificate, clientCertificate). Specifying security parameters from enumerated values (e.g., specific ciphersuites) might constrain which transport protocols can be selected during Connection establishment.

Security configuration parameters are specified in the pre-establishment phase and are created as follows:

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

Specific parameters are added using a call to Set() on the SecurityParameters.

As with the rest of the Transport Services API, the exact names of parameters and/or values of enumerations (e.g., ciphersuites) used in the security parameters are system- and implementation-specific, and ought to be chosen to follow the principle of least surprise for users of the platform / language environment in question.

For security parameters that are enumerations of known values, such as TLS ciphersuites, implementations are responsible for exposing the set of values they support. For security parameters that are not simple value types, such as certificates and keys, implementations are responsible for exposing types appropriate for the platform / language environment.

Applications SHOULD use common safe defaults for values such as TLS ciphersuites whenever possible. However, as discussed in [RFC8922], many transport security protocols require specific security parameters and constraints from the client at the time of configuration and actively during a handshake.

The set of security parameters defined here is not exhaustive, but illustrative. Implementations SHOULD expose an equivalent to the parameters listed below to allow for sufficient configuration of security parameters, but the details are expected to vary based on platform and implementation constraints. Applications MUST be able to constrain the security protocols and versions that the Transport Services System will use.

Representation of security parameters in implementations ought to parallel that chosen for Transport Property names as suggested in Section 5.

Connections that use Transport Services SHOULD use security in general. However, for compatibility with endpoints that do not support transport security protocols (such as a TCP endpoint that does not support TLS), applications can initialize their security parameters to indicate that security can be disabled, or can be opportunistic. If security is disabled, the Transport Services system will not attempt to add transport security automatically. If security is opportunistic, it will allow Connections without transport security, but will still attempt to use unauthenticated security if available.

SecurityParameters := NewDisabledSecurityParameters()

SecurityParameters := NewOpportunisticSecurityParameters()

6.3.1. Allowed security protocols

Name: allowedSecurityProtocols

Type: Implementation-specific enumeration of security protocol names and/or versions.

Default: Implementation-specific best available security protocols

This property allows applications to restrict which security protocols and security protocol versions can be used in the protocol stack. Applications MUST be able to constrain the security protocols used by this or an equivalent mechanism, in order to prevent the use of security protocols with unknown or weak security properties.

```
SecurityParameters.Set(allowedSecurityProtocols, [ tls_1_2, tls_1_3 ])
```

6.3.2. Certificate bundles

Names: serverCertificate, clientCertificate

Type: Array of certificate objects

Default: Empty array

One or more certificate bundles identifying the Local Endpoint, whether as a server certificate or a client certificate. Multiple bundles may be provided to allow selection among different protocol stacks that may require differently formatted bundles. The form and format of the certificate bundle is implementation-specific. Note that if the private keys associated with a bundle are not available, e.g., since they are stored in hardware security modules (HSMs), handshake callbacks are necessary. See below for details.

```
SecurityParameters.Set(serverCertificate, myCertificateBundle[])
SecurityParameters.Set(clientCertificate, myCertificateBundle[])
```

6.3.3. Pinned server certificate

Name: pinnedServerCertificate

Type: Array of certificate chain objects

Default: Empty array

Zero or more certificate chains to use as pinned server certificates, such that connecting will fail if the presented server certificate does not match one of the supplied pinned certificates. The form and format of the certificate chain is implementation-specific.

```
SecurityParameters.Set(pinnedServerCertificate, yourCertificateChain[])
```

6.3.4. Application-layer protocol negotiation

Name: alpn

Type: Array of Strings

Default: Automatic selection

Application-layer protocol negotiation (ALPN) values: used to indicate which application-layer protocols are negotiated by the security protocol layer. See [ALPN] for definition of the ALPN

field. Note that the Transport Services System can provide ALPN values automatically, based on the protocols being used, if not explicitly specified by the application.

SecurityParameters.Set(alpn, ["h2"])

6.3.5. Groups, ciphersuites, and signature algorithms

Names: supportedGroup, ciphersuite, signatureAlgorithm

Types: Arrays of implementation-specific enumerations

Default: Automatic selection

These are used to restrict what cryptographic parameters are used by underlying transport security protocols. When not specified, these algorithms should use known and safe defaults for the system.

SecurityParameters.Set(supportedGroup, secp256r1)
SecurityParameters.Set(ciphersuite, TLS_AES_128_GCM_SHA256)
SecurityParameters.Set(signatureAlgorithm, ecdsa_secp256r1_sha256)

6.3.6. Session cache options

Names: maxCachedSessions, cachedSessionLifetimeSeconds

Type: Integer

Default: Automatic selection

These values are used to tune session cache capacity and lifetime, and can be extended to include other policies.

SecurityParameters.Set(maxCachedSessions, 16)
SecurityParameters.Set(cachedSessionLifetimeSeconds, 3600)

6.3.7. Pre-shared key

Name: preSharedKey

Type: Key and identity (platform-specific)

Default: None

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

shared key will tend to select a single security protocol, and therefore directly select a single underlying protocol stack. A Transport Services API could express None in an environment-typical way, e.g., as a Union type or special value.

```
SecurityParameters.Set(preSharedKey, key, myIdentity)
```

6.3.8. Connection Establishment Callbacks

Security decisions, especially pertaining to trust, are not static. Once configured, parameters can also be supplied during Connection establishment. These are best handled as client-provided callbacks. Callbacks block the progress of the Connection establishment, which distinguishes them from other events in the transport system. How callbacks and events are implemented is specific to each implementation. Security handshake callbacks that could be invoked during Connection establishment include:

* Trust verification callback: Invoked when a Remote Endpoint's trust must be verified before the handshake protocol can continue. For example, the application could verify an X.509 certificate as described in [RFC5280].

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

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

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

7. Establishing Connections

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

7.1. Active Open: Initiate

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

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

The timeout parameter specifies how long to wait before aborting Active open. Before calling Initiate, the caller must have populated a Preconnection object with a Remote Endpoint object to identify the endpoint, optionally a Local Endpoint object (if not specified, the system will attempt to determine a suitable Local Endpoint), as well as all properties necessary for candidate selection.

The Initiate action returns a Connection object. Once Initiate has been called, any changes to the Preconnection MUST NOT have any effect on the Connection. However, the Preconnection can be reused, e.g., to Initiate another Connection.

Once Initiate is called, the candidate Protocol Stack(s) can cause one or more candidate transport-layer connections to be created to the specified Remote Endpoint. The caller could immediately begin sending Messages on the Connection (see Section 9.2) after calling Initiate; note that any data marked as "safely replayable" that is sent while the Connection is being established could be sent multiple times or on multiple candidates.

The following events can be sent by the Connection after Initiate is called:

```
Connection -> Ready<>
```

The Ready event occurs after Initiate has established a transport-layer connection on at least one usable candidate Protocol Stack over at least one candidate Path. No Receive events (see Section 9.3) will occur before the Ready event for Connections established using Initiate.

```
Connection -> EstablishmentError<reason?>
```

An EstablishmentError occurs either when the set of transport properties and security parameters cannot be fulfilled on a Connection for initiation (e.g., the set of available Paths and/or Protocol Stacks meeting the constraints is empty) or reconciled with the Local and/or Remote Endpoints; when a remote Endpoint Identifier

cannot be resolved; or when no transport-layer connection can be established to the Remote Endpoint (e.g., because the Remote Endpoint is not accepting connections, the application is prohibited from opening a Connection by the operating system, or the establishment attempt has timed out for any other reason).

Connection establishment and transmission of the first Message can be combined in a single action (Section 9.2.5).

7.2. Passive Open: Listen

Passive open is the action of waiting for Connections from Remote Endpoints, commonly used by servers in client-server interactions. Passive open is supported by the Transport Services API through the Listen action and returns a Listener object:

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

Before calling Listen, the caller must have initialized the Preconnection during the pre-establishment phase with a Local Endpoint object, as well as all properties necessary for Protocol Stack selection. A Remote Endpoint can optionally be specified, to constrain what Connections are accepted.

The Listen action returns a Listener object. Once Listen has been called, any changes to the Preconnection MUST NOT have any effect on the Listener. The Preconnection can be disposed of or reused, e.g., to create another Listener.

```
Listener.Stop()
```

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

```
Listener -> ConnectionReceived<Connection>
```

The ConnectionReceived event occurs when a Remote Endpoint has established or cloned (e.g., by creating a new stream in a multi-stream transport; see Section 7.4) 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 or streams of a multi-streaming transport), causing a new Connection to be created. The resulting Connection is contained within the ConnectionReceived event, and is ready to use as soon as it is passed to the application via the event.

```
Listener.SetNewConnectionLimit(value)
```

If the caller wants to rate-limit the number of inbound Connections that will be delivered, it can set a cap using SetNewConnectionLimit. This mechanism allows a server to protect itself from being drained of resources. Each time a new Connection is delivered by the ConnectionReceived event, the value is automatically decremented. Once the value reaches zero, no further Connections will be delivered until the caller sets the limit to a higher value. By default, this value is Infinite. The caller is also able to reset the value to Infinite at any point.

Listener -> EstablishmentError<reason?>

An EstablishmentError occurs either when the Properties and security parameters of the Preconnection cannot be fulfilled for listening or cannot be reconciled with the Local Endpoint (and/or Remote Endpoint, if specified), when the Local Endpoint (or Remote Endpoint, if specified) cannot be resolved, or when the application is prohibited from listening by policy.

Listener -> Stopped<>

A Stopped event occurs after the Listener has stopped listening.

7.3. Peer-to-Peer Establishment: Rendezvous

Simultaneous peer-to-peer Connection establishment is supported by the Rendezvous action:

Preconnection.Rendezvous()

A Preconnection object used in a Rendezvous MUST have both the Local Endpoint candidates and the Remote Endpoint candidates specified, along with the Transport Properties and security parameters needed for Protocol Stack selection, before the Rendezvous action is initiated.

The Rendezvous action listens on the Local Endpoint candidates for an incoming Connection from the Remote Endpoint candidates, while also simultaneously trying to establish a Connection from the Local Endpoint candidates to the Remote Endpoint candidates.

If there are multiple Local Endpoints or Remote Endpoints configured, then initiating a Rendezvous action will cause the Transport Services Implementation to systematically probe the reachability of those endpoint candidates following an approach such as that used in Interactive Connectivity Establishment (ICE) [RFC8445].

If the endpoints are suspected to be behind a NAT, and the Local Endpoint supports a method of discovering NAT bindings, such as Session Traversal Utilities for NAT (STUN) [RFC8489] or Traversal Using Relays around NAT (TURN) [RFC8656], then the Resolve action on the Preconnection can be used to discover such bindings:

```
[]LocalEndpoint, []RemoteEndpoint := Preconnection.Resolve()
```

The Resolve call returns lists of Local Endpoints and Remote Endpoints that represent the concrete addresses, local and server reflexive, on which a Rendezvous for the Preconnection will listen for incoming Connections, and to which it will attempt to establish Connections.

Note that the set of Local Endpoints returned by Resolve might or might not contain information about all possible local interfaces depending on how the Preconnection is configured. The set of available local interfaces can also change over time so care needs to be taken when using stored interface names.

An application that uses Rendezvous to establish a peer-to-peer Connection in the presence of NATs will configure the Preconnection object with at least one Local Endpoint that supports NAT binding discovery. It will then Resolve the Preconnection, and pass the resulting list of Local Endpoint candidates to the peer via a signalling protocol, for example as part of an ICE [RFC8445] exchange within SIP [RFC3261] or WebRTC [RFC7478]. The peer will then, via the same signalling channel, return the Remote Endpoint candidates. The set of Remote Endpoint candidates are then configured onto the Preconnection:

```
Preconnection.AddRemote([]RemoteEndpoint)
```

The Rendezvous action is initiated, and causes the Transport Services Implementation to begin connectivity checks, once the application has added both the Local Endpoint candidates and the Remote Endpoint candidates retrieved from the peer via the signalling channel to the Preconnection.

If successful, the Rendezvous action returns a Connection object via a RendezvousDone<> event:

```
Preconnection -> RendezvousDone<Connection>
```

The RendezvousDone<> event occurs when a Connection is established with the Remote Endpoint. For Connection-oriented transports, this occurs when the transport-layer connection is established; for Connectionless transports, it occurs when the first Message is

received from the Remote Endpoint. The resulting Connection is contained within the RendezvousDone<> event, and is ready to use as soon as it is passed to the application via the event. Changes made to a Preconnection after Rendezvous has been called MUST NOT have any effect on existing Connections.

An EstablishmentError occurs either when the Properties and Security Parameters of the Preconnection cannot be fulfilled for rendezvous or cannot be reconciled with the Local and/or Remote Endpoints, when the Local Endpoint or Remote Endpoint cannot be resolved, when no transport-layer connection can be established to the Remote Endpoint, or when the application is prohibited from rendezvous by policy:

Preconnection -> EstablishmentError<reason?>

7.4. Connection Groups

Connection Groups can be created using the Clone action:

Connection := Connection.Clone(framer?, connectionProperties?)

Calling Clone on a Connection yields a Connection Group containing two Connections: the parent Connection on which Clone was called, and a resulting cloned Connection. The new Connection is actively opened, and it will locally send a Ready event or an EstablishmentError event. Calling Clone on any of these Connections adds another Connection to the Connection Group. Connections in a Connection Group share all Connection Properties except connPriority (see Section 8.1.2), and these Connection Properties are entangled: Changing one of the Connection Properties on one Connection in the Connection Group automatically changes the Connection Property for all others. For example, changing connTimeout (see Section 8.1.3) on one Connection in a Connection Group will automatically make the same change to this Connection Property for all other Connections in the Connection Group. Like all other Properties, connPriority is copied to the new Connection when calling Clone, but in this case, a later change to the connPriority on one Connection does not change it on the other Connections in the same Connection Group.

The optional connectionProperties parameter allows passing Transport Properties that control the behavior of the underlying stream or connection to be created, e.g., Protocol-specific Properties to request specific stream IDs for SCTP or QUIC.

Message Properties set on a Connection also apply only to that Connection.

A new Connection created by Clone can have a Message Framer assigned via the optional framer parameter of the Clone action. If this parameter is not supplied, the stack of Message Framers associated with a Connection is copied to the cloned Connection when calling Clone. Then, a cloned Connection has the same stack of Message Framers as the Connection from which they are cloned, but these Framers can internally maintain per-Connection state.

It is also possible to check which Connections belong to the same Connection Group. Calling GroupedConnections on a specific Connection returns a set of all Connections in the same group.

```
[]Connection := Connection.GroupedConnections()
```

Connections will belong to the same group if the application previously called Clone. Passive Connections can also be added to the same group -- e.g., when a Listener receives a new Connection that is just a new stream of an already active multi-streaming protocol instance.

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

Note that calling Clone can result in on-the-wire signaling, e.g., to open a new transport connection, depending on the underlying Protocol Stack. When Clone leads to the opening of multiple such connections, the Transport Services system will ensure consistency of Connection Properties by uniformly applying them to all underlying connections in a group. Even in such a case, there are possibilities for a Transport Services system to implement prioritization within a Connection Group [TCP-COUPLING] [RFC8699].

Attempts to clone a Connection can result in a CloneError:

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

A CloneError can also occur later, after Clone was successfully called. In this case, it informs the application that the Connection that sends the CloneError is no longer a part of any Connection Group. For example, this can occur when the Transport Services system is unable to implement entanglement (a Connection Property was changed on a different Connection in the Connection Group, but this change could not be successfully applied to the Connection that sends the CloneError).

The connPriority Connection Property operates on Connections in a Connection Group using the same approach as in Section 9.1.3.2: when allocating available network capacity among Connections in a Connection Group, sends on Connections with numerically lower Priority values will be prioritized over sends on Connections that have numerically higher Priority values. Capacity will be shared among these Connections according to the connScheduler property (Section 8.1.5). See Section 9.2.6 for more.

7.5. Adding and Removing Endpoints on a Connection

Transport protocols that are explicitly multipath aware are expected to automatically manage the set of Remote Endpoints that they are communicating with, and the paths to those endpoints. A PathChange<> event, described in Section 8.3.2, will be generated when the path changes.

In some cases, however, it is necessary to explicitly indicate to a Connection that a new Remote Endpoint has become available for use, or to indicate that a Remote Endpoint is no longer available. This is most common in the case of peer to peer connections using Trickle ICE [RFC8838].

The AddRemote action can be used to add one or more new Remote Endpoints to a Connection:

```
Connection.AddRemote([]RemoteEndpoint)
```

Endpoints that are already known to the Connection are ignored. A call to AddRemote makes the new Remote Endpoints available to the Connection, but whether the Connection makes use of those Endpoints will depend on the underlying transport protocol.

Similarly, the RemoveRemote action can be used to tell a Connection to stop using one or more Remote Endpoints:

```
Connection.RemoveRemote([]RemoteEndpoint)
```

Removing all known Remote Endpoints can have the effect of aborting the connection. The effect of removing the active Remote Endpoint(s) depends on the underlying transport: multipath aware transports might be able to switch to a new path if other reachable Remote Endpoints exist, or the connection might abort.

Similarly, the AddLocal and RemoveLocal actions can be used to add and remove Local Endpoints to/from a Connection.

8. Managing Connections

During pre-establishment and after establishment, (Pre-)Connections can be configured and queried using Connection Properties, and asynchronous information could be available about the state of the Connection via SoftError events.

Connection Properties represent the configuration and state of the selected Protocol Stack(s) backing a Connection. These Connection Properties can be generic, applying regardless of transport protocol, or specific, applicable to a single implementation of a single transport Protocol Stack. Generic Connection Properties are defined in Section 8.1 below.

Protocol-specific Properties are defined in a transport- and implementation-specific way to permit more specialized protocol features to be used. Too much reliance by an application on Protocol-specific Properties can significantly reduce the flexibility of a transport services system to make appropriate selection and configuration choices. Therefore, it is RECOMMENDED that Generic Connection Properties are used for properties common across different protocols and that Protocol-specific Connection Properties are only used where specific protocols or properties are necessary.

The application can set and query Connection Properties on a per-Connection basis. Connection Properties that are not read-only can be set during pre-establishment (see Section 6.2), as well as on Connections directly using the SetProperty action:

```
ErrorCode := Connection.SetProperty(property, value)
```

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 application MUST be informed about this error via the ErrorCode Object. Such errors MUST NOT cause the Connection to be terminated. Note that changing one of the Connection Properties on one Connection in a Connection Group will also change it for all other Connections of that group; see further Section 7.4.

At any point, the application can query Connection Properties.

```
ConnectionProperties := Connection.GetProperties()
value := ConnectionProperties.Get(property)
if ConnectionProperties.Has(boolean_or_preference_property) then ...
```

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

* The Connection state, which can be one of the following: Establishing, Established, Closing, or Closed (see Section 8.1.11.1).

* Whether the Connection can be used to send data (see Section 8.1.11.2). A Connection can not be used for sending if the Connection was created with the Selection Property direction set to unidirectional receive or if a Message marked as Final was sent over this Connection. See also Section 9.1.3.5.

* Whether the Connection can be used to receive data (see Section 8.1.11.3). A Connection cannot be used for receiving if the Connection was created with the Selection Property direction set to unidirectional send or if a Message marked as Final was received. See Section 9.3.3.3. The latter is only supported by certain transport protocols, e.g., by TCP as half-closed connection.

* For Connections that are Established, Closing, or Closed: Connection Properties (Section 8.1) of the actual protocols that were selected and instantiated, and Selection Properties that the application specified on the Preconnection. Selection Properties of type Preference will be exposed as boolean values indicating whether or not the property applies to the selected transport. Note that the instantiated Protocol Stack might not match all Protocol Selection Properties that the application specified on the Preconnection.

* For Connections that are Established: Transport Services system implementations ought to provide information concerning the path(s) used by the Protocol Stack. This can be derived from local PVD information, measurements by the Protocol Stack, or other sources. For example, a Transport System that is configured to receive and process PVD information [RFC7556] could also provide network configuration information for the chosen path(s).

8.1. Generic Connection Properties

Generic Connection Properties are defined independent of the chosen Protocol Stack and therefore available on all Connections.

Many Connection Properties have a corresponding Selection Property that enables applications to express their preference for protocols providing a supporting transport feature.

8.1.1. Required Minimum Corruption Protection Coverage for Receiving

Name: recvChecksumLen

Type: Integer (non-negative) or Full Coverage

Default: Full Coverage

If this property is an Integer, it specifies the minimum number of bytes in a received Message that need to be covered by a checksum. A receiving endpoint will not forward Messages that have less coverage to the application. The application is responsible for handling any corruption within the non-protected part of the Message [RFC8085]. A special value of 0 means that a received packet might also have a zero checksum field, and the enumerated value Full Coverage means that the entire Message needs to be protected by a checksum. An implementation is supposed to express Full Coverage in an environment-typical way, e.g., as a Union type or special value.

8.1.2. Connection Priority

Name: connPriority

Type: Integer (non-negative)

Default: 100

This property is a non-negative Integer representing the priority of this Connection relative to other Connections in the same Connection Group. A numerically lower value reflects a higher priority. It has no effect on Connections not part of a Connection Group. As noted in Section 7.4, this property is not entangled when Connections are cloned, i.e., changing the Priority on one Connection in a Connection Group does not change it on the other Connections in the same Connection Group. No guarantees of a specific behavior regarding Connection Priority are given; a Transport Services system could ignore this property. See Section 9.2.6 for more details.

8.1.3. Timeout for Aborting Connection

Name: connTimeout

Type: Numeric (positive) or Disabled

Default: Disabled

If this property is Numeric, it specifies how long to wait before deciding that an active Connection has failed when trying to reliably deliver data to the Remote Endpoint. Adjusting this property will only take effect when the underlying stack supports reliability. If this property has the enumerated value Disabled, it means that no timeout is scheduled. A Transport Services API could express Disabled in an environment-typical way, e.g., as a Union type or special value.

8.1.4. Timeout for keep alive packets

Name: keepAliveTimeout

Type: Numeric (positive) or Disabled

Default: Disabled

A Transport Services API can request a protocol that supports sending keep alive packets (Section 6.2.10). If this property is Numeric, it specifies the maximum length of time an idle Connection (one for which no transport packets have been sent) ought to wait before the Local Endpoint sends a keep-alive packet to the Remote Endpoint. Adjusting this property will only take effect when the underlying stack supports sending keep-alive packets. Guidance on setting this value for connection-less transports is provided in [RFC8085]. A value greater than the Connection timeout (Section 8.1.3) or the enumerated value Disabled will disable the sending of keep-alive packets. A Transport Services API could express Disabled in an environment-typical way, e.g., as a Union type or special value.

8.1.5. Connection Group Transmission Scheduler

Name: connScheduler

Type: Enumeration

Default: Weighted Fair Queueing (see Section 3.6 of [RFC8260])

This property specifies which scheduler is used among Connections within a Connection Group to apportion the available capacity according to Connection priorities (see Section 7.4 and Section 8.1.2). A set of schedulers is described in [RFC8260].

8.1.6. Capacity Profile

Name: connCapacityProfile

Type: Enumeration

Default: Default Profile (Best Effort)

This property specifies the desired network treatment for traffic sent by the application and the tradeoffs the application is prepared to make in path and protocol selection to receive that desired treatment. When the capacity profile is set to a value other than Default, the Transport Services system SHOULD select paths and configure protocols to optimize the tradeoff between delay, delay variation, and efficient use of the available capacity based on the capacity profile specified. How this is realized is implementation-specific. The capacity profile MAY also be used to set markings on the wire for Protocol Stacks supporting this. Recommendations for use with DSCP are provided below for each profile; note that when a Connection is multiplexed, the guidelines in Section 6 of [RFC7657] apply.

The following values are valid for the capacity profile:

Default: The application provides no information about its expected capacity profile. Transport Services systems that map the requested capacity profile onto per-connection DSCP signaling SHOULD assign the DSCP Default Forwarding [RFC2474] Per Hop Behaviour (PHB).

Scavenger: The application is not interactive. It expects to send and/or receive data without any urgency. This can, for example, be used to select Protocol Stacks with scavenger transmission control and/or to assign the traffic to a lower-effort service. Transport Services systems that map the requested capacity profile onto per-connection DSCP signaling SHOULD assign the DSCP Less than Best Effort [RFC8622] PHB.

Low Latency/Interactive: The application is interactive, and prefers loss to latency. Response time SHOULD be optimized at the expense of delay variation and efficient use of the available capacity when sending on this Connection. This can be used by the system to disable the coalescing of multiple small Messages into larger packets (Nagle's algorithm); to prefer immediate acknowledgment from the peer endpoint when supported by the underlying transport; and so on. Transport Services systems that map the requested capacity profile onto per-connection DSCP signaling without multiplexing SHOULD assign a DSCP Assured Forwarding (AF41,AF42,AF43,AF44) [RFC2597] PHB. Inelastic traffic that is expected to conform to the configured network service rate could be mapped to the DSCP Expedited Forwarding [RFC3246] or [RFC5865] PHBs.

Low Latency/Non-Interactive: The application prefers loss to

latency, but is not interactive. Response time SHOULD be optimized at the expense of delay variation and efficient use of the available capacity when sending on this Connection. Transport system implementations that map the requested capacity profile onto per-connection DSCP signaling without multiplexing SHOULD assign a DSCP Assured Forwarding (AF21,AF22,AF23,AF24) [RFC2597] PHB.

Constant-Rate Streaming: The application expects to send/receive data at a constant rate after Connection establishment. Delay and delay variation SHOULD be minimized at the expense of efficient use of the available capacity. This implies that the Connection might fail if the Path is unable to maintain the desired rate. A transport can interpret this capacity profile as preferring a circuit breaker [RFC8084] to a rate-adaptive congestion controller. Transport system implementations that map the requested capacity profile onto per-connection DSCP signaling without multiplexing SHOULD assign a DSCP Assured Forwarding (AF31,AF32,AF33,AF34) [RFC2597] PHB.

Capacity-Seeking: The application expects to send/receive data at the maximum rate allowed by its congestion controller over a relatively long period of time. Transport Services systems that map the requested capacity profile onto per-connection DSCP signaling without multiplexing SHOULD assign a DSCP Assured Forwarding (AF11,AF12,AF13,AF14) [RFC2597] PHB per Section 4.8 of [RFC4594].

The capacity profile for a selected Protocol Stack may be modified on a per-Message basis using the Transmission Profile Message Property; see Section 9.1.3.8.

8.1.7. Policy for using Multipath Transports

Name: multipathPolicy

Type: Enumeration

Default: Handover

This property specifies the local policy for transferring data across multiple paths between the same end hosts if Multipath Transport is not set to Disabled (see Section 6.2.14). Possible values are:

Handover: The Connection ought only to attempt to migrate between different paths when the original path is lost or becomes unusable. The thresholds used to declare a path unusable are implementation specific.

Interactive: The Connection ought only to attempt to minimize the latency for interactive traffic patterns by transmitting data across multiple paths when this is beneficial. The goal of minimizing the latency will be balanced against the cost of each of these paths. Depending on the cost of the lower-latency path, the scheduling might choose to use a higher-latency path. Traffic can be scheduled such that data may be transmitted on multiple paths in parallel to achieve a lower latency. The specific scheduling algorithm is implementation-specific.

Aggregate: The Connection ought to attempt to use multiple paths in parallel to maximize available capacity and possibly overcome the capacity limitations of the individual paths. The actual strategy is implementation specific.

Note that this is a local choice the Remote Endpoint can choose a different policy.

8.1.8. Bounds on Send or Receive Rate

Name: minSendRate / minRecvRate / maxSendRate / maxRecvRate

Type: Numeric (positive) or Unlimited / Numeric (positive) or Unlimited / Numeric (positive) or Unlimited / Numeric (positive) or Unlimited

Default: Unlimited / Unlimited / Unlimited / Unlimited

Numeric values of these properties specify an upper-bound rate that a transfer is not expected to exceed (even if flow control and congestion control allow higher rates), and/or a lower-bound application-layer rate below which the application does not deem it will be useful. These rate values are measured at the application layer, i.e. do not consider the header overhead from protocols used by the Transport Services system. The values are specified in bits per second, and assumed to be measured over one-second time intervals. E.g., specifying a maxSendRate of X bits per second means that, from the moment at which the property value is chosen, not more than X bits will be send in any following second. The enumerated value Unlimited indicates that no bound is specified. A Transport Services API could express Unlimited in an environment-typical way, e.g., as a Union type or special value.

8.1.9. Group Connection Limit

Name: groupConnLimit

Type: Numeric (positive) or Unlimited

Default: Unlimited

If this property is Numeric, it controls the number of Connections that can be accepted from a peer as new members of the Connection's group. Similar to SetNewConnectionLimit, this limits the number of ConnectionReceived events that will occur, but constrained to the group of the Connection associated with this property. For a multi-streaming transport, this limits the number of allowed streams. A Transport Services API could express Unlimited in an environment-typical way, e.g., as a Union type or special value.

8.1.10. Isolate Session

Name: isolateSession

Type: Boolean

Default: false

When set to true, this property will initiate new Connections using as little cached information (such as session tickets or cookies) as possible from previous Connections that are not in the same Connection Group. Any state generated by this Connection will only be shared with Connections in the same Connection Group. Cloned Connections will use saved state from within the Connection Group. This is used for separating Connection Contexts as specified in Section 4.2.3 of [I-D.ietf-taps-arch].

Note that this does not guarantee no leakage of information, as implementations might not be able to fully isolate all caches (e.g. RTT estimates). Note that this property could degrade Connection performance.

8.1.11. Read-only Connection Properties

The following generic Connection Properties are read-only, i.e. they cannot be changed by an application.

8.1.11.1. Connection state

Name: connState

Type: Enumeration

This property informs about the current state of the Connection. Possible values are: Establishing, Established, Closing or Closed; for more details on Connection state, see Section 11.

8.1.11.2. Can Send Data

Name: canSend

Type: Boolean

This property can be queried to learn whether the Connection can be used to send data.

8.1.11.3. Can Receive Data

Name: canReceive

Type: Boolean

This property can be queried to learn whether the Connection can be used to receive data.

8.1.11.4. Maximum Message Size Before Fragmentation or Segmentation

Name: singularTransmissionMsgMaxLen

Type: Integer (non-negative) or Not applicable

This property, if applicable, represents the maximum Message size that can be sent without incurring network-layer fragmentation at the sender. It is specified as a number of bytes and is less than or equal to the Maximum Message Size on Send. It exposes a readable value to the application based on the Maximum Packet Size (MPS). The value of this property can change over time (and can be updated by Datagram PLPMTUD [RFC8899]). This value allows a sending stack to avoid unwanted fragmentation at the network-layer or segmentation by the transport layer before choosing the message size and/or after a SendError occurs indicating an attempt to send a Message that is too large. A Transport Services API could express Not applicable in an environment-typical way, e.g., as a Union type or special value (e.g., 0).

8.1.11.5. Maximum Message Size on Send

Name: sendMsgMaxLen

Type: Integer (non-negative)

This property represents the maximum Message size that an application can send. It is specified as the number of bytes. A value of 0 indicates that sending is not possible.

8.1.11.6. Maximum Message Size on Receive

Name: recvMsgMaxLen

Type: Integer (non-negative)

This property represents the maximum Message size that an application can receive. It is specified as the number of bytes. A value of 0 indicates that receiving is not possible.

8.2. TCP-specific Properties: User Timeout Option (UTO)

These properties specify configurations for the TCP User Timeout Option (UTO). This is a TCP-specific property, that is only used in the case that TCP becomes the chosen transport protocol and useful only if TCP is implemented in the Transport Services system. Protocol-specific options could also be defined for other transport protocols.

These are included here because the feature Suggest timeout to the peer is part of the minimal set of transport services [RFC8923], where this feature was categorized as "functional". This means that when a Transport Services system offers this feature, the Transport Services API has to expose an interface to the application. Otherwise, the implementation might violate assumptions by the application, which could cause the application to fail.

All of the below properties are optional (e.g., it is possible to specify User Timeout Enabled as true, but not specify an Advertised User Timeout value; in this case, the TCP default will be used). These properties reflect the API extension specified in Section 3 of [RFC5482].

8.2.1. Advertised User Timeout

Name: tcp.userTimeoutValue

Type: Integer (positive)

Default: the TCP default

This time value is advertised via the TCP User Timeout Option (UTO) [RFC5482] to the Remote Endpoint which can use it to adapt its own Timeout for aborting Connection (see Section 8.1.3) value.

8.2.2. User Timeout Enabled

Name: tcp.userTimeoutEnabled

Type: Boolean

Default: false

This property controls whether the TCP UTO option is enabled for a connection. This applies to both sending and receiving.

8.2.3. Timeout Changeable

Name: tcp.userTimeoutChangeable

Type: Boolean

Default: true

This property controls whether the TCP connTimeout (see Section 8.1.3) can be changed based on a UTO option received from the remote peer. This boolean becomes false when connTimeout (see Section 8.1.3) is used.

8.3. Connection Lifecycle Events

During the lifetime of a Connection there are events that can occur when configured.

8.3.1. Soft Errors

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

Connection -> SoftError<>

8.3.2. Path change

This event notifies the application when at least one of the paths underlying a Connection has changed. Changes occur on a single path when the PMTU changes as well as when multiple paths are used and paths are added or removed, the set of local endpoints changes, or a handover has been performed.

```
Connection -> PathChange<>
```

9. Data Transfer

Data is sent and received as Messages, which allows the application to communicate the boundaries of the data being transferred.

9.1. Messages and Framers

Each Message has an optional Message Context, which allows adding Message Properties, identify Send events related to a specific Message or to inspect meta-data related to the Message sent. Framers can be used to extend or modify the Message data with additional information that can be processed at the receiver to detect message boundaries.

9.1.1. Message Contexts

Using the MessageContext object, the application can set and retrieve meta-data of the Message, including Message Properties (see Section 9.1.3) and framing meta-data (see Section 9.1.2.2). Therefore, a MessageContext object can be passed to the Send action and is returned by each Send and Receive related event.

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

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

These Message Properties can be generic properties or Protocol-specific Properties.

For MessageContexts returned by Send events (see Section 9.2.2) and Receive events (see Section 9.3.2), the application can query information about the Local and Remote Endpoint:

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

9.1.2. Message Framers

Although most applications communicate over a network using well-formed Messages, the boundaries and metadata of the Messages are often not directly communicated by the transport protocol itself. For example, HTTP applications send and receive HTTP messages over a byte-stream transport, requiring that the boundaries of HTTP messages be parsed from the stream of bytes.

Message Framers allow extending a Connection's Protocol Stack to define how to encapsulate or encode outbound Messages, and how to decapsulate or decode inbound data into Messages. Message Framers allow message boundaries to be preserved when using a Connection object, even when using byte-stream transports. This is designed based on the fact that many of the current application protocols evolved over TCP, which does not provide message boundary preservation, and since many of these protocols require message boundaries to function, each application layer protocol has defined its own framing.

To use a Message Framer, the application adds it to its Preconnection object. Then, the Message Framer can intercept all calls to Send or Receive on a Connection to add Message semantics, in addition to interacting with the setup and teardown of the Connection. A Framer can start sending data before the application sends data if the framing protocol requires a prefix or handshake (see [RFC9329] for an example of such a framing protocol).

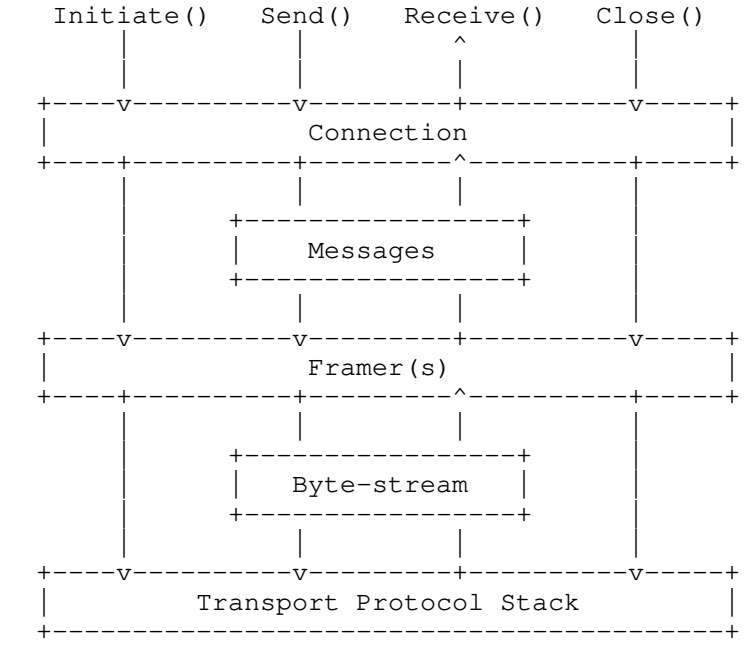


Figure 1: Protocol Stack showing a Message Framer

Note that while Message Framers add the most value when placed above a protocol that otherwise does not preserve message boundaries, they can also be used with datagram- or message-based protocols. In these cases, they add a transformation to further encode or encapsulate, and can potentially support packing multiple application-layer Messages into individual transport datagrams.

The API to implement a Message Framer can vary depending on the implementation; guidance on implementing Message Framers can be found in [I-D.ietf-taps-impl].

9.1.2.1. Adding Message Framers to Pre-Connections

The Message Framer object can be added to one or more Preconnections to run on top of transport protocols. Multiple Framers can be added to a Preconnection; in this case, the Framers operate as a framing stack, i.e. the last one added runs first when framing outbound Messages, and last when parsing inbound data.

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

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

Since Message Framers pass from Preconnection to Listener or Connection, addition of Framers must happen before any operation that might result in the creation of a Connection.

9.1.2.2. Framing Meta-Data

When sending Messages, applications can add Framer-specific properties to a MessageContext (Section 9.1.1) with the add action. To avoid naming conflicts, the property names SHOULD be prefixed with a namespace referencing the framer implementation or the protocol it implements as described in Section 4.1.

This mechanism can be used, for example, to set the type of a Message for a TLV format. The namespace of values is custom for each unique Message Framer.

```
messageContext := NewMessageContext()
messageContext.add(framer, key, value)
Connection.Send(messageData, messageContext)
```

When an application receives a MessageContext in a Receive event, it can also look to see if a value was set by a specific Message Framer.

```
messageContext.get(framer, key) -> value
```

For example, if an HTTP Message Framer is used, the values could correspond to HTTP headers:

```
httpFramer := NewHTTPMessageFramer()
...
messageContext := NewMessageContext()
messageContext.add(httpFramer, "accept", "text/html")
```

9.1.3. Message Properties

Applications needing to annotate the Messages they send with extra information (for example, to control how data is scheduled and processed by the transport protocols supporting the Connection) can include this information in the Message Context passed to the Send action. For other uses of the Message Context, see Section 9.1.1.

Message Properties are per-Message, not per-Send if partial Messages are sent (Section 9.2.3). All data blocks associated with a single Message share properties specified in the Message Contexts. For example, it would not make sense to have the beginning of a Message expire, but allow the end of a Message to still be sent.

A MessageContext object contains metadata for the Messages to be sent or received.

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

The simpler form of Send, which does not take any MessageContext, is equivalent to passing a default MessageContext without adding any Message Properties.

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

Properties can be added to a MessageContext object only before the context is used for sending. Once a MessageContext has been used with a Send action, further modifications to the MessageContext object do not have any effect on this Send call. Message Properties that are not added to a MessageContext object before using the context for sending will either take a specific default value or be

configured based on Selection or Connection Properties of the Connection that is associated with the Send call. This initialization behavior is defined per Message Property below.

The Message Properties could be inconsistent with the properties of the Protocol Stacks underlying the Connection on which a given Message is sent. For example, a Protocol Stack must be able to provide ordering if the msgOrdered property of a Message is enabled. Sending a Message with Message Properties inconsistent with the Selection Properties of the Connection yields an error.

If a Message Property contradicts a Connection Property, and if this per-Message behavior can be supported, it overrides the Connection Property for the specific Message. For example, if reliability is set to Require and a protocol with configurable per-Message reliability is used, setting msgReliable to false for a particular Message will allow this Message to be sent without any reliability guarantees. Changing the msgReliable Message Property is only possible for Connections that were established enabling the Selection Property perMsgReliability. If the contradicting Message Property cannot be supported by the Connection (such as requiring reliability on a Connection that uses an unreliable protocol), the Send action will result in a SendError event.

The following Message Properties are supported:

9.1.3.1. Lifetime

Name: msgLifetime

Type: Numeric (positive)

Default: infinite

The Lifetime specifies how long a particular Message can wait in the Transport Services system before it is sent to the Remote Endpoint. After this time, it is irrelevant and no longer needs to be (re-)transmitted. This is a hint to the Transport Services 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 perMsgReliability property (see Section 9.1.3.7). The type and units of Lifetime are implementation-specific.

9.1.3.2. Priority

Name: msgPriority

Type: Integer (non-negative)

Default: 100

This property specifies the priority of a Message, relative to other Messages sent over the same Connection. A numerically lower value represents a higher priority.

A Message with Priority 2 will yield to a Message with Priority 1, which will yield to a Message with Priority 0, and so on. Priorities can 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 connPriority - see Section 8.1.2. The priority properties might interact, but can be used independently and be realized by different mechanisms; see Section 9.2.6.

9.1.3.3. Ordered

Name: msgOrdered

Type: Boolean

Default: the queried Boolean value of the Selection Property preserveOrder (Section 6.2.4)

The order in which Messages were submitted for transmission via the Send action will be preserved on delivery via Receive events for all Messages on a Connection that have this Message Property set to true.

If false, the Message is delivered to the receiving application without preserving the ordering. This property is used for protocols that support preservation of data ordering, see Section 6.2.4, but allow out-of-order delivery for certain Messages, e.g., by multiplexing independent Messages onto different streams.

If it is not configured by the application before sending, this property's default value will be based on the Selection Property preserveOrder of the Connection associated with the Send action.

9.1.3.4. Safely Replayable

Name: safelyReplayable

Type: Boolean

Default: false

If true, safelyReplayable specifies that a Message is safe to send to the Remote Endpoint more than once for a single Send action. It marks the data as safe for certain 0-RTT establishment techniques, where retransmission of the 0-RTT data could cause the remote application to receive the Message multiple times.

For protocols that do not protect against duplicated Messages, e.g., UDP, all Messages need to be marked as "safely replayable" by enabling this property. To enable protocol selection to choose such a protocol, safelyReplayable needs to be added to the TransportProperties passed to the Preconnection. If such a protocol was chosen, disabling safelyReplayable on individual Messages MUST result in a SendError.

9.1.3.5. Final

Name: final

Type: Boolean

Default: false

If true, this indicates a Message is the last that the application will send on a Connection. This allows underlying protocols to indicate to the Remote Endpoint that the Connection has been effectively closed in the sending direction. For example, TCP-based Connections can send a FIN once a Message marked as Final has been completely sent, indicated by marking endOfMessage. Protocols that do not support signalling the end of a Connection in a given direction will ignore this property.

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

9.1.3.6. Sending Corruption Protection Length

Name: msgChecksumLen

Type: Integer (non-negative) or Full Coverage

Default: Full Coverage

If this property is an Integer, it specifies the minimum length of the section of a sent Message, starting from byte 0, that the application requires to be delivered without corruption due to lower layer errors. It is used to specify options for simple integrity protection via checksums. A value of 0 means that no checksum needs to be calculated, and the enumerated value Full Coverage means that the entire Message needs to be protected by a checksum. Only Full Coverage is guaranteed, any other requests are advisory, which may result in Full Coverage being applied.

9.1.3.7. Reliable Data Transfer (Message)

Name: msgReliable

Type: Boolean

Default: the queried Boolean value of the Selection Property reliability (Section 6.2.1)

When true, this property specifies that a Message should be sent in such a way that the transport protocol ensures all data is received by the Remote Endpoint. Changing the msgReliable property on Messages is only possible for Connections that were established enabling the Selection Property perMsgReliability. When this is not the case, changing msgReliable will generate an error.

Disabling this property indicates that the Transport Services system could disable retransmissions or other reliability mechanisms for this particular Message, but such disabling is not guaranteed.

If it is not configured by the application before sending, this property's default value will be based on the Selection Property reliability of the Connection associated with the Send action.

9.1.3.8. Message Capacity Profile Override

Name: msgCapacityProfile

Type: Enumeration

Default: inherited from the Connection Property connCapacityProfile (Section 8.1.6)

This enumerated property specifies the application's preferred tradeoffs for sending this Message; it is a per-Message override of the connCapacityProfile Connection Property (see Section 8.1.6). If it is not configured by the application before sending, this property's default value will be based on the Connection Property connCapacityProfile of the Connection associated with the Send action.

9.1.3.9. No Network-Layer Fragmentation

Name: noFragmentation

Type: Boolean

Default: false

This property specifies that a Message should be sent and received without network-layer fragmentation, if possible. It can be used to avoid network layer fragmentation when transport segmentation is preferred.

This only takes effect when the transport uses a network layer that supports this functionality. When it does take effect, setting this property to true will cause the sender to avoid network-layer source fragmentation. When using IPv4, this will result in the Don't Fragment bit being set in the IP header.

Attempts to send a Message with this property that result in a size greater than the transport's current estimate of its maximum packet size (singularTransmissionMsgMaxLen) can result in transport segmentation when permitted, or in a SendError.

Note: noSegmentation is used when it is desired to only send a Message within a single network packet.

9.1.3.10. No Segmentation

Name: noSegmentation

Type: Boolean

Default: false

When set to true, this property requests the transport layer to not provide segmentation of Messages larger than the maximum size permitted by the network layer, and also to avoid network-layer source fragmentation of Messages. When running over IPv4, setting this property to true will result in a sending endpoint setting the Don't Fragment bit in the IPv4 header of packets generated by the transport layer.

An attempt to send a Message that results in a size greater than the transport's current estimate of its maximum packet size (singularTransmissionMsgMaxLen) will result in a SendError. This only takes effect when the transport and network layer support this functionality.

9.2. Sending Data

Once a Connection has been established, it can be used for sending Messages. By default, Send enqueues a complete Message, and takes optional per-Message properties (see Section 9.2.1). All Send actions are asynchronous, and deliver events (see Section 9.2.2). Sending partial Messages for streaming large data is also supported (see Section 9.2.3).

Messages are sent on a Connection using the Send action:

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

where messageData is the data object to send, and messageContext allows adding Message Properties, identifying Send events related to a specific Message or inspecting meta-data related to the Message sent (see Section 9.1.1).

The optional endOfMessage parameter supports partial sending and is described in Section 9.2.3.

9.2.1. Basic Sending

The most basic form of sending on a Connection involves enqueuing a single Data block as a complete Message with default Message Properties.

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

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

Some transport protocols can deliver arbitrarily sized Messages, but other protocols constrain the maximum Message size. Applications can query the Connection Property sendMsgMaxLen (Section 8.1.11.5) to determine the maximum size allowed for a single Message. If a Message is too large to fit in the Maximum Message Size for the Connection, the Send will fail with a SendError event (Section 9.2.2.3). For example, it is invalid to send a Message over a UDP connection that is larger than the available datagram sending size.

9.2.2. Send Events

Like all actions in Transport Services API, the Send action is asynchronous. There are several events that can be delivered in response to sending a Message. Exactly one event (Sent, Expired, or SendError) will be delivered in response to each call to Send.

Note that if partial Send calls are used (Section 9.2.3), there will still be exactly one Send event delivered for each call to Send. For example, if a Message expired while two requests to Send data for that Message are outstanding, there will be two Expired events delivered.

The Transport Services API should allow the application to correlate which Send action resulted in a particular Send event. The manner in which this correlation is indicated is implementation-specific.

9.2.2.1. Sent

Connection -> Sent<messageContext>

The Sent event occurs when a previous Send call has completed, i.e., when the data derived from the Message has been passed down or through the underlying Protocol Stack and is no longer the responsibility of the Transport Services API. The exact disposition of the Message (i.e., whether it has actually been transmitted, moved into a buffer on the network interface, moved into a kernel buffer, and so on) when the Sent event occurs is implementation-specific. The Sent event contains a reference to the Message Context of the Message to which it applies.

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

9.2.2.2. Expired

Connection -> Expired<messageContext>

The Expired event occurs when a previous Send action expired before completion; i.e. when the Message was not sent before its Lifetime (see Section 9.1.3.1) expired. This is separate from SendError, as it is an expected behavior for partially reliable transports. The Expired event contains a reference to the Message Context of the Message to which it applies.

9.2.2.3. SendError

Connection -> SendError<messageContext, reason?>

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

9.2.3. Partial Sends

It is not always possible for an application to send all data associated with a Message in a single Send action. The Message data might be too large for the application to hold in memory at one time, or the length of the Message might be unknown or unbounded.

Partial Message sending is supported by passing an endOfMessage Boolean parameter to the Send action. This value is always true by default, and the simpler forms of Send are equivalent to passing true for endOfMessage.

The following example sends a Message in two separate calls to Send.

```
messageContext := NewMessageContext()
messageContext.add(parameter, value)

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

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

All data sent with the same MessageContext object will be treated as belonging to the same Message, and will constitute an in-order series until the endOfMessage is marked.

9.2.4. Batching Sends

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

The semantics for starting and ending a batch can be implementation-specific, but need to allow multiple Send actions to be enqueued.

```
Connection.StartBatch()
Connection.Send(messageData)
Connection.Send(messageData)
Connection.EndBatch()
```

9.2.5. Send on Active Open: InitiateWithSend

For application-layer protocols where the Connection initiator also sends the first Message, the InitiateWithSend action combines Connection initiation with a first Message sent:

```
Connection := Preconnection.InitiateWithSend(messageData,
                                             messageContext?,
                                             timeout?)
```

Whenever possible, a MessageContext should be provided to declare the Message passed to InitiateWithSend as "safely replayable" using the safelyReplayable property. This allows the Transport Services system to make use of 0-RTT establishment in case this is supported by the available Protocol Stacks. When the selected stack(s) do not support transmitting data upon connection establishment, InitiateWithSend is identical to Initiate followed by Send.

Neither partial sends nor send batching are supported by InitiateWithSend.

The events that are sent after InitiateWithSend are equivalent to those that would be sent by an invocation of Initiate followed immediately by an invocation of Send, with the caveat that a send failure that occurs because the Connection could not be established will not result in a SendError separate from the EstablishmentError signaling the failure of Connection establishment.

9.2.6. Priority and the Transport Services API

The Transport Services API provides two properties to allow a sender to signal the relative priority of data transmission: msgPriority Section 9.1.3.2 and connPriority Section 8.1.2. These properties are designed to allow the expression and implementation of a wide variety of approaches to transmission priority in the transport and application layer, including those which do not appear on the wire (affecting only sender-side transmission scheduling) as well as those that do (e.g. [RFC9218]. A Transport Services system gives no guarantees about how its expression of relative priorities will be realized.

The Transport Services API does order connPriority over msgPriority. In the absence of other externalities (e.g., transport-layer flow control), a priority 1 Message on a priority 0 Connection will be sent before a priority 0 Message on a priority 1 Connection in the same group.

9.3. Receiving Data

Once a Connection is established, it can be used for receiving data (unless the direction property is set to unidirectional send). As with sending, the data is received in Messages. Receiving is an asynchronous operation, in which each call to Receive enqueues a request to receive new data from the Connection. Once data has been received, or an error is encountered, an event will be delivered to complete any pending Receive requests (see Section 9.3.2). If Messages arrive at the Transport Services system before Receive requests are issued, ensuing Receive requests will first operate on

these Messages before awaiting any further Messages.

9.3.1. Enqueuing Receives

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

Connection.Receive(minIncompleteLength?, maxLength?)

By default, Receive will try to deliver complete Messages in a single event (Section 9.3.2.1).

The application can set a minIncompleteLength value to indicate the smallest partial Message data size in bytes to be delivered in response to this Receive. By default, this value is infinite, which means that only complete Messages should be delivered (see Section 9.3.2.2 and Section 9.1.2 for more information on how this is accomplished). If this value is set to some smaller value, the associated receive event will be triggered only when at least that many bytes are available, or the Message is complete with fewer bytes, or the system needs to free up memory. Applications SHOULD always check the length of the data delivered to the receive event and not assume it will be as long as minIncompleteLength in the case of shorter complete Messages or memory issues.

The maxLength argument indicates the maximum size of a Message in bytes that the application is currently prepared to receive. The default value for maxLength is infinite. If an incoming Message is larger than the minimum of this size and the maximum Message size on receive for the Connection's Protocol Stack, it will be delivered via ReceivedPartial events (Section 9.3.2.2).

Note that maxLength does not guarantee that the application will receive that many bytes if they are available; the Transport Services API could return ReceivedPartial events with less data than maxLength according to implementation constraints. Note also that maxLength and minIncompleteLength are intended only to manage buffering, and are not interpreted as a receiver preference for Message reordering.

9.3.2. Receive Events

Each call to Receive will be paired with a single Receive event. This allows an application to provide backpressure to the transport stack when it is temporarily not ready to receive Messages. For example, an application that will later be able to handle multiple receive events at the same time can make multiple calls to Receive without waiting for, or processing, any receive events. An

application that is temporarily unable to process received events for a connection could refrain from calling Receive or delay calling it. This would lead to a build-up of unread data, which, in turn, could result in backpressure to the sender via a transport protocol's flow control.

The Transport Services API should allow the application to correlate which call to Receive resulted in a particular Receive event. The manner in which this correlation is indicated is implementation-specific.

9.3.2.1. Received

```
Connection -> Received<messageData, messageContext>
```

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

The messageData value provides access to the bytes that were received for this Message, along with the length of the byte array. The messageContext value is provided to enable retrieving metadata about the Message and referring to the Message. The MessageContext object is described in Section 9.1.1.

See Section 9.1.2 for handling Message framing in situations where the Protocol Stack only provides a byte-stream transport.

9.3.2.2. ReceivedPartial

```
Connection -> ReceivedPartial<messageData, messageContext,
                              endOfMessage>
```

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

Multiple invocations of ReceivedPartial deliver data for the same Message by passing the same MessageContext, until the endOfMessage flag is delivered or a ReceiveError occurs. All partial blocks of a single Message are delivered in order without gaps. This event does not support delivering non-contiguous partial Messages. If, for example, Message A is divided into three pieces (A1, A2, A3) and Message B is divided into three pieces (B1, B2, B3), and preserveOrder is not Required, the ReceivedPartial could deliver them in a sequence like this: A1, B1, B2, A2, A3, B3, because the MessageContext allows the application to identify the pieces as belonging to Message A and B, respectively. However, a sequence like: A1, A3 will never occur.

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

* the underlying Protocol Stack supports message boundary preservation, and the size of the Message is larger than the buffers available for a single Message;

* the underlying Protocol Stack does not support message boundary preservation, and the Message Framer (see Section 9.1.2) cannot determine the end of the Message using the buffer space it has available; or

* the underlying Protocol Stack does not support message boundary preservation, and no Message Framer was supplied by the application

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

In the following example, an application only wants to receive up to 1000 bytes at a time from a Connection. If a 1500-byte Message arrives, it would receive the Message in two separate ReceivedPartial events.

```
Connection.Receive(1, 1000)

// Receive first 1000 bytes, message is incomplete
Connection -> ReceivedPartial<messageData(1000 bytes),
                              messageContext, false>

Connection.Receive(1, 1000)

// Receive last 500 bytes, message is now complete
Connection -> ReceivedPartial<messageData(500 bytes),
                              messageContext, true>
```

9.3.2.3. ReceiveError

```
Connection -> ReceiveError<messageContext, reason?>
```

A ReceiveError occurs when data is received by the underlying Protocol Stack that cannot be fully retrieved or parsed, and when it is useful for the application to be notified of such errors. For example, a ReceiveError can indicate that a Message (identified via the messageContext value) that was being partially received previously, but had not completed, encountered an error and will not be completed. This can be useful for an application, which might wish to use this error as a hint to remove previously received Message parts from memory. As another example, if an incoming Message does not fulfill the recvChecksumLen property (see Section 8.1.1), an application can use this error as a hint to inform the peer application to adjust the msgChecksumLen property (see Section 9.1.3.6).

In contrast, internal protocol reception errors (e.g., loss causing retransmissions in TCP) are not signalled by this event. Conditions that irrevocably lead to the termination of the Connection are signaled using ConnectionError (see Section 10).

9.3.3. Receive Message Properties

Each Message Context could contain metadata from protocols in the Protocol Stack; which metadata is available is Protocol Stack dependent. These are exposed through additional read-only Message Properties that can be queried from the MessageContext object (see Section 9.1.1) passed by the receive event. The following metadata values are supported:

9.3.3.1. UDP(-Lite)-specific Property: ECN

When available, Message metadata carries the value of the Explicit Congestion Notification (ECN) field. This information can be used for logging and debugging, and for building applications that need access to information about the transport internals for their own operation. This property is specific to UDP and UDP-Lite because these protocols do not implement congestion control, and hence expose this functionality to the application (see [RFC8293], following the guidance in [RFC8085])

9.3.3.2. Early Data

In some cases it can be valuable to know whether data was read as part of early data transfer (before Connection establishment has finished). This is useful if applications need to treat early data separately, e.g., if early data has different security properties than data sent after connection establishment. In the case of TLS 1.3, client early data can be replayed maliciously (see [RFC8446]). Thus, receivers might wish to perform additional checks for early data to ensure it is safely replayable. If TLS 1.3 is available and the recipient Message was sent as part of early data, the corresponding metadata carries a flag indicating as such. If early data is enabled, applications should check this metadata field for Messages received during Connection establishment and respond accordingly.

9.3.3.3. Receiving Final Messages

The Message Context can indicate whether or not this Message is the Final Message on a Connection. For any Message that is marked as Final, the application can assume that there will be no more Messages received on the Connection once the Message has been completely delivered. This corresponds to the final property that can be marked on a sent Message, see Section 9.1.3.5.

Some transport protocols and peers do not support signaling of the final property. Applications therefore SHOULD NOT rely on receiving a Message marked Final to know that the sending endpoint is done sending on a Connection.

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

10. Connection Termination

A Connection can be terminated i) by the Local Endpoint (i.e., the application calls the Close, CloseGroup, Abort or AbortGroup action), ii) by the Remote Endpoint (i.e., the remote application calls the Close, CloseGroup, Abort or AbortGroup action), or iii) because of an error (e.g., a timeout). A local call of the Close action will cause the Connection to either send a Closed event or a ConnectionError event, and a local call of the CloseGroup action will cause all of the Connections in the group to either send a Closed event or a ConnectionError event. A local call of the Abort action will cause the Connection to send a ConnectionError event, indicating local Abort as a reason, and a local call of the AbortGroup action will cause all of the Connections in the group to send a ConnectionError event, indicating local Abort as a reason.

Remote action calls map to events similar to local calls (e.g., a remote Close causes the Connection to either send a Closed event or a ConnectionError event), but, different from local action calls, it is not guaranteed that such events will indeed be invoked. When an application needs to free resources associated with a Connection, it ought not to therefore rely on the invocation of such events due to termination calls from the Remote Endpoint, but instead use the local termination actions.

Close terminates a Connection after satisfying all the requirements that were specified regarding the delivery of Messages that the application has already given to the Transport Services system. Upon successfully satisfying all these requirements, the Connection will send the Closed event. For example, if reliable delivery was requested for a Message handed over before calling Close, the Closed event will signify that this Message has indeed been delivered. This action does not affect any other Connection in the same Connection Group.

An application MUST NOT assume that it can receive any further data on a Connection for which it has called Close, even if such data is already in flight.

Connection.Close()

The Closed event informs the application that a Close action has successfully completed, or that the Remote Endpoint has closed the Connection. There is no guarantee that a remote Close will be signaled.

Connection -> Closed<>

Abort terminates a Connection without delivering any remaining Messages. This action does not affect any other Connection that is entangled with this one in a Connection Group. When the Abort action has finished, the Connection will send a ConnectionError event, indicating local Abort as a reason.

Connection.Abort()

CloseGroup gracefully terminates a Connection and any other Connections in the same Connection Group. For example, all of the Connections in a group might be streams of a single session for a multistreaming protocol; closing the entire group will close the underlying session. See also Section 7.4. All Connections in the group will send a Closed event when the CloseGroup action was successful. As with Close, any Messages remaining to be processed on a Connection will be handled prior to closing.

Connection.CloseGroup()

AbortGroup terminates a Connection and any other Connections that are in the same Connection Group without delivering any remaining Messages. When the AbortGroup action has finished, all Connections in the group will send a ConnectionError event, indicating local Abort as a reason.

Connection.AbortGroup()

A ConnectionError informs the application that: 1) data could not be delivered to the peer after a timeout, or 2) the Connection has been aborted (e.g., because the peer has called Abort). There is no guarantee that an Abort from the peer will be signaled.

Connection -> ConnectionError<reason?>

11. Connection State and Ordering of Operations and Events

This Transport Services API is designed to be independent of an implementation's concurrency model. The details of how exactly actions are handled, and how events are dispatched, are implementation dependent.

Some transitions of Connection states are associated with events:

* Ready<> occurs when a Connection created with Initiate or InitiateWithSend transitions to Established state.

* ConnectionReceived<> occurs when a Connection created with Listen transitions to Established state.

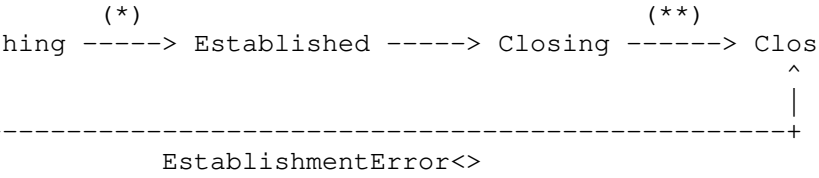
* RendezvousDone<> occurs when a Connection created with Rendezvous transitions to Established state.

* Closed<> occurs when a Connection transitions to Closed state without error.

* EstablishmentError<> occurs when a Connection created with Initiate transitions from Establishing state to Closed state due to an error.

* ConnectionError<> occurs when a Connection transitions to Closed state due to an error in all other circumstances.

The following diagram shows the possible states of a Connection and the events that occur upon a transition from one state to another.



(*) Ready<>, ConnectionReceived<>, RendezvousDone<>
(**) Closed<>, ConnectionError<>

Figure 2: Connection State Diagram

The Transport Services API provides the following guarantees about the ordering of operations:

* Sent<> events will occur on a Connection in the order in which the Messages were sent (i.e., delivered to the kernel or to the network interface, depending on implementation).

* Received<> will never occur on a Connection before it is Established; i.e. before a Ready<> event on that Connection, or a ConnectionReceived<> or RendezvousDone<> containing that Connection.

* No events will occur on a Connection after it is closed; i.e., after a Closed<> event, an EstablishmentError<> or ConnectionError<> will not occur on that Connection. To ensure this ordering, Closed<> will not occur on a Connection while other events on the Connection are still locally outstanding (i.e., known to the Transport Services API and waiting to be dealt with by the application).

12. IANA Considerations

This document has no actions for IANA. Later versions of this document might create IANA registries for generic transport property names and transport property namespaces (see Section 4.1).

13. Privacy and Security Considerations

This document describes a generic API for interacting with a Transport Services system. Part of this API includes configuration details for transport security protocols, as discussed in Section 6.3. It does not recommend use (or disuse) of specific algorithms or protocols. Any API-compatible transport security protocol ought to work in a Transport Services system. Security considerations for these protocols are discussed in the respective specifications.

[I-D.ietf-taps-arch] provides general security considerations and requirements for any system that implements the Transport Services architecture. These include recommendations of relevance to the API, e.g. regarding the use of keying material.

The described API is used to exchange information between an application and the Transport Services system. While it is not necessarily expected that both systems are implemented by the same authority, it is expected that the Transport Services Implementation is either provided as a library that is selected by the application from a trusted party, or that it is part of the operating system that the application also relies on for other tasks.

In either case, the Transport Services API is an internal interface that is used to exchange information locally between two systems. However, as the Transport Services system is responsible for network communication, it is in the position to potentially share any information provided by the application with the network or another communication peer. Most of the information provided over the Transport Services API are useful to configure and select protocols and paths and are not necessarily privacy-sensitive. Still, some information could be privacy sensitive because it might reveal usage characteristics and habits of the user of an application.

Of course any communication over a network reveals usage characteristics, because all packets, as well as their timing and size, are part of the network-visible wire image [RFC8546]. However, the selection of a protocol and its configuration also impacts which information is visible, potentially in clear text, and which other entities can access it. How Transport Services systems ought to choose protocols depending on the security properties required is out

of scope of this specification, as it is limited to transport protocols. The choice of a security protocol can be informed by the survey provided in [RFC8922].

In most cases, information provided for protocol and path selection does not directly translate to information that can be observed by network devices on the path. However, there might be specific configuration information that is intended for path exposure, e.g., a DiffServ codepoint setting, that is either provided directly by the application or indirectly configured for a traffic profile.

Applications should be aware that a single communication attempt can lead to more than one connection establishment procedure. This is the case, for example, when the Transport Services system also executes name resolution, when support mechanisms such as TURN or ICE are used to establish connectivity, if protocols or paths are raced, or if a path fails and fallback or re-establishment is supported in the Transport Services system. Applications should take special care when using 0-RTT session resumption (see Section 6.2.5), as early data sent across multiple paths during connection establishment could reveal information that can be used to correlate endpoints on these paths.

Applications should also take care to not assume that all data received using the Transport Services API is always complete or well-formed. Specifically, Messages that are received partially Section 9.3.2.2 could be a source of truncation attacks if applications do not distinguish between partial Messages and complete Messages.

The Transport Services API explicitly does not require the application to resolve names, though there is a tradeoff between early and late binding of addresses to names. Early binding allows the Transport Services 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. Names used with the Transport Services API SHOULD be fully-qualified domain names (FQDNs); not providing an FQDN will result in the Transport Services Implementation needing to to use DNS search domains for name resolution, which might lead to inconsistent or unpredictable behavior.

These communication activities are not different from what is used today. However, the goal of a Transport Services system is to support such mechanisms as a generic service within the transport

layer. This enables applications to more dynamically benefit from innovations and new protocols in the transport, although it reduces transparency of the underlying communication actions to the application itself. The Transport Services API is designed such that protocol and path selection can be limited to a small and controlled set if the application requires this or to implement a security policy. can be limited to a small and controlled set if required by the application to perform a function or to provide security. Further, introspection on the properties of Connection objects allows an application to determine which protocol(s) and path(s) are in use. A Transport Services system SHOULD provide a facility logging the communication events of each Connection.

14. Acknowledgments

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

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

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

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

Thanks to Stuart Cheshire, Josh Graessley, David Schinazi, and Eric Kinnear for their implementation and design efforts, including Happy Eyeballs, that heavily influenced this work. Thanks to Laurent Chuat and Jason Lee for initial work on the Post Sockets interface, from which this work has evolved. Thanks to Maximilian Franke for asking good questions based on implementation experience and for contributing text, e.g., on multicast.

15. References

15.1. Normative References

[ALPN] 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/rfc/rfc7301>.

[I-D.ietf-taps-arch]
Pauly, T., Trammell, B., Brunstrom, A., Fairhurst, G., and C. Perkins, "Architecture and Requirements for Transport Services", Work in Progress, Internet-Draft, draft-ietf-taps-arch-19, 9 November 2023, <https://datatracker.ietf.org/doc/html/draft-ietf-taps-arch-19>.

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

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

15.2. Informative References

[I-D.ietf-taps-impl]
Brunstrom, A., Pauly, T., Enghardt, R., Tiesel, P. S., and M. Welzl, "Implementing Interfaces to Transport Services", Work in Progress, Internet-Draft, draft-ietf-taps-impl-18, 14 December 2023, <https://datatracker.ietf.org/doc/html/draft-ietf-taps-impl-18>.

[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/rfc/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/rfc/rfc2597>.

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

[RFC3246] Davie, B., Charny, A., Bennet, J.C.R., Benson, K., Le Boudec, J.Y., Courtney, W., Davari, S., Firoiu, V., and D. Stiliadis, "An Expedited Forwarding PHB (Per-Hop Behavior)", RFC 3246, DOI 10.17487/RFC3246, March 2002, <https://www.rfc-editor.org/rfc/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/rfc/rfc3261>.

[RFC4291] Hinden, R. and S. Deering, "IP Version 6 Addressing Architecture", RFC 4291, DOI 10.17487/RFC4291, February 2006, <https://www.rfc-editor.org/rfc/rfc4291>.

[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/rfc/rfc4594>.

[RFC5280] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, DOI 10.17487/RFC5280, May 2008, <https://www.rfc-editor.org/rfc/rfc5280>.

[RFC5482] Eggert, L. and F. Gont, "TCP User Timeout Option", RFC 5482, DOI 10.17487/RFC5482, March 2009, <https://www.rfc-editor.org/rfc/rfc5482>.

[RFC5865] Baker, F., Polk, J., and M. Dolly, "A Differentiated Services Code Point (DSCP) for Capacity-Admitted Traffic", RFC 5865, DOI 10.17487/RFC5865, May 2010, <https://www.rfc-editor.org/rfc/rfc5865>.

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

[RFC7556] Anipko, D., Ed., "Multiple Provisioning Domain Architecture", RFC 7556, DOI 10.17487/RFC7556, June 2015, <https://www.rfc-editor.org/rfc/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/rfc/rfc7657>.

[RFC791] Postel, J., "Internet Protocol", STD 5, RFC 791, DOI 10.17487/RFC0791, September 1981, <https://www.rfc-editor.org/rfc/rfc791>.

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

[RFC8085] Eggert, L., Fairhurst, G., and G. Shepherd, "UDP Usage Guidelines", BCP 145, RFC 8085, DOI 10.17487/RFC8085, March 2017, <https://www.rfc-editor.org/rfc/rfc8085>.

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

[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/rfc/rfc8260>.

[RFC8293] Ghanwani, A., Dunbar, L., McBride, M., Bannai, V., and R. Krishnan, "A Framework for Multicast in Network Virtualization over Layer 3", RFC 8293, DOI 10.17487/RFC8293, January 2018, <https://www.rfc-editor.org/rfc/rfc8293>.

[RFC8303] Welzl, M., Tuexen, M., and N. Khademi, "On the Usage of Transport Features Provided by IETF Transport Protocols", RFC 8303, DOI 10.17487/RFC8303, February 2018, <https://www.rfc-editor.org/rfc/rfc8303>.

[RFC8445] Keranen, A., Holmberg, C., and J. Rosenberg, "Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal", RFC 8445, DOI 10.17487/RFC8445, July 2018, <https://www.rfc-editor.org/rfc/rfc8445>.

[RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <https://www.rfc-editor.org/rfc/rfc8446>.

[RFC8489] Petit-Huguenin, M., Salgueiro, G., Rosenberg, J., Wing, D., Mahy, R., and P. Matthews, "Session Traversal Utilities for NAT (STUN)", RFC 8489, DOI 10.17487/RFC8489, February 2020, <https://www.rfc-editor.org/rfc/rfc8489>.

[RFC8546] Trammell, B. and M. Kuehlewind, "The Wire Image of a Network Protocol", RFC 8546, DOI 10.17487/RFC8546, April 2019, <https://www.rfc-editor.org/rfc/rfc8546>.

[RFC8622] Bless, R., "A Lower-Effort Per-Hop Behavior (LE PHB) for Differentiated Services", RFC 8622, DOI 10.17487/RFC8622, June 2019, <https://www.rfc-editor.org/rfc/rfc8622>.

[RFC8656] Reddy, T., Ed., Johnston, A., Ed., Matthews, P., and J. Rosenberg, "Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN)", RFC 8656, DOI 10.17487/RFC8656, February 2020, <https://www.rfc-editor.org/rfc/rfc8656>.

[RFC8699] Islam, S., Welzl, M., and S. Gjessing, "Coupled Congestion Control for RTP Media", RFC 8699, DOI 10.17487/RFC8699, January 2020, <https://www.rfc-editor.org/rfc/rfc8699>.

[RFC8801] Pfister, P., Vyncke, É., Pauly, T., Schinazi, D., and W. Shao, "Discovering Provisioning Domain Names and Data", RFC 8801, DOI 10.17487/RFC8801, July 2020, <https://www.rfc-editor.org/rfc/rfc8801>.

[RFC8838] Ivov, E., Uberti, J., and P. Saint-Andre, "Trickle ICE: Incremental Provisioning of Candidates for the Interactive Connectivity Establishment (ICE) Protocol", RFC 8838, DOI 10.17487/RFC8838, January 2021, <https://www.rfc-editor.org/rfc/rfc8838>.

[RFC8899] Fairhurst, G., Jones, T., Tüxen, M., Rüngeler, I., and T. Völker, "Packetization Layer Path MTU Discovery for Datagram Transports", RFC 8899, DOI 10.17487/RFC8899, September 2020, <https://www.rfc-editor.org/rfc/rfc8899>.

[RFC8922] Enghardt, T., Pauly, T., Perkins, C., Rose, K., and C. Wood, "A Survey of the Interaction between Security Protocols and Transport Services", RFC 8922, DOI 10.17487/RFC8922, October 2020, <https://www.rfc-editor.org/rfc/rfc8922>.

[RFC8923] Welzl, M. and S. Gjessing, "A Minimal Set of Transport Services for End Systems", RFC 8923, DOI 10.17487/RFC8923, October 2020, <https://www.rfc-editor.org/rfc/rfc8923>.

[RFC8981] Gont, F., Krishnan, S., Narten, T., and R. Draves, "Temporary Address Extensions for Stateless Address Autoconfiguration in IPv6", RFC 8981, DOI 10.17487/RFC8981, February 2021, <https://www.rfc-editor.org/rfc/rfc8981>.

[RFC9218] Oku, K. and L. Pardue, "Extensible Prioritization Scheme for HTTP", RFC 9218, DOI 10.17487/RFC9218, June 2022, <https://www.rfc-editor.org/rfc/rfc9218>.

[RFC9329] Pauly, T. and V. Smyslov, "TCP Encapsulation of Internet Key Exchange Protocol (IKE) and IPsec Packets", RFC 9329, DOI 10.17487/RFC9329, November 2022, <https://www.rfc-editor.org/rfc/rfc9329>.

[TCP-COUPLING] Islam, S., Welzl, M., Hiorth, K., Hayes, D., Armitage, G., and S. Gjessing, "ctrlTCP: Reducing Latency through Coupled, Heterogeneous Multi-Flow TCP Congestion Control", IEEE INFOCOM Global Internet Symposium (GI) workshop (GI 2018) , 2018.

Appendix A. Implementation Mapping

The way the concepts from this abstract API map into concrete APIs in a given language on a given platform largely depends on the features and norms of the language and the platform. Actions could be implemented as functions or method calls, for instance, and events could be implemented via event queues, handler functions or classes, communicating sequential processes, or other asynchronous calling conventions.

A.1. Types

The basic types mentioned in Section 1.1 typically have natural correspondences in practical programming languages, perhaps constrained by implementation-specific limitations. For example:

* An Integer can typically be represented in C by an int or long, subject to the underlying platform's ranges for each.

* In C, a Tuple may be represented as a struct with one member for each of the value types in the ordered grouping. In Python, by contrast, a Tuple may be represented as a tuple, a sequence of dynamically-typed elements.

* A Set may be represented as a std::set in C++ or as a set in Python. In C, it may be represented as an array or as a higher-level data structure with appropriate accessors defined.

The objects described in Section 1.1 can similarly be represented in different ways depending on which programming language is used. Objects like Preconnections, Connections, and Listeners can be long-lived, and benefit from using object-oriented constructs. Note that in C, these objects may need to provide a way to release or free their underlying memory when the application is done using them. For example, since a Preconnection can be used to initiate multiple Connections, it is the responsibility of the application to clean up the Preconnection memory if necessary.

A.2. Events and Errors

This specification treats events and errors similarly. Errors, just as any other events, may occur asynchronously in network applications. However, implementations of this API may report errors synchronously, according to the error handling idioms of the implementation platform, where they can be immediately detected, such as by generating an exception when attempting to initiate a Connection with inconsistent Transport Properties. An error can provide an optional reason to the application with further details about why the error occurred.

A.3. Time Duration

Time duration types are implementation-specific. For instance, it could be a number of seconds, number of milliseconds, or a struct timeval in C or a user-defined Duration class in C++.

Appendix B. Convenience Functions

B.1. Adding Preference Properties

TransportProperties will frequently need to set Selection Properties of type Preference, therefore implementations can provide special actions for adding each preference level i.e, TransportProperties.Set(some_property, avoid) is equivalent toTransportProperties.Avoid(some_property)`:

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

B.2. Transport Property Profiles

To ease the use of the Transport Services API, implementations can provide a mechanism to create Transport Property objects (see Section 6.2) that are preconfigured with frequently used sets of properties; the following are in common use in current applications:

B.2.1. reliable-inorder-stream

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

Property	Value
reliability	require
preserveOrder	require
congestionControl	require
preserveMsgBoundaries	no preference

Table 2: reliable-inorder-stream preferences

B.2.2. reliable-message

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

Property	Value
reliability	require
preserveOrder	require
congestionControl	require
preserveMsgBoundaries	require

Table 3: reliable-message preferences

B.2.3. unreliable-datagram

This profile provides a datagram transport service without any reliability guarantee. An example of a protocol that provides this service is UDP. It consists of the following properties:

Property	Value
reliability	avoid
preserveOrder	avoid
congestionControl	no preference
preserveMsgBoundaries	require
safelyReplayable	true

Table 4: unreliable-datagram preferences

Applications that choose this Transport Property Profile would avoid the additional latency that could be introduced by retransmission or reordering in a transport protocol.

Applications that choose this Transport Property Profile to reduce latency should also consider setting an appropriate capacity profile Property, see Section 8.1.6 and might benefit from controlling checksum coverage, see Section 6.2.7 and Section 6.2.8.

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

[RFC8923] identifies a minimal set of transport services that end systems should offer. These services make all non-security-related transport features of TCP, MPTCP, UDP, UDP-Lite, SCTP and LEDBAT available that 1) require interaction with the application, and 2) do not get in the way of a possible implementation over TCP (or, with limitations, UDP). The following text explains how this minimal set is reflected in the present API. For brevity, it is based on the list in Section 4.1 of [RFC8923], updated according to the discussion in Section 5 of [RFC8923]. The present API covers all elements of this section. This list is a subset of the transport features in Appendix A of [RFC8923], which refers to the primitives in "pass 2" (Section 4) of [RFC8303] for further details on the implementation with TCP, MPTCP, UDP, UDP-Lite, SCTP and LEDBAT. This facilitates finding the specifications for implementing the services listed below with these protocols.

* Connect: Initiate action (Section 7.1).

* Listen: Listen action (Section 7.2).

* Specify number of attempts and/or timeout for the first establishment Message: timeout parameter of Initiate (Section 7.1) or InitiateWithSend action (Section 9.2.5).

* Disable MPTCP: multipath property (Section 6.2.14).

* Hand over a Message to reliably transfer (possibly multiple times) before connection establishment: InitiateWithSend action (Section 9.2.5).

* Change timeout for aborting connection (using retransmit limit or time value): connTimeout property, using a time value (Section 8.1.3).

* Timeout event when data could not be delivered for too long: ConnectionError event (Section 10).

* Suggest timeout to the peer: See "TCP-specific Properties: User Timeout Option (UTO)" (Section 8.2).

* Notification of ICMP error message arrival: softErrorNotify (Section 6.2.17) and SoftError event (Section 8.3.1).

* Choose a scheduler to operate between streams of an association: connScheduler property (Section 8.1.5).

* Configure priority or weight for a scheduler: connPriority property (Section 8.1.2).

* "Specify checksum coverage used by the sender" and "Disable checksum when sending": msgChecksumLen property (Section 9.1.3.6) and fullChecksumSend property (Section 6.2.7).

* "Specify minimum checksum coverage required by receiver" and "Disable checksum requirement when receiving": recvChecksumLen property (Section 8.1.1) and fullChecksumRecv property (Section 6.2.8).

* "Specify DF field": noFragmentation property (Section 9.1.3.9).

* Get max. transport-message size that may be sent using a non-fragmented IP packet from the configured interface: singularTransmissionMsgMaxLen property (Section 8.1.11.4).

* Get max. transport-message size that may be received from the configured interface: recvMsgMaxLen property (Section 8.1.11.6).

* Obtain ECN field: This is a read-only Message Property of the MessageContext object (see "UDP(-Lite)-specific Property: ECN" Section 9.3.3.1).

* "Specify DSCP field", "Disable Nagle algorithm", "Enable and configure a Low Extra Delay Background Transfer": as suggested in Section 5.5 of [RFC8923], these transport features are collectively offered via the connCapacityProfile property (Section 8.1.6). Per-Message control ("Request not to bundle messages") is offered via the msgCapacityProfile property (Section 9.1.3.8).

* Close after reliably delivering all remaining data, causing an event informing the application on the other side: this is offered by the Close action with slightly changed semantics in line with the discussion in Section 5.2 of [RFC8923] (Section 10).

* "Abort without delivering remaining data, causing an event informing the application on the other side" and "Abort without delivering remaining data, not causing an event informing the application on the other side": this is offered by the Abort action without promising that this is signaled to the other side. If it is, a ConnectionError event will be invoked at the peer (Section 10).

* "Reliably transfer data, with congestion control", "Reliably transfer a message, with congestion control" and "Unreliably transfer a message": data is transferred via the Send action (Section 9.2). Reliability is controlled via the reliability (Section 6.2.1) property and the msgReliable Message Property (Section 9.1.3.7). Transmitting data as a Message or without delimiters is controlled via Message Framers (Section 9.1.2). The choice of congestion control is provided via the congestionControl property (Section 6.2.9).

* Configurable Message Reliability: the msgLifetime Message Property implements a time-based way to configure message reliability (Section 9.1.3.1).

* "Ordered message delivery (potentially slower than unordered)" and "Unordered message delivery (potentially faster than ordered)": these two transport features are controlled via the Message Property msgOrdered (Section 9.1.3.3).

* Request not to delay the acknowledgment (SACK) of a message: should the protocol support it, this is one of the transport features the Transport Services system can apply when an application uses the connCapacityProfile Property (Section 8.1.6) or the msgCapacityProfile Message Property (Section 9.1.3.8) with value Low Latency/Interactive.

* Receive data (with no message delimiting): Receive action (Section 9.3.1) and Received event (Section 9.3.2.1).

* Receive a message: Receive action (Section 9.3.1) and Received event (Section 9.3.2.1), using Message Framers (Section 9.1.2).

* Information about partial message arrival: Receive action (Section 9.3.1) and ReceivedPartial event (Section 9.3.2.2).

* Notification of send failures: Expired event (Section 9.2.2.2) and SendError event (Section 9.2.2.3).

* Notification that the stack has no more user data to send: applications can obtain this information via the Sent event (Section 9.2.2.1).

* Notification to a receiver that a partial message delivery has been aborted: ReceiveError event (Section 9.3.2.3).

* Notification of Excessive Retransmissions (early warning below abortion threshold): SoftError event (Section 8.3.1).

Authors' Addresses

Brian Trammell (editor)
Google Switzerland GmbH
Gustav-Gull-Platz 1
CH- 8004 Zurich
Switzerland
Email: ietf@trammell.ch

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

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

Godred Fairhurst
University of Aberdeen
Fraser Noble Building
Aberdeen, AB24 3UE
Email: gorry@erg.abdn.ac.uk
URI: http://www.erg.abdn.ac.uk/

Mirja Kuehlewind
Ericsson
Ericsson-Allee 1
Herzogenrath
Germany
Email: mirja.kuehlewind@ericsson.com

Colin Perkins
University of Glasgow
School of Computing Science
Glasgow G12 8QQ
United Kingdom
Email: csp@csperkins.org

Philipp S. Tiesel
SAP SE
George-Stephenson-Straße 7-13
10557 Berlin
Germany
Email: philipp@tiesel.net

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

Network Working Group
Internet-Draft
Intended status: Informational
Expires: 25 October 2020

T. Enghardt
TU Berlin
T. Pauly
Apple Inc.
C. Perkins
University of Glasgow
K. Rose
Akamai Technologies, Inc.
C.A. Wood
Cloudflare

23 April 2020

A Survey of the Interaction Between Security Protocols and Transport Services

draft-ietf-taps-transport-security-12

Abstract

This document provides a survey of commonly used or notable network security protocols, with a focus on how they interact and integrate with applications and transport protocols. Its goal is to supplement efforts to define and catalog transport services by describing the interfaces required to add security protocols. This survey is not limited to protocols developed within the scope or context of the IETF, and those included represent a superset of features a Transport Services system may need to support.

Table of Contents

1. Introduction

Services and features provided by transport protocols have been cataloged in [RFC8095]. This document supplements that work by surveying commonly used and notable network security protocols, and identifying the interfaces between these protocols and both transport protocols and applications. It examines Transport Layer Security (TLS), Datagram Transport Layer Security (DTLS), IETF QUIC, Google QUIC (gQUIC), tcpcrypt, Internet Protocol Security (IPsec), Secure Real-time Transport Protocol (SRTP) with DTLS, WireGuard, CurveCP, and MinimaLT. For each protocol, this document provides a brief description. Then, it describes the interfaces between these protocols and transports in Section 4 and the interfaces between these protocols and applications in Section 5.

A Transport Services system exposes an interface for applications to access various (secure) transport protocol features. The security protocols included in this survey represent a superset of functionality and features a Transport Services system may need to support, both internally and externally (via an API) for applications [I-D.ietf-taps-arch]. Ubiquitous IETF protocols such as (D)TLS, as well as non-standard protocols such as gQUIC, are included despite overlapping features. As such, this survey is not limited to protocols developed within the scope or context of the IETF. Outside of this candidate set, protocols that do not offer new features are omitted. For example, newer protocols such as WireGuard make unique design choices that have implications for and limitations on application usage. In contrast, protocols such as SSH [RFC4253], GRE [RFC2890], L2TP [RFC5641], and ALTS [ALTS] are omitted since they do not provide interfaces deemed unique.

Authentication-only protocols such as TCP-AO [RFC5925] and IPsec Authentication Header (AH) [RFC4302] are excluded from this survey. TCP-AO adds authentication to long-lived TCP connections, e.g., replay protection with per-packet Message Authentication Codes. (TCP-AO obsoletes TCP MD5 "signature" options specified in [RFC2385].) One primary use case of TCP-AO is for protecting BGP connections. Similarly, AH adds per-datagram authentication and integrity, along with replay protection. Despite these improvements, neither protocol sees general use and both lack critical properties important for emergent transport security protocols, such as confidentiality and privacy protections. Such protocols are thus omitted from this survey.

This document only surveys point-to-point protocols; multicast protocols are out of scope.

1.1. Goals

This survey is intended to help identify the most common interface surfaces between security protocols and transport protocols, and between security protocols and applications.

One of the goals of the Transport Services effort is to define a common interface for using transport protocols that allows software using transport protocols to easily adopt new protocols that provide similar feature-sets. The survey of the dependencies security protocols have upon transport protocols can guide implementations in determining which transport protocols are appropriate to be able to use beneath a given security protocol. For example, a security protocol that expects to run over a reliable stream of bytes, like TLS, restricts the set of transport protocols that can be used to those that offer a reliable stream of bytes.

Defining the common interfaces that security protocols provide to applications also allows interfaces to be designed in a way that common functionality can use the same APIs. For example, many security protocols that provide authentication let the application be involved in peer identity validation. Any interface to use a secure transport protocol stack thus needs to allow applications to perform this action during connection establishment.

1.2. Non-Goals

While this survey provides similar analysis to that which was performed for transport protocols in [RFC8095], it is important to distinguish that the use of security protocols requires more consideration.

It is not a goal to allow software implementations to automatically switch between different security protocols, even where their interfaces to transport and applications are equivalent. Even between versions, security protocols have subtly different guarantees and vulnerabilities. Thus, any implementation needs to only use the set of protocols and algorithms that are requested by applications or by a system policy.

Different security protocols also can use incompatible notions of peer identity and authentication, and cryptographic options. It is not a goal to identify a common set of representations for these concepts.

The protocols surveyed in this document represent a superset of functionality and features a Transport Services system may need to support. It does not list all transport protocols that a Transport Services system may need to implement, nor does it mandate that a Transport Service system implement any particular protocol.

A Transport Services system may implement any secure transport protocol that provides the described features. In doing so, it may need to expose an interface to the application to configure these features.

2. Terminology

The following terms are used throughout this document to describe the roles and interactions of transport security protocols (some of which are also defined in [RFC8095]):

* Transport Feature: a specific end-to-end feature that the transport layer provides to an application. Examples include confidentiality, reliable delivery, ordered delivery, and message-versus-stream orientation.

* Transport Service: a set of Transport Features, without an association to any given framing protocol, which provides functionality to an application.

* Transport Services system: a software component that exposes an interface to different Transport Services to an application.

* Transport Protocol: an implementation that provides one or more different transport services using a specific framing and header format on the wire. A Transport Protocol services an application, whether directly or in conjunction with a security protocol.

* Application: an entity that uses a transport protocol for end-to-end delivery of data across the network. This may also be an upper layer protocol or tunnel encapsulation.

* Security Protocol: a defined network protocol that implements one or more security features, such as authentication, encryption, key generation, session resumption, and privacy. Security protocols may be used alongside transport protocols, and in combination with other security protocols when appropriate.

* Handshake Protocol: a protocol that enables peers to validate each other and to securely establish shared cryptographic context.

* Record: Framed protocol messages.

* Record Protocol: a security protocol that allows data to be divided into manageable blocks and protected using shared cryptographic context.

* Session: an ephemeral security association between applications.

* Connection: the shared state of two or more endpoints that persists across messages that are transmitted between these endpoints. A connection is a transient participant of a session, and a session generally lasts between connection instances.

* Peer: an endpoint application party to a session.

* Client: the peer responsible for initiating a session.

* Server: the peer responsible for responding to a session initiation.

3. Transport Security Protocol Descriptions

This section contains brief transport and security descriptions of various security protocols currently used to protect data being sent over a network. These protocols are grouped based on where in the protocol stack they are implemented, which influences which parts of a packet they protect: Generic application payload, application payload for specific application-layer protocols, both application payload and transport headers, or entire IP packets.

Note that not all security protocols can be easily categorized, e.g., as some protocols can be used in different ways or in combination with other protocols. One major reason for this is that channel security protocols often consist of two components:

* A handshake protocol, which is responsible for negotiating parameters, authenticating the endpoints, and establishing shared keys.

* A record protocol, which is used to encrypt traffic using keys and parameters provided by the handshake protocol.

For some protocols, such as tcpcrypt, these two components are tightly integrated. In contrast, for IPsec, these components are implemented in separate protocols: AH and ESP are record protocols, which can use keys supplied by the handshake protocol IKEv2, by other handshake protocols, or by manual configuration. Moreover, some protocols can be used in different ways: While the base TLS protocol as defined in [RFC8446] has an integrated handshake and record protocol, TLS or DTLS can also be used to negotiate keys for other protocols, as in DTLS-SRTP, or the handshake protocol can be used with a separate record layer, as in QUIC [I-D.ietf-quic-transport].

3.1. Application Payload Security Protocols

The following protocols provide security that protects application payloads sent over a transport. They do not specifically protect any headers used for transport-layer functionality.

3.1.1. TLS

TLS (Transport Layer Security) [RFC8446] is a common protocol used to establish a secure session between two endpoints. Communication over this session "prevents eavesdropping, tampering, and message forgery." TLS consists of a tightly coupled handshake and record protocol. The handshake protocol is used to authenticate peers, negotiate protocol options, such as cryptographic algorithms, and derive session-specific keying material. The record protocol is used to marshal and, once the handshake has sufficiently progressed, encrypt, data from one peer to the other. This data may contain handshake messages or raw application data.

3.1.2. DTLS

DTLS (Datagram Transport Layer Security) [RFC6347] [I-D.ietf-tls-dtls13] is based on TLS, but differs in that it is designed to run over unreliable datagram protocols like UDP instead of TCP. DTLS modifies the protocol to make sure it can still provide equivalent security guarantees to TLS with the exception of order protection/non-replayability. DTLS was designed to be as similar to TLS as possible, so this document assumes that all properties from TLS are carried over except where specified.

3.2. Application-Specific Security Protocols

The following protocols provide application-specific security by protecting application payloads used for specific use-cases. Unlike the protocols above, these are not intended for generic application use.

3.2.1. Secure RTP

Secure RTP (SRTP) is a profile for RTP that provides confidentiality, message authentication, and replay protection for RTP data packets and RTP control protocol (RTCP) packets [RFC3711]. SRTP provides a record layer only, and requires a separate handshake protocol to provide key agreement and identity management.

The commonly used handshake protocol for SRTP is DTLS, in the form of DTLS-SRTP [RFC5764]. This is an extension to DTLS that negotiates the use of SRTP as the record layer, and describes how to export keys for use with SRTP.

ZRTP [RFC6189] is an alternative key agreement and identity management protocol for SRTP. ZRTP Key agreement is performed using a Diffie-Hellman key exchange that runs on the media path. This generates a shared secret that is then used to generate the master key and salt for SRTP.

3.3. Transport-Layer Security Protocols

The following security protocols provide protection for both application payloads and headers that are used for transport services.

3.3.1. IETF QUIC

QUIC is a new standards-track transport protocol that runs over UDP, loosely based on Google's original proprietary gQUIC protocol [I-D.ietf-quic-transport] (See Section 3.3.2 for more details). The QUIC transport layer itself provides support for data confidentiality and integrity. This requires keys to be derived with a separate handshake protocol. A mapping for QUIC of TLS 1.3 [I-D.ietf-quic-tls] has been specified to provide this handshake.

3.3.2. Google QUIC

Google QUIC (gQUIC) is a UDP-based multiplexed streaming protocol designed and deployed by Google following experience from deploying SPDY, the proprietary predecessor to HTTP/2. gQUIC was originally known as "QUIC": this document uses gQUIC to unambiguously distinguish it from the standards-track IETF QUIC. The proprietary technical forebear of IETF QUIC, gQUIC was originally designed with tightly-integrated security and application data transport protocols.

3.3.3. tcpcrypt

Tcpcrypt [RFC8548] is a lightweight extension to the TCP protocol for opportunistic encryption. Applications may use tcpcrypt's unique session ID for further application-level authentication. Absent this authentication, tcpcrypt is vulnerable to active attacks.

3.3.4. MinimaLT

MinimaLT [MinimaLT] is a UDP-based transport security protocol designed to offer confidentiality, mutual authentication, DoS prevention, and connection mobility. One major goal of the protocol is to leverage existing protocols to obtain server-side configuration information used to more quickly bootstrap a connection. MinimaLT uses a variant of TCP's congestion control algorithm.

3.3.5. CurveCP

CurveCP [CurveCP] is a UDP-based transport security that, unlike many other security protocols, is based entirely upon public key algorithms. CurveCP provides its own reliability for application data as part of its protocol.

3.4. Packet Security Protocols

The following protocols provide protection for IP packets. These are generally used as tunnels, such as for Virtual Private Networks (VPNs). Often, applications will not interact directly with these protocols. However, applications that implement tunnels will interact directly with these protocols.

3.4.1. IPsec

IKEv2 [RFC7296] and ESP [RFC4303] together form the modern IPsec protocol suite that encrypts and authenticates IP packets, either for creating tunnels (tunnel-mode) or for direct transport connections (transport-mode). This suite of protocols separates out the key generation protocol (IKEv2) from the transport encryption protocol (ESP). Each protocol can be used independently, but this document considers them together, since that is the most common pattern.

3.4.2. WireGuard

WireGuard [WireGuard] is an IP-layer protocol designed as an alternative to IPsec for certain use cases. It uses UDP to encapsulate IP datagrams between peers. Unlike most transport security protocols, which rely on Public Key Infrastructure (PKI) for peer authentication, WireGuard authenticates peers using pre-shared public keys delivered out-of-band, each of which is bound to one or more IP addresses. Moreover, as a protocol suited for VPNs, WireGuard offers no extensibility, negotiation, or cryptographic agility.

3.4.3. OpenVPN

OpenVPN [OpenVPN] is a commonly used protocol designed as an alternative to IPsec. A major goal of this protocol is to provide a VPN that is simple to configure and works over a variety of transports. OpenVPN encapsulates either IP packets or Ethernet frames within a secure tunnel and can run over either UDP or TCP. For key establishment, OpenVPN can either use TLS as a handshake protocol or use pre-shared keys.

4. Transport Dependencies

Across the different security protocols listed above, the primary dependency on transport protocols is the presentation of data: either an unbounded stream of bytes, or framed messages. Within protocols that rely on the transport for message framing, most are built to run over transports that inherently provide framing, like UDP, but some also define how their messages can be framed over byte-stream transports.

4.1. Reliable Byte-Stream Transports

The following protocols all depend upon running on a transport protocol that provides a reliable, in-order stream of bytes. This is typically TCP.

Application Payload Security Protocols:

* TLS

Transport-Layer Security Protocols:

* tcpcrypt

4.2. Unreliable Datagram Transports

The following protocols all depend on the transport protocol to provide message framing to encapsulate their data. These protocols are built to run using UDP, and thus do not have any requirement for reliability. Running these protocols over a protocol that does provide reliability will not break functionality, but may lead to multiple layers of reliability if the security protocol is encapsulating other transport protocol traffic.

Application Payload Security Protocols:

* DTLS

* ZRTP

* SRTP

Transport-Layer Security Protocols:

* QUIC

* MinimaLT

* CurveCP

Packet Security Protocols:

* IPsec

* WireGuard

* OpenVPN

4.2.1. Datagram Protocols with Defined Byte-Stream Mappings

Of the protocols listed above that depend on the transport for message framing, some do have well-defined mappings for sending their messages over byte-stream transports like TCP.

Application Payload Security Protocols:

* DTLS when used as a handshake protocol for SRTP [RFC7850]

* ZRTP [RFC6189]

* SRTP [RFC4571][RFC3711]

Packet Security Protocols:

* IPsec [RFC8229]

4.3. Transport-Specific Dependencies

One protocol surveyed, tcpcrypt, has an direct dependency on a feature in the transport that is needed for its functionality. Specifically, tcpcrypt is designed to run on top of TCP, and uses the TCP Encryption Negotiation Option (ENO) [RFC8547] to negotiate its protocol support.

QUIC, CurveCP, and MinimaLT provide both transport functionality and security functionality. They depend on running over a framed protocol like UDP, but they add their own layers of reliability and other transport services. Thus, an application that uses one of these protocols cannot decouple the security from transport functionality.

5. Application Interface

This section describes the interface exposed by the security protocols described above. We partition these interfaces into pre-connection (configuration), connection, and post-connection interfaces, following conventions in [I-D.ietf-taps-interface] and [I-D.ietf-taps-arch].

Note that not all protocols support each interface. The table in Section 5.4 summarizes which protocol exposes which of the interfaces. In the following sections, we provide abbreviations of the interface names to use in the summary table.

5.1. Pre-Connection Interfaces

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

* Identities and Private Keys (IPK): The application can provide its identity, credentials (e.g., certificates), and private keys, or mechanisms to access these, to the security protocol to use during handshakes.
  - TLS
  - DTLS
  - ZRTP
  - QUIC
  - MinimaLT
  - CurveCP
  - IPsec
  - WireGuard
  - OpenVPN
* Supported Algorithms (Key Exchange, Signatures, and Ciphersuites) (ALG): The application can choose the algorithms that are supported for key exchange, signatures, and ciphersuites.
  - TLS
  - DTLS
  - ZRTP
  - QUIC
  - tcpcrypt
  - MinimaLT
  - IPsec
  - OpenVPN
* Extensions (EXT): The application enables or configures extensions that are to be negotiated by the security protocol, such as Application-Layer Protocol Negotiation (ALPN) [RFC7301].
  - TLS

- DTLS
- QUIC

* Session Cache Management (CM): The application provides the ability to save and retrieve session state (such as tickets, keying material, and server parameters) that may be used to resume the security session.
  - TLS
  - DTLS
  - ZRTP
  - QUIC
  - tcpcrypt
  - MinimaLT
* Authentication Delegation (AD): The application provides access to a separate module that will provide authentication, using Extensible Authentication Protocol (EAP) [RFC3748] for example.
  - IPsec
  - tcpcrypt
* Pre-Shared Key Import (PSKI): Either the handshake protocol or the application directly can supply pre-shared keys for use in encrypting (and authenticating) communication with a peer.
  - TLS
  - DTLS
  - ZRTP
  - QUIC
  - tcpcrypt
  - MinimaLT
  - IPsec
  - WireGuard

- OpenVPN

5.2. Connection Interfaces

* Identity Validation (IV): During a handshake, the security protocol will conduct identity validation of the peer. This can offload validation or occur transparently to the application.

  - TLS
  - DTLS
  - ZRTP
  - QUIC
  - MinimaLT
  - CurveCP
  - IPsec
  - WireGuard
  - OpenVPN

* Source Address Validation (SAV): The handshake protocol may interact with the transport protocol or application to validate the address of the remote peer that has sent data. This involves sending a cookie exchange to avoid DoS attacks. (This list omits protocols which depend on TCP and therefore implicitly perform SAV.)

  - DTLS
  - QUIC
  - IPsec
  - WireGuard

5.3. Post-Connection Interfaces

* Connection Termination (CT): The security protocol may be instructed to tear down its connection and session information. This is needed by some protocols, e.g., to prevent application data truncation attacks in which an attacker terminates an underlying insecure connection-oriented protocol to terminate the session.
  - TLS
  - DTLS
  - ZRTP
  - QUIC
  - tcpcrypt
  - MinimaLT
  - IPsec
  - OpenVPN
* Key Update (KU): The handshake protocol may be instructed to update its keying material, either by the application directly or by the record protocol sending a key expiration event.
  - TLS
  - DTLS
  - QUIC
  - tcpcrypt
  - MinimaLT
  - IPsec
* Shared Secret Export (PSKE): The handshake protocol may provide an interface for producing shared secrets for application-specific uses.
  - TLS
  - DTLS

- tcpcrypt
- IPsec
- OpenVPN
- MinimaLT

* Key Expiration (KE): The record protocol can signal that its keys are expiring due to reaching a time-based deadline, or a use-based deadline (number of bytes that have been encrypted with the key). This interaction is often limited to signaling between the record layer and the handshake layer.
  - IPsec

* Mobility Events (ME): The record protocol can be signaled that it is being migrated to another transport or interface due to connection mobility, which may reset address and state validation and induce state changes such as use of a new Connection Identifier (CID).
  - DTLS (version 1.3 only [I-D.ietf-tls-dtls13])
  - QUIC
  - MinimaLT
  - CurveCP
  - IPsec [RFC4555]
  - WireGuard

5.4. Summary of Interfaces Exposed by Protocols

The following table summarizes which protocol exposes which interface.

Protocol	IPK	ALG	EXT	CM	AD	PSKI	IV	SAV	CT	KU	PSKE	KE	ME
TLS	x	x	x	x		x	x		x	x	x		
DTLS	x	x	x	x		x	x	x	x	x	x		x
ZRTP	x	x		x		x	x		x				
QUIC	x	x	x	x		x	x	x	x	x			x
tcpcrypt		x		x	x	x			x	x	x		
MinimaLT	x	x		x		x	x		x	x	x		x
CurveCP	x						x						x
IPsec	x	x			x	x	x	x	x	x	x	x	x
WireGuard	x					x	x	x					x
OpenVPN	x	x				x	x		x		x		

Table 1

x=Interface is exposed (blank)=Interface is not exposed

6. IANA Considerations

This document has no request to IANA.

7. Security Considerations

This document summarizes existing transport security protocols and their interfaces. It does not propose changes to or recommend usage of reference protocols. Moreover, no claims of security and privacy properties beyond those guaranteed by the protocols discussed are made. For example, metadata leakage via timing side channels and traffic analysis may compromise any protocol discussed in this survey. Applications using Security Interfaces should take such limitations into consideration when using a particular protocol implementation.

8. Privacy Considerations

Analysis of how features improve or degrade privacy is intentionally omitted from this survey. All security protocols surveyed generally improve privacy by using encryption to reduce information leakage. However, varying amounts of metadata remain in the clear across each protocol. For example, client and server certificates are sent in cleartext in TLS 1.2 [RFC5246], whereas they are encrypted in TLS 1.3 [RFC8446]. A survey of privacy features, or lack thereof, for various security protocols could be addressed in a separate document.

9. Acknowledgments

The authors would like to thank Bob Bradley, Frederic Jacobs, Mirja Kuehlewind, Yannick Sierra, Brian Trammell, and Magnus Westerlund for their input and feedback on this draft.

10. Informative References

[ALTS] Ghali, C., Stubblefield, A., Knapp, E., Li, J., Schmidt, B., and J. Boeuf, "Application Layer Transport Security", <https://cloud.google.com/security/encryption-in-transit/application-layer-transport-security/>.

[CurveCP] Bernstein, D.J., "CurveCP -- Usable security for the Internet", <http://curvecp.org>.

[I-D.ietf-quic-tls] Thomson, M. and S. Turner, "Using TLS to Secure QUIC", Work in Progress, Internet-Draft, draft-ietf-quic-tls-27, 21 February 2020, <http://www.ietf.org/internet-drafts/draft-ietf-quic-tls-27.txt>.

[I-D.ietf-quic-transport] Iyengar, J. and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport", Work in Progress, Internet-Draft, draft-ietf-quic-transport-27, 21 February 2020, <http://www.ietf.org/internet-drafts/draft-ietf-quic-transport-27.txt>.

[I-D.ietf-taps-arch] Pauly, T., Trammell, B., Brunstrom, A., Fairhurst, G., Perkins, C., Tiesel, P., and C. Wood, "An Architecture for Transport Services", Work in Progress, Internet-Draft, draft-ietf-taps-arch-07, 9 March 2020, <http://www.ietf.org/internet-drafts/draft-ietf-taps-arch-07.txt>.

[I-D.ietf-taps-interface]
Trammell, B., Welzl, M., Enghardt, T., Fairhurst, G., Kuehlewind, M., Perkins, C., Tiesel, P., Wood, C., and T. Pauly, "An Abstract Application Layer Interface to Transport Services", Work in Progress, Internet-Draft, draft-ietf-taps-interface-06, 9 March 2020, <http://www.ietf.org/internet-drafts/draft-ietf-taps-interface-06.txt>.

[I-D.ietf-tls-dtls13]
Rescorla, E., Tschofenig, H., and N. Modadugu, "The Datagram Transport Layer Security (DTLS) Protocol Version 1.3", Work in Progress, Internet-Draft, draft-ietf-tls-dtls13-37, 9 March 2020, <http://www.ietf.org/internet-drafts/draft-ietf-tls-dtls13-37.txt>.

[MinimaLT] Petullo, W.M., Zhang, X., Solworth, J.A., Bernstein, D.J., and T. Lange, "MinimaLT -- Minimal-latency Networking Through Better Security", <http://dl.acm.org/citation.cfm?id=2516737>.

[OpenVPN] "OpenVPN cryptographic layer", <https://openvpn.net/community-resources/openvpn-cryptographic-layer/>.

[RFC2385] Heffernan, A., "Protection of BGP Sessions via the TCP MD5 Signature Option", RFC 2385, DOI 10.17487/RFC2385, August 1998, <https://www.rfc-editor.org/info/rfc2385>.

[RFC2890] Dommety, G., "Key and Sequence Number Extensions to GRE", RFC 2890, DOI 10.17487/RFC2890, September 2000, <https://www.rfc-editor.org/info/rfc2890>.

[RFC3711] Baugher, M., McGrew, D., Naslund, M., Carrara, E., and K. Norrman, "The Secure Real-time Transport Protocol (SRTP)", RFC 3711, DOI 10.17487/RFC3711, March 2004, <https://www.rfc-editor.org/info/rfc3711>.

[RFC3748] Aboba, B., Blunk, L., Vollbrecht, J., Carlson, J., and H. Levkowetz, Ed., "Extensible Authentication Protocol (EAP)", RFC 3748, DOI 10.17487/RFC3748, June 2004, <https://www.rfc-editor.org/info/rfc3748>.

[RFC4253] Ylonen, T. and C. Lonvick, Ed., "The Secure Shell (SSH) Transport Layer Protocol", RFC 4253, DOI 10.17487/RFC4253, January 2006, <https://www.rfc-editor.org/info/rfc4253>.

[RFC4302] Kent, S., "IP Authentication Header", RFC 4302, DOI 10.17487/RFC4302, December 2005, <https://www.rfc-editor.org/info/rfc4302>.

[RFC4303] Kent, S., "IP Encapsulating Security Payload (ESP)", RFC 4303, DOI 10.17487/RFC4303, December 2005, <https://www.rfc-editor.org/info/rfc4303>.

[RFC4555] Eronen, P., "IKEv2 Mobility and Multihoming Protocol (MOBIKE)", RFC 4555, DOI 10.17487/RFC4555, June 2006, <https://www.rfc-editor.org/info/rfc4555>.

[RFC4571] Lazzaro, J., "Framing Real-time Transport Protocol (RTP) and RTP Control Protocol (RTCP) Packets over Connection-Oriented Transport", RFC 4571, DOI 10.17487/RFC4571, July 2006, <https://www.rfc-editor.org/info/rfc4571>.

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, DOI 10.17487/RFC5246, August 2008, <https://www.rfc-editor.org/info/rfc5246>.

[RFC5641] McGill, N. and C. Pignataro, "Layer 2 Tunneling Protocol Version 3 (L2TPv3) Extended Circuit Status Values", RFC 5641, DOI 10.17487/RFC5641, August 2009, <https://www.rfc-editor.org/info/rfc5641>.

[RFC5764] McGrew, D. and E. Rescorla, "Datagram Transport Layer Security (DTLS) Extension to Establish Keys for the Secure Real-time Transport Protocol (SRTP)", RFC 5764, DOI 10.17487/RFC5764, May 2010, <https://www.rfc-editor.org/info/rfc5764>.

[RFC5925] Touch, J., Mankin, A., and R. Bonica, "The TCP Authentication Option", RFC 5925, DOI 10.17487/RFC5925, June 2010, <https://www.rfc-editor.org/info/rfc5925>.

[RFC6189] Zimmermann, P., Johnston, A., Ed., and J. Callas, "ZRTP: Media Path Key Agreement for Unicast Secure RTP", RFC 6189, DOI 10.17487/RFC6189, April 2011, <https://www.rfc-editor.org/info/rfc6189>.

[RFC6347] Rescorla, E. and N. Modadugu, "Datagram Transport Layer Security Version 1.2", RFC 6347, DOI 10.17487/RFC6347, January 2012, <https://www.rfc-editor.org/info/rfc6347>.

[RFC7296] Kaufman, C., Hoffman, P., Nir, Y., Eronen, P., and T. Kivinen, "Internet Key Exchange Protocol Version 2 (IKEv2)", STD 79, RFC 7296, DOI 10.17487/RFC7296, October 2014, <https://www.rfc-editor.org/info/rfc7296>.

[RFC7301] Friedl, S., Popov, A., Langley, A., and E. Stephan, "Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension", RFC 7301, DOI 10.17487/RFC7301, July 2014, <https://www.rfc-editor.org/info/rfc7301>.

[RFC7850] Nandakumar, S., "Registering Values of the SDP 'proto' Field for Transporting RTP Media over TCP under Various RTP Profiles", RFC 7850, DOI 10.17487/RFC7850, April 2016, <https://www.rfc-editor.org/info/rfc7850>.

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

[RFC8229] Pauly, T., Touati, S., and R. Mantha, "TCP Encapsulation of IKE and IPsec Packets", RFC 8229, DOI 10.17487/RFC8229, August 2017, <https://www.rfc-editor.org/info/rfc8229>.

[RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <https://www.rfc-editor.org/info/rfc8446>.

[RFC8547] Bittau, A., Giffin, D., Handley, M., Mazieres, D., and E. Smith, "TCP-ENO: Encryption Negotiation Option", RFC 8547, DOI 10.17487/RFC8547, May 2019, <https://www.rfc-editor.org/info/rfc8547>.

[RFC8548] Bittau, A., Giffin, D., Handley, M., Mazieres, D., Slack, Q., and E. Smith, "Cryptographic Protection of TCP Streams (tcpcrypt)", RFC 8548, DOI 10.17487/RFC8548, May 2019, <https://www.rfc-editor.org/info/rfc8548>.

[WireGuard] Donenfeld, J.A., "WireGuard -- Next Generation Kernel Network Tunnel", <https://www.wireguard.com/papers/wireguard.pdf>.

Authors' Addresses

Theresa Enghardt
TU Berlin
Marchstr. 23
10587 Berlin
Germany

Email: ietf@tenghardt.net

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@csperkins.org

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

Email: krose@krose.org

Christopher A. Wood
Cloudflare
101 Townsend St
San Francisco,
United States of America

Email: caw@heapingbits.net