

TAPS Working Group
Internet-Draft
Intended status: Standards Track
Expires: September 12, 2019

T. Pauly, Ed.
Apple Inc.
B. Trammell, Ed.
Google
A. Brunstrom
Karlstad University
G. Fairhurst
University of Aberdeen
C. Perkins
University of Glasgow
P. Tiesel
TU Berlin
C. Wood
Apple Inc.
March 11, 2019

An Architecture for Transport Services
draft-ietf-taps-arch-03

Abstract

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

Status of This Memo

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

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

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

This Internet-Draft will expire on September 12, 2019.

Copyright Notice

Copyright (c) 2019 IETF Trust and the persons identified as the document authors. All rights reserved.

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

Table of Contents

1.	Introduction	3
1.1.	Overview	3
1.2.	Event-Driven API	5
1.3.	Data Transfer Using Messages	5
1.4.	Flexible Implementation	6
2.	Background	7
2.1.	Specification of Requirements	7
3.	Design Principles	8
3.1.	Common APIs for Common Features	8
3.2.	Access to Specialized Features	8
3.3.	Scope for API and Implementation Definitions	9
4.	Transport Services Architecture and Concepts	10
4.1.	Transport Services API Concepts	11
4.1.1.	Basic Objects	13
4.1.2.	Pre-Establishment	14
4.1.3.	Establishment Actions	15
4.1.4.	Data Transfer Objects and Actions	16
4.1.5.	Event Handling	16
4.1.6.	Termination Actions	17
4.2.	Transport System Implementation Concepts	17
4.2.1.	Candidate Gathering	19
4.2.2.	Candidate Racing	19
4.2.3.	Protocol Stack Equivalence	19
4.2.4.	Separating Connection Groups	21
5.	IANA Considerations	21
6.	Security Considerations	21
7.	Acknowledgements	22
8.	Informative References	23
	Authors' Addresses	24

1. Introduction

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

The goal of the Transport Services architecture is to provide a common, flexible, and reusable interface for transport protocols. As applications adopt this interface, they will benefit from a wide set of transport features that can evolve over time, and ensure that the system providing the interface can optimize its behavior based on the application requirements and network conditions, without requiring changes to the applications. This flexibility does not only enable faster deployment of new feature and protocols, but it can also support applications with racing and fallback mechanisms which otherwise need to be implemented in each application separately.

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

1.1. Overview

The model of using sockets for networking can be represented as follows: applications create connections and transfer data using the socket API, which provides the interface to the implementations of UDP and TCP (typically implemented in the system's kernel), which in turn send data over the available network layer interfaces.

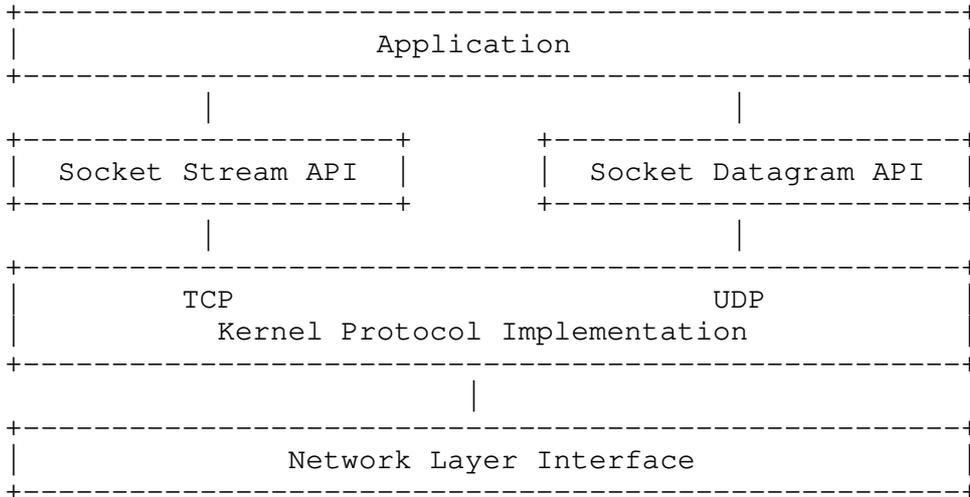


Figure 1: socket() API Model

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

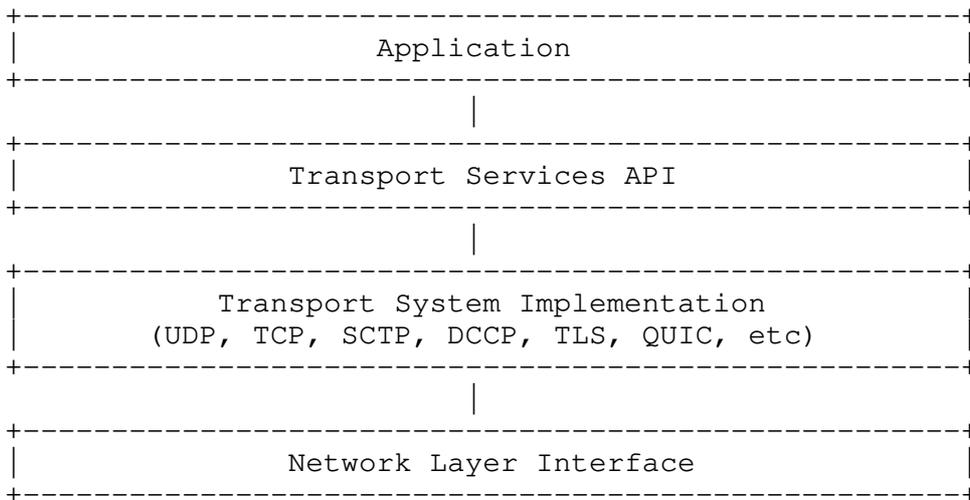


Figure 2: Transport Services API Model

The Transport Services API [I-D.ietf-taps-interface] defines the mechanism for an application to create network connections and transfer data. The Implementation [I-D.ietf-taps-impl] is

responsible for mapping the API into the various available transport protocols and managing the available network interfaces and paths.

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

1.2. Event-Driven API

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

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

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

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

1.3. Data Transfer Using Messages

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

The Transport Services API represents data as messages, so that it more closely matches the way applications use the network. Messages

seamlessly work with transport protocols that support datagrams or records, but can also be used over a stream by defining an application-layer framer Section 4.1.4. When framing protocols are placed on top of unstructured streams, the messages used in the API represent the framed messages within the stream. In the absence of a framer, protocols that deal only in byte streams, such as TCP, represent their data in each direction as a single, long message.

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

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

Allowing applications to interact with messages is backwards-compatible with existings protocols and APIs, as it does not change the wire format of any protocol. Instead, it gives the protocol stack additional information to allow it to make better use of modern transport services, while simplifying the application's role in parsing data.

1.4. Flexibile Implementation

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

Transport Services implementations are meant to be flexible at connection establishment time, considering many different options and trying to select the most optimal combinations (Section 4.2.1 and Section 4.2.2). This requires applications to provide higher-level endpoints than IP addresses, such as hostnames and URLs, which are used by a Transport Services implementation for resolution, path selection, and racing.

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. Background

The Transport Services architecture is based on the survey of Services Provided by IETF Transport Protocols and Congestion Control Mechanisms [RFC8095], and the distilled minimal set of the features offered by transport protocols [I-D.ietf-taps-minset]. These documents identified common features and patterns across all transport protocols developed thus far in the IETF.

Since transport security is an increasingly relevant aspect of using transport protocols on the Internet, this architecture also considers the impact of transport security protocols on the feature-set exposed by transport services [I-D.ietf-taps-transport-security].

One of the key insights to come from identifying the minimal set of features provided by transport protocols [I-D.ietf-taps-minset] was that features either require application interaction and guidance (referred to as Functional Features), or else can be handled automatically by a system implementing Transport Services (referred to as Automatable Features). Among the Functional 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.

2.1. 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.

3. Design Principles

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

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

The following considerations were used in the design of this architecture.

3.1. Common APIs for Common Features

Functionality that is common across multiple transport protocols SHOULD be accessible through a unified set of API calls. An application ought to be able to implement logic for its basic use of transport networking (establishing the transport, and sending and receiving data) once, and expect that implementation to continue to function as the transports change.

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

3.2. Access to Specialized Features

There are applications that will need to control fine-grained details of transport protocols to optimize their behavior and ensure compatibility with remote systems. A Transport Services system therefore SHOULD also permit more specialized protocol features to be used. The interface for these specialized options should be exposed differently from the common options to ensure flexibility.

A specialized feature could be required by an application only when using a specific protocol, and not when using others. For example, if an application is using UDP, it could require control over the checksum or fragmentation behavior for UDP; if it used a protocol to

frame its data over a byte stream like TCP, it would not need these options. In such cases, the API ought to expose the features in such a way that they take effect when a particular protocol is selected, but do not imply that only that protocol could be used. For example, if the API allows an application to specify a preference for constrained checksum usage, communication would not fail when a protocol such as TCP is selected, which uses a checksum covering the entire payload.

Other specialized features, however, could be strictly required by an application and thus constrain the set of protocols that can be used. For example, if an application requires encryption of its transport data, only protocol stacks that include some transport security protocol are eligible to be used. A Transport Services API MUST allow applications to define such requirements and constrain the system's options. Since such options are not part of the core/common features, it will generally be simple for an application to modify its set of constraints and change the set of allowable protocol features without changing the core implementation.

3.3. Scope for API and Implementation Definitions

The Transport Services API is envisioned as the abstract model for a family of APIs that share a common way to expose transport features and encourage flexibility. The abstract API definition [I-D.ietf-taps-interface] describes this interface and is aimed at application developers.

Implementations that provide the Transport Services API [I-D.ietf-taps-impl] will vary due to system-specific support and the needs of the deployment scenario. It is expected that all implementations of Transport Services will offer the entire mandatory API, but that some features will not be functional in certain implementations. All implementations are REQUIRED to offer sufficient APIs to use the distilled minimal set of features offered by transport protocols [I-D.ietf-taps-minset], including API support for TCP and UDP transport, but it is possible that some very constrained devices might not have, for example, a full TCP implementation beneath the API.

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

It is important to note that neither the Transport Services API [I-D.ietf-taps-interface] nor the Implementation document [I-D.ietf-taps-impl] defines new protocols that require any changes to a remote system. The Transport Services system **MUST** be deployable on one side only.

4. Transport Services Architecture and Concepts

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

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

1. API concepts, which are intended to be exposed to applications; and
2. System-implementation concepts, which are intended to be internally used when building systems that implement Transport Services.

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

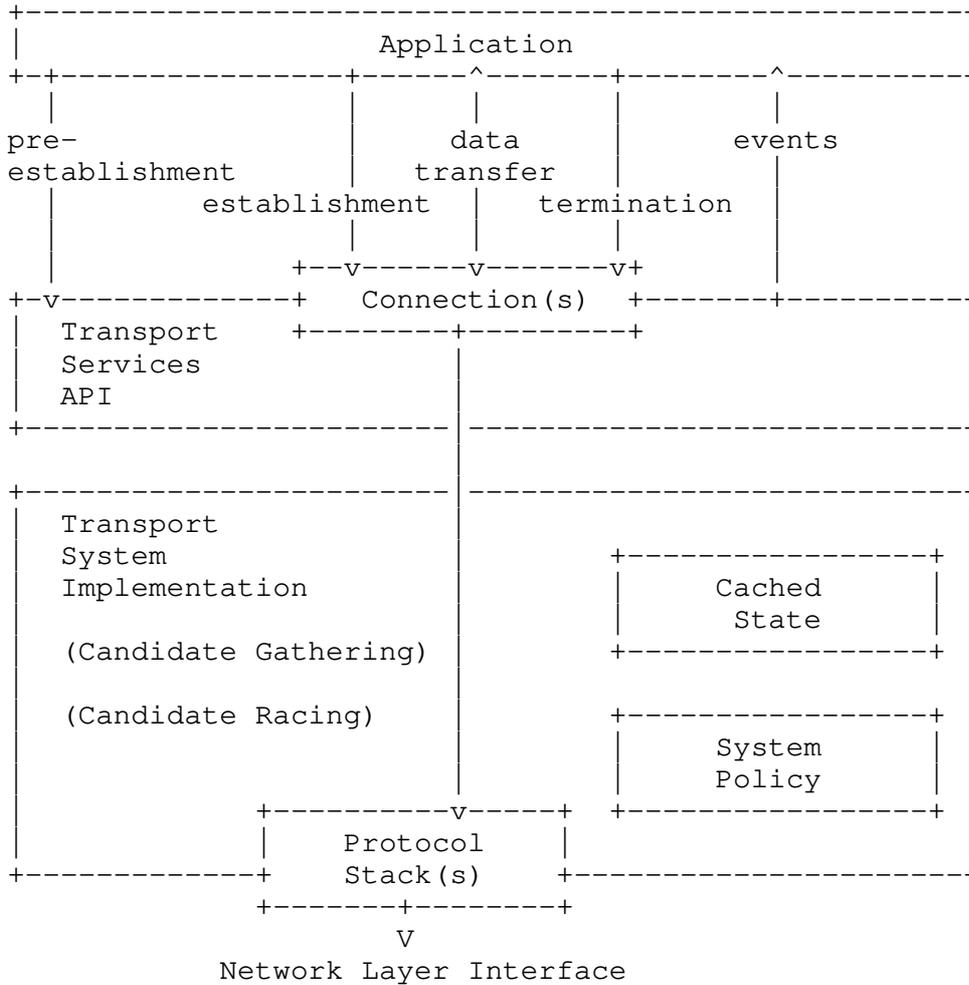


Figure 3: Concepts and Relationships in the Transport Services Architecture

4.1. Transport Services API Concepts

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

Beyond the basic objects, there are several high-level groups of actions that any Transport Services API implementing this specification MUST provide:

- o Pre-Establishment (Section 4.1.2) encompasses the properties that an application can pass to describe its intent, requirements,

prohibitions, and preferences for its networking operations. For any system that provides generic Transport Services, these properties SHOULD be defined to apply to multiple transport protocols. Properties specified during Pre-Establishment can have a large impact on the rest of the interface: they modify how establishment occurs, they influence the expectations around data transfer, and they determine the set of events that will be supported.

- o Establishment (Section 4.1.3) focuses on the actions that an application takes on the basic objects to prepare for data transfer.
- o Data Transfer (Section 4.1.4) consists of how an application represents the data to be sent and received, the functions required to send and receive that data, and how the application is notified of the status of its data transfer.
- o Event Handling (Section 4.1.5) defines the set of properties about which an application can receive notifications during the lifetime of transport objects. Events MAY also provide opportunities for the application to interact with the underlying transport by querying state or updating maintenance options.
- o Termination (Section 4.1.6) focuses on the methods by which data transmission is stopped, and state is torn down in the transport.

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

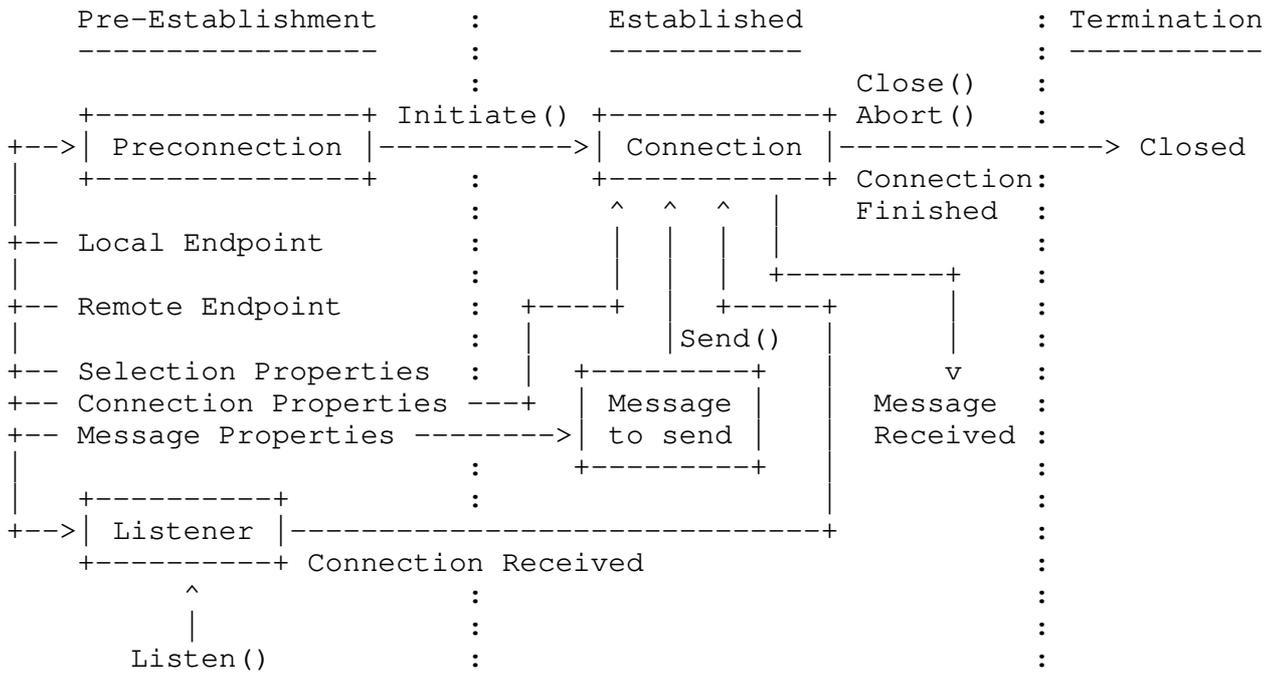


Figure 4: The lifetime of a connection

4.1.1.1. Basic Objects

- o **Preconnection:** A Preconnection object is a representation of a potential connection. It has state that describes parameters of a Connection that might exist in the future: the Local Endpoint from which that Connection will be established, the Remote Endpoint (Section 4.1.2) to which it will connect, and Selection Properties (Section 4.1.2) that influence the paths and protocols a Connection will use. A Preconnection can be fully specified and represent a single possible Connection, or it can be partially specified such that it represents a family of possible Connections. The Local Endpoint (Section 4.1.2) MUST be specified if the Preconnection is used to Listen for incoming connections. The Local Endpoint is OPTIONAL if it is used to Initiate connections. The Remote Endpoint MUST be specified in the Preconnection is used to Initiate connections. The Remote Endpoint is OPTIONAL if it is used to Listen for incoming connections. The Local Endpoint and the Remote Endpoint MUST both be specified if a peer-to-peer Rendezvous is to occur based on the Preconnection.

* **Transport Properties:** Transport Properties can be specified as part of a Preconnection to allow the application to configure the Transport System and express their requirements,

prohibitions, and preferences. There are three kinds of Transport Properties: Selection Properties (Section 4.1.2), Connection Properties (Section 4.1.2), and Message Properties (Section 4.1.4). Message Properties can also be specified during data transfer to affect specific Messages.

- o **Connection:** A Connection object represents an active transport protocol instance that can send and/or receive Messages between local and remote systems. It holds state pertaining to the underlying transport protocol instance and any ongoing data transfer. This represents, for example, an active connection in a connection-oriented protocol such as TCP, or a fully-specified 5-tuple for a connectionless protocol such as UDP.
- o **Listener:** A Listener object accepts incoming transport protocol connections from remote systems and generates corresponding Connection objects. It is created from a Preconnection object that specifies the type of incoming connections it will accept.

4.1.2. Pre-Establishment

- o **Endpoint:** An Endpoint represents an identifier for one side of a transport connection. Endpoints can be Local Endpoints or Remote Endpoints, and respectively represent an identity that the application uses for the source or destination of a connection. An Endpoint can be specified at various levels, and an Endpoint with wider scope (such as a hostname) can be resolved to more concrete identities (such as IP addresses).
- o **Remote Endpoint:** The Remote Endpoint represents the application's identifier for a peer that can participate in a transport connection. For example, the combination of a DNS name for the peer and a service name/port.
- o **Local Endpoint:** The Local Endpoint represents the application's identifier for itself that it uses for transport connections. For example, a local IP address and port.
- o **Selection Properties:** The Selection Properties consist of the options that an application can set to influence the selection of paths between the local and remote systems, to influence the selection of transport protocols, or to configure the behavior of generic transport protocol features. These options can take the form of requirements, prohibitions, or preferences. Examples of options that influence path selection include the interface type (such as a Wi-Fi Ethernet connection, or a Cellular LTE connection), requirements around the Maximum Transmission Unit (MTU) or path MTU (PMTU), or preferences for throughput and

latency properties. Examples of options that influence protocol selection and configuration of transport protocol features include reliability, service class, multipath support, and fast open support.

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

4.1.3. Establishment Actions

- o **Initiate:** The primary action that an application can take to create a Connection to a Remote Endpoint, and prepare any required local or remote state to be able to send and/or receive Messages. For some protocols, this will initiate a client-to-server style handshake; for other protocols, this will just establish local state. The process of identifying options for connecting, such as resolution of the Remote Endpoint, occurs in response to the Initiate call.
- o **Listen:** The action of marking a Listener as willing to accept incoming Connections. The Listener will then create Connection objects as incoming connections are accepted (Section 4.1.5).
- o **Rendezvous:** The action of establishing a peer-to-peer connection with a Remote Endpoint. It simultaneously attempts to initiate a connection to a Remote Endpoint whilst listening for an incoming connection from that endpoint. This corresponds, for example, to a TCP simultaneous open [RFC0793]. The process of identifying options for the connection, such as resolution of the Remote Endpoint, occurs during the Rendezvous call. If successful, the rendezvous call returns a Connection object to represent the established peer-to-peer connection.

4.1.4. Data Transfer Objects and Actions

- o **Message:** A Message object is a unit of data that can be represented as bytes that can be transferred between two systems over a transport connection. The bytes within a Message are assumed to be ordered within the Message. If an application does not care about the order in which a peer receives two distinct spans of bytes, those spans of bytes are considered independent Messages. If a received Message is incomplete or corrupted, it might or might not be usable by certain applications. Boundaries of a Message might or might not be understood or transmitted by transport protocols. Specifically, what one application considers to be two Messages sent on a stream-based transport can be treated as a single Message by the application on the other side.
- o **Message Properties:** Message Properties can be used to annotate specific Messages. These properties can specify how the transport will send the Message (for prioritization and reliability), along with any per-protocol properties to send with the Message. Message Properties MAY be set on a Preconnection to define defaults properties for sending. When receiving Messages, Message Properties can contain per-protocol properties.
- o **Send:** The action to transmit a Message or partial Message over a Connection to the remote system. The interface to Send MAY include Message Properties specific to how the Message's content is to be sent. Status of the Send operation can be delivered back to the application in an event (Section 4.1.5).
- o **Receive:** An action that indicates that the application is ready to asynchronously accept a Message over a Connection from a remote system, while the Message content itself will be delivered in an event (Section 4.1.5). The interface to Receive MAY include Message Properties specific to the Message that is to be delivered to the application.
- o **Framer:** A Framer is a data translation layer that can be added to a Connection to define how application-level Messages are transmitted over a transport protocol. This is particularly relevant for protocols that otherwise present unstructured streams, such as TCP.

4.1.5. Event Handling

This list of events that can be delivered to an application is not exhaustive, but gives the top-level categories of events. The API MAY expand this list.

- o **Connection Ready:** Signals to an application that a given Connection is ready to send and/or receive Messages. If the Connection relies on handshakes to establish state between peers, then it is assumed that these steps have been taken.
- o **Connection Finished:** Signals to an application that a given Connection is no longer usable for sending or receiving Messages. The event SHOULD deliver a reason or error to the application that describes the nature of the termination.
- o **Connection Received:** Signals to an application that a given Listener has passively received a Connection.
- o **Message Received:** Delivers received Message content to the application, based on a Receive action. This MAY include an error if the Receive action cannot be satisfied due to the Connection being closed.
- o **Message Sent:** Notifies the application of the status of its Send action. This might indicate a failure if the Message cannot be sent, or an indication that Message has been processed by the protocol stack.
- o **Path Properties Changed:** Notifies the application that some property of the Connection has changed that might influence how and where data is sent and/or received.

4.1.6. Termination Actions

- o **Close:** The action an application takes on a Connection to indicate that it no longer intends to send data, is no longer willing to receive data, and that the protocol SHOULD signal this state to the remote system if the transport protocol allows this.
- o **Abort:** The action the application takes on a Connection to indicate a Close and also indicate that the transport system SHOULD NOT attempt to deliver any outstanding data.

4.2. Transport System Implementation Concepts

The Transport System Implementation Concepts define the set of objects used internally to a system or library to implement the functionality needed to provide a transport service across a network, as required by the abstract interface.

- o **Connection Group:** A set of Connections that share properties and caches. For multiplexing transport protocols, only Connections within the same Connection Group are allowed be multiplexed

together. Applications can use their explicitly defined Connection Groups to control caching boundaries, as discussed in Section 4.2.4.

- o Path: Represents an available set of properties that a local system can use to communicate with a remote system, such as routes, addresses, and physical and virtual network interfaces.
- o Protocol Instance: A single instance of one protocol, including any state it has necessary to establish connectivity or send and receive Messages.
- o Protocol Stack: A set of Protocol Instances (including relevant application, security, transport, or Internet protocols) that are used together to establish connectivity or send and receive Messages. A single stack can be simple (a single transport protocol instance over IP), or complex (multiple application protocol streams going through a single security and transport protocol, over IP; or, a multi-path transport protocol over multiple transport sub-flows).
- o Candidate Path: One path that is available to an application and conforms to the Selection Properties and System Policy. Candidate Paths are identified during the gathering phase (Section 4.2.1) and can be used during the racing phase (Section 4.2.2).
- o Candidate Protocol Stack: One protocol stack that can be used by an application for a connection, of which there can be several. Candidate Protocol Stacks are identified during the gathering phase (Section 4.2.1) and are started during the racing phase (Section 4.2.2).
- o System Policy: Represents the input from an operating system or other global preferences that can constrain or influence how an implementation will gather candidate paths and protocol stacks (Section 4.2.1) and race the candidates during establishment (Section 4.2.2). Specific aspects of the System Policy either apply to all Connections or only certain ones, depending on the runtime context and properties of the Connection.
- o Cached State: The state and history that the implementation keeps for each set of associated Endpoints that have been used previously. This can include DNS results, TLS session state, previous success and quality of transport protocols over certain paths.

4.2.1. Candidate Gathering

- o **Path Selection:** Path Selection represents the act of choosing one or more paths that are available to use based on the Selection Properties provided by the application, the policies and heuristics of a Transport Services system.
- o **Protocol Selection:** Protocol Selection represents the act of choosing one or more sets of protocol options that are available to use based on the Transport Properties provided by the application, and a Transport Services system's policies and heuristics.

4.2.2. Candidate Racing

- o **Protocol Option Racing:** Protocol Racing is the act of attempting to establish, or scheduling attempts to establish, multiple Protocol Stacks that differ based on the composition of protocols or the options used for protocols.
- o **Path Racing:** Path Racing is the act of attempting to establish, or scheduling attempts to establish, multiple Protocol Stacks that differ based on a selection from the available Paths. Since different Paths will have distinct configurations for local addresses and DNS servers, attempts across different Paths will perform separate DNS resolution steps, which can lead to further racing of the resolved Remote Endpoints.
- o **Remote Endpoint Racing:** Remote Endpoint Racing is the act of attempting to establish, or scheduling attempts to establish, multiple Protocol Stacks that differ based on the specific representation of the Remote Endpoint, such as IP addresses resolved from a DNS hostname.

4.2.3. Protocol Stack Equivalence

The Transport Services architecture defines a mechanism that allows applications to easily use different network paths and Protocol Stacks. Transitioning between different Protocol Stacks is in some cases controlled by properties that only change when application code is updated. For example, an application can enable the use of a multipath or multistreaming transport protocol by modifying the properties in its Pre-Connection configuration. In some cases, however, the Transport Services system will be able to automatically change Protocol Stacks without an update to the application, either by selecting a new stack entirely, or by racing multiple candidate Protocol Stacks during connection establishment. This functionality can be a powerful driver of new protocol adoption, but needs to be

constrained carefully to avoid unexpected behavior that can lead to functional or security problems.

If two different Protocol Stacks can be safely swapped, or raced in parallel (see Section 4.2.2), then they are considered to be "equivalent". Equivalent Protocol Stacks need to meet the following criteria:

1. Both stacks MUST offer the same interface to the application for connection establishment and data transmission. For example, if one Protocol Stack has UDP as the top-level interface to the application, then it is not equivalent to a Protocol Stack that runs TCP as the top-level interface. Among other differences, the UDP stack would allow an application to read out message boundaries based on datagrams sent from the remote system, whereas TCP does not preserve message boundaries on its own.
2. Both stacks MUST offer the same transport services, as required by the application. For example, if an application specifies that it requires reliable transmission of data, then a Protocol Stack using UDP without any reliability layer on top would not be allowed to replace a Protocol Stack using TCP. However, if the application does not require reliability, then a Protocol Stack that adds unnecessary reliability might be allowed as an equivalent Protocol Stack as long as it does not conflict with any other application-requested properties.
3. Both stacks MUST offer the same security properties. The inclusion of transport security protocols [I-D.ietf-taps-transport-security] in a Protocol Stack adds additional restrictions to Protocol Stack equivalence. Security features and properties, such as cryptographic algorithms, peer authentication, and identity privacy vary across security protocols, and across versions of security protocols. Protocol equivalence ought not to be assumed for different protocols or protocol versions, even if they offer similar application configuration options. To ensure that security protocols are not incorrectly swapped, Transport Services systems SHOULD only automatically generate equivalent Protocol Stacks when the transport security protocols within the stacks are identical. Specifically, a transport system would consider protocols identical only if they are of the same type and version. For example, the same version of TLS running over two different transport protocol stacks are considered equivalent, whereas TLS 1.2 and TLS 1.3 [RFC8446] are not considered equivalent.

4.2.4. Separating Connection Groups

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

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

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

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

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

5. IANA Considerations

RFC-EDITOR: Please remove this section before publication.

This document has no actions for IANA.

6. Security Considerations

The Transport Services architecture does not recommend use of specific security protocols or algorithms. Its goal is to offer ease of use for existing protocols by providing a generic security-related

interface. Each provided interface translates to an existing protocol-specific interface provided by supported security protocols. For example, trust verification callbacks are common parts of TLS APIs. Transport Services APIs will expose similar functionality [I-D.ietf-taps-transport-security].

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

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

Moreover, unlike certain transport features such as TCP Fast Open (TFO) [RFC7413] or Explicit Congestion Notification (ECN) [RFC3168] which can fall back to standard configurations, Transport Services systems MUST prohibit fallback for security protocols. For example, if a client requests TLS, yet TLS or the desired version are not available, its connection will fail. Clients are thus responsible for implementing security protocol fallback or version fallback by creating multiple Transport Services Connections, if so desired.

7. Acknowledgements

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

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

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

Thanks to Stuart Cheshire, Josh Graessley, David Schinazi, and Eric Kinnear for their implementation and design efforts, including Happy Eyeballs, that heavily influenced this work.

8. Informative References

- [I-D.ietf-taps-impl]
Brunstrom, A., Pauly, T., Enghardt, T., Grinnemo, K., Jones, T., Tiesel, P., Perkins, C., and M. Welzl, "Implementing Interfaces to Transport Services", draft-ietf-taps-impl-02 (work in progress), October 2018.
- [I-D.ietf-taps-interface]
Trammell, B., Welzl, M., Enghardt, T., Fairhurst, G., Kuehlewind, M., Perkins, C., Tiesel, P., and C. Wood, "An Abstract Application Layer Interface to Transport Services", draft-ietf-taps-interface-02 (work in progress), October 2018.
- [I-D.ietf-taps-minset]
Welzl, M. and S. Gjessing, "A Minimal Set of Transport Services for End Systems", draft-ietf-taps-minset-11 (work in progress), September 2018.
- [I-D.ietf-taps-transport-security]
Wood, C., Enghardt, T., Pauly, T., Perkins, C., and K. Rose, "A Survey of Transport Security Protocols", draft-ietf-taps-transport-security-06 (work in progress), March 2019.
- [POSIX] "IEEE Std. 1003.1-2008 Standard for Information Technology -- Portable Operating System Interface (POSIX). Open group Technical Standard: Base Specifications, Issue 7", n.d..
- [RFC0793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, DOI 10.17487/RFC0793, September 1981, <<https://www.rfc-editor.org/info/rfc793>>.
- [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>>.
- [RFC3168] Ramakrishnan, K., Floyd, S., and D. Black, "The Addition of Explicit Congestion Notification (ECN) to IP", RFC 3168, DOI 10.17487/RFC3168, September 2001, <<https://www.rfc-editor.org/info/rfc3168>>.
- [RFC4303] Kent, S., "IP Encapsulating Security Payload (ESP)", RFC 4303, DOI 10.17487/RFC4303, December 2005, <<https://www.rfc-editor.org/info/rfc4303>>.

- [RFC6265] Barth, A., "HTTP State Management Mechanism", RFC 6265, DOI 10.17487/RFC6265, April 2011, <<https://www.rfc-editor.org/info/rfc6265>>.
- [RFC7413] Cheng, Y., Chu, J., Radhakrishnan, S., and A. Jain, "TCP Fast Open", RFC 7413, DOI 10.17487/RFC7413, December 2014, <<https://www.rfc-editor.org/info/rfc7413>>.
- [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>>.
- [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>>.
- [RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <<https://www.rfc-editor.org/info/rfc8446>>.

Authors' Addresses

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

Email: tpauly@apple.com

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

Email: ietf@trammell.ch

Anna Brunstrom
Karlstad University
Universitetsgatan 2
651 88 Karlstad
Sweden

Email: anna.brunstrom@kau.se

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

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

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

Email: csp@csperkins.org

Philipp S. Tiesel
TU Berlin
Marchstrasse 23
10587 Berlin
Germany

Email: philipp@inet.tu-berlin.de

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

Email: cawood@apple.com