SLIM: A Session-Layer Intermediary for Enabling Multi-Party and Reconfigurable Communication

Umar Kalim[*], Mark K. Gardner[*†], Eric J. Brown[†], Wu-chun Feng[*]
[*]Department of Computer Science, [†]Office of IT
Virginia Tech, USA
{umar, mkg, brownej, wfeng}@vt.edu

ABSTRACT

Increasingly, communication requires more from the network stack. Due to missing functionality, we see a proliferation of networking libraries that attempt to fill the void (e.g., iOS to OSX Handoff and Google Cast SDK). This leads to considerable duplication of effort. Further, the provisions for extending legacy protocol stacks is largely exhausted (e.g., TCP options space is mostly allocated) making the addition of future extensions much more challenging.

We present SLIM, an extensible session-layer intermediary that extracts the duplicate functionality from modern networking libraries and provides the means for future extensibility to the network stack. SLIM enables mobility, multiparty communication, and dynamic reconfiguration of the network stack in a straightforward and elegant way. SLIM includes an out-of-band signaling channel, which not only enables reconfiguration, but also allows for incremental evolution of the stack.

To start, we tease out elements of session management which are currently conflated with transport semantics in TCP. Doing so highlights the need for sessions in contemporary use cases. Next, we propose session, flow and endpoint abstractions that allow application developers to describe communication between any number of participants. The abstractions apply to individual or a group communication allowing them to be managed as one. We describe the abstractions and evaluate them in terms of typical communication patterns. We demonstrate the abstractions via a prototype implementation of SLIM.

Categories and Subject Descriptors

C.2.2 [**Network Protocols**]: Protocol architecture (OSI model); C.2.6 [**Internetworking**]: Standards

Keywords

Multi-party Session; Reconfigurable Communication

1. INTRODUCTION

To implement modern use cases, developers require support from the underlying stack. If such support is not available, extending existing implementations would suffice. However, legacy stacks pose challenges on both counts; while legacy stacks have met the communication needs for the proverbial 80% of the use cases, they do not enable the remaining 20% scenarios that support modern use cases. Therefore, to meet the demands on innovation in modern communication, developers implement networking libraries, which work around the limitations of legacy stacks, thereby enabling contemporary use cases; Apple Continuity [3] and Google Cast [2] are examples of such implementations among many others.

Innovation is challenging due to limiting assumptions of existing abstractions and their implementation. For example, the assumption that the network address does not change during communication, or that the duration of a connection is typically shorter than the lifetime of a network label, has led to implementations where connection labels are defined in part using network labels — i.e., services using sockets API have the network address as part of the transport label [21]. This inhibits mobility. Similarly, limited support for extensions in legacy stacks (e.g., limited space for new TCP options, which can't get through middleboxes [10]) has led to radical proposals such as QUIC [5].

We propose an extensible session-layer intermediary (SLIM), which supports functionality common to most modern use cases and provides means for future extensions to the network stack. Thus eliminating duplication of effort at different layers. This layer:

1. Enables support for mobility, multi-party communication, and dynamic reconfiguration of the network stack;

2. Uses *session*, *flow* and *end-point* abstractions to describe communication between any number of participants in an agreed upon context and allows application of primitives to individual or a group of constructs; and

3. Provides an out-of-band signaling channel for end points to exchange control messages, enabling reconfiguration and extensibility.

Consider for example a finance application, with a user interface on a smart phone and the service deployed in the cloud that interacts with stock exchanges to pull data and make trade decisions. Since the software SLA requires minimal latency, it is imperative that the service moves to a data center near the stock exchange. For international trade applications, the service would need to move across the globe without disrupting communication. SLIM can not only mitigate the issues that inhibit mobility, but also facilitate the design of the service, where the session, flow and end-point abstractions can describe communication between the building blocks (e.g., data source, trade and decision model, transaction manager and user interface). With out-of-band signaling available, extensions such as dynamic reconfiguration of the stack, service discovery or seamless migration of processes between hosts can be assisted by the session layer. Similarly, with multi-party semantics, a single session can be used by multiple participants to manage a conversation. With the ability to group flows in a session, common configuration parameters can be managed for all flows rather than defining individual configuration of constituent flows. Therefore SLIM has the potential to facilitate modern use cases as well as pave the way for further extensions to the network stack.

Several notable attempts have been made to introduce high-level abstractions that mitigate limitations of the network stack and propose extensions [9, 14, 16, 17, 20]. However, most proposals have not been widely adopted due to the resistance to move away from TCP owing to its widespread deployment; TCP's success, it seems, is a hurdle in the evolution of network communication [10]. This success has created the notion of an ossified transport, to the extent that HTTP is now being considered as the "evolvable narrow waist" and hope for evolution of the underlying stack seems to fade away [15]. Also, lack of backwards compatibility, significant porting effort and limited scope of proposed solutions does not help in tipping the scales in favor of the value offered as compared to the transition cost. Perhaps, a proposal that presents evolution of the stack and allows incremental adoption will be relatively easier to deploy.

In this paper we present:

- The design of an extensible session-layer intermediary, SLIM, based on session, flow and end-point abstractions and their associated primitives;
- A discussion of how SLIM supports typical communication patterns and how session management is coalesced with transport semantics in TCP; and
- A proof-of-concept implementation of SLIM that exposes an API for aforementioned abstractions.

2. CONFLATION OF SESSION AND TRANSPORT SEMANTICS

When the TCP/IP stack was first implemented, a single connection between two hosts was deemed sufficient for the entire conversation[1]. However, with the evolution of communication, we've seen that this assumption was invalidated soon after the proliferation of the Internet; applications established multiple streams to the same or different hosts — e.g., a browser fetching files from different servers to construct a web page.

The socket abstraction implemented TCP where it was responsible for managing the conversation, which in this case was a single stream — the 3-way hand shake establishes the communication session. However, when applications establish multiple streams between the same hosts, the 3-way hand shake with the same host becomes unnecessary for subsequent streams. The existence of a transport connection can be leveraged to bootstrap a new connection to the same host. Data can be sent as part of the first TCP segment and the network adaptation parameters (i.e., window sizes for congestion and flow control) can be derived from the existing connection without violating fairness constraints thus avoiding the slow start phase.

Note that this issue of conflating session semantics with other layers issue is not limited to transport. Higher-level layers also include session semantics. For example TLS also implements authentication per stream[2]. Once a session has been authenticated over a stream, why should we repeat the authentication process to create an additional stream, when we can record the authentication details as part of the session state?

While conflating session and transport semantics together in TCP's connection management may have been the preferred approach at the time, we can further improve the design by separating the session and transport semantics into different layers.

3. SESSION-LAYER ABSTRACTIONS

To define the role of a session layer, we describe interactions between the *session*, *flow* and *end point* abstractions.

Session: *The session abstraction represents the conversation between participating end points in an agreed upon context.* It identifies the participants, describes the conversation, and provides an interface to setup or reconfigure the communication. Description of the conversation may include the definition of data streams be-

[1]We take the example of TCP as transport and limit our discussion to it due to its wide-spread deployment.

[2]Although TLS implements the optimization where an authenticated session key can be reused by subsequent connections, this approach is not used widely, as servers need to maintain a session cache per client, which is costly for high-traffic services and is therefore disabled.

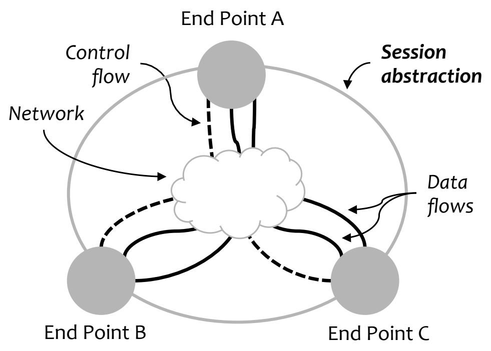

Figure 1: The session abstraction involving three participants, each with two data flows instantiated by the application. The control flow between participants is an out-of-band channel that allows setup and reconfiguration of communication.

tween participants or the choice of transport protocol for the data streams; this is not an exhaustive list. Configuration may involve enabling privacy or defining the mapping between data streams and underlying transport. Reconfiguration may involve refreshing network configuration forcing an update of transport parameters in case they changed due to mobility or enabling compression of streams.

Figure 1 is an illustration of the session abstraction involving three end points. Unlike the traditional sockets abstraction, where it is assumed that only two participants are involved in communication, the session abstraction represents one or more end points as part of the conversation.

Flow: *The flow abstraction represents a logical construct of data exchange between a set of end points.* A flow, as part of a session, can not only represent point to point communication, but also other communication patterns, as we discuss in Section 4. Flows form the corner stone around which the session abstraction is built. These flows may be mapped onto streaming or datagram transport protocols as required.

Sessions contain *data* flows that are independent logical constructs visible to the application. On the other hand, a *control* flow is not directly exposed to applications. Instead the session layer exposes primitives that indirectly enable applications to use the control flow to react to changing operating environments. This allows reconfiguration of communication. It is also used as an extension mechanism. We discuss use of the control flow in Section 3.2.

In traditional designs based on sockets (e.g., FTP, VoIP) the application is required to manage different data flows — some for application signaling and others as data streams — while doing book keeping to manage the conversation. In contrast, the session abstraction manages the book keeping and allows the developer to focus on the application logic.

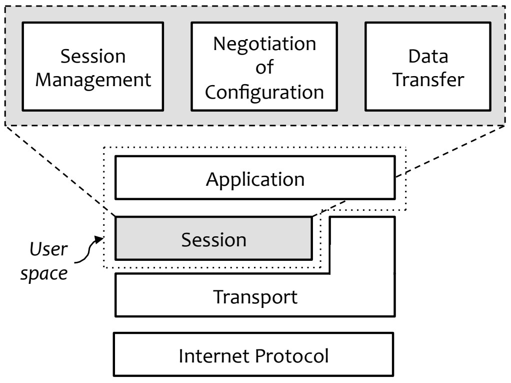

Figure 2: The session-layer services the application layer while interfacing with the transport layer below.

End Point: *The end point abstraction represents a process hosted by a node with at least one network-attachment point.* This association of the process and host is not assumed to be permanent. For example the process may change hosts (in case of migration) or may appear to change (i.e., the network address may change as the host moves from one network to another). This is in contrast with the immutable definition of an end point by the socket abstraction.

As we'll see in the following sections, these abstractions (session, flow and end point) allow us to describe and manipulate communication constructs, while being independent of the underlying services. It is the design choices in mapping the session abstraction to the underlying transport that allows SLIM to tease out session semantics from the transport layer and enable features such as mobility, multi-party communication and reconfiguration of constructs, without duplicating the functionality of underlying services.

As Figure 2 illustrates, the session layer exposes three sets of services to the application to assist with communication setup and management. These are discussed below. Note that existing applications can continue to use transport directly, or use SLIM and gain functionality, allowing incremental adoption.

3.1 Session Management

The session management services allow instantiation of the session, its configuration and reconfiguration, manipulation of the flows and termination. These all are executed through primitives defined by the layer. The relationship between some of the primitives is illustrated in Figures 3.

3.1.1 Session-Related Primitives

Typically a process expresses its willingness to communicate and is then joined by other processes. This is illustrated in Figure 3. A process can `create` a session

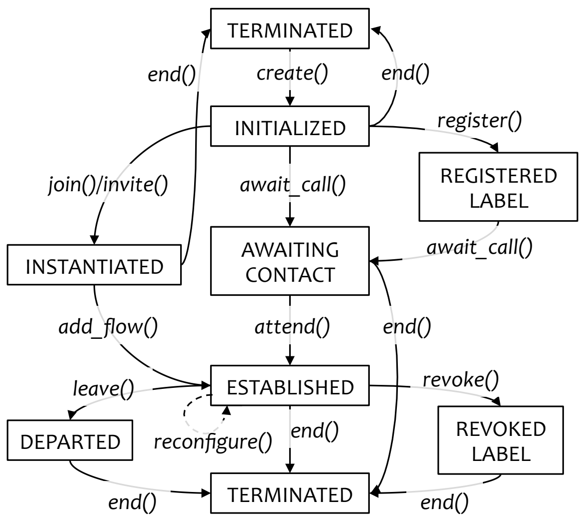

Figure 3: The session state transition digram highlighting the relationships between primitives. The transition `reconfigure`, which is not a primitive, highlights that any reconfiguration does not result in a change of session state.

to instantiate communication. At this stage, the only participant in this newly created session is the parent process. When this process wants to express its willingness to communicate publicly, it may do so by executing the `register` primitive. This registers the existence of the session recognized with `label` with the session discovery mechanism. The `endpoint` construct represents the host information of the process willing to accept contact. A process may choose not to register the session. In this case, peers interested in joining the session would be expected to be aware of at least one of the endpoints involved in the session — since there may be more than two participants in the session.

The first participant awaits contact from processes interested in establishing a communication session via the `await_call` primitive. A process that has not called `await_call` cannot accept `join` requests.

Once a process creates a session and awaits contact, another process can `join` the session. This assumes that the process wanting to join the session is aware of the session label, so that it may request a label translation using the `translate` primitive. The `translate` primitive queries the session registry for a peer's endpoint details to inform the session instance so that the peer may be contacted.

A peer process can `join` a session to setup at least one flow as part of the session. The `attend` primitive begins a fresh conversation if participants join a new session. However, if participants are conversing, the new arrival becomes a participant from there onwards. Any process can `leave` the session while allowing the remaining participants to continue communicating. The `end` primitive terminates the session. If participants exit a registered session prematurely, the heartbeat mechanism of the session registry enables a graceful tear down of the session. If a participant exits an unregistered session prematurely, then the other participants would not be aware of the departure, unless keep-alive mechanisms are implemented.

A process may be aware of potential participants. Here, instead of calling `join` to instantiate the session, the process executes the `invite` primitive. Primitives that expose the ability to negotiate configuration are discussed in Section 3.2.

3.1.2 Flow-Related Primitives

The `add_flow` primitive creates an independent flow as part of the session, which is accessible to the application. It takes a named parameter *session* and optional inputs of *[structure]* and *[type]*. The structure defines the communication pattern for the conversation. We discuss these patterns in Section 4. The type defines how the flow will be mapped onto underlying transport — i.e., whether it will be mapped to a stream- or message-oriented transport. The `terminate_flow` primitive tears down the flow.

The application views the flow abstraction as a logical construct. In typical interactions of processes there are often more than one flows between processes. They may or may not exist at the same time; together they form the conversation. It is this decoupling of the data stream from the transport connection, in the form of a logical flow, which allows us to eliminate some limiting assumptions of the network stack that we referred to in earlier sections.

3.1.3 Session Discovery Mechanism

The session discovery mechanism assists with translating the session label to information needed to participate in the session. This is done using the `translate` primitive. A mapping is added when a process registers a session, using the `register` primitive. We refer to the process that registers the label as the initiator. The mapping is removed when an authorized participant, revokes the label (using the `revoke` primitive).

Any process that wishes to participate in a session can discover the the initiator's endpoint and subsequently join the session. The only requirement is for the intended participant to know the label that it wants translated. We can imagine a hierarchical naming scheme similar to that of domain names representing communication sessions. Several established mechanisms exist that address similar problems — e.g., DNS [13]. Such solutions can be adapted to serve the purpose of session registries. Further investigation of session discovery mechanism is left for future work.

In case the participants do not wish to register their session mappings, they can communicate without doing so, however this assumes that information about the session is distributed in some other way.

3.2 Negotiation of Configuration

As shown in Figure 1 the session abstraction also maintains a control flow between participants. This control flow provides a control space for session-layer signaling as well as possible extensions that may be built on top of the session layer.

We envision the control channel being used to exchange *verbs*, along with relevant payload. Verbs are requests that together form the control-channel protocol. Peer stacks receiving the message, would decide whether to act on the verb (after authenticating the message; the authentication mechanisms can be designed to ensure both authenticity and integrity). If the stacks recognize the verbs, they can choose to take action; if they don't, they may discard the message without disrupting communication. The control messages also include the source, session label and a transaction ID. This information enables SLIM to determine the origin of the verb, the session it is affiliated with and a unique ID to distinguish between requests. These pieces of information are considered mandatory components of a control message, with an optional component of payload (relevant to the verb).

Listing 1 is an example of the `refresh` verb, represented using JSON. The verb is intended to request the peer stacks to update their end-point label mappings to network addresses, as the source may have moved between subnets resulting in a change of IP address. This would result in setting up a new transport connection if the existing transport is not be valid anymore. Applications will not be aware of this adaptive behavior, since SLIM exposes the flow abstraction to the application and not the underlying transport connection. The transition, labeled as `reconfigure`, in Figure 3, reflects such reconfiguration.

```
{
  "VERB" : "refresh",
  "SOURCE" : "alice.node",
  "TRANSACTION_ID" : "1872",
  "SESSION_LABEL" : "a.session",
  "AUTH_TOKEN" : "h@kl#S"
  "TIMEOUT" : 20
  "PAYLOAD": {
      "list": [{
      "END_POINT_LABEL" : "alice.node",
      "IP" : "192.0.1.222",
      "PORT": 5432
      }]
  }
}
```

Listing 1: An example of a *verb* and its payload, represented in JSON, describing a request to refresh the end-point label mappings.

The ability to negotiate parameters of the session enables configuration and reconfiguration of communication. For example, before setting up, the end points can negotiate to authenticate participants, to encrypt allowing privacy, or to apply compression to minimize bandwidth usage. Generic functions with configurable parameters, such as `authenticate` and `enable_compression`, can be exposed as primitives to the application. Other primitives, such as `refresh`, may remain internal to the session-layer.

Being able to exchange verbs over there control channels enables extensibility, since we may define new verbs in future to introduce new functionality. Researchers and developers can add to the vocabulary and provide supporting implementations to extend SLIM and subsequently the network stack.

3.3 Data Transfer

The data transfer services are simple `read` and `write` primitives that allow the application to read from and write to flows. Based on the characteristics of the flow (as defined by the application), they may represent streaming data or a series of messages with defined boundaries.

The data flows are mappings of logical constructs onto underlying transports. While flow abstractions allow us decouple flows from underlying transport, they do not duplicate the semantics of the transport. For example, TCP is a best effort in order, reliable transport. Because a flow is mapped onto TCP, the logical construct is not required to implement ordered delivery or reliable transport. In order delivery and reliable transport semantics are provided by TCP. The flow construct simply maps the stream onto TCP. In case of a loss of transport connection (e.g., due to mobility), a new transport connection can be setup underneath and the flow mappings updated. The application will be oblivious to this adaptation as the session layer can manage the update seamlessly. Although this would require that the session layer maintains buffers while the updates take place, to avoid any loss of data.

The data flows do not expose primitives that exhibit richer functionality as with protocols like WebRTC.

3.4 Session vs. Transport Semantics

From the discussion above, we see how the semantics of managing the session can be envisioned without conflating them with the transport semantics. This is in contrast to the socket abstraction, discussed in Section 2, which conflates the roles of both into one. We see that all the session-related concerns can be dealt with independent of the underlying transport. Whether the concerns are about creating a session, configuring it to include participants and communication flows, reconfiguring the session and or flows to accommodate change of operating environment or preferences, or gracefully tearing the session down, all are independent of the underlying transport semantics. In case of TCP, it may continue to fulfill its responsibility of transport semantics, that is in-order and reliable delivery as well as congestion and flow control to enable network adaptation.

Separating these responsibilities allows us to realize optimizations that we discussed earlier. For example, once a connection has been established to the host, we could use parameters associated with it to bootstrap any new connection to the same host, without going through the three-way handshake. This may allow us to send data along with the first TCP segment, avoid the slow start phase, and benefit from knowledge of network performance parameters associated with the existing connection.

With multi-party communication a session may include more than one flow between participants, therefore the question of congestion control per session is not as simple as in the case of a single, independent transport connection. Instead, here we may want to consider all the flows forming the session as a whole. Although, for our current prototype we do not implement such a mechanism — and we depend on congestion control independently maintained by each flow — it is possible to implement a holistic mechanism such as Congestion Manager [7] or that used by MPTCP [20].

4. COMMUNICATION PATTERNS

Below we show how well-known communication patterns [19] may be represented by the session abstraction.

4.1 Client Server

A process can also act as a proxy for a group of processes. The session abstraction uses this understanding to describe a request/reply model of communication between a pair of processes. Consider for example a user accessing a social-media service. In this client-server communication pattern, the client process acts a proxy for the human, with a single data flow to the server process. On the other hand the server process acts as a proxy for a multi-tiered social-media service implemented by multiple services/processes as an enterprise application. Here the client's view of the session represents the conversation between the client and the server process. Where as the server's view of the session includes its interaction with the multi-tiered application processes as well as the client. The simple case of a single client and a single server process can understandably be represented as a degenerate case of the previous example.

4.2 Peer to Peer

We imply peer to peer communication when processes are not confined to client and server roles, instead they may perform any of them. Here too the session abstraction describes the communication as two processes exchanging data over logical flows, as in case of the traditional client server model.

4.3 Publish Subscribe

Consider the case of a video streaming service in the cloud, designed to use a publish-subscribe model. The content distribution process may register a session to publish its willingness to communicate. The processes interested in receiving the video content may subscribe by joining the registered session. Thus, a registered session would clearly describe the publish-subscribe model of communication, where the publisher has a view of the flows to the participants while the subscriber maintain the session with a flow to the publisher.

4.4 Survey

The survey model has similarities with the publish-subscribe model. Here one process makes a request, which is answered by multiple participants. The surveyor process may register a session which participants may join to respond to the requests. The session abstraction would describe views of all the participants. Consider for example a collector in the finance application mentioned earlier, polling data from the stock exchange. The session abstractions would represent the view of communication for the collector engaging the data sources over multiple, independent flows, as well as the view of the individual sources, each with a single flow to the collector.

4.5 Pipeline

The pipeline model of communication is a limited version of the peer-to-peer model, where a process can only assume one role — either be a producer or a consumer of information. Since the session abstraction successfully describes the peer-to-peer model, describing the pipeline model is the same, except for the limitation of unidirectional communication.

4.6 Broadcast

The broadcast model represents multiple participants engaged in the same session, where all participant receive data from a sender. Here communicating over a flow implies communicating with all participants. Such a model does not guarantee ordering.

An Example with Multiple Participants: Imagine a session where a video streaming service broadcasts a soccer match with video streams to multiple devices and match statistics to selected devices, while serving the same user; this is a publish-subscribe communication pattern. A chat application is also available with a peer-to-peer communication pattern. Note that the session abstraction does not confine its use to a particular model; the session abstraction simultaneously exhibits behavior of several different models — i.e., peer to peer as well as publish and subscribe models. Here we have multiple participants interacting as part of the same session.

5. PROTOTYPE IMPLEMENTATION

We implement the prototype session layer as a user-space library in C. The application interface, control channel, and the session registry is implemented in 1929 lines of code.

Each participating process, using SLIM, maintains its view of the session with the flows to peer processes. Identities of the endpoints are also maintained as part of the session state.

A logical flow exists between the processes, depending upon the communication pattern chosen by the application. Solutions, such as NORM [6] or DDS [1] that enable multicast communication may be used to realize the publish-subscribe, survey and broadcast pattern. For our proof-of-concept implementation, we use a naive approach of all-to-all connectivity to instantiate the aforementioned patterns. Other patterns such as client-server and pipeline are implemented by a straightforward mapping of the flows to transport connections.

The application views the flow abstraction as a logical construct to exchange stream- or message-oriented data. The mapping of the flow to the underlying transport as a stream or message is dictated by the developer's preferences. After a participant joins the session, a control flow is created. Data flows are added with the `add_flow` primitive. The implementation of a control flow is similar to that of a data flow, with the only exception that it is not directly exposed to the application.

Each flow is identified by a unique flow ID, so that it may be mapped appropriately to a transport connection. The flow ID is mapped to the file descriptor exposed by the socket API. With our proof-of-concept implementation, the bookkeeping for patterns such as broadcast is done by the session-layer. The mapping of the flow to underlying transport connections between peers is handled transparently.

We leverage our experience from our previous work [8, 11,12] and create a mapping between the sequence space of the logical flow and the transport connection. When a transport connection ends prematurely, we create a new connection that is used to map the data flow. This indirection allows the session layer to enable resilience and hide the process of adaptation from the application.

The session maintains an internal session identifier and a human-readable label, which may be published in a session registry. Upon registration, the members' endpoint details are listed along with the label.

The `register`, `push_update`, `revoke`, and `translate` primitives enable interaction of the session layer with the registry and ensure that it is kept up to date.

6. DISCUSSION AND EVALUATION

In this section we evaluate SLIM and the value it offers.

6.1 Support for Communication Patterns

In Section 4 we explain how SLIM may be used to describe common communication patterns. SLIM does not guarantee the order of messages between participants in broadcast, survey and publish-subscribe patterns. Note that there is a possibility of two participants writing to the flows in aforementioned patterns at the same time. If the application chooses to use message-oriented flows, the payloads would have defined boundaries. In this case, the messages may not arrive in the same order that they were generated. The session-layer does not define message boundaries in case of stream-oriented flows, as some applications may need them while others may not. The application is expected to define message boundaries if it needs to determine the source of the messages, or choose to use communication patterns such as client-server, peer-to-peer or pipeline. The session layer guarantees correctness of data exchanges between participants in that writes from different sources will not be interleaved.

6.2 Potential for New Paradigms

The use of the control channel along with support for multi-party communication has the potential for redefining communication paradigms. Consider for example, the use of load balancers. Instead of having the load balancer distribute incoming connections to replicated services (based on some criteria), a service may direct a client to authenticate itself with a dedicated service, using the public-key infrastructure. The (dedicated) authentication service can then confirm the authentication and from there onwards, the client and the service may use symmetric keys. This entire conversation would be part of the same session, where the service invites the authentication service, which then completes the assigned task and leaves.

6.3 Transition Cost vs. Value

SLIM provides the means to describe communication with the help of the session, flow and end-point abstractions. It enables configuration and reconfiguration, which assists with features such as mobility and also supports communication patterns involving multiple participants. With SLIM managing the conversation and the bookkeeping necessary to support such features, the developers are free to focus on the application features rather than implementing the underlying plumbing that enable these features. Using SLIM is feasible since developers already use user-space libraries (e.g., Java SDK) instead of directly accessing the sockets API, thus the disruption in minimal. Also, the control channel featuring verbs enable future extensions to the layer and therefore developers may extend the session-layer implementation themselves if they feel the need to do so.

Developers who do not want to make significant changes to the application implementations for multi-party communication, can use the two-party communication primitives of the session layer API. As the implementation is backwards compatible and these primitives have strong parallels with the socket abstraction, the transition cost is minimal. In this case, although the application will not be using additional primitives, it will receive benefits of resilient communication for free. Thus there is little cost of migration for legacy applications and stacks. For a small cost of modifying applications, there is a substantial increase in functionality. Note that most of this cost could be borne the session-layer library on behalf of the application.

6.4 Backwards Compatibility and Portability Effort

The session layer includes a set of primitives with parallels to the socket abstraction [21]. This minimizes the porting effort since the socket primitives need to be simply replaced with session counter parts. Here a process states its willingness to attend calls from other processes by issuing `await_call`. When contact is made by a calling process using the `call` primitive, the session is established by attending the call. Either process can `read` and `write` to session or individual flows based on the application logic, before the session is terminated by calling `end`. As with accept, the `await_call` generates a delegate session, while the listening session continues to wait for additional contacts.

6.5 User Space vs. Kernel Implementation

The transport layer is implemented within the kernel for variety of reasons including cross-layer communication and performance. Although our prototype implementation is in the user space, we envision the release version to be implemented in the kernel and its functionality exposed with an interface accessible by applications. Supporting functionality, which does not lie on the critical path, may be implemented in the user space. However, we have not explored these possibilities yet.

6.6 Performance of Implementation

The session and flow management as well as negotiation of configuration primitives fall along the control path, where as the read and write primitives fall along the data path. Thus, once communication is setup and configured, the management and negotiation services are not involved in the communication process. This suggests that any overhead that may be observed is related to connection management and not data transfer. Since the setup cost of a SLIM session with a flow, involves instantiating two flows (one for session control and the other for a data flow), the initial cost is twice that of a TCP connection setup. All subsequent flow instantiations cost the same as setting up a TCP connection.

6.7 Middleboxes

An important consideration of adoption and deployment is how the added functionality of session layer interacts with middleboxes (e.g., firewalls and NAT). We know from existing research [10,12] that traffic with custom transport headers is almost always dropped. For extensions to traverse through middleboxes, their headers must look like TCP on the wire. The only scenario where the existence of a middlebox may interfere with the session layer implementation is during: 1) creation of a data flow and 2) creation of the control flow.

In setting up a data flow, there will be no hindrances since all `add_flow` requests go to the port that the application exposes. If the middleboxes allow connection requests to go through for legacy request, they would also allow our requests as well. Since the control flow is implemented over a known port, it assumes that middleboxes would allow traffic to the port to pass through, otherwise, the connections for control flow will fail and the session-layer implementation will fall back to behavior of legacy/socket API.

7. RELATED WORK

Several attempts have been made to introduce high-level abstractions that mitigate the limitations of the network stack and propose extensions. We discuss some of these proposals below.

The Stream Control Transmission Protocol (SCTP) [18], defined in 2000, is intended to be an alternative transport protocol. It enables multi-homing in that the application may choose to send data via one interface or the other. The protocol, unlike TCP, proposes the use of message boundaries. In spite of supporting multi-homing, SCTP has not been adopted widely perhaps because: 1) it uses headers that are different from TCP and therefore middleboxes do not let its packets through [10]; 2) it is not backwards compatible with legacy applications or TCP/IP stacks; and 3) it only supports two-party communication and does not consider the scenario where multiple processes may be involved in a session.

TESLA, a transparent extensible session layer architecture for end-to-end network services [16], proposed in 2003, presents notable session layer services that are based on a flow abstraction. The authors propose the use of flow handlers to implement higher-layer and end-to-end services (e.g., encryption). Although TESLA defines a flow abstraction and looks like TCP on the wire, it was not widely adopted because unlike SLIM: 1) it is not backwards compatible with legacy applications, unless a stub is used for interposition, which inhibits deployment; 2) it assumes that all networks stacks implement TESLA as a session layer service and therefore it is not backwards compatible and incrementally adoptable with existing network stacks; 3) it does not propose a session abstraction and therefore does not enable a representation of the conversation between processes; 4) it does not support communication between more than two processes and 5) does not support fault tolerance or mobility.

Structured Streams [9], in 2007, proposes a transport abstraction as an alternate to TCP. The intent is to create associated child transport streams from existing transport connections while incurring minimal cost. It allows the application to have parallel streams. The proposed reliable stream abstraction, has not been widely adopted, perhaps because unlike SLIM: 1) it is not like TCP on the wire and therefore middleboxes do not let the traffic through unless it is tunneled through another transport protocol (e.g., TCP or UDP), which takes away much of the advantage of the abstraction; 2) it doesn't support multi-homing; and 3) it does not support communication between multiple participants.

SERVAL, an end-host stack for service-centric networking [14], proposed in 2012, put forward a service-access layer to: 1) enable end-points to use multiple network addresses, 2) to be able to migrate flows across interfaces, and 3) to create multiple flows for communication. SERVAL suggests that the proposed layer be between the transport and IP layer to reduce the coupling between the two and enable the features listed above. The fundamental limitation in the adoption is that the traffic does not look like TCP on the wire. Finally, the research does not present an abstraction that represents the conversation between processes or support multi-party communication; the focus of SERVAL is towards a service abstraction, not a communication session abstraction.

Multipath TCP [20], proposed in 2011, suggests extensions to TCP to support multi-homing, fault tolerance and flow migration. Since the proposed extensions are based around TCP, they are backwards compatible. To use the extended functionality the applications must be modified; using the legacy API does not translate into access to all features. Our proposal SLIM has similar attributes — to use the extended functionality, the application has to use the extensions, otherwise, the application will only receive some of the benefits. Multipath TCP does not propose a session or another communication abstraction. There is no support for multi-party communication. Multipath TCP does support reconfiguration of flow setups for some use cases for example flow migration, but not general reconfiguration. The focus of the proposal is primarily on extending TCP and not about higher-level abstractions. It is beginning to gain traction [4].

Except for Multipath TCP, all proposals have not been widely adopted. It appears that the foremost reason for limited adoption is the resistance to move away from TCP due to its widespread deployment; TCP's success, it seems, is a big hurdle in the evolution of network communication [10]. Other reasons for limited adoption range from lack of backwards compatibility, to significant porting effort and limited scope of the solution such that the value offered is far less than the transition cost.

8. SUMMARY AND FUTURE WORK

In this paper we propose an extensible session-layer intermediary that extracts the duplicate functionality from modern networking libraries and provides the means for future extensibility to the network stack. SLIM enables mobility, multi-party communication, and dynamic reconfiguration of the network stack in a straightforward and elegant way. SLIM includes an out-of-band signaling channel, which not only enables reconfiguration, but also allows for incremental evolution of the stack. We present how SLIM enables variety of communication patterns. We briefly describe features of a proof-of-concept implementation. We also compare the proposed session-layer and its implementation with existing solutions and explain possible reasons why these solutions have not been widely adopted.

We plan to add verbs to the control channel to enable extensions to the stack (e.g., process migration between end-points). We also plan to conduct an empirical performance evaluation of the implementation and identify bottlenecks.

9. REFERENCES

[1] Data Distribution Service. http://www.omg.org/spec/DDS/, 2014.

[2] Google Cast. https://developers.google.com/cast/, 2015.

[3] iOS and OSX Handoff. https://developer.apple.com/handoff/, 2015.

[4] iOS: Multipath TCP Support in iOS 7. http://support.apple.com/en-us/HT201373, 2015.

[5] QUIC, a multiplexed stream transport over UDP. https://www.chromium.org/quic, 2015.

[6] B. Adamson, C. Bormann, M. Handley, and J. Macker. NACK-Oriented Reliable Multicast (NORM) Transport Protocol. RFC 5740 (Proposed Standard), 2009.

[7] H. Balakrishnan, H. S. Rahul, and S. Seshan. An Integrated Congestion Management Architecture for Internet Hosts. In *ACM SIGCOMM*, 1999.

[8] E. Brown, M. Gardner, U. Kalim, and W. Feng. Restoring End-to-End Resilience in the Presence

of Middleboxes. In *IEEE ICCCN*, 2011.

[9] B. Ford. Structured Streams:A New Transport Abstraction. In *ACM SIGCOMM Computer Communication Review*, volume 37, pages 361–372. ACM, 2007.

[10] M. Honda, Y. Nishida, C. Raiciu, A. Greenhalgh, M. Handley, and H. Tokuda. Is it Still Possible to Extend TCP? In *Internet Measurement Conference*. ACM SIGCOMM, 2011.

[11] U. Kalim, E. Brown, M. Gardner, and W. Feng. Enabling Renewed Innovation in TCP by Establishing an Isolation Boundary. In *8th PFLDNeT*, 2010.

[12] U. Kalim, M. Gardner, E. Brown, and W. Feng. Seamless Migration of Virtual Machines Across Networks. In *IEEE ICCCN*, 2013.

[13] P. Mockapetris. Domain names - implementation and specification. RFC 1035 (INTERNET STANDARD), 1987.

[14] E. Nordström, D. Shue, P. Gopalan, R. Kiefer, M. Arye, S. Y. Ko, J. Rexford, and M. J. Freedman. Serval: An End-host Stack for Service-centric Networking. In *9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12. USENIX Association, 2012.

[15] L. Popa, P. Wendell, A. Ghodsi, and I. Stoica. HTTP: An Evolvable Narrow Waist for the Future Internet. Technical Report UCB/EECS-2012-5, University of California, Berkeley, 2012.

[16] J. Salz, A. C. Snoeren, and H. Balakrishnan. TESLA: A Transparent, Extensible Session-Layer Architecture for End-to-end Network Services. In *USITS*. USENIX, 2003.

[17] A. Snoeren, H. Balakrishnan, and M. Kaashoek. Reconsidering Internet Mobility. In *Hot Topics in Operating Systems*, 2001.

[18] R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, and V. Paxson. Stream Control Transmission Protocol. RFC 2960 (Proposed Standard), Oct. 2000. Obsoleted by RFC 4960, updated by RFC 3309.

[19] A. S. Tanenbaum and M. V. Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall, 2nd edition, 2006.

[20] D. Wischik, C. Raiciu, A. Greenhalgh, and M. Handley. Design, Implementation and Evaluation of Congestion Control for Multipath TCP. In *8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11. USENIX Association, 2011.

[21] G. R. Wright and W. R. Stevens. *TCP/IP Illustrated*. Addison-Wesley Professional, 1st edition, 1995.