Transparent TCP-to-SCTP Translation Shim Layer

Ryan W. Bickhart*
Internet Technologies Division
Cisco Systems, Inc.
rbickhar@cisco.com

Paul D. Amer
Protocol Engineering Lab
University of Delaware
amer@cis.udel.edu

Randall R. Stewart
Internet Technologies Division
Cisco Systems, Inc.
rrs@cisco.com

***Abstract*— To encourage SCTP adoption and facilitate incremental deployment of the protocol, we develop a shim layer which transparently translates application-level system calls to TCP into corresponding calls to SCTP, allowing legacy TCP applications to communicate using SCTP for end-to-end transport *without any modifications to the applications themselves*. Using our shim layer implementation in the FreeBSD 4.10 operating system kernel, we experimentally demonstrate the technical feasibility of this transparent TCP-to-SCTP translation scheme for several popular network applications including HTTP, SSH, Telnet, and streaming audio. Additionally, we show application performance in terms of responsiveness and throughput using the shim and SCTP is equivalent to or better than performance when applications operate using TCP as originally designed.**

I. Introduction

This paper introduces a *Transparent TCP-to-SCTP Translation Shim Layer*. The cornerstone of this concept is *translating* application-layer system calls to TCP into equivalent calls to Stream Control Transmission Protocol (SCTP). This translation process occurs *transparently*, meaning the application is unaware its calls to TCP are being mapped to SCTP instead. Lastly, this functionality is implemented as a *shim layer*, meaning the logic to accomplish this protocol translation is inserted into the socket layer between the application and transport layers, leaving the structure of the existing network protocol stack intact. The shim is designed to be backwards compatible, automatically reverting to a normal TCP connection for communications in situations where an SCTP association between the two endpoints or services cannot be established.

A. SCTP & Multihoming Preliminaries

SCTP is a connection-oriented, reliable, messaged-based, general purpose transport protocol with congestion control similar to that used by TCP, supporting advanced features unavailable in TCP or UDP [18]. SCTP was originally developed to carry telephony signaling information over IP networks because neither TCP nor UDP could meet specific reliability requirements mandated for telephone carriers by government regulations. However, over time SCTP's features have been recognized to be generally useful in more than just the limited scope of the telephony signaling world. Consequently, SCTP morphed into an IETF standards-track, general purpose transport protocol [17]. Arguably one of the most important and distinctive features of SCTP is integrated support for *transport layer multihoming*.

A host which has more than one interface is said to be *multihomed* [2]. Historically, hosts on the Internet have been single-homed due to the relatively high expense of network interfaces, compared to any serious need to be connected to more than a single network at a time. Multihoming is rapidly becoming commonplace on today's Internet as interfaces have become inexpensive commodity items and multiple competing options often exist for Internet connectivity.

Despite the fact that today multihoming is frequently economical and practical, support for multihoming is lacking in TCP and UDP. These protocols are limited by the fact that they originated in an era where multihoming was never considered as a design issue. For example, a TCP connection is defined as a four-tuple of two addresses and two port numbers, so even when endpoints have multiple addresses, TCP may use only one address at each endpoint per connection.

SCTP natively supports multihoming at the transport layer. Consequently, an SCTP *association* between two hosts consists of the *entire set of addresses* available at each endpoint. SCTP can use *any* feasible (i.e., not restricted by routing configurations, firewalls, etc.) combination of these available addresses for communication during the lifetime of a single association, unlike a TCP connection which can only select a single pair of addresses.

B. Motivations for TCP-to-SCTP Translation

The integrated support for multihoming in SCTP is the basis of two important motivations for the TCP-to-SCTP translation shim layer. The first motivation is the ability to provide *fault tolerance* to legacy applications by using

*This research results from the first author's Master's Thesis while with the Protocol Engineering Lab at the University of Delaware.

Prepared through collaborative participation in the Communications and Networks Consortium sponsored by the U.S. Army Research Laboratory under the Collaborative Technology Alliance Program, Cooperative Agreement DAAD19-01-2-0011. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon.

Supported in part by the University Research Program of Cisco Systems, Inc.

SCTP's multihoming support. SCTP defines the concept of a *primary destination address*. New data is actively sent to this address while any remaining destination addresses of a multihomed endpoint, termed *alternate destinations*, and are held in reserve for retransmission of data in the case of loss or path failure. By using the shim, TCP applications running on multihomed hosts will be able to exploit the fault tolerant communications ability that is inherently present in SCTP. This fault tolerance is essentially available to legacy TCP applications using the shim "for free," as fault tolerance is a default ability of SCTP.

A second motivation is the possibility of taking further advantage of SCTP's multihoming capabilities to enable *concurrent multipath transfer* (CMT), an area of ongoing research which involves using multiple network paths for concurrent transfer of new data [7], [8]. The current SCTP standard specifies that new data can only be sent to a peer's primary destination; any alternate destinations are used only for retransmissions for the purposes of fault tolerance [18]. Extending SCTP to allow new data to be sent to multiple peer destinations simultaneously has the potential to allow for higher association throughput if the bandwidth to do so is available in the network. Using the shim layer, legacy TCP applications with multiple addresses will be able to take advantage of the fault tolerance provided by SCTP multihoming, and potentially the increased throughput provided by CMT.

Even in situations where endpoints using the TCP-to-SCTP translation shim are not multihomed and cannot make use of the fault tolerance and CMT features available with SCTP multihoming, the shim still serves an important purpose by facilitating SCTP's *incremental deployment*. Although SCTP provides basic TCP-like services in addition to advanced features such as increased fault tolerance with multihoming, SCTP is not directly interoperable with TCP (i.e., an SCTP endpoint cannot connect to a TCP endpoint). Being relatively new, SCTP has not yet been widely deployed in the Internet and is limited by the "chicken and egg" problem of incremental deployment. Little motivation exists among application developers to design applications that take advantage of SCTP's advanced features at the transport layer because few operating systems support SCTP. Likewise, users do not demand SCTP support for their applications because few related applications use SCTP. The shim solves this incremental deployment problem for SCTP and TCP by enabling interoperation between legacy TCP endpoints using the shim, and endpoints that natively support SCTP.

This legacy-native architecture of the shim is illustrated in Figure 1. Enabling legacy TCP applications to interact with their native SCTP peers provides an incentive to begin using SCTP in new projects because developers can be sure that the existing deployed base of legacy TCP applications will be compatible with new, native SCTP applications through the shim translation layer.

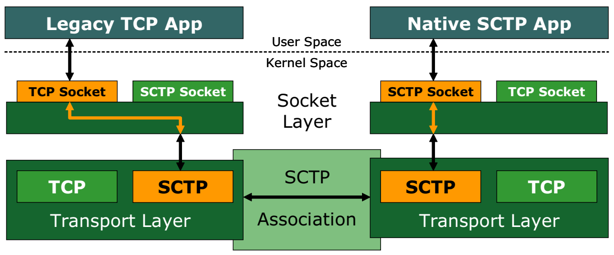

Fig. 1. Shim legacy-native architecture

The remainder of this paper is organized as follows: Sections II-VIII focus on the design and implementation of the shim layer, including the general approach taken, the functionality of the major components, and the implementation details. Sections IX and X describe the experimental evaluation performed on the shim and the results of both proof-of-concept and performance testing. Finally, Section XI concludes with final remarks about our TCP-to-SCTP shim implementation.

II. Design Overview

Currently, the most mature and stable kernel implementation of SCTP is found in the BSD family of operating systems, and distributed as part of the KAME project [9]. The KAME group is a consortium of companies primarily concerned with developing a fully-functional IPv6 and IPSec protocol stack. The KAME distributions also include support for other up-and-coming or experimental network protocols such as SCTP and DCCP [10]. Since a functional TCP-to-SCTP translation shim is absolutely dependent on a stable kernel implementation of SCTP, we selected FreeBSD as the operating system for our shim implementation. Consequently, the *specific* design details presented in this paper may not be exactly applicable to all operating systems, though we believe the *general* concepts should apply to any operating environment supporting the standard sockets API.

At the time shim implementation began, the stable KAME distribution was for FreeBSD 4.10. Although KAME (and thus SCTP) has since transitioned to the FreeBSD 5 series, we completed the shim in the 4.10 kernel in the interest of having a fully functional implementation on a stable release before attempting to port the shim to later versions of FreeBSD or other operating systems entirely. (Shim migration to FreeBSD 5 is currently in planning).

One of the first major design decisions was whether to implement the shim layer entirely within the kernel or as a library in user space. There are pros and cons to each approach, which we now discuss.

The primary advantage of a library implementation is not requiring users to modify their operating system kernel to make use of the shim functionality. Modifying the kernel can be difficult for nontechnical users, as well as potentially problematic if the kernel is improperly configured and compiled to operate with the underlying hardware. The shim implemented as a user library using the standard sockets API would be more portable between different operating systems or operating system versions than a kernel-based shim, since socket APIs are more uniform across systems than kernel implementations. Control over which applications use the shim is also simplified since only applications specifically recompiled with the shim library would actually use the shim.

However, the safety and simplicity a user-space library implementation of the shim comes with many disadvantages. Most notably, a user library implementation requires the user to recompile or relink each and every application for which they wish to enable the shim. This effort will be a nuisance when a user has a large number of applications to be shim-enabled. In other situations, recompiling all the necessary applications will be impossible; consider the case where a user has some proprietary applications which are distributed only in binary form for which no source code or object files are available to allow rebuilding.

The benefits and drawbacks of implementing the shim in the operating system kernel are essentially the inverses of implementing the shim as a user-space library. While arguably more complicated than implementation as a user library, implementing the shim layer directly in the kernel also comes with some important advantages. The main advantage is that all applications on the system can make use of the shim's functionality without requiring rebuilding or any other modifications whatsoever. Additionally, because the shim is inside the kernel, the designer has more flexibility and discretion about exactly how the shim will work than is possible with a user-space implementation. The penalties of a kernel implementation include a less portable design (a separate shim implementation is needed for each operating system) and the requirement for control mechanisms to decide which applications should use the shim (use cannot be controlled by linking or not linking with a shim library user-space library).

Because one of our core goals is the *transparent translation* from TCP to SCTP without *any* modifications to legacy applications, a kernel implementation of the shim layer was the logical (and only) choice. Moreover, despite the danger of less portability between operating systems, we felt the design advantages of having full-scale kernel control over the shim would enable a more robust, feature-rich, and production-quality implementation than is possible with a user library.

A. Sockets Model

The sockets API allows applications to interact with the network in a uniform, protocol-independent and system-independent way. A *socket* is a data structure that encapsulates all of the state information required for a communications endpoint. While the actual number of fields contained inside a socket in a true implementation is large, the contents of the socket data structure can be generally summarized as state and configuration information, and I/O buffers. One additional field of primary importance to our shim work is the *protocol field*. The protocol field is the single aspect that distinguishes what would otherwise be a generic socket as specifically a TCP, UDP, or SCTP socket.

The operating system kernel maintains a set of data structures that completely define each supported protocol. These data structures, called the *protocol switch structures*, contain fields that specify the *domain* (address family) and *type* (communication semantics) of the protocol, the assigned protocol number, a series of flags describing protocol characteristics, and lists of the interfaces/entry points into the protocol. The protocol field of each socket points to the appropriate protocol switch structure. This link to a specific protocol definition in the kernel specifies the exact functionality for what would otherwise be a generic socket data structure. In the case of a TCP socket, the link to the TCP protocol switch structure makes the socket specifically a TCP socket, and not a socket bound to some other protocol.

The binding between a generic socket object and a specific protocol switch structure occurs when a socket is created. An application creates a new socket using the `socket()` system call, as in this example:

```
sockdesc = socket(AF_INET, SOCK_STREAM,
                  IPPROTO_TCP);
```

When `socket()` is called, the kernel uses the specified domain (*AF_INET*), type (*SOCK_STREAM*), and protocol (*IPPROTO_TCP*) parameters to find the appropriate switch structure for the requested protocol. Then, the kernel calls the `attach()` function found in that protocol's interface table. The `attach()` call implements the binding between the socket and the specific protocol, notifies the protocol that it must support a new socket, and reserves any resources necessary to accomplish that task. The counterpart to a protocol's `attach()` function is called `detach()`. As expected, `detach()` deallocates any resources previously allocated by `attach()` when the socket was created, and removes the binding between the socket data structure and the protocol.

The most obvious approach to support a TCP-to-SCTP translation shim is to take an existing TCP socket, detach TCP from the socket, and then attach SCTP instead, effectively transforming a TCP socket to an SCTP socket. Unfortunately, this approach is not possible given the specific operation of the `detach()` function. In addition to deallocating resources and removing the binding between a socket and protocol, `detach()` goes one step further, actually deallocating the entire socket data structure. The underlying assumption is that `attach()` and `detach()` should be called exactly once each during the lifetime of a socket, since changing the protocol a socket during active use is not an expected behavior.

B. *Hidden SCTP Socket*

Due to the behavior of `attach()` and `detach()`, as well as the need to revert to TCP when shim connections using SCTP are not possible, the TCP-to-SCTP translation shim requires two separate socket data structures, one bound to TCP and the other bound to SCTP. On a system incorporating the shim, we modify the traditional `socket()` system call so in addition to creating the normal TCP socket as usual, a second *hidden* SCTP socket is created. It is termed *hidden* because it is created by the kernel but not exposed to the application. The existence of a traditional socket is known to the application because a socket descriptor is returned and used by the application to access networking functionality. The shim's SCTP socket is created but remains hidden from the application in accordance with the goal of *transparent* translation from TCP to SCTP.

Because the hidden SCTP socket is inaccessible via a normal socket descriptor, the kernel needs to keep track of each pair of normal TCP and hidden SCTP sockets that are created. This tracking is accomplished by adding a new field to the system's standard socket data structure, which allows the hidden SCTP socket to be linked from its corresponding parent TCP socket. Figure 2 illustrates the relationship between a normal TCP socket and its hidden SCTP socket, as well as the corresponding protocol switch structures. The new *shim state* and *shim parent* fields visible in the figure are discussed in Sections IV and VI-C, respectively.

C. *Socket Layer in Detail*

Networking specialists define a five-layer TCP/IP Internet model consisting of the application, transport, network, link, and physical layers. However, from an implementation point of view, the five-layer model neglects the important socket layer. The focus of the shim implementation is the socket layer, which acts as an intermediary between an application and the transport protocols. The socket layer itself is organized as several sublayers as shown in Figure 3.

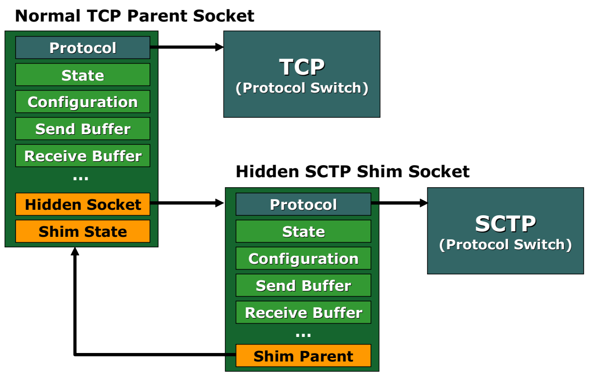

Fig. 2. Normal TCP socket with hidden SCTP socket

The top sublayer of the socket layer is made up of the *socket system call stubs*. These functions form the API that applications use to make requests on the lower layers. The stub functions do not perform any actual networking actions themselves; they only package application layer arguments into a format expected by the kernel, and then make the system call to enter into kernel execution mode. The socket system call stubs are typically included in a library that is linked with any application wishing to make use of networking services.

The sublayer immediately below the socket system call stub functions is the layer where the socket *system call implementations* lie. These are the functions inside the kernel which actually implement the functionality of the sockets API.

The system call implementations also make extensive use of a set of lower level functions called the *socket layer functions*. Each of these functions performs a specific networking task on a specific socket passed as an argument. In turn, the socket layer functions then make calls directly into the transport protocol modules, requesting specific functionality from TCP, or SCTP in the case of the shim.

An important distinction among the application layer, the various sublayers of the socket layer, and the transport layer is how sockets are treated at each level. Applications and the socket system call stub functions operate on *socket descriptors*: integer indexes into a lookup table maintained inside the kernel used to find the *socket data structure* the application is using (similar to how UNIX file descriptors work). Once the socket data structure has been located, the socket layer functions, transport protocol modules, and all lower layers use this socket data structure directly. The system call implementation sublayer is the transition point where a socket descriptor is mapped to a specific underlying socket object. Mapping the socket descriptor to the hidden SCTP socket rather than the normal TCP socket at this point is the basis of our TCP-to-SCTP translation scheme.

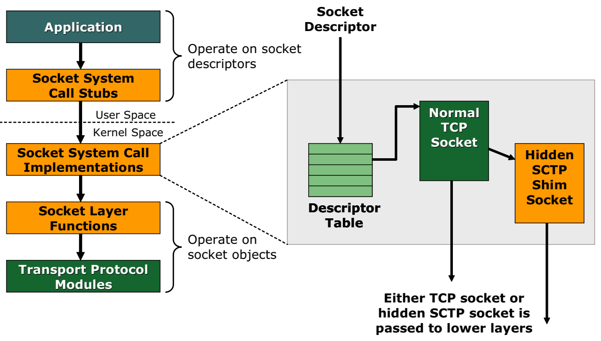
Application
Socket System Call Stubs
User Space
Kernel Space
Socket System Call Implementations
Socket Layer Functions
Transport Protocol Modules
Operate on socket descriptors
Operate on socket objects
Socket Descriptor
Descriptor Table
Normal TCP Socket
Hidden SCTP Shim Socket
Either TCP socket or hidden SCTP socket is passed to lower layers

Fig. 3. Socket descriptor/object mapping and socket substitution for lower layers

D. *Socket Substitution in System Calls*

Section II-A describes how every socket object points to the protocol switch structure for the transport protocol associated with that socket. Generic network requests from the application made through the sockets API are mapped to the specific protocol implementations that satisfy those requests using the interface function table found in the protocol switch structure. For example, the functionality of the `connect()` system call depends on the underlying protocol in use. TCP's `connect()` function must perform a 3-way SYN, SYN-ACK, ACK handshake, while SCTP requires a 4-way INIT, INIT-ACK, COOKIE-ECHO, COOKIE-ACK handshake. The actual code that is executed to fulfill the `connect()` request is selected via the mapping established by the protocol pointer in each socket data structure.

The existence of this socket-protocol mapping means the behavior at the transport layer depends entirely upon the protocol of the socket passed down through the kernel. The shim manipulates whether TCP or SCTP is used at the transport layer by intelligently deciding which of the two sockets to pass to the kernel's lower layers. *This technique of deciding which socket to substitute into calls to the lower layers of the kernel is the basis of our approach to implement transparent translation from TCP to SCTP.* While some system calls will clearly require more extensive modifications to support the desired shim functionality, many of the sockets-related system calls require only simple logic that decides to pass either the TCP socket or the SCTP socket to the lower layers, depending on the shim's current operating state. In the following sections, we describe exactly what the desired behavior of the shim will be for both client and server applications, and detail the changes made to the major sockets system calls.

III. Controlling Shim Use

To be practical, an operating system implementing the transparent TCP-to-SCTP translation shim needs to have an effective means for controlling the shim functionality. Whether in an experimental or production environment, the shim may be desirable for some applications but not for others. To encourage users to experiment with and make use of the shim when practical, we have designed and implemented a system to control the operation of the shim on both a system-wide and per-application basis.

In FreeBSD, most tunable operating system parameters are implemented as *sysctls*. The *sysctl* interface allows an administrator to configure the system's kernel variables dynamically [6]. Since network protocols typically have a large number of tunable parameters, the *sysctl* interface facilitates adjusting system properties without requiring editing of source code and kernel recompilation. We use *sysctls* for many of the shim's configuration variables.

The basic means of controlling the shim is through a global on/off switch for the entire system. Since this approach offers only a crude level of control, we enhance flexibility by dividing shim control into two broad classes: control over applications with local listening sockets (typically servers), and control over applications connecting to remote systems (typically clients). We define two new *sysctls* to specify the *default policy* for each of these two classes. The *sysctls*, shown below, are boolean variables where zero specifies the shim is disabled, and any nonzero value indicates the shim is enabled for that class of applications.

```
net.inet.sctp.shim.default_local_enable
net.inet.sctp.shim.default_remote_enable
```

A. *Shim Use Rules*

Dividing all applications into two control classes to manage shim functionality is not nearly fine-grained enough for serious use. However, the global default policies do form the basis for a more precise rule-based shim control system that allows an administrator to selectively enable or disable the shim on a per-application basis. The basic building block of the shim control system is a *rule* object, consisting of an IP address, a subnet mask, and two port numbers, illustrated in the following code listing:

```
struct shim_rule {
   int chain;
   int policy;
   int type;
   uint16_t port1;
   uint16_t port2;
   struct in_addr address;
   struct in_addr netmask;
};
```

In our design, a rule has only one address and one netmask, so a rule can represent a single address or all addresses on a certain network, but not both. Similarly, a rule has a maximum of two port numbers, so a rule can specify a single port or a range of ports, but not both simultaneously. However, because the storage for the address and mask is independent of the storage for ports, a rule can combine addressing and port information.

The combination of address, network, or port number(s) in use by a particular rule object is specified by the flags set in the *type* field. A rule also has two additional fields: *chain* and *policy*. The chain is either *local* or *remote*, specifying whether the rule applies to local listeners or applications connecting to remote peers. The policy is either *enable* or *disable*, reflecting whether or not the shim will be used for applications whose network parameters match the other fields of the rule (i.e., address, network, and port(s)). If any field of a rule is unspecified (i.e., only the address information is used and ports are unspecified), then the unspecified field is considered to be a wildcard and does not restrict the matching process.

B. *Shim Rules Organization & Operation*

All shim rules currently in effect on a system are grouped based on their *chain* and *policy* fields, resulting in four separate chains or classes of rules: *local-enable*, *local-disable*, *remote-enable*, and *remote-disable*. Subdividing the rules into these four chains enables quick location of relevant rules when the kernel initiates a lookup to determine whether to enable or disable the shim for a particular application. These four chains of rules comprise the *shim rules table*.

The shim rules table is maintained inside the kernel, and is consulted by the kernel whenever it needs to decide whether to enable or disable the shim for a particular application. Two system calls trigger a lookup using the shim rules table: `connect()` and `bind()`. These calls represent the two points where the kernel can easily examine the socket addresses (objects containing an address and port number) being passed in from the application. Specifically, the address and port number of the remote endpoint a client application wishes to contact are passed to `connect()`. Likewise, a local address and port to bind to are passed by a server application to `bind()`. Both `connect()` and `bind()` occur early in a socket's lifetime before any connection exists or data transfer occurs, so they are logical decision points to enable or disable the shim.

The per-application decision process to enable or disable the shim begins when a client application calls `connect()`, or when a server application calls `bind()`. The kernel first checks the value of the global default shim policy for the class of the application, governed by either the `default_remote_enable` *sysctl* for a client, or the `default_local_enable` *sysctl* for a server. Next, one of the four chains in the shim rules table is searched depending on the governing default policy. The rules in the four chains represent *exceptions* to the global default policies. For example, a default policy disabling the shim for local listening servers would require the kernel to search the *local-enable* chain for any rule specifically overriding the default and enabling the shim for matching server applications. If a match is found, the policy from the matching rule/chain is used instead of the global default policy. The other three cases are analogous to this example.

The search process in the shim rules table returns the policy of the *first* rule in the appropriate chain that matches the address and port information passed in by the application, *even if that rule is not the most specific match present in the list.* For example, assume a chain to be searched has one rule that matches any port in the range $1 - 1024$, and any IP address. Also assume that a rule later in the list has the specific port number 1000 and the specific network 10.1.2.0/24. Even if an application passes a socket address structure with port 1000 and address 10.1.2.3, the first rule is selected despite a more exact match appearing later in the list. Because of "first match" rather than "best match" behavior, administrators must consider rule ordering, placing the most specific rules first and more general rules afterwards.

IV. Shim States

To support the hidden socket first introduced in Section II-B, three new fields were added to the system's normal socket data structure: a pointer to the hidden socket, a shim state variable, and a pointer to the hidden socket's parent socket. Section III-B describes the operation of the shim rules table, and how it is used to enable or disable the shim for specific applications. In this section, we explain how the shim state variable is used in conjunction with the shim rules table to control the state of the TCP-to-SCTP translation shim for each application. The purpose and use of each state is defined below:

- **SHIM_NOTINIT:** The default shim state when a new TCP socket is created. When a new TCP socket is created using the `socket()` system call, initially the hidden SCTP socket does not exist. A state of *NOTINIT* in the normal socket declares to the kernel that the hidden socket is not yet created and should not be used. During the later stages of the `socket()` call, the hidden socket is created and the shim state of the normal parent socket changes to *READY*.
- **SHIM_READY:** After the hidden SCTP socket has been successfully created during the execution of the `socket()` system call, the normal parent TCP socket that points to the hidden SCTP socket receives

the state *READY* to indicate to the kernel that the hidden socket exists and is ready to be used if needed.

- **SHIM_ENABLE_MANUAL:** For debugging purposes, a special socket option exists to allow an application to directly enable the shim. While manual enabling of the shim would never occur in the case of a legacy TCP application, the option exists to allow shim developers to write test suites that work independently of the shim rules table. The *ENABLE_MANUAL* state indicates to the kernel that shim use has been manually enabled and should be used (when possible) for any interaction with peer endpoints.
- **SHIM_ENABLE_SRULES:** When the shim has been enabled for a particular application by the rules table lookup process described in Section III-B, the state of the normal parent TCP socket is set to *ENABLE_SRULES*. This state indicates to the kernel that the shim has been enabled and should be used (when possible) for any interaction with peer endpoints.
- **SHIM_ACTIVE:** During the `connect()` system call, if the kernel finds the shim state is *READY* and shim use has been allowed by either *ENABLE_MANUAL* or *ENABLE_SRULES*, then the kernel attempts to use an SCTP association to communicate with the remote endpoint, falling back to TCP if SCTP is unavailable. If the connection establishment phase with SCTP is successful, the shim transitions to the *ACTIVE* state, indicating to the kernel that the application has successfully connected to a remote peer using SCTP. Once the *ACTIVE* state is set in the parent TCP socket, the kernel knows to instead use the hidden SCTP socket for all network interaction with the peer.
- **SHIM_LISTEN:** Unlike a client application where the shim is enabled and enters active use during the span of a single system call, the process is divided into two steps for a server application. When a server application makes the call to `bind()`, the shim rules table is consulted and the shim state changes to *ENABLE_SRULES* if the bind address and port number match a rule in the appropriate chain of the rules table. Although enabled, the shim does not enter active use until the `listen()` system call is made. During the `listen()` call, if the kernel finds the shim state is *READY* and shim use has been allowed by either *ENABLE_MANUAL* or *ENABLE_SRULES*, the kernel then activates the shim by listening on the hidden SCTP socket and enters the *LISTEN* state.
- **SHIM_HIDDEN:** Unlike all of the previous shim states which are set in the normal parent TCP socket, the *HIDDEN* state is set on every hidden SCTP socket. This state allows the kernel to easily identify if a particular socket is a hidden SCTP socket rather than a normal TCP socket. The *HIDDEN* state also indicates that the parent pointer of the hidden socket is valid, and points to the parent TCP socket as shown in Figure 2.
- **SHIM_NATIVE:** Each of the shim states introduced so far applies to either a normal parent TCP socket or a hidden SCTP socket. The commonality is that parent and hidden sockets are created in direct response to a `socket()` request by an application. The *NATIVE* state instead applies to *new sockets created by the kernel* to represent the local communications endpoint for remote peers connecting to a local listening server. The sockets created by the kernel for each incoming connection to a shim-enabled TCP server are *native* SCTP sockets — they are independent sockets that do not have a parent or child relationship with any other sockets. While the functionality of the shim in a server scenario is described in detail in Section VI, the importance of the *NATIVE* state is that it allows the kernel to distinguish a native SCTP socket from the hidden SCTP socket paired with a normal TCP parent socket.

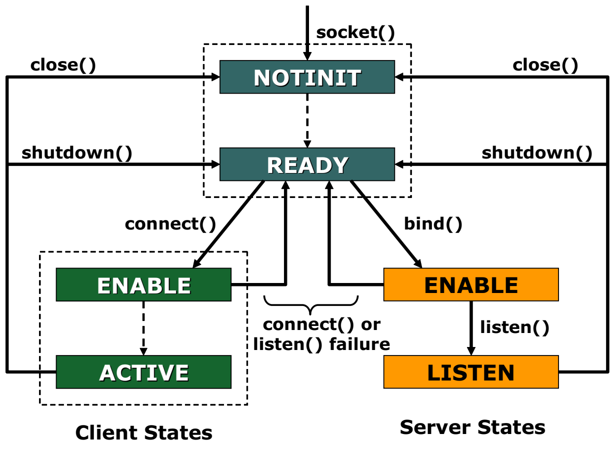

Fig. 4. Shim states and typical transitions for client and server applications (states grouped inside dotted boundaries occur in the span of a single system call)

Figure 4 shows the typical transitions between shim states for both client and server applications. Note the *HIDDEN* and *NATIVE* states are not pictured because they are immutable and any socket created with one of those states remains in that state forever (i.e., no transitions exist to or from these states).

V. Client Socket Functionality: Connect

The `connect()` system call is one of the major functions modified to support the TCP-to-SCTP translation shim. At a high level, the behavior of the `connect()` system call is as follows: first, the client application makes the `connect()` call, passing a socket address consisting of the address and port number of the remote endpoint into the kernel. Using the process described in Section III-B, the kernel decides if the shim should be enabled for the application making the call to `connect()`. If the shim is not enabled, the normal TCP connection process continues without modification. However, if the rules indicate the shim is to be enabled and the socket is in the *SHIM_READY* state, a different sequence of events takes place. Instead of establishing a TCP connection with the normal TCP socket, the kernel initiates the establishment of an SCTP association to the same remote address and port using the hidden SCTP socket. If the server running at that remote address and port supports SCTP (over the shim or natively), an SCTP association will be set up and all communications will be over SCTP. Once the peers are associated successfully using SCTP, the shim state for the application transitions to *SHIM_ACTIVE* signifying the SCTP socket is in active use for communications, and all future calls (i.e., `send()`, `recv()`, etc.) should be performed on the hidden SCTP socket rather than the normal TCP socket. If the SCTP association establishment fails the kernel falls back to a regular TCP connection. In that case the shim will remain in the *SHIM_READY* state, indicating the hidden SCTP socket is available for use but currently inactive.

Before association establishment is attempted using the shim, several socket and socket buffer configurations from the normal TCP socket are *cloned* and applied to the hidden SCTP socket. The reason for this cloning operation is that an application could create a socket and change several configuration parameters before the shim is actually enabled by the rules table lookup during the `connect()` call. If these parameters are set before enabling the shim, the hidden shim socket will not receive the updated configuration — the normal TCP socket will. The cloning step ensures the hidden SCTP socket receives any configurations applications make before the shim becomes active. Identical configurations guarantee the SCTP association behaves as the application expects in terms of the following:

- All socket options
- Socket linger settings
- Nonblocking and asynchronous I/O states
- Connection queue limit
- I/O signal handling function
- High/low water marks for socket I/O buffers
- Socket buffer asynchronous I/O flags
- Socket accept filter

To control the number of shim attempts to establish the SCTP association before falling back to a normal TCP connection, we have introduced a new *sysctl* to the kernel:

```
net.inet.sctp.shim.init_rtx_max
```

This parameter specifies the number of times an SCTP INIT PDU will be retransmitted before `connect()` falls back to using a regular TCP connection, allowing an administrator to balance the value of using SCTP versus the potential delay in connection setup.

VI. Server Socket Functionality

While a client application using the shim first attempts to establish an SCTP association and reverts back to TCP as a failsafe option when SCTP is unavailable, the server application functionality of the shim follows a *hybrid* approach that allows a single instance of a server process to serve *both* TCP *and* SCTP clients concurrently.

A. Bind

The `bind()` system call has two major roles when modified to support the shim. First, `bind()` performs a lookup into the shim rules table with the address and port number to be bound to, determining whether to enable or disable the shim for the application calling `bind()` using the process described in Section III-B. If the shim is to be enabled for the application, `bind()` will initially bind the hidden SCTP socket to the address and port number specified by the application. If this binding fails, the error is silently discarded and the application will only serve TCP clients. After binding the hidden SCTP socket with the address and port specified by the application, `bind()` then performs the normal bind with the TCP socket. Any error with the TCP bind is reported to the application, since the current design of the shim mandates that TCP always be available as a failsafe option, whereas it is expected that SCTP and the shim may sometimes be unavailable.

Additionally, we have implemented a new boolean *sysctl* to force `bind()` to override the application and bind the SCTP socket to all possible addresses on a multihomed system, rather than just the single address passed to `bind()` by the application:

```
net.inet.sctp.shim.force_bindall
```

This parameter avoids unnecessarily restricting the local addresses used by SCTP on a multihomed system when legacy TCP server applications without knowledge of multihoming blindly bind to a single address.

B. Listen

After calling `bind()`, an application next calls `listen()` to put the server's hidden SCTP socket (if the shim is enabled) and normal TCP socket into the listening state, allowing remote clients to begin connecting. If the shim is enabled, `listen()` also changes the shim state for the application to *SHIM_LISTEN* as described in Section IV, indicating that the application

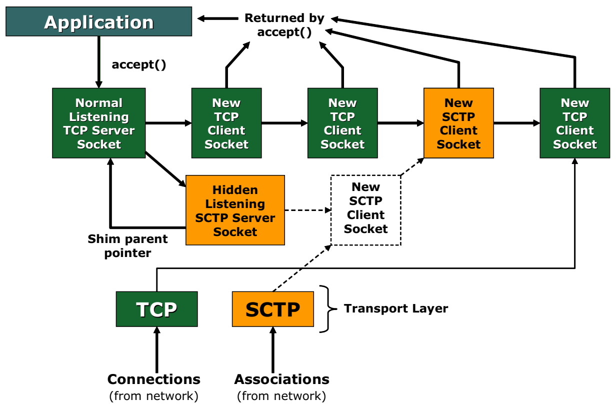

Fig. 5. Shim-enabled server architecture overview

is in a hybrid mode supporting both SCTP clients and normal TCP clients.

As well as enabling listening on the SCTP and TCP sockets, `listen()` also *clones* several socket and socket buffer configuration parameters from the normal TCP parent socket, and copies them to the hidden SCTP socket. This cloning operation allows an application to create a socket and change configuration parameters before the shim is actually enabled by the rules table lookup in the `bind()` call. Additionally, the cloning step ensures that servers operating in hybrid mode have the same configurations for both their SCTP and TCP sockets. (The cloned parameters are the same as those cloned by `connect()`.) Once the SCTP and TCP sockets are bound and listening, relevant TCP socket configuration parameters are cloned for the hidden SCTP socket, and the shim state for the application is set to *SHIM_LISTEN*, the server application may begin serving clients that connect via either TCP or SCTP.

C. Servicing Connecting Peers

Remote peer endpoints initiate communication with a TCP or SCTP server application by sending SYN or INIT PDUs, respectively. The transport protocol then handles the connection (TCP) or association (SCTP) handshaking process. During establishment, the transport protocol creates a new socket to represent the local communications endpoint for each connecting client. Upon completion of the establishment phase, this socket is then inserted into the listening socket's queue of established connections (or associations) awaiting service by the server application. The mechanism used to extract the newly created sockets from the waiting queue is the `accept()` system call. An application calls `accept()` to retrieve the first fully established, waiting socket from the front of the queue. The returned socket is then used according to the application protocol.

Recall that with the shim enabled, a server application actually has two listening server sockets: the normal TCP socket and the hidden SCTP socket. New sockets from connecting TCP clients are queued in the listening TCP socket's list, while SCTP sockets from newly connected SCTP clients would normally be queued in the listening SCTP hidden socket's list. To support a hybrid server approach, a server application needs some way of retrieving sockets from both lists, and a policy for deciding which list to choose from if both have waiting clients.

Rather than implementing the retrieval logic by modifying the `accept()` system call to retrieve sockets from both queues, we designed the shim to handle the problem at a lower level with a cleaner overall design. Our approach modifies the kernel's `sonewconn()` function, which is responsible for queuing the sockets of completed connections (associations) into the waiting list in the corresponding listening socket. If a newly created socket is slated for insertion into the hidden SCTP socket's waiting list, `sonewconn()` follows the hidden SCTP socket's *parent pointer* (introduced in Section II-B) and *queues the new socket into the normal TCP listening socket's queue instead*. Consequently, sockets for newly established client connections from both the TCP and SCTP listening sockets are queued and intermixed in a single list in the normal TCP listening socket. When the unmodified `accept()` call is made, both TCP and SCTP sockets can be returned from the waiting list to the server application, allowing for the desired dual-protocol hybrid operation of the shim. Figure 5 illustrates the architecture of the TCP and SCTP listening sockets and how newly created sockets are maintained in a single list.

VII. Socket I/O

Although the I/O system calls of the sockets API have some of the most complicated implementations compared to all of the other socket operations, the modifications required to allow them to support the shim are straightforward. The nature of a socket requires two sets of I/O functions to operate seamlessly with the operating system: dedicated network I/O calls (i.e., `send()`, `recv()`, etc.), and the standard UNIX I/O calls (i.e., `read()`, `write()`, `ioctl()`, etc.) implemented especially for sockets. The second set of functions is required because a socket descriptor can be used by an application similarly to a normal file descriptor, thus sockets need to support these standard file I/O calls. We now describe the shim modifications required for these two classes of I/O operations.

A. Network I/O Operations: Send & Receive Family

The six network I/O system calls of the sockets API are `send()`, `recv()`, `sendto()`, `recvfrom()`, `sendmsg()`, and `recvmsg()`. All of the sending and receiving functions are similar, differing only in the number of parameters that can be specified by an application at the time the I/O call is made [3], [4]. Though the differences between the calls may be relevant to application developers, they are treated identically by the shim inside the kernel.

All six of the send and receive system calls are built upon two fundamental kernel functions, `sendit()` and `recvit()`, which implement the core sending and receiving behavior, respectively. The six visible socket I/O calls build on these two base functions by adding additional processing to handle the various additional supported parameters. Consequently, adding support to the shim for network I/O only requires modification of `sendit()` and `recvit()`. Unlike the system calls that require significant additional code to support the desired shim features (i.e., `connect()`'s ability to fall back to TCP), the network I/O functions require only the simple socket substitution described in Section II-D to operate properly. The excerpt below illustrates the code used to substitute the hidden SCTP socket in place of the normal TCP socket to support the network I/O functions of the shim.

```
if(SHIM_ACTIVE(so)) {
   so = so->so_shimsock;
}
```

B. File Operations: Read, Write, Ioctl, Poll, & Stat

To support standard file I/O functionality, the file structure for a socket descriptor has a table of implementations of `read()`, `write()`, `ioctl()`, `poll()`, and `stat()` specific to a network socket rather than the usual file-specific implementations that are associated with a traditional file. The socket-specific versions of the standard file I/O calls are implemented by the `soo_read()`, `soo_write()`, `soo_ioctl()`, `soo_poll()`, and `soo_stat()` functions in the kernel. As with the network I/O system calls described above, these calls can be modified to support the shim simply by adding code to perform the substitution of the hidden SCTP socket in place of the normal TCP socket when the shim is in active use.

VIII. Socket Options

The client socket behavior with `connect()`, the server socket functionality of `bind()`, `listen()`, and `sonewconn()`, and the operation of normal socket I/O are the most critical aspects of the TCP-to-SCTP translation shim. However, the proper handling of socket options is essential to ensure all legacy TCP applications running over the shim work without unexpected side effects or problems. In the following sections, we describe the implementation of socket options and how the shim is designed to handle them correctly.

A. Socket Options Overview

Socket options are application-configurable protocol parameters maintained at different levels in the Internet protocol stack, manipulated using two system calls, `getsockopt()` and `setsockopt()`. All socket options are divided into levels based on the protocol that actually implements the option in question. Socket options that apply directly to the socket itself are included in the level *SOL_SOCKET*. All other levels are specified by the standard protocol number that is assigned to the protocol implementing the option. For example, options directed to TCP have the level *IPPROTO_TCP*, options to IP have the level *IPPROTO_IP*, and so on.

When an application gets or sets a socket option, each level of the protocol stack examines the level parameter of the specified option, checking for a match. If the protocol finds its own level matches the level of the option, the protocol module then checks the name of the option and handles the option as appropriate. If the current protocol is not the responsible level, the option is passed down the protocol stack to be handled by the correct lower layer.

B. Translating Socket Options

When the shim is introduced into the existing socket option system, some modifications are required for correct operation. While any options for the network layer and below will work normally even with the shim, socket options that affect the socket layer itself or the transport layer need to take special care to function correctly in the presence of the shim.

In the case of socket layer options, when the shim is enabled the options need to be applied to the hidden SCTP socket rather than the normal TCP socket. This functionality only requires a simple check: if the shim is in the enabled state, the kernel follows the link from the normal TCP socket to the hidden SCTP socket and passes the hidden socket to the lower layers instead. The lower layers then apply the specified option to the hidden socket rather than the normal socket.

In the case of socket options destined for the transport layer, an additional translation step is necessary to support the shim. Most TCP implementations have only two standard socket options: *TCP_MAXSEG* for getting and setting the maximum size of a TCP segment, and *TCP_NODELAY* for enabling and disabling the Nagle algorithm. If the shim is in the *ACTIVE* state and special precautions are not taken, these options might be invoked on the hidden SCTP socket by a legacy TCP application. However, with the shim enabled, TCP is not in the protocol stack as the socket option is passed down through the layers. The option would travel from the socket layer, to SCTP, to IP, and then down through the link layer. At no point along this path would TCP be able to intercept the option and handle it properly. Eventually the option would pass unclaimed through every layer, at which point an error would be returned to the application.

To avoid this situation when the shim is enabled, the kernel needs to translate TCP socket options to their SCTP equivalents. Since both the maximum segment size and Nagle algorithm are standard transport protocol attributes, SCTP also implements these options (*SCTP_MAXSEG* and *SCTP_NODELAY*). The code below is used in the kernel to translate both the socket option level from *IPPROTO_TCP* to *IPPROTO_SCTP* and the socket option names from their TCP versions to the corresponding SCTP versions when necessary:

```
switch(sopt.sopt_level) {
   case IPPROTO_TCP:
      sopt.sopt_level = IPPROTO_SCTP;
      break;
}

switch(sopt.sopt_name) {
   case TCP_NODELAY:
      sopt.sopt_name = SCTP_NODELAY;
      break;
   case TCP_MAXSEG:
      sopt.sopt_name = SCTP_MAXSEG;
      break;
}
```

IX. Experimental Evaluation

Using our TCP-to-SCTP shim implementation, we evaluated several popular applications running over the shim in terms of usability and performance. From a usability standpoint, we are interested in determining whether applications operate correctly if calls to TCP are transparently translated to SCTP, and SCTP replaces TCP at the transport layer without the application's knowledge. Additionally, we are interested in whether the user perceives any difference due to this change. Our experiments serve as a proof-of-concept that the shim idea is not only theoretically feasible, but also technically feasible.

Sections IX-A and IX-B describe the applications we verified to work correctly running over the shim in legacy-legacy mode and legacy-native mode, respectively. Besides showing applications can run over the shim without any visible changes in behavior or functionality, we quantify that applications running over the shim achieve performance equivalent to or greater than when running over a normal TCP connection. We describe these performance experiments in more detail in Section X.

A. Legacy-legacy Configuration

The shim's legacy-legacy mode is used to allow two legacy TCP peer applications to communicate using SCTP at the transport layer rather than TCP. This mode of shim operation allows the applications to take advantage of some of SCTP's advanced features without requiring any modifications to the applications themselves. We selected four test applications that represent the network usage of a typical Internet user: Telnet, SSH, HTTP, and Icecast [19] streaming audio. *Each application was compiled and installed in the standard fashion without any modification to the source code*. The particular implementations and versions of each application used in testing are as follows:

- **Telnet:** The standard Telnet [5] client and server applications distributed as part of the FreeBSD 4.10 operating system were used to test the functionality of a remote Telnet login session operating over the shim. Both the Telnet client and server functioned correctly while running over the shim and no errors or unusual behavior occurred during testing.
- **SSH:** In addition to Telnet, we also experimented with running SSH over the shim. SSH is a remote login protocol that incorporates encryption and is significantly more complex than Telent. The client and server programs used for our SSH experimentation were those included in OpenSSH 3.9p1 [14]. Our experiments found that SSH operated over the shim without any identified errors. Additionally, the SSH application suite includes the file transfer utility SCP that was used in the quantitative shim performance evaluation described in Section X. SCP also performed without error when operating using the shim rather than a normal TCP connection.
- **HTTP:** In the legacy-legacy configuration, we tested HTTP over the shim using Apache 2.0.43 [1] as the web server, and Firefox 1.04 [12] as the web browser client. In our experiments, we verified web pages (including all embedded objects, such as images) downloaded and displayed correctly in the browser when the interactions between the client and server were run over the shim's SCTP associations rather than TCP connections.
- **Icecast Streaming Audio:** Icecast [19] is a streaming audio server that streams music in the Ogg Vorbis [20] format. For our testing, we used the Icecast 2.2.0 streaming audio server and the XMMS [21] media player as the client. In our experiments, we found the audio quality when using the shim to be identical to the quality when using a normal TCP connection and did not encounter any problems such as skipping, distortions, or hangups in playback. In an additional experiment with Icecast, we played streaming audio across the shim between two multihomed systems. During playback, we disabled the currently active network path (by physically unplugging the associated interface) to test the

fault tolerance provided by the shim's underlying SCTP association. As expected, SCTP failed over to the remaining functional interface and playback continued without interruption. Performing the same experiment over TCP resulted in a complete failure of the connection and halted playback. We believe this experiment shows the shim's potential to bring SCTP's fault tolerance to legacy TCP applications.

Although the set of applications we experimented with is not exhaustive, we believe the normal, expected operation of these four applications when using the shim indicates the shim will be a viable and practical tool for existing legacy TCP applications. We are currently aware of only one potential problem with the shim, relating to legacy TCP applications that explicitly depend on the behavior of TCP's half-closed state. SCTP's design does not incorporate a half-closed state (either endpoint calling `close()` will terminate the whole association), so applications which depend on the semantics of TCP's half-close may exhibit unusual behavior when running over the shim.

B. *Legacy-native Configuration*

In addition to testing between legacy TCP endpoints, we also experimented with HTTP in the legacy-native configuration illustrated in Figure 1 using a version of the Apache webserver that was rewritten to *natively* support SCTP clients [16], and the same Firefox browser used in the HTTP experiments for the legacy-legacy configuration. In the legacy-native configuration, one endpoint, in this case the modified Apache server, is an application that is written to natively support SCTP, while the other endpoint, in this case the Firefox web browser, is a legacy TCP application using the shim to translate calls to TCP into corresponding calls to SCTP. Using Firefox, we conducted a sequence of page requests over the shim in legacy-native mode, browsing the documents available on our test server as a regular end-user would. The web pages, including all embedded graphics, were rendered identically to what users would experience when using normal TCP connections. The success of this test validates the incremental deployment motivation of the TCP-to-SCTP translation shim.

X. Performance Analysis

The experiments described in Sections IX-A and IX-B serve as a proof-of-concept for the shim, showing that the idea of translating calls to TCP into equivalent calls to SCTP without the application's knowledge is practical for several popular network applications. Showing that applications running over the shim function correctly is an important component of testing the TCP-to-SCTP translation shim. However, such tests do not quantitify the performance of application interaction when using the shim compared to when applications use normal TCP connections. In Section X-A, we describe the setup of our experimental evaluation of shim's performance in terms of application throughput during file transfers. Section X-B discusses the results and conclusions of the performance evaluation.

A. *Experimental Setup*

For our experiments, we measure the total time required to transfer files of various sizes using the SSH suite's SCP tool when running over a normal TCP connection and when running over the shim using an SCTP association. We compare the transfer times for TCP and the shim at a variety of loss rates.

- **Bandwidth/Delay Configuration:** We use a 1.5 Mbps, 35 ms delay path in all of our experiments, simulating the bandwidth and delay for a typical broadband Internet user in a US coast-to-coast connection configuration. The path is symmetric, so the bandwidth and delay are the same for client to server, and server to client.
- **Packet Loss Rates:** We examined transfer time with uniform loss rates of 0, 1, 3, 6, and 10%. Similar to the bandwidth-delay configuration, the loss rates are symmetric so the paths from client to server and server to client experience the same loss rate.
- **File Sizes:** To determine if the size of the file being transferred affects the transfer times for TCP or the shim disproportionally, we transfer files of size 50 KB, 500 KB, 5 MB, and 25 MB.

Each experiment required three nodes: a server machine running the SSH/SCP service, an SCP client, and an intermediate node running *Dummynet* [15] to simulate bandwidth, propagation delay, and loss rate configurations. The intermediate Dummynet router node was configured with a tail-drop queue of 50 packets and performed uniform random loss at the rates described above. Each node was a Pentium 4 system running FreeBSD 4.10 with a KAME kernel supporting SCTP. The SCTP version used was patch level 25, released in February 2005. To prevent the experiments with the shim from naturally taking advantage of the SCTP's multihoming ability and using other paths not part of the simulation topology, we disabled all interfaces besides the ones attached to the Dummynet-simulated network on the client and server systems before beginning the experiments. Disabling the alternate interfaces allowed for a fair comparison between TCP and the shim because SCTP was restricted to the simulated network, and was unable to use any alternate paths between the client and server nodes.

Each run of the experiment involved measuring the total time required to issue the SCP command to retrieve a single file on the client system, and then transfer the entire file from the server, including the SSH key exchange overhead. We used a public-key authentication configuration rather than passwords with SSH to allow the experiments to be run in a non-interactive batch mode. Every combination of file size and loss rate was run with the 1.5 Mbps/35 ms bandwidth-delay configuration a total of 30 times, except the 50K file experiments which were run 90 times per loss rate due to higher variance in the transfer times. Thus, each data point in the graphs shown in Section X-B is the average of 30 (or 90) runs of the same file size/loss rate configuration.

B. *Experimental Results*

Figures 6, 7, 8, and 9 display our recorded transfer times for each simulated loss rate for 50 KB, 500 KB, 5 MB, and 25 MB files, respectively. In situations where SCTP is available on both peer endpoints, *the shim adds no significant communications overhead beyond the inherent differences in the transport layer protocols being used or substituted*. The case where SCTP is unavailable on the remote system *can* be a source of overhead during the connection establishment process because at least one additional RTT is required to detect that SCTP is unavailable before reverting to TCP and starting data transfer. However, because the nodes in our experiment support SCTP and both the SSH/SCP client and server have the shim enabled, no additional overhead is introduced by using the shim. The differences in transfer times are entirely due to the specific features and implementations of the underlying transport protocols, not the translation process.

The graphs yield two main observations about application throughput over a TCP connection versus the shim with an SCTP association. First, in situations without any network-induced loss, TCP and the shim perform approximately equivalently. Although not visible in the graphs, TCP had a slight edge with average transfer times between 4 and 240 ms faster than the shim at 0 percent loss across all four file sizes tested. This difference is likely a result of SCTP's more complex four-way association establishment handshake compared to TCP's three-way handshake, and SCTP's more expensive CRC32 checksum. The second observation is that for all runs with loss rates greater than 1 percent, the shim running over SCTP outperforms TCP by an increasing margin as loss rates increase. The trend of the shim providing better application throughput at all loss rates greater than 1 percent holds across all files sizes, with longer transfers (i.e., 25 MB files) seeing a greater improvement than short transfers (i.e., 50 KB files). We argue that the greater throughput afforded by the shim in high loss conditions is due to the advanced congestion control features in SCTP, such as Limited Transmit, Appropriate Byte Counting, and Selective Acknowledgments, that are not present in FreeBSD 4.10's version of TCP (New Reno) [13]. Our results from experimenting with SCP over the shim confirm the same trends at high loss rates as earlier work [13] which experimented with various implementations of FTP running over SCTP. We speculate that if TCP were to incorporate the same congestion control features that SCTP currently supports, throughput for applications running over both the shim and normal TCP connections would be similar at all loss rates.

We feel our experimental results show the transparent TCP-to-SCTP translation shim is technically feasible, functions effectively with common network applications under realistic conditions, and provides performance (measured in terms of application throughput) that is equivalent to or better than what is possible using TCP.

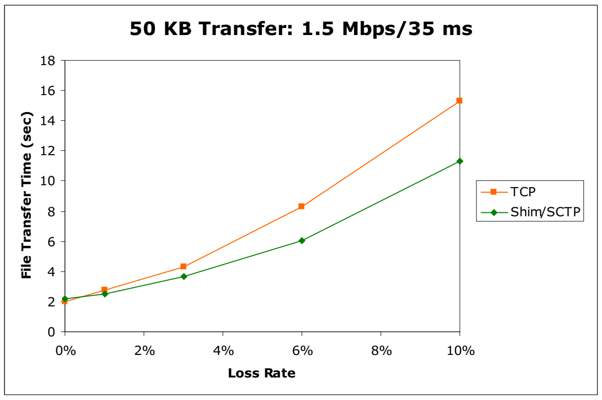

Fig. 6. Transfer Time vs. Loss Rate for 50 KB file transfer over 1.5 Mbps/35 ms delay link

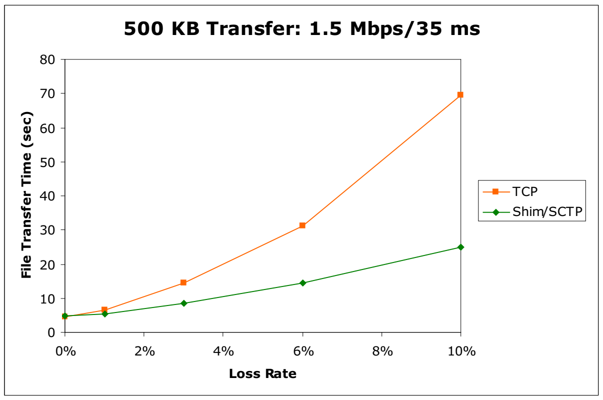

Fig. 7. Transfer Time vs. Loss Rate for 500 KB file transfer over 1.5 Mbps/35 ms delay link

XI. Summary

The transparent TCP-to-SCTP translation shim allows legacy TCP applications to enjoy SCTP's multihoming

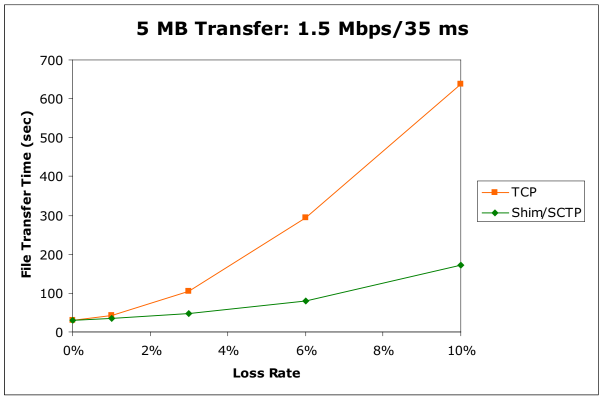

Fig. 8. Transfer Time vs. Loss Rate for 5 MB file transfer over 1.5 Mbps/35 ms delay link

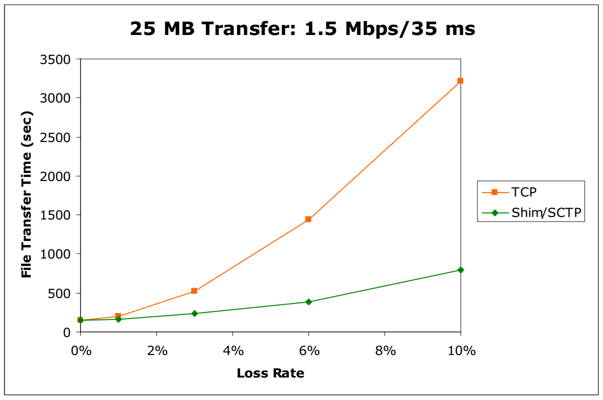

Fig. 9. Transfer Time vs. Loss Rate for 25 MB file transfer over 1.5 Mbps/35 ms delay link

advantages (i.e., fault tolerance and potentially concurrent multipath transfer) without requiring any modifications to the legacy applications themselves. Additionally, in non-multihoming situations, the shim encourages increased SCTP deployment by providing a path for gradual migration from legacy TCP applications to native SCTP applications, solving the incremental deployment problem typically experienced with new protocols.

Experimental results presented in Section IX illustrate that not only is the shim approach an interesting theoretical concept, but that the shim is technically feasible in practice with real applications under typical network conditions. The hope of this work is that users and developers alike will begin to appreciate how the advanced features provided by SCTP can be valuable for network applications. By ensuring that existing legacy TCP applications can seamlessly interact with new SCTP counterpart applications, the shim encourages innovation and the increased deployment of SCTP throughout the Internet.

Disclaimer

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U. S. Government.

References

[1] The Apache Software Foundation. Apache HTTP Server Project, July 2005. http://httpd.apache.org/.

[2] B. Braden. Requirements for Internet Hosts – Communication Layers. RFC 1122, Internet Engineering Task Force, October 1989.

[3] The FreeBSD Project. RECV(2) - FreeBSD System Manager's Manual, February 1994.

[4] The FreeBSD Project. SEND(2) - FreeBSD System Manager's Manual, February 1995.

[5] The FreeBSD Project. TELNET(1) - FreeBSD General Commands Manual, January 2000.

[6] The FreeBSD Project. SYSCTL(8) - FreeBSD System Manager's Manual, March 2002.

[7] J. Iyengar, P. Amer, and R. Stewart. Concurrent Multipath Transfer Using SCTP Multihoming over Independent Paths. *IEEE Transactions on Networking*, (to appear).

[8] J. Iyengar, K. Shah, P. Amer, and R. Stewart. Concurrent Multipath Transfer Using SCTP Multihoming. In *SPECTS 2004*, San Jose, July 2004.

[9] The KAME Project. Overview of the KAME Project, April 2005. http://www.kame.net/project-overview.html.

[10] E. Kohler, M. Handley, and S. Floyd. Datagram Congestion Control Protocol (DCCP). Internet draft, Internet Engineering Task Force, March 2005. http://www.ietf.org/internet-drafts/draft-ietf-dccp-spec-11.txt.

[11] S. Ladha and P. Amer. Improving Multiple File Transfers Using SCTP Multistreaming. In *IPCCC 2004*, Phoenix, April 2004.

[12] Mozilla Foundation. Mozilla Firefox Web Browser Homepage, July 2005. http://www.mozilla.org/products/firefox/.

[13] P. Natarajan, P. Amer, R. Bickhart, and S. Ladha. Corrections on: Improving Multiple File Transfers Using SCTP Multistreaming. Corrections to [11], Protocol Engineering Lab, June 2005. http://pel.cis.udel.edu/poc/index.html.

[14] OpenBSD Project. OpenSSH Homepage, July 2005. http://www.openssh.org/.

[15] L. Rizzo. Dummynet: A Simple Approach to the Evaluation of Network Protocols. In *ACM Computer Communication Review*, January 1997.

[16] R. Stewart. Port of Apache 2 Webserver to SCTP, July 2005. http://www.sctp.org/download.html.

[17] R. Stewart and Q. Xie. *Stream Control Transmission Protocol (SCTP): A Reference Guide*. Addison Wesley Professional, New York, NY, 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, Internet Engineering Task Force, October 2000.

[19] Xiph.Org Foundation. Icecast Homepage, July 2005. http://www.icecast.org/.

[20] Xiph.Org Foundation. Ogg Vorbis Homepage, July 2005. http://www.vorbis.com/.

[21] XMMS Team. X Multimedia System (XMMS), July 2005. http://www.xmms.org/.