Stream: Internet Engineering Task Force (IETF)
RFC: 9621
Category: Standards Track
Published: January 2025
ISSN: 2070-1721
Authors: T. Pauly, Ed. *Apple Inc.*
B. Trammell, Ed. *Google Switzerland GmbH*
A. Brunstrom *Karlstad University*
G. Fairhurst *University of Aberdeen*
C. S. Perkins *University of Glasgow*

RFC 9621
Architecture and Requirements for Transport Services

Abstract

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

Status of This Memo

This is an Internet Standards Track document.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Further information on Internet Standards is available in Section 2 of RFC 7841.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at https://www.rfc-editor.org/info/rfc9621.

Copyright Notice

Table of Contents

1. Introduction 3
   1.1. Background 4
   1.2. Overview 5
   1.3. Specification of Requirements 5
   1.4. Glossary of Key Terms 5
2. API Model 8
   2.1. Event-Driven API 9
   2.2. Data Transfer Using Messages 10
   2.3. Flexible Implementation 11
   2.4. Coexistence 11
3. API and Implementation Requirements 12
   3.1. Provide Common APIs for Common Features 12
   3.2. Allow Access to Specialized Features 13
   3.3. Select Between Equivalent Protocol Stacks 14
   3.4. Maintain Interoperability 15
   3.5. Support Monitoring 15
4. Transport Services Architecture and Concepts 15
   4.1. Transport Services API Concepts 17
      4.1.1. Endpoint Objects 18
      4.1.2. Connections and Related Objects 19
      4.1.3. Preestablishment 20
      4.1.4. Establishment Actions 20
      4.1.5. Data Transfer Objects and Actions 21
      4.1.6. Event Handling 22

4.1.7. Termination Actions 22
4.1.8. Connection Groups 23
4.2. Transport Services Implementation 23
4.2.1. Candidate Gathering 24
4.2.2. Candidate Racing 25
4.2.3. Separating Connection Contexts 25
5. IANA Considerations 25
6. Security and Privacy Considerations 26
7. References 26
7.1. Normative References 26
7.2. Informative References 27
Acknowledgements 28
Authors' Addresses 29

1. Introduction

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

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

As applications adopt this interface, they will benefit from a wide set of transport features that can evolve over time and will ensure that the system providing the interface can optimize its behavior based on the application requirements and network conditions, without requiring changes to the applications. This flexibility enables faster deployment of new features and protocols.

This architecture can also support applications by offering racing mechanisms (attempting multiple IP addresses, protocols, or network paths in parallel), which otherwise need to be implemented in each application separately (see Section 4.2.2). Racing selects one or more candidates, each with equivalent Protocol Stacks that are used to identify an optimal combination of a transport protocol instance such as TCP, UDP, or another transport, together with configuration of parameters and interfaces. A Connection represents an object that, once established, can be used to send and receive Messages. A Connection can also be created from another Connection, by cloning, and then forms a part of a Connection Group whose Connections share Properties.

This document was developed in parallel with the specification of the Transport Services API [RFC9622] and implementation guidelines [RFC9623]. Although following the Transport Services Architecture does not require all APIs and implementations to be identical, a common minimal set of features represented in a consistent fashion will enable applications to be easily ported from one implementation of the Transport Services System to another.

1.1. Background

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

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

One of the key insights to come from identifying the minimal set of features provided by transport protocols [RFC8923] was that features either (1) require application interaction and guidance (referred to in that document as Functional or Optimizing Features) or (2) can be handled automatically by an implementation of the Transport Services System (referred to as Automatable Features). Among the identified Functional and Optimizing Features, some are common across all or nearly all transport protocols, while others present features that, if specified, would only be useful with a subset of protocols, but would not harm the functionality of other protocols. For example, some protocols can deliver messages more quickly for applications that do not require messages to arrive in the order in which they were sent. This functionality needs to be explicitly allowed by the application, since reordering messages would be undesirable in many cases.

1.2. Overview

The following sections describe the Transport Services System:

- Section 2 describes how the Transport Services API model differs from that of socket-based APIs. Specifically, it offers asynchronous event-driven interaction, the use of Messages for data transfer, and the flexibility to use different transport protocols and paths without requiring major changes to the application.
- Section 3 explains the fundamental requirements for a Transport Services System. These principles are intended to make sure that transport protocols can continue to be enhanced and evolve without requiring significant changes by application developers.
- Section 4 presents the Transport Services Implementation and defines the concepts that are used by the API [RFC9622] and described in the implementation guidelines [RFC9623]. This introduces the Preconnection, which allows applications to configure Connection Properties.

1.3. Specification of Requirements

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

1.4. Glossary of Key Terms

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

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

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

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

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

Client: The peer responsible for initiating a Connection.

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

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

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

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

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

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

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

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

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

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

Local Endpoint: The local Endpoint.

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

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

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

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

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

Peer: An Endpoint application party to a Connection.

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

Preference: A preference for prohibiting, avoiding, ignoring, preferring, or requiring a specific transport feature.

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

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

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

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

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

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

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

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

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

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 Candidate Protocol Stacks and race the candidates during establishment of a Connection.

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

Transport Property: A property of a transport protocol and the services it provides [RFC8095].

Transport Service: A set of transport features, not associated with any given framing protocol, that provides a complete service to an application.

Transport Services API: The abstract interface [RFC9622] to a Transport Services Implementation [RFC9623].

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

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

2. API Model

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

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

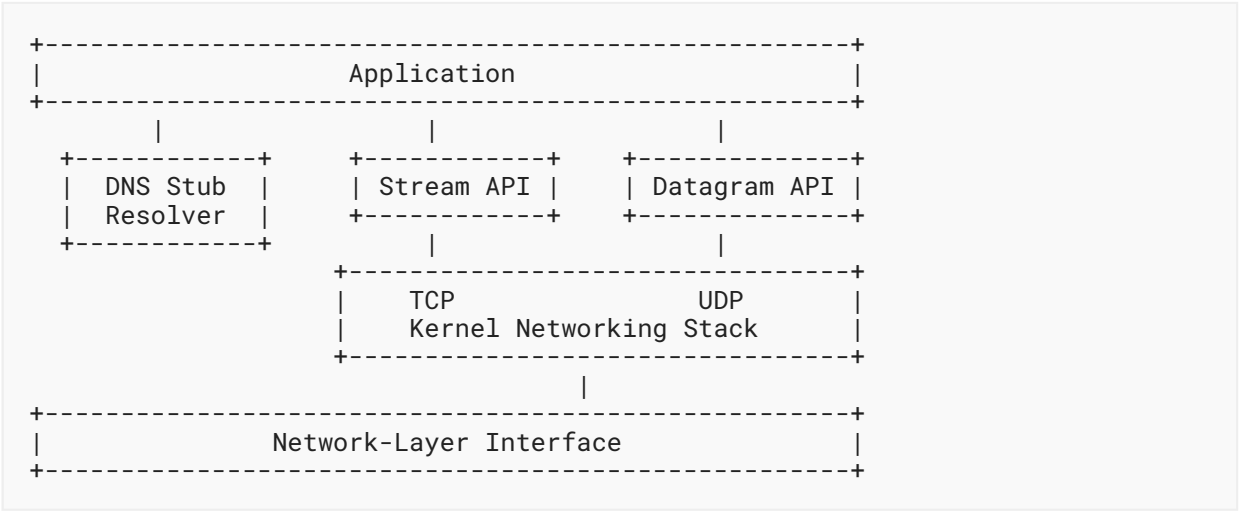

Figure 1: Socket API Model

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

The Transport Services API [RFC9622] defines the interface for an application to create Connections and transfer data. It combines interfaces for multiple interaction patterns into a unified whole (see Figure 2). This offers generic functions and also the protocol-specific mappings for TCP, UDP, UDP-Lite, and other protocol layers. These mappings are extensible. Future documents could define similar mappings for new layers and for other transport protocols, such as QUIC [RFC9000].

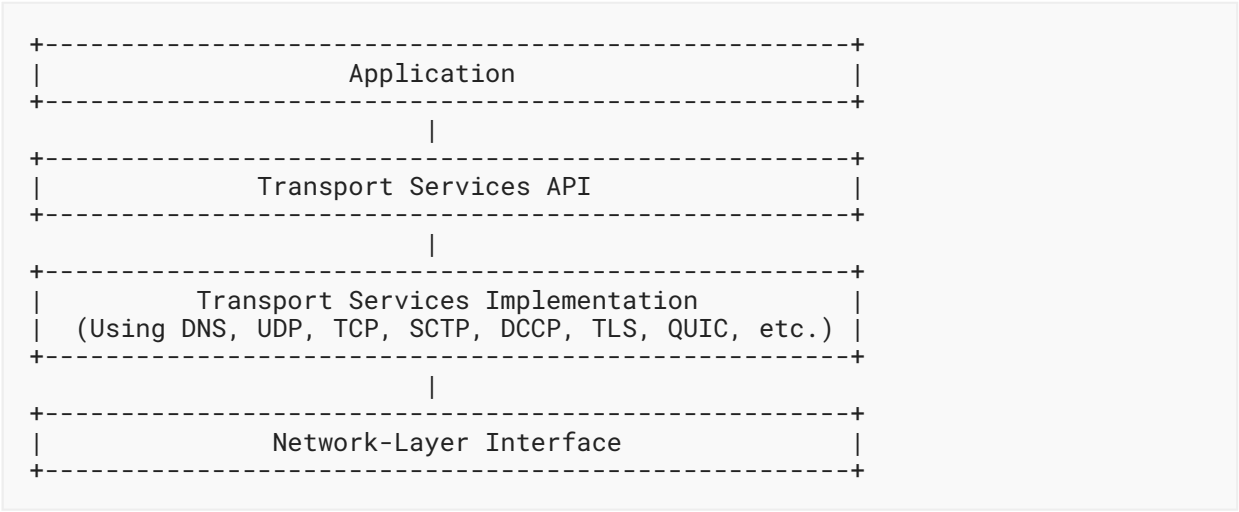

Figure 2: Transport Services API Model

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 [RFC9623] is the component of the Transport Services System that implements the transport-layer protocols and other functions needed to send and receive data. It is responsible for mapping the API to a specific available transport Protocol Stack and managing the available network interfaces and paths.

There are key differences between the architecture of the Transport Services System and the architecture of the Socket API. The API of the Transport Services System:

- is asynchronous and event-driven;
- uses Messages for representing data transfer to applications;
- describes how a Transport Services Implementation can resolve Endpoint Identifiers to use multiple IP addresses, multiple protocols, and multiple paths and to provide multiple application streams.

2.1. Event-Driven API

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

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

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

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

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

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

2.2. Data Transfer Using Messages

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

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

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

Allowing applications to interact with Messages is backward-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 protocols, while simplifying the application's role in parsing data. For protocols that inherently use a streaming abstraction, Framers (Section 4.1.5) bridge the gap between the two abstractions.

2.3. Flexible Implementation

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

- handle multiple candidate endpoints, protocols, and paths;
- support candidate protocol racing to select the most optimal stack in each situation;
- support multipath and multistreaming protocols;
- 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 Sections 4.2.1 and 4.2.2). This requires applications to specify identifiers for the Local and Remote Endpoint that are at a higher level than IP addresses, such as a hostname or URL. These identifiers are used by a Transport Services Implementation for resolution, path selection, and racing. An implementation can further implement fallback mechanisms if connection establishment for one protocol fails or performance is determined 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) is cached in the Transport Services Implementation. Applications have control over whether this information is used for a specific establishment, in order to allow trade-offs between efficiency and linkability.

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

2.4. Coexistence

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

3. API and Implementation Requirements

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

There are several ways the Transport Services System can offer flexibility to an application. It can:

- provide access to transport protocols and protocol features;
- use these protocols across multiple paths that could have different performance and functional characteristics;
- 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 the Transport Services Implementations 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 the following:

- 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, which can be set on individual Messages.

It is **RECOMMENDED** that the Transport Services API offer Properties that are common to multiple transport protocols. This enables a Transport Services System to appropriately select between protocols that offer equivalent features. Similarly, it is **RECOMMENDED** that the Properties offered by the Transport Services API be applicable to a variety of network-layer interfaces and paths, to permit 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 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 the Transport Services API need to be designed to be robust to the automated selection provided by the Transport Services System. This automated selection is constrained by the 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.

Some specialized features 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]. Such features 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 for using the User Timeout Option, communication would not fail when a protocol such as UDP is selected.

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

To control these specialized features, the application can declare its preference: whether the presence of a specific feature is prohibited, should be avoided, can be ignored, is preferred, or is required in the preestablishment 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 successful establishment.

3.3. Select Between Equivalent Protocol Stacks

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

The following two examples show non-equivalent Protocol Stacks:

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

The following example shows equivalent Protocol Stacks:

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

A Transport Services Implementation can race different security protocols, e.g., if the System Policy is explicitly configured to consider them equivalent. A Transport Services Implementation **SHOULD** only race Protocol Stacks where the transport security protocols within the stacks are identical. To ensure that security protocols are not incorrectly swapped, a Transport Services Implementation **MUST** only select Protocol Stacks that meet application requirements [RFC8922]. A Transport Services Implementation **MUST NOT** automatically fall back from secure protocols to insecure protocols or fall back 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 if TLS 1.3 is not available.

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

3.4. Maintain Interoperability

It is important to note that neither the Transport Services API [RFC9622] nor the guidelines for implementation of the Transport Services System [RFC9623] 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 use 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, independently of whether or not the initiator uses the Transport Services System.

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

3.5. Support Monitoring

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

4. Transport Services Architecture and Concepts

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

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

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

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

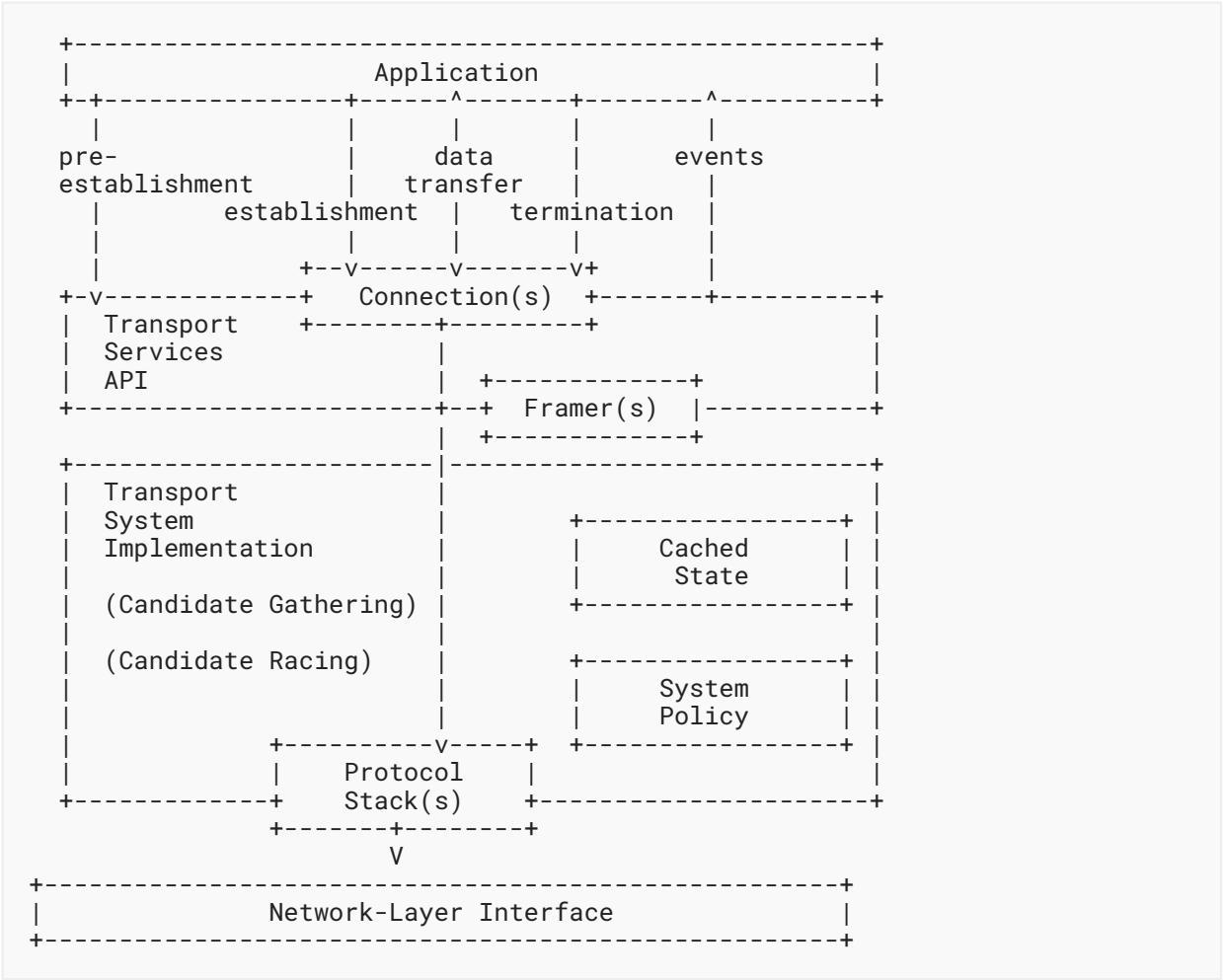

Figure 3: Concepts and Relationships in the Architecture of the Transport Services System

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

The System Policy provides input from an operating system or other global preferences that can constrain or influence how an implementation will gather Candidate Paths and Protocol Stacks and race the candidates when establishing a Connection. As the details of System Policy configuration and enforcement are largely dependent on the platform and implementation and do not affect application-level interoperability, the Transport Services API [RFC9622] does not specify an interface for reading or writing System Policy.

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

4.1. Transport Services API Concepts

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

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

- Preestablishment (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 preestablishment can have a large impact on the rest of the interface: they modify how establishment occurs, influence the expectations around data transfer, and 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 listen for incoming connections), while others are optional (e.g., setting Connection and Message Properties in preestablishment) or have been omitted for brevity and simplicity.

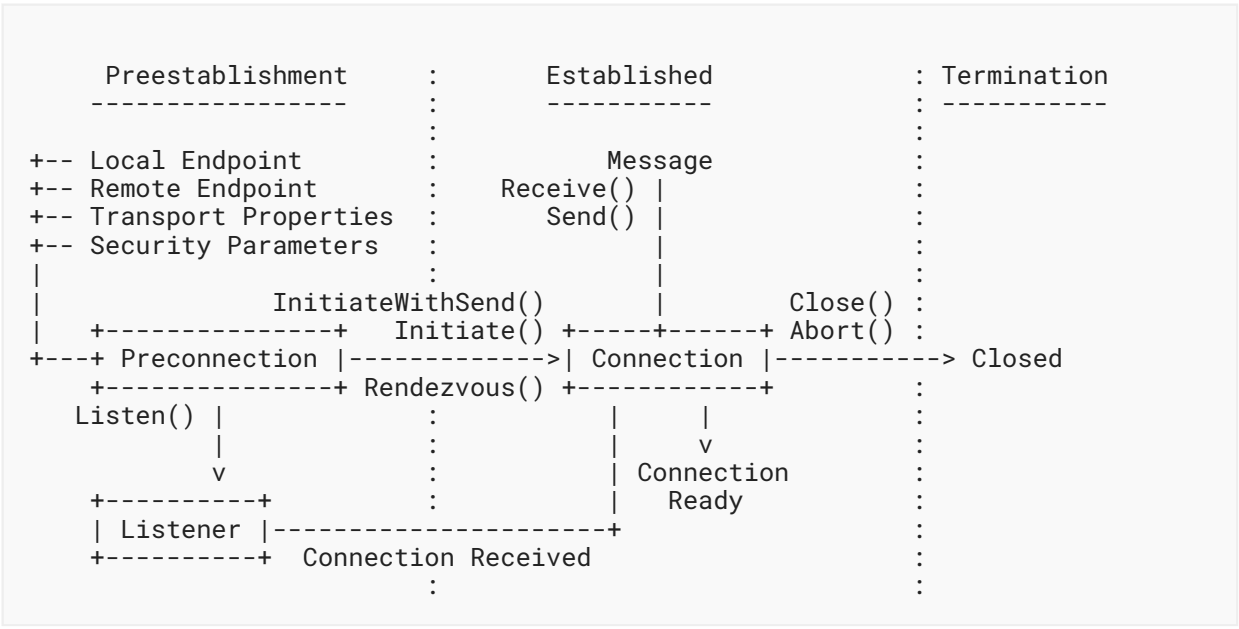

Figure 4: The Lifetime of a Connection Object

In this diagram, the lifetime of a Connection object is divided into three phases: preestablishment, the Established state, and termination of a Connection.

Preestablishment is based around a Preconnection object containing 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 can receive state updates.

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

4.1.1. Endpoint Objects

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

Remote Endpoint Identifier:

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

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

4.1.2. Connections and Related Objects

Connection: A Connection object represents one or more active transport protocol instances that can send and/or receive Messages between Local and Remote Endpoints. It is an abstraction that represents the communication. The Connection object holds state pertaining to the underlying transport protocol instances and any ongoing data transfers. For example, an active Connection can represent a connection-oriented protocol such as TCP, or it 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 multistreaming transport protocol instance. Connections can be created from a Preconnection or by a Listener.

Preconnection: A Preconnection object is a representation of a Connection that has not yet been established. It has state that describes parameters of the Connection: the Local Endpoint Identifier from which that Connection will be established, the Remote Endpoint Identifier 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 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 is optional if it is used to `Initiate` a Connection. The Remote Endpoint Identifier is required in a Preconnection that is used to `Initiate` a Connection but is optional if it is used to `Listen` for incoming Connections. The Local Endpoint Identifier and the Remote Endpoint Identifier are both required if a peer-to-peer `Rendezvous` is to occur based on the Preconnection.

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

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

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

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

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

4.1.3. Preestablishment

Selection Properties: Selection Properties consist of the Properties that an application can set to influence the selection of paths between the Local and Remote Endpoints, influence the selection of transport protocols, or 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 support for TCP Fast Open.

Connection Properties: 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 be used. In general, these Properties do not explicitly determine the selection of paths or protocols but can be used by an implementation during Connection establishment. Connection Properties are specified on a Preconnection prior to Connection establishment and can be modified on the Connection later. Changes made to Connection Properties after Connection establishment take effect on a best-effort basis.

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

4.1.4. Establishment Actions

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

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

Rendezvous: The action of establishing a peer-to-peer connection with a Remote Endpoint. It simultaneously attempts to initiate a connection to a Remote Endpoint while listening for an incoming connection from that Endpoint. The process of identifying options for the connection, such as resolution of the Remote Endpoint Identifier(s), occurs in response to calling `Rendezvous`. As with Listeners, the set of local paths and endpoints is constrained by Selection Properties. If successful, calling `Rendezvous` generates and asynchronously returns a Connection object to represent the established peer-to-peer connection. The processes by which connections are initiated during a `Rendezvous` action will depend on the set of Local and Remote Endpoints configured on the Preconnection. For example, if the Local and Remote Endpoints are TCP host candidates, then a TCP simultaneous open [RFC9293] might be performed. However, if the set of Local Endpoints includes server-reflexive candidates, such as those provided by STUN (Session Traversal Utilities for NAT) [RFC8489], a `Rendezvous` action will race candidates in the style of the ICE (Interactive Connectivity Establishment) algorithm [RFC8445] to perform NAT binding discovery and initiate a peer-to-peer connection.

4.1.5. Data Transfer Objects and Actions

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

Message Properties: Message Properties are used to specify details about Message transmission. They can be specified directly on individual Messages or can be set on a Preconnection or Connection as defaults. These Properties might only apply to how a Message is sent (such as how the transport will treat prioritization and reliability) but can also include Properties that specific protocols encode and communicate to the Remote Endpoint. When receiving Messages, Message Properties can contain information about the received Message, such as metadata generated at the receiver and information signaled 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 `Send` action transmits 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` action is delivered back to the sending application in an event (Section 4.1.6).

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

Framer: A Framer is a data translation layer that can be added to a Connection. Framers allow extending a Connection's Protocol Stack to define how to encapsulate or encode outbound Messages and how to decapsulate or decode inbound data into Messages. In this way, Message boundaries can be preserved when using a Connection object, even with a protocol that otherwise presents unstructured streams, such as TCP. This is designed based on the fact that many of the current application protocols evolved over TCP, which does not provide Message boundary preservation, and since many of these protocols require Message boundaries to function, each application-layer protocol has defined its own framing. For example, when an HTTP application sends and receives HTTP Messages over a byte-stream transport, it must parse the boundaries of HTTP Messages from the stream of bytes.

4.1.6. Event Handling

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

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

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

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

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

Message Sent: Notifies the application of the status of its `Send` action. This might indicate a failure if the Message cannot be sent or might indicate 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 or is no longer willing to receive data. The protocol should signal this state to the Remote Endpoint if the transport protocol permits it. (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 that the Transport Services System should not attempt to deliver any outstanding data and that it should immediately close and drop the connection. This is intended for immediate, usually abnormal, termination of a connection.

4.1.8. Connection Groups

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

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

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

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

4.2. Transport Services Implementation

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

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

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

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

Protocol Stack: A set of protocol instances (including relevant application, security, transport, or Internet protocols) that are used together to establish connectivity or send and receive Messages. A single stack can be simple (e.g., one application stream carried over TCP running over IP) or complex (e.g,. multiple application streams carried over a multipath transport protocol using multiple subflows over IP).

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 Candidate Protocol Stacks (Section 4.2.1) and race the candidates during establishment (Section 4.2.2). Specific aspects of the System Policy apply to either all Connections or only certain Connections, depending on the runtime context and Properties of the Connection.

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

4.2.1. Candidate Gathering

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

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

4.2.2. Candidate Racing

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

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

Path Racing: Path Racing is the act of attempting to establish, or scheduling attempts to establish, multiple Protocol Stacks that differ based on a selection from the available paths. Since different paths will have distinct configurations (see [RFC7556]) for local addresses and DNS servers, attempts across different paths will perform separate DNS resolution steps, which can lead to further racing of the resolved Remote Endpoint Identifiers.

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

4.2.3. Separating Connection Contexts

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

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

- reusing cached protocol state, as this 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.
- 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 an HTTP/2 or QUIC stream.

5. IANA Considerations

This document has no IANA actions.

6. Security and Privacy Considerations

The Transport Services System does not recommend the 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 trade-offs between efficiency (through racing) and privacy (via information that might leak from the cache toward an on-path observer). Some applications have features (e.g., "incognito mode") that align with this functionality.

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

A Transport Services System **MUST NOT** automatically fall back from secure protocols to insecure protocols or fall back 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. References

7.1. 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/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>.

7.2. Informative References

[POSIX] "IEEE/Open Group Standard for Information Technology - Portable Operating System Interface (POSIX(TM)) Base Specifications, Issue 8", IEEE Std 1003.1-2024, DOI 10.1109/IEEESTD.2024.10555529, 2024, <https://ieeexplore.ieee.org/document/10555529>.

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

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

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

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

[RFC8489] Petit-Huguenin, M., Salgueiro, G., Rosenberg, J., Wing, D., Mahy, R., and P. Matthews, "Session Traversal Utilities for NAT (STUN)", RFC 8489, DOI 10.17487/RFC8489, February 2020, <https://www.rfc-editor.org/info/rfc8489>.

[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>.

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

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

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

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

[RFC9622] Trammell, B., Ed., Welzl, M., Ed., Enghardt, R., Fairhurst, G., Kühlewind, M., Perkins, C. S., Tiesel, P., and T. Pauly, "An Abstract Application Programming Interface (API) for Transport Services", RFC 9622, DOI 10.17487/RFC9622, January 2025, <https://www.rfc-editor.org/info/rfc9622>.

[RFC9623] Brunstrom, A., Ed., Pauly, T., Ed., Enghardt, R., Tiesel, P., and M. Welzl, "Implementing Interfaces to Transport Services", RFC 9623, DOI 10.17487/RFC9623, January 2025, <https://www.rfc-editor.org/info/rfc9623>.

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 from the DFG - German Research Foundation: Gottfried Wilhelm Leibniz-Preis 2011 (FKZ FE 570/4-1).
- the UK Engineering and Physical Sciences Research Council under grant EP/R04144X/1.

Thanks to Reese Enghardt, Max Franke, Mirja Kühlewind, Jonathan Lennox, and Michael Welzl for the discussions and feedback that helped shape the architecture of the system described here. Particular thanks are also due to Philipp S. Tiesel and Christopher A. Wood, who were both coauthors of this specification as it progressed through the Transport Services (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.

Authors' Addresses

Tommy Pauly (EDITOR)
Apple Inc.
One Apple Park Way
Cupertino, CA 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
United Kingdom
Email: gorry@erg.abdn.ac.uk
URI: https://erg.abdn.ac.uk/

Colin S. Perkins
University of Glasgow
School of Computing Science
Glasgow G12 8QQ
United Kingdom
Email: csp@csperkins.org