TRANSPARENT TCP-TO-SCTP TRANSLATION SHIM LAYER

by

Ryan W. Bickhart

A thesis submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Master of Science in Computer Science

Summer 2005

TRANSPARENT TCP-TO-SCTP TRANSLATION SHIM LAYER

by

Ryan W. Bickhart

Approved: _____

Paul D. Amer, Ph.D.
Professor in charge of thesis

Approved: _____

Henry Glyde, Ph.D.
Interim Chair of the Department of Computer Science

Approved: _____

Tom Apple, Ph.D.
Dean of the College of Arts and Sciences

Approved: _____

Conrado M. Gempesaw II, Ph.D.
Vice Provost for Academic and International Programs

ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor, Professor Paul D. Amer, for his exceptional guidance over the course of my research and time at the University. While giving his students a large degree of independence and individual responsibility to manage their time and projects, he is always available and willing to help in any way possible. Whether reviewing presentations, helping to fine-tune wording, or forcing you to think critically about your project and goals, his attention to detail and genuine interest in students' work are admirable and deeply appreciated.

I would also like to thank my fellow researchers at the Protocol Engineering Lab, Armando Caro, Phill Conrad, Mark Hufe, Jana Iyengar, Sourabh Ladha, and Preethi Natarajan, for making my time at the lab such an enjoyable and memorable experience. Armando, Jana, and Sourabh brought me up to speed when I first joined the lab and were always a source of good advice and discussion. Their networking expertise continues to impress me to this day. Mark always had a great story to tell and his enthusiasm for his work was contagious. Preethi was very tolerant of my seemingly endless series of new kernels and constant rebooting of her machine. I have to give special thanks to Phill for his extensive technical advice. Whether helping me remember that the proper order for crimping wires in an RJ-45 connector is a sandwich with orange bread and blue meat, helping me prepare to take the plunge into the FreeBSD kernel, or giving me a crash course on VLAN configuration, he has been an invaluable resource. Thanks to all of you for making my time at PEL great.

In particular, I would also like to thank Randall Stewart of Cisco Systems for his unfaltering willingness to help me track down and fix bugs in SCTP and his insightful

advice relating to my project. Randall always impresses me by getting updated SCTP code to my inbox in what seems like mere minutes after I report a problem.

I would additionally like acknowledge the financial support for this thesis provided by Cisco Systems, Inc. through the University Research Program, and the U.S. Army Research Laboratory Collaborative Technology Alliance in Communications and Networks.

Last but certainly not least, I would like to thank my friends and family for all the support over the years, particularly the last several months of intense work on this research project. I appreciate you putting up with my limited amount of free time, long days at the lab, delays before returning phone calls, and frequent (and often unsolicited!) technical talk. I would like to give special thanks to my roommate Matt for helping me proofread this thesis and being there throughout this whole research adventure. I appreciate all the times you made dinner while I was busy programming, running experiments, or writing.

Finally, I'd especially like to thank my parents, John and Lisa, for taking interest in my work and encouraging me from my finger painting days all the way through graduate school. Thanks for always providing me with the tools I needed to succeed and for supporting me along the way. You guys are the best!

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Stream Control Transmission Protocol (SCTP) is a connection-oriented, reliable, messaged-based, general purpose transport protocol with congestion control similar to that used by TCP, supporting advanced features not available in TCP or UDP, such as multistreaming and multihoming capabilities. Because SCTP is still relatively new, it has not yet been widely deployed in the Internet despite its many advantages over TCP and UDP, particularly the fault tolerance provided by multihoming and potential for concurrent multipath transfer. The current state of SCTP deployment is essentially a "chicken and egg" type problem, where application developers are not interested in using SCTP at the transport layer because end users do not demand its services, but end users do not demand SCTP services because no current applications are written to support them.

To encourage developers and end users to begin adopting SCTP and build momentum for more widespread SCTP deployment, we have developed a shim layer which translates application-level system calls to TCP into corresponding calls to SCTP, allowing legacy TCP applications to communicate using SCTP as the end-to-end transport protocol *without any modifications to the applications themselves*. This translation occurs transparently, so legacy TCP applications are *unaware* translation to SCTP is occurring. If the shim detects that translation from TCP to SCTP is not possible for a particular endpoint or service, the shim will fall back to using a normal TCP connection, ensuring backwards compatibility.

The TCP-to-SCTP translation shim layer has been implemented in the FreeBSD 4.10 operating system kernel, and supports both client and server functionalities. The

shim enables communication between peers using two general architectures: a legacy-legacy configuration allowing two legacy TCP applications to communicate over SCTP, and a legacy-native configuration allowing a legacy TCP application to use the shim to communicate with a native SCTP application. The legacy-legacy mode allows TCP applications to gain use of some of SCTP's advanced features without requiring modification of the applications themselves, while the legacy-native modes allow legacy TCP applications to interact with their native SCTP counterparts, providing a gradual migration path to increased SCTP deployment.

Experimental results 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 (gauged qualitatively in terms of user perceptions and measured quantitatively in terms of throughput) using the shim and SCTP is equivalent to or better than performance when applications operate using TCP as originally designed.

Using the transparent TCP-to-SCTP translation shim lets legacy TCP applications take advantage of some of SCTP's advanced features, and allows for interoperability between legacy TCP applications and their SCTP equivalents. The shim provides a gradual migration path from TCP to SCTP and facilitates incremental deployment, encouraging developers and users to begin taking advantage of SCTP's advanced features.

Chapter 1

INTRODUCTION

1.1 Shim Overview

This thesis 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 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. In the following sections, we first introduce SCTP and the services it provides, as well as the concept of multihoming. Next, we motivate the TCP-to-SCTP translation shim layer by illustrating the advantages of replacing TCP with SCTP as the end-to-end transport between two communicating peers. Finally, we conclude with an overview of the organization of the remaining topics of the thesis.

1.2 Stream Control Transmission Protocol

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 the current transport protocol workhorses of the Internet, TCP

and UDP [24]. 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 set of 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 [23]. Arguably one of the most unique and important features of SCTP is support for *transport layer multihoming*. We describe multihoming and mention the impacts multihoming has on the transport layer in the following section.

1.3 Multihoming

A host which has more than one interface is said to be *multihomed* [1]. Historically, hosts on the Internet have typically 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. As such, the traditional Internet protocols, IP, TCP, and UDP, have no concept of multihoming since they were not designed with multihoming in mind.

While uncommon in the past, 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. For example, almost all laptops sold today include both wired (Ethernet) and wireless (802.11) interfaces as part of the standard configuration. Simultaneous network connectivity in the form of wireless LAN technologies such as 802.11 and wireless WAN technologies such as cellular links is often possible for mobile users in many areas. Likewise, home users often have the potential to obtain Internet access through different providers offering high-speed cable or DSL connections.

Despite the fact that today multihoming is frequently economical and practical, support for multihoming is lacking in the current versions of the major Internet protocols,

IP, TCP, and UDP. These protocols are limited by the fact that they were designed in an era where multihoming was never considered as a design issue. Consider the situation with TCP: a TCP connection is defined by a four-tuple consisting of a pair of IP addresses and a pair of port numbers, one per endpoint. No provision exists in TCP for handling more than one IP address at a given endpoint of a connection. SCTP breaks this mold by providing integrated support for multihoming at the transport layer. We illustrate the differences between TCP and SCTP when considering multihoming using Figure 1.1.

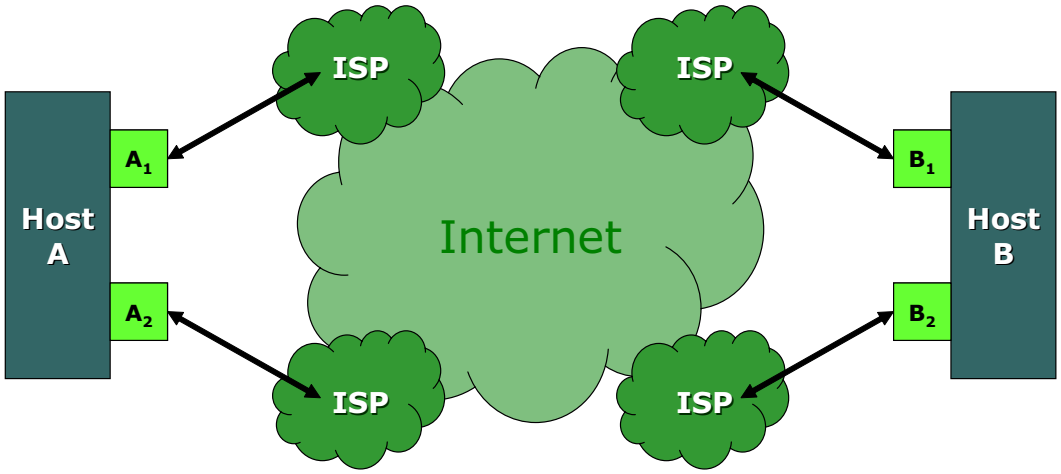

Figure 1.1: Example multihoming topology

1.3.1 Multiple Addresses in TCP

Consider the example multihoming configuration in Figure 1.1 where two hosts, A and B, each have two interfaces and IP addresses, A_1 and A_2 for host A, and B_1 and B_2 for host B. Because TCP has no conception of multihoming, only one of the four combinations of address pairs, $\{(A_1, B_1), (A_1, B_2), (A_2, B_1), (A_2, B_2)\}$, between the two hosts can be used for a single TCP *connection*. TCP was simply not designed to allow for more than one address per endpoint of a connection. As such, even if endpoints have multiple addresses, TCP may use only one address at each endpoint per connection.

1.3.2 Multihoming in SCTP

SCTP natively supports multihoming at the transport layer. Consequently, an SCTP *association* between hosts *A* and *B*, unlike a TCP connection, consists of the *entire set of addresses* available at each endpoint, in this case $(\{A_1, A_2\}, \{B_1, B_2\})$. SCTP can use *any* feasible (i.e., allowable by the routing and firewall situation on intermediate networks) 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. While not all possible combinations of source and destination addresses may be functional in every situation due to routing configurations and the intermixing of public and private addresses, in theory all possible combinations are initially available to SCTP and unworkable combinations of addresses are removed from use.

1.4 Motivations for TCP-to-SCTP Translation

The integrated support for multihoming in SCTP is the basis of two important motivations for replacing calls to TCP with equivalent calls to SCTP using the transparent shim translation layer. The first motivation is the ability to provide *fault tolerance* to legacy applications by using SCTP's multihoming support. A second motivation is the possibility of taking further advantage of SCTP's multihoming capabilities to enable *concurrent multipath transfer* [7]. Figure 1.2 shows the general architecture for two legacy TCP applications communicating using the shim. By translating calls to TCP into their SCTP equivalents, the shim allows legacy TCP applications to communicate using an SCTP association, potentially taking advantage of SCTP's fault tolerance and concurrent multipath transfer abilities. Even in situations where multihoming is unavailable on the endpoints using the TCP-to-SCTP translation shim, the shim provides the ability for integration between legacy TCP applications and native SCTP applications, as shown in Figure 1.3. Thus, the shim solves the "chicken and egg" problem and allows for the *incremental deployment* of SCTP.

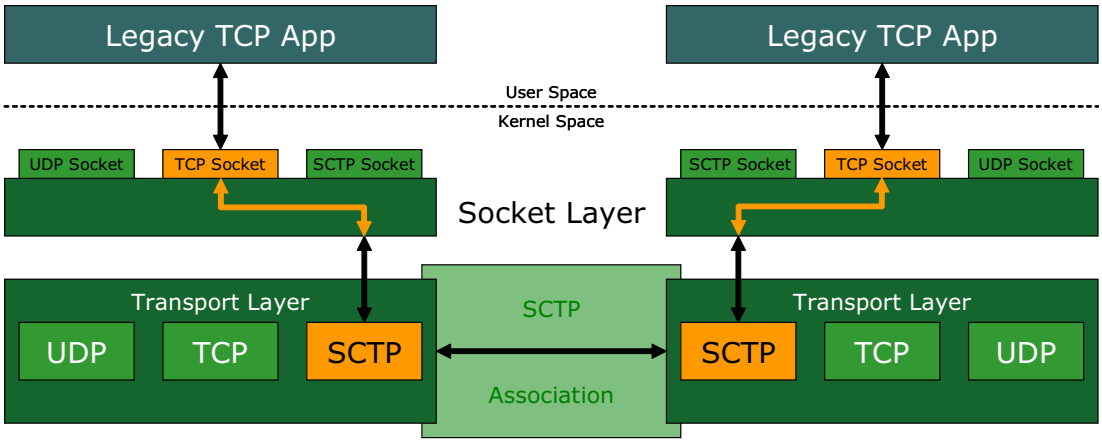

Figure 1.2: Shim legacy-legacy architecture

1.4.1 Fault Tolerance

SCTP defines the concept of a *primary destination address*. New data is actively sent to this address while any remaining addresses on a multihomed system, termed *alternate destinations*, and are held in reserve for retransmission of data originally sent to the primary destination in the case of loss or path failure. By using the TCP-to-SCTP translation shim, TCP applications running on multihomed hosts will be able to take advantage of 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.

1.4.2 Concurrent Multipath Transfer

Concurrent multipath transfer (CMT) is an area of ongoing research which involves using multiple network paths for concurrent transfer of new data [7]. 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 [24]. 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 TCP-to-SCTP translation shim layer, legacy TCP applications with multiple addresses will be able to take advantage of the fault tolerance provided by SCTP multihoming, and eventually the increased throughput of CMT.

1.4.3 Gradual Migration from TCP to SCTP

Even when the endpoints using the TCP-to-SCTP translation shim are not multihomed and therefore cannot make use of the fault tolerance and CMT features available with SCTP multihoming, the shim still serves an important purpose by allowing for a gradual migration path from TCP to SCTP.

One of the main problems with the adoption of any new network protocol is the issue of deployment. As an example, consider the situation with IP version 4 and IP version 6. IPv4 has shortcomings which are rectified in IPv6. However, there is also a widely deployed base of IPv4 users already in existence. How to smoothly transition all of the existing IPv4 users to IPv6 without interrupting service across the entire Internet is a difficult problem because the two protocols are not directly interoperable.

A similar situation exists when considering TCP and SCTP deployment. While 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. Because of this situation, little motivation exists among application developers to design applications that take advantage of SCTP's advanced features at the transport layer because few users demand SCTP support. Likewise, users do not demand SCTP support for their applications because support for SCTP is currently limited.

The TCP-to-SCTP translation shim layer encourages migration from TCP to SCTP (when appropriate and desirable) by allowing for integration and 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 1.3. Enabling legacy TCP

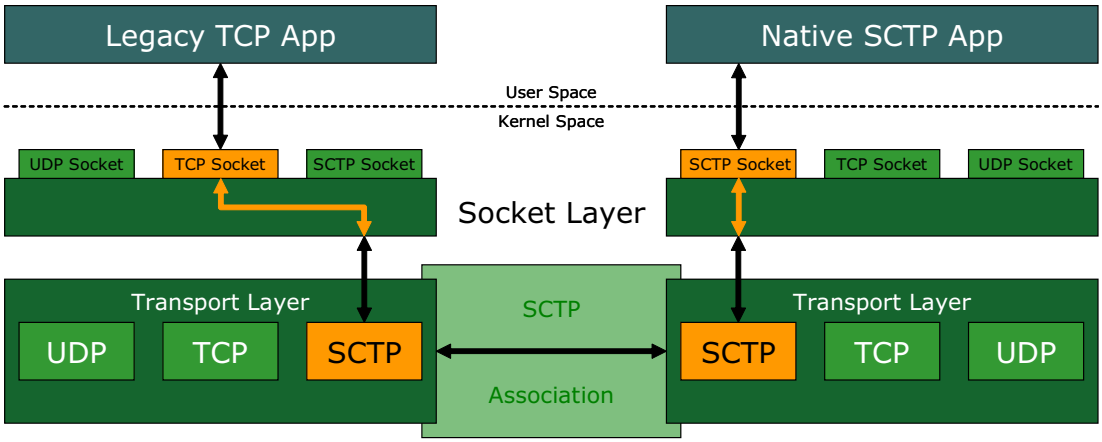

Figure 1.3: Shim legacy-native architecture

applications to interact with their native SCTP peers gives developers 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.

1.5 Organization

The remainder of this thesis is organized as follows: Chapter 2 focuses on the design and implementation of the shim layer, including the general approach taken, the functionality of of the major components, and the implementation details. Chapter 3 describes the experimental evaluation performed on the shim and the results of both proof-of-concept and performance testing. Finally, Chapter 4 begins with concluding remarks about the TCP-to-SCTP shim, and then suggests several areas of future future work and research directions to investigate.

Chapter 2

DESIGN & IMPLEMENTATION

2.1 Design Approach

Currently, the most mature, robust, feature-rich, and stable kernel implementation of SCTP is found in the BSD family of operating systems and distributed as part of the KAME project [18]. The KAME group is a consortium of companies primarily concerned with developing a fully-functional IPv6 and IPSec protocol stack. In addition, the KAME distributions also include support for other up-and-coming or experimental network protocols such as SCTP and DCCP [8]. Since a functional TCP-to-SCTP translation shim is absolutely dependent on a stable kernel implementation of SCTP, we selected FreeBSD to be the operating system used for our shim implementation. Consequently, the *specific* design details presented in this thesis are not totally and exactly applicable to all operating systems, though we believe the *general* concepts should be widely applicable to any operating environment supporting the standard sockets API.

At the time the shim implementation work 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 before attempting to port the shim to later versions of FreeBSD or other operating systems entirely.

The following sections introduce the major design aspects of the transparent TCP-to-SCTP translation shim. As a preliminary, we first discuss the rationale for implementing the shim in the kernel rather than as a user library (Section 2.1.1). Next, we introduce

the system's socket data structure (Section 2.1.2) and describe how it is coupled with a specific transport protocol (Section 2.1.3). Afterwards, we introduce the key insight into the design of the shim and show how translation from TCP to SCTP is accomplished in general (Sections 2.1.4, 2.1.5 and 2.1.6). The remaining sections describe in detail how each socket system call is modified to support the shim, and how each component of the shim operates.

2.1.1 Kernel Implementation Versus User Library

One of the first major design decisions faced when beginning the TCP-to-SCTP shim project was the issue of 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 overview. Implementing the shim as a user library has several advantages over an implementation in the kernel:

- The primary advantage of a library implementation is not requiring users to recompile or otherwise upgrade their operating system kernel to make use of the shim functionality. Modifying the kernel can be a difficult task for nontechnical users, as well as potentially dangerous if the kernel is not precisely configured and compiled to operate correctly with the underlying hardware.

- The shim implemented as a user library using the standard sockets API would likely be more portable between different operating systems or operating system versions than a kernel-based shim.

- Control over which applications use the shim is simplified since only applications specifically recompiled with the shim library would actually use the shim.

However, the safety and simplicity of using a user-space library for the shim has its disadvantages. Most notably, the use of a user library requires the user to recompile/relink each and every application for which they wish to enable the shim. In some

cases this effort may be a nuisance; as when a user has a large number of applications that should be shim-enabled. In other situations, recompiling all the necessary applications might be impossible; consider the case where a user has some proprietary applications which are distributed only in binary form for which no source code is available to allow recompilation.

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 recompilation, relinking, or any other modifications whatsoever. Additionally, because the shim is inside the kernel, the designer has more flexibility and broad discretionary powers about exactly how the shim will work than is possible in 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 the shim library as in a user-space implementation).

Because one of our main project goals is *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 and production-quality implementation than is possible with a user library.

2.1.2 Socket Structures & Socket Model

The sockets API allows applications to interact with the network in a uniform, protocol-independent and system-independent way. At the core of the sockets model is

the notion of a *socket*. A socket is a single 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 socket state and configuration information, as well as input and output buffers. In addition to the aforementioned components, one additional field is of primary importance to the shim work: the protocol field. The protocol field distinguishes sockets of different protocols. For example, applications might create and use a UDP socket, a TCP socket, or an SCTP socket. The socket data structures created in each of these three cases would be practically identical in terms of their basic makeup except for the protocol field.

Transport protocols supported by an operating system are grouped according to *domains* and *types* in the sockets model. The domain of a protocol refers to its communications domain, in other words, the addressing scheme. The two most notable domains on a typical system are the *AF_INET* (IPv4) and *AF_INET6* (IPv6) domains. Within a domain, protocols are subgrouped by type, where the type broadly identifies the communication semantics associated with a protocol. The typical types in a contemporary operating system supporting SCTP are *SOCK_DGRAM* for unreliable datagram-based protocols, *SOCK_SEQPACKET* for reliable datagram-based protocols, and *SOCK_STREAM* for reliable stream-based protocols. Although the original sockets model design allowed for multiple protocols to implement a certain type of service within a particular domain, this design is not the case in practice. In reality a one-to-one mapping exists between a type of service and the protocol that implements that service within a given domain. For example, in contemporary systems, the only protocol that supports the *SOCK_DGRAM* type of service is UDP, likewise for *SOCK_SEQPACKET* and SCTP, and for *SOCK_STREAM* and TCP.

The operating system kernel maintains a list of protocols for each domain where the members of the list are data structures that completely define a protocol. The data

structures, called the *protocol switch structures*, contain fields that specify the domain and type of the protocol, the protocol number, a series of flags (detailing characteristics such as whether or not a protocol is connection-oriented, or if addresses are passed with each write), and lists of the interfaces/entry points into the protocol. The protocol field of each socket points to one of these protocol switch structures, as shown in Figure 2.1. 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.

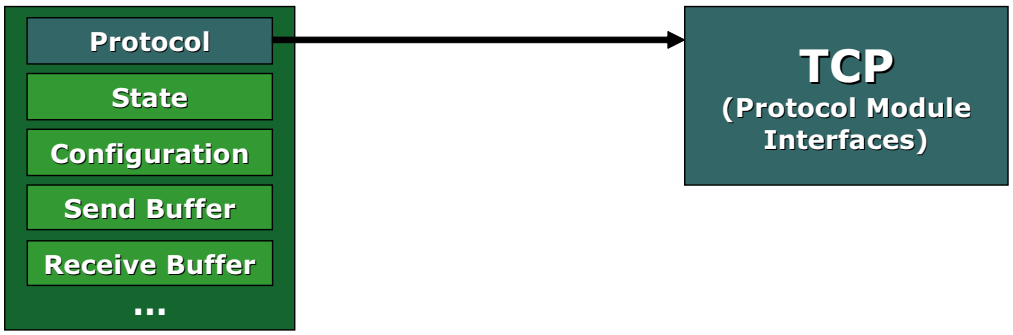

Figure 2.1: Normal TCP socket format

An important distinction to make when dealing with sockets is the difference between a socket data structure, which is an object allocated inside the operating system kernel, and a socket descriptor, which is an identifier used by the application to reference a certain socket object inside the kernel. The mapping from a socket descriptor to its corresponding socket object occurs through the kernel's descriptor table. The importance of this distinction in relation to the TCP-to-SCTP translation shim will be explained in Section 2.1.5.

2.1.3 Protocol Attach & Detach

The binding between a generic socket object and a specific protocol that transforms a socket into a socket of a specified protocol occurs when a socket is created. Recall the usual process by which an application creates a new socket using the `socket()` system call, shown in the code listing below. Note how the domain, type, and protocol values described earlier are passed to `socket()` so that the requested type of socket is created by the kernel.

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

When `socket()` is called, the kernel uses the specified domain, type, and protocol parameters to find the switch structure for the requested protocol in the lists of protocols the kernel maintains. After locating the proper protocol switch structure, the kernel uses the protocol's interface table to call that protocol's `attach()` function. The `attach()` call serves two purposes. First, `attach()` implements the binding between the socket and the specific protocol. Second, `attach()` 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.

One might initially conclude that to support a TCP-to-SCTP translation shim, the most obvious approach 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 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 as well. The reason for deallocating the socket is a design issue. When the FreeBSD networking subsystem was designed, the assumption was that both `attach()` and `detach()` would be called exactly once during the lifetime of a socket, since changing

the protocol a socket is bound to while it is in use would not typically be considered desirable or useful.

2.1.4 Hidden SCTP Socket

As a result of the behavior of `attach()` and `detach()`, the TCP-to-SCTP translation shim requires two separate socket data structures, one bound to TCP and the other bound to SCTP. The TCP socket is created normally as a result of the usual `socket()` system call. On a system incorporating the TCP-to-SCTP translation shim, the traditional `socket()` system call is then modified so that in addition to creating the normal TCP socket, a second *hidden* SCTP socket is created as well. It is termed *hidden* because it is created by the kernel but not exposed to the application. The existence of a traditional socket is made known to the application because a socket descriptor is returned; 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 some way of keeping 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 TCP socket. Figure 2.2 illustrates the relationship between a normal TCP socket and its hidden SCTP socket. The new *shim state* and *shim parent* fields visible in the figure are discussed in Sections 2.1.6 and 2.5.3, respectively.

2.1.5 Socket Layer in Detail

Networking specialists think of the 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 one of the most critical layers: the socket layer. The primary focus of the shim work is the socket layer, acting as an

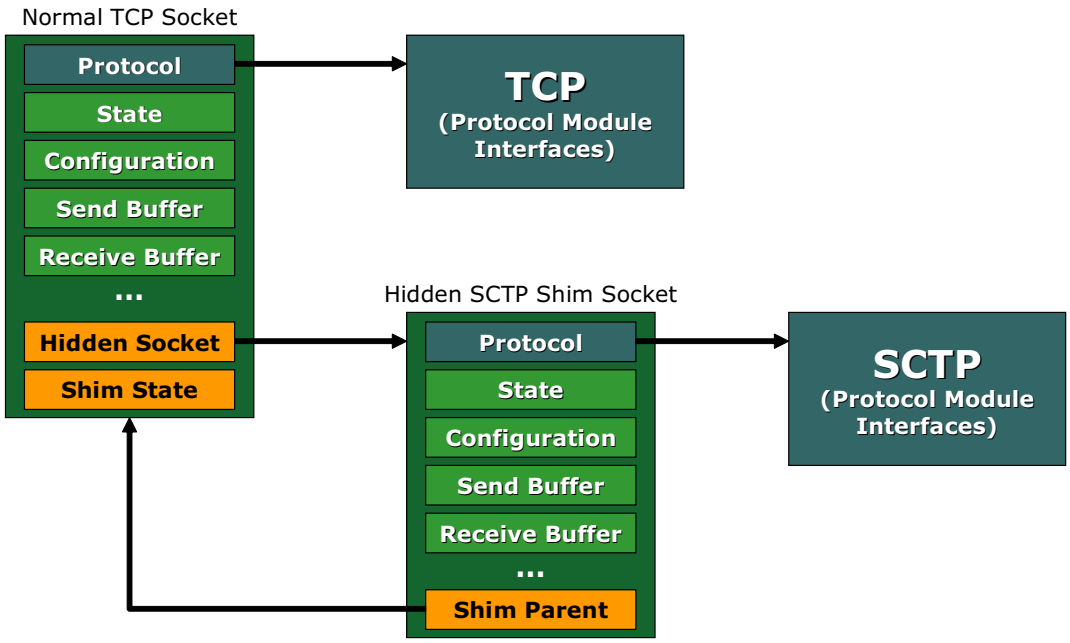

Figure 2.2: Normal TCP socket with hidden SCTP socket

intermediary between an application and the transport protocols. The socket layer itself is actually composed of several sublayers as shown in Figure 2.3.

The top sublayer of the socket layer is made up of the *socket system call stubs*. These functions form the API applications use to make requests upon the lower layers. The stub functions do not perform any actual networking actions themselves; they only package application layer arguments into the 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 network services.

The sublayer immediately below the level of 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. All of the typical socket system calls such as `socket()`, `connect()`, `bind()`, `listen()`,

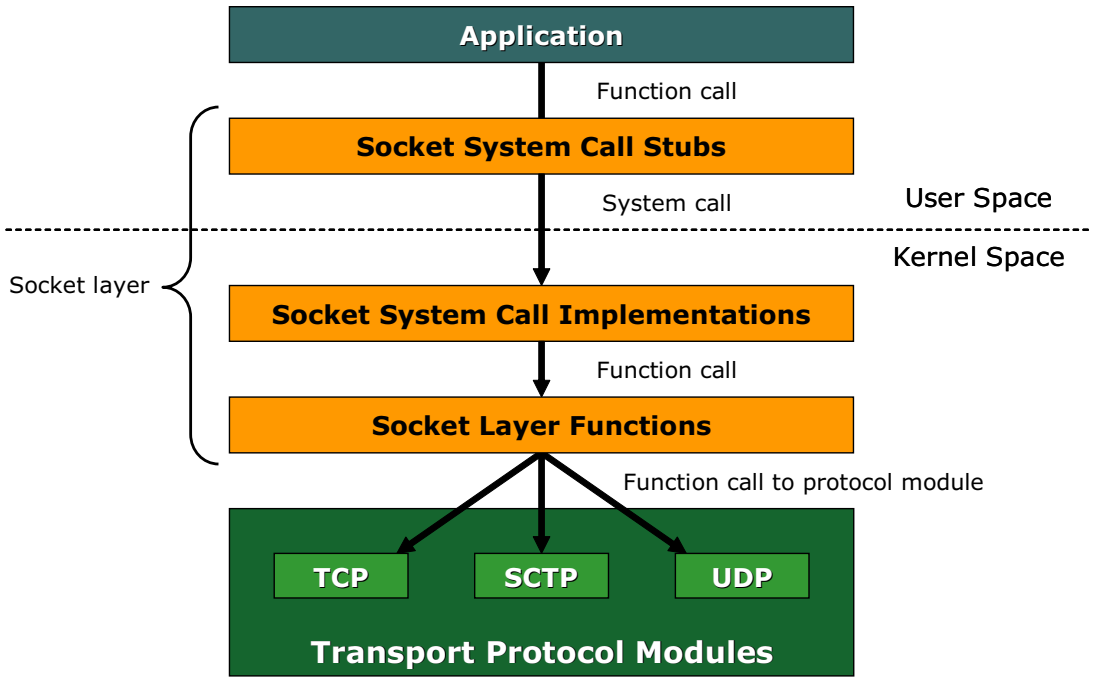

Figure 2.3: Socket layer in detail

`accept()`, etc., are implemented here.

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 themselves, 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 that is 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,

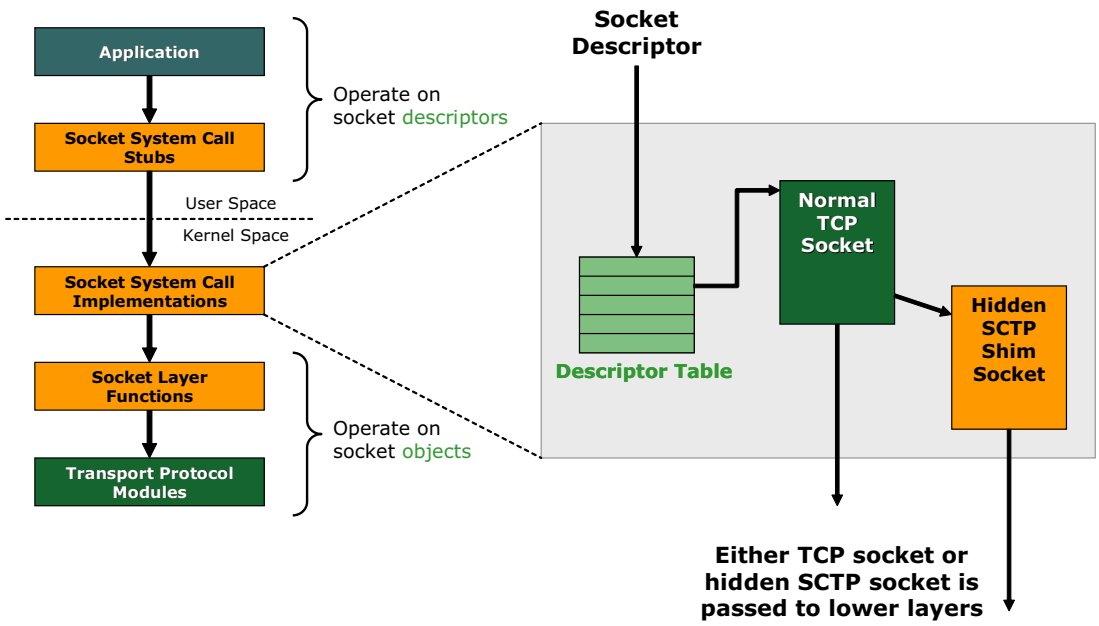

Figure 2.4: Socket descriptor/object mapping and socket substitution for lower layers

and all lower layers use this socket data structure directly. The transition point where a socket descriptor is mapped to the specific socket object that the descriptor references occurs at the socket system call implementation level. At this point, the kernel can locate the normal TCP socket object referenced with a descriptor by the application, and operate on the socket object corresponding to that descriptor. Figure 2.4 illustrates this mapping from socket descriptors to socket objects in the kernel. Extended to incorporate the TCP-to-SCTP translation shim idea, this point is where the kernel can follow the link from a normal TCP socket to its hidden SCTP socket (described in Section 2.1.4), and then operate on the hidden SCTP socket as well. We describe how this selection between a normal TCP socket and its hidden SCTP socket relates to the functioning of the translation shim in Section 2.1.6.

2.1.6 Socket Substitution in System Calls

As described in Section 2.1.2, every socket object has a pointer to the protocol switch structure for the transport protocol associated with the socket. Using the table of interface functions found in the protocol switch structure, generic network requests made through the sockets API are mapped to the specific protocol implementation that executes to serve those requests, as shown in Figure 2.5. For example, suppose an application calls `connect()`. The functionality of the `connect()` system call depends on the underlying protocol in use. TCP's `connect()` function must perform a three-way SYN, SYN-ACK, ACK handshake; while calling `connect()` with UDP simply sets state on the local system; and calling `connect()` in SCTP requires an INIT, INIT-ACK, COOKIE-ECHO, COOKIE-ACK four-way 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.

Because of the existence of this socket-protocol mapping, the behavior at the transport layer depends entirely upon the protocol of the socket passed down through the kernel. Using the TCP-to-SCTP translation shim, every TCP application has two associated sockets: a normal TCP socket that is created by default, and a hidden SCTP socket that is linked to the normal TCP socket. The shim manipulates whether TCP or SCTP is used at the transport layer by intelligently deciding which of the two sockets to pass down to the lower layers of the kernel. *This technique of deciding which socket to substitute into calls to the lower layers of the kernel is the core approach to how transparent TCP-to-SCTP translation is accomplished in this work.* While some system calls 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 each affected

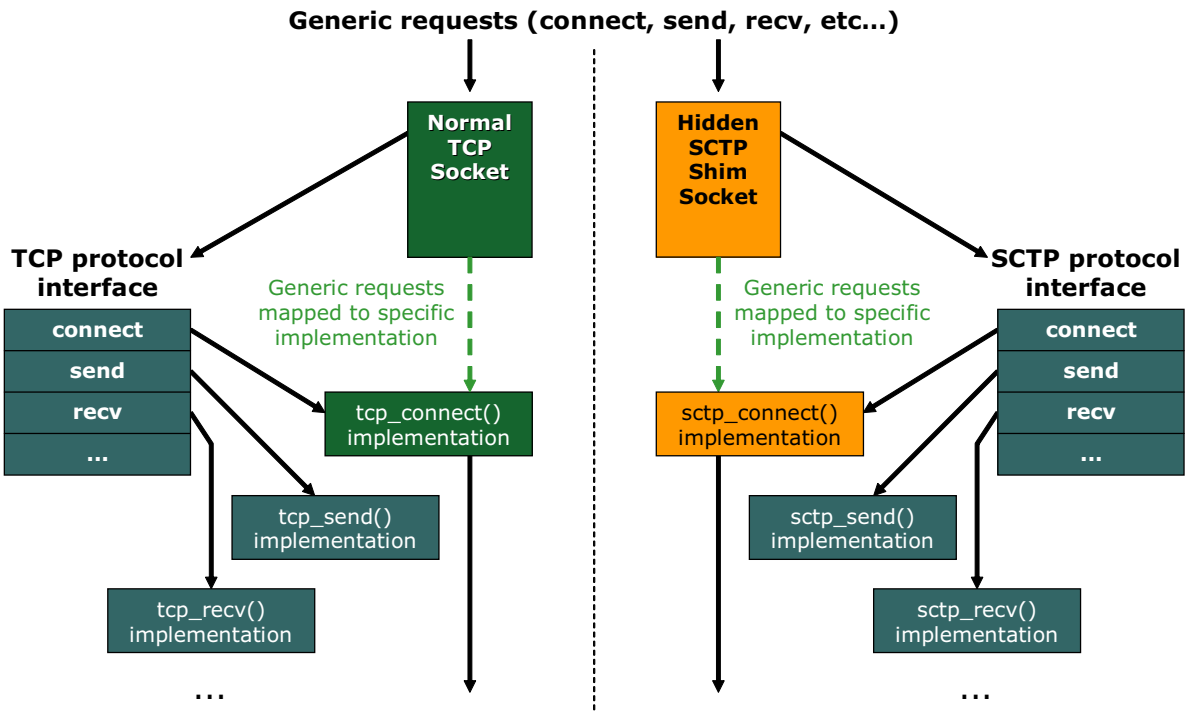

Figure 2.5: Mapping of generic application requests to specific protocol implementations

Table 2.1: System calls and functions modified by shim

System Call / Function	Source File	Type
socket()	uipc_syscalls.c	Socket API system call
bind()	uipc_syscalls.c	Socket API system call
listen()	uipc_syscalls.c	Socket API system call
connect()	uipc_syscalls.c	Socket API system call
sendit()	uipc_syscalls.c	Socket API system call
recvit()	uipc_syscalls.c	Socket API system call
shutdown()	uipc_syscalls.c	Socket API system call
setsockopt()	uipc_syscalls.c	Socket API system call
getsockopt()	uipc_syscalls.c	Socket API system call
getsockname()	uipc_syscalls.c	Socket API system call
getpeername()	uipc_syscalls.c	Socket API system call
sendfile()	uipc_syscalls.c	Socket API system call
soclose()	uipc_socket.c	Socket layer function
sosetopt()	uipc_socket.c	Socket layer function
sogetopt()	uipc_socket.c	Socket layer function
sonewconn()	uipc_socket2.c	Socket layer function
soo_read()	sys_socket.c	Socket descriptor system call
soo_write()	sys_socket.c	Socket descriptor system call
soo_ioctl()	sys_socket.c	Socket descriptor system call
soo_poll()	sys_socket.c	Socket descriptor system call
soo_stat()	sys_socket.c	Socket descriptor system call

sockets-related system call or function listed in Table 2.1.

2.2 Controlling Shim Use

To be of practical use to typical users, 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 feature may be desirable for some applications but not for others. To encourage users to experiment with and hopefully adopt the shim when it could be practical, we have designed and implemented a system to control the use and behavior of the shim. This

section describes the control mechanisms that are in place to decide which applications should use the shim, and under what circumstances. Section 2.9 describes the configuration parameters available to tailor and enhance shim performance.

2.2.1 Global Default Policies

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 [15]. 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 recompilation of the kernel. We take the same approach with many of the shim's configuration variables.

The primary means of controlling the TCP-to-SCTP translation shim is through a global on/off switch for the entire system. This approach offers only a crude level of control over when to use the shim. To enhance an administrator's flexibility, we divide 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 of applications. The *sysctls*, shown below, are boolean variables where zero specifies the shim is disabled, and any nonzero value indicates the shim is to be enabled for that class of applications.

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

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 advanced rule-based shim control system. To provide more precise control over application use of the shim, we extended the system to allow an administrator to selectively enable or disable the shim on a per-application basis using rules.

2.2.2 Shim Use Rules

The basic building block of the shim control system is a *rule* object. A rule consists of an IP address, a subnet mask, and two port numbers, as 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;
};
```

These fields allow a rule to represent one of the following atomic units:

- A specific IP address (*address*)
- A specific network (*address* + *netmask*)
- A single port (*port1*)
- A contiguous range of ports (*port1* + *port2*)

Since a rule only has enough storage for one address and one netmask, a rule can be used to represent a single address or all addresses on a certain network, but not both at the same time. Similarly, a rule only has enough storage for 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 also represent several combinations of address/network and port information:

- An IP address and single port (*address* + *port1*)

- An IP address and a port range (*address* + *port1* + *port2*)
- A network and single port (*address* + *netmask* + *port1*)
- A network and a port range (*address* + *netmask* + *port1* + *port2*)

The combination of address, network, and port number(s) in use by a particular rule 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 that the rule applies to one of the broad classes of applications (i.e., clients or servers) described in Section 2.2.1. The policy is either *enable* or *disable*, reflecting whether or not the shim should be used for applications that 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.

2.2.3 Shim Rules Organization

All shim rules currently in effect are grouped logically into chains of similar rules based on the *chain* and *policy* fields. Recall that the chain field defines whether a rule applies to applications with *local* listening sockets (i.e., servers) or applications which connect to *remote* peers (i.e., clients). The chains of local and remote rules are further subdivided based on a rule's policy, resulting in four separate chains or classes of rules: *local-enable, local-disable, remote-enable,* and *remote-disable*. The reason for subdividing all of the rules into these four particular chains is to enable 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*.

2.2.4 Operation of 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 two 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 the lifetime of a socket before any connection exists or data transfer occurs, so they are logical decision points to enable or disable the shim.

The decision process to enable or disable the shim begins when a client application calls `connect()`, or a server application calls `bind()`. At these points, the kernel checks the value of the global default shim policy for the class of the application, either a client (governed by the `default_remote_enable` *sysctl*) or a server (governed by the `default_local_enable` *sysctl*). One of the four chains in the shim rules table is searched depending on whether the governing default policy is to enable or disable the shim. In the case of a `connect()` call, the kernel first checks the `default_remote_enable` variable. As an example, if the remote default policy is set to disable, the shim is disabled by default for client applications connecting to remote servers. Based on this default policy, the kernel searches the *remote-enable* chain in the shim rules table, determining if the address and port number passed by the application to the `connect()` call match a rule in the chain that overrides the default policy, specifically enabling the shim for the application. For a server application making a `bind()` call, the kernel checks the value of the `default_local_enable` *sysctl* variable. Supposing that the local default policy is set to enable, the kernel would then search the *local-disable* chain of the shim rules table. If the local address and port the application wishes to bind to match with one of the rules found

in the *local-disable* chain, then the kernel overrides the default, specifically disabling use of the shim for the application. The other two situations not described that use the *remote-disable* and *local-enable* chains follow along the same lines as the two examples we have illustrated.

The search process in the shim rules table returns the policy of the *first* rule 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, suppose the chain of the rules table 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 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.

2.2.5 User-Kernel Interface for Shim Rules Table

Configuration of the shim rules table is accomplished using the `shimrules` userspace tool we designed and implemented. The tool provides a command line interface to allow a system administrator to perform rule configuration tasks, such as adding and deleting rules, setting the default policies, flushing the rules table, and getting a listing of all the rules currently in place on the system. Usage information for the `shimrules` configuration program is shown in Figure 2.6. The commands and rule specification options allowed by the `shimrules` tool correspond to the types of rules supported by the rules table implementation in the kernel.

The interface between user space and the kernel is through the addition of new `shimrules` socket options. The new options are implemented in the socket layer and support all of the `shimrules` functionalities. The way the user space `shimrules` program uses the socket options to interact with the kernel is similar to the design of the FreeBSD

```
Usage: shimrules -[AD] -c <chain> -p <policy> <rule specification>
       shimrules -L [-c <chain>] [-p <policy>]
       shimrules -[FH]
       shimrules -G -c <chain>
       shimrules -S -c <chain> -p <policy>
       Commands:
       -A Add rule
       -D Delete rule
       -L List rules
       -F Flush all rules
       -G Get default policy
       -S Set default policy
       -H Show usage information
       Chains: local, remote
       Policies: enable, disable
       Rule Specifications:
       -h <address> [-s <port> | -r <port1>:<port2>]
       -n <address>/<netmask> [-s <port> | -r <port1>:<port2>]
       -s <port> [-h <address> | -n <address>/<netmask>]
       -r <port1>:<port2> [-h <address> | -n <address>/<netmask>]
       Specification parameters:
       -h Host IP address
       -n Network address and netmask
       -s Single port number
       -r Range of port numbers
```

Figure 2.6: Usage information for `shimrules` tool

`ipfw` firewall configuration tool [16]. The options to support the `shimrules` tool are implemented as part of the `sogetopt()` and `sosetopt()` socket layer functions as shown in Table 2.2. The `shimrules` socket options, which update the state of the kernel's shim rules table, are implemented in `sosetopt()`, and include the options to add rules, remove rules, flush all rules, and set the system global default policies. The read-only `shimrules` socket options are implemented in `sogetopt()` and include the options to get a listing of all rules currently in the shim rules table, and to get the current default global policies.

Applications use the `setsockopt()` and `getsockopt()` system calls, which in turn invoke the socket option implementations in the lower level `sosetopt()` and `sogetopt()` functions, respectively. The parameters and use of each of the `shimrules` socket options for `setsockopt()` and `getsockopt()` are as follows:

- **SHIM_RULE_ADD & SHIM_RULE_DEL:** The two options to add and delete a rule both take an argument of type `shim_rule`, the format of which was shown earlier. An application specifies the *chain* and *policy* parameters that the kernel uses to classify the rule, and place it into the proper chain inside the shim rules table as described in Section 2.2.3. Once in the correct chain, the address, network, and port information of the rule is used for matching application network usage as explained in Section 2.2.4. These socket options are accessed by the application using the `setsockopt()` system call.

- **SHIM_RULE_FLUSHALL:** The *FLUSHALL* socket option takes a single integer as an argument. The integer argument is used as a confirmation value. If the value is non-zero, the entire rules table is flushed, otherwise no action is taken. Applications access this option using the `setsockopt()` system call.

- **SHIM_RULE_DEFAULT:** The *DEFAULT* socket option is used in conjunction with the `getsockopt()` and `setsockopt()` system calls to get and set the global default policies for the *local* and *remote* chains of the shim rules table. The argument to the *DEFAULT* option is a `shim_default_policy` object, the format of which is shown below.

  ```
  struct shim_default_policy {
     int chain;
     int policy;
  };
  ```

  When setting a policy, an application specifies both the *chain* and *policy* for the kernel to set for that chain. When getting the current default policy, an application

Table 2.2: New `shimrules` socket options

Socket Option	System Call Used	Implementation Function
SO_SHIM_RULE_ADD	setsockopt()	sosetopt()
SO_SHIM_RULE_DEL	setsockopt()	sosetopt()
SO_SHIM_RULE_FLUSHALL	setsockopt()	sosetopt()
SO_SHIM_RULE_DEFAULT	{set/get}sockopt()	so{set/get}opt()
SO_SHIM_RULE_GETALL	getsockopt()	sogetopt()

specifies only the chain for which the policy should be retrieved, and the kernel fills in the policy for the requested chain, returning the whole default policy object to the application.

- **SHIM_RULE_GETALL:** The *GETALL* option is used by applications to retrieve a listing of all current rules in every chain of the shim's rules table. An application makes a call to `getsockopt()`, specifying a buffer large enough to hold all of the rules. The kernel fills the buffer space with a listing of all the rules, and then returns the buffer to the application, updating the size to reflect the total amount of the buffer used for rule storage. In the event the buffer passed in by the application is too small to hold all of the rules, the kernel returns an error. In response, the application can begin increasing the buffer size until the call returns successfully. Once the application has access to all the rules in the buffer, the rules can be accessed or displayed by the application as desired.

2.3 Shim States

To support the hidden socket first introduced in Section 2.1.4, 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 2.2.4 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. 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 because using the shim-enabling socket option requires knowledge of the existence of the shim, 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 for the application, and use of the shim should be attempted for any interaction with peer endpoints.

- **SHIM_ENABLE_SRULES:** When the shim has been enabled for a particular application as determined by the rules table lookup process described in Section 2.2.4, 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 for the application, and shim use should be attempted 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. This state indicates to the kernel that the application is a client which has successfully connected to a remote server using SCTP, and the shim is in active use for communications between the client its peer. Because of the *ACTIVE* state set in the parent TCP socket, the kernel knows to use the hidden SCTP socket for all network interaction instead of the normal TCP socket.

- **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 in the 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*, then the kernel activates the shim for 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 each and every hidden SCTP socket. This state allows the kernel to identify if a particular socket is a hidden SCTP socket and not 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 2.2.

- **SHIM_NATIVE:** Each of the shim states introduced so far applies to either a normal parent TCP socket (*NOTINIT*, *READY*, *ACTIVE*, *ENABLE_MANUAL*, *ENABLE_SRULES*, and *LISTEN*) or a hidden SCTP socket (*HIDDEN*). The common thread among all of the states introduced for the parent and hidden sockets is that those sockets are created directly by the application. The *NATIVE* state applies to *new sockets created by the kernel* to represent the communications endpoint for remote peers connecting to the local listening server. When a TCP server application uses the shim, SCTP peers can connect and communicate with the server. However, the sockets created for each connection 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 2.5, the importance of the *SHIM_NATIVE* state is that it allows the kernel to distinguish a native SCTP socket used by the shim from a hidden SCTP socket used by the shim. The kernel operates differently on these two types of sockets, thus the importance in keeping them properly classified.

Figure 2.7 shows the typical transitions between the shim states for a client and server application. Note the *SHIM_HIDDEN* and *SHIM_NATIVE* states are immutable. Any socket with one of those states remains in that state for the lifetime of the socket.

2.4 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 2.2.4, the kernel searches the appropriate chain of the shim rules table to decide if the shim should be enabled or disabled for the application making the call to `connect()`. If the shim is disabled, the normal TCP connection

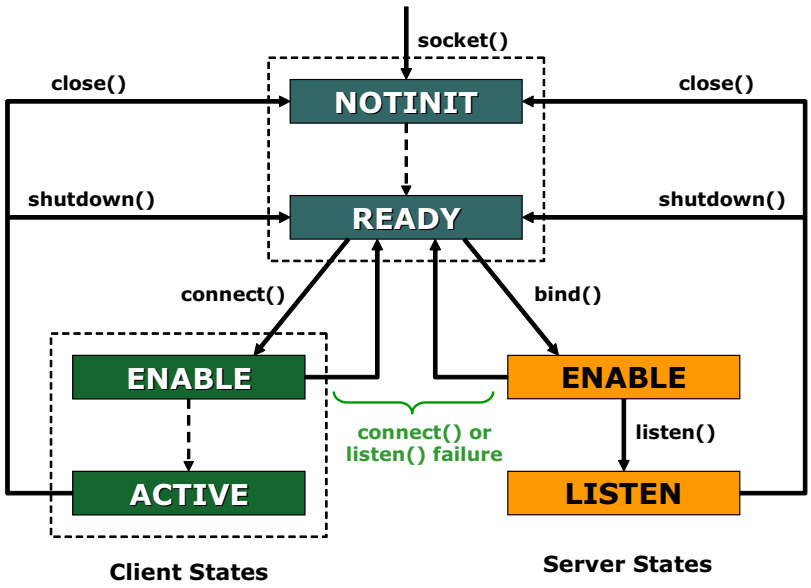

Figure 2.7: Shim states and typical transitions for client and server applications

process continues without modification. However, if the result of the rules table lookup is that the shim is to be enabled for the application calling `connect()` and the normal socket is in the *SHIM_READY* state, a different sequence of events takes place. Rather than initiating the establishment of 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. In the event the server running at that remote address and port also supports SCTP (over the shim or natively), an SCTP association will be set up and all communications between the two endpoints will be over SCTP. Once the peers are associated successfully using SCTP, the shim state for the normal TCP socket is changed to *SHIM_ACTIVE* to signify that 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 for some reason the SCTP association fails to be established, the kernel falls back to a regular TCP connection. In such a case the socket's shim state will remain *SHIM_READY*, indicating the hidden SCTP socket is

available but currently inactive.

Before association establishment is attempted using the shim, several of the 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 the application may make to the normal TCP socket before the shim becomes active. Having identical configurations guarantees 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 and low water marks for send and receive socket buffers
- Socket buffer asynchronous I/O flags
- Socket accept filter

To control how long the kernel attempts to establish the SCTP association before falling back to a normal TCP connection, we added the `shim.init_rtx_max` *sysctl* described in Section 2.9.2. The `connect()` system call sets this parameter using an SCTP socket option before the SCTP association establishment is attempted. Also before initiating the association establishment, `connect()` sets another configuration parameter based

on the `shim.path_rtx_max` *sysctl* for the association being established. This value is used to configure path failover thresholds and is described in Section 2.9.3.

2.5 Server Socket Functionality

Unlike the functionality of the TCP-to-SCTP translation shim for a client application, which initially tries to establish communications with the peer endpoint and fails back to normal TCP if SCTP is unavailable, the server application functionality of the shim is a hybrid approach that allows a single instance of a server process to serve both TCP *and* SCTP clients concurrently. At a high level, the sequence of events for a server application on a system supporting the shim is as follows: First, an application makes the `bind()` system call. Inside `bind()`, the kernel decides if the shim should be enabled or disabled and changes the shim state for the application accordingly before actually performing the binding of the SCTP (if the shim is enabled) and TCP sockets. When the application calls `listen()` next, the sockets which were bound during the `bind()` call are then enabled for listening, and the server is ready to accept and serve SCTP (if the shim is enabled) and TCP clients. We describe the changes to the `bind()` and `listen()` system calls in Sections 2.5.1 and 2.5.2, and then explain how the shim modifies the `sonewconn()` function to allow the server to handle connecting TCP and SCTP clients in Section 2.5.3.

2.5.1 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 determine whether to enable or disable the shim for the application using the process described in Section 2.2.4. 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 fail back to only serving TCP clients. After binding the hidden SCTP

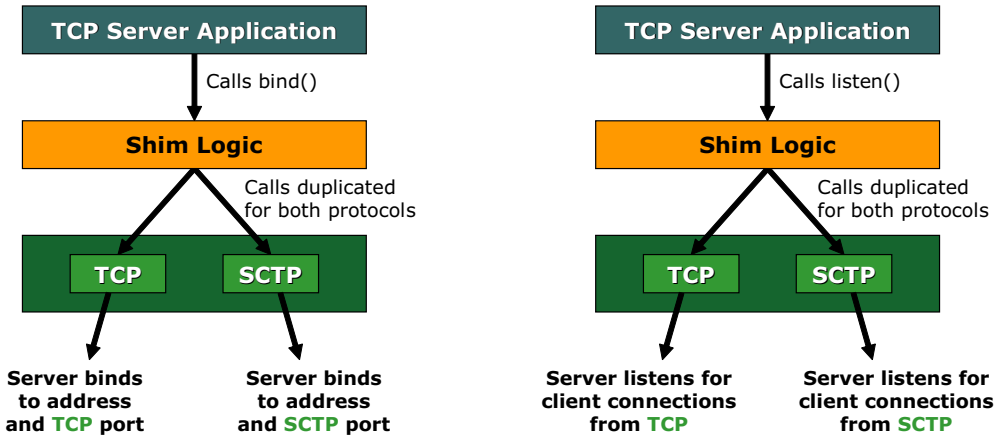

Figure 2.8: Duplication of `bind()` and `listen()` calls for server applications

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 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. Figure 2.8 illustrates the duplication of the `bind()` call for the normal TCP and hidden SCTP sockets.

As an extension to the shim binding process with the SCTP socket, a configuration option is implemented to force `bind()` to override the application, binding the SCTP socket to all possible addresses on a multihomed system, rather than just the single address passed by the application to `bind()`. This option, described in Section 2.9.1, is designed to improve the chances that multihoming provides fault tolerance and the possibility of increased throughout (with CMT) to the application.

2.5.2 Listen

After calling `bind()`, the application next calls `listen()` to put the server's hidden SCTP (if the shim is enabled) and normal TCP sockets into the listening state, allowing clients to begin connecting. Figure 2.8 shows the duplication of the `listen()` call

for the two sockets. If the shim is enabled, `listen()` also changes the shim state for the application to *SHIM_LISTEN* as described in Section 2.3, indicating that the application 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 sets them on the hidden SCTP socket. This cloning operation allows an application to create a socket and change several configuration parameters before the shim is actually enabled by the rules table lookup in the `bind()` call. If these parameters are set before enabling the shim, the shim will not receive the same updated configuration that the normal TCP socket receives, resulting in inconsistency between how clients of the two protocols are handled. The cloning step ensures that both sockets have the same configuration at the time the shim enters the *SHIM_LISTEN* state and begins serving clients. The cloned parameters include the following:

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

Once the SCTP and TCP sockets are bound and listening, the 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 may begin serving clients that connect via either TCP or SCTP. We describe how the connecting clients are made accessible to the server application in Section 2.5.3.

2.5.3 Servicing Connecting Peers

Remote peer endpoints initiate communication with a TCP or SCTP server application by sending SYN or INIT messages, 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 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 behavior.

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 are normally 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 into the waiting lists in the corresponding listening sockets. If a newly created socket is going to be inserted into the hidden SCTP socket's waiting list,

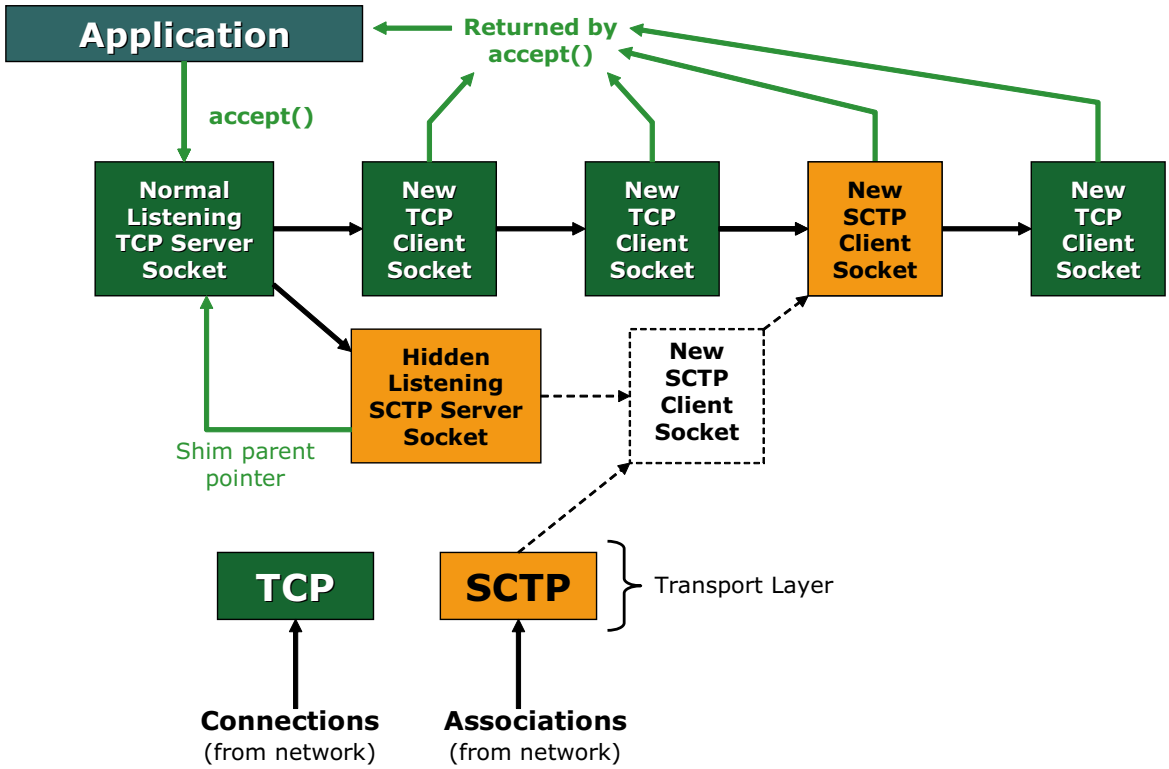

Figure 2.9: Shim-enabled server architecture overview

`sonewconn()` instead follows the hidden SCTP socket's *parent pointer* (introduced in Section 2.1.4) and *queues the new socket into the normal TCP listening socket's queue*. 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 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 2.9 illustrates the architecture of the TCP and SCTP listening sockets and how newly created sockets are maintained in a single list.

2.6 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 rest of the operating system. The first set consists of the specifically designed network I/O calls, such as `send()`, `recv()`, `sendto()`, `recvfrom()`, `sendmsg()`, and `recvmsg()`. The second set consists of the standard UNIX I/O calls, implemented especially for use with sockets. The second set of functions is required because a socket descriptor is allowed to be used by an application in a similar fashion to a true file descriptor, thus applications need to support the standard `read()`, `write()`, `ioctl()`, `poll()`, and `stat()` system calls normally associated with file I/O. We describe the shim modifications required for the two classes in Sections 2.6.1 and 2.6.2.

2.6.1 Socket Operations: Send, Receive, & Sendfile

The six network I/O system calls of the sockets API are `send()`, `recv()`, `sendto()`, `recvfrom()`, `sendmsg()`, and `recvmsg()`. All of the sending (receiving) functions are similar, differing in the number of parameters that can be specified by an application at the time the I/O call is made. For example, `send()` allows an application to specify

only a buffer of data to send and a set of message flags. The `sendto()` function extends `send()`'s feature set by additionally allowing the application to specify a destination address. The `sendmsg()` function further extends `sendto()` by allowing an application to additionally specify ancillary data (control information) using a message header structure that is passed into the kernel [13]. The relationships between the three receiving system calls are identical to the relationships between their respective send calls [12].

Inside the kernel, all six of the send and receive system calls are built upon two fundamental functions, `sendit()` and `recvit()`, which implement the core sending and receiving behavior, respectively. The visible socket I/O calls build on the two base functions by adding additional processing to handle the increasing level of features supported by the `send()`, `sendto()`, and `sendmsg()`, or `recv()`, `recvfrom()`, and `recvmsg()` system calls. Consequently, adding support to the shim for network I/O requires modification to only `sendit()` and `recvit()`. Unlike the system calls that require significant additional code to support the desired shim features (i.e., `connect()`), the network I/O functions require only the socket substitution described in Section 2.1.6 to operate properly. The excerpt below illustrates the code to substitute the hidden SCTP socket in place of the normal TCP socket that is necessary to support the network I/O functions with the shim.

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

The `sendfile()` system call is used by applications to simplify the efficient transfer of files through a socket. As with `sendit()` and `recvit()`, substitution of the hidden SCTP socket in place of the normal TCP socket is the only functionality required to add support to the shim for this system call.

2.6.2 File Operations: Read, Write, Ioctl, Poll, & Stat

A socket descriptor can be used as a normal file descriptor by applications performing I/O. Thus, the standard `read()`, `write()`, `ioctl()`, `poll()`, and `stat()` system calls typically associated with file I/O must also work with a socket descriptor. To support this 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 normally associated with a traditional file. The socket-specific versions of the standard `read()`, `write()`, `ioctl()`, `poll()`, and `stat()` 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 in Section 2.6.1, the standard file I/O system calls for sockets can be modified to support the shim by adding code to perform the substitution of the hidden SCTP socket in place of the normal TCP socket when the shim is in use. No further modifications are required for these functions to support the shim functionality.

2.7 Close & Shutdown

One traditional file I/O system call that has not been modified to do hidden socket substitution similar to the other calls is `close()`. Recall from Section 2.1.4 that the `socket()` system call is modified by the shim to create both a normal TCP socket and a hidden SCTP socket together. Substituting the hidden SCTP socket in place of the normal TCP socket when calling close would result in only the hidden socket being deallocated; the resources from the normal socket would not be reclaimed and a memory leak would occur in the kernel. Rather than substituting one socket object for the other, `close()` must check the normal TCP socket to determine if it has a hidden child SCTP socket, deallocating that hidden socket before deallocating the normal TCP socket. The lack of any need for socket substitution when closing the socket descriptor results in the `soo_close()` function remaining unmodified. All of the modifications necessary for

proper deallocation of both the hidden and normal sockets are made in the `soclose()` socket layer function.

```
if(SHIM_READY(so)) {
   soclose(so->so_shimsock);
   so->so_shimsock = NULL;
   so->so_shimstate = SS_SHIM_NOTINIT;
}
```

The listing above shows the code added to the `soclose()` socket layer function to handle proper deallocation of the hidden SCTP shim socket. The kernel first checks the shim state of the parent socket. If the shim is in the *SHIM_READY* state, a hidden SCTP socket was previously allocated, and must be closed and deallocated (see the description of shim states in Section 2.3). After deallocation of the hidden socket, the shim state is set to *SHIM_NOTINIT* to indicate that no hidden SCTP socket is linked to the normal TCP parent socket. Since the operating system ensures that `close()` is called on every open file and socket descriptor when an application exits, `soclose()` runs for each and every socket, guaranteeing the proper deallocation of all shim resources.

The `shutdown()` system call is similar to `close()` but is seldom used in most applications. Unlike the `close()` system call, which closes the network connection and then deallocates the socket descriptor, `shutdown()` only closes one or both directions of the open full-duplex connection and leaves the socket descriptor allocated. In the case of the shim, calls to `shutdown()` while the shim is enabled are handled identically to the other usual I/O calls by passing the hidden SCTP socket to the lower layers rather than the normal TCP socket. This behavior results in the ongoing SCTP association being terminated. After an application closes a network connection with `shutdown()`, the application must still use `close()` to deallocate the socket descriptor as usual.

2.8 Socket Options & Socket Addresses

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 and socket address-related system calls 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 shim's handling of socket options and socket address operations.

2.8.1 Socket Options

Socket options are protocol parameters maintained at different levels in the Internet protocol stack that are exposed to the application. The options are represented as name-value pairs where the value might be as simple as a boolean on/off setting or a more complex data structure. Through the sockets API, an application may request the current value of a certain option, and in some cases, the application may also specify a new value to be used instead. An application gets and sets these socket options using two system calls, `getsockopt()` and `setsockopt()`, respectively.

```
if(getsockopt(sock_fd, SOL_SOCKET, SO_ERROR, &value, &len) == -1) {
   perror("getsockopt");
   exit(1);
}
```

The sample code above illustrates how an application uses the `getsockopt()` system call with a typical socket option. In the example, the first parameter, *SOL_SOCKET*, is the *level* of the option, the second parameter, *SO_ERROR*, is the *name* of the option, and the last two parameters specify the value and size of the value, respectively.

2.8.2 Socket Option Levels

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 using the appropriate system call, each level of the protocol stack examines the level parameter of the specified option, checking for a match. If the protocol finds it matches the level of the option, the protocol module then checks the name of the option and handles the option appropriately. If the protocol is not the responsible level, the option is passed down the protocol stack to be handled at the correct level.

2.8.3 Translating Socket Options

When the TCP-to-SCTP translation 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 TCP-to-SCTP translation shim. In particular, most TCP implementations have only two 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 were in the enabled state and special precautions were not taken, these options might be invoked by a legacy TCP application. However, with the shim enabled, TCP is not in the stack of protocols when the socket option is passed down the stack. The option would travel from the socket layer, to SCTP, to IP, and then

down through the link layer. At no point along this line would TCP be able to intercept the option and handle it properly. Eventually the option would pass down 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.

```
if(translate_options) {
   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;
   }
}
```

2.8.4 Socket Addresses

An application can use the `getpeername()` and `getsockname()` system calls to retrieve the socket address of the remote endpoint and the local endpoint of a connection, respectively. (A socket address is a structure consisting primarily of the address and port number.) Socket addresses are maintained by the transport protocols when connected, so to support the TCP-to-SCTP translation shim, the kernel needs to determine if either the normal TCP socket or the hidden SCTP socket is currently the active endpoint. The proper active socket is then passed to the appropriate transport protocol for the retrieval of the address and port information, which is then returned to the application.

2.9 Shim Performance Configuration

Application performance when running over the TCP-to-SCTP translation shim depends on SCTP, and consequently, on SCTP's configuration. To allow key SCTP settings for the shim to be configured independently from those for standard native SCTP applications, we created additional *sysctl* variables to allow an administrator to fine-tune the shim's key SCTP settings. The three areas of focus are address use on multihomed systems, association startup time, and path failover time.

2.9.1 Controlling Address Use on Multihomed Systems

A typical legacy TCP application uses the `bind()` system call to associate a socket with a local address and port number. Because TCP allows binding to only a single address, a traditional TCP application creates a socket address structure with the desired address and port number and passes the structure to the kernel via `bind()`. Passing a specific address presents a problem when the shim is in use, since the application's call to `bind()` is actually being made on the hidden SCTP socket rather than a TCP socket. The semantics of the `bind()` call in SCTP are such that the application will use only the specified address and leave all other addresses on a multihomed system unused. While a native SCTP application can use the `sctp_bindx()` call to bind to specify multiple address to bind to, a legacy TCP application cannot. Binding to a single address by using a normal `bind()` call overrides the natural multihoming that takes place on a multihomed system using SCTP by artificially restricting the possible set of address which can be used.

As a counter to applications running over the TCP-to-SCTP shim exhibiting this type of behavior, we have introduced a new *sysctl*, `net.inet.sctp.shim.force_bindall`, which allows an administrator to force a shim-enabled kernel to ignore the address passed by the application to `bind()`. When `force_bindall` is enabled, instead of binding a socket to the single address passed in by a legacy TCP application, the kernel overrides

the application and uses every available address for the local SCTP endpoint. The override is accomplished by substituting the wildcard address *INADDR_ANY* in place of the original address passed by the application. (A `bind()` call on an SCTP socket using *INADDR_ANY* effectively binds to all possible local addresses.) Allowing the endpoint to use all available addresses increases the likelihood that multihoming can be exploited to increase the fault tolerance of an application using the shim. Multihoming also allows for the possibility that CMT could be used between the two association endpoints to improve application throughput.

2.9.2 Controlling Association Startup Time

SCTP normally retransmits an *INIT* message up to *Max.Init.Retransmits* times (implemented as the `net.inet.sctp.init_rtx_max` *sysctl* in FreeBSD) when attempting to establish an association with a peer endpoint. The SCTP Implementor's Guide [21] currently recommends that this value be set to a default of 8. With exponential backoff, the time required to retransmit the INIT message 8 times is approximately 4 minutes ($1+2+4+8+16+32+64+128$ seconds). As discussed in Section 2.4, when the transparent TCP-to-SCTP translation shim is enabled, the kernel first attempts to set up an SCTP association with the remote endpoint. If SCTP is unavailable on that endpoint or no server process is listening on the specified port, the kernel will fall back to a traditional TCP connection. To remain transparent to the application, the length of time to decide whether or not SCTP is available must be limited to a reasonable amount that will not significantly impact the application. To avoid waiting over 4 minutes for 8 consecutive retransmissions of the INIT message, we introduce a new *sysctl* variable, `net.inet.sctp.shim.init_rtx_max`, which serves the same purpose as the corresponding global SCTP *sysctl* variable, but only applies to associations initiated by the shim. This variable allows an administrator to restrict the worst-case connection startup delay for applications using the shim to a configurable length of time, while leaving the default value for native SCTP applications unaffected. We currently implement

the `shim.init_rtx_max` variable with a default value of 1, meaning that the kernel will send a total of 2 *INIT* messages before falling back to TCP. This value avoids SCTP being abandoned due to a single random *INIT* loss, while keeping the application wait time short. Administrators should choose a value that balances their interest in using SCTP with the wait times they are willing to tolerate before giving up and resorting to TCP. Section 4.2.1 describes a proposed solution to reduce the time overhead required to detect endpoints where SCTP is unavailable.

2.9.3 Controlling Shim Path Failover Time

SCTP's *Path.Max.Retrans*, or PMR, configuration parameter (implemented as the `net.inet.sctp.path_rtx_max` *sysctl* in FreeBSD) can have a direct effect on the throughput of an application operating in a multihomed environment. The PMR value is a threshold that defines when SCTP considers the primary address of a multihomed peer to be unreachable and consequently initiate a failover to an alternate address. Recent results [2] illustrate how varying the aggressiveness of the PMR threshold affects application throughput.

In addition to raw throughput, the PMR setting affects how responsive an application running over SCTP or the shim "feels" to the user. In the event of a failure of one destination of a multihomed endpoint, retransmission timeouts follow an exponential backoff up until *Path.Max.Retrans* retransmissions have occurred, at which point SCTP will fail over and begin using an alternate destination of the endpoint for data delivery. The SCTP Implementor's Guide currently recommends that PMR default to 5, meaning that approximately 1 minute ($1+2+4+8+16+32$ seconds) of minimal throughput occurs before SCTP fails over to an alternate destination and resumes normal data transfer [3, 21]. In human-interactive applications, such as streaming music, login sessions, or web browsing, this delay could be easily perceived by the user. Allowing SCTP's fault tolerant multihoming to engage more rapidly when using the shim could prevent users from closing and reconnecting the application, as the typical response to such a

failure with a traditional TCP application would be. We introduce a new shim *sysctl*, `net.inet.sctp.shim.path_rtx_max`, to allow administrators to independently define SCTP's PMR value for applications using the shim to allow flexible configuration of this parameter.

2.10 Design Summary

We have designed and implemented a *transparent* TCP-to-SCTP translation shim layer in the FreeBSD 4.10 operating system kernel. Due to limitations in the original design of the sockets model, our approach introduces a *hidden* SCTP socket to allow legacy TCP applications to operate in a hybrid fashion. A hybrid client approach provides a shim user with flexibility, allowing client applications to first attempt to use SCTP, falling back to TCP should SCTP be unavailable, while a hybrid server enables both TCP and SCTP clients to be served concurrently by a single server instance. Our design and implementation features a control system based on rule matching with application address and port usage, allowing the shim to be selectively enabled or disabled on a per-application basis. The shim supports all standard socket API system calls and I/O functionality, including the ability to properly translate TCP socket options into their SCTP equivalents. In our implementation, we have enhanced the kernel with several new *sysctl* variables, giving administrators the ability to dynamically configure the shim's most important parameters to achieve the desired shim behavior. Our shim design gives legacy TCP applications access to many of SCTP's advanced features without requiring any modifications to the applications themselves.

Chapter 3

EXPERIMENTAL EVALUATION

3.1 Applications Running Successfully with Shim

Using the initial implementation of the TCP-to-SCTP translation shim, we experimentally 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 3.1.1 and 3.1.2 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 experiments in more detail in Section 3.3.

3.1.1 Legacy-legacy Configuration

Recall that the shim's legacy-legacy mode (Figure 1.2) 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 SCTP's advanced features without requiring any modifications to the applications themselves. We

selected four types of applications that represent the network usage of a typical Internet user: Telnet, SSH, HTTP, and Icecast [17] 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 [14] 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 TCP-to-SCTP translation shim. Both the Telnet client and server functioned correctly while running over the shim and no errors 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 [11]. Our experiments found that SSH operated over the shim flawlessly 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 3.3. 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 [5] as the web server, and Firefox 1.04 [4] 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 is a streaming audio server that streams music in the Ogg Vorbis [6] format. For our testing, we used the Icecast 2.2.0 streaming audio server and the XMMS [25] 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.

Although the set of applications we experimented with is not exhaustive, we believe the flawless operation of these four applications when using the shim indicates the shim will be a viable and practical tool for almost all existing legacy TCP applications. We are only aware of one potential problem relating to legacy TCP applications that explicitly depend on the behavior of TCP's half-closed state. We discuss this case in Section 3.2.

3.1.2 Legacy-native Configuration

In addition to testing the interaction of two legacy TCP endpoints using the shim, we also experimented with HTTP in the legacy-native configuration (Figure 1.3) using a version of the Apache webserver that was rewritten to *natively* support SCTP clients [20], and the 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. The results of testing HTTP over the shim in legacy-native mode were identical to the results from the legacy-legacy mode test; the web pages were downloaded and rendered flawlessly. The success of this test validates the gradual migration path motivation of the TCP-to-SCTP translation shim project.

3.2 Shim Limitation: Lack of Half-Closed State

In our experimentation with various legacy TCP applications operating over the TCP-to-SCTP translation shim, we found most applications functioned normally without any detectable problems. One notable exception to this rule are applications which explicitly depend on the semantics of TCP's half-closed connection ability. We discovered the half-closed problem while testing a version of FTP rewritten to run over SCTP. Although the problem is actually caused by differences between how TCP and SCTP handle the closing of a connection and not a design flaw in the shim itself, the problem should be addressed to allow the shim to experience the widest real world deployment.

3.2.1 TCP & SCTP Connection Closing

Closing a TCP connection involves a four-way handshake between the two connected peers. One endpoint first sends a FIN packet to its peer, indicating the desire to close the the connection. The endpoint receiving the FIN packet responds with an ACK, indicating that the FIN was received. At this point in the process, the connection is half closed. The TCP connection remains in the half-closed state until the remaining active endpoint sends its own FIN packet to fully close the the connection. By design, SCTP does not have a half-closed state. When one endpoint in an established SCTP association sends a SHUTDOWN packet, the peer receiving the shutdown must immediately stop accepting new data from the application. Once any data already accepted for transmission before the SHUTDOWN was received has been delivered, the receiver of the SHUTDOWN packet replies with a SHUTDOWN-ACK packet. Once the sender of the SHUTDOWN receives the SHUTDOWN-ACK, the SHUTDOWN sender replies with a SHUTDOWN-COMPLETE packet and the connection closes. The exact sequence of events followed by TCP and SCTP when closing a connection or association is not the focus, however. The important distinction between how the two close methods work is *in TCP both endpoints must agree to close before the connection is torn down*. With SCTP, *one endpoint can unilaterally decide to close the association* and the other peer has no

alternative but to comply and begin shutdown procedures. We describe how the differences in close semantics between TCP and SCTP can affect legacy application behavior in Section 3.2.2, using FTP as an example.

3.2.2 Effect of Half-Close on Shim Operation

To illustrate how the lack of a half-closed state in SCTP can cause problems for some applications using the shim, we show an example consisting of one segment of an FTP file transfer over the shim. Using the shim, the FTP control connection uses one association, and each data connection uses a separate SCTP association. (Although the shim cannot use two SCTP streams within a single association for control and data because streams require *application awareness*, work in [10] investigates this idea.) Figure 3.1 shows a timing diagram for a file transfer over FTP in normal, non-passive mode. (Solid lines represent exchanges over the control connection, dotted lines represent data connection exchanges.) In non-passive FTP, the client creates a listening socket and sends the address and port number of the listening socket to the FTP server with a PORT command. After receiving a 200 message reply from the server, the client issues a RETR command for the file to be retrieved. The FTP server then establishes an SCTP association for the file transfer (the data connection) by connecting to the client's specified address and port number, and sends a 150 message after the connection is open. Upon receiving the 150 message, the client calls `accept()` to handle the server's incoming data connection and begins reading as the server sends the file data. Once the file transfer is complete, the server closes the data connection and sends a 226 message.

While the shim would operate correctly in the case just described, now consider what would happen if the 150 message were to be lost and retransmitted, as outlined in Figure 3.2. Since the client does not call `accept()` until receiving the 150 message from the server, the data connection is set up, the file contents are transferred, and the server closes the connection *all before the client even has a chance to read the data*. (The operating system kernel is responsible for handling all of the connection setup and can

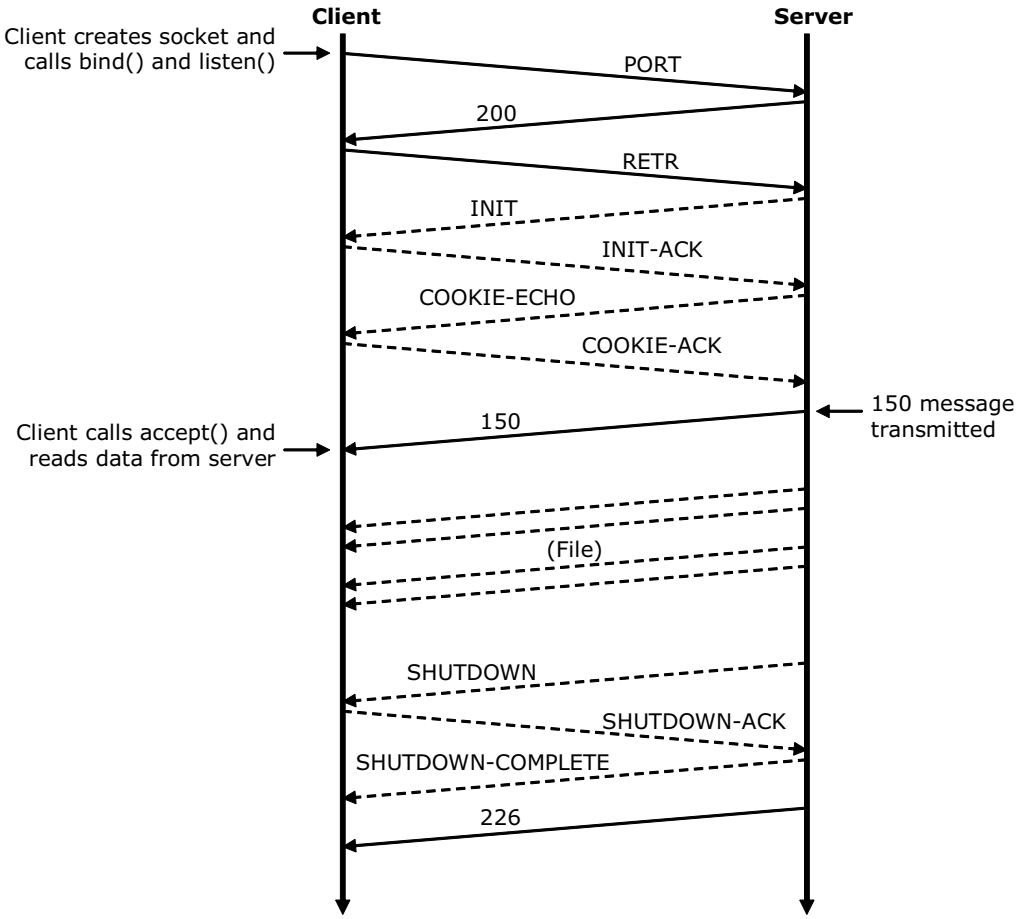

Figure 3.1: FTP file transfer over shim (SCTP)

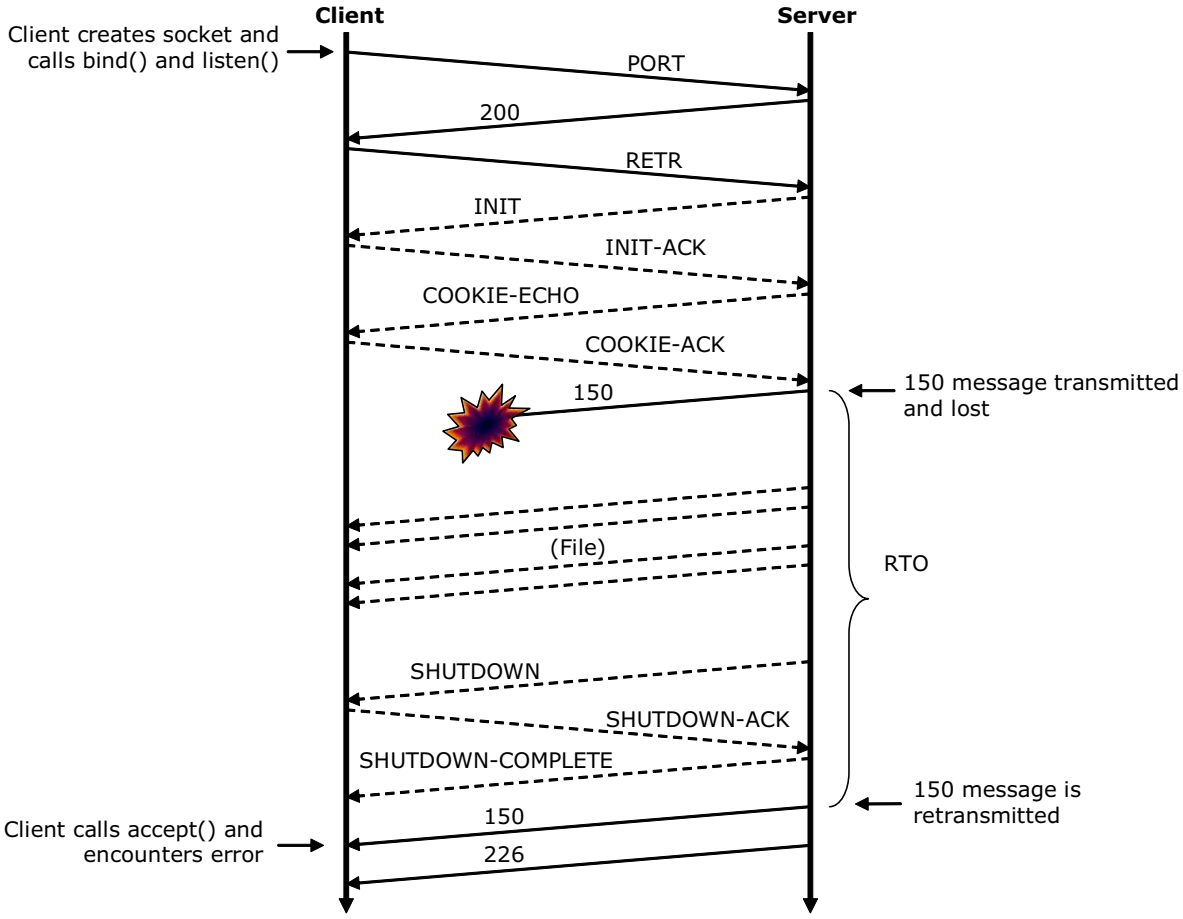

Figure 3.2: FTP file transfer over shim (SCTP) with loss of 150 message

receive up to a window of data before the application calls `accept()`). After the 150 message is retransmitted and finally received, the client tries to call `accept()`, but the data connection has already been closed, causing an error. (Note: this situation arises when the time to transfer the file is less than the retransmission time for the lost 150 message, which is common for small files on broadband links.)

Consider the same situation where the original 150 message is lost when FTP is running over a normal TCP connection rather than the shim, pictured in Figure 3.3. Because TCP supports a half-closed state, when the server issues a `close()` command

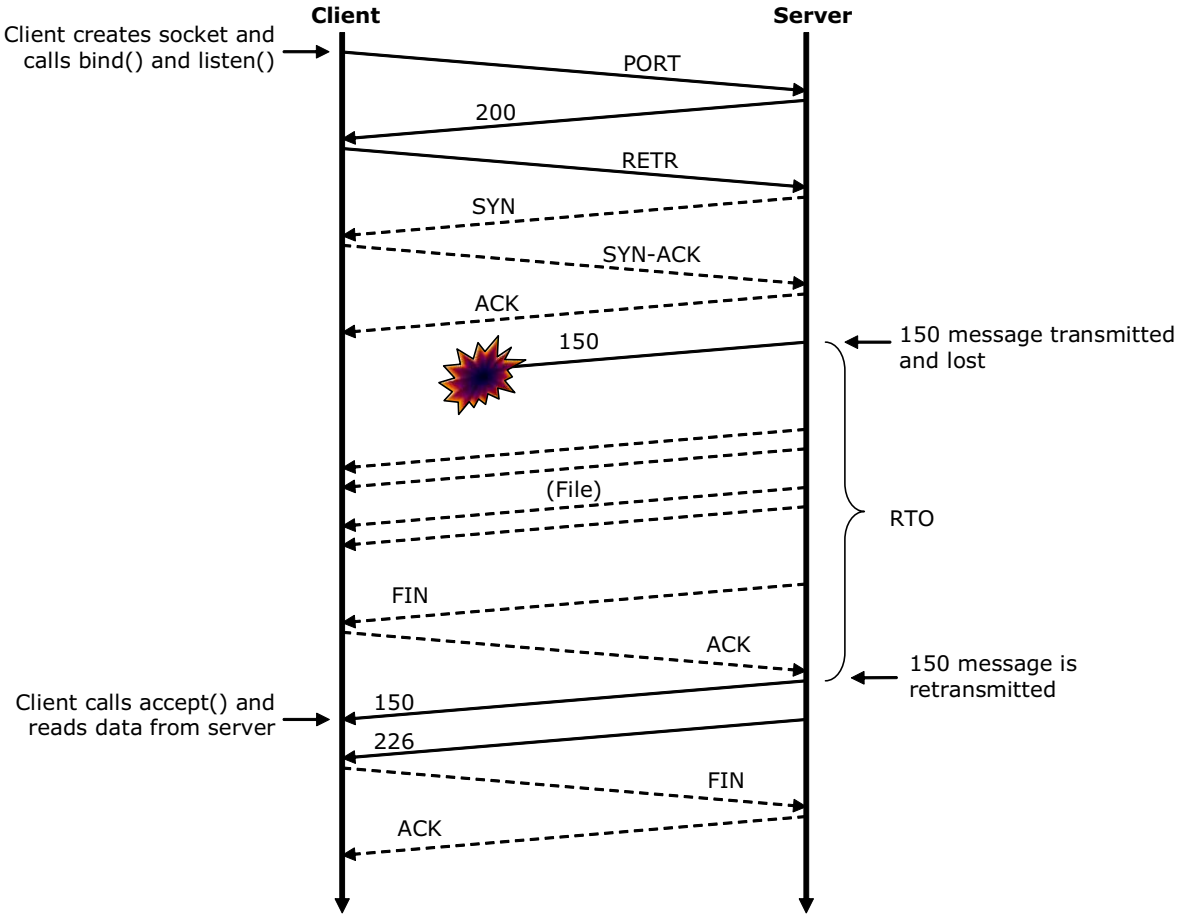

Figure 3.3: FTP file transfer over TCP with loss of 150 message

after transferring the file contents, *only half of the connection is closed*. For the connection to be completely closed, the client also has to issue a `close()` command. Because one endpoint cannot unilaterally close a connection completely in TCP, the client is able to call `accept()` and then read the file data before eventually calling `close()` itself. Only after the client calls `close()` will the TCP connection be completely closed.

Due to the differences between how TCP and SCTP close connections, applications running over the shim that *depend on the half-closed state* may sometimes experience unexpected behavior or errors when situations similar to those described with FTP

arise in practice. In an effort to make the shim robust against incompatibilities arising from SCTP's lack of a half-closed state, we propose a possible solution to allow the shim to simulate TCP's half close, which we describe in Section 4.2.3.

3.3 Performance Analysis

The experiments described in Sections 3.1.1 and 3.1.2 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 quantitatively show how the performance of application interaction when using the shim compares to the performance when applications use normal TCP connections. In Section 3.3.1, we describe the setup of our experimental evaluation of shim's performance in terms of application throughput during file transfers. Section 3.3.2 discusses the results and conclusions of the performance evaluation.

3.3.1 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/Propagation 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 both the client to the server, and the server to the client.

- **Packet Loss Rates:** We examined transfer time with uniform loss rates of 0, .01, .03, .06, and .1. Similar to the bandwidth-delay configuration, the loss rates are symmetric so the paths from the client to the server and from the server to the 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 running SSH (and consequently SCP), an SCP client, and an intermediate node running *Dummynet* [19] 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 the alternate 100 Mbps, negligible-delay paths between the client and server nodes.

Each run of the experiment involved measuring the total time required to issue the SCP command on the client to retrieve a single file, 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 3.3.2 is the average of 30 (or 90) runs of the same file size/loss rate configuration.

3.3.2 Experimental Results

Figures 3.4, 3.5, 3.6, and 3.7 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 for application use on the peer system can be a source of overhead during the connection establishment process as described in Section 2.9.2; a proposed solution to this situation is described in Section 4.2.1. 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.

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. 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) [10]. Our results from experimenting with SCP over the shim confirm the same trends at high loss rates as the work in [10] 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.

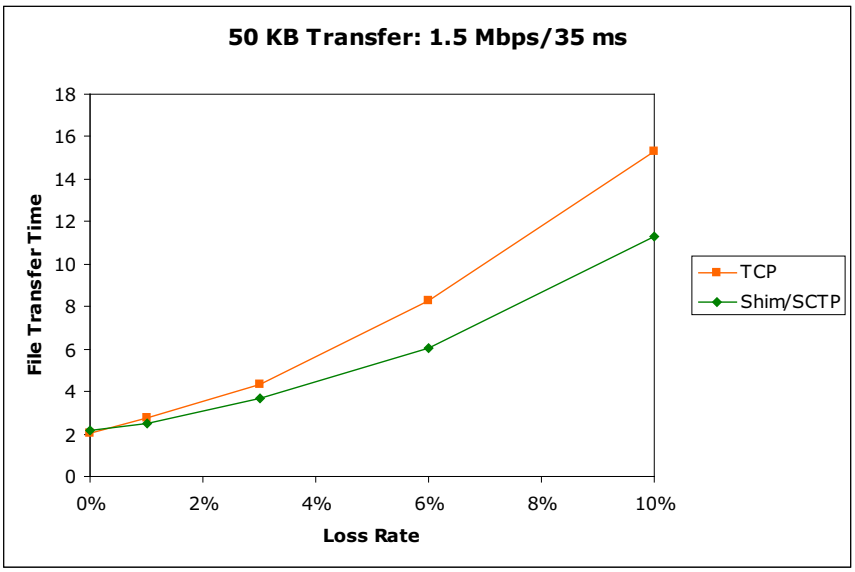

Figure 3.4: Transfer Time vs. Loss Rate for 50 KB file transfer over 1.5 Mbps/35 ms delay link

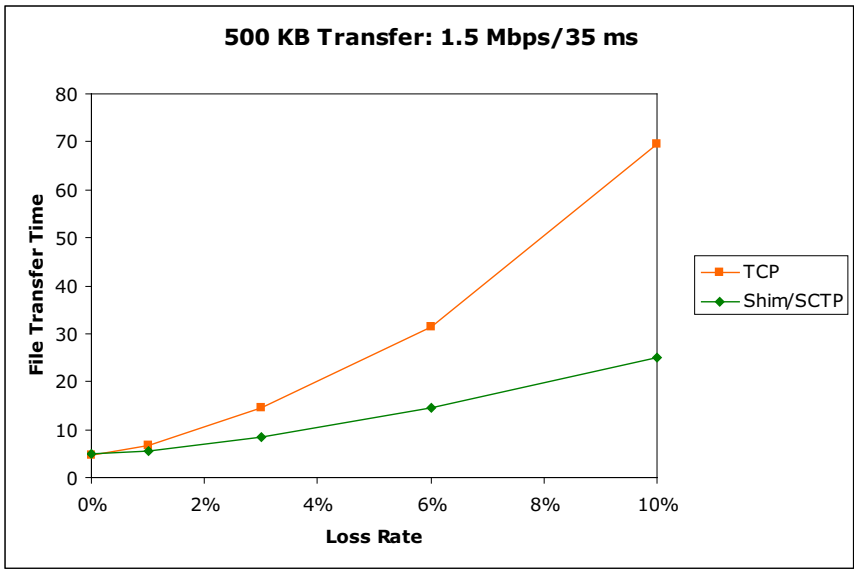

Figure 3.5: Transfer Time vs. Loss Rate for 500 KB file transfer over 1.5 Mbps/35 ms delay link

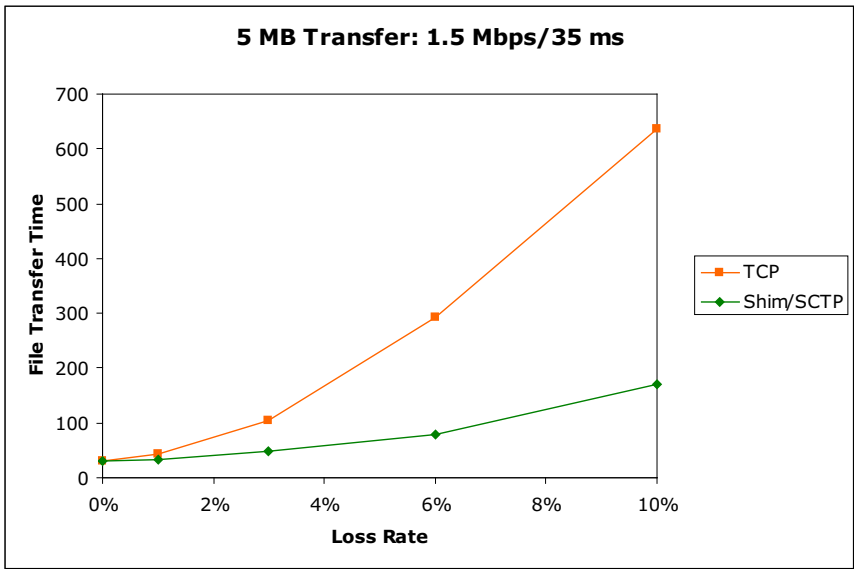

Figure 3.6: Transfer Time vs. Loss Rate for 5 MB file transfer over 1.5 Mbps/35 ms delay link

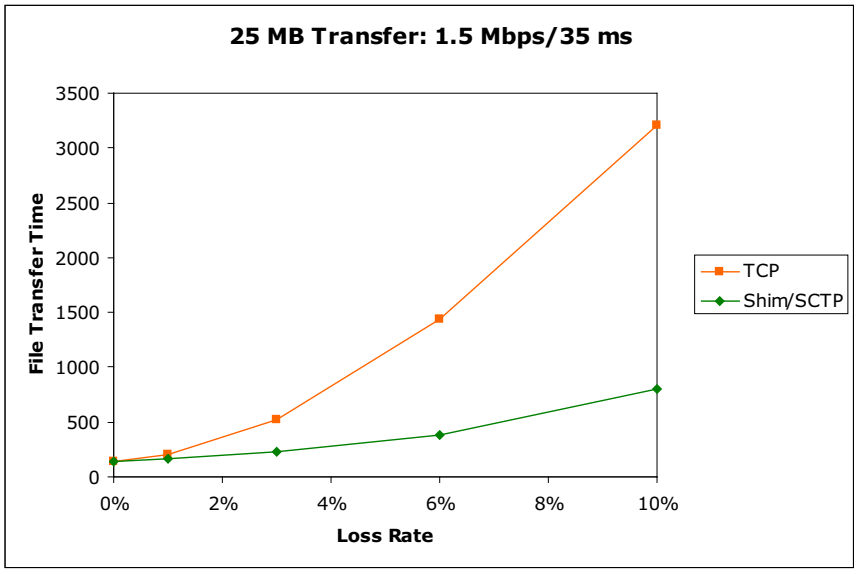

Figure 3.7: Transfer Time vs. Loss Rate for 25 MB file transfer over 1.5 Mbps/35 ms delay link

Chapter 4

CONCLUSION & FUTURE WORK

4.1 Conclusion

The transparent TCP-to-SCTP translation shim encourages the increased deployment of SCTP by providing a path for gradual migration from legacy TCP applications to native SCTP applications. In addition to encouraging the use of SCTP by ensuring compatibility with legacy TCP applications, the shim also allows legacy TCP applications to enjoy SCTP's multihoming advantages (i.e., fault tolerance and concurrent multipath transfer) without requiring any modifications to the legacy applications themselves.

The experimental results presented in Chapter 3 illustrate that not only is the shim approach an interesting theoretical concept, but 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 extremely valuable for network applications. By ensuring that existing legacy TCP applications can seamlessly interact with new SCTP applications, the shim encourages innovation and the increased deployment of SCTP throughout the Internet.

4.2 Future Work

Although the TCP-to-SCTP translation shim as currently implemented works well with many applications and is robustly designed, the shim has some limitations, and has the potential for additional features. The following sections address some of the areas where improvement might be useful or additional features could be explored to enhance the shim into more valuable tool.

4.2.1 Parallel TCP and SCTP Connection Attempts

Data transfer between two applications using the shim does not incur any additional overhead (i.e., delay or throughput) beyond the normal overhead of the underlying transport protocol in use. Specifically, using the shim does not affect the performance of application data transfer; only the inherent differences between TCP and SCTP, and how they handle loss, congestion, and failure affect performance. The one situation where using the shim can degrade performance is during the connection or association establishment phase.

As described earlier in Section 2.4, the connect system call is modified in order to support the TCP-to-SCTP translation shim. When the shim is enabled for a particular application, that application first tries to connect to its peer using an SCTP association, falling back to a regular TCP connection if the peer does not support SCTP. In cases where the remote endpoint does not support SCTP, this detection process takes at least one RTT to complete and fall back to TCP, which then requires another RTT in order to set up the regular TCP connection.

This additional delay of at least one RTT in situations where SCTP is unsupported by the remote system could be avoided by having the shim initiate an SCTP association and a TCP connection to the remote endpoint in parallel. By the time the unavailability of SCTP is detected, the TCP connection should already be established, so data transfer could then begin immediately. If a parallel connection approach is used in scenarios where SCTP *is* available on the remote system, the TCP connection would simply be torn down once the SCTP association has been established. The decision to attempt connections with both transport protocols in parallel using the shim could be configurable by an administrator using a *sysctl* variable.

4.2.2 Caching SCTP Availability of Peers

Along similar lines with Section 4.2.1's idea of initiating both SCTP associations and TCP connections in parallel, another approach to reduce setup overhead might be

to have the shim cache knowledge about whether SCTP was available to a particular application on a particular host. This cached information would allow the shim to skip the SCTP or TCP setup phases in situations where the shim knows *a priori* that recent interactions with that application or host have successfully used either SCTP or TCP. A caching feature could be integrated into the shim and then enabled or disabled using a *sysctl* variable. Additional configuration parameters could include the duration of the cache lifetime, and how many recent entries to cache.

4.2.3 Simulating TCP's Half-Close

The most drastic difference between TCP and SCTP that has a visible effect on application functionality when using the TCP-to-SCTP translation shim is the lack of a half-closed state in SCTP. Because some existing TCP applications depend on TCP's half-closed state to operate correctly, SCTP's lack of a half-closed state can present complications for legacy TCP applications attempting to use the shim. A prime example of how this difference between TCP and SCTP can cause problems was described using FTP as an example in Section 3.2.

To ensure compatibility for all legacy TCP applications running over the TCP-to-SCTP translation shim, the shim may be able to exchange control information between the two connected peers to simulate the behavior of TCP's half close. Simulating the half-closed state would allow any TCP applications that depend on the semantics of the half-close to function correctly even when running over the shim. Legacy TCP applications using the shim have no knowledge of SCTP streams and use stream 0 for communication by default. A topic of future investigation would be to examine the feasibility of simulating TCP's half-closed state by exchanging control messages over another stream that is unused by the applications. This approach would add complexity because the shim would need to monitor the application data stream for control messages on other streams, but would make the shim compatible with legacy TCP applications that rely on the half-close. One possibility is adding support for this feature to the shim but using an expanded

rules table to only enable the half-close compatibility mode when deemed necessary, for instance when connecting to an FTP server port.

4.2.4 Shim Applicability to UDP Using PR-SCTP

Currently, the shim is designed only to translate system calls to TCP into equivalent calls to SCTP. One area of future work might be to investigate the applicability of a translation shim technique for UDP. Using the partial reliability extension to SCTP (PR-SCTP) [22], UDP-like services may be able to be provided in addition to taking advantage of the fault tolerance and possible CMT benefits of SCTP multihoming.

The existing design of the translation shim is TCP-centric, so some extensions would be required to extend support to UDP. Particularly, the shim rules table would need additional fields to allow each rule to specify whether the protocol the rule applies to is TCP or UDP. Any protocol to be used with the shim needs to have an IP protocol number, for example *IPPROTO_TCP*, *IPPROTO_UDP* or *IPPROTO_SCTP*. The reason for this requirement is that the normal protocol and the protocol being invoked through the shim need to have independent port spaces.

TCP and SCTP are more similar than UDP and SCTP, so extending the shim to support UDP and then experimenting to see how UDP applications react to running over SCTP via the shim might be an interesting research effort.

4.2.5 Multiplexing TCP Connections in SCTP Streams

SCTP supports multiple data streams within a single association. These streams are independent flows of data which do not block each other, essentially "virtual connections" flowing inside a single, real SCTP association. Since legacy TCP applications running over the TCP-to-SCTP shim are unaware of the existence of multiple streams, one interesting area of exploration would be to consider multiplexing multiple TCP connections between two hosts through a single SCTP association between the two endpoints.

For example, suppose a user on one system is running several legacy TCP applications over the shim, all of which connect to the same remote host. Rather than having an SCTP association for each application, perhaps a single real association could be used in place of all of the individual associations, where each individual association is mapped to one *stream* within the single, real SCTP association.

Motivations for why this multiplexed architecture might be desirable include keeping congestion information and RTT information current and accurate. Consider the case where several connections exist between two endpoints, but not all of the connections are active at once. After enough time passes, idle connections will have stale RTT and congestion measurements that may no longer reflect the present network conditions. If these independent data flows were instead multiplexed through a single SCTP association via separate streams, activity on any given flow/stream would maintain accurate network statistics for all of the flows since they are part of the same association. This strategy would make network operation more efficient for general cases where several long-term flows exists between two endpoints that are not continuously transmitting.

4.2.6 Comprehensive Shim Logging

Currently, the shim supports a single level of logging where only major error situations are written to the system log. A future area of work to simplify debugging of the shim would be to implement customizable log levels so that amount of detail appearing in the system log could be configured by the system administrator. Logging could be turned off completely or kept at a minimal level when the shim is operating properly for increased performance, with the option to enable more detailed logging for shim debugging purposes.

BIBLIOGRAPHY

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

[2] A. Caro. *End-to-End Fault Tolerance Using Transport Layer Multihoming*. Ph.D. dissertation, University of Delaware, Newark, Delaware, 2005.

[3] A. Caro, P. Amer, and R. Stewart. End-to-End Failover Thresholds for Transport Layer Multihoming. In *MILCOM 2004*, Monterey, November 2004.

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

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

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

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

[8] 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.

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

[10] P. Natarajan, P. Amer, R. Bickhart, and S. Ladha. Corrections on: Improving Multiple File Transfers Using SCTP Multistreaming. Corrections to [9], Protocol Engineering Lab, June 2005. http://pel.cis.udel.edu/poc/pdf/IPCCC2004CORRECTED-FTP-over-SCTP-Natarajan-6-6-2005.pdf.

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

[12] The FreeBSD Project. RECV(2) - FreeBSD System Manager's Manual, February 1994. http://www.freebsd.org/cgi/man.cgi?query=recv&sektion=2.

[13] The FreeBSD Project. SEND(2) - FreeBSD System Manager's Manual, February 1995. http://www.freebsd.org/cgi/man.cgi?query=send&sektion=2.

[14] The FreeBSD Project. TELNET(1) - FreeBSD General Commands Manual, January 2000. http://www.freebsd.org/cgi/man.cgi?query=telnet&sektion=1.

[15] The FreeBSD Project. SYSCTL(8) - FreeBSD System Manager's Manual, March 2002. http://www.freebsd.org/cgi/man.cgi?query=sysctl&sektion=8.

[16] The FreeBSD Project. IPFW(8) - FreeBSD System Manager's Manual, March 2005. http://www.freebsd.org/cgi/man.cgi?query=ipfw&sektion=8.

[17] The Icecast Project. Icecast Homepage, July 2005. http://www.icecast.org/.

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

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

[20] R. Stewart. Apache 2 for SCTP, July 2005. http://www.sctp.org/download.html.

[21] R. Stewart, I. Arias-Rodriguez, K. Poon, A. Caro, and M. Tuexen. Stream Control Transmission Protocol (SCTP) Implementer's Guide. Internet draft, Internet Engineering Task Force, June 2005. http://www.ietf.org/internet-drafts/draft-ietf-tsvwg-sctpimpguide-14.txt.

[22] R. Stewart, M. Ramalho, Q. Xie, M. Tuexen, and P. Conrad. Stream Control Transmission Protocol (SCTP) Partial Reliability Extension. RFC 3758, Internet Engineering Task Force, May 2004.

[23] R. Stewart and Q. Xie. *Stream Control Transmission Protocol (SCTP): A Reference Guide*. Addison Wesley Professional, New York, NY, 2001.

[24] 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.

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