# Post Sockets: Towards an Evolvable Network Transport Interface



**Author(s):**
Trammell, Brian; Perkins, Colin; Kühlewind, Mirja

**Publication date:**
2017

**Permanent link:**
https://doi.org/10.3929/ethz-b-000247956



**Originally published in:**
https://doi.org/10.23919/IFIPNetworking.2017.8264874

# Post Sockets: Towards an Evolvable Network Transport Interface

Brian Trammell
ETH Zürich

Colin Perkins
University of Glasgow

Mirja Kühlewind
ETH Zürich

***Abstract*—The traditional Sockets API is showing its age, and no longer provides effective support for modern networked applications. This has led to a proliferation of non-standard extensions, alternative APIs, and workarounds that enable new features and allow applications to make good use of the network, but are difficult to use, and require expert knowledge that is not widespread. In this paper, we present Post Sockets, a proposed new standard network API, that is designed to support modern network transport protocols and features, while raising the level of abstraction and enhancing usability. Specifically, Post Sockets aims to give portable applications the ability to use a clear, messages based, interface to multi-path and multi-stream transports, rendezvous and connection racing, and fast connection re-establishment.**

## I. Introduction

Many networked systems use the Berkeley sockets API as their interface to the transport layer protocols. This API was highly appropriate for 1980s era Unix, when it was designed, but is showing its age, and cannot effectively support modern transport protocols and network environments.

The positive effect that the simplicity of the Sockets API had on the development of the Internet cannot be understated: treating a remote endpoint like a file made a network application programmer out of anyone who knew how to do file I/O. The network, however, is not a file, and the demands both of current applications as well as current network conditions stretch this simplifying assumption to its breaking point.

Specifically, the sockets API doesn't effectively support asynchronous message-oriented applications, multi-streaming and multi-path communications, zero-RTT connection resumption with idempotent data, happy eyeballs and peer-to-peer rendezvous, and so on. This is not to say that such features cannot be implemented over the sockets API. Rather, their implementations are clumsy, complex, and error-prone, and require specialist expertise. Making effective use of the network is not accessible to typical developers.

In this paper, we present Post Sockets, a new transport layer API that is designed to support modern transport protocols. We describe our motivation and design rationale, present a brief overview of the key features of the API, and outline how Post Sockets can support new transport protocol implementations and applications. This paper does not go into implementation details that may be handle quite differently when Post Socket is implemented in a user space library over the existing API or integrated or partly integrated into kernel space. Our contributions are: 1) identification of the limitations of the Sockets API, leading to design rationale and requirements for a new API; 2) an overview of our proposed Post Sockets abstract API; and 3) a discussion of how the Post Sockets API can support new transport protocols; framing and asynchronous message-oriented transport services; and rendezvous, connection racing, and resumption.

Post Sockets is not the first proposal for an alternative to the Sockets API. However, it is novel in that it builds on trends in network APIs for programming languages and systems, is fully asynchronous, and supports the range of new transport protocols under development in the IETF, while raising the level of abstraction for applications. 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, including protocols for mobility support.
- 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 0-RTT resumption of communication, and for the implementation of the API to explicitly take care of connection establishment mechanics such as connection racing [1] and peer-to-peer rendezvous [2].
- 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, following the ideas in the IETF Transport Services (TAPS) working group [3].

## II. Overview of the Post Sockets API

Post Sockets replaces the traditional `SOCK_STREAM` abstraction with an Message abstraction, which can be seen as a generalization of the Stream Control Transmission Protocol's [4] `SOCK_SEQPACKET` service. The API is centered around a *Message Carrier* which logically group Messages for transmission and reception. A Carrier can be created actively via ***initiate()*** or passively via a ***listen()***; the latter creates a Listener from which new Carriers can be ***accept()***ed. For backward compatibility, these Carriers can also be opened as

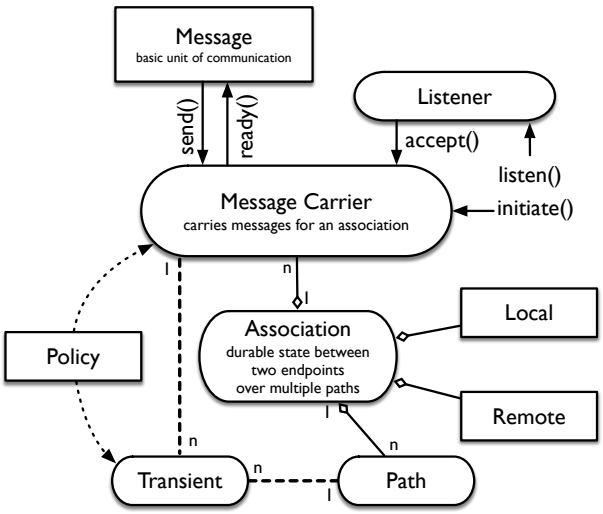


Figure 1. Abstractions and relationships in Post Sockets.

Streams, presenting a file-like interface to the network as with `SOCK_STREAM`. *Messages* may be created explicitly and ***send()*** over the 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). Whenever a Message is fully reassembled at receiver side, an asynchronous callback will notify the application that a received message is ***ready()***.

Message Carriers are bound to an long-lived *Association* which stores information about the identity of a *Local* and a *Remote* endpoint, as well as cryptographic session resumption parameters. New Message Carriers will reuse an Association if they can be carried from the same Local to the same Remote over an existing or newly found network *Path*; this re-use of an Association together with implementation of a *Policy* or a set of Policies defined by the application or system may imply the creation of a new *Transient* representing a concrete Protocol Stack Instance (PSI) assigned to an active Carrier.

The relationships among these elements are shown in Figure 1 and detailed in this section.

### A. 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. 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.

A Message Carrier that is backed by current transport protocol stack state (such as a TCP connection) 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 next section), and it may be able to reactivated, but messages cannot be sent and received immediately.

To exchange messages with a given remote endpoint, an application may initiate a Message Carrier given its Remote and Local identities; this is an equivalent to an active open. There are four special cases of Message Carriers, supporting different initiation and interaction patterns:

*Listener:* A Listener only responds to requests to create a new Carrier, 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. Accepting an incoming request will fork a fully fledged Message Carrier.

*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.

*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.

*Responder:* A Responder 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 Message Carrier may be 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 data, based on application layer knowledge about the configuration used by the other endpoint. This is mainly for backwards comparability with existing non-Post-Sockets stacks as well as an easy path for migration for existing application implementations. If supported by the underlying transport protocol stack, a Stream may be forked: creating a new Message Carrier associated with a new Message Carrier at the same remote endpoint.

### B. 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 file system. 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.

Applications can register callbacks to 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.

A Message has the following properties that allow the application to specify its requirements if applicable:

*Lifetime:* A wallclock duration before which the Message must be available to the application layer at the remote end or otherwise will be useless. As soon as it is know that a lifetime cannot be met, the Message is discarded. Messages without lifetimes are sent reliably if supported by the transport protocol stack. Lifetimes are also used to prioritize Message delivery. 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.

*Antecedents:* Other Messages on which it depends, which must be delivered before it (the successor) is delivered.

*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 which is highest priority. By prioritization of certain messages against others, e.g., blocking of smaller, latency-sensitive messages by large non-latency-sensitive messages can be avoided. Niceness may be translated to a priority signal for exposure to path elements (e.g., DSCP code-point) to allow prioritization along the path.

*Immediacy:* Marking a message as *immediate* signals to the transport protocol stack that its application semantics require it to be sent out immediately, instead of waiting to be combined with other messages or parts thereof (i.e., for media transports and interactive sessions with small messages). This allows the receiver make effective use of messages in the event of packet loss when messages do not have any Antecedents and therefore can be delivered independently [5], e.g., when applying application level Forward Error Correction (FEC). Such a restriction can either be expressed on a per-message base or as a policy for all messages send over a Carrier.

*Idempotence:* If marked as *idempotent* the underlying transport protocol stack knows 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).

Messages may also have arbitrary properties which provide additional information on how they should be handled.

The sending transport uses the message properties, along with information about the properties of the Paths available, to determine when to send which Message down which Path.

### C. Transient

A Transient represents a binding between an active Carrier and the instance of the transport protocol stack that implements it. A Transient contains ephemeral state for a single transport protocol stack over a single Path at a given point in time. A Carrier may be served by multiple Transients at once, e.g., when implementing multi-path 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., in a multi-streaming protocol.

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

### D. Association

An Association contains the long-term state necessary to support communications between a Local and a Remote 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 to carry Messages; and information about the Paths through the network available between them. Three inputs are needed to establish an Association: a *remote*, a *local*, and a *policy*.

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 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. 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.

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 identify it.

A Policy describes restriction and requirements from the application to select and configure Transients for a communication between a Local and a Remote. For instance, an application may require or prefer certain transport features [3] in the PSI(s) used by the Transient(s) for a given Message 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). Policy information, encapsulating application intent and constraint, is thus expressed as implementation-specific configuration for the Message Carrier and the Transient(s).

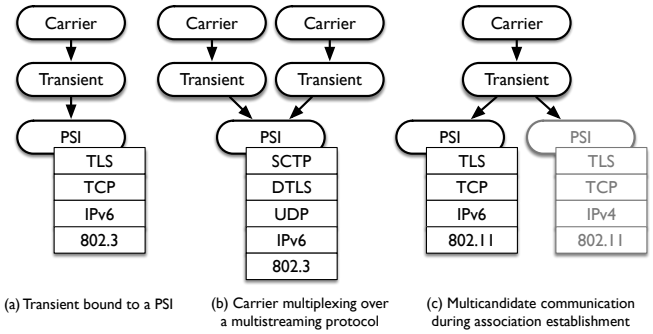


Figure 2. Protocol stack instances; multistreaming and happy-eyeballing

### E. Path

A Path represents information about a single path through the network known by an Association, in terms of source and destination network and transport layer addresses within an addressing context, and the provisioning domain [6] of the local interface. This information may be learned through a resolution, discovery, or rendezvous process (e.g., DNS, ICE), by active or passive measurements taken by the transport protocol stack, or by some other path information discovery mechanism.

The set of available properties is a function of the transport protocol stacks such as the MTU, expected one-way delay, expected probability of packet loss, expected maximum available data rate or reserved data rate.

## III. Protocol Implementation using Post Sockets

Post Sockets describes an abstract API that is intended to be broadly useful for applications, and that can support a wide range of transport protocols and services. The API we propose is deliberately tightly coupled with applications in a number of places, since many of the protocols of interest also exhibit similarly tight coupling [7]. In the following, we outline how support for new protocols can be added to Post Sockets. The intent is to provide an extensible library of protocols that applications can use.

### A. 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 [3]. 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. For example, figure 2(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 message carriers, as shown in figure 2(b): a multistreaming transport protocol like QUIC or SCTP can support one carrier per stream. Where multistreaming 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 2(c). Here, the losing PSI in a happy-eyeballs race will be terminated, and the carrier will continue using the winning PSI.

### B. The Message-based API, Parsing, and Serialisation

The byte stream API provided by TCP sockets is a mistake. We are aware of no protocol, except perhaps for an echo server, for which a byte stream is the correct transport abstraction: all impose some structure, some meaning, onto the byte stream.[1] 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.

Transports other than TCP recognise this. UDP, SCTP, and DCCP are all message-oriented. QUIC is stream oriented, but is usually used with HTTP framing that provides messages over the QUIC channel, and there are efforts to use the multi-streaming features of QUIC to provide message framing. Post Sockets follows this trend, and exposes a message-based API to applications as the primary abstraction, offering a stream-based API for ease of porting and backwards compatibility only.

There are other benefits of providing a message-oriented API beyond simply framing PDUs [8]. 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 canceled or paused anymore 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 other 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 somewhat similar features. However, they limit this to framed blocks of bytes. This is an advantage compared to a stream API, since framing is a frequent source of bugs in application code, but is still lacks semantic richness. Our intent with Post Sockets is to go beyond the existing message APIs, and raise the level of abstraction and support processing messages, rather than framed chunks of bytes.

[1] An FTP data channel might be regarded as a protocol use of a byte stream, but we believe it's correctly interpreted as a structured transfer of an object representing a single file, with out-of-band control.

```
trait ProtocolStackInstance<PDU> {
  ...
  fn send(pdu : &PDU) -> Result<(), IoError>;
  fn recv() -> PDU;
  ...
}

trait MessageCodec<PDU> {
  fn encode(&self, pdu : &PDU, buffer : &[u8]);
  fn decode(&self, data : &[u8]) ->
              Result<(Option<PDU>, &[u8]), ParseError>;
}
```

Figure 3. Post Sockets message-oriented API

Post Sockets is primarily intended to be used with a high-level systems programming languages (e.g., Swift, Go, Rust), rather than as a low-level C API. In such languages, we expose a semantically meaningful object-based API. That is, we push message parsing and serialisation down into the PSI, and let applications send and receive *strongly typed* data objects. Our approach is to raise the semantic level of the transport API: applications should send messages in the form of meaningful, strongly typed, data; parsing and serialising such data is the job of the protocol stack instance, not the application.

There are two parts to our API, shown in Rust-like pseudo-code in Figure 3. The protocol stack instance itself has `send()` and `recv()` functions as normal, except that those functions take and return high-level data objects. The protocol stack instance is parameterised by the type of the PDUs, and takes and returns objects of that type, rather than byte buffers. To support this, we also provide a `MessageCodec` API. An implementation of this trait is passed to the Protocol Stack Instance. This implement an `encode()` function that takes a PDU and buffer, and serialises the PDU into that buffer according to the protocol format. It also implements a `decode()` function that takes a pointer to a buffer of data received from the network, and returns either an optional PDU along with any outstanding data remaining to be parsed, or an I/O error.

By supplying parsing and serialisation via the `MessageCodec` API, and changing the `send()` and `recv()` functions to take polymorphic high-level data types, we raise the level of abstraction of the API. Passing references ensures this is a zero-cost abstraction, with no unnecessary data copies. The result is flexible, *safe* and easy to use, and efficient.

### C. Associations, Transients, Racing, and Rendezvous

As the network has evolved, even the simple act of establishing a connection has become increasingly complex. TCP clients now regularly race multiple connections, for example over IPv4 and IPv6. 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:

*1) 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 [2]. 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.

*2) 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.

*3) 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 [1], or it could involve a STUN exchange to establish peer-to-peer UDP connectivity [2], or some other means.

*4) 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 partialy during a connection to enable seamless hand-over and mobility within the network stack.

## IV. Related Work

Post Sockets is by far not the first attempt to modernize the Berkeley Sockets API; we refer the reader to section IV.B. of [9] for a current survey of this work. Most of this work, in the interests of easier deployability, has been explicitly evolutionary, and we take particular inspiration from some of these. SCTP [4] provided an API [10] adding sequential packet service and notifications to the Sockets API, which we see as a first step in the direction of Post Sockets. Message-oriented communication with antecedent-based ordering was implemented in TCP Minion [11]. Looking further back, dynamic instantiation of protocol stacks has been well-explored by the active networking community; e.g. by the Autonomic Networking Architecture [12].

## V. Conclusions and Outlook

In this work we presented Post Sockets, a new standard network API, providing a higher layer of abstraction that enables application developers easier access to novel transport features provided by new protocols. A key goal of Post Sockets is to raise the bar for all at once, rather than relying on education and incremental software updates to slowly bring support for modern connection establishment to applications over a period of many years. Post provides a message based interface, with asynchronous message reception, to multi-path and multistreaming protocols and integrates services for connection establishment and resumption. Thereby we provide the necessary hooks to allow effective use of the modern networking, and to provide generic services that can be shared across different classes of application. Post Sockets is a richer API than the traditional Sockets interface, but does little that is not part of existing applications. Rather it exposes the features of those applications, abstracts them, and makes them easily reusable. As Post Sockets semantics can be accessed on different levels of abstraction, e.g., also supporting the well-known byte stream interface and providing different interaction patterns, it supports a broad variation of applications. Post Sockets message-based interface requires, in addition to the actually transport protocol stack, a framing protocol. Where appropriate existing framing protocols such as HTTP can be used when integrated in to the protocol stack underneath the Post Sockets API. Alternatively a generic light-weight framing protocol could be develop for Post Socket's needs. However, this require some capability negotiation with the other endpoint to detect if this framing and thereby Post Sockets in support, while otherwise Post Socket can be used independent of the transport stack implementation of the other endpoint. Post sockets can be implemented as an abstraction layer on top of the current socket API in user space. However, the long term goals is the replacement of the Berkley socket API in kernel space, which may enable to more efficient integration of feature that currently not provided in the kernel network stack.

## VI. Acknowledgements

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. We are grateful to participants in the IETF Transport Services working group, for feedback on early versions of these ideas. The work was further developed in a workshop held at ETH Zürich in February 2017, attended by the authors as well as Tommy Pauly, Michael Welzl, Gorry Fairhurst, Anna Brunstrom, Marwan Fayed, Michael Tuexen, Zdravko Bozakov, Eric Vyncke, Mikael Abrahamsson, Erik Kline, and Basile Bruneau.

## References

[1] D. Wing and A. Yourchenko, "Happy eyeballs: Success with dual-stack hosts," IETF, RFC 6555, April 2012.

[2] J. Rosenberg, "Interactive connectivity establishment (ICE): A protocol for network address translator (NAT) traversal for offer/answer protocols," IETF, RFC 5245, April 2010.

[3] G. Fairhurst, B. Trammell, and M. Kuehlewind, "Services provided by IETF transport protocols and congestion control mechanisms," Working Draft, IETF, RFC 8095, March 2017. [Online]. Available: http://www.ietf.org/internet-drafts/draft-ietf-taps-transports-14.txt

[4] R. Stewart, "Stream Control Transmission Protocol," Internet Requests for Comments, IETF, RFC 4960, September 2007. [Online]. Available: http://www.rfc-editor.org/rfc/rfc4960.txt

[5] M. Handley and C. Perkins, "Guidelines for writers of RTP payload format specifications," Internet Requests for Comments, IETF, BCP 36, December 1999.

[6] D. Anipko, "Multiple provisioning domain architecture," IETF, RFC 7556, June 2015.

[7] D. D. Clark and D. L. Tennenhouse, "Architectural considerations for a new generation of protocols," in *SIGCOMM*. Philadelphia, PA, USA: ACM, 1990.

[8] S. McQuistin, C. S. Perkins, and M. Fayed, "Implementing real-time transport services over an ossified network," in *Applied Networking Research Workshop*. ACM/IRTF/ISOC, July 2016.

[9] G. Papastergiou *et al.*, "De-ossifying the internet transport layer: A survey and future perspectives," *IEEE Communications Surveys Tutorials*, vol. 19, no. 1, 2017.

[10] R. Stewart, M. Tuexen, K. Poon, P. Lei, and V. Yasevich, "Sockets API extensions for the stream control transmission protocol (SCTP)," Internet Requests for Comments, RFC, RFC 6458, December 2011.

[11] M. F. Nowlan, N. Tiwari, J. Iyengar, S. O. Amin, and B. Ford, "Fitting Square Pegs Through Round Pipes: Unordered Delivery Wire-Compatible with TCP and TLS," in *NSDI*. USENIX, 2012.

[12] A. Keller, T. Hossmann, M. May, G. Bouabene, C. Jelger, and C. Tschudin, "A system architecture for evolving protocol stacks," in *ICCCN*, 2008.