TAPS Working Group
Internet-Draft
Intended status: Informational
Expires: April 30, 2017

B. Trammell
ETH Zurich
C. Perkins
University of Glasgow
T. Pauly
Apple Inc.
M. Kuehlewind
ETH Zurich

October 27, 2016

Post Sockets, An Abstract Programming Interface for the Transport Layer
draft-trammell-post-sockets-00

Abstract

This document describes Post Sockets, an asynchronous abstract programming interface for the atomic transmission of objects in an explicitly multipath environment. Post replaces connections with long-lived associations between endpoints, with the possibility to cache cryptographic state in order to reduce amortized connection latency. We present this abstract interface as an illustration of what is possible with present developments in transport protocols when freed from the strictures of the current sockets API.

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 http://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 April 30, 2017.

Copyright Notice

Table of Contents

1. Introduction

The BSD Unix Sockets API's SOCK_STREAM abstraction, by bringing network sockets into the UNIX programming model, allowing anyone who knew how to write programs that dealt with sequential-access files to also write network applications, was a revolution in simplicity. It would not be an overstatement to say that this simple API is the reason the Internet won the protocol wars of the 1980s. SOCK_STREAM is tied to the Transmission Control Protocol (TCP), specified in 1981 [RFC0793]. TCP has scaled remarkably well over the past three and a half decades, but its total ubiquity has hidden an uncomfortable fact: the network is not really a file, and stream abstractions are too simplistic for many modern application programming models.

In the meantime, the nature of Internet access is evolving. Many end-user devices are connected to the Internet via multiple interfaces, which suggests it is time to promote the "path" by which a host is connected to a first-order object; we call this "path primacy".

Implicit multipath communication is available for these multihomed nodes in the present Internet architecture with the Multipath TCP extension (MPTCP) [RFC6824]. Since many multihomed nodes are connected to the Internet through access paths with widely different properties with respect to bandwidth, latency and cost, adding explicit path control to MPTCP's API would be useful in many situations. Path primacy for cooperation with path elements is also useful in single-homed architectures, such as the mechanism proposed by the Path Layer UDP Substrate (PLUS) effort (see [I-D.trammell-plus-statefulness] and [I-D.trammell-plus-abstract-mech]).

Another trend straining the traditional layering of the transport stack associated with the SOCK_STREAM interface is the widespread interest in ubiquitous deployment of encryption to guarantee confidentiality, authenticity, and integrity, in the face of pervasive surveillance [RFC7258]. Layering the most widely deployed encryption technology, Transport Layer Security (TLS), strictly atop TCP (i.e., via a TLS library such as OpenSSL that uses the sockets API) requires the encryption-layer handshake to happen after the transport-layer handshake, which increases connection setup latency on the order of one or two round-trip times, an unacceptable delay for many applications. Integrating cryptographic state setup and maintenance into the path abstraction naturally complements efforts in new protocols (e.g. QUIC [I-D.hamilton-quic-transport-protocol]) to mitigate this strict layering.

From these three starting points - more flexible abstraction, path primacy, and encryption by default - we define the Post-Socket Application Programming Interface (API), described in detail in this work. Post is designed to be language, transport protocol, and architecture independent, allowing applications to be written to a common abstract interface, easily ported among different platforms, and used even in environments where transport protocol selection may be done dynamically, as proposed in the IETF's Transport Services wotking group (see https://datatracker.ietf.org/wg/taps/charter).

Post replaces the traditional SOCK_STREAM abstraction with an Object abstraction, which can be seen as a generalization of the Stream Control Transmission Protocol's [RFC4960] SOCK_SEQPACKET service. Objects can be small (e.g. messages in message-oriented protocols) or large (e.g. an HTTP response containing header and body). It

replaces the notions of a socket address and connected socket with an Association with a remote endpoint via set of Paths. Implementation and wire format for transport protocol(s) implementing the Post API are explicitly out of scope for this work; these abstractions need not map directly to implementation-level concepts, and indeed with various amounts of shimming and glue could be implemented with varying success atop any sufficiently flexible transport protocol.

For compatibility with situations where only strictly stream-oriented transport protocols are available, applications with data streams that cannot be easily split into Objects at the sender, and and for easy porting of the great deal of existing stream-oriented application code to Post, Post also provides a SOCK_STREAM compatible abstraction, unimaginatively named Stream.

The key features of Post as compared with the existing sockets API are:

- Explicit Object orientation, with framing and atomicity guarantees for Object transmission.
- Asynchronous reception, allowing all receiver-side interactions to be event-driven.
- Explicit support for multipath transport protocols and network architectures.
- Long-lived Associations, whose lifetimes may not be bound to underlying transport connections. This allows associations to cache state and cryptographic key material to enable fast (0-rtt) resumption of communication.

This work is the synthesis of many years of Internet transport protocol research and development. It is heavily inspired by concepts from the Stream Control Transmission Protocol (SCTP) [RFC4960], TCP Minion [I-D.iyengar-minion-protocol], MinimaLT[MinimaLT], and various bulk object transports.

We present Post Sockets as an illustration of what is possible with present developments in transport protocols when freed from the strictures of the current sockets API. While much of the work for building parts of the protocols needed to implement Post are already ongoing in other IETF working groups (e.g. TAPS, MPTCP, QUIC, TLS), we argue that an abstract programming interface unifying access all these efforts is necessary to fully exploit their potential.

2. Abstractions and Terminology

gratuitously colorful SVG goes here; see slide six of

https://www.ietf.org/proceedings/96/slides/slides-96-taps-2.pdf

in the meantime

Figure 1: Abstractions and relationships in Post Sockets

Post is based on a small set of abstractions, the relationships among which are shown in Figure Figure 1 and detailed in this section.

2.1. Association

An Association is a container for all the state necessary for a local endpoint to communicate with a remote endpoint in an explicitly multipath environment. It contains a set of Paths, certificate(s) for identifying the remote endpoint, certificate(s) and key(s) for identifying the local endpoint to the remote endpoint, and any cached cryptographic state for the communication to the remote endpoint. An Association may have one or more Streams active at any given time. Objects are sent to Associations, as well.

Note that, in contrast to current SOCK_STREAM sockets, Associations are meant to be relatively long-lived. The lifetime of an Association is not bound to the lifetime of any transport-layer connection between the two endpoints; connections may be opened or closed as necessary to support the Streams and Object transmissions required by the application, and the application need not be bothered with the underlying connectivity state unless this is important to the application's semantics.

Paths may be dynamically added or removed from an association, as well, as connectivity between the endpoints changes. Cryptographic identifiers and state for endpoints may also be added and removed as necessary due to certificate lifetimes, key rollover, and revocation.

2.2. Listener

In many applications, there is a distinction between the active opener (or connection initiator, often a client), and the passive opener (often a server). A Listener represents an endpoint's willingness to start Associations in this passive opener/server role. It is, in essence, a one-sided, Path-less Association from which fully-formed Associations can be created.

Listeners work very much like sockets on which the listen(2) call has been called in the SOCK_STREAM API.

2.3. Remote

A Remote represents all the information required to establish and maintain a connection with the far end of an Association: network-layer address, transport-layer port, information about public keys or certificate authorities used to identify the remote on connection establishment, etc. Each Association is associated with a single Remote, either explicitly by the application (when created by active open) or by the Listener (when created by passive open). The resolution of Remotes from higher-layer information (URIs, hostnames) is architecture-dependent.

2.4. Local

A Local represents all the information about the local endpoint necessary to establish an Association or a Listener: interface and port designators, as well as certificates and associated private keys.

2.5. Path

A Path represents a local and remote endpoint address, an optional set of intermediary path elements between the local and remote endpoint addresses, and a set of properties associated with the path.

The set of available properties is a function of the underlying network-layer protocols used to expose the properties to the endpoint. However, the following core properties are generally useful for applications and transport layer protocols to choose among paths for specific Objects:

- Maximum Transmission Unit (MTU): the maximum size of an Object's payload (subtracting transport, network, and link layer overhead) which will likely fit into a single frame. Derived from signals sent by path elements, where available, and/or path MTU discovery processes run by the transport layer.

- Latency Expectation: expected one-way delay along the Path. Generally provided by inline measurements performed by the transport layer, as opposed to signaled by path elements.

- Loss Probability Expectation: expected probability of a loss of any given single frame along the Path. Generally provided by inline measurements performed by the transport layer, as opposed to signaled by path elements.

- Available Data Rate Expectation: expected maximum data rate along the Path. May be derived from passive measurements by the transport layer, or from signals from path elements.

- Reserved Data Rate: Committed, reserved data rate for the given Association along the Path. Requires a bandwidth reservation service in the underlying transport and network layer protocol.

- Path Element Membership: Identifiers for some or all nodes along the path, depending on the capabilities of the underlying network layer protocol to provide this.

Path properties are generally read-only. MTU is a property of the underlying link-layer technology on each link in the path; latency, loss, and rate expectations are dynamic properties of the network configuration and network traffic conditions; path element membership is a function of network topology. In an explicitly multipath architecture, application and transport layer requirements are met by having multiple paths with different properties to select from. Post can also provide signaling to the path, but this signaling is derived from information provided to the Object abstraction, below.

Note that information about the path and signaling to path elements could be provided by a facility such as PLUS [I-D.trammell-plus-abstract-mech].

2.6. Object

Post provides two ways to send data over an Association. We start with the Object abstraction, as a fundamental insight behind the interface is that most applications fundamentally deal in object transport.

An Object is an atomic unit of communication between applications; or in other words, an ordered collection of bytes B0..Bm, such that every byte Bn depends on every other byte in the Object. An object that cannot be delivered in its entirety within the constraints of the network connectivity and the requirements of the application is not delivered at all.

Objects can represent both relatively small structures, such as messages in application-layer protocols built around datagram or message exchange, as well as relatively large structures, such files of arbitrary size in a filesystem. Objects larger than the MTU on the Path on which they are sent will be segmented into multiple frames. Multiple objects that will fit into a single frame may be concatenated into one frame. There is no preference for transmitting the multiple frames for a given Object in any particular order, or by

default, that objects will be delivered in the order sent by the application. This implies that both the sending and receiving endpoint, whether in the application layer or the transport layer, must guarantee storage for the full size of an object.

Three object properties allow applications fine control ordering and reliability requirements in line with application semantics. An Object may have a "lifetime" - a wallclock duration before which the object must be available to the application layer at the remote end. If a lifetime cannot be met, the object is discarded as soon as possible; therefore, Objects with lifetimes are implicitly sent non-reliably, and lifetimes are used to prioritize Object delivery. Lifetimes may be signaled to path elements by the underlying transport, so that path elements that realize a lifetime cannot be met can discard frames containing the object instead of forwarding them.

Second, Objects may have a "niceness" - a category in an unbounded hierarchy most naturally represented as a non-negative integer. By default, Objects are in niceness class 0, or highest priority. Niceness class 1 Objects will yield to niceness class 0 objects, class 2 to class 1, and so on. Niceness may be translated to a priority signal for exposure to path elements (e.g. DSCP codepoint) to allow prioritization along the path as well as at the sender and receiver. This inversion of normal schemes for expressing priority has a convenient property: priority increases as both niceness and deadline decrease.

An object may have both a niceness and a lifetime - objects with higher niceness classes will yield to lower classes if resource constraints mean only one can meet the lifetime.

Third, an Object may have "antecedents" - other Objects on which it depends, which must be delivered before it (the "successor") is delivered. The sending transport uses deadlines, niceness, and antecedents, along with information about the properties of the Paths available, to determine when to send which object down which Path.

When an application has hard semantic requirements that all the frames of a given object be sent down a given Path or Paths, these hard constraints can also be expressed by the application.

After calling the send function, the application can register event handlers to be informed of the transmission status of the object; the object can either be acknowledged (i.e., it has been received in full by the remote endpoint) or expired (its lifetime ran out before it was acknowledged).

2.7. Stream

The Stream abstraction is provided for two reasons. First, since it is the most like the existing SOCK_STREAM interface, it is the simplest abstraction to be used by applications ported to Post to take advantages of Path primacy. Second, some environments have connectivity so impaired (by local network operation policy and/or accidental middlebox interference) that only stream- based transport protocols are available, and applications should have the option to use streams directly in these situations.

A Stream is a sequence of bytes B0 .. Bm such that the reception (and delivery to the receiving application of) Bn always depends on Bn-1. This property is inherited from the BSD UNIX file abstraction, which in turn inherited it from the physical limitations of sequential access media (stacks of punch cards, paper and/or magnetic tape).

A Stream is bound to an Association. Writing a byte to the stream will cause it to be received by the remote, in order, or will cause an error condition and termination of the stream if the byte cannot be delivered. Due to the strong sequential dependence on a stream, streams must always be reliable and ordered. If frames containing Stream data are lost, these must be retransmitted or reconstructed using an error correction technique. If frames containing Stream data arrive out of order, the remote end must buffer them until the unordered frames are received and reassembled.

As with Objects, Streams may have a niceness for prioritization. When mixing Stream and Object data on the same Path in an association, the niceness classes for Streams and Objects are interleaved; e.g. niceness 2 Stream frames will yield to niceness 1 Object frames.

The underlying transport protocol may make whatever use of the Paths and known properties of those Paths it sees fit when transporting a Stream.

3. Abstract Programming Interface

We now turn to the design of an abstract programming interface to provide a simple interface to Post’s abstractions, constrained by the following design principles:

- Flexibility is paramount. So is simplicity. Applications must be given as many controls and as much information as they may need, but they must be able to ignore controls and information irrelevant to their operation. This implies that the "default"

interface must be no more complicated than BSD sockets, and must do something reasonable.

- A new API cannot be bound to a single transport protocol and expect wide deployment. As the API is transport-independent and may support runtime transport selection, it must impose the minimum possible set of constraints on its underlying transports, though some API features may require underlying transport features to work optimally. It must be possible to implement Post over vanilla TCP in the present Internet architecture.

- Reception is an inherently asynchronous activity. While the API is designed to be as platform-independent as possible, one key insight it is based on is that an object receiver's behavior in a packet-switched network is inherently asynchronous, driven by the receipt of packets, and that this asynchronicity must be reflected in the API. The actual implementation of receive and event callbacks will need to be aligned to the method a given platform provides for asynchronous I/O.

The API we define consists of three classes (listener, association, and stream), four entry points (listen(), associate(), send(), and open_stream()) and a set of callbacks for handling events at each endpoint. The details are given in the subsections below.

3.1. Active Association Creation

Associations can be created two ways: actively by a connection initiator, and passively by a Listener that accepts a connection. Connection initiation uses the associate() entry point:

association = associate(local, remote, receive_handler)

where:

- local: a resolved Local (see Section 3.8) describing the local identity and interface(s) to use

- remote: a resolved Remote (see Section 3.8) to associate with

- receive_handler: a callback to be invoked when new objects are received; see Section 3.4

The returned association has the following additional properties:

- paths: a set of Paths that the Association can currently use to transport Objects. When the underlying transport connection is closed, this set will be empty. For explicitly multipath

architectures and transports, this set may change dynamically during the lifetime of an association, even while it remains connected.

Since the existence of an association does not necessarily imply current connection state at both ends of the Association, these objects are durable, and can be cached, migrated, and restored, as long as the mappings to their event handlers are stable. An attempts to send an object or open a stream on a dormant, previously actively-opened association will cause the underlying transport connection state to be resumed.

3.2. Listener and Passive Association Creation

In order to accept new Association requests from clients, a server must create a Listener object, using the listen() entry point:

listener = listen(local, accept_handler)

where:

- local: resolved Local (see Section 3.8) describing the local identity and interface(s) to use for Associations created by this listener.

- accept_handler: callback to be invoked each time an association is requested by a remote, to finalize setting the association up. Platforms may provide a default here for supporting synchronous association request handling via an object queue.

The accept_handler has the following prototype:

accepted = accept_handler(listener, local, remote)

where:

- local: a resolved Local on which the association request was received.

- remote: a resolved Remote from which the association request was received.

- accepted: flag, true if the handler decided to accept the request, false otherwise.

The accept_handler() calls the accept() entry point to finally create the association:

association = accept(listener, local, remote, receive_handler)

3.3. Sending Objects

Objects are sent using the send() entry point:

send(association, bytes, [lifetime], [niceness], [oid], [antecedent_oids], [paths])}

where:

- association: the association to send the object on
- bytes: sequence of bytes making up the object. For platforms without bounded byte arrays, this may be implemented as a pointer and a length.
- lifetime: lifetime of the object in milliseconds. This parameter is optional and defaults to infinity (for fully reliable object transport).
- niceness: the object's niceness class. This parameter is optional and defaults to zero (for lowest niceness / highest priority)
- oid: opaque identifier for an object, assigned by the application. Used to refer to this object as a subsequently sent object's antecedent, or in an ack or expired handler (see Section 3.6). Optional, defaults to null.
- antecedent_oids: set of object identifiers on which this object depends and which must be sent before this object. Optional, defaults to empty, meaning this object has no antecedent constraints.
- paths: set of paths, as a subset of those available to the association, to explicitly use for this object. Optional, defaults to empty, meaning all paths are acceptable.

Calls to send are non-blocking; a synchronous send which blocks on remote acknowledgment or expiry of an object can be implemented by a call to send() followed by a wait on the ack or expired events (see Section 3.6).

3.4. Receiving Objects

An application receives objects via its receive_handler callback, registered at association creation time. This callback has the following prototype:

receive_handler(association, bytes)

where: - association: the association the object was received from. - bytes: the sequence of bytes making up the object.

For ease of porting synchronous datagram applications, implementations may make a default receive handler available, which allows messages to be synchronously polled from a per-association object queue. If this default is available, the entry point for the polling call is:

bytes = receive_next(association)

3.5. Creating and Destroying Streams

A stream may be created on an association via the open_stream() entry point:

stream = open_stream(association, [sid])

where:

- association: the association to open the stream on
- sid: opaque identifier for a stream. For transport protocols which do not support multiple streaming, this argument has no effect.

A stream with a given sid must be opened by both sides before it can be used.

The stream object returned should act like a file descriptor or bidirectional I/O object, according to the conventions of the platform implementing Post.

3.6. Events

Message reception is a specific case of an event that can occur on an association. Other events are also available, and the application can register event handlers for each of these. Event handlers are registered via the handle() entry point:

handle(association, event, handler) or

handle(oid, event, handler)

where

- association: the association to register a handler on, or
- oid: the object identifier to register a handler on
- event: an identifier of the event to register a handler on
- handler: a callback to be invoked when the event occurs, or null if the event should be ignored.

The following events are supported; every event handler takes the association on which it is registered as well as any additional arguments listed:

- receive (bytes): an object has been received
- path_up (path): a path is newly available
- path_down (path): a path is no longer available
- dormant: no more paths are available, the association is now dormant, and the connection will need to be resumed if further objects are to be sent
- ack (oid): an object was successfully received by the remote
- expired (oid): an object expired before being sent to the remote

Handlers for the ack and expired events can be registered on an association (in which case they are called for all objects sent on the association) or on an oid (in which case they are only called for the oid).

3.7. Paths and Path Properties

As defined in Section 2.5, the properties of a path include both the addresses of elements along the path as well as measurement-derived latency and capacity characteristics. The path_up and path_down events provide access to information about the paths available via the path argument to the event handler. This argument encapsulates these properties in a platform and transport-specific way, depending on the availability of information about the path.

3.8. Address Resolution

Address resolution turns the name of a Remote into a resolved Remote object, which encapsulates all the information needed to connect (address, certificate parameters, cached cryptographic state, etc.); and an interface identifier on a local system to information needed

to connect. Remote and local resolvers have the following entry points:

```
remote = resolve(endpoint_name, configuration)

local = resolve_local(endpoint_name, configuration)
```

where:

- endpoint_name: a name identifying the remote or local endpoint, including port
- configuration: a platform-specific configuration object for configuring certificates, name resolution contexts, cached cryptographic state, etc.

4. Acknowledgments

Many thanks to Laurent Chuat and Jason Lee at the Network Security Group at ETH Zurich for contributions to the initial design of Post Sockets.

This work is partially supported by the European Commission under Horizon 2020 grant agreement no. 688421 Measurement and Architecture for a Middleboxed Internet (MAMI), and by the Swiss State Secretariat for Education, Research, and Innovation under contract no. 15.0268. This support does not imply endorsement.

5. Informative References

[I-D.hamilton-quic-transport-protocol]
Hamilton, R., Iyengar, J., Swett, I., and A. Wilk, "QUIC: A UDP-Based Multiplexed and Secure Transport", draft-hamilton-quic-transport-protocol-00 (work in progress), July 2016.

[I-D.iyengar-minion-protocol]
Jana, J., Cheshire, S., and J. Graessley, "Minion - Wire Protocol", draft-iyengar-minion-protocol-02 (work in progress), October 2013.

[I-D.trammell-plus-abstract-mech]
Trammell, B., "Abstract Mechanisms for a Cooperative Path Layer under Endpoint Control", draft-trammell-plus-abstract-mech-00 (work in progress), September 2016.

[I-D.trammell-plus-statefulness]
Kuehlewind, M., Trammell, B., and J. Hildebrand, "Transport-Independent Path Layer State Management", draft-trammell-plus-statefulness-00 (work in progress), October 2016.

[MinimaLT]
Petullo, W., Zhang, X., Solworth, J., Bernstein, D., and T. Lange, "MinimaLT, Minimal-latency Networking Through Better Security", May 2013.

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

[RFC4960] Stewart, R., Ed., "Stream Control Transmission Protocol", RFC 4960, DOI 10.17487/RFC4960, September 2007, <http://www.rfc-editor.org/info/rfc4960>.

[RFC6824] Ford, A., Raiciu, C., Handley, M., and O. Bonaventure, "TCP Extensions for Multipath Operation with Multiple Addresses", RFC 6824, DOI 10.17487/RFC6824, January 2013, <http://www.rfc-editor.org/info/rfc6824>.

[RFC7258] Farrell, S. and H. Tschofenig, "Pervasive Monitoring Is an Attack", BCP 188, RFC 7258, DOI 10.17487/RFC7258, May 2014, <http://www.rfc-editor.org/info/rfc7258>.

Authors' Addresses

Brian Trammell
ETH Zurich
Gloriastrasse 35
8092 Zurich
Switzerland

Email: ietf@trammell.ch

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

Email: csp@cperkins.net

Tommy Pauly
Apple Inc.
1 Infinite Loop
Cupertino, California 95014
United States of America

Email: tpauly@apple.com

Mirja Kuehlewind
ETH Zurich
Gloriastrasse 35
8092 Zurich
Switzerland

Email: mirja.kuehlewind@tik.ee.ethz.ch