TAPS Working Group
Internet-Draft
Intended status: Informational
Expires: September 9, 2017

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

March 08, 2017

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

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

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 September 9, 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, 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. Applications also need control over cooperation with path elements via mechanisms such as that 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.ietf-quic-transport]) to mitigate this strict layering.

To meet these challenges, we present 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 working group.

Post replaces the traditional SOCK_STREAM abstraction with an 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, these Carriers can also be opened as Streams, presenting a file-like interface to the network as with SOCK_STREAM.

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 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. 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

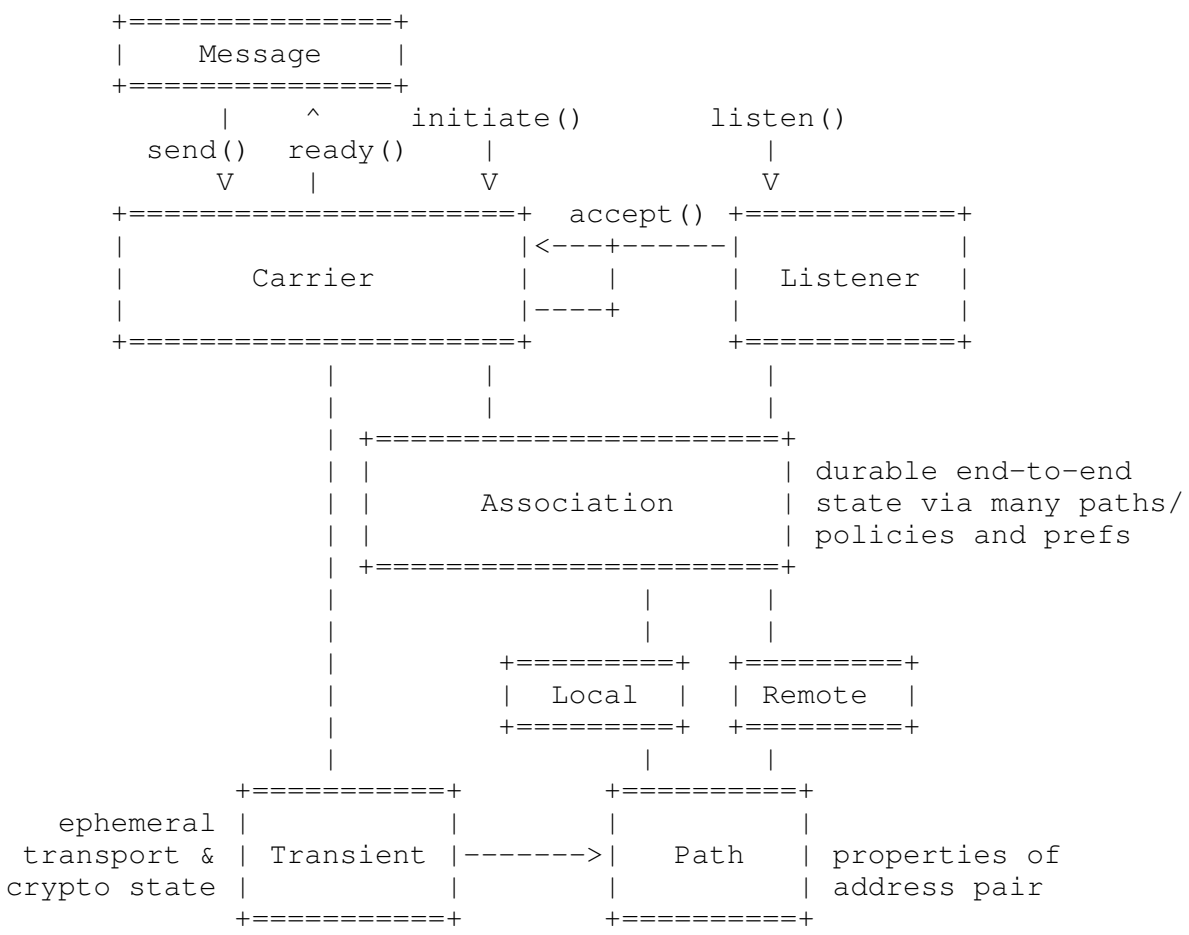

Figure 1: Abstractions and relationships in Post Sockets

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

A Message Carrier that is backed by current transport protocol stack state (such as a TCP connection; see Section 2.6) is said to be "active": messages can be sent and received over it. A Message 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 Message Carrier may be forked: creating a new Message Carrier associated with a new Message Carrier at the same remote endpoint. The semantics of the usage of multiple Message Carriers based on the same Association are application-specific. When a Message Carrier is forked, its corresponding Message Carrier at the remote endpoint receives a fork request, which it must accept in order to fully establish the new carrier. Multiple message 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 Message 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 five special cases of Message Carriers, as well, supporting different initiation and interaction patterns, defined in the subsections below.

2.1.1. 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 Message Carriers.

2.1.2. 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.

2.1.3. 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.

2.1.4. 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.

2.1.5. Stream

A Message Carrier may be irreversibly morphed into a Stream, in order to provide a strictly ordered, reliable service as with SOCK_STREAM. Morphing a Message Carrier into a Stream should return a "file-like object" as appropriate for the platform implementing the API. Typically, both ends of a communication using a stream service will morph their respective Message Carriers independently before sending any Messages.

Writing a byte to a 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. A Message Carrier may only be morphed to a Stream if it uses transport protocol stack that provides reliable, ordered service, and only before it is used to send a Message.

2.2. Message

A Message is an atomic unit of communication between applications. A Message 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.

Messages can represent both relatively small structures, such as requests in a request/response protocol such as HTTP; as well as relatively large structures, such as files of arbitrary size in a filesystem.

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, if it will fit.

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 Message.

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.

2.2.1. Lifetime and Partial Reliability

A Message may have a "lifetime" - a wallclock 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 signaled 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.

2.2.2. Priority

Messages have a "niceness" - a priority among other messages sent over the same Message 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 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 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.

2.2.3. 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.

2.2.4. 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).

2.2.5. 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).

2.2.6. Additional Events

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/acknowledgment. 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 cryptographic session resumption parameters or rendezvous information; information about the policies constraining the selection of transport protocols and local interfaces to create Transients (see Section 2.6) to carry Messages; and information about the paths through the network available available between them (see Section 2.7).

All Message Carriers are bound to an Association. New Message 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.

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; information about public keys or certificate authorities 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 Message Carrier) or a Listener (when created by forking a Message 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 Message 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, as well as certificates and associated private keys to use to identify this endpoint.

2.6. Transient

A Transient represents a binding between a Message 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 Message 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 Message 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.

2.7. 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.

2.8. Policy Context

A Local and a Remote is not necessarily enough to establish a Message Carrier between two endpoints. For instance, an application may require or prefer certain transport features (see [I-D.ietf-taps-transports]) in the transport protocol stacks used by the Transients underlying the Carrier; it may 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 a Policy Context bound to an Association. Multiple policy contexts may be active at once; e.g. a system Policy Context expressing administrative preferences about interface and protocol selection, an application Policy Context expressing transport feature information. The expression of policy contexts and the resolution of conflicts among Policy Contexts is currently implementation-specific;

note that these are equivalent to the Policy API in the NEAT architeture [NEAT].

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).

The current state of API development is illustrated as a set of interfaces and function prototypes in the Go programming language in Appendix A; future revisions of this document will give more a more abstract specification of the API as development completes.

3.1. Example Connection Patterns

Here, we illustrate the usage of the API outlined in Appendix A 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
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. Implementation Considerations

Here we discuss an incomplete list of API implementation considerations that have arisen with experimentation with the prototype in Appendix A.

3.2.1. Message Framing and Deframing

An obvious goal of Post Sockets is interoperability with non-Post Sockets endpoints: a Post Sockets endpoint using a given protocol stack must be able to communicate with another endpoint using the same protocol stack, but not using Post Sockets. This implies that the underlying transport protocol stack must support object framing, in order to delimit Messages carried by protocol stacks that are not themselves message-oriented.

Another goal of Post Sockets is to work over unmodified TCP. We could simply define a Message Carrier over TCP to support only stream morphing, but this would fall far short of our goal to transport independence. Another approach is to recognize that almost every protocol using TCP already has its own message delimiters, and to allow the receiver of a Message to provide a deframing primitive to the API. Experimentation with the best way to achieve this within Post Sockets is underway.

3.2.2. 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.

3.2.3. Backpressure

Regardless of how asynchronous reception is implemented, it is important for an application to be able to apply receiver backpressure, 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, for example.

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. 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.

5. References

5.1. 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.

5.2. Informative References

[I-D.ietf-quic-transport]
Iyengar, J. and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport", draft-ietf-quic-transport-01 (work in progress), January 2017.

[I-D.ietf-tls-tls13]
Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", draft-ietf-tls-tls13-18 (work in progress), October 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-02 (work in progress), December 2016.

[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, <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>.

[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, <http://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, <http://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, <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>.

[RFC7413] Cheng, Y., Chu, J., Radhakrishnan, S., and A. Jain, "TCP Fast Open", RFC 7413, DOI 10.17487/RFC7413, December 2014, <http://www.rfc-editor.org/info/rfc7413>.

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

Appendix A. API sketch in Golang

The following sketch is a snapshot of an API currently under development in Go, available at https://github.com/mami-project/postsocket. The details of the API are still under development; once the API definition stabilizes, this will be expanded into prose in a future revision of this draft.

```
// The interface to path information is TBD
type Path interface{}

// An association encapsulates an endpoint pair and the set of paths between them
type Association interface {
    Local() Local
    Remote() Remote
    Paths() []Path
}

// A message together with with metadata needed to send it
type OutMessage struct {
    // The content of this message, as a byte array
    Content []byte
    // The niceness of this message. 0 is highest priority.
    Niceness uint
    // The lifetime of this message. After this duration, the message may expire.
    Lifetime time.Duration
    // Pointers to messages that must be sent before this one.
    Antecedent []*OutMessage
    // True if the message is safe to send such that it may be received multiple
    Idempotent bool
}

// A message received from a stream
type InMessage []byte

// A 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.
type Carrier interface {
    // Send a byte array on this Carrier as a message with default metadata
    // and no notifications.
    Send(buf []byte) error

    // Send a message on this Carrier. The optional onSent function will be
```

```
    // called when the protocol stack instance has sent the message. The
    // optional onAcked function will be called when the receiver has
    // acknowledged the message. The optional onExpired function will be
    // called if the message's lifetime expired before the message coult be
    // sent. If the Carrier is not active, attempt to activate the Carrier
    // before sending.
    Sendmsg(msg *OutMessage, onSent func(), onAcked func(), onExpired func()) err

    // Signal that an application is ready to receive messages via a given callba
    // Messages will be given to the callback until it returns false, or until th
    // Carrier is closed.
    Ready(receive func(InMessage) bool) error

    // Retrieve the Association over which this Carrier is running.
    Association() *Association

    // Retrieve the active Transients over which this carrier is running, if acti
    Transients() []Transient

    // Determine whether the Carrier is currently active
    IsActive() bool

    // Ensure that the Carrier is active and ready to send and receive messages.
    // Attempts to bring up at least one Transient.
    Activate(isActive func()) error

    // Terminate the Carrier
    Close()

    // Mutate to a file-like object
    AsStream() io.ReadWriteCloser

    // Attempt to fork a new Carrier for communicating with the same Remote
    Fork() (Carrier, error)

    // Signal that an application is ready to accept forks via a given callback.
    // Forked carriers will be given to the callback until it returns false or
    // until the Carrier is closed.
    Accept(accept func(Carrier) bool) error
}

// Initiate a Carrier from a given Local to a given Remote. Returns a new
// Carrier, which may be bound to an existing or a new Association. The
// initiated Carrier is not yet active.
func Initiate(local Local, remote Remote) (Carrier, error)

type Listener interface {
    // Signal that an application is ready to accept forks via a given callback.
```

```
    // Accept will terminate when the callback returns false, or until the
    // Listener is closed.
    Accept(accept func(Carrier) bool) error

    // Terminate this Listener
    Close()
}

// Create a Listener on a given Local which will pass new Carriers to the
// given channel until that channel is closed.
func Listen(local Local) (Listener, error)

// A Source is a unidirectional, send-only Carrier.
type Source interface {
    // Send a byte array on this Source as a message with default metadata
    // and no notifications.
    Send(buf []byte) error

    // Send a message on this Source. The optional onSent function will be
    // called when the protocol stack instance has sent the message. The
    // optional onAcked function will be called when the receiver has
    // acknowledged the message. The optional onExpired function will be
    // called if the message's lifetime expired before the message coult be
    // sent. If the Source is not active, attempt to activate the Source
    // before sending.
    Sendmsg(msg *OutMessage, onSent func(), onAcked func(), onExpired func()) err

    // Retrieve the Association over which this Source is running.
    Association() *Association

    // Determine whether the Source is currently active
    IsActive() bool

    // Ensure that the Source is active and ready to send messages.
    // Attempts to bring up at least one Transient.
    Activate() error

    // Terminate the Source
    Close()
}

// Initiate a Source from a given Local to a given Remote. Returns a new
// Source, which may be bound to an existing or a new Association. The
// initiated Source is not yet active.
func NewSource(local Local, remote Remote) (Source, error)

// A Sink is a unidirectional, receive-only Carrier, bound only to a local.
type Sink interface {
```

```
    // Signal that an application is ready to receive messages via a given callba
    // Messages will be given to the callback until it returns false, or until th
    // Sink is closed.
    Ready(receive func(InMessage) bool) error

    // Retrieve the Association over which this Sink is running.
    Association() *Association

    // Terminate the Sink
    Close()
}

// Initiate a Sink on a given Local. Returns a new
// Sink, which may be bound to an existing or a new Association.
func NewSink(local Local) (Sink, error)

// Initiate a Responder on a given Local. For each incoming Message, calls the
// respond function with the Message and a Sink to send replies to. Calls the
// Responder until it returns False, then terminates
func Respond(local Local, respond func(msg InMessage, reply Sink) bool) error

// A local identity
type Local struct {
    // A string identifying an interface or set of interfaces to accept messages
    Interface string
    // A transport layer port
    Port int
    // A set of zero or more end entity certificates, together with private
    // keys, to identify this application with.
    Certificates []tls.Certificate
}

// Encapsulate a remote identity. Since the contents of a Remote are highly
// dependent on its level of resolution; some examples are below.
type Remote interface {
    // Resolve this Remote Identity to a
    Resolve() ([]RemoteIdentity, error)
    // Returns True if the Remote is completely resolved; i.e., cannot be resol
    Complete() bool
}

// Remote consisting of a URL
type URLRemote struct {
    URL string
}

// Remote encapsulating a name and port number
type NamedEndpointRemote struct {
```

```
    Hostname string
    Port     int
}

// Remote encapsulating an IP address and port number
type IPEndpointRemote struct {
    Address net.IP
    Port    int
}

// Remote encapsulating an IP address and port number, and a set of presented cer
type IPEndpointCertRemote struct {
    Address      net.IP
    Port         int
    Certificates []tls.Certificate
}
```

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