SLIM: Enabling Transparent Extensibility and Dynamic Configuration via Session-Layer Abstractions

Umar Kalim, Mark K. Gardner, Eric J. Brown, Wu Feng
Virginia Tech
{umar,mkg,brownej,wfeng}@vt.edu

ABSTRACT

Increasingly, communication requires more from the network stack, e.g., seamless handoff and synchronization of state between multiple participants. Due to the lack of support for desired functionality, networking libraries are created to fill the void. This leads to considerable duplication of effort and complicates cross-platform development. Furthermore, the means for extending legacy protocol stacks is largely exhausted (e.g., the TCP options space in the SYN message is mostly allocated), making the addition of future extensions much more challenging.

In this paper, we tease apart elements of session management that are currently conflated with the transport semantics in TCP and highlight the need for sessions in contemporary communications. Next, we propose session, flow, and endpoint abstractions that lead to a clearer description of advanced communication models. This effort results in an extensible session-layer intermediary (SLIM) that leverages the above abstractions to support the additional functionality needed by modern applications, such as mobility, communication between two or more participants, and dynamic reconfiguration. SLIM's approach also provides the means for future extensibility of the network stack in a backward-compatible way, thus enabling incremental adoption.

CCS CONCEPTS

•Networks → Network architectures; Session protocols;

ACM Reference format:
Umar Kalim, Mark K. Gardner, Eric J. Brown, Wu Feng. 2017. SLIM: Enabling Transparent Extensibility and Dynamic Configuration via Session-Layer Abstractions. In *Proceedings of ACM/IEEE ANCS, China, 2017,* 13 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

To achieve greater functionality, modern applications require more from networks stacks. When the desired functionality is not available, developers are burdened with creating support mechanisms to mitigate the limitations of the network stack and enable modern use cases. This is in addition to developing features for the applications. Apple Continuity [1] and Android OS [3] are examples of such cases. They both implement support for synchronizing state between processes on multiple hosts as well as resilient communications to enable seamless handoff. Such duplicate functionality makes software development for modern cross-platform solutions far more complex than it needs to be.

Part of the reason for the difficulty is due to the accumulation of technical debt in the form of limiting assumptions, simplistic abstractions, and (once expedient) implementation shortcuts. For example, the assumption that network addresses do not change during communication (i.e., the duration of a connection is shorter than the lifetime of network address assignment) led to identifying connections by network addresses. However, this makes mobility complicated and fragile. Similarly, the socket abstraction is too simple to describe communication involving more than two participants, which we discuss further in § 2.

To realize modern use cases, there is a need for abstractions that enable greater functionality (e.g., mobility, migration, dynamic configuration), the implementations of which must allow for further extensions as well as a path for incremental adoption. Notable attempts have been made to minimize duplication of effort, enable extensibility, or mitigate limitations of legacy stacks. These vary from higher-layer abstractions (e.g., [29, 31]), transport protocols (e.g., [11, 34, 39]), and network-layer proposals (e.g., [22, 23]) to clean-slate designs (e.g., [12, 35]). While these approaches highlight the need for greater functionality, none of them have achieved significant adoption, except for Multipath TCP [39]. The reasons for this include narrow scope, limited abstractions, lack of backward compatibility, high transition cost vs. value, or addressing the symptoms and not the root cause of problems.

We propose an extensible session-layer intermediary (SLIM), which provides session semantics to support innovative communications and enables future extensions to the network stack. SLIM (1) uses *session*, *flow*, and *endpoint* abstractions to support current and future communication models; (2) enables interactions between two or more participants and support for mobility; and (3) provides an out-of-band channel for the exchange of control messages that enables dynamic reconfiguration of ongoing communications.

Our contributions in this paper are as follows:

- A characterization of challenges when using legacy communications for modern use cases, such as conflation of session and transport semantics, and therefore, the need for separate session, flow, and endpoint abstractions (§ 2);
- The architecture of an extensible session-layer intermediary (§ 4) and the roles of the session, flow, and endpoint abstractions (§ 3) that support typical and advanced communications; and
- A prototype implementation of SLIM (§ 5) that exposes an API for the aforementioned abstractions, followed by an

ACM/IEEE ANCS, China

DOI: 10.1145/nnnnnnn.nnnnnnn

evaluation of the session layer and the backward-compatible extensions enabling incremental adoption (§ 6).

2 CHALLENGES

Modern communications have outstripped the extensibility of network stacks [13] leading to a growing notion that network transport has ossified and that HTTP is now the "evolvable narrow waist" [25]. Hope for evolution of the network stack seems to be fading away and is instead being replaced by application-layer "kludges," which are needlessly complicated due to attempting to fix problems at the wrong layers. In spite of a large corpus of research (§ 7), we see that enabling greater functionality and encouraging adoption continues to be a challenge [28]. The following aspects play a significant role in this challenge:

Lack of Backward Compatibility and Limited Extensibility: It is widely accepted that today TCP supports nearly 90% of the Internet traffic [18]. With such tremendous momentum behind legacy networks, any proposal for extensions that is not backward compatible or does not look like TCP on the wire will not stand a chance of adoption because of the significant transition cost. The proposals that are backward compatible are constrained by the means for extension, such as the shrinking TCP option space, particularly the option space of the SYN message during TCP connection setup [13, 15]. Thus, the key to introducing change would therefore be to enable incremental adoption, thereby facilitating a smooth evolution of network communication and providing a means for extensibility, which not only cater to modern use cases but also enable those that we have not considered yet.

Conflation of Session and Transport Semantics: Using the Socket API, legacy communications instantiate a transport connection. These communications are based on the end-to-end model. A file descriptor is created that serves as a handler for the transport connection [4]. The instantiation of a socket not only represents a transport connection, but also the communication session. Doing so implicitly ties the semantics of the session with the semantics of the transport connection and couples the life of the session to the lifetime of the TCP connection. If the transport connection fails, e,g., due to the change of network address, the session fails.

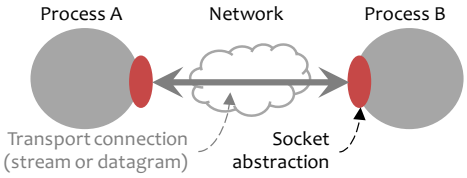

Figure 1: The socket abstraction, identified with a file descriptor, strictly tied to the underlying network interface.

Limiting or Missing Abstractions: Figure 1 shows that the file descriptor serves as a representation of the socket abstraction and a surrogate for the transport connection. It does not present an explicit representation of the communication endpoint, the communication streams between participants or the conversation as a whole. This lack of explicit abstractions is the reason that participants are identified using inferences when it is required to explicitly identify communication endpoints. Similarly, network interface labels and port identifiers are used to indirectly represent communication streams. The labels used are by definition strictly tied to the underlying networking interface identifiers, creating a strong coupling between concepts that span multiple layers in the network stack. In addition, current implementations assume that communications involve two participants alone. Moreover developers are expected to maintain bookkeeping about the conversation.

Thus, it is due to the lack of higher-layer abstractions that we see developers duplicating effort and implementing mechanisms to support greater functionality at the application layer (e.g., supporting mechanisms for state sharing/management [3]).

Limited Security Considerations: Typically, security considerations have been an afterthought when proposing network extensions [5]. Today, however, information security concerns are of paramount importance in network communications. Whether it is the matter of access control (e.g., identification, authentication, authorization), confidentiality, integrity, or non-repudiation, contemporary proposals must enable solutions that address such concerns; Proposals must not create impediments or introduce weaknesses in meeting information security goals. Designs that do not take such aspects into account, struggle in convincing the wider community to adopt their proposals.

Immutable Configuration of Communications: Legacy implementations assume that once communication is setup, the configuration will not change. In other words, it is assumed that the lifetime of the communications (i.e., transport) is the same as the duration of the conversations (i.e., session). Since device or service mobility is a very important capability in today's mobile and cloud computing environments, tying the lifetime of the session to that of an underlying TCP connection gets in the way of clients as they roam between networks and services as they migrate between data centers or cloud providers. Developers and service providers work around this issue but at great cost; working around the issue distracts them from focusing on their product's unique value and contributes to brittle network behavior.

Conclusions: With legacy TCP implementations we see that sessions are constrained by transport semantics. When these stacks were implemented, conflation of session and transport resulted in an efficient solution that contributed to the success of the Internet [8]. However, today this conflation is impeding the implementation of desirable services [28] and thus needs to be addressed. Sessions and transport connections need to be managed separately and explicitly. This requires separating and improving the abstractions used to describe communications. Therefore, we propose three additional abstractions beyond those provided by TCP: *session*, *flow*, and *endpoint*. These are described in the following section. These abstractions not only make it easier to support greater functionality (e.g., describing two or more participants in a conversation), but also enable other avenues of extensibility (e.g., enabling dynamic reconfiguration and mobility).

3 SESSION-LAYER ABSTRACTIONS

Here we describe the *endpoint*, *flow*, and *session* abstractions that form the session layer. We use Figures 2–7 in the descriptions that follow to illustrate a representative session involving three endpoints with two data flows and one control flow. Each endpoint may create or participate in one or more sessions. Each session

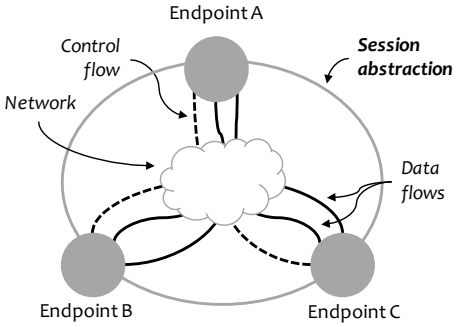

Figure 2: The session abstraction involving three participants, each with two data flows instantiated by the application. The control flow enables setup and reconfiguration.

may have one or more participants. Flows may or may not exist between endpoints.

3.1 Endpoint Abstraction

An endpoint is an entity participating in a conversation and represents a source and destination of communications. Note that the real endpoint is often the user with some process serving as proxy; in computer-to-computer communications, endpoints are processes. Our definition of an endpoint is in contrast with the definition of an endpoint in the socket API. The socket API defines the endpoint as an immutable entity associated with a 5-tuple ⟨*local IP, local port, remote IP, remote port, protocol*⟩ at the time of instantiation and made available to the process as a file descriptor [4]. The contrast is illustrated in Figure 3.

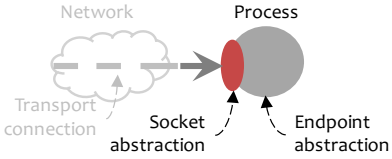

Figure 3: Contrast of endpoint and socket abstractions.

Modern use cases require that the association of the process and the host that houses the endpoint not be permanent. Instead, an endpoint should be identified by a label independent of the network address of its host. A process may change hosts (in case of service migration), may appear to change network attachment points (in case of mobility), or may want to use multiple network paths (and therefore potentially use multiple network interfaces). Therefore, to construct an endpoint label independent of the underlying layers, we draw inspiration from the Host Identity Protocol (HIP) [22] and use public keys as the foundation for building identifiers. Since private-public key pairs are uniquely associated with users or services [9], using the public key as the foundation of a unique label is prudent for modern communications.

As shown in Figure 4, we construct a unique endpoint label using a cryptographic hash [22] (or a fingerprint) of the user's, service's, or client's public key along with a suitable tag to distinguish between endpoints. Based on the mathematics of the birthday problem, we can conclude that with the hash size of more than 100 bits, we can safely assume that a collision will not occur until one quadrillion (i.e., 2^{50}) hashes are generated. For this reason, we chose 128 bits as the hash size, 24 bits as the distinguishing tag size and 8 bits for the hash-algorithm type, resulting in a label size of 160 bits (20 bytes) and allowing more than 16 million endpoints per key and 256 hash algorithms to choose from. Implementations may choose different label sizes based on the hash algorithm type and the number of endpoints allowed per key. Note that Figure 4 is the logical representation of the label and does not cater to implementation concerns. For convenience, we also use human-readable tags mapped to endpoint labels (e.g., `meetup.alice`).

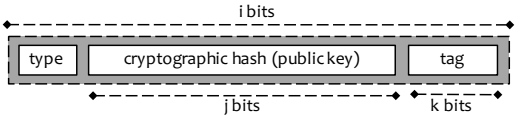

Figure 4: Endpoint label.

The endpoint labels may be translated to obtain information that describes how the endpoint may be contacted. In the case of TCP, this contact information would be an IP address and port number. We can imagine adapting the domain name system (DNS) [38] to be a translation service.

Deriving the endpoint labels from the public key enables intrinsic security in the design of communications. For example, access control services (of identification, authentication, and authorization) may be enabled by verifying digitally-signed endpoint labels, which we will discuss further in § 4.2. In the same vein, symmetric ciphers may be derived using associated private-and-public keys to enable confidentiality. Note that the use of PKI does not require deployment of additional infrastructure beyond what already exists.

3.2 Flow Abstraction

A flow represents a data exchange between a set of endpoints. It gives a name to the concept of communication but requires mapping onto underlying transport connections before communication actually occurs. Because flows are independent of transport connections, the concept of a flow can precede the creation of a transport connection and can persist after the transport connection has been closed. This separation allows us to distinguish between session and transport semantics. Doing so further enables reconfiguration of flows on the fly (and subsequently reconfiguration of underlying transports). This is illustrated in Figure 5 where a flow may be mapped onto a transport connection `p` and later mapped onto a transport connection `r` (e.g., when transport `p` is disrupted due to reconfiguration or migration of the client into a different subnet). Reconfiguration of transport connections is not possible when using the socket abstraction since it essentially means terminating and setting up a new instance with intended parameters.

We identify each flow with a human-readable label that is mapped to an opaque identifier, unique within the scope of the session. Initially, the endpoints have different opaque labels for the same flow. However, endpoints may exchange and agree upon flow labels for the duration of the conversation. We explain this further in § 4.2.

SLIM supports the notion of *data* and *control* flows, both shown in Figures 2 and 5. SLIM data flows are visible to the application; SLIM control flows are not. SLIM exposes data flows to the application as a means for exchanging data between participants. On the other hand, SLIM uses control flows to configure or reconfigure session-layer abstractions. For example, the control flow may be used to seamlessly create and destroy transport connections (as

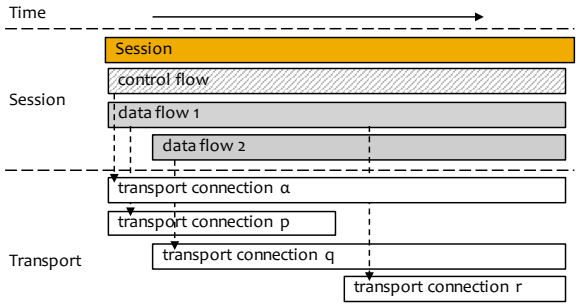

Figure 5: The flow abstractions and their mappings onto underlying transports in relation to time.

needed) to support mobility, resilience, and dynamic reconfiguration. It also uses control flows as an extension mechanism. We discuss control flows further in § 4.2.

The manner in which a flow is mapped onto a transport is guided by its structure. We define two types of structures for flows: 1) broadcast and 2) one-to-one. With a broadcast structure, the reads and writes to the flow involve all participants. On the other hand, a one-to-one structure suggests a point-to-point link between the participants. This is illustrated in Figure 6. Other forms or structures are worthwhile; however, we limit the scope of our work to these two common structures in this paper.

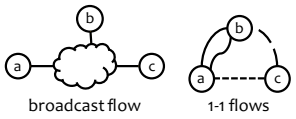

Figure 6: The structure of flows in relation to the endpoints.

3.3 Session Abstraction

A session represents the complete conversation between participants in an agreed-upon context. It encapsulates endpoints and flows that constitute the conversation and allows them to be reasoned about together. This is illustrated in Figure 2.

Each session is labeled with a session identifier, chosen by the endpoint initiating the session. As illustrated in Figure 7, the label consists of the initiator's endpoint label and a distinguishing tag. We chose the tag size to be 32 bits, which allows for about four billion simultaneous sessions per endpoint. As with the endpoint labels, different session tag sizes may be implemented. Our choice of 32 bits is intended towards accommodating a sufficient number of sessions per endpoint and thus future-proofing labels. For convenience, we also map a human-readable label to the session label — e.g., `org.meetup`. The session label must be globally unique if it is to serve as a publicly accessible session. This is implicitly achieved since the session label is based on the initiator's endpoint label, which itself is based on a globally unique public key. Session labels may be published and publicly visible or may remain unpublished. We discuss aspects of session labels registration further in § 4.1.

Each endpoint maintains a local view of the session. The identities of the endpoints are maintained as part of the session state. In addition, identities and descriptions of flows originating from or terminating at the end point are also recorded as part of the session state. A local session state available at an endpoint would not include information about flows that the endpoint is not involved with (e.g., endpoint a would not be aware of the 1-1 flow between endpoints b and c in Figure 6).

Figure 7: Session label.

4 SLIM'S ARCHITECTURE

As Figure 8 illustrates, SLIM exposes an API while providing three sets of services to the application to assist with communication setup and management. These services fulfill three roles: 1) session management, 2) negotiation of configuration, and 3) data transfer. SLIM uses the underlying transport services to realize the session abstractions.

SLIM does not force the application to gain network access through it, as shown in Figure 8. The reasons for this are two-fold. First, it highlights that SLIM is designed for incremental adoption and therefore does not force applications to use SLIM to access network services; applications may continue to use the legacy socket API if they choose to do so. Second, it emphasizes that SLIM is primarily engaged in communication setup (in the beginning) and management (in case of reconfiguration) and does not interfere with data exchange. As we explain later, SLIM is a "pass through" for data exchange once communications are setup.

4.1 Session Management

SLIM session services allow sessions to be instantiated, configured, reconfigured, and torn down. Figure 9 provides a state transition diagram illustrating relationships between session primitives and states of the session abstraction.

SLIM's state-transition diagram may appear to share similarities with the TCP state-transition diagram [26]. This is not because SLIM's design is a derivative of TCP's implementation; rather, as explained in § 2, legacy TCP implementations conflate session and transport semantics, hence its state-transition diagram has aspects relevant to session management. It is these shared aspects of session management that are reflected in both the state-transition diagrams.

Session-Related Primitives: Typically, an endpoint expresses its willingness to communicate and is then joined by other endpoints. An endpoint *creates* a session to encompass the intended communication and *awaits* contact from other endpoints. Later, other endpoints *join* the session and one or more data flows are *added* to begin communication. Alternatively, endpoints may be *invited* to participate in a session. Endpoints can *leave* the session at any time without disrupting the ongoing communications.

Joining a session requires knowledge of the session label. The session can be publicly advertised by *registering* the label with a session discovery mechanism. An endpoint wishing to join the session looks up the label from the session registry and requests a *translation* of the label into contact information. This contact information consists of the details necessary to reach the endpoint. An implementation of SLIM over TCP considers the network and port addresses as contact information and allows these to change during the lifetime of the session. Alternatively, the contact information for a session can be conveyed to an endpoint by other means (e.g., if the session is not registered publicly).

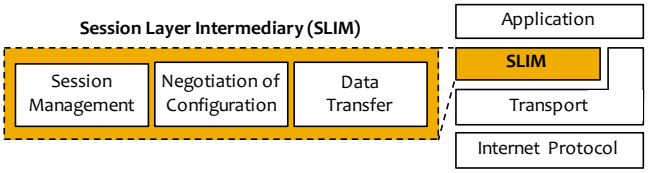

Figure 8: SLIM in relation to the network stack.

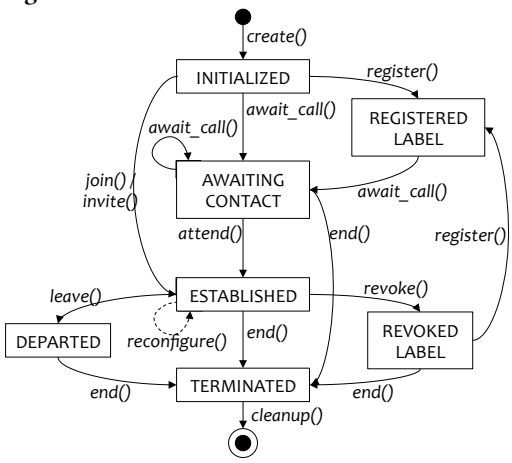

Figure 9: Session state-transition diagram.

Although highly unlikely, there is a possibility that an endpoint joining a session may face a collision of endpoint labels. In this case the collision is resolved by choosing an alternate distinguishing tag (see Figure 4). Further naming issues, e.g., multiple identities per endpoint, are beyond the scope of this paper and will be tackled as part of future work.

The session state recorded at the registry needs to be managed. The record is expunged when an endpoint *ends* the session, or it is maintained via a heartbeat mechanism that indicates the session's continued existence. Other primitives that assist with negotiation of configuration are discussed in § 4.2.

Flow-Related Primitives: The `add_flow` primitive creates a data flow within the session. It takes a named parameter *session* and optional inputs of *[structure]* and *[type]*. The structure defines the communication model for the conversation (e.g., broadcast or one-to-one). The type defines the mapping onto the underlying transport (e.g., stream- or message-oriented transport). There may be more than one flow between processes. Although flows originating from the same endpoint are independent of each other, yet they are associated with each other through the endpoint. This is relevant because flow reconfiguration primitives may be applied to the session as a whole instead of individual flows — e.g., to change underlying transport protocol from TCP to Multipath TCP. The `terminate_flow` primitive tears down the flow.

Session Registration and Discovery: The session registration and discovery mechanism assists with recording and translating session labels to the information needed to participate in the session. Upon registration, a mapping is created between the session label and at least one endpoint participating in the session. The mapping is removed when an authorized participant *revokes* the label.

We refer to the process that registers the label as the initiator. Any process that wishes to participate in a session can query the registry to acquire contact information about the participants to subsequently join the session. The only requirement is for the intended participant to know the label that it wants translated. For this purpose, the human-readable labels associated with session labels make this requirement relatively convenient to manage.

Mechanisms exist that address similar challenges of label registration and translation, which can be adapted as a session registry. For example, we envision a DNS [21]-like hierarchical LDAP [33] service that would not only meet timeliness requirements, but would also scale well. The use of hierarchical human-readable labels is expected to enable scalable solutions. Note that participants that do not wish to register their session can still communicate. However this assumes that contact details of at least one participant are available. An in-depth investigation of session registries are beyond the scope of this paper.

4.2 Negotiation of Configuration

In support of more sophisticated communications, SLIM uses the control flow to exchange commands (or control signals) between participating network stacks and peers. These commands, called *verbs*, include, for example, requests to suspend or resume a flow or change the underlying transport protocol.

Verbs: Control flows exchange verbs to configure communications. The facility to exchange control information or alert peers of a change in the communication context, along with the ability to reconfigure abstractions after communication has been setup, enables *dynamic configuration*.

For example, Listing 1 shows an example JSON representation of the `sync` verb requesting the peer stacks to update their endpoint to network address mappings (e.g., when the source moves between subnets). This may result in a new transport connection if the existing transport is not valid anymore. Applications will not be aware of this adaptive behavior, since SLIM insulates them from the underlying transport via the flow abstraction. The transition, labeled as `reconfigure`, in Figure 9, reflects such reconfiguration.

Required fields of the verb indicate the source of the request, the session label, a transaction ID, and an authentication token. These enable stacks to distinguish between requests and to ensure authorization. The remaining fields are verb-specific.

```
{"VERB"   : "sync",
 "SOURCE": "meetup.alice",
 "TRANSACTION_ID": "1872", "TIMEOUT" : 20,
 "SESSION_LABEL" : "org.meetup",
 "AUTH_TOKEN": "2N8ISiGELBzNw1sOunAxgOF3MrQF4ugf",
 "PAYLOAD"    : { "list" : [{
     "END_POINT_LABEL": "meetup.alice",
     "IP"   : "192.0.1.222", "PORT": 5432 }]}}
```

Listing 1: An example of a *verb* and its payload, represented in JSON, requesting an update of endpoint label mappings.

Activities (e.g., endpoints joining a session, reconfiguration requests through verbs) may be authenticated using the authentication tokens. These tokens represent digitally-signed endpoint labels, which can be verified as the endpoint label is based on the public key of the user, client, or service that the endpoint is representing [9]. Also, the authentication process creates an opportunity

for the recipient of verbs to pose compute-intensive puzzles before taking action and thus preempt denial-of-service attacks.

Much as network stacks disregard unknown TCP options and thereby facilitate incremental adoption of new extensions [26], the SLIM control protocol requires stacks to disregard unknown verbs. Currently, the set of verbs are limited to those needed to support migration and resilience (e.g., `sync`, `suspend_flow`, and `resume_flow` are useful for migration and recovery from disruption). Other verbs, such as `change_protocol`, allow on-the-fly reconfiguration of the underlying transport protocol.

Enabling Future Extensions: It is well established that sufficient TCP option space, particularly as part of the TCP SYN message, is no longer available for extensions that require space for control signaling [20]. To further add to the complexity, we know that middleboxes either do not allow TCP packets with custom options to traverse through them or strip custom options [6, 13, 16, 27]. The control flow provided by SLIM serves as a signaling channel for future extensions. For example, to implement support for process migration or state synchronization between peers, verbs may be defined and implemented over the control channel. New primitives may then be implemented and exposed to support future network services. Examples of such extensions include `change_protocol` and `sync`. In § 5 we explain how verbs are implemented with corresponding handlers. To add extensions, developers implement handlers that receive, consume, and act upon the verb and its parameters.

Context Management: In addition to allowing the application to indirectly use the control flow as a means for exchanging control signals, a *context manager*, also makes use of the control flow to exchange control signals between the network stacks. For example, when a network interface is disconnected, the transport connection using the interface would timeout. To avoid having communications fail, the context manager recognizes the change and triggers an instantiation of a new transport connection that uses an alternate interface, if one is available. This would also trigger an exchange of a `sync` verb to synchronize state.

Here we have considered one aspect of context management (i.e., resilient communications). However, there are other possibilities that can be explored to enable the network stack to be cognizant of its operating environment. An example of such context awareness may be to recognize that multiple network paths to peers exist through different network interfaces on the host and thus enabling multi-homing, fail-over, or redundant communications. Similarly the network stack may recognize the existence of an accelerator or a gateway in the network and enable configuration of communications — e.g., engaging SSL accelerators or interacting with captive portals. With session-based abstractions, SLIM serves at a suitable vantage point to realize such a variety of goals that include policy enforcement, and dynamic reconfiguration.

4.3 Data Services

Once communications are setup, the flow abstraction merely serves as an indirection to the transport connection. Thus, SLIM essentially acts as a pass-through for an application writing to or reading data from a flow, until there is a disruption (e.g., connection loss due to migration), or a verb is triggered to reconfigure communications — e.g., SLIM participates to restore communications.

5 PROTOTYPE IMPLEMENTATION

Here we discuss a prototype of the session-layer intermediary, SLIM, implemented as a user-space library in C. The application interface, session-layer primitives, control signaling, session registry and a shim layer for backwards compatibility with legacy applications are implemented in 3189 lines of source code (without comments).

5.1 Session State

In the prototype, a view of the session is maintained for each participating process. This session state includes details of participating endpoints, the flows that exist between them, the configuration and structure of the flows, the flow-to-transport mappings, the available network interfaces on the host where the endpoint resides, and the session type. In addition, the identities of the session, flows and endpoints and the mappings of those identities — e.g., endpoint label to location mapping (IP and port addresses) — are maintained as part of the session state.

During the lifetime of communication, the session state changes. The changes that are local to the endpoint need not be shared with the other participants (e.g., the sequence space mappings of the flows to underlying transport). However, the changes in state that are relevant to the entire session may be shared with participants to maintain consistency (e.g., endpoint label to location mappings). These updates are shared through the control flow (using the `sync` verb) and are initiated by the source of the change.

5.2 Data Flows

Data flows are added with the `add_flow` primitive. Each data flow is identified internally by a unique flow label and mapped to an appropriate transport connection (stream or message) depending on the flow type. The flow label exposed to the application is mapped to the file descriptor returned by the standard `libc` socket API.

5.3 Flow Labels and Greater Functionality

For legacy application behavior of the flow labels merely serves as a reference to the session-level concept of a flow. However, they play a significant role in supporting extensions to the stack, for example, in case of mobility. After the host servicing an endpoint moves to a different subnet, the network address assigned to the interface may change. This invalidates the transport connection that the flow was mapped onto (for simplicity, consider the example of two participants involved in the session). However, SLIM recognizes the change and instantiates a new transport connection, onto which the flow is mapped. The participants are able to recognize that this is the same flow since the `sync` verb initiated by the source triggers an update of mappings between the flow and new transport connection at the recipient. Note that this allows the communications to work in spite of the presence of middleboxes (e.g., NATs that may change the IP addresses associated of traffic flowing through them).

5.4 Structure of Flows

To realize a one-to-one structure for flows between endpoints, we have a correspondence between a flow and a transport connection. On the other hand, for a broadcast structure, we implement an all-to-all connectivity of transport connections between the endpoints, to which the flows are mapped. Note that the focus on broadcast and

one-to-one flows is not fundamental and that the entire spectrum of configurations can be supported. For example, for peer-to-peer communications, a broadcast structure might not be feasible and a Chord-like configuration [36] may be appropriate. However, we leave that exploration for future work.

5.5 Flow-to-Transport Mappings

The flow-to-transport mapping requires a mapping of sequence spaces between the two. The session-layer implementation delegates the responsibility of managing transport semantics (e.g., retransmission of lost bytes) to the underlying transport implementation. Doing so allows the book-keeping of delivered and undelivered bytes in the context of flows to be relatively straight-forward. We leverage our experience from our previous work [15, 16] in creating this mapping of sequence spaces. If a transport connection ends prematurely, a new connection is created and the data flow is mapped onto it. Without duplication of effort, we are able to use the mapping of sequence spaces to ensure that no data is lost during reliable transfers. This indirection allows the session layer to provide resilience and hide the messy details of adaptation from the application. Comprehensive details of the sequence space mapping are documented in our prior work [15, 16].

5.6 Control Flows

A control flow is created when an endpoint joins the session; the intention here is to have a control flow between endpoints participating in the session to allow exchange of control signaling. Note that in our implementation, a control flow[1] is the same as a data flow with the exception that the abstraction is not directly exposed to the application.

Control flows are used to exchange verbs, which are commands that trigger corresponding handlers. For example, the receipt of the *verb* `suspend_flow` triggers the launch of the corresponding handler `v_suspend_flow()`. After ensuring that the request is valid and authorized, the parameters are processed to effectively pause further writes and reads to and from the flow — e.g., this may be required when migrating between subnets for which new transport connections need to be established to resume connectivity. Note that extending SLIM to include additional verbs translates into registering the verb and the corresponding verb issuer and handler with the library. For example, the verb `sync` has its corresponding issuer `i_sync` and handler `v_sync` registered with the library.

Typical verbs are implemented as non-blocking requests. The recipient acknowledges the receipt with (success or failure) codes. However, this does not preclude implementation of verbs as blocking requests. Blocking requests are prudent where transactions may need to be rolled back if unsuccessful. This behavior is implemented as part of the verb issuer.

Context Manager: A context management thread runs in the background with a low profile. The intent here is to allow the session layer to be cognizant of the circumstances in which communication is taking place.

In the prototype implementation, the context manager creates a `netlink` socket and listens for network interface events through `RTMGRP_LINK`. If the interface is disconnected (i.e., the link goes down) and an alternate interface is available, the context manager triggers the setup of a new transport connection over the alternate interface and issues a `sync` between the participants to resume connectivity. Alternate solutions that listen for OS events may be used instead, to solve this problem. We plan to evaluate such methods with SLIM's implementation as a kernel module. We implemented this feature as an example of other possibilities that range from variety of dynamic configurations to applications of policy enforcement (e.g., assisting captive portals).

[1]The management of control flows and execution of verbs is implemented in independent threads. This avoids interference with the application's execution thread.

5.7 Session Labels and Registry

The session maintains a session identifier and a human-readable label, which may be published in a session registry. We implement a rudimentary LDAP service to act as a session registry and anticipate that a production deployment would involve a DNS [21] like hierarchical LDAP [33] service. 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. Further details are available in the technical report [17].

5.8 Support for Legacy Applications

Since the prototype implementation of SLIM is available as a user-space library, applications using the API simply link to the library for network access. To support legacy applications, which use the Socket API, we have implemented a shim layer, which intercepts Socket API calls using `LD_PRELOAD` and then maps these to the SLIM API. This is illustrated in Figure 10. Doing so allows SLIM to be backwards compatible with the legacy applications.

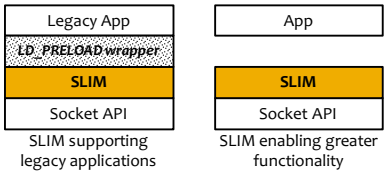

Figure 10: SLIM in relation to legacy applications and those using the library.

Understandably, with the use of the shim layer, legacy applications will not be able to make use of all the features supported by SLIM, other than those that we implement as part of the shim layer. On the other hand, applications that are programmed using the SLIM API are able to benefit from the greater functionality that the library enables. Note that the illustrations in Figures 8 and 10 highlight that SLIM serves as a wrapper around the Socket API.

5.9 Support for Mobility, Migration, and Resilient Communications

Leveraging our experience from our prior work [16] we highlight how separation of session and transport semantics can enable *migration* of services between networks without disrupting communications. We demonstrated a live migration of a virtual machine (hosting an SSH server) from Blacksburg, Virginia on the East Coast to Sunnyvale, California on the West Coast. In spite of the migration to a different network, the client application remained connected

to the service and continued to operate successfully [14]. This was only possible because the session-layer flow was able to update its mapping from one transport connection (to the service hosted in the Blacksburg network) to another after the migration (in the Sunnyvale network) as illustrated in Figure 5. The indirection introduced due to separation of session and transport semantics allowed SLIM to enable resilient communications.

We also highlight the ability of SLIM to demonstrate resilience, by leveraging our experience from prior work [6, 15], where we interrupt communications by physically disconnecting the network between a legacy client application (Video LAN) and a media streaming server. The physical disconnection resulted in the network interface going down. Upon reconnection, the context manager recognizes the change in interface status and triggers a synchronize event. Following which the media continues to stream to the client application without losing the session. This would not have been possible with legacy TCP, for which the socket would become invalid after the disconnection.

6 DISCUSSION AND EVALUATION

With explicit session management along with the built-in control flow, SLIM provides support for several innovative uses of the network not well served by legacy TCP, including resilience, migration, and stack extension. It also has the potential for simplifying and refining existing communications. In this context, we discuss and evaluate SLIM's contributions.

6.1 Separation of Session and Transport Semantics via Session-Based Abstractions

In the previous sections, particularly in § 2 and 3 we made a case in favor of separating session and transport semantics as the means to rejuvenate innovation in network stacks. We explained that legacy stack implementations have conflated session and transport semantics within the Socket API and why this makes future innovation a challenge. For example, from the application developer's perspective, when using legacy implementations the conflation of semantics results in the transport connection appearing as if it were the entire session; the beginning and end of the connection is the beginning and end of the session, when in fact the session and flow semantics are fundamentally independent of how the transport transfers data across the network. Thus, a feasible way forward is to decouple these semantics and subsequently avoid having the limitations of underlying layers permeate through to the applications; unless the session and transport semantics are decoupled, developers will be forced to implement modern applications with limited abstractions, they will continue to be constrained by the limitations of the underlying implementations, and they will continue to duplicate efforts in implementing session-management to support their use cases [1, 3].

With SLIM we are able to decouple session management from transport semantics and subsequently the limitations of the underlying implementations. As a result, the SLIM API only exposes session semantics to the developer and thereby simplifies the design of network applications and services. For example, unlike the transport implementations that couple transport labels to network labels, the session abstractions insulate themselves from the limitations of such cross-layer coupling and are therefore not limited by naming constraints. As a result, developers are relieved from the concerns of maintaining resilient communications, and instead focus on session semantics alone. We demonstrate the efficacy of this decoupling with our prototype implementation (§ 5) and through the virtual machine migration demonstration [14, 16], where we highlight support for mobility and resilient communications. This separation of session and transport semantics also helps in minimizing duplication of effort as we see developers tackling the same challenges over and over again [1, 3], which are in part addressed through SLIM. Table 1 summarizes SLIM's contributions along with other notable proposals, which we discuss further in § 7.

6.2 Enabling Greater Functionality

All communication models [37] benefit from the separation of session and transport semantics. Explicit session management opens avenues for innovation and extensibility. For example, explicit session management enables: sessions with more than two participants, session migration between hosts (in contrast to host mobility), adaptive configuration of communications, multihoming — explicitly binding flow-to-transport mappings to use different network interfaces (when available), transformation of flows, and variety of flow to transport mappings, etc.

Adding Value to Communication: Consider the example of a browser accessing a web page typically creates multiple connections to a server to obtain content for constructing the page. In this *client-server* model, once a session has been established, the transport connection-setup cost for subsequent connections can be avoided because transport parameters between the same two hosts can be used as such or derived rather than recreated — e.g., the three-way handshakes become redundant for subsequent connections because the session has already incurred the cost of bootstrapping and thus data can be sent along with the first TCP segment of subsequent connections. This is especially beneficial when flows within a session share authentication and encryption parameters.

Enabling Richer Functionality: SLIM benefits applications using the *publish-subscribe* model by supporting sessions involving two or more participants with dynamic reconfiguration. For example, clients can subscribe and unsubscribe to a video streaming service yet be treated as a whole by the server through the session abstraction. While responsibility for changes in video quality in response to varying network conditions belongs higher in the stack (in the "presentation layer"), feedback from the flows to the application is required for the quality to be properly adapted.

Another potential benefit, which we do not support yet, is for *peer-to-peer* communication where processes are not confined to specific roles — e.g., bittorent protocol simultaneously creates connections to multiple peers, where distributed hash table records, available files, and content segments can be shared over the independent flows in the same session. In such a case, flows may be structured between endpoints in a manner suitable to the algorithms (e.g., in a Chord-like fashion [36]). This simplifies the design of protocols between peers.

Table 1: Comparison of SLIM's contributions (✓does/can support, ✗: does not support, ✗ / ✓: subjective).

		SIP [29]	TESLA [31]	SCTP [34]	SST [11]	MPTCP [27]	SERVAL [23]	III [35]	SLIM
Separation of semantics & enabling innovation	Session abstractions	✗ / ✓	✗ / ✓	✗	✗	✗	✗	✗	✓
	Flow migration	✗	✗	✗ / ✓	✗	✓	✗ / ✓	✗	✓
	Endpoint migration	✗ / ✓	✗	✗	✗	✗	✗	✓	✓
	Extensibility	✗	✗ / ✓	✗	✗	✗	✗	✗	✓
Backward compatibility	Like TCP on the wire	✓	✓	✗	✗	✓	✗	✗	✓
	Compat. with legacy applications	✗	✗	✗	✗ / ✓	✓	✗	✗	✓
	Compat. with TCP stacks	✓	✗	✗	✗	✓	✗	✗	✓
Enabling greater functionality	Transport independence	✓	✗	✗	✓	✗	✗	✓	✓
	Fault tolerance	✗	✓	✗	✓	✓	✓	✗ / ✓	✓
	Dynamic reconfiguration	✓	✗	✗	✗	✗	✗	✗ / ✓	✓
	Multihoming	✗	✗	✓	✗	✓	✓	✗	✓

6.3 Enabling Innovation and Extensibility

Through the control flow, SLIM presents a large control-signaling space for future extensions. This is in sharp contrast to the shrinking option space available through TCP options — i.e., the space in the TCP SYN message [13, 15]. Having a signaling space to exchange control information between stacks is necessary to enable future extensions. Developers can use the control space to integrate extensions that operate within session semantics by defining verbs and registering corresponding handlers with the SLIM framework, for applications and services to use.

For example, clients connecting to a secure web server may connect through a SSL accelerator which terminates the secure connection and proxies the now unencrypted traffic to the servers that make up the service. Because of the need for all traffic to pass through the accelerator, it has the potential to become the bottleneck. An alternative approach enabled by SLIM is to set up a session between the client and all the servers implementing the service, including the SSL accelerator. Once authenticated, the session is secure and the participants can switch to using less computationally expensive symmetric ciphers allowing flows to go directly between endpoints without going through the accelerator. Alternatively, the accelerator can migrate the flow to the servers. Here all the control signaling necessary to manage the session activities can be implemented through SLIM's control flow.

6.4 Backward Compatibility and Adoption

As we discuss in § 2 and § 5, backward compatibility is critical for adoption of the proposals by the wider community. With SLIM, we successfully achieve backward compatibility with both legacy applications and the network stacks. We implement a shim library along with SLIM, which allows us to intercept legacy socket API calls from the applications using `LD_PRELOAD` and then pass it on to the SLIM API. This is when the intent is to indirectly use the SLIM API. If the intent of the application developer is not to use the SLIM API, the application may be programmed traditionally to use the Socket API and without `LD_PRELOAD` SLIM will not interfere with the communications. Similarly, SLIM uses the TCP custom options as part of the TCP SYN message to determine if the peer stacks support SLIM. If they do then enriched communications are setup as planned. However, if peer stacks do not support SLIM, the implementation gracefully falls back to legacy TCP. Details of the use of custom options is discussed in the technical report [17] and our prior work [15].

By enabling backward compatibility and allowing applications as well as network stacks to gracefully fall back to legacy TCP, SLIM supports incremental adoption by not forcing everyone to opt into using SLIM to setup communications.

6.5 In the Presence of Middleboxes

Research indicates that traffic with custom transport headers is either dropped altogether or that the options are stripped off when packets pass through middleboxes [13, 15, 16, 20, 36]. This poses a significant challenge towards adoption when our network infrastructures today host many middleboxes serving useful purposes. Thus the proposal for evolution of communications not only has to be backward compatible with legacy applications and network stacks, but also with network elements that form part of the infrastructure. Also, in some cases, the middleboxes modify the traffic flowing through them to provide their services. For example, NATs change the IP addresses of outgoing traffic and therefore the recipients cannot assume that the source IP address is that of the original client (or that of a NAT box). Therefore, careful considerations need to be made about assumptions.

With SLIM, because we decouple session semantics from that of transport, communications are independent of changes in underlying configuration or modifications to in-flight traffic. The only scenarios where existence of a middlebox could possibly interfere with the SLIM session layer are during: 1) creation of a data flow or 2) creation of the control flow. Setting up a data flow maps to a TCP connection via the `add_flow` primitive so middleboxes will react the same whether or not the TCP connection is initiated directly or via SLIM. Since control flows are also mapped to TCP connections, middlebox are not be able to tell the difference either. While NATs, accelerators, and load balancers do not interfere with SLIM, firewalls may cause disruptions since SLIM uses custom TCP options along with the TCP SYN message to setup communications. Surveys [13] show that firewalls typically let custom options through if they are associated with the TCP SYN message (and not otherwise), which we have also confirmed with testing [16]. Nevertheless, if the control flow setup fails, SLIM gracefully falls back to legacy behavior. The key enabler is that SLIM and legacy traffic behavior are identical on the wire.

6.6 Development Effort — Cost vs. Value

With SLIM managing the conversation (session management necessary to support dynamic reconfiguration, communication involving

two or more participants, mobility, session abstractions to transport mappings etc.) developers are free to focus on their application rather than implementing the underlying plumbing to enable application features.

Using the additional functionality that SLIM enables does require some modifications to applications. However, a subset of the SLIM session API is designed to mimic traditional socket semantics, allowing existing code to run on top of a SLIM stack without change by way of shared library interposition. We demonstrate this work by developing a shim library that uses `LD_PRELOAD` to intercept Socket API calls and implement a wrapper for SLIM. While the application will not gain all the benefits, e.g., communications involving more than two participants or mobility, it will automatically gain the increased resilience that comes from being able to restart flows in a session.

Automated Code Refactoring We have implemented a prototype source-to-source translator that takes code using the Socket API as input and translates it into code that uses SLIM API. We have successfully tested the prototype for correctness on simple client and server applications. While this is work-in-progress, we've concluded that such transformations are possible and would facilitate evolution towards SLIM. Note that a simple refactoring enables support for fault-tolerant communication [15, 16].

6.7 User Space vs. Kernel

The transport layer is implemented within the kernel for various reasons including cross-layer communication and performance. Our prototype is primarily in the user space, with the modifications for custom TCP options as the only portion implemented as part of the kernel. We envision a production version to be implemented in the kernel and its functionality exposed with an interface to applications. Functionality which does not lie on the critical path may continue to be implemented in the user space. However, we have not explored other kernel-implementation concerns yet.

6.8 Security Considerations

As explained in § 3, we intend to lay a foundation that enables well-established information-security methods in assisting session management. We argue that if efficient methods of access control (i.e., identification, authentication, authorization), confidentiality, integrity, or other information security concerns are to be realized, then incorporating such concerns in the design is important (§ 3). On the other hand, opaque identifiers may be used for endpoints, flows, and sessions to develop simpler session management solutions. While doing so would not preclude higher-layer solutions of information security, they would not benefit from the gains of an integrated session-layer. Even when simpler/opaque identifiers are used, SLIM extensions do not expose network communications to additional security threats when compared to legacy TCP.

While the use of public keys as basis for endpoint labels might suggest that anonymity may become a challenge, this is not the case. In fact significant efforts are already underway to address such aspects [24]. Also, since the public-key infrastructure is already well established and deployed worldwide, its use doesn't require deployment of additional resources in the network.

6.9 Performance Evaluation

The overarching question that we try to answer here is: *does the use of session abstractions incur a negative impact when compared to the use of the Socket API*? We find that there is no statistically significant difference between the use of SLIM and the Socket API. The reasons behind this are two-fold: 1) SLIM is engaged in configuration of communications during setup, reconfiguration and tear down phases and is not actively involved during data transfer; 2) The flow abstractions, which are primarily used during data transfer, are implemented using the Socket API to manage transport semantics in the prototype.

We set up the environment using `dummynet` [7]. `Dummynet` is configured to obtain precise measurements (e.g., set the kernel frequency timer to 4000 *Hz* since `Dummynet`'s emulation is coarse grained and bursty in nature for microsecond-level precision, which becomes apparent at low latencies). We define maximum window sizes and buffer sizes to ensure that they do not gate throughput tests. We also enable window scaling and selective acknowledgments. We define link capacities of 1 *Gbps* between nodes using `dummynet` pipes. We vary round trip times (RTTs) and packet loss rates based on typical values [10] — < 1 *ms*, 5 *ms*, 10 *ms*, 25 *ms*, 50 *ms* and 100 *ms* for RTTs and 0%, 1%, and 10% for packet loss rates. Here we show select results due to limited space.

Throughput: After a session is setup, the flow abstractions are mapped on to the underlying transport. Therefore, we would expect both Socket API and SLIM to achieve similar throughputs. Our measurements confirm that results for different RTTs are statistically the same for all variations of configuration that we tested. Figure 11 shows the results for one configuration where the endpoints are able to saturate the link to the achievable throughputs of about 94% of link capacity. With increasing RTTs, achievable throughputs are met after relatively longer running tests. Since both SLIM and the Socket API use TCP as the transport protocol we verified achieved throughputs with Mathis' estimates [19]. With packet loss, we see that throughputs for both the Socket implementation and SLIM are adversely effected with increasing loss rate (see Figure 12).

Figure 13 summarizes the throughputs from the flow's perspective with a broadcast structure. Figure 14 on the other hand summarizes throughputs of underlying transport connections for the

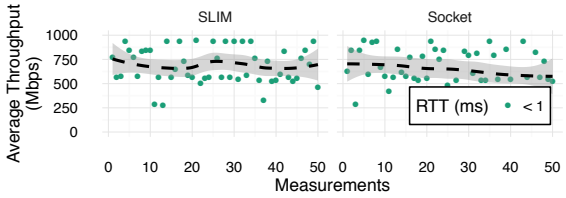

Figure 11: Average throughput, with 90% confidence interval, for Socket and SLIM (1 *Gbps* link capacity, 0% loss).

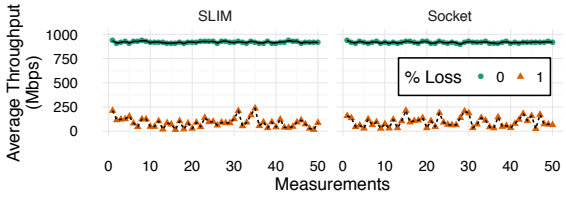

Figure 12: Average throughput for Sockets and SLIM (1 *Gbps* link, < 1 *ms* RTT 0% and 1% loss.

same flow. While it may seem that increasing number of participants reduces throughput, this is not the case. When there are three participants in a session, from the sender's perspective there are two underlying transport connections that implement the flow with a broadcast structure. Therefore, in this case when the flow observes a throughput of 468.5 *Mbps* to the endpoints, it is because the underlying transports observe a throughput of 449 and 488 *Mbps* to each process. These adds up to about 937 *Mbps*, which is close to the achievable peak throughput. Note that throughputs for underlying transports increase until link capacity is reached.

Setup Time: Time to initialize a session is expected to be nearly equal to the time to allocate memory for the instance. We measured this to be 5 μs, which is the same as initializing a socket. There is no statistically-significant difference between the time to create a session with a single data flow and a TCP connection, which is dictated by the RTT between the endpoints (shown in Table 2). Usually, sessions will include a control flow and at least one data flow. The setup time is not impacted by the creation of the control flow as its setup is managed independently in parallel. Once a session context has been established, subsequent data flows do *not* require three-way handshakes before sending data.

Table 2: Setup Time between Peers

	$RTT\ (ms)$	$\bar{x}\ (ms)$	$s\ (ms)$
TCP connection	< 1	0.16	0.06
	50	50.35	0.56
Session with no flows	$< 1, 50$	0.005	0.001
Session with a data and control flow	< 1	0.16	0.07
	50	50.52	0.70

Exchange of Verbs and Reconfiguration: When an asynchronous request is issued, the receiving endpoint returns an acknowledgment with a code. We measure the time it takes for the request to be issued and the return code to be received. A subset of results are presented in Table 3, where we issue consecutive requests to obtain a trace and also capture the variance. We see that the latency is dictated by the RTT between the endpoints. The variation measured in our traces is a characteristic of `dummynet` due to the coarseness of its implementation when inducing delays through pipes. For blocking reconfigurations, the the response time would depend on the type and context of the verb.

Table 3: Response Time of Non-Blocking Reconfigurations

RTT (ms)	$\bar{x}\ (ms)$	$s\ (ms)$
10	10.36	0.61
50	50.77	0.65

Memory Footprint: With support for higher layer abstractions, a session instance's memory footprint is larger than that of a socket abstraction, which essentially is a file descriptor. When two endpoints are involved in a session, the instance has a memory footprint of 132 *bytes*. The breakdown is shown in Table 4. The addition of every subsequent endpoint increases the footprint by 42 *bytes*, while the addition of a data flow increases the footprint by 18 *bytes*. The above profile is calculated with the assumption that endpoint hosting the session has a single interface. Every additional interface on the endpoint increases the footprint by 22 *bytes*. While the footprint of 132 *bytes* seems large in contrast to a 4 *byte* socket, note that the developer would have to implement similar bookkeeping as part of the application for similar use cases.

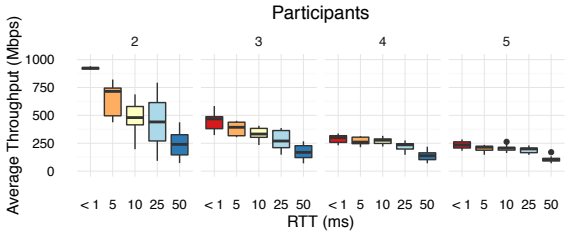

Figure 13: Flow's perspective with SLIM and increasing number of participants (1 *Gbps* link, varying RTTs, 0.01% loss)

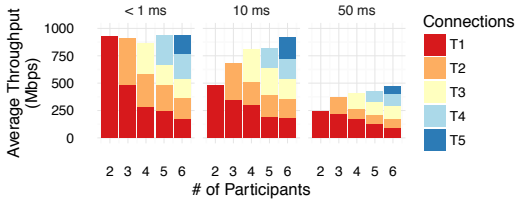

Figure 14: TCP's perspective with SLIM and increasing number of participants (1 *Gbps* link, varying RTTs, 0.01% loss)

Table 4: Session's Memory Footprint

	Bytes
Human-readable label	20
Opaque ID	24
One endpoint	20 (`label`) + 22 (`sockaddr_in`)
Control flow	10 (`label`) + 4 (ID) + 4 (`handler`)
One Data flow	10 (`label`) + 4 (ID) + 4 (`handler`)
Misc. bookkeeping	10
Total	132

CPU Overhead: We studied the CPU usage of applications conducting small (e.g., 10 *KB*), medium (e.g., 1 *MB*) and large volume (e.g., 10 *GB*) transfers. We did not observe any statistically significant difference between applications using the SLIM and the Socket API. This is in spite that the fact that SLIM implements the control flow and a context manager; the control flow is not over bearing since it only plays a role during communications management and is not involved in the data transfer phase.

7 RELATED WORK

There have been several notable attempts when it comes to innovation in and extensions of communications to enable modern use cases. However, the corpus of research is too vast to cover comprehensively within this paper. Nevertheless, we select few prominent proposals [11, 22, 23, 27, 29, 31, 34, 35, 40] and classify them into groups of session-layer proposals, modern transport-layer proposals, network-stack extensions, and clean-slate designs. Table 1 summarizes the contributions of SLIM and these proposals.

Session-Layer Proposals: TESLA [31] presents notable session layer services 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 perhaps because: 1) it assumes that all networks stacks implement TESLA as a session layer service; and 2) it focuses on extensions to transport services (e.g., encryption of flows) and only proposes the flow abstraction, which alone is not sufficient to describe conversations. This is in contrast with SLIM, which does not assume that all stacks

integrate SLIM services and that SLIM proposes the session, flow and endpoint abstractions to describe conversations.

The Session-Initiation Protocol (SIP) [29], is one of the well-known session-based communication protocols. As the name suggests, it assists with the setup and tear down of a session. It does not engage with data communication itself. It is primarily geared towards multimedia services and is the cornerstone of most VoIP services. It handles call management, while the voice and video data is transferred using RTP [32]. Although SIP has seen tremendous success with call services, it has not been adopted for use outside of multimedia. This may be because, unlike SLIM which proposes the session, flow and endpoint abstractions, SIP does not present constructs that describe the communication, thus limiting support to managing conversations. Unlike SLIM, which addresses the limiting assumptions of existing stacks (e.g., conflation of session and transport semantics) SIP builds on top of existing stacks. Thus, services built using SIP are not only constrained by the limitations of the existing stacks, but will also incur additional overheads for working around these limitations. Also unlike SLIM which provides seamless support for mobility and resilience, SIP is dependent on the application developer to monitor the session state and assist with reconfiguring or resetting communication. Although SIP has not seen adoption outside of multimedia (perhaps due to the challenges listed above), it still presents useful insights into session management. SLIM leverages the lessons learned by SIP and proposes a generic framework that extends the network stack by providing session-layer services to the application.

The OSI model [40] includes a session layer in its design. However, it too was not widely adopter, perhaps due to its slow development and also because TCP/IP, at the time, was proving to be an effective alternate [30]. Today, the modern use cases demand much more than what the reference model was designed for and therefore it would not meet the needs of contemporary communications.

Modern Transport-Layer Proposals: Structured Streams [11] proposes a transport abstraction as an alternative to TCP. The intent is to create child transport streams from existing transport connections while incurring minimal cost. It allows the application to have parallel streams. The proposed stream abstraction has not been widely adopted perhaps because: 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 exposes a different API to the application, which implies that the implementation is not backwards compatible with legacy applications; and 3) unlike SLIM, it doesn't support modern use cases (e.g., multi-homing) or richer abstractions.

Multipath TCP [39] provides extensions to TCP to support multi-homing, fault tolerance and flow migration. As proposed extensions are based around TCP, they are backwards compatible but applications must be modified if extended functionality is to be used; using the legacy API does not translate into access to all features. Multipath TCP focuses on extending transport services and unlike SLIM does not propose richer communication abstractions or encourage decoupling of session and transport semantics. Nevertheless, among all proposals, Multipath TCP is the only proposal that has seen adoption (by Apple [2], perhaps because the company has complete control over its software and can have a flag day for deployment). We compare and contrast SLIM with SCTP [34] in our technical report [17].

Network-Layer Extensions: SERVAL, an end-host stack for service-centric networking [23], proposes a service-access layer between the TCP and IP layers to: 1) enable endpoints to use multiple network addresses, 2) enable flows to migrate across interfaces, and 3) 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 session abstraction.

The focus of the Host Identity Protocol (HIP) [22] is to address an issue of mobile networking. The intention is to allow hosts to maintain shared IP-layer state, which removes the coupling between the locater and identified roles of the network address. This allows the network application to continue communication in spite of change of network (i.e., IP) address. HIP does so while taking into account several information security considerations. While the proposal addresses an important problem for modern use cases, the solution addresses one aspect of a larger problem, which is to enable greater functionality for modern communications.

Clean-Slate Designs: Stoica et al., urge that we reconsider the way we approach networking; they propose the use of rendezvous-based communication, Internet Indirection Infrastructure (i3) [35], that decouples the act of sending from the act of receiving. Such a model decouples the notion of communication labels from the location of the end hosts and in doing so mitigates concerns of mobility. This model also describes different communication paradigms (e.g., multicast, broadcast, anycast). While these are valuable contributions, we see that any proposal that is not backwards compatible has not gained traction with the wider community — perhaps because of the momentum behind TCP.

8 SUMMARY AND FUTURE WORK

In this paper, we propose an extensible session-layer intermediary, called SLIM, built upon explicit session, flow, and endpoint abstractions. The separation of session and transport concerns leads to clearer descriptions of communication models and enables advance network functionality: mobility, communications involving two or more participants, and dynamic reconfiguration. Even more important, SLIM's control flow provides a mechanism for revitalizing network stack innovation by providing a space for control signaling. A prototype SLIM implementation has been created to explore various design alternatives.

In the future, we plan to explore the benefits of more internal cross-layer communication within the stack to allow for better coordination. We also wish to further expand the set of control flow verbs with a view towards creating mechanisms upon which applications can build policies and functionality. We also plan to explore which portions of SLIM should be in kernel space and which should be in user space, leading to a more production-ready implementation. Finally, we will modify additional applications to utilize the advanced functionality of SLIM and report an empirical performance evaluation of its use.

REFERENCES

[1] 2015. iOS and OSX Handoff - Apple Continuity. https://developer.apple.com/handoff. (2015). Accessed December 16, 2016.

[2] 2015. iOS: Multipath TCP Support in iOS 7. http://support.apple.com/en-us/HT201373. (2015).

[3] 2015. New Features in Android OS 5.0 - Multi Networking. http://android-developers.blogspot.com/2014/10/whats-new-in-android-50-lollipop.html. (2015). Accessed December 16, 2016.

[4] 2015. POSIX.1-2008 Specification. http://pubs.opengroup.org/onlinepubs/9699919799/functions/contents.html. (2015). Accessed December 16, 2016.

[5] David G. Andersen, Hari Balakrishnan, Nick Feamster, Teemu Koponen, Daekyeong Moon, and Scott Shenker. 2008. Accountable Internet Protocol (AIP). In *ACM SIGCOMM*.

[6] Eric Brown, Mark Gardner, Umar Kalim, and Wu Feng. 2011. Restoring End-to-End Resilience in the Presence of Middleboxes. In *IEEE International Conference on Computer Communication and Networks (ICCCN)*.

[7] Marta Carbone and Luigi Rizzo. 2010. Dummynet Revisited. http://doi.acm.org/10.1145/1764873.1764876, *ACM SIGCOMM Computer Communication Review* (2010).

[8] Vinton G. Cerf and Munindar P. Singh. 2010. Internet Predictions: Future Imperfect. http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5370817, *IEEE Internet Computing* (2010).

[9] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. 2008. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 5280 (Proposed Standard). (2008). Updated by RFC 6818.

[10] Les Cottrell. 2016. Ping End-to-End Reporting. http://www-iepm.slac.stanford.edu/pinger/. (2016). Accessed December 16, 2016.

[11] Bryan Ford. 2007. Structured Streams: A New Transport Abstraction. In *ACM SIGCOMM Computer Communication Review*.

[12] Dongsu Han, Ashok Anand, Fahad Dogar, Boyan Li, Hyeontaek Lim, Michel Machado, Arvind Mukundan, Wenfei Wu, Aditya Akella, David G. Andersen, John W. Byers, Srinivasan Seshan, and Peter Steenkiste. 2012. XIA: Efficient Support for Evolvable Internetworking. In *USENIX NSDI*.

[13] Michio Honda, Yoshifumi Nishida, Costin Raiciu, Adam Greenhalgh, Mark Handley, and Hideyuki Tokuda. 2011. Is it Still Possible to Extend TCP?. In *ACM Internet Measurement Conference*.

[14] Umar Kalim. 2011. Demonstration Video of Seamless Virtual Machine Migration. http://www.cs.vt.edu/ umar/vm-demo. (2011).

[15] Umar Kalim, Eric Brown, Mark Gardner, and Wu Feng. 2010. Enabling Renewed Innovation in TCP by Establishing an Isolation Boundary. http://pfld.net/2010/technical.php. In *8th International Workshop on Protocols for Future, Large-Scale and Diverse Network Transports (PFLDNeT)*.

[16] Umar Kalim, Mark Gardner, Eric Brown, and et al. 2013. Seamless Migration of Virtual Machines Across Networks. In *IEEE International Conference on Computer Communication and Networks (ICCCN)*.

[17] Umar Kalim, Mark Gardner, Eric Brown, and Wu Feng. 2015. *SLIM: A Session-Layer Intermediary for Enabling Multi-Party and Reconfigurable Communication*. Technical Report TR-15-04. Department of Computer Science, Virginia Tech.

[18] Craig Labovitz. 2008. Internet Traffic Trends. NANOG 43. (2008). North American Network Operators' Group.

[19] Matthew Mathis, Jeffrey Semke, Jamshid Mahdavi, and Teunis Ott. 1997. The Macroscopic Behavior of the TCP Congestion Avoidance Algorithm. *ACM SIGCOMM Computer Communication Review* (1997).

[20] Alberto Medina, Mark Allman, and Sally Floyd. 2005. Measuring the evolution of transport protocols in the Internet. http://portal.acm.org/citation.cfm?doid=1064413.1064418, *ACM SIGCOMM Computer Communication Review* (2005).

[21] P.V. Mockapetris. 1987. Domain names - implementation and specification. RFC 1035 (STANDARD). (1987). Updated by RFCs 1101, 1183, 1348, 1876, 1982, 1995, 1996, 2065, 2136, 2181, 2137, 2308, 2535, 2845, 3425, 3658, 4033, 4034, 4035, 4343, 5936, 5966, 6604.

[22] R. Moskowitz, T. Heer, P. Jokela, and T. Henderson. 2015. Host Identity Protocol Version 2 (HIPv2). RFC 7401 (Proposed Standard). (2015).

[23] Erik Nordström, David Shue, Prem Gopalan, Robert Kiefer, Matvey Arye, Steven Y. Ko, Jennifer Rexford, and Michael J. Freedman. 2012. Serval: An End-host Stack for Service-centric Networking. In *USENIX NSDI*.

[24] S. Park, H. Park, Y. Won, J. Lee, and S. Kent. 2009. Traceable Anonymous Certificate. RFC 5636 (Experimental). (2009).

[25] Lucian Popa, Patrick Wendell, Ali Ghodsi, and Ion Stoica. 2012. *HTTP: An Evolvable Narrow Waist for the Future Internet*. Technical Report UCB/EECS-2012-5. University of California Berkeley.

[26] J. Postel. 1981. Transmission Control Protocol. RFC 793 (INTERNET STANDARD). (1981). Updated by RFCs 1122, 3168, 6093, 6528.

[27] Costin Raiciu, Christoph Paasch, Sebastien Barre, Alan Ford, Michio Honda, Fabien Duchene, Olivier Bonaventure, and Mark Handley. 2012. How Hard Can It Be? Designing and Implementing a Deployable Multipath TCP. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*.

[28] Jennifer Rexford and Constantine Dovrolis. 2010. Future Internet Architecture. http://portal.acm.org/citation.cfm?doid=1810891.1810906, *Communications of the ACM* (2010).

[29] J. Rosenberg, H. Schulzrinne, G. Camarillo, and et al. 2002. SIP: Session Initiation Protocol. RFC 3261 (Proposed Standard). (2002). Updated by RFCs 3265, 3853, 4320, 4916, 5393, 5621, 5626, 5630, 5922, 5954, 6026, 6141, 6665.

[30] Andrew L. Russell. 2013. OSI: The Internet that Wasn't — How TCP/IP eclipsed the Open Systems Interconnection standards to become the global protocol for computer networking. http://spectrum.ieee.org/computing/networks/osi-the-internet-that-wasnt, (July 2013).

[31] Jon Salz, Alex C. Snoeren, and Hari Balakrishnan. 2003. TESLA: A Transparent, Extensible Session-Layer Architecture for End-to-end Network Services. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.13.5061. In *USITS*. USENIX.

[32] H. Schulzrinne, S. Casner, R. Frederick, and et al. 2003. RTP: A Transport Protocol for Real-Time Applications. RFC 3550 (INTERNET STANDARD). (2003). Updated by RFCs 5506, 5761, 6051, 6222.

[33] J. Sermersheim. 2006. Lightweight Directory Access Protocol (LDAP): The Protocol. RFC 4511 (Proposed Standard). (2006).

[34] R. Stewart. 2007. Stream Control Transmission Protocol. RFC 4960 (Proposed Standard). (2007).

[35] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana. 2004. Internet Indirection Infrastructure. *IEEE/ACM Transactions on Networking* (2004).

[36] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. 2001. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *ACM SIGCOMM*.

[37] Andrew S. Tanenbaum and Maarten Van Steen. 2006. *Distributed Systems: Principles and Paradigms*. Prentice Hall.

[38] P. Vixie, S. Thomson, Y. Rekhter, and J. Bound. 1997. Dynamic Updates in the Domain Name System (DNS UPDATE). RFC 2136 (Proposed Standard). (1997). Updated by RFCs 3007, 4035, 4033, 4034.

[39] Damon Wischik, Costin Raiciu, Adam Greenhalgh, and Mark Handley. 2011. Design, Implementation and Evaluation of Congestion Control for Multipath TCP. In *8th USENIX NSDI*.

[40] Hubert Zimmermann. 1980. OSI Reference Model — The ISO Model of Architecture for Open Systems Interconnection. http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=1094702. In *IEEE Transactions on Communications*. IEEE, 425–432.