TAPS Working Group
Internet-Draft
Intended status: Standards Track
Expires: 6 May 2021

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
P. Tiesel
TU Berlin
C.A. Wood
Cloudflare

2 November 2020

An Architecture for Transport Services
draft-ietf-taps-arch-09

Abstract

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

Status of This Memo

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

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

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

This Internet-Draft will expire on 6 May 2021.

Table of Contents

1. Introduction 3
1.1. Background 4
1.2. Overview 4
1.3. Specification of Requirements 5
2. API Model 5
2.1. Event-Driven API 7
2.2. Data Transfer Using Messages 7
2.3. Flexibile Implementation 8
3. API and Implementation Requirements 9
3.1. Provide Common APIs for Common Features 9
3.2. Allow Access to Specialized Features 10
3.3. Select Equivalent Protocol Stacks 11
3.4. Maintain Interoperability 12
4. Transport Services Architecture and Concepts 12
4.1. Transport Services API Concepts 13
4.1.1. Endpoint Objects 15
4.1.2. Connections and Related Objects 15
4.1.3. Pre-Establishment 16
4.1.4. Establishment Actions 17
4.1.5. Data Transfer Objects and Actions 18
4.1.6. Event Handling 19
4.1.7. Termination Actions 20
4.1.8. Connection Groups 20
4.2. Transport Services Implementation Concepts 21
4.2.1. Candidate Gathering 22
4.2.2. Candidate Racing 22
4.2.3. Separating Connection Groups 23
5. IANA Considerations 23
6. Security Considerations 24
7. Acknowledgements 24

8. References 25
8.1. Normative References 25
8.2. Informative References 25
Authors' Addresses 27

1. Introduction

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

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

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

1.1. Background

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

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

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

1.2. Overview

This document describes the Transport Services architecture in three sections:

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

* Section 3 explains the fundamental requirements for a Transport Services API. 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 architecture diagram and defines the concepts that are used by both the API and implementation documents. The Preconnection allows applications to configure Connection Properties, and the Connection represents an object that can be used to send and receive Messages.

1.3. Specification of Requirements

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

2. API Model

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

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

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

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

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

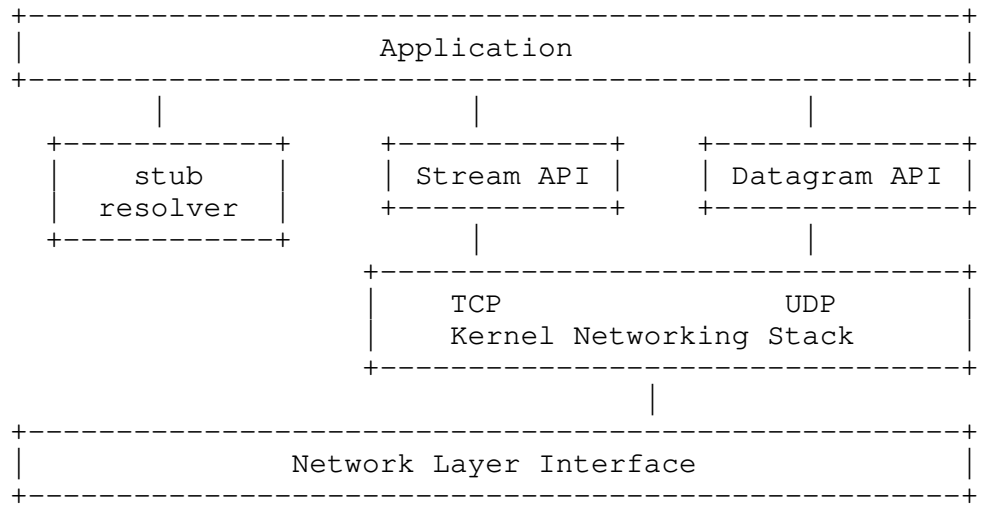

Figure 1: Socket API Model

The Transport Services architecture evolves this general model of interaction, aiming to both modernize the API surface presented to applications by the transport layer and enrich the capabilities of the Transport Services implementation. It combines interfaces for multiple interaction patterns into a unified whole. By combining name resolution with connection establishment and data transfer in a single API, it allows for more flexible implementations to provide path and transport protocol agility on the application's behalf.

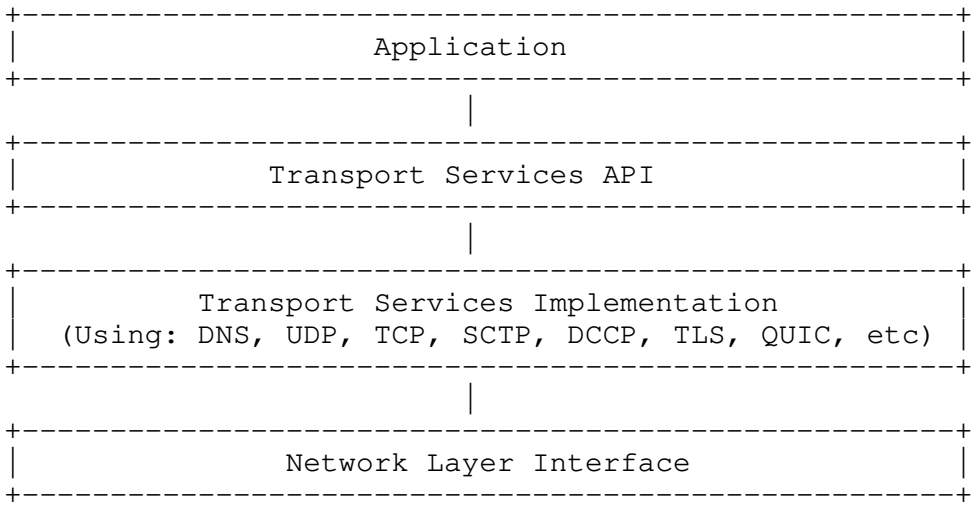

Figure 2: Transport Services API Model

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

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

2.1. Event-Driven API

Originally, sockets 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 sockets generally uses a try-and-fail model. If the application wants to read, but data has not yet been received from the peer, the call to read will fail. The application then waits and can try again later.

In contrast to sockets, all interaction with a Transport Services system is expected to be asynchronous, and use an event-driven model (see Section 4.1.6). For example, if the application wants to read, its call to read will not complete immediately, but will deliver an event containing the received data once it is available. Error handling is also asynchronous; a failure to send results in an asynchronous send error as an event.

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

Using asynchronous events allows for a 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 sockets represent network resources as file system objects that may be temporarily unavailable.

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

2.2. Data Transfer Using Messages

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

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

* the ability to associate deadlines with messages, for applications that care about timing;

* the ability to provide control of reliability, choosing which messages to retransmit when there is packet loss, and how best to make use of the data that arrived;

* the ability to manage dependencies between messages, when the Transport Services system could decide to not deliver a message, either following packet loss or because it has missed a deadline. In particular, this can avoid (re-)sending data that relies on a previous transmission that was never received.

* the ability to automatically assign messages and connections to underlying transport connections to utilize multi-streaming and pooled connections.

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

2.3. Flexibile Implementation

Sockets, for protocols like TCP, are generally limited to connecting to a single address over a single interface. They also present a single stream to the application. Software layers built upon sockets often propagate this limitation of a single-address single-stream model. The Transport Services architecture is designed to handle multiple candidate endpoints, protocols, and paths; and support multipath and multistreaming protocols.

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

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

3. API and Implementation Requirements

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

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

The normative requirements described here allow Transport Services APIs and Implementations to provide this functionality without causing incompatibility or introducing security vulnerabilities. The rest of this document describes the architecture non-normatively.

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 Transport Services API calls. As a baseline, any Transport Services API MUST allow access to the minimal set of features offered by transport protocols [RFC8923].

An application can specify constraints and preferences for the protocols, features, and network interfaces it will use via Properties. A Transport Services API SHOULD offer Properties that are common to multiple transport protocols, which enables the system to appropriately select between protocols that offer equivalent features. Similarly, a Transport Services API SHOULD offer Properties that are applicable to a variety of network layer interfaces and paths, which permits racing of different network paths without affecting the applications using the system. Each Property is expected to have a default value.

The default values for Properties SHOULD be 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 a Transport Services system interface are REQUIRED to be robust to the automated selection provided by the system, where the automated selection is constrained by the requirements and preferences expressed by the application.

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. A Transport Services system therefore SHOULD permit more specialized protocol features to be used.

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

Other specialized features, however, can be strictly required by an application and thus constrain the set of protocols that can be used. For example, if an application requires support for automatic handover or failover for a connection, only protocol stacks that

provide this feature are eligible to be used, e.g., protocol stacks that include a multipath protocol or a protocol that supports connection migration. A Transport Services API needs to allow applications to define such requirements and constrain the system's options. Since such options are not part of the core/common features, it will generally be simple for an application to modify its set of constraints and change the set of allowable protocol features without changing the core implementation.

3.3. Select Equivalent Protocol Stacks

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

The following two examples show non-equivalent Protocol Stacks:

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

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

The following example shows equivalent Protocol Stacks:

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

To ensure that security protocols are not incorrectly swapped, Transport Services systems MUST only select Protocol Stacks that meet application requirements ([RFC8922]). Systems SHOULD only race Protocol Stacks where the transport security protocols within the stacks are identical. Transport Services systems MUST NOT automatically fall back from secure protocols to insecure protocols,

or to weaker versions of secure protocols. A Transport Services system MAY allow applications to explicitly specify that fallback to a specific other version of a protocol is allowed, e.g., to allow fallback to TLS 1.2 if TLS 1.3 is not available.

3.4. Maintain Interoperability

It is important to note that neither the Transport Services API [I-D.ietf-taps-interface] nor the Implementation document [I-D.ietf-taps-impl] define new protocols or protocol capabilities that affect what is communicated across the network. Use of 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 system acting as a connection initiator is able to communicate with any existing system that implements the transport protocol(s) and all the required properties selected by the Transport Services system. Similarly, a Transport Services system acting as a listener can receive connections for any protocol that is supported by the system from existing initiators that implement the protocol, independent of whether the initiator uses a Transport Services system or not.

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

4. Transport Services Architecture and Concepts

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

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

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

2. System-implementation concepts, which are intended to be internally used when building systems that implement Transport Services.

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

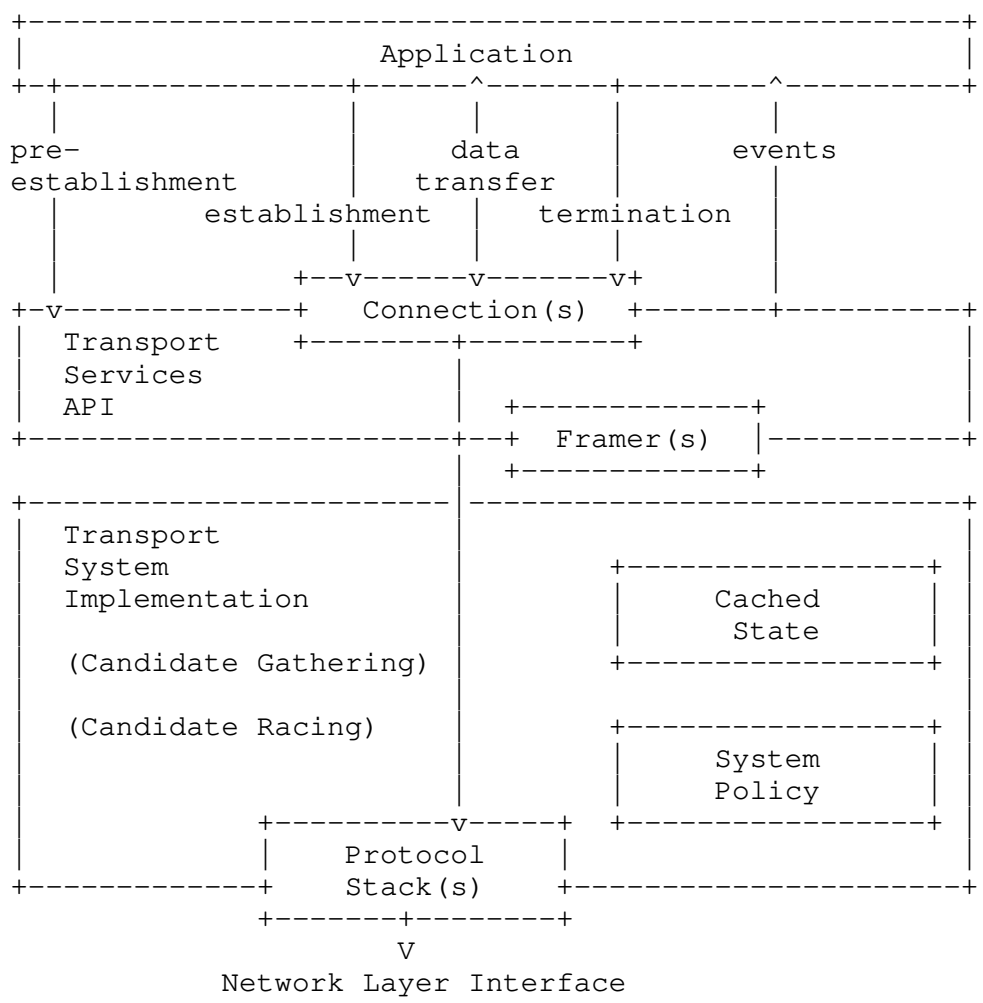

Figure 3: Concepts and Relationships in the Transport Services Architecture

4.1. Transport Services API Concepts

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

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

* Pre-Establishment (Section 4.1.3) encompasses the properties that an application can pass to describe its intent, requirements, prohibitions, and preferences for its networking operations.

These properties apply to multiple transport protocols, unless otherwise specified. Properties specified during Pre-Establishment can have a large impact on the rest of the interface: they modify how establishment occurs, they influence the expectations around data transfer, and they determine the set of events that will be supported.

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

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

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

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

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

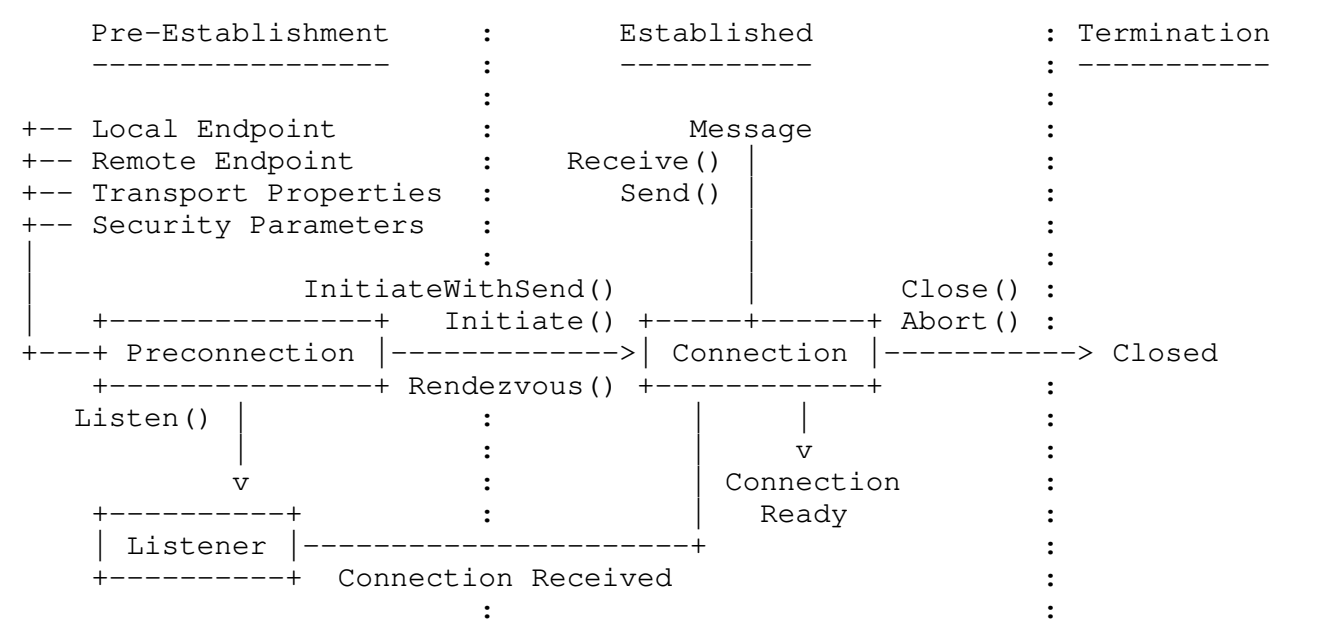

Figure 4: The lifetime of a Connection object

4.1.1. Endpoint Objects

* Endpoint: An Endpoint represents an identifier for one side of a transport connection. Endpoints can be Local Endpoints or Remote Endpoints, and respectively represent an identity that the application uses for the source or destination of a connection. An Endpoint can be specified at various levels of abstraction, and an Endpoint at a higher level of abstraction (such as a hostname) can be resolved to more concrete identities (such as IP addresses).

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

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

4.1.2. Connections and Related Objects

* Preconnection: A Preconnection object is a representation of a potential Connection. It has state that describes parameters of a Connection that might exist in the future: the Local Endpoint from which that Connection will be established, the Remote Endpoint

(Section 4.1.3) to which it will connect, and Transport Properties that influence the paths and protocols a Connection will use. A Preconnection can be fully specified such that it represents a single possible Connection, or it can be partially specified such that it represents a family of possible Connections. The Local Endpoint (Section 4.1.3) is required if the Preconnection is used to Listen for incoming Connections. The Local Endpoint is optional if it is used to Initiate Connections. The Remote Endpoint is required in the Preconnection that is used to Initiate Connections. The Remote Endpoint is optional if it is used to Listen for incoming Connections. The Local Endpoint and the Remote Endpoint 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 the Transport Services system. There are three kinds of Transport Properties:

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

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

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

* Connection: A Connection object represents one or more active transport protocol instances that can send and/or receive Messages between local and remote systems. It holds state pertaining to the underlying transport protocol instances and any ongoing data transfers. This represents, for example, an active Connection in a connection-oriented protocol such as TCP, or a fully-specified 5-tuple for a connectionless protocol such as UDP. It can also represent a pool of transport protocol instances, e.g., a set of TCP and QUIC connections to equivalent endpoints, or a stream of a multi-streaming transport protocol instance. Connections can be created from a Preconnection or by a Listener.

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

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

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

4.1.4. Establishment Actions

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

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

* Rendezvous: The action of establishing a peer-to-peer connection with a Remote Endpoint. It simultaneously attempts to initiate a connection to a Remote Endpoint while listening for an incoming connection from that endpoint. The process of identifying options for the connection, such as resolution of the Remote Endpoint, occurs in response to the Rendezvous call. As with Listeners, the set of local paths and endpoints is constrained by Selection Properties. If successful, the Rendezvous call returns a Connection object to represent the established peer-to-peer connection. The processes by which connections are initiated during a Rendezvous action will depend on the set of Local and Remote Endpoints configured on the Preconnection. For example, if the Local and Remote Endpoints are TCP host candidates, then a TCP simultaneous open [RFC0793] will be performed. However, if the set of Local Endpoints includes server reflexive candidates, such as those provided by STUN, a Rendezvous action will race candidates in the style of the ICE algorithm [RFC8445] to perform NAT binding discovery and initiate a peer-to-peer connection.

4.1.5. Data Transfer Objects and Actions

* Message: A Message object is a unit of data that can be represented as bytes that can be transferred between two systems over a transport connection. The bytes within a Message are assumed to be ordered. If an application does not care about the order in which a peer receives two distinct spans of bytes, those spans of bytes are considered independent Messages.

* Message Properties: Message Properties are used to specify details about Message transmission. They can be specified directly on individual Messages, or can be set on a Preconnection or Connection as defaults. These properties might only apply to how a Message is sent (such as how the transport will treat prioritization and reliability), but can also include properties that specific protocols encode and communicate to the Remote Endpoint. When receiving Messages, Message Properties can contain information about the received Message, such as metadata generated at the receiver and information signalled by the remote endpoint. For example, a Message can be marked with a Message Property indicating that it is the final message on a connection if the peer sent a TCP FIN.

* Send: The action to transmit a Message over a Connection to the remote system. 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 system, while the Message content itself will be delivered in an event (Section 4.1.6). The interface to Receive can include Message Properties specific to the Message that is to be delivered to the application.

* Framer: A Framer is a data translation layer that can be added to a Connection to define how application-layer Messages are transmitted over a transport protocol. This is particularly relevant for protocols that otherwise present unstructured streams, such as TCP.

4.1.6. Event Handling

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

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

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

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

* Message Received: Delivers received Message content to the application, based on a Receive action. This can include an error if the Receive action cannot be satisfied due to the Connection being closed.

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

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

4.1.7. 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 system 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. Indicating the end of a stream in the Transport Services architecture is possible 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. This is intended for immediate termination of a connection, without cleaning up state.

4.1.8. Connection Groups

A Connection Group is a set of Connections that share properties and caches. For multiplexing transport protocols, only Connections within the same Connection Group are allowed to be multiplexed together. An application can explicitly define Connection Groups to control caching boundaries, as discussed in Section 4.2.3.

4.2. Transport Services Implementation Concepts

This section defines the set of objects used internally to a system or library to implement the functionality needed to provide a transport service across a network, as required by the abstract interface.

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

* 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, of which there can be several. Candidate Protocol Stacks are identified during the gathering phase (Section 4.2.1) and are started during the racing phase (Section 4.2.2).

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

* Cached State: The state and history that the implementation keeps for each set of associated Endpoints that have been used previously. This can include DNS results, TLS session state, previous success and quality of transport protocols over certain paths, as well as other information.

4.2.1. Candidate Gathering

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

* 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 system.

4.2.2. Candidate Racing

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

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

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

* Remote Endpoint Racing: Remote Endpoint Racing is the act of attempting to establish, or scheduling attempts to establish, multiple Protocol Stacks that differ based on the specific representation of the Remote Endpoint, such as a particular IP address that was resolved from a DNS hostname.

4.2.3. Separating Connection Groups

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

There are several reasons, however, that an application might want to explicitly isolate some Connections. These reasons include:

* Privacy concerns about re-using cached protocol state that can lead to linkability. Sensitive state may include TLS session state [RFC8446] and HTTP cookies [RFC6265].

* Privacy concerns about allowing Connections to multiplex together, which can tell a Remote Endpoint that all of the Connections are coming from the same application (for example, when Connections are multiplexed HTTP/2 or QUIC streams).

* Performance concerns about Connections introducing head-of-line blocking due to multiplexing or needing to share state on a single thread.

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

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

5. IANA Considerations

RFC-EDITOR: Please remove this section before publication.

This document has no actions for IANA.

6. Security Considerations

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

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

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

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

7. Acknowledgements

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

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

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

Thanks to Theresa Enghardt, Max Franke, Mirja Kuehlewind, Jonathan Lennox, and Michael Welzl for the discussions and feedback that helped shape the architecture described here. Thanks as well to Stuart Cheshire, Josh Graessley, David Schinazi, and Eric Kinnear for their implementation and design efforts, including Happy Eyeballs, that heavily influenced this work.

8. References

8.1. Normative References

[I-D.ietf-taps-interface]
Trammell, B., Welzl, M., 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-09, 27 July 2020, <http://www.ietf.org/internet-drafts/draft-ietf-taps-interface-09.txt>.

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

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

8.2. Informative References

[I-D.ietf-taps-impl]
Brunstrom, A., Pauly, T., Enghardt, T., Grinnemo, K., Jones, T., Tiesel, P., Perkins, C., and M. Welzl, "Implementing Interfaces to Transport Services", Work in Progress, Internet-Draft, draft-ietf-taps-impl-07, 13 July 2020, <http://www.ietf.org/internet-drafts/draft-ietf-taps-impl-07.txt>.

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

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

[RFC4303] Kent, S., "IP Encapsulating Security Payload (ESP)", RFC 4303, DOI 10.17487/RFC4303, December 2005, <https://www.rfc-editor.org/info/rfc4303>.

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

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

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

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

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

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

[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>.

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

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

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

Philipp S. Tiesel
TU Berlin
Einsteinufer 25
10587 Berlin
Germany

Email: philipp@tiesel.net

Christopher A. Wood
Cloudflare
101 Townsend St
San Francisco,
United States of America

Email: caw@heapingbits.net