University of Glasgow | School of Computing Science

Post Sockets - A Modern Systems Network API

Mihail Yanev (2065983)

April 23, 2018

ABSTRACT

The Berkeley Sockets API has been the de-facto standard systems API for networking. However, designed more than three decades ago, it is not a good match for the networking applications today. We have identified three different sets of problems in the Sockets API. In this paper, we present an implementation of Post Sockets - a modern API, proposed by Trammell et al. that solves the identified limitations. This paper can be used for basis of evaluation of the maturity of Post Sockets and if successful to promote its introduction as a replacement API to Berkeley Sockets.

1. INTRODUCTION

Over the years, writing networked applications has become increasingly difficult. In part, this is because the applications have become more complex; the demands they place on the network are not the same as they were 30 years ago. It is also because the network has changed, and offers more features with more complexity. And, finally, it is because the de-facto standard API (Berkeley Sockets) was designed more than three decades ago, to match the network state at the time, and has not kept up with the evolution of the network and applications.

Modern applications' requirements demand for different features compared to applications from three decades ago. As a result, network administrators have introduced middleboxes, such as remote cache servers or firewalls, that intercept and change the exchanged information. On the other hand, new network techniques and protocols have been introduced by network standards organisations, such as IETF and IEEE . in an attempt to solve that same network complexity problem. The combined result from both of these interactions is that we are now left with a network, which is extremely hard to evolve, referred as the 'high-ossification problem' [10].

Communication using strongly typed data, choosing a remote address, when multiple are present and transport protocol selection are some of the issues with todays network. One way to address them is to revise the network programming API, making it aware of the specific problems and work towards their resolution. An example for such activity in the recent years is the concept of connection racing, as described in 'Happy Eyeballs: Success with Dual-Stack Hosts'[21] and updated by [15]. The technique allows for choosing the optimal address with respect to latency, in cases where multiple addresses are provided by a server.

In this paper, we present a new look of what we hope to be the new network programming API which incorporates solutions to the aforementioned problems in its core. The idea is based on the IETF Post Sockets draft [18].

In addition to providing native solutions for problems, establishing optimal connection with a Remote point or structured data communication, we have identified that a common issue with many of the previously provided networking solutions has been leaving the products vulnerable to code injections and/or buffer overflow attacks. The root cause for such problems has mostly proved to be poor memory or resource management. For this reason, we have decided to use the Rust programming language to implement the new API. Rust is a programming language, that provides data integrity and memory safety guarantees. It uses region based memory, freeing the programmer from the responsibility of manually managing the heap resources. This in turn eliminates the possibility of leaving the code vulnerable to resource management errors, as this is handled by the language's runtime. The language's strongly typed system also allows for better modeling the requirements and the interface of the new API, as restricting certain functionality to a particular type class (trait), ensures that any item, willing to use that functionality will implement the required type and expose the expected interface.

The remainder of this paper is structured as follows. Section 2 introduces limitations identified with the Berkeley Sockets and gives further details on our motivation for introducing Post Sockets as systems network API. Section 3 describes the Post Sockets idea as defined by Trammell et al. [18] and suggests how it solves the limitations, described in section 2. Section 4 presents evaluations conducted using a sample implementation of Post that we created. Section 5 outlines existing work, related to Post Sockets and overcoming the Berkeley Sockets' limitations in general. Finally section 6 wraps up and concludes the paper.

2. MOTIVATION AND CONTEXT

In this chapter, we open with a brief introduction of Berkeley Sockets, followed by a discussion of how its components interact. Then based on the discussion, we present the three different limitations identified with the current network. Finally, the chapter concludes with our motivation on why new API is needed.

Berkeley Sockets, also known as POSIX Sockets or BSD Sockets is the current de-facto standard network systems API. It provides simple interface, that exposes sockets abstraction as handles for the local and remote endpoints in a communication. These handles are treated as regular UNIX file descriptors. This design decision made the API extremely popular when it was released, as essentially any developer that knew how to operate files was able to easily get some sort of network communication going. The main components of the Berkeley API are presented in Figure 1,

where the items with solid border are exposed structures while the dotted border items are logical representations of entities in the system.

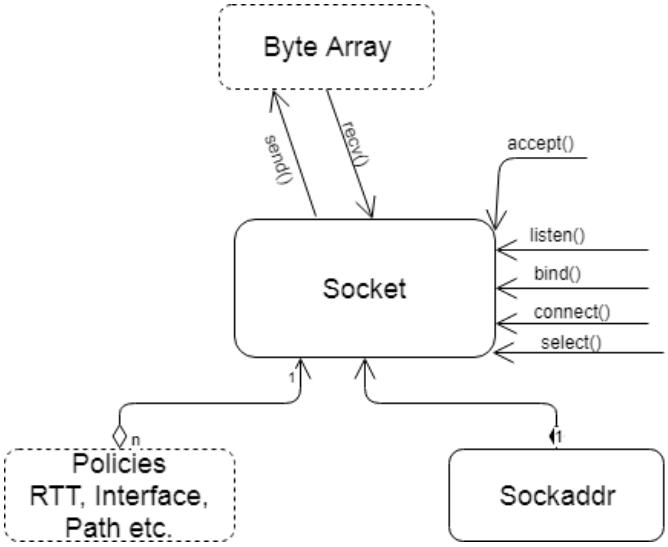

Figure 1: Main Components of the POSIX API

The API is centered around the Socket entity. The main operations for a Socket are: creation, configuration, connection, communication and termination. Following, we inspect each operation in detail.

Socket creation, is performed by specifying three arguments - address family (Internet protocol version), type for the data over the socket (usually datagram or stream) and transport protocol to be used (TCP/UDP/SCTP[17] ,etc).

After a socket is created, it must be bound to two "Sockaddr"s, representing the local and remote addresses used for the communication. Additionally, developers might wish to express preference for specific network options to be provided, if the underlying operating system supports them. These options might influence the policies for choosing when to send a packet, whether to fragment a packet, what route to use for packet transmission, for how long the connection should be kept alive ,etc. Notable such options are: 'NODELAY', used to disable Nagle's algorithm[9][3], 'KEEPALIVE', used to send periodic liveness pings to prevent connection termination, 'IP6_DONTFRAGMENT', used to instruct that large packets should be dropped, instead of fragmented and 'DONTROUTE', used to bypass the lookup table when transmitting packets.

Finally, after a socket is created and configured with the correct addresses and options, it must be connected to another socket via the connect routine, or set in a state that expects connections. The latter is done using one of the listen or select functions, followed by an accept, to establish the connection.

When two sockets are connected, they can initiate a communication process via the send() and recv() methods. Both of these operations work with byte array abstractions. For example in send() you provide a buffer with information to be send and you get returned a value of how many bytes were actually send. Similarly for recv() you need to provide an empty buffer and state its length as an additional parameter, which effectively acts as an upper bound of the bytes that can be received in that buffer. Since, both of these operations work with raw byte structures, they do not provide any guarantees about message framing. If supporting framing was desired, then it has to be added on top of the sockets send() and recv() methods.

After no communication is expected over the socket it is closed with a call to the close function. Whenever this happens, only the interface to the socket is destroyed and it is the kernel's responsibility to destroy the socket internally[8].

In terms of the OSI model, the Sockets API goes as high as layer 4, as shown in Figure 2

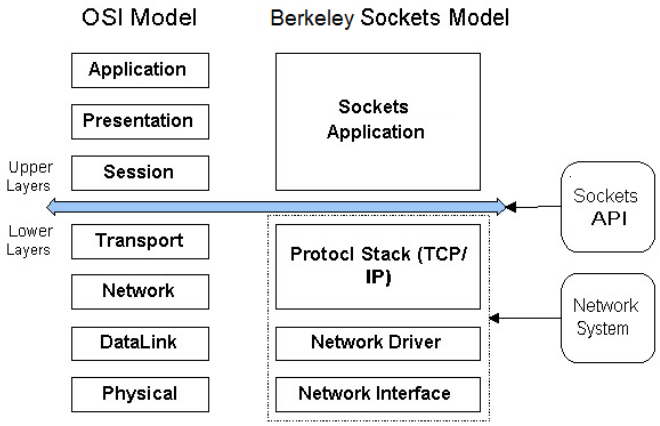

Figure 2: Sockets API in the OSI model. Image modified from [1]

The consequence of going as high as the transport layer is that some widely used network services are not directly supported and require additional application logic to be added. Examples for such services are: adding security to the transport, performing domain name resolution and packet framing. For example, when a socket is created, the IP version number needs to be specified, as well as the address to be used for this connection. This essentially means that you cannot create a socket and configure it to communicate with a domain name, such as 'google.co.uk'. Instead, for such a connection DNS lookup needs to be performed first, which would most likely result in multiple addresses being resolved and then a connection must be opened with one of those addresses.

As a result of the reach of POSIX in the OSI stack, we identify three key problems with the POSIX networkings APIs: lack of support for connection racing, lack of support for multipath communication, and inability to support structured data transfer. Following, we introduce each one in a more formal way and relate them to the discussion of POSIX, presented so far.

We first consider connection racing. "Happy Eyeballs: Success with Dual-Stack Hosts"[21] updated by [15] introduce a straight-forward idea of racing multiple connections to improve optimality and reduce network latency. The concept is: try to open multiple connections using different options, and use the connection that is fastest. This reduces latency, as it finds the optimal connection, but it does so at

the expense of increased network traffic, generated by the simultaneous connections. An example of that could be a server, which offers both an IPv4 and an IPv6 addresses for a single task.

Implementing connection racing has proven to be a challenging task with the current interface provided. This is because the POSIX API requires a single address to be given when creating a network socket, which makes supply of multiple such instances impossible. This idea made sense, when the POSIX API was emerging, as initially the network demands were not latency critical, so there was no need of multiple instances to serve the same resources.

Another limitation, related to the transport protocols domain is multipath promotion and support. As defined in MPTCP [5] multipath connections, allow more than one channels to be used for communication between two end points. In theory, any device with more than one network controller should be able to leverage this functionality, for example, mobile device with celluar and wifi capabilities. There have been reports that Apple is using this technology, in order to provide better service for Siri[4]. However, porting most of the internet applications would once again prove to be a challenge given the current API. This is partly because the developers are forced to pick a transport protocol when creating a socket. The choice of picking a transport protocol that has large network support (TCP/UDP) would seem very natural. However, this limits the need of support of any other protocol over the network, so a large portion of the middleboxes, such as firewalls, block any other traffic. Which in turn reduces the possibility of any new protocols emerging.

Finally, a core, well-known issue with the design of POSIX is the inability to support structured data transfer. The issue here is motivated by the fact that technologies using the API have evolved and have started to leverage higher-level abstractions for communication within the given process. However, carrying the same communication practices over the network is not directly supported, as the POSIX API, only provides functionality for transmitting byte sequences. For example, if a programmer wants to transmit a structured object, they have to resort to their language's serialization framework to make the item transmittable over the network. However, providing that functionality at systems level would eliminate the need for such solutions. It would also be more flexible and friendly to other technologies that have been newly introduced or have not added the custom functionality on their own.

One way to address these limitations would be to introduce a new systems API with the outlined issues in mind. For example, this new API could perform connection transparent to the user, offer connections to be opened with a set of transport protocols in mind and perform the necessary abstraction needed, in order to expose structured data for communications. In order to provide strongly structured types to be transmitted, it would make sense that the new API uses a technology that supports type classes as defined in Haskell or an alternative of that [19][2].

The requirement for type class support, eliminates the standard systems language, C, or at least in its standard form as an option for the new API, since C has a weak type system and is memory unsafe. An alternative to this technology could be Rust, which has been introduced recently, but makes a strong candidate for an adequate choice for any systems application. Rust is a recent, expressive language with strong type system [14]. The Type Classes, or Traits, as referred by Rust is a mechanism to support ad hoc polymorphism and so to define constraints on certain variable types. These constraints may be a minimal set of methods, required by the variable type, or concrete fields, etc.

In addition, using a strongly typed language, such as Rust, would make protocols modeling an easier task. In essence, this means that the language could be made to define a minimum set of features that any transport protocol should implement in order to be valid and ready for exploitation[20].

On another note, some of the limitations presented have clear resolution plans, for example: Happy Eyeballs. However, plans others are less clear, or not well defined, for example, protocol independent connections. And finally, there has been an effort to resolve part of the limitations given the current technologies. For example, MPTCP's push for enabling multipath connections or different languages providing abstractions to support structured data communications.

Given all of that, we have observed that fundamental ideas in the POSIX API limit the design space for future network protocols and make solving the current problems harder. As an example, we refer to the practice of initially choosing an IP address for socket communications. Introducing connection racing, as defined by Happy Eyeballs, in such an environment would require changes to the core API functions. Alternatively it can make use of several sockets to race the connections and would then proceed closing all but the winner from the race. Nonetheless, whichever solution is picked, it is evident that the POSIX limits the ways how the problem could be approached.

The introduction of MPTCP was a brilliant idea, that again, would perhaps be hard to tailor with the existing systems API. For example, making use of any transport protocol that is not TCP or UDP is a real challenge for programmers using POSIX. Perhaps another API would allow selecting a set of protocols to be attempted: MPTCP, for example, and if it fails then fallback to regular TCP.

Finally, POSIX fails to provide support for modern demands such as structured data communication (objects/structures/primitive types). As it only exposes an interface for managing byte transfers. This then resulted in each technology adding their own abstraction layer on top of POSIX, in order to provide the required communication. However, this means that any new technology would also need to add their own version of the same abstraction on top of POSIX, if aimed to gain the benefits from higher abstracted communication.

In conclusion, we believe that a new API, which accounts for the given limitation at its implementation core would be a better option than modifying POSIX to address them. This belief is based on the fact that it is already hard to implement solutions for some of the problems in POSIX. Furthermore, when future network evolution is taken in mind, it is natural that new sets of limitations will emerge. In such cases, we would not want to integrate "clumsy and error-prone"[18] solutions. Instead, a new API that addresses the current limitations and exposes flexible interface, through methods and hooks, ready to address future limitations should be used.

3. POST SOCKETS

Post Sockets, as introduced by Trammell et al. [18] is an alternative to the Berkeley Sockets API. Designed with the identified so far problems in the network in mind, Post Sockets offers an elegant solution to them. It does so by raising the abstraction of the transport layer and presenting an alternative view on the communication process. The system builds on trends in network API for programming languages and systems. Its expressiveness makes it easy to support all IETF transport protocols or to model future ones.

Post Sockets is a richer API than POSIX, but does little that is not part of the existing applications. For example, it introduces new abstractions items, such as *Locals, Remotes, Messages, Message Carriers, Associations, Policies and Transients.* With a few exceptions, these are not new concepts to the applications, but rather logically defined and group abstractions that have been used so far.

Post Sockets' manifesto promotes solving the current problems by raising the abstraction layer all at once. As opposed to 'relying on education and incremental software updates to slowly bring over a period of many years'[18]. Following, we introduce the main components of the system and present an example of how they interact.

Trammell et al. internet draft provides a diagram, which illustrates the system and how it interacts, displayed in Figure 3.

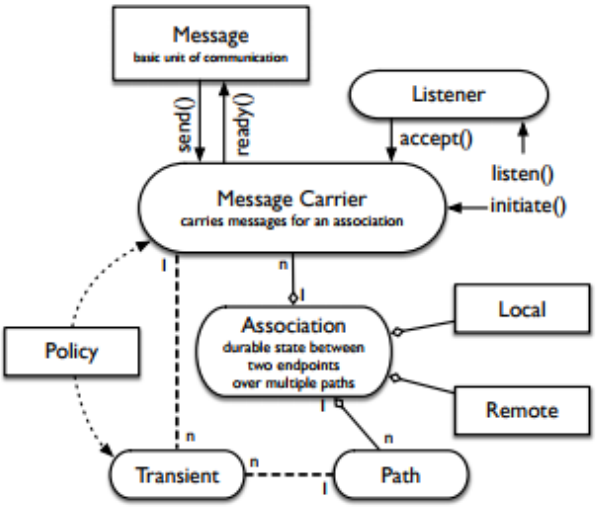

Figure 3: Components interaction in Post Sockets. Appears in [18]

The API is centered around Message Carriers, which logically group messages for transmission and receipt. *Message Carriers* are transport protocol-stack independent abstractions for sending and receiving *messages* between an application and a *remote* end point.

Messages are atomic units of communication between applications. Messages can represent structures of arbitrary size - both relatively small, such as HTPP Request/Response objects or relatively large, such as files. There is no strict mapping between a message and the packets sent by the underlying protocol stack on the wire. That is, multiple messages can be amalgamated into packets. Also, a single message might be segmented over multiple packets.

Associations store information, needed to maintain a long term state between local and remote endpoints. Such information includes - cryptographic state, required for fast connection resumption, session parameters as well as information on resolved names. For example, URL might be used for communication. The first time this URL is resolved by the DNS, the information would be kept in the association, in order to improve any subsequent requests.

Remotes represent information information required to establish and maintain connection with a far end of an *association.*

Analogous to Remote, *Local* abstractions store all the information about the local endpoint, such as interface, port or transport protocol stack information.

Finally, *Transients* are an active binding between an active carrier and an instance of the transport protocol stack that implements it. Transients contain ephemeral state for a single transport protocol stack over a single path at given point of time.

The terms described so far provide the necessary abstraction required, in order to solve the problems, identified earlier. For instance, transients could help in implementing multipath. To achieve this, multiple transients would serve a single carrier at a given time, where each transient would be bound to a separate path. This would grant that the multiple paths are exposed to the API by the underlying transport protocol stack. In addition, multiple transients may share the same protocol stack, achieving multistream capabilities that way. Multiple transients may also be active in for short period of time in the event of connection racing.

In order to achieve connection racing, the logic that performs it needs to be added to the specific carrier's implementation init method(this decision is later discussed and justified in section 4.3). For example, in a case where connection racing is performed for a HTTP client, the code might be similar to the one suggested in Listing 1

Listing 1: Connection Racing

```
impl ICarrier for HttpClient {

        fn init(mut self) -> Self {
                // Race multiple addresses
                    , contained in self
        // Remember fastest connection for
             future use
        ...
                self
        }

    ..
}
```

In the example above, the necessary operations, required to achieve connection racing are: firstly, gather the multiple addresses from the *Remote* instance contained in the *Carrier(self)*. Then create *Transients* from the obtained addresses and simultaneously try to establish connections with each one. Finally remember the fastest for future use.

For completeness, we supply the main components of the *Local*, *Remote* and *Transient* instances, respectively in Listings 2, 3, 4

Listing 2: Local Structure

```
pub struct Local {
        pub addr: String,
        pub port: i32
}
```

Currently in the implementation, Local is a minimalistic wrapper that only contains address and port of the local instance. In future iterations the Local structure would hold protocol stack information as well as certificates and associated with its private keys.

Listing 3: Remote Structure

```
pub struct Remote {
        pub preferred: Option<String>,
        alternatives: Vec<(String, String)
            >,
        pub port: Option<i32>
}
```

The Remote structure holds preferred pair of address and port, if such are provided, as well as alternative addresses and ports associated with this remote. Support for public keys and certificate authorities is to be added. The interface exposed by the Remote allows an instance to be created by either providing the necessary parameters or by providing a name to be resolved.

Listing 4: Transient Structure

```
pub struct Transient {
        address_family: IpAddr,
        transport: Transport
}
```

Finally, Transients are very lightweight structures, that hold empirical state about a given transport stack for a connection at a given state of time. The implemented fields store information about the remote address and address family in address_family and the type of transport. TcpStream and UdpSocket are the currently supported types of transport for stream and datagram connections respectively.

On another note, to address strongly typed data exchange, we have defined a Carrier type class(trait) as shown in Listing 5. Where *Item* is the type of the structured data and *Transmitter* is always a byte sequence.

Listing 5: Carrier Type Class

```
pub trait ICarrier {
        type Item; // Type of messages the
             carrier will work with
        type Transmitter;

        fn init(self) -> Self;

        fn data_recv<T>(received: T) ->
            Option<Self::Item>
                where T: IReceivable<Self
                    ::Item>;

        fn msg_recv(message: &Self::Item )
            ;

        fn send_msg<T>(&mut self, message:
             T) where T: ISendable<Self::
            Transmitter>;

}
```

These traits ensure that any item, sent over the network, has encode method defined for itself (granted by the ISendable trait) and any byte sequence received over the network can be composed back to the original item via a decode method, or indicate that reassembly was not possible (granted by the IReceivable trait).

For completeness, the ISendable(shown in Listing 6) and IReceivable(shown in Listing 7) traits are presented below in Listing 6 and 7 respectively.

Listing 6: ISendable Type Class

```
pub trait ISendable<T> {

        fn encode(&self) -> T;

}
```

Listing 7: IReceivable Type Class

```
pub trait IReceivable<T> {

        fn decode(&mut self) -> Option<T>;
}
```

The process of sending an item over the network now becomes - the item is passed to the Carrier via the send_msg() method, referred to as send() in Figure 3. The method then calls the encode function on the item (this why *self* is present as argument in Listing 6), which converts it to a byte sequence and sends it over the network.

For receiving, we have a longer process. We first receive a byte sequence from the network, via the recv_data function. Then we try to reassemble that data into a structured object, explained in detail in Section 4.1. If the item cannot be reassembled, it is assumed that insufficient data is provided, in which case the failed data must be prepeneded to the next bulk that arrives and the process should be repeated. When a structured item is fully reassembled, the msg_recv function is called, referred to as ready in Figure 3.

There have also been a number of items from Figure 3 that are currently missing from the provided implementation. Mostly, our motivation for not including these components has been that, we have not added any cryptographic support, as these tend to be quite large and could make a project of their own. Also, this first iteration focused on providing the user-space abstractions that are to be exposed by the API and most of the unreflected items would ideally be placed in the Kernel. Following, we present each item individually along with a brief justification on it not being reflected.

Firstly, we have seen the Path abstraction as a non-critical system component, as it essentially stores address information and provision domain for the local interface. The former is already contained in other parts of the system. The latter, would most likely require support for ICE[11] or STUN[12] to be added, which we think is not as important for demonstrating the principles of the API.

Secondly, Associations and Policies have also been omitted from the implementation. The items are closely related, the draft suggests that creating an Association requires a set of Policies to be provided. Furthermore, as connection racing and fragmentation are supported by default if present, we have not added an option to exclude those by configur-

ing a Policy. This further reduces the options exposed by the Policy and essentially limits it to requesting that the connection is kept alive until explicitly closed or expressing preference of one address family over another.

From the introduction of the Post Sockets idea and components, so far, we can see that it adds an additional abstraction layer on top of the currently provided services by POSIX, as shown in Figure 4.

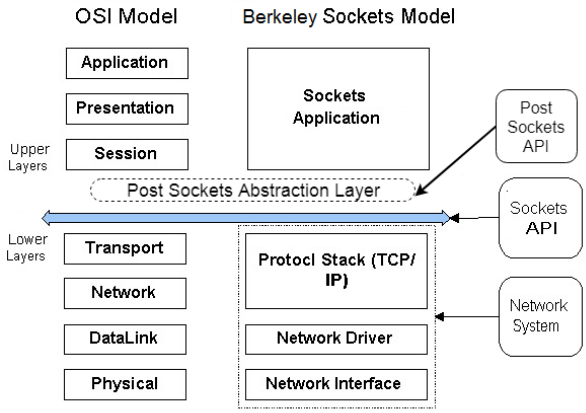

Figure 4: Post Sockets as an Abstraction Layer over POSIX

In comparison, this additional layer of abstraction, provides access to services, that were not present in POSIX, as discussed in Section 2. For example, a *Remote* may be created by supplying its URL. The Post Sockets API will then take care of performing the necessary DNS resolution operations and establishing a connection. Furthermore, the intended design (not present in this implementation) allows for providing the required cryptographic information, such as certificates, associated private keys, etc. directly to the Remote and Local abstractions and letting them establish secure channels. In other words, providing security to the transport requires minimal changes to the codebase and is directly supported and does not need external libraries or modules to be added on top. Finally, by providing an interface that ensures items send over the network can be serialized and reassembled, grants that message framing is also handled by the API.

In summary, we have introduced the Post Sockets idea and presented its capabilities to address the issues, we have earlier introduced with POSIX. In addition, we have seen that the richer nature of Post Sockets exposes wider spectrum of services, freeing developers from the need to manually add them. The next chapter presents a closer look of the process we took, to provide resolution for the issues identified with POSIX in the Post Sockets implementation.

4. EVALUATION AND CASE STUDY

In this section, we support our claims about Post Sockets, made in Section 3 by providing the results from their implementation. Additionally, we provide a discussion about how the items showcased here can be generalized. Furthermore, we provide a mapping from the items here to the internal concepts and architecture of the Post Sockets.

4.1 Minimal Set of Transport Services

We first consider the API's flexibility and ease to model current and future protocols. To support this, we refer to the ITEF's Transport Services (TAPS) draft of minimal set of transport features [20]. According to this draft, a transport service must cover the following cases. For flows - creation, configuration, connection and destruction. For frames (units of the transfer) send and receipt.

In terms of Post Sockets, flow operations are handled by Carriers, Policies and Transients and Paths. Flow creation is done by requesting a message carrier, configuration is handled via the policies given to that carrier and finally when a connection is established, the carrier is bound to a specific transport protocol stack and a transient is created to record that. The distinction between Transient and Carrier is present, since in some cases, there is no one to one mapping between the two. For example, multiple transients may compete to serve a given carrier in case of connection racing, or multiple transients may simultaneously serve a single carrier, in case of multipathing.

On another note, framing operations are supported by the Carrier's data_recv and send_msg functions as shown in Listing 5 and the IReceivable and ISendable traits.

Providing framing guarantees, proved to be a challenging task, that required multiple iterations. The first iteration achieved framing but violated the backwards compatibility criteria. The second iteration provided framing and backwards compatibility but excluded direct support for the message interface (setting priority, dependency, deadlines, etc.)

Initially, fragmentation relied on modifying the transmitted data and added an extra bit that indicated whether a message has been fragmented. This bit was set, when the data to be transmitted was bigger than the path's maximum data transmission unit. On the receiver side, whenever a fragmented message had been detected, the data was passed to the data_recv function, and if a message could not be reassembled it was awaiting for future such calls, to eventually reconstruct the message. Whenever a message was reconstructed, the carrier's msg_rcv function was called (referred to as ready in 3) and a on_msg_rcv event was fired. However, as stated above, this approach relied on modifying the data and transmitting the whole message abstraction over the network. In turn, this meant that communication would be possible only if both the receiver and the sender were using this implementation.

The limitations of the initial approach, demanded revisiting the implementation of the framing. The revision, introduced the idea that the recv_data function might fail, not succeed if given insufficient data to reassemble the object. This reflected on returning 'Option' results from the de-serialization method(decode in IReceivable) and the Carrier's data receipt method (recv_data in ICarrier). These functions return the reassembled item, or indicate that reassembly was not possible. In the latter case, the when the next batch of bytes arrives it has to be appended to the bytes that failed and the function should be invoked again. This is a refined approach for handling message framing, although it also contains a number of limitations. Such limitations include: the developer has to manually append the new data to the data that failed to be reassembled and call the receipt function. A better approach would be that the function stashes the bytes that failed to construct an item. Then upon subsequent calls of the function, if not empty, the

stash would automatically be perpended to the input data and then reassembly would be attempted. Another limitation of the function is that it does not account for other errors with the reassembly and only assumes that the bytes were insufficient.

The last limitation that comes to our mind with this type of framing is that the implementation does not clearly define what the message that Post Sockets operates with is. This means that any type, which has ISendable defined for it can be send over the network. However, since ISendable does not provide any information about deadlines, impotency, immediacy, etc. The items sent cannot be prioritized for example. This grants easier definition of the item transmitted over the network, but relies on the developer to handle the behaviour exposed to the message. For example, an item might look as shown in Listing 8.

Listing 8: Message Type Class

```
pub struct Message<T> {
        uid: i64,
        successors: Vec<i64>,
        is_partial: bool,
        chunks: Vec<i64>,
        is_immediate: bool,
        is_idempotent: bool,
        lifetime: u32,
        data: T,
        extra_fields: Value
}
```

But then the developer should only send the data field over the network and use the rest of the fields to fulfill the message requirements. Another approach would be to define a new type class for the messages and restrict that items sent over the network should implement this type class (trait). An example of such trait might be as shown in Listing 9

Listing 9: Message Type Class

```
pub trait IMessage {
        fn is_immediate(&self) -> bool;
        fn is_idempotent(&self) -> bool;
        fn get_deadline(&self) -> u64;
        fn depends_on(&self) -> Vec<Self>;
}
```

Such a trait can be used to restrict the input parameters of send_msg, which would allow for the logic for operations such as ordering and dropping to be handled internally.

To reinforce the ideas made in this section, we provide the codebase, containing concrete implementation for the examples provided here [22]. It contains HTTP server and client, as well as a DNS based on the interfaces explained. The reasoning behind choosing the concrete technologies is that the HTTP modules use the streams abstraction and TCP as transport protocol, while the DNS resolver leverages datagrams and UDP.

4.2 Event Driven Interactions

The Post Sockets main document suggests that some message events should be done asynchronously. Such events include, message receipt, message's failure for transmission, message's acknowledgement from the far end in an association, etc. This makes sense, since many of today's applications cannot or should not allow for these events to happen in sequential order. For example, a hypothetical application, which needs to perform an HTTP request, in order to display some information. It would not make sense for the whole product to be unresponsive, while a response comes back. This is where asynchronous events become useful. Another benefit of having such events is that they increase customization of the product. For example, letting users subscribe for a particular hooks or events, alleviates that responsibility from the systems developers and shifts it to the users, so that they can perform whatever actions they like.

Another use-case for event driven operations would be to define custom behaviour on when packets start to be fragmented, such as, sending smaller packets. Also, for events where too many of the items expire and cannot meet their conditions, the deadlines could be extended or the send queue may have to be depleted.

Other types of events, suggested by the Post Sockets draft trigger when a specific action for a given transient has happened. For example, specific types of messages may be sent when a particular type of connection is established, or alternatively, messages transmission might stop as soon as a given connection is stopped.

To demonstrate the capability of firing asynchronous events, the implementation provides a mean for subscription to message receipt events. As previously mentioned, such events are fired upon successful reassembly of an item over the network. In order to subscribe for message receipt events, developers need to implement the MessageHandler trait, shown in Listing 10

Listing 10: MessageHandler Type class

```
pub trait MessageHandler {
        type Item; // Item transmitted
            over the network

        fn on_msg_recv(message: &Self::
            Item);
}
```

Support for other types of events is not currently present in this iteration. Although, for particular classes of events providing support would be trivial. An example of that is connection establishment, such event would be fired as soon as a transient is created. On the other hand, support for other events such as message expiry would require changes to the API, such as the ones suggested in Section 4.1. However, providing support for all types of events has not been marked as critical for such an early iteration of the implementation, instead attention should be drawn on the fact that support for such events is possible and has been demonstrated.

4.3 Connection Racing

As discussed, connection racing is a technique, made to reduce latency at the cost increased network traffic. Such nature makes it extremely useful for long lived connections, since the initial overhead of an increase in the data transferred would be negligible. However, for connectionless protocols, such as UDP, or for connections that only exchange a low number of messages and then abandon the communication, connection racing does not make sense. It would delay the establishment and the trade-off will not be worth.

This is why we recommend connection racing to take place in Carrier's init function but do not enforce it. Following, we present a sample implementation of connection racing, that we created for the HTTP Client structure with relevant fields as shown in Listing 11

Listing 11: HTTP Client Structure

```
struct HttpClient<'a> {
        r: &'a Remote,
        query_addrs: Vec<(String, String)
            >,
        preferred_addr: Option<IpAddr>,
        ...
}
```

The client contains a Remote reference. The Remote might consist of address port tuples, or alternatively may contain a name to be resolved. After resolution occurs, the resolved address port tuples are stored in. Additionally, the client will keep track of the fastest address in the preferred_addr field.

In order to achieve the connection racing, we must put the relevant logic in the Carrier's setup (the init function as shown in Figure 3 and Listing 5). In essence, connection racing would simply take all addresses from the query_addrs collection and try to establish a connection with each one simultaneously, finally it will keep track of the fastest one and use it later.

Following, we present the effect of having implemented connection racing for our client. First, we setup our local HTTP Server on an IPv4 and IPv6 interfaces. Then, we create the carrier for the client as shown in Listing 12.

Listing 12: Sending multiple messages from the client

```
fn main()
{
        let alternatives = vec![(String::
            from("[::1]"), String::from
            ("3006"))];
        let remote = Remote::new(Some(
            String::from("127.0.0.1")),
            alternatives, 3005);

        let mut http_client = HttpClient::
            new(&remote);
        ...
}
```

When the client's new method is called, it does the resolution if required from the remote and then calls the Carrier's init method internally. Finally it returns the fully initialised client. Connection racing takes place when the Carrier's init method is called. The result from the instantiation process is displayed in Figure 5

```
addrs to race: [("[::1]", "3006"), ("127.0.0.1", "3005")]
Racing ("[::1]", "3006")
Racing ("127.0.0.1", "3005")
Winner addr: V6(::1)
Setting preferred addrs to: Some(V6(::1))
```

Figure 5: Connection Racing in Practice

By only adding the racing logic to the Carrier's init method when the trade-off is worth, grants greater flexibility to the developers, as for connectionless protocols or where it does not make sense to include racing, it simply is not. However, providing such flexibility firstly relies on developers be aware that connection racing is possible and secondly know where to add it. An alternative approach might have the init function calling a race function in the trait's implementation. This way, we would ensure that race has to be implemented, in order to initiate a carrier and for cases where it does not make sense to add it, developers would simply choose the first and in many cases only address as preferred. On the other hand, providing a fixed implementation for the init does not allow for it to be easily changed. For this reason, we have decided to give the developers the freedom of adding the racing logic to the init function, without enforcing it, perhaps at a later iteration, when all the actions which should happen in the init function are well defined, a fixed implementation might be the better choice.

When discussing the implementation of connection racing, it is also important that it also came with a number of challenges. Firstly, as it turns out almost none of the current Rust libraries for HTTP provide support for IPv6 addresses. A resolution for this was to use a wrapper cURL library to handle the HTTP communication.

Another challenge was getting the concept right, as in earlier iterations, rather than connection racing, we had implemented request racing. This raced the connection, as soon as the first request was made, rather than where it should be done, in a much earlier state. The result from this wrong implementation was that much more network traffic had been generated in order to discover the less latent connection. That traffic was a result of sending both the initial packets with the handshake as well as a request. Not only it was generating more traffic, but this behaviour could have easily introduced unforeseen bugs. For example, when a non-idempotent request is submitted multiple times to a server, the replies generated for that request will almost certainly be different. For example, supposed we wanted to upload a file to a server. If not done correctly, the server might end up with two copies of that same file.

Furthermore, since racing was taking place at the send_msg function, it would have only used the faster connection for every request, after the first one. An example output from that previous behaviour is given in Figure 6.

```
no preferred addr, racing connections...
addrs to race: [("[::1]", "3006"), ("127.0.0.1", "3005")]
Racing 127.0.0.1
Racing [::1]
on msg recv called
Message { uid: 0, successors: [], is_partial: false, chunks: 0, is_immediate: false, is_idempotent:
 false, lifetime: 0, data: Point { x: 5, y: 42 }, extra_fields: Object({}) }
Type of message: Message<Point<f64>>
on msg recv called
Message { uid: 0, successors: [], is_partial: false, chunks: 0, is_immediate: false, is_idempotent:
 false, lifetime: 0, data: Point { x: 5, y: 42 }, extra_fields: Object({}) }
Type of message: Message<Point<f64>>
Winner addr: RaceResult { addr: "127.0.0.1", port: "3005" }
Setting preferred addrs to: Some(V4(127.0.0.1))
------Sending second message------
preferred addr is Some(V4(127.0.0.1))
on msg recv called
Message { uid: 0, successors: [], is_partial: false, chunks: 0, is_immediate: false, is_idempotent:
 false, lifetime: 0, data: Point { x: 5, y: 42 }, extra_fields: Object({}) }
Type of message: Message<Point<f64>>
```

Figure 6: Request Racing. Bad Interpretation of Connection Racing

4.4 Adding New Protocols

The additional abstractions, provided by Post Sockets and the interface provided make creating new protocols an easy task. Suppose we were to design a HTTP based protocol with timing constraints for the items that are sent over the network. Such a protocol would be helpful for real time streaming applications. Justification for this would be that, whenever an item cannot be delivered within a given time-

frame, there is no point in even attempting to transmit other related items.

The first step in creating the new protocol would be to define how an individual transmission unit would look like. It would probably contain some data and have meta data about its timing constraints.

After designing the item, if we have chosen the approach where the Message type is a trait, we would have to implement the it for the object. Then, we would have to provide a mechanism for that object to be serialized and de-serialized with the encode and decode functions in ISendable and IReceivable.

Next, we would have to implement a new *Carrier* type that will serve our object, this is the most involved part. To help make the following explanation clearer, we provide the flowchart in Figure 7.

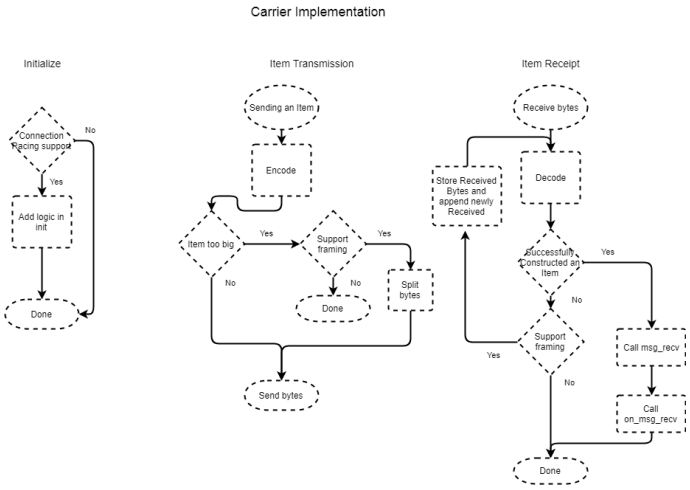

Figure 7: Request Racing. Bad Interpretation of Connection Racing

In the Carrier's implementation process we have several decisions to make. Firstly, would our protocol benefit from connection racing? If the answer is yes, then we would have to add this functionality in the Carrier's init method.

Next, we would have to decide whether fragmented messages would be of any use and whether we would attempt to reassemble fragmented pieces. If the answer is yes, we would have to account for failures for de-serialization of the Carrier's data_recv method. Whenever the method received an insufficient amount of data and did not reassemble the data, we would have to store it, append any new data arrived over the link and try de-serializing again.

Finally, we would have to define behaviour that would happen when we receive a message (done via the *MessageHandler*'s on_msg_rcv method).

By following the above steps, we can now create a Carrier for our protocol. Instantiate it and begin data transmission.

4.5 Syntax and Semantic Comparison with POSIX

In POSIX applications, if a developer wanted to create a HTTP client, they would have to create a STREAM socket and add the necessary tooling to transmit HTTP objects over the socket. In contrast, if a developer wanted to perform the same task, using Post Sockets, all they have to do is request a HTTP carrier to be opened. This results in applications, written using Post Sockets to be much more concise, as Post hides a lot of common repetitive functionality, which would otherwise be needed. However, there are some limitations in choosing this approach. For example, the current implementation of the HTTP carrier performs connection racing and uses the fastest link, but developers might prefer to make a decision based on another criterion, rather than speed. This is currently not supported and the alternatives would be to either fall back to using POSIX or to create a new carrier that fits the developers needs.

Another semantic difference between the two systems is how they establish connections with far ends. For example, the POSIX API requires an IP address with which it establishes a connection. Post Sockets, on the other hand, require a Remote structure. This Remote structure might contain a single IP address, an IP address and a collection of alternative IPs or a string representing the public domain name of the remote server (e.g. www.dcs.gla.ac.co.uk)

When discussing the API differences, it is also important to note that POSIX requires external modules and code modification in order to support transport layer security. While Post Sockets can optionally take information required to establish such secure channels in its data abstractions, such as policies, locals, remotes, etc.

Finally, POSIX does not provide native support for giving transmission items different priorities or expressing preference of a type of link over the other. Post Sockets on the other hand lets the developer manipulate both inputs.

As presented, Post Sockets provides support for a greater number of services, making it easier for application developers to communicate with remote applications and promoting transport evolution with its well defined abstractions. Although the benefits of using Post Sockets as API may be evident for a wide range of applications, there are specific circumstances, in which a developer might prefer POSIX. An example of such case might be when a developer would not benefit by any of the additional services. Another example may be when a developers wants support for raw sockets, which is currently not possible in this iteration of the Post Sockets implementation.

5. RELATED WORK

Post Sockets is not the first project that attempts to modernize or replace the POSIX API. The concept of expressing preference for a certain type of connection over another existed in 'Intentional networking' and was first realized in Multi-Sockets[6] and Socket Intents [16].

SCTP exposed connecting to multiple IP addresses and multi-streaming to the Sockets API. It also enabled signaling for certain transport-specific events, such as changes to connection status.

Another API, proposed as a replacement for Berkeley Sockets is NEAT[7]. Implemented as a user space library, it allowed applications to express preference of specific transport features and other policies. This enabled transport protocol selection to take place at runtime.

The tools explained so far target mainly the deployability issue in POSIX. Other frameworks exist and target different limitations of the API. For example, Stackmap[23] and Netmap[13] address the performance limitations. Netmap

is a high speed packet processing tool, which leverages the so called 'kernel-bypassing' technique. It performs zero-copying by taking over the Network Interface Controller and passing pointers to the data read. Stackmap, on the other hand is a full featured TCP Stack implementation that uses Netmap as its dataplane and the POSIX API as a control plane.

6. CONCLUSIONS

This paper presented how Post Sockets can be used to address limitations of Berkeley Sockets systems network API. It also proposed and evaluated a sample implementation of Post Sockets in Rust. Furthermore, the implementation demonstrated that deployability might be promoted as modeling new protocols is made easy by the new API. In addition, we have compared the API, identifying specific types of services that Post Sockets provides on top of Berkeley Sockets.

The implementation provided in this paper is but a single point in a much larger plan for deployment of Post Sockets as a new systems API. We identify three major categories for future work: stylistic, functional and innovative. The stylistic future work would involve improving consistency in the existing codebase, this includes - naming conventions, providing documentation for the public interface, leveraging more canonical Rust practices with respect to the implementation, etc. The functional future work would involve: adding the missing components of the system (Policies, Associations, etc.), providing better support for existing functionality (e.g. message framing) and experimenting with alternative designs for current implementation, as suggested for the message abstraction in section 4, for example. Finally, the innovative work would aim to provide a Kernel replacement for the POSIX components in the transport layer, such as path information (RTT, route, etc.). Lastly, another field to be considered is providing support for security in the transport layer, by adding the necessary abstractions for certificates, public and private keys, etc.

Acknowledgements

Thank you to Colin Perkins, for the initial project idea, the constant flow of feedback, support and guidance throughout.

7. REFERENCES

[1] Sockets concepts. `http://diranieh.com/SOCKETS/Concepts`. (Accessed on 04/17/2018).

[2] Why type systems matter. `http://pcwalton.blogspot.co.uk/2010/10/rust-features-i-type-inference.html`.

[3] L. E. A. Zimmermann, W. Eddy. Rfc 7805 - moving outdated tcp extensions and tcp-related documents to historic or informational status. `https://tools.ietf.org/html/rfc7805`, 2016. (Accessed on 04/20/2018).

[4] O. Bonaventure and S. Seo. Multipath tcp deployments. *IETF Journal*, 2016, November 2016. `http://www.ietfjournal.org/multipath-tcp-deployments/`.

[5] A. Ford, C. Raiciu, M. J. Handley, O. Bonaventure, and C. Paasch. TCP Extensions for Multipath Operation with Multiple Addresses. Internet-Draft draft-ietf-mptcp-rfc6824bis-10, Internet Engineering Task Force, Mar. 2018. Work in Progress.

[6] B. D. Higgins, A. Reda, T. Alperovich, J. Flinn, T. J. Giuli, B. Noble, and D. Watson. Intentional networking: Opportunistic exploitation of mobile network diversity. In *Proceedings of the Sixteenth Annual International Conference on Mobile Computing and Networking*, MobiCom '10, pages 73–84, New York, NY, USA, 2010. ACM.

[7] N. Khademi, D. Ros, M. Welzl, Z. Bozakov, A. Brunstrom, G. Fairhurst, K.-J. Grinnemo, D. Hayes, P. Hurtig, T. Jones, S. Mangiante, M. TÃijxen, and F. Weinrank. Neat: A platform- and protocol-independent internet transport api. *IEEE Communications Magazine*, 55(6):46–54, 6 2017.

[8] S. E. Laboratory. Programming unix sockets in c - frequently asked questions: Writing server applications (tcp/sock_stream). `https://www.softlab.ntua.gr/facilities/documentation/unix/unix-socket-faq/unix-socket-faq-4.html`. (Accessed on 04/21/2018).

[9] J. Nagle. Rfc 896 - congestion control in ip/tcp internetworks. `https://tools.ietf.org/html/rfc896`. (Accessed on 04/20/2018).

[10] G. Papastergiou, G. Fairhurst, D. Ros, A. Brunstrom, K. J. Grinnemo, P. Hurtig, N. Khademi, M. TÃijxen, M. Welzl, D. Damjanovic, and S. Mangiante. De-ossifying the internet transport layer: A survey and future perspectives. *IEEE Communications Surveys Tutorials*, 19(1):619–639, Firstquarter 2017.

[11] C. Perkins and M. Westerlund. IANA Registry for Interactive Connectivity Establishment (ICE) Options. RFC 6336, July 2011.

[12] M. Petit-Huguenin and G. Salgueiro. Datagram Transport Layer Security (DTLS) as Transport for Session Traversal Utilities for NAT (STUN). RFC 7350, Aug. 2014.

[13] L. Rizzo. netmap: a novel framework for fast packet I/O. *Proceeding USENIX ATC'12 Proceedings of the 2012 USENIX conference on Annual Technical Conference*, 2012.

[14] rust lang. Papers influenced rust (rust's background). `https://github.com/rust-lang/rust-wiki-backup/blob/master/Note-research.md#type-system`.

[15] D. Schinazi and T. Pauly. Happy Eyeballs Version 2: Better Connectivity Using Concurrency. RFC 8305, Dec. 2017.

[16] P. S. Schmidt, T. Enghardt, R. Khalili, and A. Feldmann. Socket intents: Leveraging application awareness for multi-access connectivity. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '13, pages 295–300, New York, NY, USA, 2013. ACM.

[17] R. Stewart and C. Metz. Sctp: new transport protocol for tcp/ip. *IEEE Internet Computing*, 5(6):64–69, Nov 2001.

[18] B. Trammell, C. Perkins, and M. KÃijhlewind. Post sockets: Towards an evolvable network transport interface. In *2017 IFIP Networking Conference (IFIP Networking) and Workshops*, pages 1–6, June 2017.

[19] P. Wadler and S. Blott. How to make ad-hoc

polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pages 60–76, New York, NY, USA, 1989. ACM.

[20] M. Welzl and S. Gjessing. A Minimal Set of Transport Services for TAPS Systems. Internet-Draft draft-ietf-taps-minset-03, Internet Engineering Task Force, Mar. 2018. Work in Progress.

[21] D. Wing and A. Yourtchenko. Happy Eyeballs: Success with Dual-Stack Hosts. RFC 6555, Apr. 2012.

[22] x. Sample post sockets implementation. `https://github.com/2065983Y/POST-Implementation`. (Accessed on 04/18/2018).

[23] K. Yasukata, M. Honda, D. Santry, , and L. Eggert. StackMap: Low-Latency Networking with the OS Stack and Dedicated NICs. *Proceedings of the 2016 USENIX Annual Technical Conference (USENIC ATC âĂŹ16)*, 2016.