TAPS Working Group
Internet-Draft
Intended status: Informational
Expires: March 12, 2018

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

September 08, 2017

Post Sockets, An Abstract Programming Interface for the Transport Layer
draft-trammell-taps-post-sockets-01

Abstract

This document describes Post Sockets, an asynchronous abstract programming interface for the atomic transmission of messages in an inherently 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

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, and the variety of Internet transport protocols, is evolving. The challenges that new protocols and access paradigms present to the sockets API and to programming models based on them inspire the design elements of a new approach.

Many end-user devices are connected to the Internet via multiple interfaces, which suggests it is time to promote the paths by which two endpoints are connected to each other to a first-order object. While implicit multipath communication is available for these multihomed nodes in the present Internet architecture with the Multipath TCP extension (MPTCP) [RFC6824], MPTCP was specifically designed to hide multipath communication from the application for purposes of compatibility. 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.

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.ietf-quic-transport]) to mitigate this strict layering.

To meet these challenges, we present the Post-Sockets 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 working group.

Post replaces the traditional SOCK_STREAM abstraction with a Message abstraction, which can be seen as a generalization of the Stream Control Transmission Protocol's [RFC4960] SOCK_SEQPACKET service. Messages are sent and received on Carriers, which logically group Messages for transmission and reception. For backward compatibility, bidirectional byte stream protocols are represented as a pair of Messages, one in each direction, that can only be marked complete when the sending peer has finished transmitting data.

Post 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.

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

- Explicit Message orientation, with framing and atomicity guarantees for Message transmission.

- Asynchronous reception, allowing all receiver-side interactions to be event-driven.

- Explicit support for multistreaming and 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 resumption of communication, and for the implementation of the API to explicitly take care of connection establishment mechanics such as connection racing [RFC6555] and peer-to-peer rendezvous [RFC5245].

- Transport protocol stack independence, allowing applications to be written in terms of the semantics best for the application’s own design, separate from the protocol(s) used on the wire to achieve them. This enables applications written to a single API to make use of transport protocols in terms of the features they provide, as in [I-D.ietf-taps-transports].

This work is the synthesis of many years of Internet transport protocol research and development. It is inspired by concepts from the Stream Control Transmission Protocol (SCTP) [RFC4960], TCP Minion [I-D.iyengar-minion-protocol], and MinimaLT [MinimaLT], among other transport protocol modernization efforts. We present Post 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. 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

```
           +===============+
           |    Message    |
           +===============+
                |    ^         |               |
          send()|    |ready()  |initiate()     |listen()
                V    |         V               V
           +=====================+           +============+
           |                     |  accept() |            |
           |      Carrier        |<----------|  Listener  |
           |                     |           |            |
           +=====================+           +============+
            |1        |        n|                  |          +=========+
            |         |         |1                 |      +---|  Local  |
            |   +=========+   +=======================+   |   +=========+
            |   | Policy  |n  |                       |---+
            |   | Context |---|      Association      |       +=========+
            |   |         |  1|                       |-------|  Remote |
            |   +=========+   +=======================+       +=========+
            |         |                1| durable end-to-end
            +-------+ |                 | state via many paths,
                    | |                 | policies, and prefs
                   n| |                n|
               +===========+       +==========+
     ephemeral |           |       |          |
   transport & | Transient |-------|   Path   | properties of
  crypto state |           |n     1|          | address pair
               +===========+       +==========+
```

Figure 1: Abstractions and relationships in Post Sockets

Post is based on a small set of abstractions, centered around a Message Carrier as the entry point for an application to the networking API. The relationships among them are shown in Figure Figure 1 and detailed in this section.

2.1. Message Carrier

A Message Carrier (or simply Carrier) is a transport protocol stack-independent interface for sending and receiving messages between an application and a remote endpoint; it is roughly analogous to a socket in the present sockets API.

Sending a Message over a Carrier is driven by the application, while receipt is driven by the arrival of the last packet that allows the Message to be assembled, decrypted, and passed to the application. Receipt is therefore asynchronous; given the different models for asynchronous I/O and concurrency supported by different platforms, it may be implemented in any number of ways. The abstract API provides only for a way for the application to register how it wants to handle incoming messages.

All the Messages sent to a Carrier will be received on the corresponding Carrier at the remote endpoint, though not necessarily reliably or in order, depending on Message properties and the underlying transport protocol stack.

A Carrier that is backed by current transport protocol stack state (such as a TCP connection; see Section 2.7) is said to be "active": messages can be sent and received over it. A Carrier can also be "dormant": there is long-term state associated with it (via the underlying Association; see Section 2.3), and it may be able to reactivated, but messages cannot be sent and received immediately.

If supported by the underlying transport protocol stack, a Carrier may be forked: creating a new Carrier associated with a new Carrier at the same remote endpoint. The semantics of the usage of multiple Carriers based on the same Association are application-specific. When a Carrier is forked, its corresponding Carrier at the remote endpoint receives a fork request, which it must accept in order to fully establish the new carrier. Multiple Carriers between endpoints are implemented differently by different transport protocol stacks, either using multiple separate transport-layer connections, or using multiple streams of multistreaming transport protocols.

To exchange messages with a given remote endpoint, an application may initiate a Carrier given its remote (see Section 2.4 and local (see Section 2.5) identities; this is an equivalent to an active open. There are four special cases of Carriers, as well, supporting different initiation and interaction patterns, defined in the subsections below.

o Listener: A Listener is a special case of Message Carrier which only responds to requests to create a new Carrier from a remote endpoint, analogous to a server or listening socket in the present sockets API. Instead of being bound to a specific remote endpoint, it is bound only to a local identity; however, its interface for accepting fork requests is identical to that for fully fledged Carriers.

o Source: A Source is a special case of Message Carrier over which messages can only be sent, intended for unidirectional applications such as multicast transmitters. Sources cannot be forked, and need not accept forks.

- Sink: A Sink is a special case of Message Carrier over which messages can only be received, intended for unidirectional applications such as multicast receivers. Sinks cannot be forked, and need not accept forks.

- Responder: A Responder is a special case of Message Carrier which may receive messages from many remote sources, for cases in which an application will only ever send Messages in reply back to the source from which a Message was received. This is a common implementation pattern for servers in client-server applications. A Responder's receiver gets a Message, as well as a Source to send replies to. Responders cannot be forked, and need not accept forks.

2.2. Message

A Message is the unit of communication between applications. Messages can represent relatively small structures, such as requests in a request/response protocol such as HTTP; relatively large structures, such as files of arbitrary size in a filesystem; and structures of indeterminate length, such as a stream of bytes in a protocol like TCP.

In the general case, there is no mapping between a Message and packets sent by the underlying protocol stack on the wire: the transport protocol may freely segment messages and/or combine messages into packets. However, a message may be marked as immediate, which will cause it to be sent in a single packet when possible.

Content may be sent and received either as Complete or Partial Messages. Dealing with Complete Messages should be preferred for simplicity whenever possible based on the underlying protocol. It is always possible to send Complete Messages, but only protocols that have a fixed maximum message length may allow clients to receive Messages using an API that guarantees Complete Messages. Sending and receiving Partial Messages (that is, a Message whose content spans multiple calls or callbacks) is always possible.

To send a Message, either Complete or Partial, the Message content is passed into the Carrier, and client provides a set of callbacks to know when the Message was delivered or acknowledged. The client of the API may use the callbacks to pace the sending of Messages.

To receive a Message, the client of the API schedules a completion to be called when a Complete or Partial Message is available. If the client is willing to accept Partial Messages, it can specify the minimum incomplete Message length it is willing to receive at once, and the maximum number of bytes it is willing to receive at once. If the client wants Complete Messages, there are no values to tune. The scheduling of the receive completion indicates to the Carrier that there is a desire to receive bytes, effectively creating a "pull model" in which backpressure may be applied if the client is not receiving Messages or Partial Messages quickly enough to match the peer's sending rate. The Carrier may have some minimal buffer of incoming Messages ready for the client to read to reduce latency.

When receiving a Complete Message, the entire content of the Message must be delivered at once, and the Message is not delivered at all if the full Message is not received. This implies that both the sending and receiving endpoint, whether in the application or the carrier, must guarantee storage for the full size of a Message.

Partial Messages may be sent or received in several stages, with a handle representing the total Message being associated with each portion of the content. Each call to send or receive also indicates whether or not the Message is now complete. This approach is necessary whenever the size of the Message does not have a known bound, or the size is too large to process and hold in memory. Protocols that only present a concept of byte streams represent their data as single Messages with unknown bounds. In the case of TCP, the client will receive a single Message in pieces using the Partial Message API, and that Message will only be marked as complete when the peer has sent a FIN.

Messages are sent over and received from Message Carriers (see Section 2.1).

On sending, Messages have properties that allow the application to specify its requirements with respect to reliability, ordering, priority, idempotence, and immediacy; these are described in detail below. Messages may also have arbitrary properties which provide additional information to the underlying transport protocol stack on how they should be handled, in a protocol-specific way. These stacks may also deliver or set properties on received messages, but in the general case a received messages contains only a sequence of ordered bytes. Message properties include:

- Lifetime and Partial Reliability: A Message may have a "lifetime" - a wall clock duration before which the Message must be available to the application layer at the remote end. If a lifetime cannot be met, the Message is discarded as soon as possible. Messages without lifetimes are sent reliably if supported by the transport protocol stack. Lifetimes are also used to prioritize Message delivery.

There is no guarantee that a Message will not be delivered after the end of its lifetime; for example, a Message delivered over a strictly reliable transport will be delivered regardless of its lifetime. Depending on the transport protocol stack used to transmit the message, these lifetimes may also be signalled to path elements by the underlying transport, so that path elements that realize a lifetime cannot be met can discard frames containing the Messages instead of forwarding them.

- Priority: Messages have a "niceness" - a priority among other messages sent over the same Carrier in an unbounded hierarchy most naturally represented as a non-negative integer. By default, Messages are in niceness class 0, or highest priority. Niceness class 1 Messages will yield to niceness class 0 Messages sent over the same Carrier, class 2 to class 1, and so on. Niceness may be translated to a priority signal for exposure to path elements (e.g. DSCP code point) 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 lifetime decrease. A Message may have both a niceness and a lifetime - Messages with higher niceness classes will yield to lower classes if resource constraints mean only one can meet the lifetime.

- Dependence: A Message may have "antecedents" - other Messages 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 Message down which Path.

- Idempotence: A sending application may mark a Message as "idempotent" to signal to the underlying transport protocol stack that its application semantics make it safe to send in situations that may cause it to be received more than once (i.e., for 0-RTT session resumption as in TCP Fast Open, TLS 1.3, and QUIC).

- Immediacy: A sending application may mark a Message as "immediate" to signal to the underlying transport protocol stack that its application semantics require it to be placed in a single packet, on its own, instead of waiting to be combined with other messages or parts thereof (i.e., for media transports and interactive sessions with small messages).

Senders may also be asynchronously notified of three events on Messages they have sent: that the Message has been transmitted, that the Message has been acknowledged by the receiver, or that the

Message has expired before transmission/acknowledgement. Not all transport protocol stacks will support all of these events.

2.3. Association

An Association contains the long-term state necessary to support communications between a Local (see Section 2.5) and a Remote (see Section 2.4) endpoint, such as trust model information, including pinned public keys or anchor certificates, cryptographic session resumption parameters, or rendezvous information. It uses information from the Policy Context (see Section 2.6) to constrain the selection of transport protocols and local interfaces to create Transients (see Section 2.7) to carry Messages; and information about the paths through the network available available between them (see Section 2.8).

All Carriers are bound to an Association. New Carriers will reuse an Association if they can be carried from the same Local to the same Remote over the same Paths; this re-use of an Association may implies the creation of a new Transient.

Associations may exist and be created without a Carrier. This may be done if peer cryptographic state such as a pre-shared key is established out-of-band. Thus, Associations may be created without the need to send application data to a peer, that is, without a Carrier. Associations are mutable. Association state may expire over time, after which it is removed from the Association, and Transients may export cryptographic state to store in an Association as needed. Moreover, this state may be exported directly into the Association or modified before insertion. This may be needed to diversify ephemeral Transient keying material from the longer-term Association keying material.

A primary use of Association state is to allow new Associations and their derived Carriers to be quickly created without performing in-band cryptographic handshakes. See [I-D.kuehlewind-taps-crypto-sep] for more details about this separation.

2.4. Remote

A Remote represents information required to establish and maintain a connection with the far end of an Association: name(s), address(es), and transport protocol parameters that can be used to establish a Transient; transport protocols to use; trust model information, inherited from the relevant Association, used to identify the remote on connection establishment; and so on. Each Association is associated with a single Remote, either explicitly by the application (when created by the initiation of a Carrier) or a Listener (when created by forking a Carrier on passive open).

A Remote may be resolved, which results in zero or more Remotes with more specific information. For example, an application may want to establish a connection to a website identified by a URL https://www.example.com. This URL would be wrapped in a Remote and passed to a call to initiate a Carrier. The first pass resolution might parse the URL, decomposing it into a name, a transport port, and a transport protocol to try connecting with. A second pass resolution would then look up network-layer addresses associated with that name through DNS, and store any certificates available from DANE. Once a Remote has been resolved to the point that a transport protocol stack can use it to create a Transient, it is considered fully resolved.

2.5. Local

A Local represents all the information about the local endpoint necessary to establish an Association or a Listener: interface, port, and transport protocol stack information, and, per [I-D.pauly-taps-transport-security], cryptographic identities (certificates and associated private keys) bound to this endpoint.

2.6. Policy Context

The Policy Context describes preferences for, and restrictions on, how to configure Transients to support communication between a Local and a Remote over one or more Paths between endpoints. For instance, an application may require, or prefer to use, certain features (see [I-D.ietf-taps-transports]) of the transport protocol stacks used by the Transients underlying the Carrier. Alternatively, it might also prefer Paths over one interface to those over another (e.g., WiFi access over LTE when roaming on a foreign LTE network, due to cost).

These policies are expressed in the Policy Context(s) that are bound to the Association. Multiple policy contexts can be active at once. For example, a system Policy Context can express the administrative preferences around network interface and protocol selection, while an application Policy Context expresses preferences for use of different transport services. Expression of policy contexts and the resolution of conflicts among Policy Contexts is currently implementation-specific (the Policy API in the NEAT architecture [NEAT] provides an example of how this can be done).

2.7. Transient

A Transient represents a binding between a Carrier and the instance of the transport protocol stack that implements it. As an Association contains long-term state for communications between two endpoints, a Transient contains ephemeral state for a single transport protocol over a single Path at a given point in time.

A Carrier may be served by multiple Transients at once, e.g. when implementing multipath communication such that the separate paths are exposed to the API by the underlying transport protocol stack. Each Transient serves only one Carrier, although multiple Transients may share the same underlying protocol stack; e.g. when multiplexing Carriers over streams in a multistreaming protocol.

Transients are generally not exposed by the API to the application, though they may be accessible for debugging and logging purposes.

2.8. Path

A Path represents information about a single path through the network used by an Association, in terms of source and destination network and transport layer addresses within an addressing context, and the provisioning domain [RFC7556] of the local interface. This information may be learned through a resolution, discovery, or rendezvous process (e.g. DNS, ICE), by measurements taken by the transport protocol stack, or by some other path information discovery mechanism. It is used by the transport protocol stack to maintain and/or (re-)establish communications for the Association.

The set of available properties is a function of the transport protocol stacks in use by an association. However, the following core properties are generally useful for applications and transport layer protocols to choose among paths for specific Messages:

- Maximum Transmission Unit (MTU): the maximum size of an Message’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 protocol stack.

- 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 can be met by having multiple paths with different properties to select from. Transport protocol stacks can also provide signaling to devices along the path, but this signaling is derived from information provided to the Message abstraction.

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.

- 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 Message 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 handling will need to be aligned to the method a given platform provides for asynchronous I/O.

- 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.

The API we design from these principles is centered around a Carrier, which can be created actively via initiate() or passively via a listen(); the latter creates a Listener from which new Carriers can be accept()ed. Messages may be created explicitly and passed to this Carrier, or implicitly through a simplified interface which uses default message properties (reliable transport without priority or deadline, which guarantees ordered delivery over a single Carrier when the underlying transport protocol stack supports it).

For each connection between a Local and a Remote a new Carrier is created and destroyed when the connection is closed. However, a new Carrier may use an existing Association if present for the requested Local-Remote pair and permitted by the PolicyContext that can be provided at Carrier initiation. Further the system-wide PolicyContext can contain more information that determine when to create or destroy Associations other than at Carrier initiation. E.g. an Association can be created at system start, based on the configured PolicyContext or also by a manual action of an single application, for Local-Remote pairs that are known to be likely used soon, and to pre-establish, e.g., cryptographic context as well as potentially collect current information about path capabilities. Every time an actual connection with a specific PSI is established between the Local and Remote, the Association learns new Path information and stores them. This information can be used when a new transient is created, e.g. to decide which PSI to use (to provide the highest probably for a successful connection attempt) or which PSIs to probe for (first). A Transient is created when an application actually sends a Message over a Carrier. As further explained below this step can actually create multiple transients for probing or assign a new transient to an already active PSI, e.g. if multi-streaming is provided and supported for these kind of use on both sides.

3.1. Example Connection Patterns

Here, we illustrate the usage of the API for common connection patterns. Note that error handling is ignored in these illustrations for ease of reading.

3.1.1. Client-Server

Here's an example client-server application. The server echoes messages. The client sends a message and prints what it receives.

The client in Figure 2 connects, sends a message, and sets up a receiver to print messages received in response. The carrier is inactive after the Initiate() call; the Send() call blocks until the carrier can be activated.

```
// connect to a server given a remote
func sayHello() {

    carrier := Initiate(local, remote)

    carrier.Send([]byte("Hello!"))
    carrier.Ready(func (msg InMessage) {
        fmt.Println(string([]byte(msg))
        return false
    })
    carrier.Close()
}
```

Figure 2: Example client

The server in Figure 3 creates a Listener, which accepts Carriers and passes them to a server. The server echos the content of each message it receives.

```
// run a server for a specific carrier, echo all its messages
func runMyServerOn(carrier Carrier) {
    carrier.Ready(func (msg InMessage) {
        carrier.Send(msg)
    })
}

// accept connections forever, spawn servers for them
func acceptConnections() {
    listener := Listen(local)
    listener.Accept(func(carrier Carrier) bool {
        go runMyServerOn(carrier)
        return true
    })
}
```

Figure 3: Example server

The Responder allows the server to be significantly simplified, as shown in Figure 4.

```
   func echo(msg InMessage, reply Sink) {
       reply.Send(msg)
   }

   Respond(local, echo)
```

Figure 4: Example responder

3.1.2. Client-Server with Happy Eyeballs and 0-RTT establishment

The fundamental design of a client need not change at all for happy eyeballs [RFC6555] (selection of multiple potential protocol stacks through connection racing); this is handled by the Post Sockets implementation automatically. If this connection racing is to use 0-RTT data (i.e., as provided by TCP Fast Open [RFC7413], the client must mark the outgoing message as idempotent.

```
// connect to a server given a remote and send some 0-RTT data
func sayHelloQuickly() {

    carrier := Initiate(local, remote)

    carrier.SendMsg(OutMessage{Content: []byte("Hello!"), Idempotent: true}, nil,
    carrier.Ready(func (msg InMessage) {
        fmt.Println(string([]byte(msg)))
        return false
    })
    carrier.Close()
}
```

3.1.3. Peer to Peer with Network Address Translation

In the client-server examples shown above, the Remote given to the Initiate call refers to the name and port of the server to connect to. This need not be the case, however; a Remote may also refer to an identity and a rendezvous point for rendezvous as in ICE [RFC5245]. Here, each peer does its own Initiate call simultaneously, and the result on each side is a Carrier attached to an appropriate Association.

3.1.4. Multicast Receiver

A multicast receiver is implemented using a Sink attached to a Local encapsulating a multicast address on which to receive multicast datagrams. The following example prints messages received on the multicast address forever.

```
func receiveMulticast() {
    sink = NewSink(local)
    sink.Ready(func (msg InMessage) {
        fmt.Println(string([]byte(msg)))
        return true
    })
}
```

3.2. Association Bootstrapping

Here, we show how Association state may be initialized without a carrier. The goal is to create a long-term Association from which Carriers may be derived and, if possible, used immediately. Per [I-D.pauly-taps-transport-security], a first step is to specify trust model constraints, such as pinned public keys and anchor certificates, which are needed to create Remote connections.

We begin by creating shared security parameters that will be used later for creating a remote connection.

```
// create security parameters with a set of trusted certificates
func createParameters(trustedCerts []Certificate) Parameters {
    parameters := Parameters()
    parameters = parameters.SetTrustedCerts(trustedCerts)
    return parameters
}
```

Using these statically configured parameters, we now show how to create an Association between a Local and Remote using these parameters.

```
// create an Association using shared parameters
func createAssociation(local Local, remote Remote, parameters Parameters) Associa
    association := AssociationWithParameters(local, remote, parameters)
    return association
}
```

We may also create an Association with a pre-shared key configured out-of-band.

```
// create an Association using a pre-shared key
func createAssociationWithPSK(local Local, remote Remote, parameters Parameters,
    association := AssociationWithParameters(local, remote, parameters)
    association = association.SetPreSharedKey(preSharedKey)
    return association
}
```

We now show how to create a Carrier from an existing, pre-configured Association. This Association may or may not contain shared cryptographic static between the Local and Remote, depending on how it was configured.

```
// open a connection to a server using an existing Association and send some data
// which will be sent early if possible.
func sayHelloWithAssociation(association Association) {
    carrier := InitiateWithAssociation(association)

    carrier.SendMsg(OutMessage{Content: []byte("Hello!"), Idempotent: true}, nil,
    carrier.Ready(func (msg InMessage) {
        fmt.Println(string([]byte(msg)))
        return false
    })
    carrier.Close()
}
```

4. Implementation Considerations

Here we discuss an incomplete list of API implementation considerations that have arisen with experimentation with prototype implementations of Post.

4.1. Protocol Stack Instance (PSI)

A PSI encapsulates an arbitrary stack of protocols (e.g., TCP over IPv6, SCTP over DTLS over UDP over IPv4). PSIs provide the bridge between the interface (Carrier) plus the current state (Transients) and the implementation of a given set of transport services [I-D.ietf-taps-transports].

A given implementation makes one or more possible protocol stacks available to its applications. Selection and configuration among multiple PSIs is based on system-level or application policies, as well as on network conditions in the provisioning domain in which a connection is made.

```
   +=========+    +=========+   +==========+      +==========+
   | Carrier |    | Carrier |   | Carrier  |      | Carrier  |
   +=========+    +=========+   +==========+      +==========+
        |               |              |                 |
   +=========+    +=========+   +==========+      +==========+
   |Transient|    |Transient|   |Transient |      |Transient |
   +=========+    +=========+   +==========+      +==========+
        |                  \     /                 /        \
   +=========+           +=========+      +=========+      +=========+
   |   PSI   |           |   PSI   |      |   PSI   |      |   PSI   |
   +===+-----++          +===+-----++     +===+-----++    ++-----+===+
       |TLS   |              |SCTP  |         |TLS   |    |   TLS|
       |TCP   |              |DTLS  |         |TCP   |    |   TCP|
       |IPv6  |              |UDP   |         |IPv6  |    |  IPv4|
       |802.3 |              |IPv6  |         |802.11|    |802.11|
       +------+              |802.3 |         +------+    +------+
                             +------+
   (a) Transient  (b) Carrier multiplexing   (c) Multiple candidates
    bound to PSI   over a multi-streaming     racing during session
                   transport protocol         establishment
```

Figure 5: Example Protocol Stack Instances

For example, Figure 5(a) shows a TLS over TCP stack, usable on most network connections. Protocols are layered to ensure that the PSI provides all the transport services required by the application. A single PSI may be bound to multiple Carriers, as shown in Figure 5(b): a multi-streaming transport protocol like QUIC or SCTP can support one carrier per stream. Where multi-streaming transport is not available, these carriers could be serviced by different PSIs on different flows. On the other hand, multiple PSIs are bound to a single transient during establishment, as shown in Figure 5(c). Here, the losing PSI in a happy-eyeballs race will be terminated, and the carrier will continue using the winning PSI.

4.2. 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, Post Sockets exposes a message-based API to applications 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.

There are other benefits of providing a message-oriented API beyond framing PDUs that Post Sockets should provide when supported by the underlying transport. These include:

- 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 Post Sockets, 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. Post Sockets provides this by handling message, 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, Post Sockets must provide APIs for parsing and serialising messages that understand the protocol data. That is, we push message parsing and serialisation down into the Post Sockets stack, allowing applications to send and receive strongly typed data objects (e.g., a receive call on an HTTP Message Carrier should return an object representing the HTTP response, with pre-parsed status code, headers, and any message body, rather than returning a byte array that the application has to parse itself). This is backwards compatible with existing protocols and APIs, since the wire format of messages does not change, but gives a Post Sockets stack additional information to allow it to make better use of modern transport services.

The Post Sockets approach is therefore to raise the semantic level of the transport API: applications should send and receive messages in the form of meaningful, strongly typed, protocol data. Parsing and serialising such messages should be a re-usable function of the protocol stack instance not the application. This is well-suited to implementation in modern systems languages, such as Swift, Go, Rust, or C++, but can also be implemented with some loss of type safety in C.

4.3. Message Size Limitations

Ideally, Messages can be of infinite size. However, protocol stacks and protocol stack implementations may impose their own limits on message sizing; For example, SCTP [RFC4960] and TLS [I-D.ietf-tls-tls13] impose record size limitations of 64kB and 16kB, respectively. Message sizes may also be limited by the available buffer at the receiver, since a Message must be fully assembled by the transport layer before it can be passed on to the application layer. Since not every transport protocol stack implements the signaling necessary to negotiate or expose message size limitations, these are currently configured out of band, and are probably best exposed through the policy context.

A truly infinite message service - e.g. large file transfer where both endpoints have committed persistent storage to the message - is probably best realized as a layer above Post Sockets, and may be added as a new type of Message Carrier to a future revision of this document.

4.4. Back-pressure

Regardless of how asynchronous reception is implemented, it is important for an application to be able to apply receiver back-pressure, to allow the protocol stack to perform receiver flow control. Depending on how asynchronous I/O works in the platform, this could be implemented by having a maximum number of concurrent receive callbacks, or by bounding the maximum number of outstanding, unread bytes at any given time, for example.

4.5. Associations, Transients, Racing, and Rendezvous

As the network has evolved, even the simple act of establishing a connection has become increasingly complex. Clients now regularly race multiple connections, for example over IPv4 and IPv6, to determine which protocol to use. The choice of outgoing interface has also become more important, with differential reachability and performance from multiple interfaces. Name resolution can also give different outcomes depending on the interface the query was issued from. Finally, but often most significantly, NAT traversal, relay discovery, and path state maintenance messages are an essential part of connection establishment, especially for peer-to-peer applications.

Post Sockets accordingly breaks communication establishment down into multiple phases:

- Gathering Locals

  The set of possible Locals is gathered. In the simple case, this merely enumerates the local interfaces and protocols, and allocates ephemeral source ports for transients. For example, a system that has WiFi and Ethernet and supports IPv4 and IPv6 might gather four candidate locals (IPv4 on Ethernet, IPv6 on Ethernet, IPv4 on WiFi, and IPv6 on WiFi) that can form the source for a transient.

  If NAT traversal is required, the process of gathering locals becomes broadly equivalent to the ICE candidate gathering phase [RFC5245]. The endpoint determines its server reflexive locals (i.e., the translated address of a local, on the other side of a NAT) and relayed locals (e.g., via a TURN server or other relay), for each interface and network protocol. These are added to the set of candidate locals for this association.

  Gathering locals is primarily an endpoint local operation, although it might involve exchanges with a STUN server to derive server reflexive locals, or with a TURN server or other relay to derive relayed locals. It does not involve communication with the remote.

- Resolving the Remote

  The remote is typically a name that needs to be resolved into a set of possible addresses that can be used for communication. Resolving the remote is the process of recursively performing such name lookups, until fully resolved, to return the set of candidates for the remote of this association.

  How this is done will depend on the type of the Remote, and can also be specific to each local. A common case is when the Remote is a DNS name, in which case it is resolved to give a set of IPv4 and IPv6 addresses representing that name. Some types of remote might require more complex resolution. Resolving the remote for a peer-to-peer connection might involve communication with a rendezvous server, which in turn contacts the peer to gain consent to communicate and retrieve its set of candidate locals, which are returned and form the candidate remote addresses for contacting that peer.

Resolving the remote is _not_ a local operation. It will involve a directory service, and can require communication with the remote to rendezvous and exchange peer addresses. This can expose some or all of the candidate locals to the remote.

- Establishing Transients

  The set of candidate locals and the set of candidate remotes are paired, to derive a priority ordered set of Candidate Paths that can potentially be used to establish a connection.

  Then, communication is attempted over each candidate path, in priority order. If there are multiple candidates with the same priority, then transient establishment proceeds simultaneously and uses the transient that wins the race to be established. Otherwise, transients establishment is sequential, paced at a rate that should not congest the network. Depending on the chosen transport, this phase might involve racing TCP connections to a server over IPv4 and IPv6 [RFC6555], or it could involve a STUN exchange to establish peer-to-peer UDP connectivity [RFC5245], or some other means.

- Confirming and Maintaining Transients

  Once connectivity has been established, unused resources can be released and the chosen path can be confirmed. This is primarily required when establishing peer-to-peer connectivity, where connections supporting relayed locals that were not required can be closed, and where an associated signalling operation might be needed to inform middleboxes and proxies of the chosen path. Keep-alive messages may also be sent, as appropriate, to ensure NAT and firewall state is maintained, so the transient remains operational.

By encapsulating these four phases of communication establishment into the PSI, Post Sockets aims to simplify application development. It can provide reusable implementations of connection racing for TCP, to enable happy eyeballs, that will be automatically used by all TCP clients, for example. With appropriate callbacks to drive the rendezvous signalling as part of resolving the remote, we believe a generic ICE implementation ought also to be possible. This procedure can even be repeated fully or partially during a connection to enable seamless hand-over and mobility within the network stack.

5. 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. Thanks to Joe Hildebrand, Martin Thomson, and Michael Welzl for their feedback, as well as the attendees of the Post Sockets workshop in February 2017 in Zurich for the discussions, which have improved the design described herein.

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.

6. References

6.1. Normative References

[I-D.ietf-taps-transports]
Fairhurst, G., Trammell, B., and M. Kuehlewind, "Services provided by IETF transport protocols and congestion control mechanisms", draft-ietf-taps-transports-14 (work in progress), December 2016.

6.2. Informative References

[I-D.ietf-quic-transport]
Iyengar, J. and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport", draft-ietf-quic-transport-05 (work in progress), August 2017.

[I-D.ietf-tls-tls13]
Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", draft-ietf-tls-tls13-21 (work in progress), July 2017.

[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.kuehlewind-taps-crypto-sep]
Kuehlewind, M., Pauly, T., and C. Wood, "Separating Crypto Negotiation and Communication", draft-kuehlewind-taps-crypto-sep-00 (work in progress), July 2017.

[I-D.pauly-taps-transport-security]
Pauly, T. and C. Wood, "A Survey of Transport Security Protocols", draft-pauly-taps-transport-security-00 (work in progress), July 2017.

[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-03 (work in progress), March 2017.

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

[NEAT] Grinnemo, K-J., Tom Jones, ., Gorry Fairhurst, ., David Ros, ., Anna Brunstrom, ., and . Per Hurtig, "Towards a Flexible Internet Transport Layer Architecture", June 2016.

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

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

[RFC5245] Rosenberg, J., "Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols", RFC 5245, DOI 10.17487/RFC5245, April 2010, <https://www.rfc-editor.org/info/rfc5245>.

[RFC6555] Wing, D. and A. Yourtchenko, "Happy Eyeballs: Success with Dual-Stack Hosts", RFC 6555, DOI 10.17487/RFC6555, April 2012, <https://www.rfc-editor.org/info/rfc6555>.

[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, <https://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, <https://www.rfc-editor.org/info/rfc7258>.

[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>.

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

Appendix A. Open Issues

This document is under active development; a list of current open issues is available at https://github.com/mami-project/draft-trammell-post-sockets/issues

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@csperkins.org

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

Chris Wood
Apple Inc.
1 Infinite Loop
Cupertino, California 95014
United States of America

Email: cawood@apple.com