Workgroup: TAPS Working Group
Internet-Draft: draft-ietf-taps-arch-18
Published: 30 May 2023
Intended Status: Standards Track
Expires: 1 December 2023
Authors: T. Pauly, Ed. B. Trammell, Ed.
Apple Inc. Google Switzerland GmbH
A. Brunstrom G. Fairhurst
Karlstad University University of Aberdeen
C. Perkins
University of Glasgow

An Architecture for Transport Services

Abstract

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

Status of This Memo

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

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

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

This Internet-Draft will expire on 1 December 2023.

Copyright Notice

Copyright (c) 2023 IETF Trust and the persons identified as the document authors. All rights reserved.

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

Table of Contents

1. Introduction

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

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

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

1.1. Background

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

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

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

1.2. Overview

This document describes the Transport Services architecture in three sections:

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

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

* Section 4 presents a diagram showing the Transport Services architecture and defines the concepts that are used by both the API [I-D.ietf-taps-interface] and implementation guidelines [I-D.ietf-taps-impl]. The Preconnection allows applications to configure Connection Properties.

* Section 4 also presents how an abstract Connection is used to select a transport protocol instance such as TCP, UDP, or another transport. The Connection represents an object that can be used to send and receive messages.

1.3. Specification of Requirements

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

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 implementation keeps for each set of 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 Group: A set of Connections that shares properties and caches.

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

* Endpoint: An identifier for one side of a Connection (local or remote), such as a hostnames 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: 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: 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 System: The Transport Services implementation and the Transport Services API

2. API Model

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

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

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

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

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

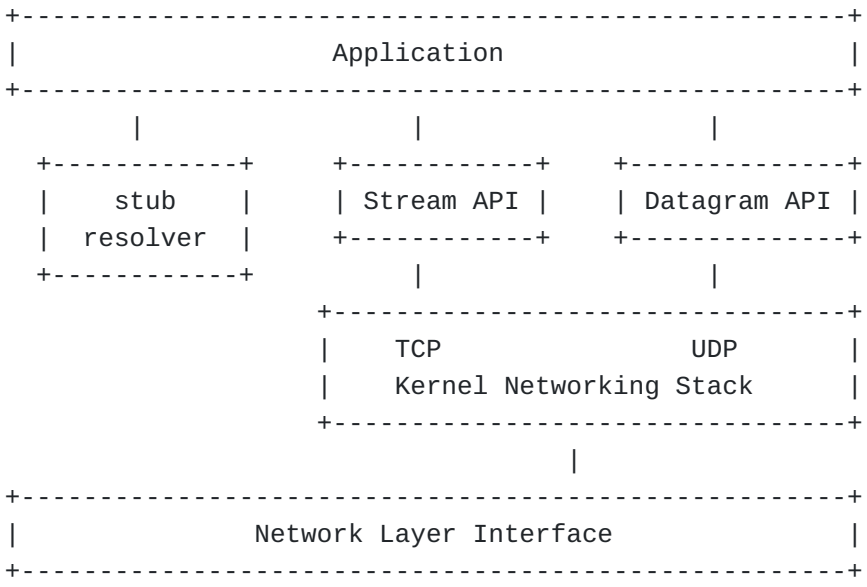

Figure 1: Socket API Model

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

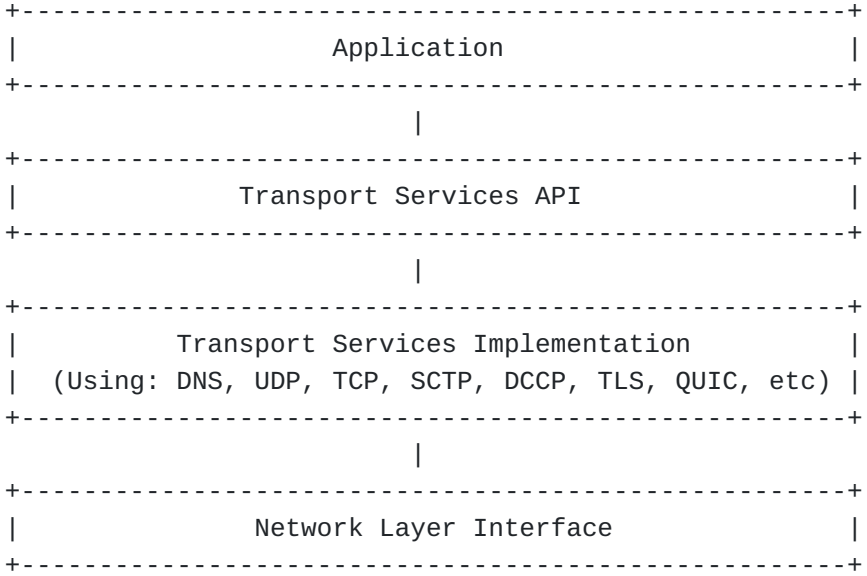

Figure 2: Transport Services API Model

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

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

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

2.1. Event-Driven API

Originally, the Socket API presented a blocking interface for establishing connections and transferring data. However, most modern applications interact with the network asynchronously. Emulation of an asynchronous interface using the Socket API generally uses a try-and-fail model. If the application wants to read, but data has not yet been received from the peer, the call to read will fail. The application then waits and can try again later.

In contrast to the Socket API, all interaction using the Transport Services API is expected to be asynchronous. The API is defined around an event-driven model (see Section 4.1.6) in order to model this asynchronous interaction, though other forms of asynchronous communication may be available to applications depending on the platform implementing the interface.

For example, an application first issues a call to receive new data from the connection. When delivered data becomes available, this data is delivered to the application using asynchronous events that contain the data. Error handling is also asynchronous; a failure to send data results in an asynchronous error event.

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

Using asynchronous events allows for a more natural interaction model when establishing connections and transferring data. Events in time more closely reflect the nature of interactions over networks, as opposed to how the Socket API 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 natively use a streaming abstraction, framers (Section 4.1.5) bridge the gap between the two abstractions.

2.3. Flexible Implementation

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

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

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

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

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

2.4. Coexistence

Note that while the Transport Service architecture is designed as an enhanced replacement for the Socket API, it need not replace it entirely on a system or platform; indeed, incremental deployability [RFC8170] requires coexistence. 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

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

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

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

3.1. Provide Common APIs for Common Features

Any functionality that is common across multiple transport protocols SHOULD be made accessible through a unified set of calls using the Transport Services API. As a baseline, any Transport Services API SHOULD allow access to the minimal set of features offered by transport protocols [RFC8923]. 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 implementation to appropriately select between protocols that offer equivalent features. Similarly, it is RECOMMENDED that the Properties offered by the Transport Services API are applicable to a variety of network layer interfaces and paths, which permits racing of different network paths without affecting the applications using the API. Each is expected to have a default value.

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

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

3.2. Allow Access to Specialized Features

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

A specialized feature could be needed by an application only when using a specific protocol, and not when using others. For example, if an application is using TCP, it could require control over the User Timeout Option for TCP [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 QUIC 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.

To ensure that security protocols are not incorrectly swapped, a Transport Services implementation MUST only select Protocol Stacks that meet application requirements ([RFC8922]). A Transport Services implementation SHOULD only race Protocol Stacks where the transport security protocols within the stacks are identical. A Transport Services implementation MUST NOT automatically fall back from secure protocols to insecure protocols, or to weaker versions of secure protocols. A Transport Services implementation MAY allow applications to explicitly specify 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.

3.4. Maintain Interoperability

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

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

4. Transport Services Architecture and Concepts

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

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

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

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

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

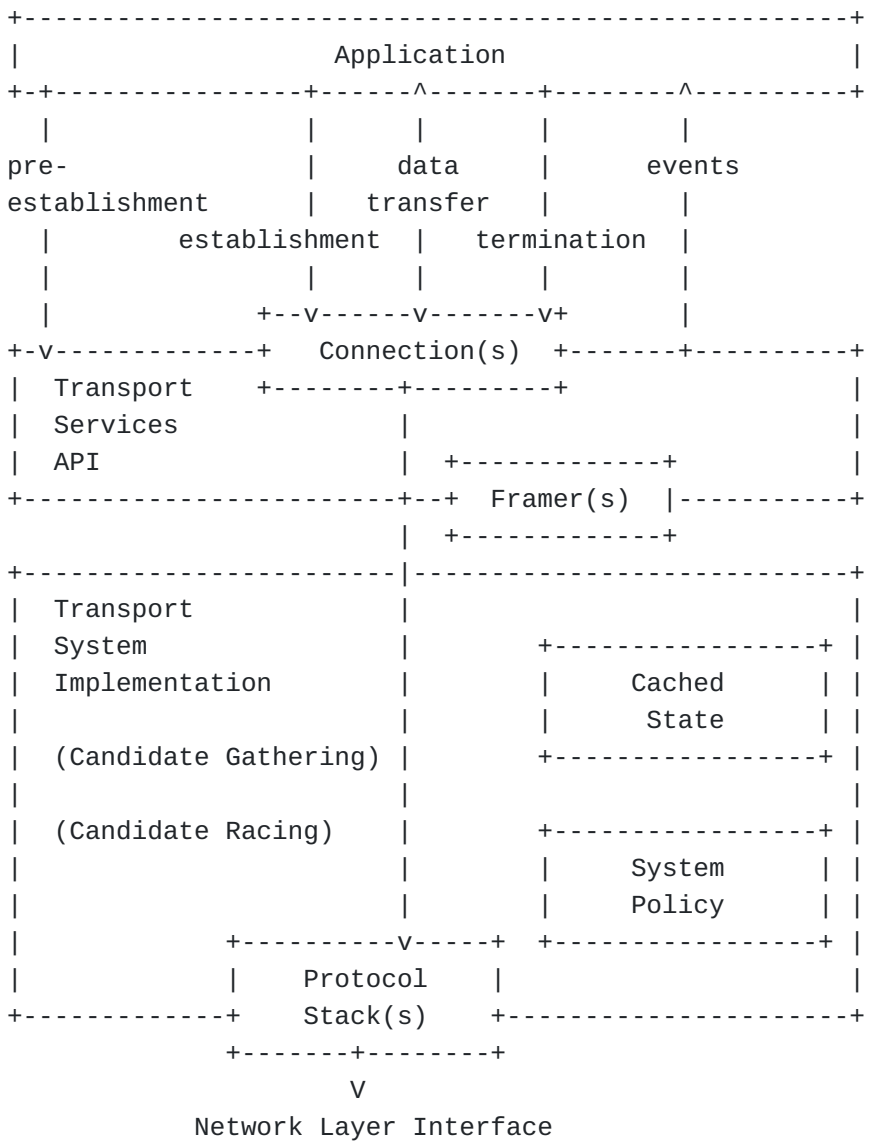

Figure 3: Concepts and Relationships in the Transport Services Architecture

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 implementation keeps for each set of associated endpoints that have previously been used.

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.

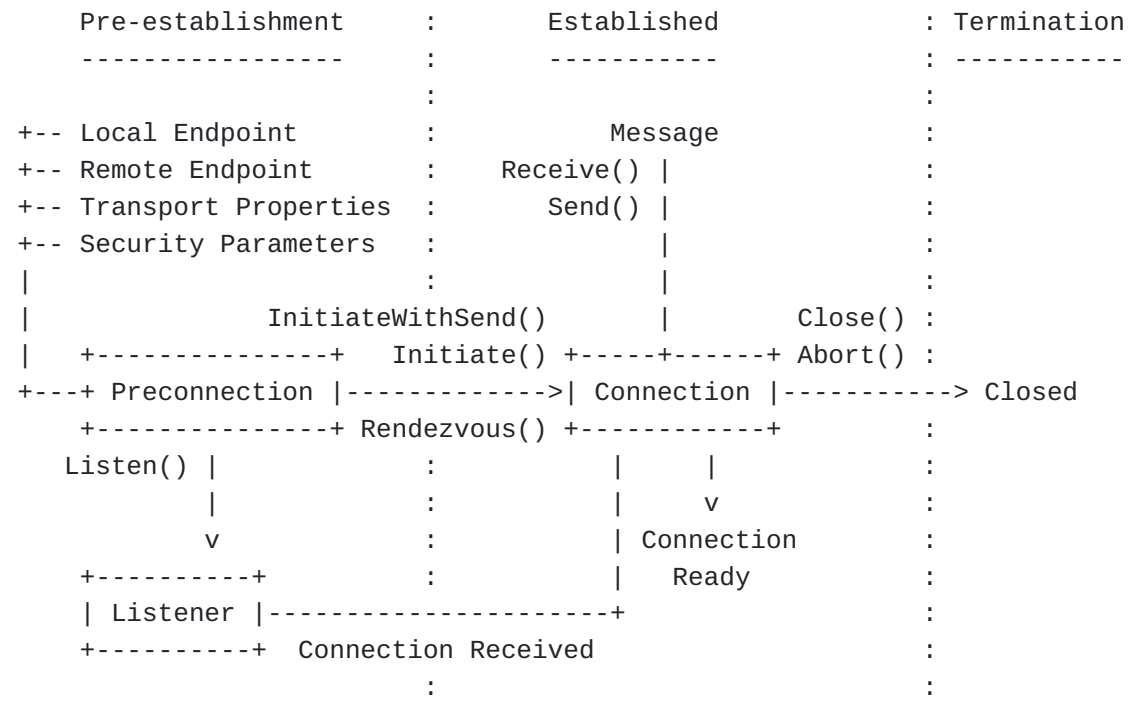

Figure 4: The lifetime of a Connection object

In this diagram, the lifetime of a Connection object is broken 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

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

can be resolved to more concrete identities (such as IP addresses). An endpoint may also represent a multicast group, in which case it selects a multicast transport for communication.

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

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

4.1.2. Connections and Related Objects

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

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

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

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

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

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

4.1.3. Pre-establishment

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

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

later. Changes made to Connection Properties after Connection establishment take effect on a best-effort basis.

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

4.1.4. Establishment Actions

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

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

* Rendezvous: The action of establishing a peer-to-peer connection with a Remote Endpoint. It simultaneously attempts to initiate a connection to a Remote Endpoint while listening for an incoming connection from that endpoint. The process of identifying options for the connection, such as resolution of the Remote Endpoint, occurs in response to the Rendezvous call. As with Listeners, the set of local paths and endpoints is constrained by Selection Properties. If successful, the Rendezvous call returns a Connection object to represent the established peer-to-peer connection. The processes by which connections are initiated during a Rendezvous action will depend on the set of Local and Remote Endpoints configured on the Preconnection. For example, if the Local and Remote Endpoints are TCP host candidates, then a TCP simultaneous open [RFC9293] will 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 packet. 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. This can include an error if the Receive action cannot be satisfied due to the Connection being closed.

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

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

4.1.7. Termination Actions

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

4.1.8. Connection Groups

A Connection Group is a set of Connections that shares 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 multiplexing transport protocols, 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.

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

While Connection Groups are managed by the Transport Services system, an application can define 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 concepts of the Transport Services architecture.

* Transport Service 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: This consists of the Transport Service implementation and the Transport Services API.

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

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

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

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

*Candidate Protocol Stack: One Protocol Stack that can be used by an application for a Connection, 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 the Transport Services architecture 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 Endpoints provided by the application, as well as the policies and heuristics of a Transport Services implementation.

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

4.2.2. Candidate Racing

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

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

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

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

4.2.3. Separating Connection Contexts

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

RFC-EDITOR: Please remove this section before publication.

This document has no actions for IANA.

6. Security and Privacy Considerations

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

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

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

have native concepts (e.g. "incognito mode") that align with this functionality.

Applications need to ensure that they use security APIs appropriately. In cases where applications use an interface to provide sensitive keying material, e.g., access to private keys or copies of pre-shared keys (PSKs), key use needs to be validated 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 described here. Particular thanks is also due to Philipp S. Tiesel and Christopher A. Wood, who were both co-authors of this architecture specification as it progressed through the TAPS working group. Thanks as well to Stuart Cheshire, Josh Graessley, David Schinazi, and Eric Kinnear for their implementation and design efforts, including Happy Eyeballs, that heavily influenced this work.

8. References

8.1. Normative References

[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-15, 9 March 2023, <https://datatracker.ietf.org/doc/html/draft-ietf-taps-impl-15>.

[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-20, 29 March 2023, <https://datatracker.ietf.org/doc/html/draft-ietf-taps-interface-20>.

[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>.

[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>.

[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