TAPS Working Group
Internet-Draft
Intended status: Standards Track
Expires: April 25, 2019

T. Pauly, Ed.
Apple Inc.
B. Trammell, Ed.
ETH Zurich
A. Brunstrom
Karlstad University
G. Fairhurst
University of Aberdeen
C. Perkins
University of Glasgow
P. Tiesel
TU Berlin
C. Wood
Apple Inc.
October 22, 2018

An Architecture for Transport Services
draft-ietf-taps-arch-02

Abstract

This document provides an overview of the architecture of Transport Services, a system for exposing the features of transport protocols to applications. This architecture serves as a basis for Application Programming Interfaces (APIs) and implementations that provide flexible transport networking services. It defines the common set of terminology and concepts to be used in more detailed discussion of Transport Services.

Status of This Memo

Table of Contents

1. Introduction

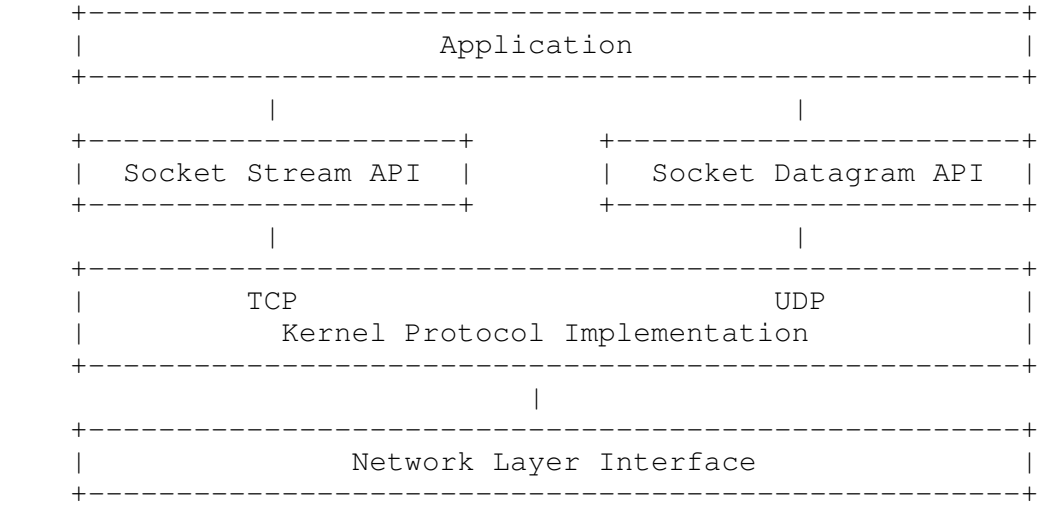
Many application programming interfaces (APIs) to perform transport networking have been deployed, perhaps the most widely known and imitated being the BSD socket() [POSIX] interface. The names and functions between these APIs are not consistent, and vary depending on the protocol being used. For example, sending and receiving on a stream of data is conceptually the same between operating on an unencrypted Transmission Control Protocol (TCP) stream and operating on an encrypted Transport Layer Security (TLS) [I-D.ietf-tls-tls13] 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 protocols offering transport services vary based on the context of the protocols themselves. 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.

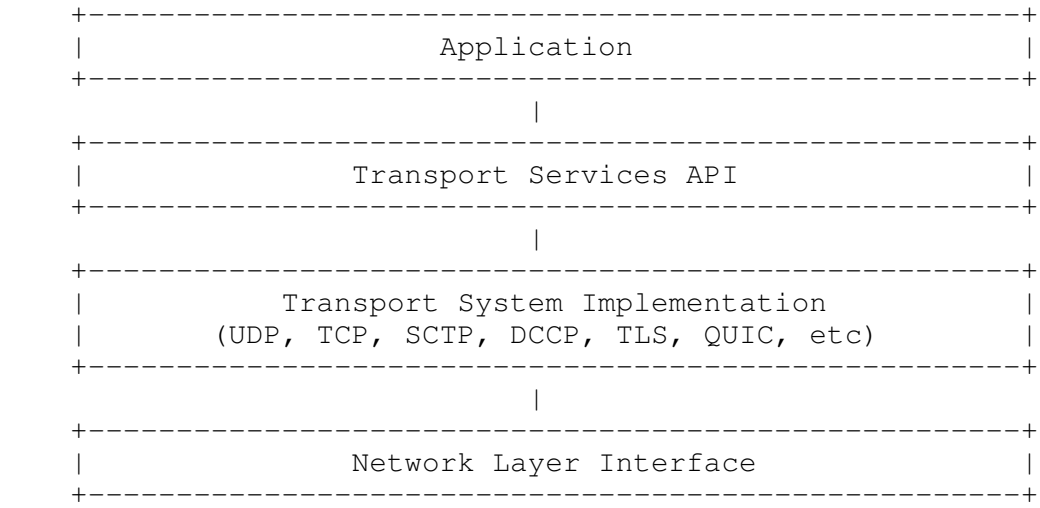
This document is developed in parallel with the specification of the Transport Services API [I-D.ietf-taps-interface] and Implementation [I-D.ietf-taps-impl] documents.

1.1. Overview

The model for 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.



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.



The Transport Services API [I-D.ietf-taps-interface] defines the mechanism for an application to create and monitor 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 respresenting 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 for a notification that it should try again.

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. 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.

In order to more closely match the way applications use the network, the Transport Services API respresents data as messages. Messages seamlessly work with transport protocols that support datagrams or records, but can also be used over a stream by defining the application-layer framing being used Section 4.4.

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]. This work has 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, or perhaps even a single transport

protocol, but would not harm the functionality of other protocols. For example, some protocols can deliver messages faster for applications that do not require them to arrive in the order in which they were sent. However, this functionality must be explicitly allowed by the application, since reordering messages would be undesirable in many cases.

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 may have different performance and functional characteristics, and it can communicate with different Remote Endpoints 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 may 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 should 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 must 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 peers,. A Transport Services system will therefore also needs to allow 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 should 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 if there are equivalent options.

Other specialized features, however, may 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 should 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 must 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.

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 [TAPS-IMPL], which also explain the implications of the feature

provided by 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 nor the Implementation document defines new protocols that require any changes to a remote host. The Transport Services system must be deployable on one side only, as a way to allow an application to make better use of available capabilities on a system and protocol features that may be supported by peers across the network.

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 may 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 meant to be exposed to applications; and

2. System-implementation concepts, which are meant 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.

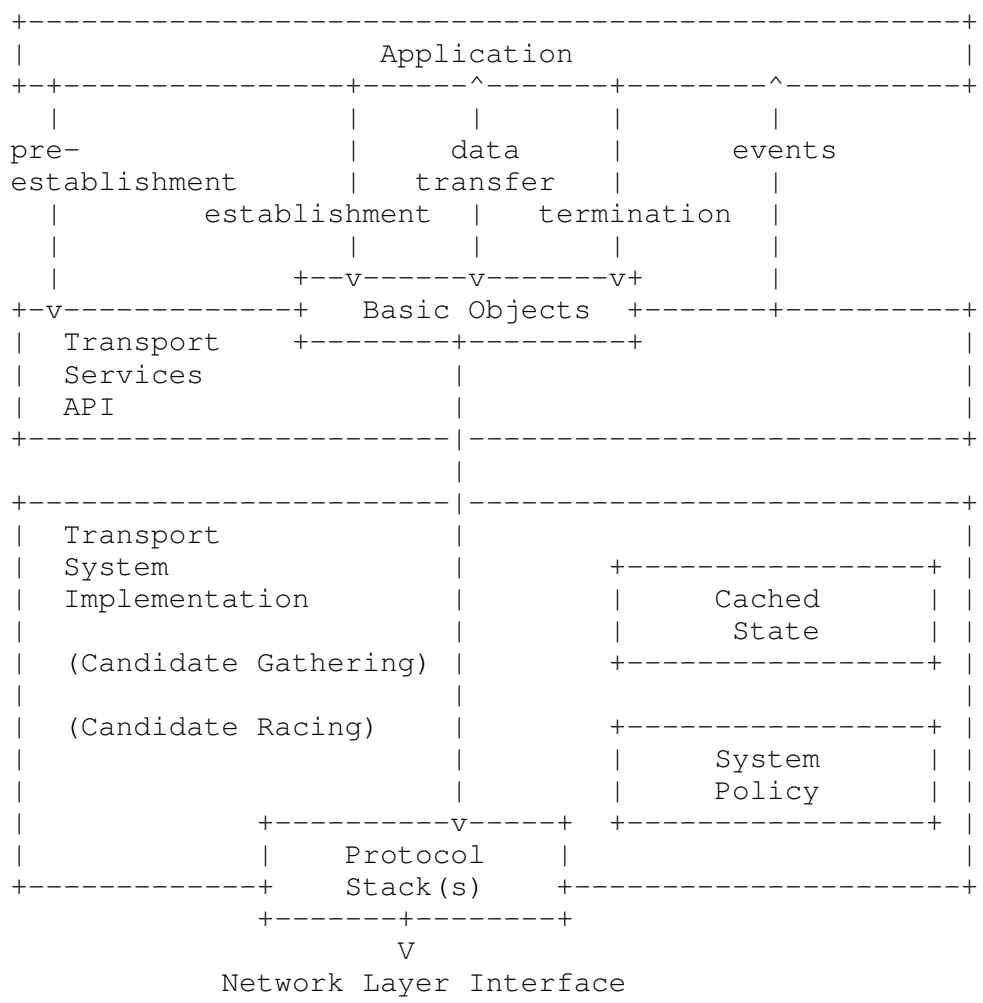


Figure 1: Concepts and Relationships in the Transport Services Architecture

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 may 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 must provide:

- 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 primarily be defined to apply to multiple transports. Properties may have a large impact on the rest of the aspects of the interface: they can modify how establishment occurs, they can influence the expectations around data transfer, and they determine the set of events that will be supported.

- Establishment (Section 4.1.3) focuses on the actions that an application takes on the basic objects to prepare for data transfer.

- Data Transfer (Section 4.1.4) consists of how an application represents 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 4.1.5) defines the set of properties about which an application can receive notifications during the lifetime of transport objects. Events can also provide opportunities for the application to interact with the underlying transport by querying state or updating maintenance options.

- 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.

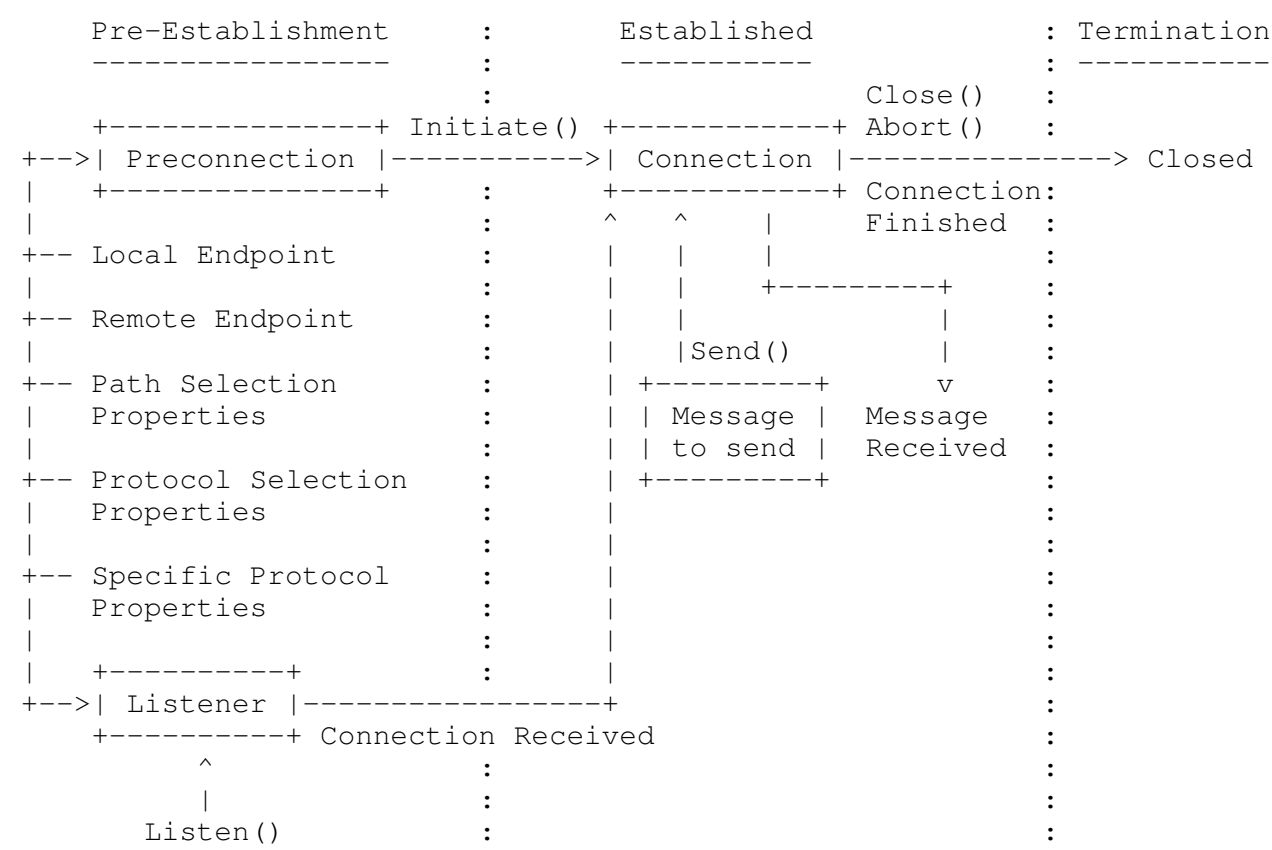


Figure 2: The lifetime of a connection

4.1.1. Basic Objects

- Preconnection: A Preconnection object is a representation of a potential connection. It has state that describes parameters of a Connection that might exist in the future: the Local Endpoint from which that Connection will be established, the Remote Endpoint to which it will connect, and Path Selection Properties, Protocol Selection Properties, and Specific Protocol Properties that influence the choice of transport that 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 must be specified if the Preconnection is used to Listen for incoming connections, but is optional if it is used to Initiate connections. The Remote Endpoint must be specified in the Preconnection is used to Initiate connections, but 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.

- Connection: A Connection object represents an active transport protocol instance that can send and/or receive Messages between a Local Endpoint and a Remote Endpoint. 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.

- 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.

4.1.2. Pre-Establishment

- Endpoint: An Endpoint represents 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 may 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).

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

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

- Path Selection Properties: The Path Selection Properties consist of the options that an application may set to influence the selection of paths between the Local Endpoint and the Remote Endpoint. These options can take the form of requirements, prohibitions, or preferences. Examples of options that may influence path selection include the interface type (such as a Wi-Fi Ethernet connection, or a Cellular LTE connection), characteristics of the path that are locally known like Maximum Transmission Unit (MTU) or discovered like Path MTU (PMTU), or predicted based on cached information like expected throughput or latency.

- Protocol Selection Properties: The Protocol Selection Properties consist of the options that an application may set to influence the selection of transport protocol, or to configure the behavior of generic transport protocol features. These options can take

the form of requirements, prohibitions, and preferences. Examples include reliability, service class, multipath support, and fast open support.

- Specific Protocol Properties: The Specific Protocol Properties refer to the subset of Protocol Properties options that apply to a single protocol (transport protocol, IP, or security protocol). The presence of such Properties does not necessarily require that a specific protocol must be used when a Connection is established, but that if this protocol is employed, a particular set of options should then be used..

4.1.3. 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 be able to send and/or receive Messages. For some protocols, this may initiate a client-to-server style handshake; for other protocols, this may just establish local state. The process of identifying options for connecting, such as resolution of the Remote Endpoint, occurs in response the Initiate call.

- 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).

- 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

- 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 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 may or may not be usable by certain applications. Boundaries of a Message may or may not be understood or transmitted by transport

protocols. Specifically, what one application considers to be two Messages sent on a stream-based transport may be treated as a single Message by the application on the other side.

- Send: The action to transmit a Message or partial Message over a Connection to a Remote Endpoint. The interface to Send may include options specific to how the Message's content is to be sent. Status of the Send operation may be delivered back to the application in an event (Section 4.1.5).

- Receive: An action that indicates that the application is ready to asynchronously accept a Message over a Connection from a Remote Endpoint, while the Message content itself will be delivered in an event (Section 4.1.5). The interface to Receive may include options specific to the Message that is to be delivered to the application.

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.

- 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 Finished: Signals to an application that a given Connection is no longer usable for sending or receiving Messages. This should deliver an error to the application that describes the nature of the termination.

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

- 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.

- Message Sent: Notifies the application of the status of its Send action. This may be an error if the Message cannot be sent, or an indication that Message has been processed by the protocol stack.

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

4.1.6. Termination Actions

- Close: The action an application may take on a Connection to indicate that it no longer intends to send data, is no longer willing to receive data, and that the protocol should signal this state to the remote endpoint if applicable.

- Abort: The action the application may take on a Connection to indicate a Close, but with the additional indication 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 provide the functionality required to provide a transport service across a network, as required by the abstract interface.

- Connection Group: A set of Connections that share properties. For multiplexing transport protocols, the Connection Group defines the set of Connections that can be multiplexed together.

- Path: Represents an available set of properties that a Local Endpoint may use to send or receive packets with a Remote Endpoint.

- Protocol Instance: A single instance of one protocol, including any state it has 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 may 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).

- Candidate Path: One path that is available to an application and conforms to the Path Selection Properties and System Policy. Candidate Paths are identified during the gathering phase (Section 4.2.1) and may be used during the racing phase (Section 4.2.2).

- Candidate Protocol Stack: One protocol stack that may be used by an application for a connection, of which there may be several.

Candidate Protocol Stacks are identified during the gathering phase (Section 4.2.1) and may be started during the racing phase (Section 4.2.2).

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

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

4.2.1. Candidate Gathering

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

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

4.2.2. Candidate Racing

- 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.

- 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.

- Endpoint Racing: 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 and the Local Endpoint, such as IP addresses resolved from a DNS hostname.

4.3. 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 may in some cases be controlled by properties that only change when application code is updated. For example, an application may 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 racing multiple candidate Protocol Stacks during connection establishment. This functionality can be a powerful driver of new protocol adoption, but must 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 must 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 Endpoint, 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. See the security protocol equivalence section below for futher discussion (Section 4.3.1).

4.3.1. Transport Security Equivalence

The inclusion of transport security protocols [I-D.ietf-taps-transport-security] in a Protocol Stack adds extra 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 should not 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 system should 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 may be considered equivalent, whereas TLS 1.2 and TLS 1.3 [I-D.ietf-tls-tls13] should not be considered equivalent.

4.4. Message Framing, Parsing, and Serialization

While some transports expose a byte stream abstraction, most higher level protocols impose some structure onto that byte stream. That is, the higher level protocol operates in terms of messages, protocol data units (PDUs), rather than using unstructured sequences of bytes, with each message being processed in turn. Protocols are specified in terms of state machines acting on semantic messages, with parsing the byte stream into messages being a necessary annoyance, rather than a semantic concern. Accordingly, the Transport Services architecture exposes messages as the primary abstraction. Protocols that deal only in byte streams, such as TCP, represent their data in each direction as a single, long message. When framing protocols are placed on top of byte streams, the messages used in the API represent the framed messages within the stream.

Providing a message-based abstraction also provides:

- the ability to associate deadlines with messages, for transports that care about timing;
- the ability to provide control of reliability, choosing what messages to retransmit in the event of packet loss, and how best to make use of the data that arrived;
- the ability to manage dependencies between messages, when some messages may not be delivered due to either packet loss or missing

a deadline, in particular the ability to avoid (re-)sending data that relies on a previous transmission that was never received.

All require explicit message boundaries, and application-level framing of messages, to be effective. Once a message is passed to the transport, it can not be cancelled or paused, but prioritization as well as lifetime and retransmission management will provide the protocol stack with all needed information to send the messages as quickly as possible without blocking transmission unnecessarily. The transport services architecture facilitates this by handling messages, with known identity (sequence numbers, in the simple case), lifetimes, niceness, and antecedents.

Transport protocols such as SCTP provide a message-oriented API that has similar features to those we describe. Other transports, such as TCP, do not. To support a message oriented API, while still being compatible with stream-based transport protocols, implementations of the transport services architecture should provide APIs for framing and de-framing messages. That is, we push message framing down into the transport services API, allowing applications to send and receive complete messages. This is backwards compatible with existing protocols and APIs, since the wire format of messages does not change, but gives the protocol stack additional information to allow it to make better use of modern transport services.

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 mimics 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.

Clients must take care to use security APIs appropriately. In cases where clients use said interface to provide sensitive keying material, e.g., access to private keys or copies of pre-shared keys (PSKs), key use must be validated. For example, clients should not use PSK material created for the Encapsulating Security Protocol (ESP, part of IPsec) [RFC4303] with QUIC, and clients must not 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 not permit fallback for security protocols. For example, if a client requests TLS, yet TLS or the desired version are not available, its connection must fail. Clients are responsible for implementing protocol or version fallback using a Transport Services API if so desired.

7. Acknowledgements

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

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-01 (work in progress), July 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-01 (work in progress), July 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]
Pauly, T., Perkins, C., Rose, K., and C. Wood, "A Survey of Transport Security Protocols", draft-ietf-taps-transport-security-02 (work in progress), June 2018.

[I-D.ietf-tls-tls13]
Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", draft-ietf-tls-tls13-28 (work in progress), March 2018.

[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>.

[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>.

[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>.

[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-01 (work in progress), July 2018.

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)
ETH Zurich
Gloriastrasse 35
8092 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