Chapter 1
Introduction

From the debut of the first simulator, people have kept asking the same question: "How realistic the simulation result can be?" The reason for asking this question is quite reasonable, because a simulator is actually useless if it cannot mimic the behaviors of the real thing.

In the simulator's world, the words like accuracy or exactness are normally used to describe how realistic a test result is. To achieve accuracy, developers have made several efforts in the past decade or so. The focus, however, has been primarily on the use of real network stacks. As can be seen from some previous tests, a highly abstract network stack makes the test result neither convincing nor realistic.

Undoubtedly, a real world network stack is the most critical part of a realistic network simulator, but it is not the only part to achieve accuracy. The modern network simulators are quite complex systems and the other parts of a simulator such as event scheduler, timing, queuing and application layer, can also affect the final results of simulations. Among these factors, there are increased interests in the application layer of simulator.

1.1 Why Application Layer Is so Important

In the real world, different data sources make network stack behave differently which, therefore, affects network performance in different ways. When investigating the performance of network stack, it is unavoidable to associate such kind of investigation with a specific data source. The interaction between application layer and network stack is a major topic in networking history and will be more important in future.

The traditional HTTP 1.0 over TCP problem is perhaps, the clearest instance to demonstrate how application layer affects performance of network layer.

The HTTP 1.0 protocol creates a new TCP connection for each item that it requests from server. If, for example, a page contains nothing except three images, it will result in four TCP connections - one for the page itself and three others for each of the images. This is inefficient in terms of bandwidth utilization. A three-way handshake process is needed to set up a connection, and two additional

packets are required to close it. Even for a very simple page, the burden of setting up and tearing down of connections are still considerably heavy.

Another problem with the HTTP 1.0 is the poor bandwidth utilization incurred by the slow start mechanism. Instead of transmitting packets as fast as possible, the HTTP 1.0 sets the TCP window size to one segment initially. The TCP window size doubles each time a whole window of packets are transferred successfully. This mechanism works fine when the connection is long-lived, but for short-lived connections this slow start mechanism is quite inefficient in terms of network bandwidth utilization. Several studies have shown that the vast majority of web accesses retrieve small files, on the order of 6 KB [15] [16]. In other words, if the payload of packet is 1500 bytes in size, a 6K bytes web page needs only 4 packets. Because slow start is initialized each time when a connection is created, so for most HTTP 1.0 connections it always works at very low data rate and utilizes network bandwidth poorly.

Unfortunately, in spite of tremendous development in technology, the modern world still suffers the same problem as the old HTTP 1.0 did. The Internet is now increasingly being used to deliver real time data such as video-conferencing, streaming, HDTV, P2P and so on. Most of real time applications are now running on top of UDP, which is a highly efficient, but unreliable transport layer protocol. The real time UDP packets may experience different bandwidth and congestion conditions, so an annoying drawback with UDP is the data loss. For some real time communications, data loss can affect Quality of Service (QoS) significantly and thus, is difficult to tolerate. In video-conferencing, the data being transferred is typically significantly compressed to save network bandwidth, and this makes it very sensitive to data loss. A single packet loss corrupting an I-frame can stop reception of video for several seconds. To achieve high QoS, other reliable transport layer protocols such as TCP have to be considered.

TCP is the most well developed, extensively used and widely available Internet transport protocol. TCP is reliable and responsive to network congestion conditions. Another reason to choose TCP to run real time applications is the reality that many firewalls reject all non-TCP traffic. On the other hand, the fatal drawback of TCP is its low efficiency compared to UDP. TCP is not suitable for transmitting real time traffic because it favors reliable delivery over timely delivery.

Both TCP and UDP are not perfect for transmitting real time multimedia data. To adapt to future network environment, TCP and UDP have to be modified or even new protocols will have to be introduced into transport layer such as Stream Control Transmission Protocol (SCTP).

Thus, network applications affect the performance of network stack and even

make the network stack to be modified. Application layer is critical to network performance. However, in simulators data sources are implemented using very simple model and have nothing to do with real world network applications. There are several drawbacks with such simulated user applications.

1.2 Limitations of Simulated Applications

Simulated applications are usually over-simplified and network simulators take a simple way to implement data sources. Timer is usually used to control when to send out the next message. After starting the application, a timer is initialized to wait for a short period of time and an event handler is set to handle timeout events. When the timer expires, simulator's scheduler invokes the indicated event handler, which, in turn, sends out messages and then resets timer for the next message. In the NS-2, a simulated "telnet" is just about 30 lines of C++ code. Such implementation is not programmed using real world application code such as BSD Socket API; and even worse, many details of real world network applications are ignored intentionally to facilitate designing. The question, therefore, remains as to what extent such over-simplified simulated applications mimic the behaviors of real world network applications.

In the real world, countless network applications are available. These come with either source code or only binaries, but just limited amount of applications are provided in simulators. At times, new applications in simulator have to be created by extending the existing application class. For example, in NS-2 you can derive your own application class from the existing class "Application". Such implementations are still employing over-simplified model and time-consuming.

The simulated applications are neither realistic nor sufficient. Thus, to solve this problem, a new model called “BSD Socket API for Simulator” is proposed here to run untouched real world network application binaries on top of general-purpose Simulators.

1.3 BSD Socket API for Simulator

Running real world code on top of simulator has been implemented in several projects such as embedded network simulator, ENTRAPID, Alpine, NCTCns, and Lunar. All of these projects, however, have some limitations such as:

- Some of the projects, such as embedded network simulator and NCTUns, need the simulator to be integrated into kernel. Only simple simulators can be used in such project as these cannot make full use of the powerful functionality and versatile tools provided by modern general-purpose

simulators such as NS-2, Omnet++ and JSim.

- Some of the projects, such as ENTRAPID and Lunar, need the real world applications to be recompiled and re-linked against some special libraries. This approach is not working for those real world applications that do not come with source codes.
- Some of the projects, such as Alpine and NCTUns, need root privileges.
- All of the projects use exclusive simulators

On the other hand, the BSD Socket API for Simulator is designed to eliminate most of these drawbacks. It is fully compatible with BSD Socket API, simulator independent, transparent to real world applications and it keeps the original real world application binaries untouched.

1.4 Contributions of this Thesis

The achievements of the BSD Socket API for Simulator are as follows:

- A new simulation framework is designed and implemented. The new model is based on the concept of “message redirecting”. Real world applications redirect messages by loading customized shared libraries. Simulator exchanges information with the shared libraries through its customized application layer.
- A multi-threaded simulator application layer is implemented. The single thread simulator, NS-2, does not suit for running unmodified real world applications due to the existence of blocked calls. To solve the problem, a multi-threaded model is proposed and a multi-threaded simulator application layer is implemented.
- A shared library is implemented as the interface of real world applications to redirect messages to simulator. The shared library is loaded into the address space of real world application. It catches the socket related calls and redirects them to simulator.
- Unit tests have testified that the 29 socket-related functions in the shared library are compatible with their counterparts in the BSD Socket API.
- Three groups of real world applications are run on top of the BSD Socket API for Simulator as functional tests.
- Efficiency tests are done to measure to what degree the BSD Socket API for Simulator slows down simulations.

1.5 The Rest of this Document

- Chapter 2 "Background" introduces background of this project. Some previous works are presented and their advantages and disadvantages are

analyzed. Among the previous works, the Network Simulation Cradle (NSC) is explained in detail.

- Chapter 3 "Architecture Overview" explains briefly how the two major parts of the project, namely, the simulator application pair and the BSD Socket API shared library, are implemented and how they interact with each other.
- Chapter 4, "Multi-threaded Simulator Application Pair", describes why the multi-threaded simulator application pair is necessary in this project, how it is implemented and the mechanism employed to balance work load among multiple simulator application pairs.
- Chapter 5 "Shared Library" is core to this document. This chapter describes the procedure to set up and tear down basic TCP and UDP connections; and introduces the implementations of various sending and receiving functions such as send(), recv(), sendto(), recvfrom(), sendmsg() and recvmsg(); closeing socket by close () and shutdown (); I/O multiplexing by select(), pselect() and poll(); running standard I/O library functions over sockets; getting and setting socket options.
- Chapter 6 "Tests" introduces unit tests, functional tests and efficiency tests done in this project.
- Chapter 7 "Limitations and Future Works" analyzes limitations of "BSD Socket API for Simulator" and what else can be done to improve it in future works.
- Appendix A "Function list" shows all the socket-related function prototypes implemented in the BSD Socket API for Simulator shared library.
- Appendix B "Kernel Data Structures" lists data structures being moved out of kernel. Due to the name conflict when compiling the BSD Socket API for Simulator, some kernel data structures have to be moved to a user-level header file.
- Appendix C "Modifications to code from UNIX Network Programming". Examples from "Unix Network Programming" [W. Richard Stevens et al] are supposed to run on UNIX, but the BSD Socket API for Simulator is running on Linux. So when compiling the code on Linux, a number of "Symbol missing" errors occurred. To eliminate such errors, some symbols are defined.

Chapter 2
Background

Running real world applications on top of simulator is not a new effort, but organizing powerful general-purpose modern network simulator, real world network stack and real world application in one system is new to the world. In this chapter, some related past works are reviewed; advantages and disadvantages of each work are analyzed. Subsequently, the Network Simulation Cradle (NSC) is introduced in depth. The NSC is the first effort ever to integrate general purpose simulator with real world network stack and is chosen to be the base of the BSD Socket API for Simulator.

2.1 Past works

From the perspective of network simulator history, the efforts of using real world network stack and the efforts of running real world application on top of simulator are closely related. Each of the following past works is reviewed from two aspects: the way the real world network stack is integrated with simulator and the way the real world application is integrated with real world network stack.

2.1.1 Embedded Network Simulator

The Embedded network simulator [Luigi Rizzo, 1997] is a protocol development environment. To make use of the advantages of network simulator and experimental test bed, this project proposes to embed network simulator in operational systems. A simulator is built in kernel at the interface between TCP and IP.

The embedded simulator intercepts calls between the two layers and generates the effects of queues, bandwidth limitations, delays and noises. Not surprisingly, such system enables real world network applications to run without the need of modification. Embedded network simulator moves simulator into kernel instead of moving network stacks out of kernel. In the past when simulators were simple and of limited functionality, this approach was viable. Actually, the first simulator built in this project was only 300 lines of code.

The limitations of this approach are the need of modifications to all the kernels in

which the simulators will be running and the efficiency cost when the simulators are quite complex. It is impossible to build a modern simulator such as NS-2 or Omnet++ inside kernel, not mentioning the simulators using JAVA like J-Sim.

2.1.2 ENTRAPID

ENTRAPID [X.W. Huang et al. 1999] is a real time network simulator and a Protocol Development Environment (PDE) as well. Unlike the previous embedded network simulator, the ENTRAPID is nothing but a user space process. It is built on two concepts: Virtualized Networking Kernel (VNK) and Virtualized Process. VNK is the 4.4 BSD network stacks, which is moved out of kernel and modified to let it run in user space. By means of kernel virtualization, an ENTRAPID process can contain multiple VNKs. Applications are also virtualized to run on top of VNK. System calls issued by virtualized applications are redirected to VNKs. ETRAPID can also control physical network devices to communicate with external processes.

ENTRAPID can organize a quite complex and powerful test framework, but it has some limitations. Firstly, the network kernel is modified manually. The process of moving the stack out of kernel is quite a lot of work and it is hard to keep the modified stack up to date. Secondly, the ENTRAPID cannot run unmodified programs. The program's source code is linked with a proxy library that first converts networking and file system calls to messages, and then sends the messages to a virtualized process. Subsequently, the virtualized process decodes the message and executes the network or file system calls in the context of ENTRAPID. Thirdly, because a message from an application is copied to VNK, from VNK to wire, from wire to another VNK, from VNK to destination application, so ENTRAPID lacks efficiency.

2.1.3 Alpine

Alpine [David Ely et al. 2001] is a user-level infrastructure for network protocol development. Alpine converts FreeBSD 3.3 network stack into a user-level shared library with a few hand modifications. There are two critical modifications that need to be mentioned.

The first one is to add a new layer under the IP layer called "Faux-Ethernet Driver" to the shared library. This new layer is the lowest layer of the shared library, it is used to send packets using a raw socket and receive packet using a packet capture tool (libpcap).

The second modification is to add a new layer on top of the system call layer. The new layer is the topmost layer of the shared library. The new layer implements a new programming interface to replace the traditional system calls. This layer captures all the system calls on socket descriptors or file descriptors and deals with them transparently.

Alpine moves network stack out of kernel with fewer efforts than embedded network simulator and ENTRAPID. Another attractive feature of Alpine is that application binaries can run on top of it without the need of recompiling and re-linking.

There are, however, two drawbacks for Alpine. Firstly, some operations, like opening a raw socket, capturing packets by libpcap, need root privileges; secondly, the fork() system call is not implemented inside the new application interface layer, a large portion of network applications cannot run on Alpine.

2.1.4 NCTUns

NCTUns [S.Y Wang et al. 2003] is a real time network emulator and simulator. Instead of moving network stack out of kernel, NCTUns keeps network stack where it was but modified by hand to integrate simulation functionality into kernel. NCTUns takes a similar way as that of embedded network simulator.

Tunnel Network Interface is key to NCTUns and is a pseudo network interface that does not have a real physical network attached to it. The interface has a corresponding device special file in the /dev directory. When applications write to the special file, the packet enters the kernel. To the kernel, there is no difference between packets from a real interface and that from the tunnel network interface. A read from the tunnel network interface causes one packet to be copied from kernel to user space and passed to the reading application. Therefore, the tunnel network interface behaves completely the same way as the real network interface.

Besides real world network stack, another benefit NCTUns provides is the ability to run unmodified real world network applications and real world network utility programs by exposing the standard UNIX POSIX API.

There are, however, several drawbacks of NCTUns: the first one is the scalability limitation, because there is only one network stack on a host, to simulate a complex network with different network stacks will need multiple hosts and a distributed test environment. The second limitation is that the user needs root privileges to modify and recompile the kernel and run simulations.

2.1.5 Lunar

Lunar [Christopher C. Knestrick. 2004] is a Linux User-level Network Architecture. Lunar is designed as part of Open Network Emulator (ONE), a large-scale network emulation test-bed. In Lunar, the network stack of Linux operating system (2.4.3 version of the Linux kernel) is extracted and compiled as a user-level library. The shared library provides an interface which implements the BSD socket API and related functions. The unmodified real world application source code is recompiled and re-linked against Lunar's shared library.

Lunar acts as an interface between user applications and network simulator. Network traffic is generated using real-world programs and regulated by network simulator. Lunar combines the benefits of direct code execution with the control and scalability of modern network simulator.

In Lunar, real world applications have to be recompiled and linked against Lunar before running. The real world applications which source codes are not available cannot run on Lunar.

2.1.6 Network Simulation Cradle

Network Simulation Cradle (NSC) [Sam Jansen. 2005], like Alpine and Lunar, moves network stack out of kernel and compiles it as a shared library. The NSC uses the shared library to create Agent for NS-2 network simulator.

NSC integrates real world network stacks with the NS-2 by creating an NSC agent. The agent creates, initializes and interacts with real world network stacks. The network stacks themselves become part of the network simulator.

Real world network stacks are modified automatically using a tool, so there is no manual modification in NSC. Multiple modified network stacks can be run in the same simulation and on the same physical machine.

The disadvantage, however, of NSC is that it still uses the simulated applications to generate network traffic. NS-2's applications are simply instances of some C++ classes employing quite simple model.

2.1.7 Characteristics of Past Works

The following Table summarizes the characteristics of past works in terms of simulator, network stack and application.

Name	Simulator	Network stack	Application
Embedded Network Simulator	In kernel Simple Exclusive	In kernel Modified by hand	(1) Run unchanged real world application
ENTRAPID	In user space Exclusive	In user space Modified by hand	(1) Real world application's source code is compiled and linked against a proxy library
Alpine	In user space Exclusive	In user space Modified by hand Shared library	(1) Only run part of unchanged real world programs (2) Need root privileges
NCTUns	In kernel Exclusive	In kernel Modified by hand	(1) Run unchanged real world application (2) Need root privileges
Lunar	In user space Exclusive	In user space slightly modified user-level library	(1) Real world application's source code is compiled and linked with a user-level library
NSC	In user space Eeneral-purpose	In user space slightly modified user-level library	(1) Simulated Application

Table 2.1 Characteristics of Past Works

From the above table, there are some options for three interesting aspects of past works. These options lead to different features.

- Simulator can be in user space or in kernel. In kernel simulator is usually functionally limited and it is a painful process to debug the simulator or add new features. Simulator can be exclusive or general-purpose; the exclusive simulator is built to meet some special needs, so it does not contain versatile tools provided by general-purpose simulators.
- Network stack can be in kernel or user space. In kernel network, stack suffers the same problem as that of in kernel simulator. Besides, in kernel network stack is not scalable. In user space network, stack is usually compiled as a library. It increases the flexibility and scalability.
- Application can be:

(1) Untouched real word code.
(2) Untouched real world code but need special conditions such as root privileges.
(3) Real world code but need to be recompiled and re-linked.
(4) Simulated application.

It is, therefore, obvious that the "untouched real world code" is the ideal solution.

2.2 Goals of BSD Socket API for Simulator

To eliminate limitations of past works and distinguish the BSD Socket API for Simulator with previous works, several goals are set up for BSD Socket API for Simulator.

1. BSD Socket API for simulator is completely compatible with BSD Socket API. Socket API is part of IEEE1003.1g standard, which is also called POSIX.1g. Most UNIX and Linux systems today conform to Portable Operating System Interface (POSIX). Socket API has been a networking standard for UNIX-like systems.
2. BSD Socket API for Simulator is simulator independent. Currently, the BSD Socket API for Simulator is developed and tested on NS-2, and it is supposed to be moved to other general-purpose simulators such as Omnet++ with trivial works. It can make use of the powerful functionality and tools provided by general-purpose simulators.
3. BSD Socket API for Simulator keeps real world application binaries untouched, no recompilation and re-linking are necessary before running them.
4. BSD Socket API for Simulator is transparent for real world applications. These applications run on simulator exactly the same as they do on an operating system. No special conditions, such as root privileges, are necessary when running them.

From table 2.1, an interesting thing worth noting is that NSC is the only attempt ever made to integrate general-purpose simulator with real world network stack. It is the only one that satisfies the second goal of BSD Socket API for Simulator, simulator independent. Besides, NSC has other advantages to make it the best candidate to implement BSD Socket API for Simulator.

2.3 Network Simulation Cradle

NSC is the first development of integrating real world stacks with existing general-purpose network simulator. It has been implemented on top of NS-2 and is being transferred to Omnet++. The figure below, taken from “Simulation with Real Network Stacks” [Sam Jansen and Anthony McGregor], shows per-network stack interactions of NSC:

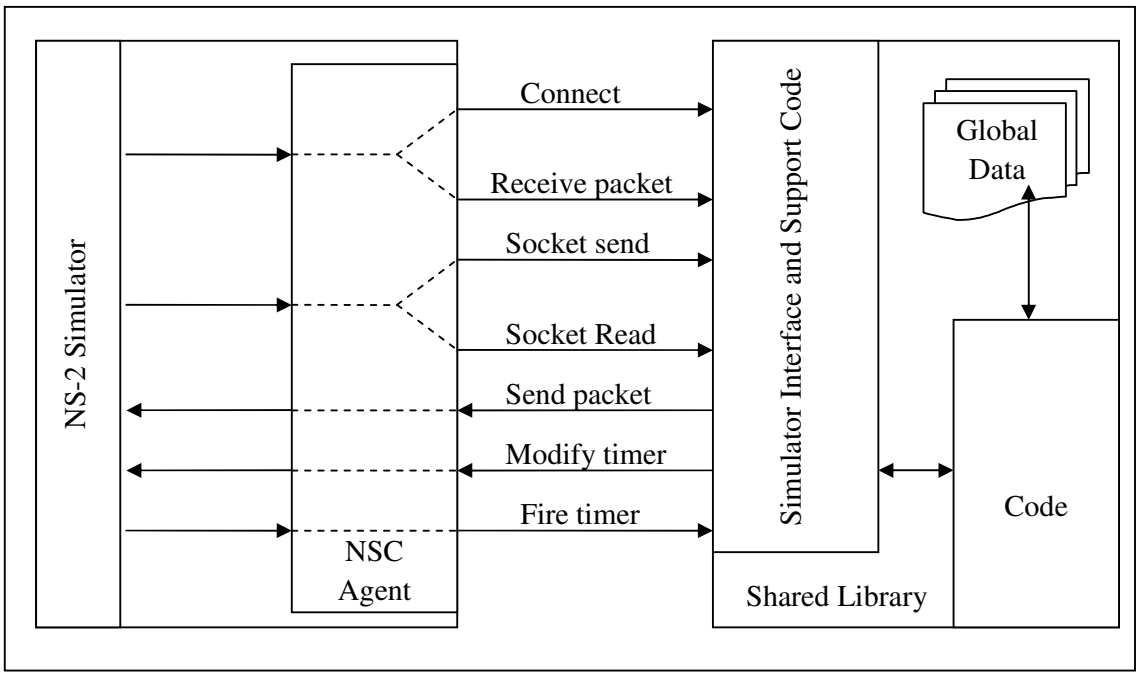

Figure 2-1 Per-network stack interactions

NSC consists of two parts, NSC agent and shared library that communicate through a well defined interface. There are two tasks for NSC agent. Firstly, it is used to create and interact with another part of NSC, the shared library. Secondly, it is used to interact with other parts of NS-2 simulator such as NS-2's application layer. The shared library contains the network stack and supporting code that implements the interface necessary to communicate with NS-2.

NSC has several characteristics that can distinguish it with other past works.

1. Real network stacks are integrated with simulator seamlessly. Real network stack becomes part of simulator. Users create an NSC-agent completely the same as they do with normal NS-2 agents.
2. NSC is simulator independent. NSC does not depend on any specific simulator; theoretically, it can be used with any general-purpose modern network simulator. Originally it was

implemented in NS-2, and it is going to work with Omnet++ as well.

3. There is no hand modification to network stack in NSC. A C parser modifies the network stack programmatically to automate the virtualization of network stacks.
4. Different network stacks or multiple instances of the same network stacks can be used in one simulation and on the same host machine. This feature increases the scalability of simulation greatly. NSC is able to employ network stacks from Linux, FreeBSD, OpenBSD and the network stacks designed for embedded systems such as IwIP.

NSC is the first effort ever made to make use of the strength of both real network stacks and modern general purpose simulators. Due to its nature of simulator independence, NSC is chosen as the base of the BSD Socket API for Simulator.

2.4 Summary

Several past works are reviewed and their strengths and drawbacks analyzed in terms of simulator, network stack and application. Among the past works, the NSC has some advantages over the others that can best satisfy the need of the BSD Socket API for Simulator and, thus, is chosen as the base of the project.

Chapter 3
Architecture Overview

This chapter introduces the overall architecture of BSD Socket API for Simulator. The introduction begins with an abstract block diagram, and then followed by introductions to functions of each items on the diagram including simulator application pair, integrating simulator application pair with NSC, BSD Socket API shared library and the communication between shared library and simulator application pair.

BSD Socket API for Simulator consists of two parts: shared library and simulator application pair.

The shared library is pre-loaded by setting the environment variable LD_PRELOAD when running real world applications (RWAs). As a result, system calls and function calls on socket descriptors is intercepted and messages contained inside the calls are redirected to simulator by the shared library.

The redirected messages are accepted by one side of simulator application pair and then passed down to Network Simulation Cradle (NSC). Subsequent to this, it is the responsibility of simulator and NSC to deal with the message. How the message traverses simulator and NSC will not be discussed here. Actually, it has no difference from those that are passed by normal simulators without NSC. The messages ultimately reach the other end of simulator application pair.

The other side of simulator application pair gets the message from NSC's interface and relays it to the destined real world application. The destined real world application has shared library pre-loaded as well. The shared library extracts payload the message originally sent by sender, from the received message. Finally, the payload is passed to the destined real world application.

3.1 Structure of BSD Socket API for Simulator

The BSD Socket API for Simulator is composed of two objects, shared library and simulator application pair. As real world applications and simulator run in different address spaces, to make data flow between source real world application and destination real world application via NS-2, there must be an inter-process communication mechanism. For simplicity, an assumption is made here: the

communication mechanism does exist, so different parts of BSD Socket API for Simulator can talk to each other. The inter-process mechanism used in this project will be explained in a following section.

The following figure shows the typical data flow of BSD Socket API for Simulator: a source real world application is sending a message out; its destination is another real world application. The arrows show the direction of data flow; solid lines represent inter-process communication; dashed lines represent intra-process communication.

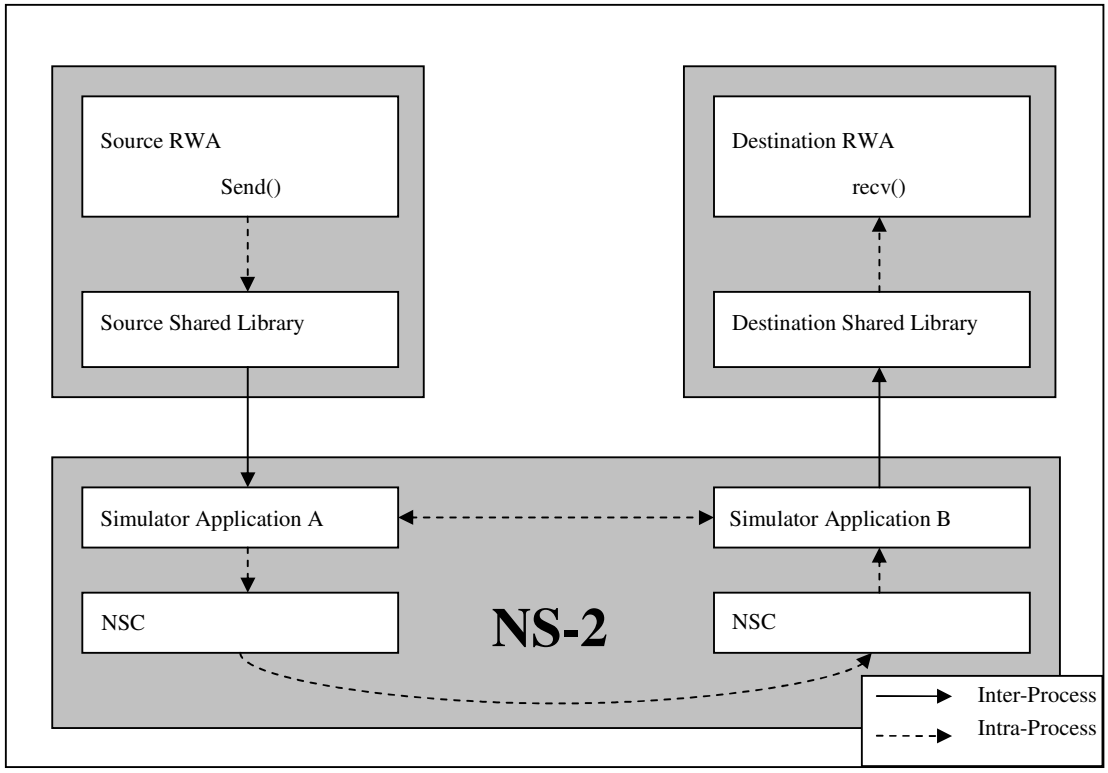

Figure 3-1 Structure of BSD Socket API for Simulator

Each simulator application has two interfaces: one receives messages from shared library and sends the received message to NSC. The other one receives message from NSC and forwards the received message to shared library.

The scenario shown in figure 3.1 is quite straightforward. Both the source and destination RWAs pre-load shared libraries which are called source shared library and destination shared library respectively. System calls or function calls issued by source RWA, such as send () or fputs(), are intercepted by source shared library and processed accordingly. The processing here involves appending extra information, such as where the message comes from and where it is heading to, to the original message contained in the original calls. By this way, each of the intermediate nodes on the route from source RWA to destination RWA will learn

about the destination of the message. This extra information will be used by simulator application B when it receives message from NSC agent. It investigates the extra information to obtain the destination address and then relays the message to the destination RWA. The new message is then redirected to simulator and finally reaches simulator application B.

The destination application B redirects the received message out of simulator to destination RWA.

The destination shared library catches the message sent by simulator application B and copies payload, the message sent by source RWA, to destination RWA's receiving buffer.

3.2 Simulator Application Pair

The process shown in figure 3-1 is actually a bi-directional communication in BSD Socket API for Simulator. The two-way communication is achieved by connecting simulator application A and B together to form a simulator application pair.

The simulator application is a multi-threaded implementation. There is one administration thread and multiple serving threads. Each serving thread is responsible for communicating with a specific shared library. One simulator application can serve more than one real world application. In other words, one simulator application pair can serve multiple real world connections. The following diagram shows how simulator application pairs connect real world applications and simulator together.

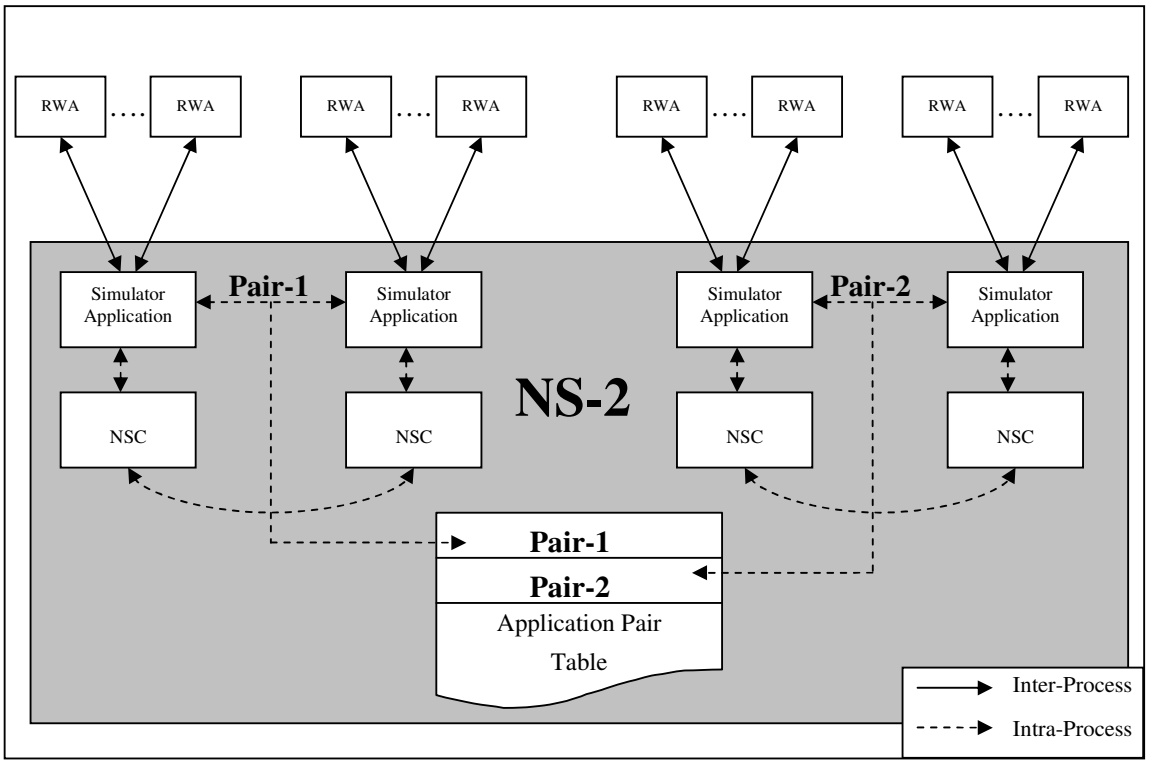

Figure 3-2 Simulator application pair

In the figure above, there are two simulator application pairs, pair-1 and pair-2. Information about the pairs is stored in application pair table. The application pair table is in shared memory; all the application pairs share the same table. As can be seen from the figure above, simulator application pairs act as the interface of simulator to outer world and each simulator application can serve multiple RWAs. When a real world application starts up with pro-loaded shared library, the shared library will register him with administration thread. The administration thread then looks up the application pair table to choose a simulator application to serve the starting real world application as well as provides load-balancing among multiple simulator application pairs.

When an simulator application receives message from real world application, it extracts payload and pass the payload to NSC by calling NSC agent's method recv(). Because NS-2 only transmits the number of bytes other than the real message, so the simulator application only passes the number of payload in byte down to NSC agent.

Transmitting the number of bytes is fine for simulated applications, but for real world applications the real message must be transmitted to achieve two-way communication. There are several ways to accomplish this and most of them need to modify the NS-2 simulator itself. Besides, the current implementation of NSC does not support real message transmission, so the NSC has to be modified as

well. These modifications change the basic simulation structure of NS-2 substantially. It may weak the credibility of simulation results and may make the simulation results incomparable and invincible. The modification is time-consuming as well.

The solution of BSD Socket API for Simulator to the problem is to keep NS-2 and NSC untouched, all the tricks are done inside the simulator application: two simulator applications are connected in some way to form a pair as shown in figure 3-2. Two ends of an NS-2 application pair can communicate with each other.

One end of application pair, which receives message from real world application, keeps the actual message received in its inner buffer and transmits the byte number of payload down to NSC. When the other end of application pair receives the message (number of bytes), it fetches the actual information from the inner buffer of its partner and passes it up to destination RWA.

Fetching from partner is only necessary for simulators such as NS-2 that do not transmit actual message, but for those network simulators that support real message transmission, such as Omnet++, the fetching stage can be ignored.

The other feature of simulator application pair, which is not yet being presented, is the real world application multiplexing: one simulator application can serve multiple RWAs. RWA multiplexing is achieved by adding a small header to the actual message, and this is done in shared libraries. The following section will cover how shared library and simulator application interact with each other and how the real world application multiplexing is achieved.

3.3 BSD Socket API Shared Library

The shared library is supposed to run in the same address space of Real world application (RWA) by setting the environment variable LD_PRELOAD. Functions in shared library have the same names and signatures as those of corresponding system calls and function calls. If RWA issues such kind of calls, they will be caught by the shared library other than executing the intended system calls or function calls. In shared library, these calls will be processed in some way and messages will be exchanged with simulator. The process of executing a system call or a function call is depicted in the following figure.

In the figure below, a typical case is taken as example: a real world program is calling send() system call to send something to another real world program, which is calling recv() system call. The steps BSD Socket API for Simulator takes to

redirect that message are shown as arrows in the figure. The solid lines represent inter-process communication and the dashed ones represent intra-process communication.

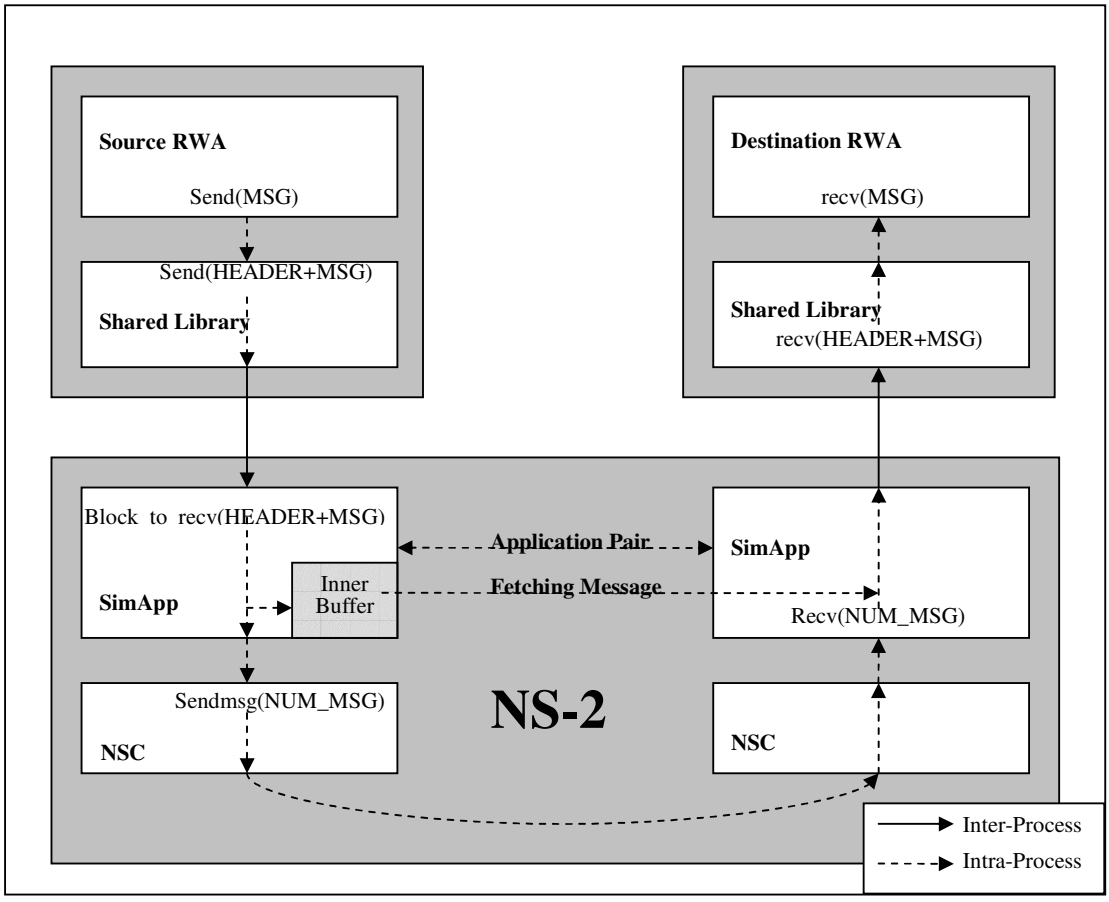

Figure 3-3 Data flow between RWAs via NS-2

System calls or function calls issued by real world applications are caught by shared library in the first place. The destination address and port number are obtained through socket descriptor contained in the original system call send (). The information is then used to generate a small header; the header describes source address, source port number, destination address and destination port number of current message.

The header together with the message found in the original send () call are sent to the connected simulator application. The simulator application puts the received message (header and payload) into an inner buffer and sends the byte number of payload down to NSC. The message (byte number of payload) travels through NS-2 and finally reaches the other side of simulator application pair.

The simulator application that received message from NSC fetches message from its partner's inner buffer. The message, header and payload, is then sent to

destined real world application according to the destination address and destination port number found in the header.

Precisely speaking, the message goes to the pre-loaded shared library other than RWA. The shared library copies the payload contained in the received message to receiving buffer found in the recv() system call of RWA.

3.4 Summary

BSD Socket API for simulator consists of two distinct objects: shared library and simulator application.

Every real world application in a simulation is started with shared library pre-loaded. As a result, all the socket-related system calls and function calls are caught by shared library in the first place. The shared library is responsible for exchanging message with simulator application.

Simulator application is either accepting message from RWA (shared library) and sending the received message to NSC or accepting message from NSC and sending the received message to RWA (shared library).

Shared library and simulator application altogether act as the interface between RWA and simulator.

Chapter 4
Multi-Threaded Simulator Application

The simulator application acts as the interface of NS-2 to communicate with real world applications. On the one hand, it receives messages from source real world applications and hands the received message to Network Simulation Cradle (NSC); on the other hand, it receives messages from NSC and relays the received message to the destination Real World Applications (RWA).

This chapter investigates NS-2's scheduler and draws a conclusion that the current NS-2 infrastructure does not suit for running unmodified RWAs due to the existence of blocked calls in RWA. To solve this problem a multi-threaded model is proposed. The rest of this chapter explains how the proposed multi-threaded model interacts with RWA, NSC and TCL script. In the model, simulator applications exist in pair, an application pair table is used to record application pair information and distribute work load evenly among application pairs.

This chapter takes code from NS-2 as examples to illustrate why the multi-threaded model is necessary. All the source file names are relative to the root directory of NS-2.

4.1 Why Multi-Thread

In real world applications, there are blocked calls such as accept (), recv() read() and select(). Program will stop if such blocked calls are encountered; it is blocked until some conditions are satisfied, for instance, the other side of a TCP connection issues connect () system call, some messages are received or changes in file descriptor sets are detected. These blocked calls are substantial to real world applications.

In simulators, these blocked calls are implemented in a different way from that in real world applications. Other than blocking on some conditions, simulators simply use timers to mimic blocking. NS-2's "telnet" (apps/telnet.cc) is listed as follows to show the basic implementation scheme of simulator's application layer.

```
1: void TelnetApp::start()
```

```
2: {
3:          running_ = 1;
4:        double t = next();
5:        timer_.sched(t);
6:}
7:void TelnetApp::stop()
8:{
9:             running_ = 0;
10:}
11:void TelnetApp::timeout()
12:{
13:              if (running_) {
14:                /* call the TCP advance method */
15:             agent_->sendmsg(agent_->size());
16:             /* reschedule the timer */
17:             double t = next();
18:             timer_.resched(t);
19:      }
20:}
21:double TelnetApp::next()
22:{
23:              if (interval_ == 0)
24:                    /* use tcplib */
25:                    return tcplib_telnet_interarrival();
26:             else
27:                    return Random::exponential() * interval_;
28:}
29:void TelnetApp::recv(int num)
30:{
31:      printf("telnet received message\n");
32:}
```

The start() method starts "telnet" by calling a timer to schedule a time to send out message (lines 1 to 6) and the stop() is used to "stop" "telnet" by setting a flag to false (lines 7 to 10). The core to "telnet" is the timeout() method (lines 11 to 20), it is invoked by NS-2's scheduler each time when the timer timeouts, after sending out message (actually, the byte number of current message), the method calls next() (lines 21 to 28) to generate another interval randomly and reschedules the timer using the new interval. The recv() method is nothing but printing out a line showing that a message has been received, no blocking at all.

As can be seen from last code snippet, "telnet" calls sched() (for the first message only) or resched() to schedule when to send out next message, so it is helpful to have a look at how the two methods are implemented (common/timer-handler.cc).

```
1:void TimerHandler::sched(double delay)
2: {
3:      if (status_ != TIMER_IDLE) {
4:              fprintf(stderr,"Couldn't schedule timer");
5:              abort();
6:      }
6:      _sched(delay);
8:      status_ = TIMER_PENDING;
9: }
10: void TimerHandler::resched(double delay)
11: {
12:    if (status_ == TIMER_PENDING)
13:    _cancel();
14:    _sched(delay);
15:    status_ = TIMER_PENDING;
16:}
```

Both sched() and resched() finally call _sched() method of Scheduler to actually schedule sending out message, which can be found in "common/timer-handler.h" as follows.

```
1:inline void _sched(double delay) {
2:      (void)Scheduler::instance().schedule(this, &event_, delay);
3:}
```

The inline definition of _sched() calls the class Scheduler's static method instance() which return the unique instance of Scheduler in NS-2, and then actually schedules the new event.

From the last three lists it can be seen that all the sending and receiving actions are scheduled and performed by the scheduler of NS-2, although there maybe lots of simulated applications issuing send() and recv() at the same time, these calls are actually running in the same thread, the thread in which NS-2's scheduler is running. In fact, the whole NS-2 is running as a unique thread in the unique process. So if any one of the calls issued by real world applications were blocked, the NS-2 simulator would be blocked.

The first problem needs to be solved before running real world programs on top of simulator is to avoid blocking simulation while blocked system calls are encountered. There are several possible solutions to this problem, for example, we can modify the "Scheduler" class of NS-2 to support blocking calls, this will need to change its simulation scheme substantially, this is undesirable because it does not satisfy the rule of "simulator independence" as described in chapter 2. An

alternative way is taken in BSD Socket API for Simulator: all the blocked calls from real world applications will be processed inside application layer of NS-2, it is transparent to all the other parts of simulator and there is no need to modify scheduler. This approach keeps the simulator untouched, and thus achieves simulator independence. Due to the its nature of simulator independence, although BSD Socket API for Simulator is currently running on NS-2, it will be fairly easy to transfer it to other simulators such as Omnet++.

4.2 Implementing multi-thread in NS-2

Multi-threading is not a new topic, a thread exists within a process and uses the process resources. It has its own independent flow of control and thus can be scheduled and executed independently. A thread is also called a "lightweight" process because it demands much less system overhead than a process. Because threads within the same process share resources, changes made to the shared resources by one thread can be seen by others, this can cause race conditions and therefore requires explicit synchronization by the programmer such as mutexes or condition variables.

As NS-2 is implemented using C++, its application layer is nothing but C++ classes. Multi-threaded programming in C++ class is a little different from normal POSIX thread programming using C.

To create a thread, a function must be specified to be the entry point for the thread. In C programming the entry point is just a normal function; but in C++ the entry point can not be a normal member function of a class, it can be a static member function of the class.

In NS-2, an application is started in TCL script as:

```
$ns at 0.1 “$application start”
```

When TCL script starts an application, it is actually invoking the start() method of the application. Supposing the application is an object of the proposed custom application class of NS-2, the start() will pop up a new thread to serve the application and assign a static method to be the entry point of the new thread. Actually, all the serving threads are using the same entry point. Because the entry point is a static method, it cannot have access to instance fields such as the inner buffer of the application, these instance fields must be accessed in an instance method. A little trick has been done here to solve this problem.

The static method (entry point for all threads) takes one argument of type “void

*", each time when a thread is started, the method is invoked by passing it the pointer "this", the pointer to current instance of proposed simulator application class. An instance method is then assigned to serve current thread like:

```
This -> some_instance_method();
```

The instance member function finally acts as an accessing point of NS-2 for real world applications, the section that follows explains how the custom simulator application interact with RWA and NSC.

4.3 Interacting with Real World Application and NSC

Because real world application and NS-2 run in different address space, there must be Inter-Process Communication (IPC) mechanism for them to share information. The UNIX-like systems support lots of IPC methods such as pipe, named pipe, socket, message queue, semaphore and shared memory. All the methods have advantages and disadvantages of their own. BSD Socket API for Simulator is supposed to run in a distributed test environment, socket is chosen to be the IPC mechanism because it is the only one to make messages passed across different platforms. Socket is not an efficient mechanism, if efficiency is favored, other IPC mechanism such as shared memory can use used instead.

When using socket inside NS-2, there are several compiling errors due to symbol names conflicting. That means socket API cannot be used directly inside NS-2, so another way is taken here, the "GNU C Library" (Glibc) shared library is loaded dynamically into the address space of simulator.

The Glibc is GNU's C standard library. It is free software and is available under the GNU Lesser General Public License. Glibc is used in systems which run many different kernels and different hardware architectures. Its most common use is in Linux systems on x86 hardware, but officially supported hardware includes: x86, Motorola 680x0, DEC Alpha, PowerPC, ARM, ETRAX CRIS, MIPS, s390, and SPARC. It officially supports Linux kernels.

The implementation of simulator application simply makes use of the shared library provided by the host system to interact with the outer world. The interaction has two aspects: receiving message from RWA and push it into NSC; receiving message from NSC and sending messages to RWA.

4.3.1 Receiving from Outer World

As all the other simulations using NS-2, the proposed simulator application must

connect to the underlying agent (NSC) in some way to allow messages being passed to the agent. The connection is done through TCL script like:

```
$application attach-agent $agent
```

Doing so, a pointer “agent_” inside application is set to point to the attached agent and a pointer “app_” inside agent is set to point to the connected application. The pointers are used for the proposed simulator application and agent to exchange message as can be seen later.

Like mentioned in the previous section, the instance method is used to receive message from RWA and relay the received message down to NSC. The method achieves this by dynamically loading Glibc to obtain various function pointers of socket-related system calls. An UDP socket is then created to exchange information with RWA. The instance method is actually a simple loop; the loop consists of three parts.

(1) Firstly, the instance method calls Glibc’s recvfrom() system call, it blocks the execution of current thread until message is received.
(2) Secondly, when a message is received, it is put into an inner buffer. The inner buffer will keep the message until the other side of application pair fetches it. To prevent the message from overwriting before it is fetched, MUTEX is used to guarantee the delivery of the message.
(3) Thirdly, the number of byte of payload in the received message is then passed to NSC by calling NSC’s method sendmsg() like:

```
Agent_ -> sendmsg(NUM_MESSAGE);
```

The effect of the line above is sending message down to transport layer of NS-2, the NSC.

4.3.2 Sending to Outer World

After traversing through NS-2, message is finally handed to the other side of an application pair by NSC; the following line is adapted from source code of NSC which shows how NSC passes a message up to the connected application layer.

```
app_->recv(buflen);
```

The "app_", as explained in last section, is a pointer to the application layer connected to current NSC agent. Application layer's " recv()" method is called with the number of message in byte as the only parameter. Therefore we are going to process the message in method recv() of the proposed simulator application class.

The recv() method fetches the original message from its partner's inner buffer and releases its partner's MUTEX so that the partner can receive further messages.

There exists a small application level header in the original message, this header is added by shared library, it contains information about where the destination of current message is. The structure and functionality of the header will be explained in next chapter. The recv() method then extracts the header from the message received.

A UDP socket is created with its destination set to what we got from the header by calling Glibc's corresponding system calls such as socket() and bind(). The original message is sent to real world application by calling Glibc's "sendto()".

4.4 Establishing Application Pair

As explained in the previous chapter, simulator applications exist in pair and are connected in some way. Application pair is especially useful for simulators that do not actually transfer real messages such as NS-2. In BSD Socket API for Simulator, application pairs are built up through TCL script as:

```
$application1 connect $application2
```

The TCL code issues a "connect" command to establish bi-directional connection between two instances of the proposed simulator application. The TCL command "connect" is actually running part of the method command () of the proposed simulator application class. The command () provides TCL with various instructions that can be used when building up the simulation infrastructure, it is a communication mechanism between TCL script and C++ objects. There are several commands have been implemented inside command () including "connect".

The command "connect" tries to find "$application1" and "$application2" among TCL objects. If they are found, pointers to the two applications are obtained; otherwise, a TCL error is returned.

"target_" is a member of the proposed simulator application class; it is a pointer of the type of the proposed application class. The member is used to store the address of partner, namely, "target_" of "$application1" will be set to the address of "$application2" and vice versa. By this way, a bidirectional connection is established; the two partners can communicate through this connection. Although NS-2 carries only the number of bytes, partners can make use of the connection to exchange real messages.

A simulation generally consists of multiple application pairs. The information about these pair is stored in a table.

4.5 Application Pair Table and Load Balancing

Application Pair Table is created in shared memory. When the first instance of the proposed simulator application class is initialized, a shared memory is created and initialized. The memory is shared among all the instances of the proposed simulator application class. Each time an application is started, its information is stored inside the table. The structure of the table entry is as follows.

.....	.....	.....	.....
Port-I	Counter-I	Port-II	Counter-II
.....	.....	.....	.....

Figure 4-1 Structure of application pair table

“Port-I” and “Port-II” record the port numbers on which the two RWAs are listening, there is a counter for each port. When a RWA is started, a port number is chosen from the table and returned to the RWA, so that the messages from outer world can be sent to these ports and redirected to NS-2. The counters record the number of real world applications that are using the corresponding port, so that work load can be distributed fairly among multiple application pairs, which is called load balancing in BSD Socket API for Simulator.

Load balancing is managed by a special thread in the proposed simulator application class. Not surprisingly, it is called load balancing thread. This unique load balancing thread is started at the same time when the application pair table is constructed. The thread listens on a special port so that it can accept the requests from BSD Socket API shared libraries. Whenever a BSD Socket API shared library is started, it sends a request to the load balancing thread. The thread looks up application pair table, chooses one entry based on some rules and returns the port number to the requesting shared library. The rules are listed as follows. This is an ordered list, and any rule has higher priority over the rules that follow it.

1. If any application's reference counter is zero, it is return immediately.
2. If two reference counters in an application pair are not equal, return the one which is least referenced.
3. Return the least referenced application, if multiple such applications exist, return the first one on the list.

These rules guarantee that the overall work loads are distributed fairly among application pairs.

4.6 Interacting with TCL

TCL, the Tool Command Language, is an interpreted scripting language. It is used by the NS-2 network simulator to build simulation scenarios. The interaction between TCL script and C++ object is a critical part of NS-2.

In TCL script, there are two kinds of such interactions: you can change the values of C++ class member variables or invoke “command” implemented in the method command () to affect the behavior of C++ object. This mechanism makes simulation flexible and powerful. The proposed simulator application also provides such functionality to facilitate creating simulation environment.

As each of the proposed simulator application has its own listening port number to receive messages from real world applications, and this port number can not be hard-coded, so the TCL script is the best place to set such parameter. Actually, it is very easy to do it in TCL by just a single line of instruction. The following line, for example, shows how to set the listening port number of an application to 4000.

```
$application set port_ 4000
```

Another important action a simulator application must take before running a simulation is to bind itself with an agent. Because both the agents and applications are created in TCL script, so they are bound together in the script as well. The "binding" action has been implemented in the "command ()" member function, a single line like "$app1 attach-agent $NSC_agents(0)" will do the trick.

The proposed simulator application class exposes several other member variables and commands.

4.7 Summary

A multi-threaded simulator application layer is necessary to run unmodified binaries of real world applications especially when dealing with blocked calls.

The proposed simulator application interacts with RWA by dynamically loading Glibc and interact with NSC by calling its interface methods, sendmsg () and recv().

The proposed simulator applications exist in pair to facilitate transmitting real message. Application pair table records information about the pairs and provides load balancing.

As usual, the proposed application provides variables and commands that can be used by the TCL script.

Chapter 5
Shared Library

The BSD Socket API for Simulator shared library is pre-loaded with real world application (RWA) by setting environment variable LD_PRELOAD. The shared library catches socket-related calls issued by RWA and processes them in some way. The shared library and simulator application have two things in common:

1. GNU C Library is loaded, and Glibc functions are used to communicate with simulator.
2. Socket is used for inter-process communication.

In the context of the BSD Socket API for Simulator, each call has three forms. Take socket() as an example, RWA issues socket () to get a socket descriptor; the RWA socket () is then caught by the shared library socket () function immediately; the shared library invokes Glibc socket () function to get a socket and finally, return the socket descriptor to RWA. For clarity, these are called RWA socket (), shared library socket () and Glibc socket (), respectively.

The shared library is built on the concept of “message redirecting”. The intended destination of messages sent by a RWA is to the other end of a connection or to a specified “name”. The destination is kept in an application level header and redirected to simulator together with the original message. The original message will be called “payload” throughout this document.

In socket world, “name” is actually a data structure describing address and port number, which can uniquely identify a position on network. Therefore, the term “name” in the context of socket programming really means "address and port". The man pages of socket-related system calls use both "name" and "address" interchangeably.

At receiving side, the shared library accepts messages from simulator only. Whenever a message is received, it is processed and payload is passed to the intended RWA. For the RWA, the message comes as if it is from the sender directly. The BSD Socket API shared libraries are transparent for RWAs.

There are 29 functions are implemented in the BSD Socket API for Simulator Shared Library. This chapter divides these functions into several categories:

- Functions for connection establishment.
- Functions for sending and receiving.
- Functions for closing sockets.
- Functions for I/O multiplexing
- Functions for duplicating socket
- Functions for socket options

The sections that follow will be introducing each category in detail as well as presenting how multi-clients server is supported by the shared library.

All the 29 functions are related to their counterparts in Glibc in some way, thus it is unavoidable to concern socket programming when presenting implementation of the functions, but this chapter has no intention to be a detailed socket programming tutorial, so the material on socket programming is kept as brief as possible.

5.1 API Header

In the shared library, a message will travel through three processes: the sending RWA, simulator and receiving RWA. To make the message redirecting mechanism work, an application level header is introduced.

The message redirecting mechanism is illustrated in figure 3-3. The arrows in the figure show how messages flow between two RWAs via NS-2. Message sent by source RWA is captured by shared library and then it is redirected to NS2. A problem arises when the message reaches SimApp on the right hand side, where to send this message next? The only one who can answer this question is the source RWA. The destination can be found in its socket (i.e. TCP connected sockets) or data structures inside the sending system calls (i.e. structure sockaddr in sendto()). Due to message redirecting, a different socket or data structure might be used by shared library and therefore the original destination information was lost.

To keep the destination information when redirecting messages, the shared library pre-loaded with the sending RWA is responsible for appending an API Header to each of the messages it redirects. The API Header tells each of the nodes on the message redirecting path where is the final the destination of the current message. The API Header is defined as follows.

Source Address	Source Port
Destination Address	Destination Port
Socket Type	

Figure 5-1 API header

There are five fields in the definition of API Header: source address, source port, destination address, destination port and socket type, these names are self-descriptive.

The API header is added and striped off by shared library. It is added at the sending side and stripped off at the receiving side. When a simulator application receives message (the number of bytes) from NSC, it will take the following three actions:

- Fetching real message from the internal buffer of its partner.
- Obtaining destination address and port number from API header.
- Redirecting the real message to its destination - the receiving RWA.

Subsequently, the message is sent out of NS2 to its destination.

5.2 File Descriptor Table (FDT)

Besides API Header, another interesting thing to note is that different sockets might be used in message redirecting. The one used by RWA and the one used by shared library might be different. It demands a mechanism to link the two sockets together, so that when RWA sends message using its own socket, the shared library will map that RWA socket to another socket which can send message to simulator, this is achieved in a socket descriptor mapping table - File Descriptor Table (FDT).

Each entry of the table maps a real world application socket to a socket, which sends message to simulator. The former is called real socket or RWA socket, while the latter is called simulator socket. Socket descriptor is nothing but file descriptor; they occupy the same "number space". Thus, the socket descriptor mapping table is named, File Descriptor Table (FDT). FDT entries have structure as the following figure.

Real FD	Simulator FD	Type	How
Local Address			
Destination Address			
Local Port	Destination Port		

Figure 5-2 FDT table entry

- Real FD is real File Descriptor (RWA socket). This is the socket descriptor used by RWA.
- Simulator FD is simulator File Descriptor or simulator socket. It is invisible to RWA. The simulator FD is used by shared library to redirect information to network simulator. Because the simulator is supposed to work under highly reliable environment, to achieve high efficiency, simulator FD is of type "SOCK_DGRAM" which is based on the unreliable transport layer protocol UDP. When real world socket is of type SOCK_DGRAM, simulator FD is identical to Real FD; otherwise, a new UDP socket has to be created and thus the simulator FD and Real FD are different.
- Local address and local port store local host IP address and the port number of the RWA socket.
- If current real socket is connecting to a remote socket (by issuing Glibc accept () or Glibc connect ()), the IP address and port number of the remote socket are stored into fields “Destination Address and “Destination Port”.
- Type is the socket type of Real FD.
- The field “How” is used for closing sockets only. It represents the current state of a socket; it can be one the three socket status, SHUT_RD, SHUT_WR or SHUT_RDWR.

The mapping table is core to the shared library, all the functions in shared library need to look up or modify this FDT as can be seen later.

The verb "register" is used when information need to be stored into FDT. For example, "registering real socket" means that real socket is stored into FDT.

5.3 Initialization and Cleanup of the Shared Library

The shared library needs to do some preparation before any of its socket-related functions can be invoked. The initialization routine of the shared library has two major tasks:

1. The initialization routine registers current shared library with load balancing thread (see section 4.5 “Application Pair Table and Load Balancing”) and establishes connection with a simulator application. The shared library will redirect messages to simulator through the connection.
2. The initialization routine loads “GNU C Library” and acquires function pointers to Glibc’s socket related functions. These function pointers will be used for the shared library to communicate with simulator.

The destructor of the shared library simply de-allocates memory space to avoid memory leak.

The shared library exports initialization and cleanup routines using two gcc function attributes [GCC 4.1.1 Manual, Richard M Stallman and the GCC developer community]:

- __attribute__((constructor))
- __attribute__((destructor))

The shared libraries are loaded into memory by dlopen() system call and closed by dlclose() system call. Constructor routine is executed when the shared library is loading into memory and before dlopen() returns. Destructor routine is executed when the shared library is closing and before dlclose() returns.

5.4 Establishment of TCP and UDP Connection

The introduction to establishment of TCP and UDP connections uses the client/server model. In both TCP and UDP connections, two processes, which act as server and client respectively, will be communicating with each other in a connected manner. There are three key points for both TCP and UDP connections, namely:

- Client must know the name (address and port number) of server before a connection can be established.
- Server does not need to know the name of client before a connection is established.
- Both server and client can send and receive information.

The processes of establishing connection are somewhat different between TCP and UDP, but both involve several same system calls such as socket (), bind () and connect (), the three system calls handle the differences between TCP and UDP. Besides the previous three system calls TCP connection needs two more

system calls: listen () and accept (), which can not be used to establish UDP connection.

5.4.1 TCP Connection

The section introduces the process of establishing TCP connection without using the shared library and the processes of establishing TCP client and server when the shared libraries are pre-loaded.

5.4.1.1 Establishment of TCP Connection without Shared Library

The following figure shows the process of establishing TCP connection on both client and server and how they interact with each other.

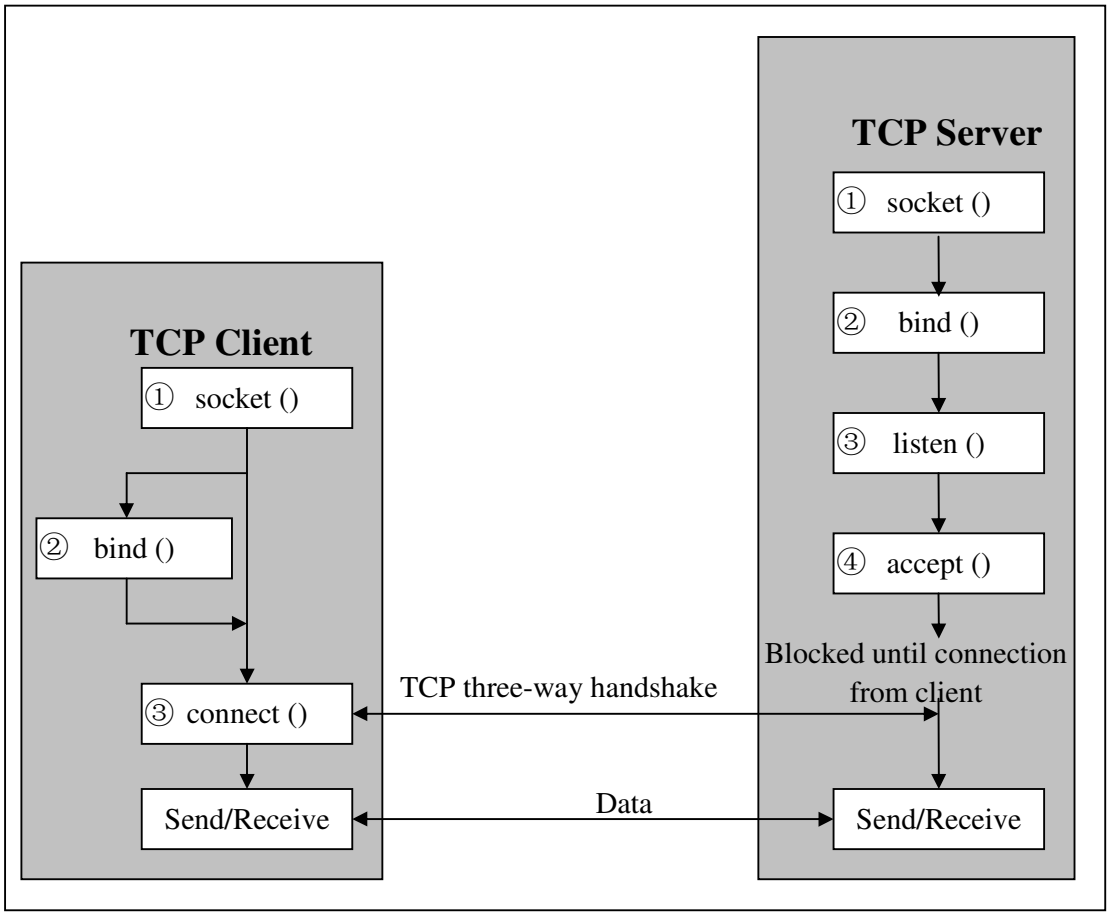

Figure 5-3 Establishment of TCP connection

The following three steps are needed to establish a TCP client.

Step 1: Create a socket with the socket () system call.

Step 2: This is an optional step, the socket created at last step is bound to a name with bind () system call. If client does not bind socket to a specific port, an available port number is chosen by kernel and bound to the socket. The bind() system call is usually not called by most of network clients.
Step 3: Connect the socket to server using connect () system call.

Further, four more steps are needed to establish a TCP server.

Step 1: Create a socket with the socket () system call
Step 2: Bind the socket to a name using the bind () system call. Server must specify address and port number so that clients can connect to it, the name must be known for all clients.
Step 3: Listen for connections with the listen () system call
Step 4: Accept a connection with accept () system call.

The interaction occurs between step 3 of client and step 4 and server, server blocks until client connects to the server. A two way connection is established eventually; both server and client can send and receive information on this connection.

5.4.1.2 Establishment of TCP Client Using Shared Library

When shared library is pre-loaded with the RWA, the process introduced in last section is changed due to the existence of FDT. Each RWA call needs to interact with FDT in some way: adding an entry to FDT or modify fields of a specific FDT entry. The optional bind () causes a little complexity in the process.

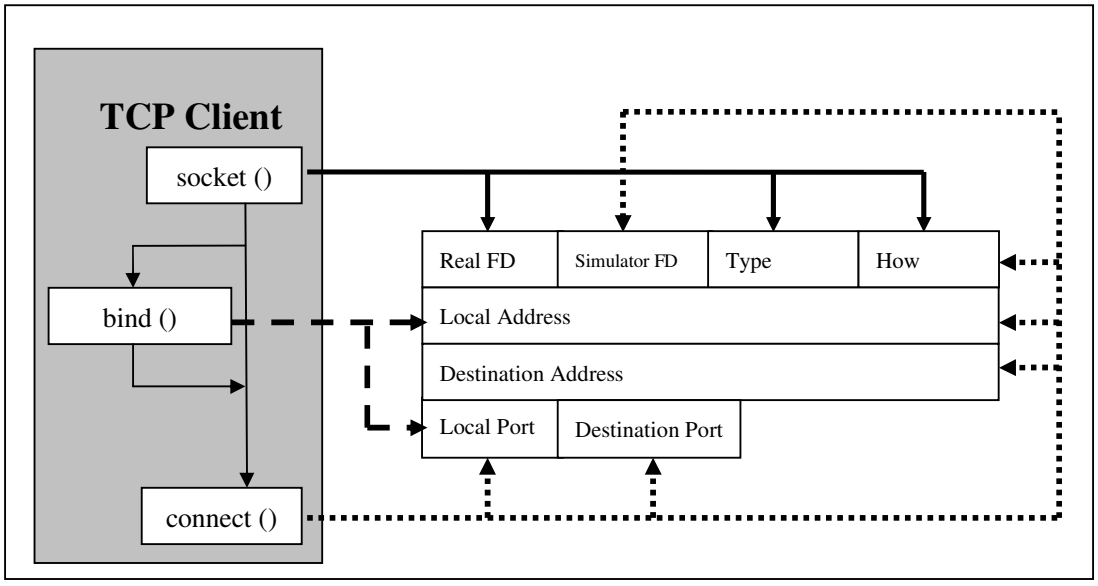

Figure 5-4 Establishment of TCP client using shared library

The shared library socket () creates a new entry in FDT for each RWA socket. Shared library socket () invokes Glibc socket () first; if the call returns successfully, a RWA socket descriptor is obtained by the shared library. The shared library then adds and initializes a new FDT table entry as follows:

- “Real FD" is registered using the socket descriptor returned by Glibc socket () call.
- "Type" is registered using the second parameter to RWA socket () call.
- "How" is initialized to be CLOSED, that means current RWA socket is in the CLOSED state.
- All the other fields keep their initial states after calling the shared library socket () call.

The optional bind () gives a name (address and port number) to RWA socket and register local port and local address in FDT. The shared library bind () checks the second argument to RWA bind () call in the first place, a data structure of type “struct sockaddr". If the port number inside the data structure is 0, it means that RWA leaves kernel to choose an available port number for it. The shared library bind () function is doing the same thing: a port is chosen randomly between 1024 and 65535; and Glibc bind () is then invoked on RWA socket. If the port number in the data structure is not 0, it means that RWA has assigned a port number for its bind () call. The shared library bind () just passes the call to Glibc directly. If the Glibc bind () call succeeds, a port number and the local host IP address have been effectively bound to RWA socket. Local port is registered using the port number either chosen randomly or assigned by RWA; and local address is registered using local host IP address.

The shared library connect() function connects client to server. In TCP connections, connect () and accept () appear in pair. As a result, both sides are connected tightly. Generally, at client side, the bind () call is not invoked before connect (), so the shared library connect () need to check if the current RWA socket has been bound to a name. If not, a port is chosen randomly between 1024 and 65535; and Glibc bind () is invoked to bind the chosen port and local host IP address to the RWA socket. At last, local address and local port is registered in FDT.

The RWA connect () uses a data structure “struct sockaddr *serv_addr” to indicate the IP address and port number of server. In the data structure, server address might have several forms:

- INADDR_ANY or loop back address, the server is on the same local host.
- Normal address, the server is on a remote host.

The shared library connect () examines the data structure provided by RWA connect () first. If the address is INADDR_ANY or loop back address, the server IP address in the data structure is replaced with local host IP address. After processing the data structure, the shared library connect () function invokes Glibc connect () to actually connect to server. If the Glibc connect () succeeds, "Destination Address" is registered using the server IP address found in "serv_addr".

In the data structure, there exists a server listening port number, but the listening port number cannot be used to register "Destination Port". The interaction between connect () and accept () will result in a new socket on server side. The new server side socket connects to client. It is bound to a new port other than the listening port (see the next section for more details). The problem here is that the client side has not known of the new port number. Therefore, on the server side, the shared library accept () must send the new port number over the RWA socket to client side. Accordingly, the shared library connect () must receive the port number via its own RWA socket. Subsequently, the client knows where to redirect messages.

Another action the shared library connect () takes after calling the Glibc connect () function is that the "How" field of FDT entry will be change from CLOSED to ESTABLISHMENT.

Thus far, the only field in FDT entry yet to be initialized is the simulator FD. As explained above, a UDP socket is created, and his new socket is bound to the same address and port number as that of RWA socket (SOCK_STREAM or SOCKET_SEQPACKET). A TCP socket and a UDP socket can bind to the same port number without confliction. The simulator FD is registered using the newly-created UDP socket.

5.4.1.3 Establishment of TCP server Using Shared Library

Establishing a TCP server involves four functions: socket (), bind (), listen () and accept (). The first two functions are almost identical to those of establishing TCP clients. The only difference is that the bind () on server side is compulsory. The name of server must be known to all clients before any connection can be built. Therefore, only the last two functions, listen () and accept () will be discussed in this section. Figure 5-5 shows interactions between functions and FDT entry when establishing TCP server.

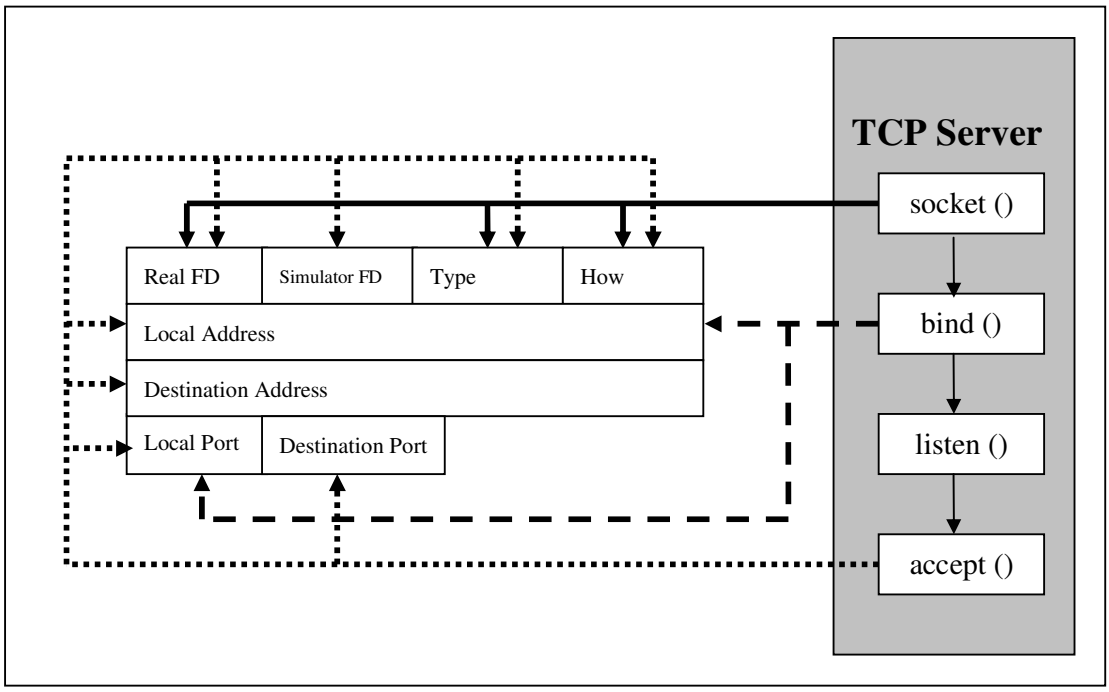

Figure 5-5 Establishment of TCP server using shared library

The shared library listen () and accept () are used for connection-based socket only. The listen () is called after both socket () and bind () have been issued and must before calling accept (); accept () runs in an environment which has been set up by listen (). The two functions are tightly coupled.

In the shared library listen () function, RWA listen () will be passed directly to Glibc listen() function. The Glibc listen () is actually doing two things: firstly, it moves RWA socket from the CLOSE state to the LISTEN state so that accept () can be called later; secondly, it specifies the maximum number of outstanding connection requests in listen () input queue. The shared library listen () has no interaction with FDT entry at all.

The RWA accept () is passed to Glibc accept () by the shared library accept (). The Glibc accept () fetches the next available connection from LISTEN queue and returns a new socket descriptor automatically generated by kernel. The new socket refers to the connection to client.

As the call to Glibc accept () results in a new RWA socket and, therefore, a new entry is added to FDT. The new entry is initialized as follows:

- The “Real FD” is registered using the socket descriptor returned by Glibc accept ().
- The “Type” is registered with the type of the listening socket.
- The “How” is registered as ESTABLISHMENT.

- The “Local address” is registered with local host IP address.
- “Destination address” and “destination port” can be found in the data structure returned by Glibc accept ().

From the list above, there are two fields in the new FDT entry, namely, “Local port” and “Simulator FD”, yet to be given a meaningful value. The initialization of the two fields on client side is simple: a UDP socket is created as “Simulator FD”, the socket is bound to the same port as that used by RWA socket (TCP socket). However, on the server side the simple scheme does not work. Because a TCP server may accept multiple clients, this can be achieved by I/O multiplexing (by select () or poll ()), multi-processing (by fork () function) or multi-threading. After creating a UDP simulator socket for each of the new TCP sockets, a problem arises: the simulator sockets cannot be bound to the same listening port. Thus, the solution to the problem is: a random port number is chosen for each simulator socket and the port number is returned to the connected client. Client will use it to register its destination port (please refer to previous section for more details).

5.4.2 UDP Connection

This section introduces the general process of establishing UDP connection and then followed by the process of establishing UDP connection using the shared library.

In the processes, the term "connection" has different meaning compared to that in TCP. The UDP server and client are not actually connected, although both server and client can issue connect () system call, the only effect of these connect () calls is to set the destination address and destination port number. There is no interaction between the UDP server and client when establishing UDP connection.

5.4.2.1 Establishment of UDP connection without the shared library

Establishing UDP server and client involve the same functions: socket (), bind () and connect () as well as the similar processes. The following figure shows the process of establishing UDP connection on both client and server.

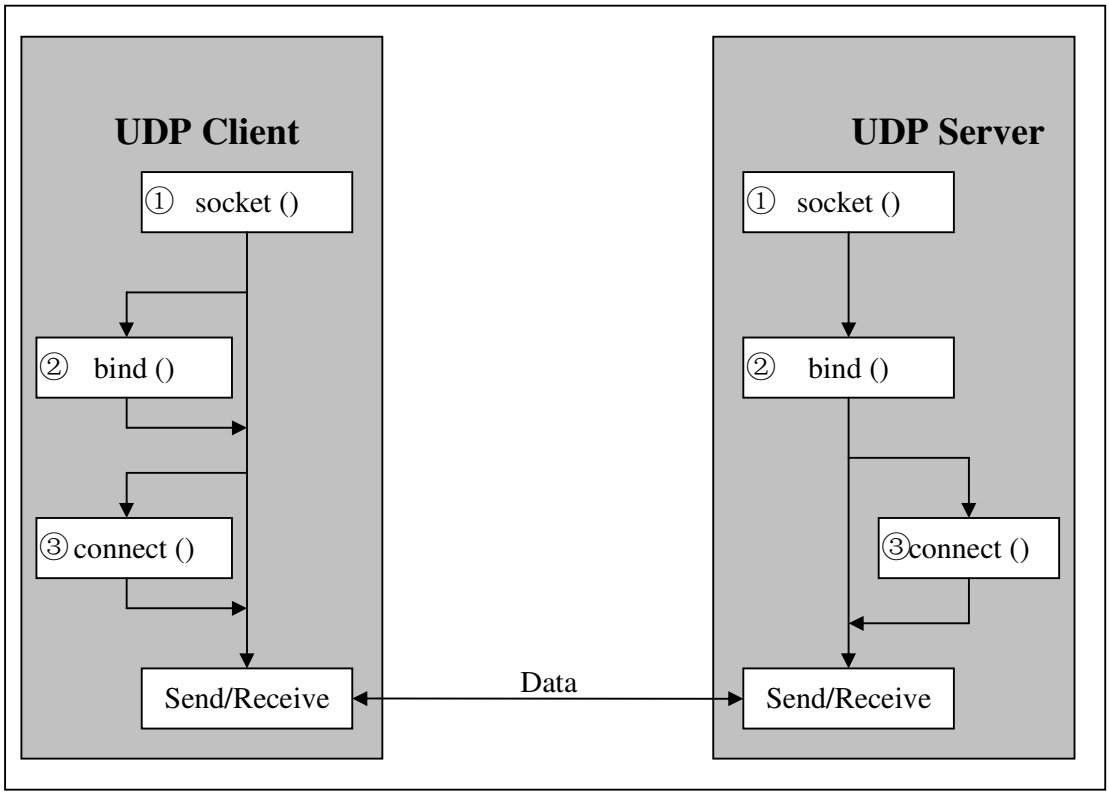

Figure 5-6 Establishment of UDP connection

One or three steps of the following are needed to establish UDP client:

Step 1: Create a socket with the socket () system call.
Step 2: Bind the socket to an address using the bind () system call. This is an optional step and is usually omitted by most of UDP clients.
Step 3: Connect the socket to a specific name using connect () system call. This is an optional step as well. The step merely specifies a name to which datagrams are sent by default, and the only name from which datagrams are received. This step is necessary only when the UDP client tries to use send ()/recv () instead of sendto()/recvfrom() to send or receive messages.

Similarly, either two or three steps are needed to establish UDP server:

Step 1: Create a socket with the socket () system call.
Step 2: Bind the socket to an address using the bind () system call. Unlike step 2 of establishing UDP client, this step is not optional, becuause a server's name (address and port) has to be known by all clients before connection can be established.
Step 3: Connect the socket to a specific name using connect () system call. This is an optional step. The step merely specifies a name to which

datagrams are sent by default, and the only address from which datagrams are received. Unlike TCP connection, a UDP server can also "connect to" a UDP client. Its effect is completely the same as step 3 of establishing UDP client. But there is a side effect when doing so, the server can communicate only with the client specified. In most cases this is undesirable.

Establishing UDP client and UDP server are quite similar. The UDP server is similar to the UDP client except that the UDP client has a known name.

5.4.2.2 Establishment of UDP Client Using Shared Library

The shared library socket () behaves a bit differently from how it does in TCP connection. Both the bind () and connect () on client side are optional, and a “simulator FD” is necessary so that messages can be exchanged between the shared library and simulator. Thus, the shared library socket () is the only place to create a simulator FD. In shared library socket (), “Real FD”, “Type” and “How” are initialized as they are in TCP connection. The initialization of “simulator FD” is actually very simple: because UDP socket is used in the shared library to communicate with simulator, we simply use the “Real FD” as the “Simulator FD”.

There is one problem for such design: when the optional connect () is issued on the “Real FD”, the “Real FD” will be “connected” to a remote server. A UDP connection means that messages can only be sent to or received from the connected server. This is undesirable in BSD Socket API for simulator, because messages will be redirected between the shared library and simulator.

The solution to this problem is: when the RWA connect () is issued, the shared library connect () catches this call and checks the type of RWA socket. If it is UDP socket, the Glibc connect () will not be called. The destination address and destination port found in RWA connect () are used to registered “Destination Address” and “Destination Port” in FDT entry. The operation has two consequences:

1. The shared library knows of the destination of subsequent messages. The destination information can be put into an application level header, so that messages can be redirected.
2. The RWA sockets on both server side and client side are not actually connected. The “Real FD” can be used to redirect information, there is no need to create a different “Simulator FD” and thus the file descriptor number space is saved.

In the FDT table entry, there are four fields may not be initialized: “Local

Address", "Local Port", "Destination Address" and "Destination Port". "Local Address" and "Local Port" are registered in shared library bind (). "Destination Address" and "Destination Port" are registered in shared library connect (). Whether these fields are filled depends on what functions are used to send/receive messages. (Please refer to section 5.5 "Sending and receiving" for more details.).

The interaction between UDP client and FDT entry is shown in figure 5-7.

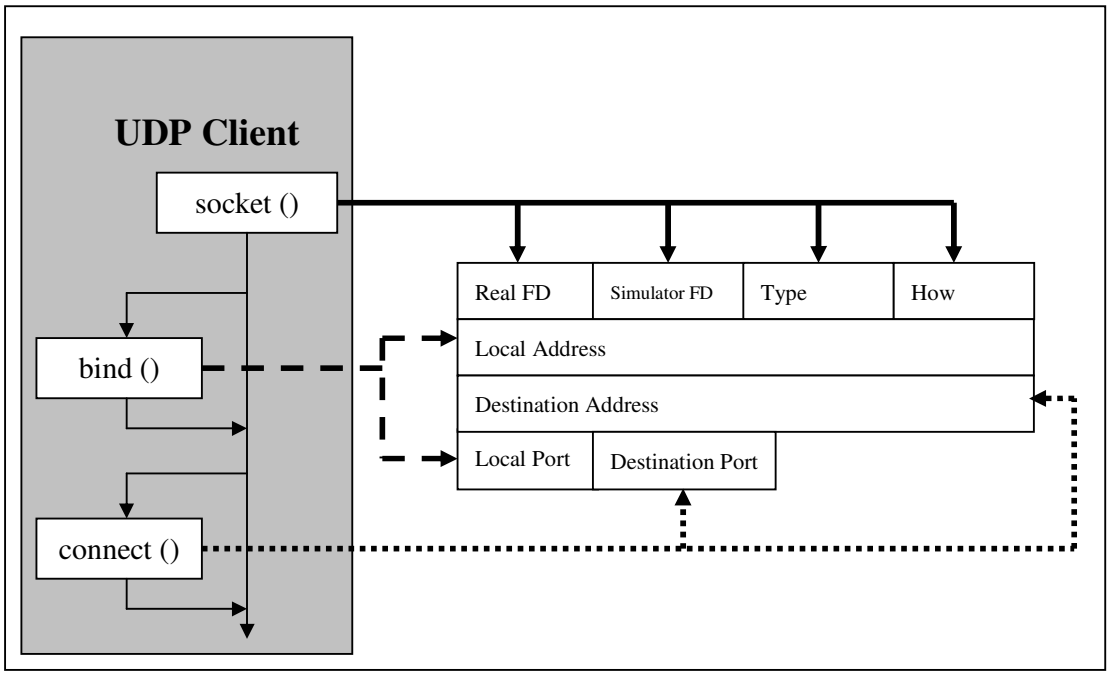

Figure 5-7 Establishment of UDP client using shared library

5.4.2.3 Establishment of UDP Server Using Shared Library

The establishment of UDP server is nearly identical to establishing UDP client except that the bind () is compulsory on server but optional on client. The process is shown in figure 5-8.

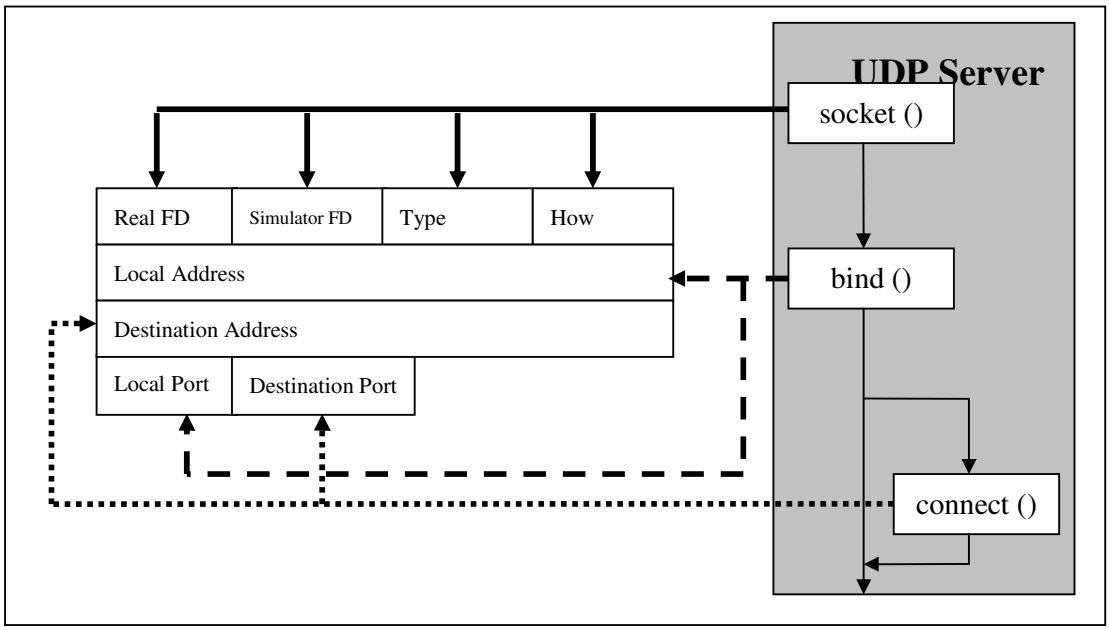

Figure 5-8 Establishment of UDP server using shared library

5.5 Socket pair

The socketpair () is an easy way of establishing connection between a pair of sockets. The call to socketpair () creates two connected sockets, so socketpair() function results in two entries in FDT.

The implementation of the shared library socketpair () function is straightforward. The function first invokes the Glibc socketpair () call. If it returns successfully, two entries are added to the FDT and two RWA sockets are linked with each other. Figure 5.9 shows how the socket pair connection is established in the shared library.

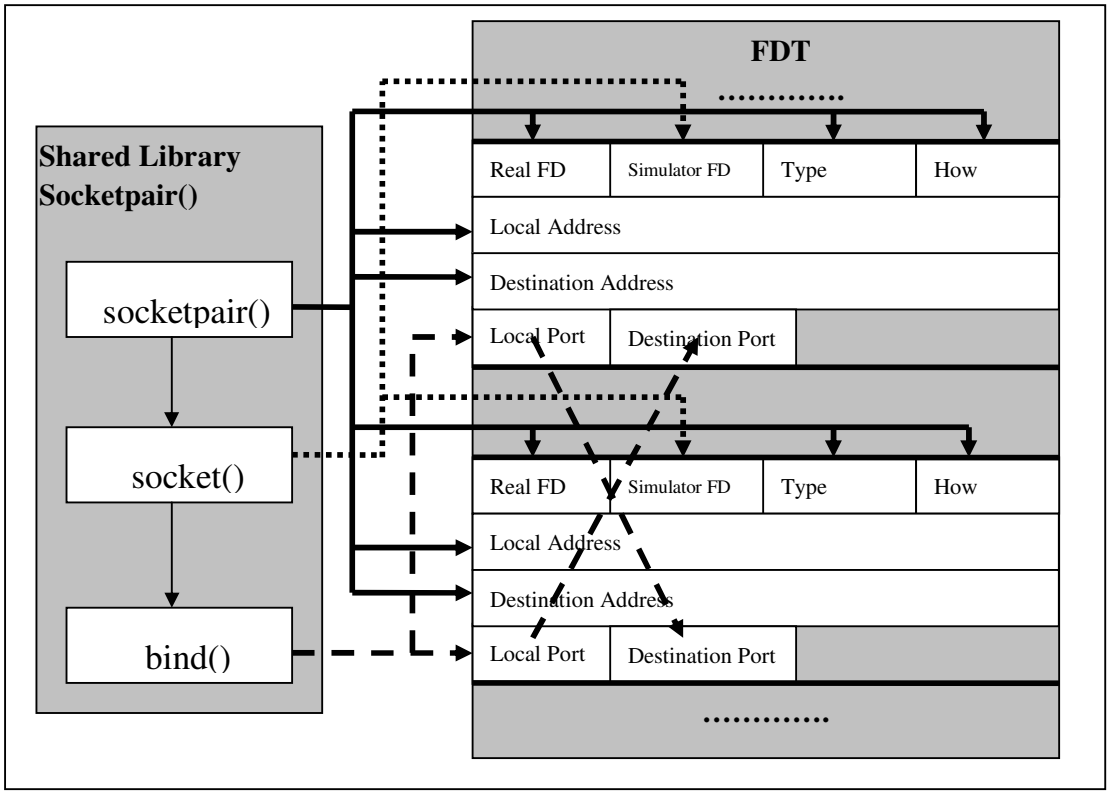

Figure 5.9 Connecting sockets using socketpair ()

The shared library socketpair() function involves three Glibc functions: socketpair(), socket() and bind(). After each of the Glibc functions returns successfully, some fields of the two new FDT entries are registered.

The call to Glibc socketpair() creates two sockets, they are then used to register “Real FD” of the two new entries. Because the two new sockets must be on the same host, so the “Local Address” and “Destination Address” of the two new FDT entries are registered using the address of local machine. “Type” and “How”

are registered as they are in the implementation of the shared library socket () function.

The Glibc socket () is called twice to create two UDP sockets for the two connected RWA sockets. The UDP sockets are then used to register “Simulator FD” of the two new FDT entries.

Two ports are chosen and bound to the two UDP sockets. Local Ports of the two new FDT entries are registered using the two port numbers. Destination ports of the two new FDT entries are registered using the local port numbers of the other ends. Bi-directional connection is eventually set up through simulator FDs.

5.6 Sending and Receiving

Sending and receiving information is the most interesting part of this project, whereas, it is actually incredibly straightforward. The sending or receiving call issued by RWA is captured by the shared library; a FDT table entry is then located via the RWA socket contained in the RWA call. If the RWA is sending, a small application level header is appended to the original message and the new message is sent to simulator through the simulator FD in the FDT entry. If the RWA call is receiving, the shared library receives message from simulator through the simulator FD in FDT entry and payload is then extracted from the received message by stripping off application level header; the payload is handed to the receiving RWA.

The process above is relatively abstract, as there exist multiple sending/receiving pairs, the implementation of these functions are actually slightly different.

There are five pairs of functions involving sending/receiving messages: send()/recv(), sendto()/recvfrom(), read()/write(), readv()/writev() and sendmsg()/recvmsg(). These functions are categorized as two families: sending family and receiving family. In BSD Socket API for Simulator, the implementations of functions in each family are similar in some way, so this section chooses one function from each family, send() and recv(), to demonstrate in depth.

5.6.1 Sending

As can be seen from previous sections, messages take several steps from the sending RWA to the receiving RWA: sending RWA delivers message to the shared library; the shared library redirects received message to simulator; message traverses simulator; simulator delivers message to the shared library; the

shared library delivers message to the receiving RWA.

In the process above, sender's shared library will add an application level header to each of the messages it transmits, the header will be checked at some later stage by simulator application before redirecting message to the receiving RWA.

In BSD socket API, the send () system call may be used only when socket is connected. In a connected state, the intended recipient is known. Both TCP sockets and UDP sockets can be connected, but TCP connection and UDP connection are different. This difference is reflected in the FDT entry. Because the difference has been dealt with in the connection setup stage, so the shared library send () call does not care about the type of the RWA socket.

The implementation of the shared library send () call is illustrated in figure 5-10.

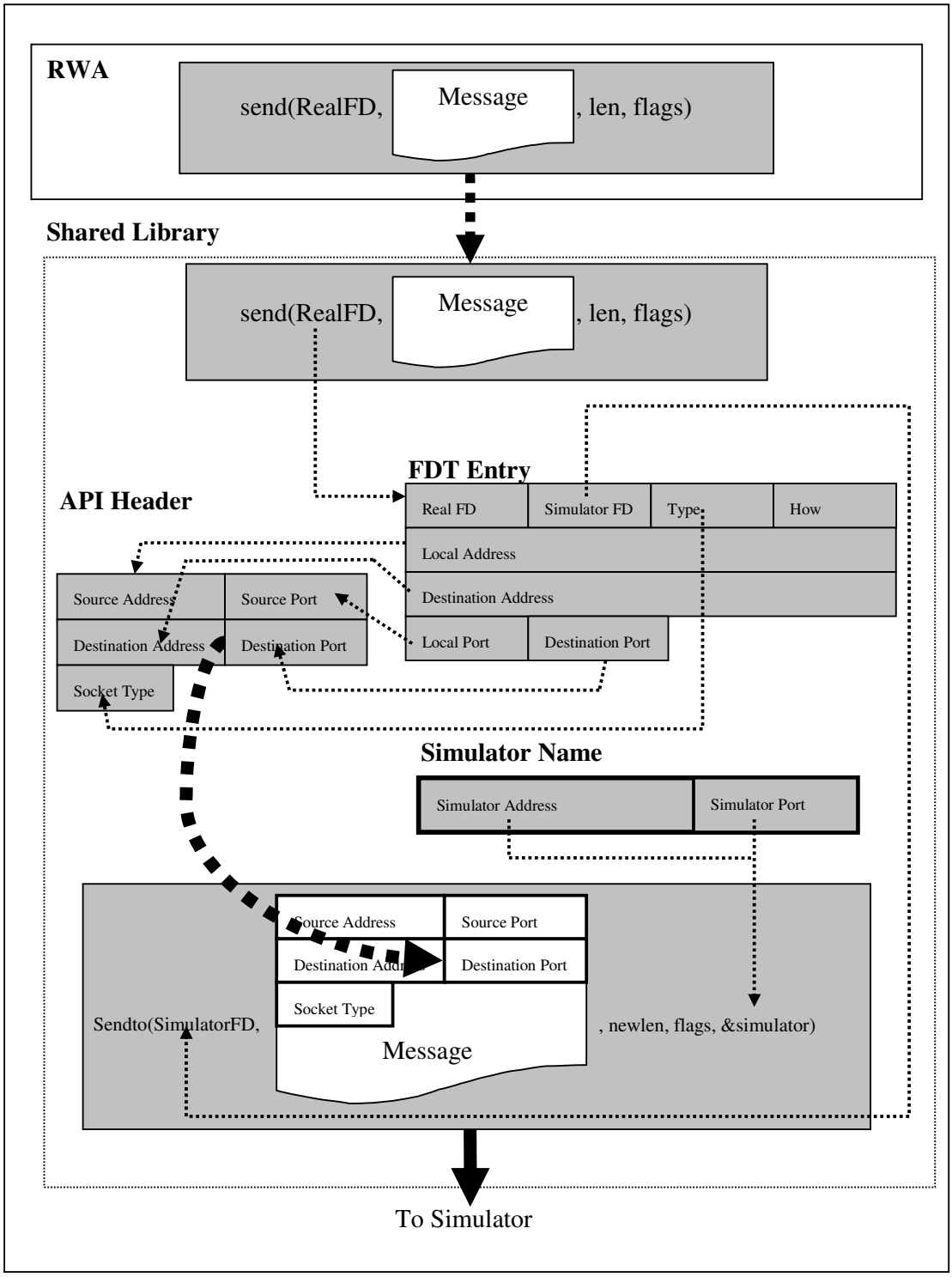

Figure 5-10 Sending message in the shared library

The shared library catches RWA send () call in the first place; the shared library then looks up the FDT using RWA socket found in the RWA send () call. As long as an entry is found, an API header is constructed by extracting the following fields from the found FDT entry: local address, local port, destination address,

destination port and socket type. The header is then appended to the payload sent by RWA to form a new message.

The new message is sent to simulator by calling Glibc sendto () function using simulator FD in the found FDT entry. The destination of the new message is the one that have been obtained when initializing the shared library (see section 5.3 “Initialization and cleanup of the shared library”).

As mentioned at the beginning of the section, there are other sending family functions that are implemented in the shared library: sendto(), write(), writev() and sendmsg(). All these functions take similar way as above, so they are not elaborated here. The paragraphs that follow introduce the differences between these functions and the send () call.

- The sendto() function, unlike the send () call, can be used with both connected-mode and unconnected-mode. When it is used with connected-mode, its implementation is almost identical to that of send (). When it is used with unconnected mode, the name of the destination can be found in the argument list to the RWA sendto() call. While constructing API header, the destination, either in FDT entry or in the RWA call, is used to fill out fileds “Destination Address” and “Destination Port”. All the other fields are still got from the corresponding FDT entry. In short, the sendto() has to deal with the difference between connected-mode and unconnected-mode and construct API header differently.
- The implementation of sendmsg() is quite similar to that of sendto(). The name of destination can also be found in the argument list to RWA sendmsg() call. The only difference of the two functions is that they use different ways of carrying messages: sendto() puts the whole message into an unique buffer but the message being transferred by sendmsg() may be found in multiple buffers. A simple solution is taken: the messages from multiple buffers are collected and put into a single buffer inside the shared library. It is then appended with API header and sent by Glibc sendto() call. This simple solution causes a problem: sendmsg() call in Glibc can carry ancillary data (control information). It will get lost when using this simple solution. This is a limitation of the project. A possible solution is the ancillary data may be put into API header, which will result in a bigger and variable length API header.
- The write () and writev () functions are not socket-specific. They can be used with any file descriptor. When using with sockets, as stated in the man page of send (), “the only

difference between send and write is the presence of flags. With zero flags parameter, send is equivalent to write." Thus the implementation of write () is totally based on send () in the shared library. As for writev(), it resembles sendmsg(), the messages to be transferred distributed in multiple vectors, all these messages are collected and put into one unique buffer and then transferred all at once.

5.6.2. Receiving

The shared library recv() function is closely related to the send() function. The recv() receives message by calling Glibc recvfrom() call. It then strips off API header and delivers payload to RWA recv() call. The process is depicted as the following figure.

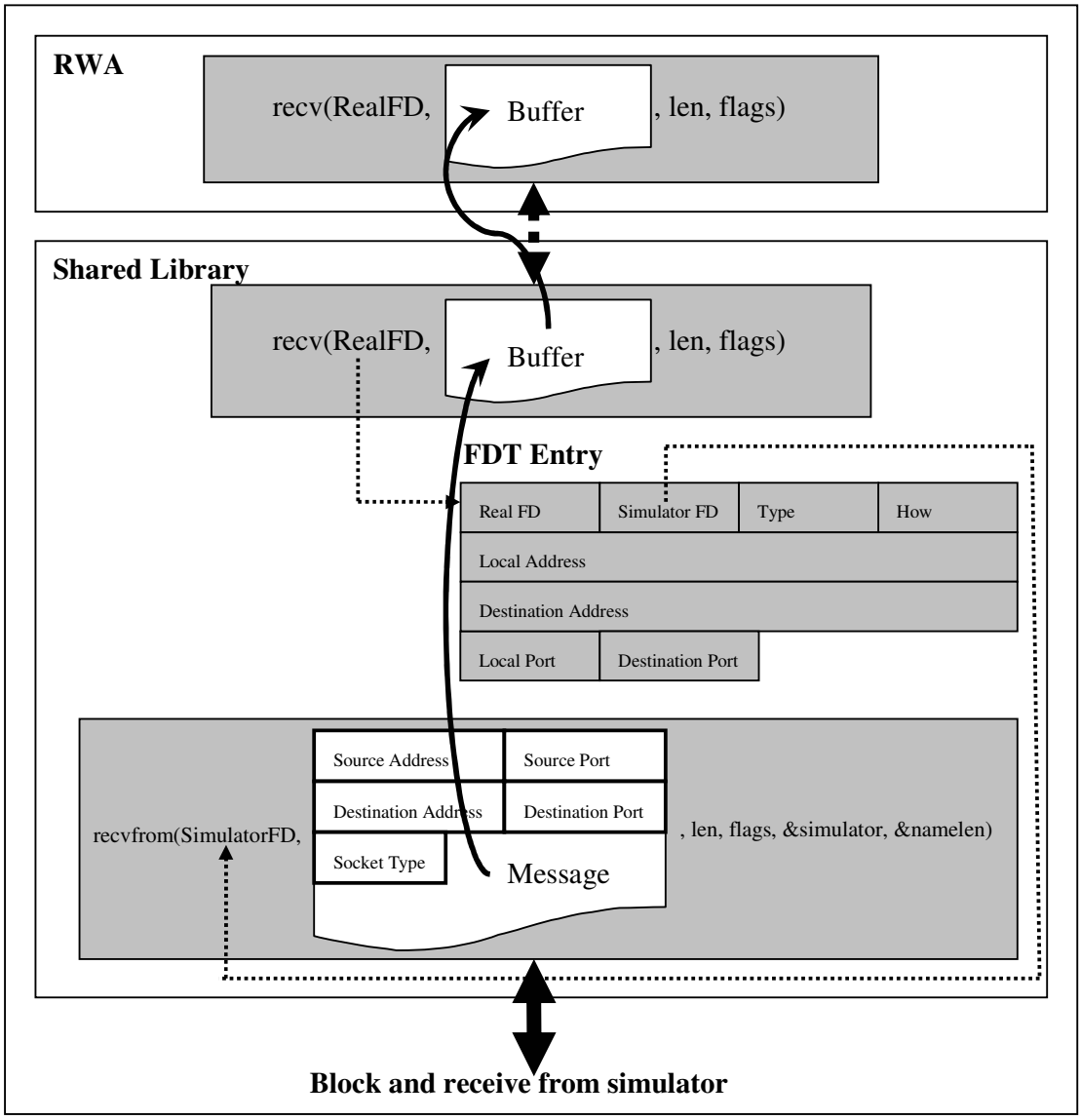

Figure 5-11 Receiving message in the shared library

After capturing RWA recv() call, the shared library locates an entry in FDT via the “Real FD” found in the RWA call. This is completely the same as that in the implementation of send() call. The shared library is then blocked by invoking Glibc recvfrom() call. The call is shown at the bottom of the figure 5-11. Please refer to the man page of recvfrom() for more detail on the arguments.

The call will block until message comes from simulator. When the Glibc recvfrom() returns, the received message can be found in the inner buffer of the call. The received message again consists of an API header and payload.

The shared library then strips of API header and passes the payload to RWA. The bold curly lines and arrows in figure 5-11 show how the received message

traverses through the shared library and reaches its destination, a receiving RWA.

In the process above, there is one special case need to be dealt with: because the recv() call only works in connected mode, and the "Simulator FD" used by the Glibc recvfrom() is not connected to any socket. To mimic the behavior of connected mode, when message arrives, the "sources address" and "source port" in API header are compared with the "destination address" and "destination port" in the found FDT entry. The payload is delivered to RWA only when they are identical. All the messages not from the "connected" source are rejected by the shared library.

Besides recv(), the receiving family has other members: recvfrom (), recvmsg (), read () and readv().

- The recvfrom() may be used to receive data on both a connected socket and an unconnected socket. When it is used with a connected socket, its implementation is completely the same as that of recv(). When it is used with an unconnected socket, there is one subtle but critical action in the implementation. The recvfrom() function takes six arguments, two of them are of special importance here: the first one is a data structure recording the source address and source port number. The second one is the length of the data structure. Due to message redirecting, the shared library gets messages from the simulator not from the RWA directly. After receiving message via the shared library recvfrom(), the source address and source port number will be those of a simulator application. It is inappropriate to pass the data structure to the receiving RWA, because this breaks the rule of "BSD Socket API for Simulator is transparent to RWA". To solve the problem, an important action is taken before the message is handed to receiving RWA: the content of the data structure is replaced with "Source Address" and "Source Port" found in API header. As a result, the received message for receiving RWA seems coming from source RWA and thus RWA transparency is achieved.
- The implementation of the shared library read () is based on recv(). With zero flags, recv() is equivalent to read (). So when the file descriptor in RWA read () call can be found in FDT, the call is passed to the shared library recv(); otherwise, the RWA read () is not called on a normal file descriptor not a socket, it is passed to Glibc read () call.
- The shared library recvmsg() is quite similar to the shared library recvfrom(). They both work with connected mode and

unconnected mode. The difference is that recvmsg() uses multiple buffers to receive messages. The shared library recvmsg() receives message from simulator in one buffer and then distributes the received message into multiple RWA buffers. Each RWA buffer and its size can be found in an argument of the RWA recvmsg() call.

- The shared library readv() is a combination of read() and recvmsg(). The function can be used with socket and normal file descriptors. The function receives message in one buffer and then distributes the received message into multiple RWA vectors.

5.7 I/O multiplexing

The term “multiplexing” is borrowed from telecommunications. “Multiplexing” is sending multiple signals or streams of information on a carrier at the same time in the form of a single complex signal and then recovering the separate signals at the receiving end [20]. In socket world, even a trivial program, such as a simple telnet client, needs to deal with more than one file descriptor. The telnet client accepts texts from standard input (file descriptor), sends the text out and receives text from server via socket. There are several ways of handling multiple file descriptors at the same time in the same thread: blocking I/O, non-blocking I/O and I/O multiplexing.

In blocking I/O, all descriptors get blocked. If one descriptor is blocked, the state changes of the others will not be notified until the blocked call returns. In the case of a telnet client, if the client blocks for accepting text from keyboard, and at the same time a server response comes. The client cannot get the response message until user finishes inputting text and sends text out. In most cases, this causes long delay.

In non-blocking I/O, all the descriptors are set to non-blocking mode. In non-blocking mode, process is not put to sleep even if a request can not be satisfied. Generally, all the calls on descriptors are sitting in a tight loop. Application polls kernel continuously to see if some operation is ready, this is called polling. Polling wastes CPU time and affects the performance of a multi-tasking system.

In I/O multiplexing, all the descriptors are blocking as they are in blocking I/O. Instead of blocking on each descriptor, I/O multiplexing is blocking on a single system call, select(), poll() or pselect(). Each time when state changes of descriptors are detected, blocking is released so that application can respond to the changes. The advantages of I/O multiplexing are quite obvious: it is not using

polling and thus not wasting CPU time; it is monitoring multiple descriptors simultaneously and responding any change without reasonable short delay. For such reasons, I/O multiplexing is the preferred I/O mode for single-thread networking applications.

I/O multiplexing is provided by three functions: select (), poll () and pselect (). The pselect() is a POSIX variation of select (). This section takes select () as an example to show how I/O multiplexing is implemented in the shared library.

When the select () is called, it instructs kernel to wait for some events to occur and then current process goes to sleep. When one or more specified events occur or when a specified amount of time has passed, kernel wakes up the process.

The select () call provides three groups of file descriptors, these file descriptors are for reading, writing and exception condition. The call also provides the maximum of time kernel has to wait. If the designated amount of time has passed, this call will return immediately no matter any of the specified events has occurred or not.

In BSD Socket API for simulator, the socket used by RWA and the socket used by the shared library maybe different, this causes complexity when implementing the shared library select () function: when a RWA is expecting message via RWA socket, but the message is actually coming via another socket, a simulator socket. Thus, the file descriptor sets constructed in RWA have to be modified before they are submitted to kernel. The modification involves adding the corresponding simulator file descriptors to the each of the sets.

The re-constructed sets are then submitted to kernel by calling Glibc select () call. Process is blocked until changes are detected or timeout.

The Glibc select () returns with new version of these sets. The new sets reflect the changes occurred. Still, the changes maybe on simulator file descriptors. So the returned sets have to be recovered in some way before they are returned to RWA, because the RWA does not even know the existence of simulator File descriptors. The process of recovery involves eliminating simulator sockets and adding corresponding RWA sockets.

As can be seen above, there are two important critical operations in the implementation of the shared library select () function: modifying file descriptor sets and recovering file descriptor sets.

5.7.1 Modifying File Descriptor Sets

The shared library catches RWA select () call and modifies the three file descriptor sets found in the RWA select () call. The shared library select () first tries to find an entry in FDT for each RWA socket in the file descriptor sets and:

- If such entry is not found, the current file descriptor is a normal file descriptor other than a socket, so it remains unchanged in the sets. The shared library does not handle file descriptors such as standard input or standard output; it simply leaves RWA and Glibc to do that.
- If such entry is found, the current file descriptor is a socket, so the sets need to be modified according to the type of the socket.
  - If current socket is UDP socket, RWA and the shared library use the same UDP socket. There will be no change to the sets.
  - If the current socket is TCP socket, RWA and the shared library are using different sockets; changes maybe made to the sets. The TCP sockets are categorized as two classes: the listening socket and transmitting socket (the two kinds of sockets have different pattern in their FDT entry). Different sockets cause the sets to be modified differently. Listening sockets exist only on server. Such sockets are used by server to accept connections from clients. There are no message exchange between listening sockets and the simulator. Listening sockets have no corresponding simulator socket in its FDT entry. Therefore, listening sockets cause no change in the sets. Unlike listening socket, transmitting sockets need to exchange messages with simulator, so there must be a UDP socket accompanying each transmitting socket. The UDP socket can be found in FDT entry. The shared library select () clears the entry for RWA transmitting socket and then adds an entry for the corresponding UDP socket.

5.7.2 Recovering file descriptor sets

Recovering file descriptor sets is a reverse process of modifying file descriptor sets. When the call to Glibc select () returns, the file descriptor sets reflect the changes of file descriptors. These file descriptors could be RWA sockets, simulator sockets or normal file descriptors. The shared library select () picks each of the changed file descriptors contained in the returned sets, and:

- Looks up the column of “Real FD” in FDT. If the changed file descriptor is a RWA socket, the entry of the changed file descriptor in the returned sets remains as it was.
- Looks up the column of “simulator FD” in FDT. If the changed file

descriptor is a simulator FD, the following two modifications will be made to the returned sets:

1. Clear the entry for the changed simulator FD.
2. Set the entry for the corresponding RWA socket.

- If both of the two searches above fail, that means the current changed file descriptor is not a socket. As mentioned before, a normal file descriptor is not handled in the shared library. The entry for the changed file descriptor remains unchanged.

In short, the implementation of the shared library select () function consists of three stages:

1. Modifying file descriptor sets.
2. Invoking Glibc select () function
3. Recovering file descriptor sets.

The above three stages apply to the other two I/O multiplexing functions: pselect () and poll () as well.

The pselect() is a POSIX variant of the select () function and is now supported by most of the UNIX-like systems. It is different from select () in three aspects (see man page of select () and pselect()):

1. The pselect() employs different data structure for timeout. The new data structure enables RWA to specify nanoseconds, but the old data structure in select () just specifies microseconds.
2. The select () function may change the timeout parameter to indicate how much time was left. The pselect() does not change this parameter.
3. The pselect () function first replaces the current signal mask, then does the select () function, and at last restores the original signal mask.

The implementation of the shared library pselect() is similar to that of shared library select () function. It also takes three steps. The only difference is in the second step: the Glibc pselect () other than the Glibc select () function is called.

The poll () function provides similar functionality as select () does, the function enhances select () by specifying events on each file descriptor. The events are actually a bit mask specifying the events the application is interested in. The implementation of the shared library poll () takes three stages:

1. Modifying poll file descriptor set.
2. Invoking Glibc poll () function.
3. Restoring poll file descriptor set.

5.8 Closing socket

There are two functions that can be used to close a socket and terminate a TCP connection: close () and shutdown (). Although the two functions have similar functionality, they are actually different in many aspects.

The former is a general purpose close function. It can close any file descriptor. The latter one can only close sockets. This is the most obvious difference between the two functions. Further deep analysis reveals more interesting and important differences.

The man page of close () says “If fd is the last copy of a particular file descriptor the resources associated with it are freed; if the descriptor was the last reference to a file which has been removed using unlink the file is deleted.” It reveals a very important fact of the function: there exists a “Descriptor Reference Count” for each file descriptor. The close () does not actually close a file descriptor unless its count reaches zero. Whereas the shutdown () function “causes all or part of a full-duplex connection on the socket associated with socket to be shut down.” (See man page of shutdown () function). A socket may be duplicated in several ways: fork (), dup () or dup2 (), no matter how many times a socket has been duplicated, shutdown () will close all of them straight away.

Another difference between close () and shutdown () is that the shutdown () can partially closes a connection. There is an argument to the shutdown () function indicating how the socket is closed, the argument can be one of the three values:

Value	Description
SHUT_RD	No more receives on the socket; messages can still be sent out; send buffer is discarded.
SHUT_WR	No more sends on the socket; messages can still be received; read buffer is discarded.
SHUT_RDWR	No more receives and sends on the socket; both read buffer and send buffer discarded.

Table 5-1 How to shutdown () a socket

As seen from the table above, the socket can be in different states after the shutdown () function is called.

In the BSD Socket API for Simulator, both close () and shutdown () are

implemented. Because the two functions have little in common, their implementations are quite different.

- In the shared library close () function, the file descriptor passed by RWA close () call is used to locate an entry in FDT. If such entry is not found, current file descriptor is not a socket, so the shared library leaves Glibc to handle the RWA close () call. Otherwise, the function close RWA socket first and then close the corresponding simulator socket if such simulator socket exists and is different form the RWA one. Closing is done using the Glibc close () function, so both the RWA socket and simulator socket are closed (both the descriptor reference counters are decremented) simultaneously. At the very last stage, the corresponding entry is deleted from the FDT.
- Closing socket by the shared library shutdown () is a little complicated. Because the call may leave socket in different state, a field in FDT entry, “How”, is used to record the current state of RWA socket. The socket descriptor in RWA shutdown () is used to locate an entry in FDT. If such entry is not found, an error returns to RWA as an indication of “the socket does not exist”, this is different from the close () call, because the shutdown () is used with socket only, so there is no need to pass the call to Glibc if the RWA socket can no be found in FDT. If such entry is found, the shared library shutdown () performs the following actions:
    1. Closing RWA socket by invoking Glibc shutdown () function and pass it the “how” argument of RWA shutdown () function.
    2. If necessary, closing corresponding simulator socket by invoking Glibc shutdown () function and pass it the “how” argument of RWA shutdown () call.
    3. If the last two actions succeed, the current socket state, “How”, is registered to FDT entry.
    4. If the current state of RWA socket is SHUT_RDWR, the corresponding entry is then deleted from FDT.

5.9 Multi-Clients Server

In the traditional client/server architecture, serving multiple clients at the same time is necessary for any server. The server must share its resources among all the connected clients. For any client, the server should be like a dedicated server. There are several ways to achieve a concurrent server: I/O multiplexing, multi-processing and multi-threading.

5.9.1 I/O multiplexing

The first approach is the I/O multiplexing that has been presented in section 5.6. The server accepts multiple clients; each of the connected clients has a corresponding server-side socket to which it is connecting. The server puts all the server-side sockets to a data structure, which data structure is used depends on which function is chosen for I/O multiplexing. The server then chooses one of the three functions (select (), pselect () or poll ()) to block and wait for the response from kernel. Each time when one or more socket state change is detected, kernel informs the server of the change by modifying the pattern of the data structure. The blocking is then released and server takes appropriate actions to serve the clients. I/O multiplexing has been implemented in the shared library. As shown in section 5.6 “I/O multiplexing”, all the three functions: select (), pselect () and poll (), are implemented in the shared library and they can deal with multiple clients simultaneously.

5.9.2 Multi-processing

The second approach of multi-client server is multi-process by fork () call. In UNIX, fork () is the only way to create a new process and it may be the simplest way to service multiple clients simultaneously. In I/O multiplexing, the I/O multiplexing functions need to be put in an infinite loop, when a request takes long to service, the other clients still have to wait. But by forking a new process for each client, all the clients are serviced in parallel. Although context switching still takes some time, no one has to wait for a long time for the others to finish their work. So multi-processing is preferred when the service routine takes much time to finish.

When server receives connection request from a client, the server invokes fork () to create a child process and the child is then assigned to service the connected client. The parent process continues to wait for other connections. The parent closes the socket connecting to client by the close () call. The shutdown () can not be used because child is still using the socket to handle the new connection (see section 5.8 “Closing Socket”).

In the shared library, the fork () call is not implemented because the original one still works. The man page of fork () describes the function as “fork creates a child process that differs from the parent only in its PID and PPID, and in the fact that resource utilizations are set to 0. File locks and pending signals are not inherited.” As a result, the whole shared library is copied from parent to child. The new process contains FDT and everything else making the critical message redirecting mechanism work. The multi-process server can work on top of the shared library

transparently.

5.9.3 Multi-threading

The multi-process approach does achieve better parallelism, but it still suffers from several drawbacks. For example, sharing information among processes becomes a tricky issue, thus, it requires complex inter-process communication mechanism such as shared memory, message queue, socket and semaphore. Another drawback with the multi-process approach is the cost in terms of server resources. It requires more CPU time to start and manage new processes.

The multi-processing approach does not make a scalable server, one of the most import goals of network servers. Scalability is better achieved by the multi-threading approach. Multi-threading is relatively new to UNIX and LINUX, LINUX kernel 2.0.0 started to support multi-threading. A thread is a light-weight process, it demands less resource compared to a process. As the multi-thread approach employs a complex mechanism, it is very hard to program and debug a multi-thread server.

In the shared library, there is nothing special for multi-threading as it supports multi-threading inherently.

A multi-threaded network server accepts request from clients and pop up a new thread for each connected client. For example, by calling the POSIX thread (pthread) library function pthread_create(). The new thread is then assigned to deal with the connected client. In multi-threading, the multiple threads in a process share process resources and the multiple threads execute independently, so the multiple threads need somehow to be synchronized. Thread synchronization can be achieved by mutexes and condition variables, which have been implemented in the POSIX thread library.

When such network server is running on top of the BSD Socket API for Simulator shared library, all the pop-up threads shared FDT and the other data structures in the shared library. The shared library just catches socket-related calls and deals with them as explained in the previous sections, nothing is special here. The message redirecting mechanism works with multi-threaded network server just as it does with the other kinds of servers.

As for the complexities in multi-threading such as popping up new threads, joining threads, exiting threads and synchronization, it is the responsibility of the programmer of the server to make the server work correctly. There is no need to implement any thread-specific function in the shared library.

5.10 Standard I/O on Sockets

Thus far, all the socket programming is about using socket, a special file descriptor, directly. In the real world, however, rather than employing file descriptors programmers commonly prefer the FILE streams when doing I/O. The read () and write () system calls can send and receive messages on a file descriptor, but the two functions are functionally limited, in some cases they are inconvenient. For example, suppose we try to read from a socket one line at a time, the write () system call can not do this. We have to read as much as possible into a buffer and then extract one line at a time from the buffer. It needs lots of extra programming efforts.

The standard I/O (stdio) library is the best solution to such kind of problems. "The standard I/O library provides a simple and efficient buffered stream I/O interface" (see man page of stdio), there are plenty of functions in the library to suit probably any I/O requirement. Besides, it has other advantages.

5.10.1 Advantages of standard I/O library

File descriptors provide a low-level interface for I/O operations, whereas streams provide a higher level interface on top of the low-level file descriptors. All the standard I/O functions are using FILE stream for I/O, but they actually invoke the read () and write () system calls internally.

As file descriptors and streams are doing the same thing, a problem arises: which one should we use in programming? As suggested in the GNU C Library Reference Manual: "In general, you should stick with using streams rather than file descriptors, unless there is some specific operation you want to do that can only be done on a file descriptor." Streams have several advantages to support this suggestion.

- By using stream, the programs can be portable, a non-GNU system may not support file descriptors at all, but it must support streams. So it is very easy to move a program using standard I/O library from one system, say UNIX, to another system, say LINUX.
- The standard I/O library functions provide much powerful functionalities than the primitive read () and write (). You can read the whole buffer or one line at a time or even one character at a time if you like.
- The standard I/O library provides buffers for I/O. there are different kinds of buffering strategies such as un-buffered, line

buffered and fully buffered streams in standard I/O library. The library also enables users to get full control over the buffers. The buffering in the library can greatly improve the performance of programs.

5.10.2 How to Use Standard I/O Library on Sockets

The standard I/O library is very powerful, but we should not ignore its other side. For example, using standard library functions is not as easy as using the primitive read () and write (). This requires solving some tricky problems.

In socket programming, what we get is always a socket number, a raw file descriptor. To make use of the powerful standard I/O library functions, the socket number must be converted to stream in the first place, this is done using fdopen() function. The fdopen() wraps a file descriptor inside a FILE stream structure and then the structure can be used by the versatile standard I/O library functions.

Thus far, everything seems going very well: streams are created from sockets by fdopen () and lots of standard I/O library functions are already available. Is that enough? The answer is not yet.

Two simple programs were written to investigate this interesting idea. The programs were just simple server and client. Both the server and client used TCP sockets to exchange messages. Rather than using send () and recv (), they generated two streams for input and output respectively (as suggested in [Linux Socket Programming by Example, Warren Gay], it is a safer practice to open separate streams for input and output). Both the server and client can send and receive messages over the streams. The programs worked until a small modification was made: before terminating the connection, the client sent out a special “STOP” message to the server and then closed the output stream by the fclose () function (no more message to send out), but it was still receiving on the input stream for the acknowledgement from server. The server was supposed to receive the termination signal and then sent back an acknowledgement. The modification caused the client crashed.

When investigating the reason for the crash, it was found that something interesting happened in both fdopen() and fclose (). In fdopen(), the file descriptor is not dup’ed, and will be closed when the stream created by fdopen () is closed (see man page of fdopen ()). In fclose (), the function performs two actions: flushing the stream to be closed using fflush () (writing any buffered output data) and closing the underlying file descriptor by invoking the close () system call (see man page of fclose ()).

The reason why the client crashed is that when the output stream was closed, the underlying socket was closed as well. Because both the input and output stream are on top of the same socket, the input stream had no associated socket at all, therefore the call on the input stream caused an error.

Besides the fdopen () function, another very important condition for running the standard I/O library functions on socket has been found here: the socket needs to be duplicated so that input and output streams can be created on different sockets. Fortunately, duplicating file descriptor has been available in Glibc by two functions: dup () and dup2 ().

5.10.3 Duplicating socket

The two functions, dup () and dup2 (), are functionally equivalent. UNIX and LINUX allow multiple file descriptors to refer to the same open file (or socket). They return a duplicated file descriptor of the original one. The original file descriptor and the dup'ed one can be used interchangeably. Kernel will actually close the original file descriptor only when the last copy of the original file descriptor is closed ().

As can be seen from section 5.7 "Closing socket", there are two functions to close a socket: close () and shutdown (). The shutdown () should not be used here for the reason made clear already in the section. The best practice is: never close a socket manually, always use fclose() to close streams, the function will automatically call close () internally.

In the shared library, things are getting more complicated due to the existence of the mapping between "Real FD" and "Simulator FD". When RWA issues dup () or dup2 (), the shared library must catch the function, add and initialize a new entry in FDT. The following figure illustrates a typical scenario of duplicating a socket by the dup () function in shared library.

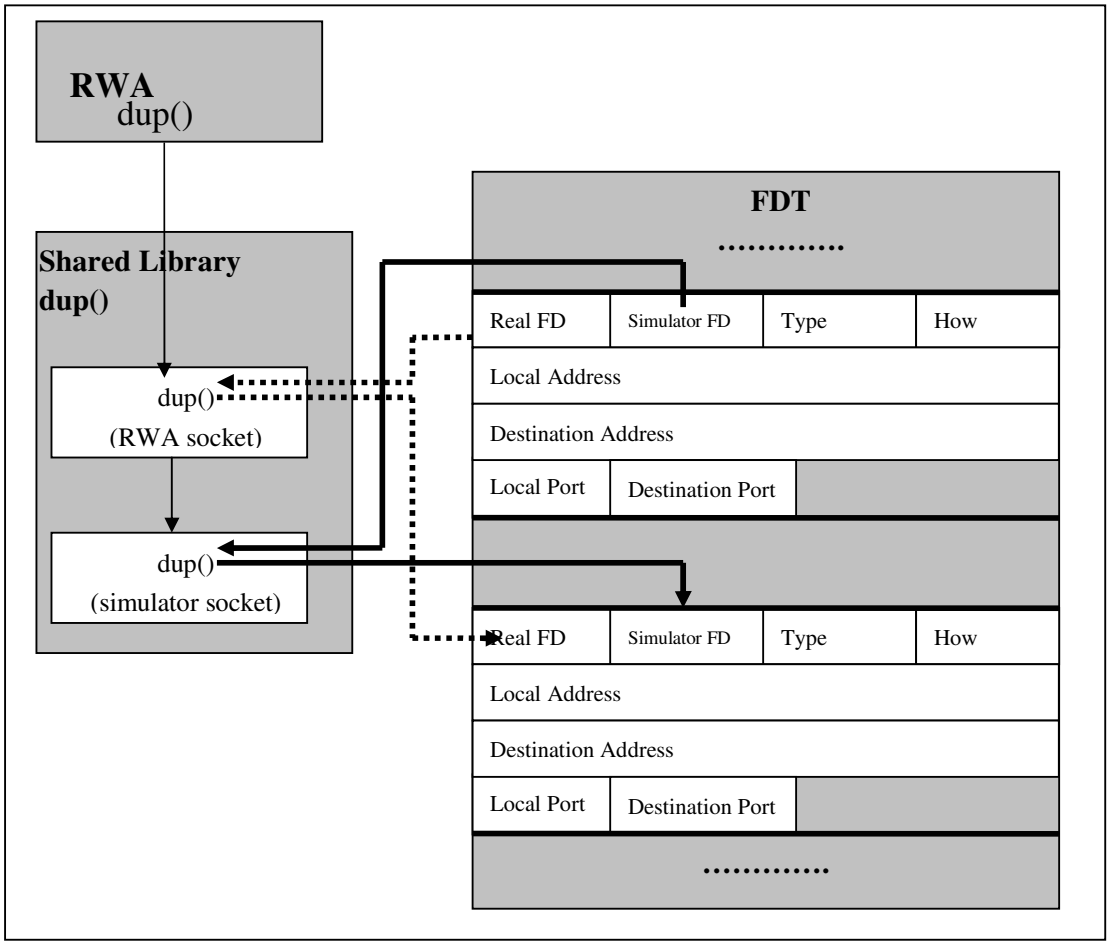

Figure 5-12 Duplicating socket in the shared library by dup ()

The basic idea of the shared library dup () function is to duplicate real socket and simulator socket so that the original socket and the dup'ed socket share the same connection. As usual, each time a RWA dup () call is captured, a FDT entry is located using the RWA socket, the RWA socket is the original socket that will be dup'ed later. As long as such entry is found, the shared library then takes two actions:

1. Duplicate RWA socket by calling Glibc dup () function.
2. Duplicate simulator socket in the found FDT entry.

If both of the last two actions succeed, two new sockets are obtained: one is the copy of real FD; the other is the copy of simulator FD.

At the very last stage, a new entry is added and the two new sockets are used to initialize the "Real FD" and "Simulator FD" respectively. All the other fields are copied from the entry of RWA socket to the new entry.

In the shared library, the original RWA socket and the dup'ed socket have the

same source address, source port, destination address and destination port; therefore they share the same connection.

5.11 Socket Options

At times, network application needs to set socket options to control the behavior of a socket, or on the other hand, needs to get the socket options to determine what to do next. For example, before closing a socket, the SO_LINGER option might need to be examined to see if the current setting is appropriate. If an application wants its messages being received by all the hosts on its network segment, it must enable broadcasting by setting the SO_BROADCAST socket option. There are dozens of socket options available on different levels. Network applications can get or set these options by the following four functions:

1. The getsockopt and setsockopt functions.
2. The fcntl function.
3. The ioctl function

There are several other functions, which are extracting some useful information from socket but not using socket options, are also discussed in the section. These functions are getsockname and getpeername.

The implementation of shared library getsockopt () function is pretty simple, the function does nothing but pass the RWA getsockopt () call to Glibc.

The shared library setsocketopt () function is a little complicated. As there is a simulator FD for each Real FD, and socket options need to be set on both Real FD and simulator FD. The shared library catches the RWA setsockopt() function and then locates an entry in FDT by the RWA socket. The function then sets socket option on the RWA socket and simulator socket in the found FDT entry.

The fcntl () function controls the behavior of file descriptor, so it can control the behavior of socket. The function controls various aspects of a file descriptor: close-on-execution, file status flags, advisory locking, mandatory locking, signals and notifying file/directory change and so on. In the shared library, the "command" passed by RWA fcntl () is examined first, if the "command" is socket-related and the file descriptor can be found in FDT, the call will be processed in the shared library:

- If the "command" is getting some information from socket, the RWA call is passed to Glibc on the Real socket.
- If the "command" is setting some options on socket, Glibc fcntl () is called on both Real FD and Simulator FD.

- If the file descriptor passed by RWA is not a socket, the call will be passed to Glibc straight away.

In the man page of fcntl (), the function can take variable number of arguments. The shared library fcntl () function supports this feature by declaring function prototype as:

int fcntl(int fd, int cmd, …)

In the shared library fcntl() function, some "command" takes a third argument, it could be either a long integer or a pointer of type "struct flock".

The commands in the following table have been implemented in the shared library fcntl () function. These commands cause the Glibc fcntl () function is invoked in different styles. The man page of fcntl () is the best place to find how the function should be called for various commands.

fcntl command	Description
F_SETFL	Set the file status flags to a specified value, such as O_NONBLOCK
F_GETFL	Read the file descriptor's flags.
F_DUPFD	Duplicate a file descriptor, different form of dup2 ().
F_GETOWN	Get the process ID or process group currently receiving SIGIO and SIGURG.
F_SETOWN	Set the process ID or process group currently receiving SIGIO and SIGURG.

Table 5-2 Socket-related commands of fcntl() function

The ioctl () function, is quite similar to fcntl () in terms of functionality and implementation in the shared library. But the ioctl () function has more options on socket. The function takes variable number of arguments as well, its prototype in the shared library is declared as:

int ioctl(int d, int request,…)

The first argument is a file descriptor. The second argument, request, just needs command in function fcntl (). It decides how the third argument is chosen. The third argument could be an integer, a struct ifconf, a struct ifreq, a struct arpreq or a struct rtentry. The following table is adapted from "UNIX Network Programming" [W Richard Stevens et al]. All the "requests" shown in the table have been implemented in the shared library ioctl() function. In the table, the first three columns are self-descriptive, the last column is the type of the last argument to the ioctl () function.

Category	Request	Description	Data type
Socket	SIOCATMARK	At out-of-band mark?	int
	SIOCSPGRP	Set process ID or process group ID of socket	int
	SIOCGPGRP	Get process ID or process group ID of socket	int
File	FIONBIO	Set/clear nonblocking flag	int
	FIOASYNC	Set/clear asynchronous I/O flag	int
	FIONREAD	Get # bytes in receive buffer	int
	FIOSETOWN	Set process ID or process group ID of file	int
	FIOGETOWN	Get process ID or process group ID of file	int
Interface	SIOCGIFCONF	Get list of all interfaces	struct ifconf
	SIOCSIFADDR	Set interface address	struct ifreq
	SIOCGIFADDR	Get interface address	struct ifreq
	SIOCSIFFLAGS	Set interface flags	struct ifreq
	SIOCGIFFLAGS	Get interface flags	struct ifreq
	SIOCSIFDSTADD	Set point-to-point address	struct ifreq
	SIOCGIFDSTADD	Get point-to-point address	struct ifreq
	SIOCGIFBRDADD	Get broadcast address	struct ifreq
	SIOCSIFBRDADD	Set broadcast address	struct ifreq
	SIOCGIFNETMAS	Get subnet mask	struct ifreq
	SIOCSIFNETMAS	Set subnet mask	struct ifreq
	SIOCGIFMETRIC	Get interface metric	struct ifreq
	SIOCSIFMETRIC	Set interface metric	struct ifreq
	SIOCGIFMTU	Get interface MTU	struct ifreq
ARP	SIOCSARP	Create/modify ARP entry	struct arpreq
	SIOCGARP	Get ARP entry	struct arpreq
	SIOCDARP	Delete ARP entry	struct arpreq
Routing	SIOCADDRT	Add route	struct rtentry
	SIOCDELRT	Delete route	struct rtentry

Table 5-3 Summary of networking ioctl requests

5.12 Compilation

Compilation of the shared library is identical to compiling any other shared library. However, there is only one thing which needs mention here: because the shared library functions have completely the same return values and signatures with some system calls or function calls, it causes some symbol name conflicts. To solve this problem, some data structures are moved out of kernel and put into a header file. Appendix B lists all such data structures.

5.13 Summary

The shared library is the most critical part of the BSD socket API for simulator. It acts as the interface of Real World Application and redirect messages between RWA and simulator. There are 29 functions are implemented in the shared library, these functions are classified into six categories: connection establishment, sending and receiving, closing socket, I/O multiplexing, duplicating socket, socket options. A more detailed function list can be found in appendix A.

1. Connection establishment: these functions are used to establish TCP and UDP connections or connecting sockets to form socket pairs. Generally, these functions need to add one or more FDT entries and initialize the new entries accordingly. There are six functions in this category.

    - socket()
    - bind()
    - connect()
    - listen()
    - accept()
    - socketpair()

2. Sending and receiving: these functions actually redirect messages from RWA to simulator or receive message from simulator and pass payload to RWA. There are 10 functions in this category. The 10 functions are further sub-classified as 5 pairs.

    - send()/recv()
    - sendto()/recvfrom()
    - read()/write()
    - readv()/writev()
    - sendmsg()/recvmsg()

3. Closing socket: these functions close socket in various ways. There are two functions in this category.

    - close()
    - shutdown()

4. I/O multiplexing: in the same thread, these functions achieve parallelism when dealing with multiple file descriptors. There are three functions in this category.

    - select()

- pselect()
- poll()

5. Duplicating socket: when using standard I/O library on socket, the original socket has to be duplicated by these functions. There are two functions in this category.
    - dup()
    - dup2()

6. Socket options: these functions get or set socket options in various way, so that socket information can be retrieved or socket behavior can be controlled. There are six functions in this category.

    - getsockopt()
    - setsockopt()
    - fcntl()
    - ioctl()
    - getsockname()
    - getpeername()

The above functions, however, are not a complete list of all the socket-related functions. There are a number of other socket functions that are not implemented in the shared library. The real world applications just use the original implementation of these functions. These functions will not be captured by the shared library. The following are some network support functions:

- inet_addr()
- htonl()
- htons()
- ntohl()
- ntohs()
- inet_aton()

Chapter 6
Tests

The BSD Socket API for Simulator is a complex system. The system has several features that make it error-prone and hard to debug.

- The two main parts of the system, simulator application pair and shared library, are working in different address spaces. Each instance of the shared library may fork other new processes. To make multiple processes work together in harmony is always a tough task for programmers.
- Shared library, the core of the system, consists of 29 socket-related functions and dozens of support functions. All the socket related functions are relatively independent. It is the Real World Application (RWA) that organizes them together to form a program. The shared library has no idea about how these functions will be called and how they will be combined together. The shared library must deal with these complicated situations.

The complexity of the system and the complicated situations the system has to deal with, mean tests are a critical part of this project. The tests that have been done to the BSD Socket API for Simulator are categorized as unit tests, functional tests and efficiency tests.

In software Quality Assurance (QA) world, unit tests are written from a programmer's perspective, they ensure that a particular function or a method of class performs a set of specific tasks. Each test confirms that a function or a method produces the expected output when given a known input. In other words, unit tests ensure that a function or a method performs correctly under any circumstances. Dozens of small programs are written in this project to test each of the 29 functions in the shared library.
Functional tests, on the other hand, are written from a user's perspective. These tests confirm that the system does what users are expecting it to. There are two key points for functional tests:

1. It is from user's perspective;
2. It tests the whole system other than individual functions.

To satisfy such conditions, three groups of real world applications are taken for

functional tests. Theses real world applications are compiled as usual and run on top of the BSD Socket API for simulator. These applications are:

1. programs from UNIX Network Programming, The sockets Networking API, third edition [W. Richard Stevens et al];
2. programs from Linux Socket Programming by Example [Warren Gay];
3. TELNET.

The former two are taken from examples of two books. The last one is the popular TELNET available in any UNIX-like system.

Efficiency tests try to evaluate the performance of network simulations using the BSD Socket API for Simulator compared to those without using it. The message redirecting mechanism used in BSD Socket API for Simulator will definitely slow down simulations. The tests done under this category are going to test to what extent the overall performance of a simulation is affected by the system.

6.1 Unit Tests

The Unit tests are used for testing individual functions. The development of each function is accompanied by its unit tests. At times, unit tests are even written before writing the function.

A gradually-expanding strategy is taken when developing the BSD Socket API for Simulator. All the socket-related functions are classified in six categories (please refer to section 5.12 for more details): connection establishment, sending and receiving, closing socket, I/O multiplexing, duplicating socket and socket options. The gradually-expanding strategy is that each category is given a priority. The category with higher priority is implemented first.

Among the six categories, the connection establishment category has the highest priority. It is used for adding and initializing File Descriptor Table (FDT) entries. It lays down the foundation for functions in other categories. The second priority is given to the sending and receiving category; functions in this category employs FDT entries to redirect messages. The closing socket category gets the third priority. The above three categories are core of the project; nearly all the network programs contain functions from these categories. The other three categories are given the same priority, it does not matter in what order they are implemented.

The gradually-expanding strategy is taken in unit tests as well. Functions in each category must be fully tested before moving to the next category. Actually,

functions in the same category have priority as well, for example, in the connection establishment category socket () has the highest priority, it must be implemented first.

The unit tests for the last three categories are not presented in this section for the following reasons:

1. Some unit tests are done together with functional tests. For example, the I/O multiplexing category and duplicating socket category are intensively tested in functional tests.
2. Some unit tests are fairly easy and straight forward, such as the socket options category.

As the functions in the same category are doing something similar and so do their unit tests, a general process is presented for each category. The process applies to all the functions in the category.

6.1.1 Connection Establishment

Functions in this category add entries to FDT or initialize some fields of a specific entry. The functions involving adding FDT entries are: socket (), socket pair () and accept (); all the functions in the category initialize FDT entry.

The unit tests of the connection establishment category are to verify that new FDT entry is added and initialized properly. For such purposes, a test function is added to the shared library which displays current state of FDT. Under debug mode, all functions in the category invoke the test function each time when changes are made to FDT.

As there are large amounts of possible ways to establish a connection, functions in the connection establishment category may behave differently under various circumstances. For example, in the process of establishing a TCP client, the optional bind () function can cause the subsequent connect () function to behave in different manner. (Please refer to section 5.4.1.2 “Establishment of TCP client using the shared library” for more details). A series of typical unit tests are taken as examples to demonstrate how unit tests are done in the connection establishment category. The scenario chosen here is the establishment of TCP connection (see figure 5-3).

The following two figures illustrate the debugging screens of server and client. All the messages are produced by shared libraries.

```
ns2 SOCK API socket() is called.
RealFD  ns2FD   MyAddress       MyPort  DstAddr         DstPort how
3       -1      234543490       0       0               0       0
ns2 SOCK API bind() is called.
RealFD  ns2FD   MyAddress       MyPort  DstAddr         DstPort how
3       -1      234543490       3490    0               0       0
ns2 SOCK API listen() is called.
ns2 SOCK API accept() is called.
RealFD  ns2FD   MyAddress       MyPort  DstAddr         DstPort how
3       -1      234543490       3490    0               0       0
4       5       234543490       47304   234543490       45758   0
server: got connection (fd:4) from 130.217.250.13
ns2 SOCK API send() is called.
ns2 SOCK API accept() is called.
```

Figure 6-1 Unit tests for TCP server – connection establishment

```
ns2 SOCK API socket() is called.
RealFD  ns2FD   MyAddress       MyPort  DstAddr         DstPort how
3       -1      234543490       0       0               0       0
ns2 SOCK API connect() is called.
RealFD  ns2FD   MyAddress       MyPort  DstAddr         DstPort how
3       4      234543490        45758   234543490       47304   0
ns2 SOCK API recv() is called.
Get message from ns2:Hello World!(0)
```

Figure 6-2 Unit tests for TCP client – connection establishment

Both client and server are using the minimum subsets of the functions from connection establishment category to build TCP connection. Each bold line in the pictures represents that a shared library function is called.

The server invokes socket (), bind (), listen () and accept () one by one. The client invokes socket () and connect (). The FDT entries are added and initialized as expected. Please refer to section 5.4.1 “TCP connection” for how each of the functions behaves in the process of connection establishment.

As a result the server’s FDT has two entries. The first entry is a listening socket (the simulator FD is set to -1). The other entry represents a connection to client; its local port is 45758 (chosen randomly); its destination port is 47304 (got from

the connected client). The client's FDT has one entry. The entry has local port of 47304(chosen randomly) and destination port of 45758 (server's shared library sent this port number to client "under the hood").

Another thing to note is that because both the client and server are running on the same host, local addresses and destination addresses have exactly the same value. From the two figures above, a two way connection are built up through FDTs. Besides the unit tests above, other tests that have been done for the connection establishment category are:

- Establishment of TCP connection with client calls the optional bind ().
- Establishment of UDP connection with server not calling connect () and client not calling bind () and connect ().
- Establishment of UDP connection with server not calling connect () and client not calling connect ().
- Establishment of UDP connection with server not calling connect () and client not calling bind ().
- Establishment of UDP connection with server calls connect () and client not calling bind () and connect ().
- Establishment of UDP connection with server calls connect () and client not calling connect ().
- Establishment of UDP connection with server calls connect () and client not calling bind ().
- Establishment of connection by socketpair().

6.1.2 Sending and receiving

The sole purpose of unit test for the sending and receiving category is to verify that the message redirecting mechanism works. The approach taken here is to check each node on the message redirecting path: sending shared library, simulator application pair and receiving shared library.

As there are six pairs of sending/receiving functions in this category, and most of them can be used with connected mode and unconnected mode. Further, the connected mode could be either TCP or UDP. So there are dozens of possible combinations of function pairs and connection mode. Fortunately, all these combinations employ same program templates. For different unit tests, what you need to do is just changing the sending/receiving functions or a single parameter to the socket () call. (For example, change the socket type from DGRAM to STREAM). Therefore, a typical scenario is chosen as example of unit tests of the sending and receiving category: sending/receiving messages over TCP connection using the send()/recv() function pair.

In the chosen unit test, server and client take the same processes seen in last section to establish TCP connection. Then the server sends out a message “Hello World!” together with a number to show how many “Hello World!” has been sent (the number begin with 0). The client receives the messages and sends back acknowledgements. By such way, the two way communication is verified.

The following three figures show the screen outputs of all the nodes on the message redirecting path. All the outputs shown in the figures are produced by shared libraries and simulator applications pairs.

```
1: shared momory address: 1431752704
2: Application Pair Table:
3:          4000     5000     0        0
4:          6000     7000     0        0
5:
6: Load balancing thread: got connection from 130.217.250.13
7: Load balancing thread: send port back: 4000
8:
9: Load balancing thread: got connection from 130.217.250.13
10: Load balancing thread: 5000
11:
12: Simulator Applicaion gets message from shared library:Hello World!(0)
13: Simulator Applicaion sends message to NSC:Hello World!(0)
14: Simulator application gets message from NSC:Hello World!(0)
15: Simulator application sends message to shared library:Hello World!(0)
16:
17: Simulator Applicaion gets message from shared library:Hello World!(1)
18: Simulator Applicaion sends message to NSC:Hello World!(1)
19: Simulator application gets message from NSC:Hello World!(1)
20: Simulator application sends message to shared library:Hello World!(1)
.....................
```

Figure 6-3 Unit tests for application pair – sending and receiving

Figure 6-3 shows the output of the simulator (NS-2). The shared library is created first (line 1); two pairs of simulator applications start up (lines 2 to 4); when server is starting up, the load balancing thread provides it with a port number, so that the server can communicate with a simulator application via the port (lines 6 to 7); the client then starts up (lines 9 to 10).

Lines 12 to 15 show how the message redirecting mechanism works in a simulator: message arrives at simulator (line12); it is then sent to NSC (line 13);

the other end of the application pair receives message from NSC (line 14); and, finally, the received message is delivered to another shared library which has been preloaded with a receiving RWA (line 15).
The unit tests above verify that the simulator application pair is working correctly: the load balancing thread manages the application pair table and responds shared libraries with port numbers. The messages from the outer world are properly passed through simulator and redirected to the outer world.

The process shown in figure 6-3 is completely the same as depicted in figure 3-3. A detailed description can be found in section 4.3 “Interacting with Real World Application and NSC”.

Figure 6-4 below shows how messages are sent and acknowledgements received at the server side:

```
1: zl88@voodoo:~$ LD_PRELOAD=../../libSockAPI.so ./tcpserver_snd
2: Reading from config file: voodoo.cs.waikato.ac.nz
3: Host name  : voodoo.cs.waikato.ac.nz
4: IP Address : 130.217.250.13
5: ns-2 will be listening on port 4000
6: ns2 SOCK API socket() is called.
7: ns2 SOCK API bind() is called.
8: ns2 SOCK API listen() is called.
9: ns2 SOCK API accept() is called.
10: server: got connection (fd:4) from 130.217.250.13
11:
12: [send()]Shared library gets message from RWA:Hello World!(0)
13: [send()]Shared library sends message to simulator:Hello World!(0)
14: RWA gets ACK.
.....................
```

Figure 6-4 Unit tests for server – sending and receiving

The server starts up with a preloaded shared library (line 1). At the initialization stage, the shared library reads a file, in which file the name of the host running simulator can be obtained (lines 2 to 4); the initialization routine then acquires a port number from the load balancing thread (line 5); several functions are then called to actually build up the server (lines 6 to 9); a connection is established between the server and a client (line 10).

Line 12 to 14 shows the process of message redirection on the server. The RWA send () call is captured by the shared library (line 12); the shared library redirects captured message to simulator; the server at last receives an acknowledgement

from the connected client.

```
1: zl88@voodoo:~ $ LD_PRELOAD=../../libSockAPI.so ./tcpclient_rcv voodoo
2: Reading from config file: voodoo.cs.waikato.ac.nz
3: Host name  : voodoo.cs.waikato.ac.nz
4: IP Address : 130.217.250.13
5: ns-2 will be listening on port 5000
6: ns2 SOCK API socket() is called.
7:
8: [recv()]Share library gets message from simulator:Hello World!(0)
9: RWA receives: Hello World!(0)
10: RWA sending ACK back
11:
12: [recv()]Share library gets message from simulator:Hello World!(1)
13: RWA receives: Hello World!(1)
14: RWA sending ACK back
.....................
```

Figure 6-5 Unit tests for client – sending and receiving

Figure 6-5 is the output of the client. The client starts up using a command line (line 1); it reads configuration file and acquires the name of the host running simulator (lines 2 to 5). The client is built up using a sole line of code (line 6).

The client receives message from simulator (line 8); the received message is then delivered to RWA (line 9); an acknowledgement is sent back to server (line 10).

From figure 6-3, figure 6-4 and figure 6-5, a bi-directional communication is achieved between server and client via simulator. The message redirecting mechanism works on each node of the redirecting path.

6.1.3 Closing Socket

In the BSD Socket API for Simulator, functions in the closing socket category are doing two things:

- Close both real FD and simulator FD.
- Delete corresponding FDT entry.

If both the real FD and simulator FD are closed successfully, their FDT entry is then deleted. Therefore, unit tests of closing socket category are to verify that the FDT entry is deleted.

The unit tests of closing socket category make use of the similar server and client programs as those used by the previous two categories; only one modification is done to the servers. The modification is to make the server support multiple clients by multi-processing. Each time a client tries to connect to the server, a new process is created by the server. The parent process closes the new child socket and then waits for other connections to come; the child process closes the parent's listening socket and then exchanges message with the connected client. Figure 6-6 is the output of such server.

```
   ....................
1: ns2 SOCK API socket() is called.
2: RealFD  ns2FD   MyAddress       MyPort  DstAddr         DstPort how
3: 3       -1      234543490       0       0               0       0
4: ns2 SOCK API bind() is called.
5: ns2 SOCK API listen() is called.
6: ns2 SOCK API accept() is called.
7: RealFD  ns2FD   MyAddress       MyPort  DstAddr         DstPort how
8: 3       -1      234543490       3490    0               0       0
9: 4       5       234543490       49258   234543490       16462   0
10: Socket 3 deleted.
11: RealFD  ns2FD   MyAddress       MyPort  DstAddr         DstPort how
12: 4       5       234543490       49258   234543490       16462   0
13: [send()]Shared library gets message from RWA:Hello World!(0)
14: [send()]Shared library sends message to simulator:Hello World!(0)
15: Socket 4 deleted.
16: RealFD  ns2FD   MyAddress       MyPort  DstAddr         DstPort how
17: 3       -1      234543490       3490    0               0       0
18: ns2 SOCK API accept() is called.
19: RWA gets ACK.
....................
```

Figure 6-6 Unit tests for server – closing socket

In figure 6.6, the socket () function creates one entry in FDT (lines 2 to 3); the accept () function create another entry for a new socket, so there exists two entries in FDT (lines 7 to 9); child process deletes the FDT entry of the listening socket (lines 10 to 12) and exchanges messages with client (lines 13,14 and 19); parent process deletes the FDT entry of the new socket (line 15 to 17) and gets ready for other connections (line 18).

The unit tests above not only verify that the close () function behaves as expected (see section 5.8 "Closing socket") but also testify that the shared library supports multi-clients server (see section 5.9 "Multi-clients server").

6.2 Functional Tests

Running "real" code on BSD Socket API for Simulator is definitely the most interesting part of the project. There are three sources for the functional tests: the programs from "UNIX Network Programming, The Sockets Networking API, third edition" [W. Richard Stevens et al]; the programs from "Linux Socket Programming by Example" [Warren Gay]; the popular TELNET.

6.2.1 Tests by "UNIX Network Programming"

"UNIX Network Programming, the Sockets Networking API, third edition" [W. Richard Stevens et al] is recognized as the definitive reference for programmers to learn network programming techniques. The book covers nearly every aspect of UNIX network programming. It explains the complicated theories by providing interesting and understandable examples. Some of these examples are used to do functional tests on the BSD Socket API for Simulator.

The source code of these examples can be downloaded from website of the book [22]. These examples are supposed to run in UNIX only. There is no effort made to unpack and build them cleanly on various systems such as LINUX. As the BSD Socket API for Simulator is built on LINUX, these examples are modified so that it can be used for the functional tests that will be done in LINUX. The modifications include:

- Defining symbol names. Because UNIX and LINUX may use different symbol names, so compiling the examples in LINUX results in undefined identifiers. A simple way is taken to eliminate such errors: defining the lost symbol whenever it is found.
- Changing the header name or function name. At times, UNIX and LINUX use different header file and function name, they must be changed before it can be compiled in LINUX. The modifications made to the source code are listed in appendix C Modifications to Code from UNIX Network Programming.

There are hundreds of programs available from the book; some of them have been selected here for the functional tests. The chosen programs are from five directories. The following table shows the locations, names and brief descriptions on what those programs do. The locations use relative path names that are relative to the root directory of examples (Please refer to appendix C for more details).

Location	Name	Description
server\	client.c	A simple multi-process TCP server which creates multiple connections with a server
	serv01.c	A multi-process TCP server using polling
	serv02.c	A multi-process TCP server using signaling.
	serv08.c	A synchronized multi-threaded TCP server.
select\	tcpcli01.c	A typical TCP client which can exchange messages with server.
tcpcliserv\	tcpcli04.c	The TCP client creates up to 5 connections to server
	tcpserv02.c	A multi-process TCP server, un-used sockets are closed in each process.
	tcpservpoll01.c	A TCP server, I/O multiplexing by poll ().
	tcpservselect01.c	A TCP server, I/O multiplexing by select ().
udpcliserv\	udpcli01.c	A typical UDP client
	udpserv07.c	A typical UDP server
	udpservselect01.c	A UDP server, I/O multiplexing by select ().
sockopt\	checkopts.c	Check values of various socket options.
	prdefaults.c	Check default values of socket options.
	recbufset.c	Set receiving buffer size.
	sockopt.c	Get and set TCP maximum segment size and sending buffer size.

Table 6.1 Tests from "UNIX Network Programming"

The functional tests by the programs above cover all the five aspects of the BSD Socket API for Simulator; the only one yet to test is the duplicating socket category. The category is tested by programs from "LINUX Socket Programming by Example".

6.2.2 Tests by "Linux Socket Programming by Example"

As implied by its name, the book is in favor of two things: Linux socket programming and example. It introduces some important aspects of Linux socket programming using various examples: the fundamental concepts of socket programming, forming network addresses, Ipv6, protocols, writing multi-clients servers, security, using standard I/O libraries on socket and so on.

"Linux Socket Programming by Example" is a kind of socket programming tutorial and it does not try to cover every aspect of Linux socket programming. As a matter of fact, it is not for advanced readers. So if you are trying to make deep research into socket programming, the book introduced in last section, "UNIX Network Programming", is the best source.

There are two features of "Linux Socket Programming by Example" that is

attractive to make it as source of functional tests on the BSD Socket API for Simulator:

(1) All the examples in the book are tailored specially for Linux, so there is no need to make any modification to these examples.
(2) The book touched the topic of using Standard I/O on sockets which are not covered in "UNIX Network Programming".

Programs in "Linux Socket Programming by Example" employ similar paradigm as those in "UNIX Network Programming" except for those from chapter 10 "Using Standard I/O on Sockets" and chapter 11 "Concurrent Clients Servers". Examples of the two chapters use standard I/O library functions to exchange messages between server and client. This is the only category of the BSD Socket API for Simulator not yet tested.

Some programs of the two chapters are chosen for functional tests as shown in table 6.2.

Location	Name	Description
Chapter 10	mkaddr.c	The program is a subroutine used by server.
	rpneng.c	RPN calculator code, it is used by server as well.
	rpnsrv.c	RPN server which server one client
Chapter 11	rpnsrv1.c	Multi-process RPN server by fork ().
	rpnsrv2.c	I/O multiplexing RPN server by select ().

Table 6.2 Tests from "Linux Socket Programming by Example"

RPN (Random Prime Number) server generates prime number randomly and returns it to client; client can make server to perform arithmetic operations such as addition and subtraction on the generated prime numbers, the result is returned to client as well. In the table above, there is no client provided. Actually "telnet" is used as client in the tests of this section. Running telnet on the BSD Socket API for Simulator is demonstrated in the section that follows.

6.2.3 Tests by Telnet

The tests here are still using the traditional server/client model. The telnet clients are started by the Linux "telnet" command and preloaded with the BSD Socket API for Simulator shared library. So the telnet clients are purely untouched real world applications. The telnet server is more difficult, as the default Linux telnet server (listening on port 23) can not be used here. First of all, only the root user can configure or turn on the telnet server. Besides, for the server to exchange messages with client via simulator, a BSD Socket API for Simulator shared library has to be preloaded together with the server. Generally, the two conditions

above can not be satisfied.

Instead of using the default Linux telnet server, a simple telnet server is written only for the tests here. The telnet server is a multi-clients server through I/O multiplexing; it uses the select () call to manage listening socket and multiple client sockets. When message come from a client, the message is relayed to all the connected clients except for the original sender.

The server is started by the following command line:

```
LD_PRELOAD=./libSockAPI.so ./telnetserver
```

Telnet client is started simply using the command line as following:

```
LD_PRELOAD=./libSockAPI.so telnet
```

After telnet client is started, connection with server is built using a command line (suppose the server is running on a host whose name is voodoo and the listening port number is 3000):

```
open –a voodoo 3000
```

6.3 Efficiency Tests

The messages in the BSD Socket API for Simulator are redirected among processes; that will definitely affect the efficiency. The efficiency tests designed here are to investigate to what degree the message redirecting mechanism affect the efficiency of real world programs.

By the message redirecting mechanism, a message takes several steps to reach its destination. In the functional tests, the purpose is to distinguish the part of time consumed by the BSD Socket API for Simulator from those consumed by other parts of a simulation. For such purpose, a server/client pair is designed to run on different levels of a simulation.

The server mimics a physic host and the client “pings” the “host” using TCP protocol. The tests end up with RTT (Round Trip Time) between the server and client.

As mentioned above, analyzing the RTT is a tricky thing, because the RTT consists of several parts:

1. The time taken by the two real world applications, the server and the client. In this test, the two real world applications use the

standard network stack, so simulator is not started at all.

2. The time taken by the BSD Socket API for Simulator.
3. The time taken by the simulator.

The different parts of RTT are separated by three tests as follows:

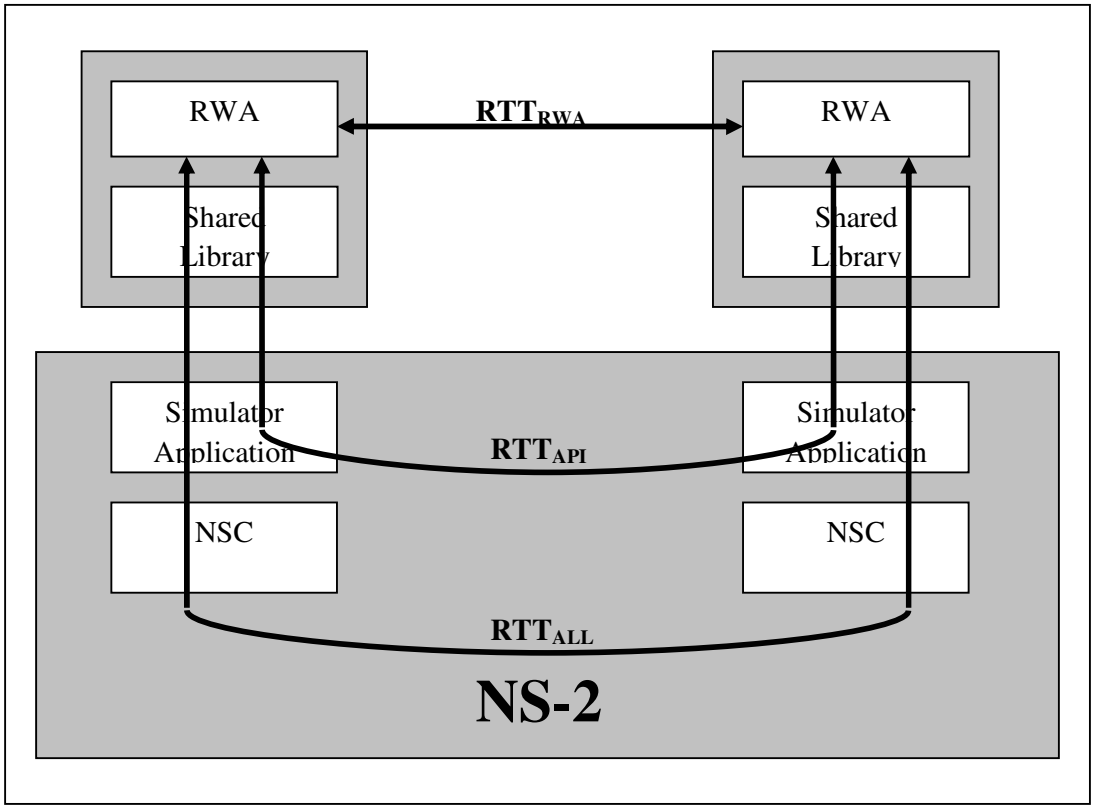

Figure 6-7 Efficiency Tests

RTT_{RWA} is the round trip time when real world applications communicate with each other directly; the messages are not redirected to simulator. Because the simulator and the shared library are not running at all, the RTT_{RWA} is the shortest one among the efficiency tests; it sets up the baseline for the following tests.

RTT_{API} is the round trip time when message is redirected to simulator, more precisely, the simulator application, but it is then passed to the other end of application pair other than NSC. This is achieved by setting up a "short circuit" between the two ends of an application pair. The "short circuit" is established using a TCL instruction which has been implemented in the simulator application.

When "short circuit" is enabled in a simulation, messages will only traverse components of the BSD Socket API for Simulator. Thus RTT_{API} is the best measurement of how the BSD Socket API for Simulator affects the performance

of real world applications.

RTT_{ALL} is the round trip time when a message travels through each components of a simulation framework: real world applications, simulator applications, shared libraries, network simulation cradle and the rest of a simulator. RTT_{ALL} is the overall performance of entire simulation framework using the BSD Socket API for Simulator.

In the tests above, when simulator is started, the bandwidth of the link between server and client is set to be very large (100Gbps) and the latency is set to be zero.

Table 6-4 shows the tests results of the last three categories. For each of the tests, client "pings" server 1000,000 times to get an average round trip time.

	#1	#2	#3	#4	#5	#6	#7	#8	#8	#9	AVG
$RTT_{RWA(us)}$	13	15	13	14	15	14	14	15	13	16	14.2
$RTT_{API(us)}$	24	21	27	25	19	22	21	18	16	21	21.4
$RTT_{ALL(us)}$	96	105	116	108	99	101	122	105	114	94	106

Table 6-3 Results of efficiency tests

From the results above, the average RTT_{RWA} is 14.2us, the average RTT_{API} is 21.4us and the average RTT_{ALL} is 106us. So the BSD Socket API for Simulator increases message transmission time by 50.7%. The cost of using the BSD Socket API for Simulator is acceptable. The overall RTT performance of simulation framework using the BSD socket API for Simulator increases by about 646.48%. The BSD Socket API for Simulator slows down the entire simulation framework a lot. There are two possible components in the simulation framework could be the source of the burden: the network simulation cradle (NSC) and the real time scheduler.

According to the tests in [Simulation with Real World Network Stacks, Sam Janson and Anthony McGregor.], "the CPU performance differs between ns-2's TCP implementations and using the Network Simulation Cradles TCP implementation in ns-2. The ns-2 FullTCP agent is up to six times faster than simulating with NSC network stack." So the NSC may cause the performance to drop significantly.

The other factor that may affect the performance is the real time scheduler. However, no tests are done to verify to what degree the real time scheduler slows down the simulation, as the BSD Socket API for Simulator now works only with real time scheduler.

Throughput is another very important measurement of network performance. The

throughput tests take similar ways as those in figure 6-7. In the tests, server sends out data as much as it can. There are three tests as follows:

- $Throughput_{RWA:}$ the test uses operating system network stack, and the simulator is unused in the test. It is actually the memory bandwidth.
- $Throughput_{API}$: the test uses "short circuit".
- $Throughput_{ALL}$: the test uses the BSD socket API for simulator and NSC.

The results are: $Throughput_{RWA}$ is 24.34 Gbps; $Throughput_{API}$ is 16.46 Gbps; $Throughput_{ALL}$ is 4.87 Gbps. So the BSD Socket API for Simulator decreases throughput by about 32.37% and the overall simulation framework decreases throughput by about 79.99%.

6.4 Summary

Three categories of tests are done to the BSD Socket API for Simulator: unit tests, functional tests and efficiency tests.

The unit tests have testified that the 29 socket-related functions in the shared library are just works the same way as that of their counterpart in the BSD Socket API.

The functional tests are taken from three sources, the real world applications are compiled in a normal way and the untouched binaries are run on top of the BSD Socket API. These tests have testified that the BSD Socket API for Simulator supports real world applications to run on top of it.

Compared to the last two categories, the efficiency tests are not quite satisfactory. The good news is that the BSD Socket API for simulator only slows down simulations to a reasonable degree, by around 50% as shown in the tests. The bad news, however, is that it is unclear that to what degree the two factors, NSC and real time scheduler, are slowing down the overall performance of simulations using the BSD Socket API for Simulator.

Chapter 7
Limitations and Future Works

The BSD Socket API for Simulator is now working only with the real time scheduler, so it is more network emulation than network simulation. As shown in previous research, the current real time scheduler has some obvious drawbacks that make it not perfect for network emulation. The ns-2 scheduler has a virtual clock. The virtual clock is used to schedule when to fire next event. In real time scheduler, the virtual clock is synchronized with the system clock by calling gettimeofday(). The synchronization process introduces a problem of delayed execution of events [23]. Accumulated delays make the network protocols behaves strangely [23]. The problems caused by real time scheduler could be solved in two distinct ways:

- The first solution is to use non-real time scheduler instead of the real time scheduler. As stated in chapter 2, the real world applications have to be moved into simulator when using non-real time scheduler. That will be a completely different framework from that of the BSD Socket API for Simulator. It eliminates some complexities such as inter-process communication and message redirecting. But the real world applications have to be modified more or less before they can be integrated into simulator's address space. The problem of such approach is: Does a modified program still behave exactly the same as the original one?
- The second solution is to modify the real time scheduler to make it better. Several efforts have been make in [24].

The BSD Socket API for Simulator is using the real world file descriptor for message redirecting. It may consume more descriptors more than a real world application does. As the file descriptor space is precious and limited system resource, the socket API for simulator is not quite scalable. Distributing simulation onto multiple hosts can solve the problem. As the BSD Socket API for Simulator is using socket to exchange messages between real world applications and simulator, it is, therefore, convenient to modify the current structure and make it a distributed system.

The BSD Socket API for Simulator is using the unreliable UDP for Inter-Process Communication (IPC). There are other possible IPC mechanisms can be used to exchange messages between the shared library and simulator. These IPC

mechanisms may have their own strengths and drawbacks, and the following are two possible options:

- TCP socket. It is reliable and easy to distribute as well. But it is not as efficient as its unreliable counterpart.
- Shared memory. It is reliable and much efficient than both UDP and TCP. But all the real world applications have to be in the same physical host so that they can use the shared memory. It is difficult to distribute simulation using shared memory.

The BSD Socket API for Simulator is designed to be simulator independent. It is currently working with Ns-2, but it is possible to move it to other simulators such as Omnet++ and JSim.

References

[1] W Richard Stevens, Bill Fenner, Andrew M Rudoff. UNIX Network Programming: The Sockets Networking API, Volume 1, Third Edition. Addison-Wesley, 2004.
[2] Sam Janson, Anthony McGregor. Simulation With Real World Network Stacks. Proceedings of the 2005 Winter Simulation Conference.
[3] Warren Gay. Linux Socket Programming by Example. QUE.
[4] David Ely, Stefan Savage and David Wetheral. Alpine: A User-Level Infrastructure for Network Protocol Development. Department of Computer Science and Engineering University of Washington, Seattle WA.
[5] Christopher C Knestrick. Lunar: A User-Level Stack Library for Network Emulation. Virginia Polytechnic Institute and State University, 2004.
[6] Roland Bless, Mark Doll. Integration of the FREEBSD TCP/IP-Stack Into the Discrete Event Simulator OMNET++. Institute of Telematics, University of Karlsruhe, 76128 Karlsruhe, Germany.
[7] X.W. Huang, R.Sharma, and S.Keshav. The ENTRAPID Protocol Development Environment. Cornell Netwrok Research Group. Department of Computer Science. Cornell University, Ithaca, NY 14853.
[8] S.Y.Wang, C.L. Huang, C.C. Hwang, Z.M. Yang, C.C. Chiou, and C.C. Lin. The Design and Implementation of the NCTCns 1.0 Network Simulator. Departmetn of Computer Science and Information Engineering National Chiao Tung University, Hsinchu, Taiwan.
[9] S.Y.Wang, C.L. Huang, C.C. Hwang, Z.M. Yang, C.C. Chiou, and C.C. Lin. Using the NCTUns 2.0 Network Simulator and Emulator to facilitate Network Researches. Department of Computer Science and Information Engineering National Chiao Tung University, Hsinchu, Taiwan.
[10] S. Y. Wang and H.T. Kung. A Simple Methodology for Constructing Extensible and High-Fidelity TCP/IP Network Simulators. Division of Engineering and applied Sciences Harvard UniversityCambridge, MA 02138, US**A.**
[11] Marco Zec, Miljenko Mikuc. Real-Time IP Network Simulation at Gigabit Data Rates. Faculty of Electronic Engineering and Computing, University of Zagrab
[12] Kevin Fall, Kannan Varadhan. The Ns Manual. The Vint project. A Collaboration between Researchers at UC Berkley, LBL, USC/ISI, and Xerox PARC. February 26 2006
[13] Luigi Rizzo. An Embedded Network Simulator to Support Network Protocols' Development. 9th International Conference on Computer Performance Evaluation: Modelling Techniques and Tools, St. Malo, France, June 1997.
[14] Joe Touch, John Heidemann, and Katia Obraczka. Aug, 16, 1996.

USC/Information Sciences Institute. USC/ISI Research Report 98-463/Dec. 1998.
[15] Sam Liang, David Cheriton. TCP-RTM: Using TCP for Real Time Multimedia Applications. Distributed Systems Group, Stanford University.
[16] Andrei Sukhov, Prasad Calyam, Warren Daly, Alexander Iliin. Network Requirement for High-Speed Real-Time Multimedia Data Streams. Laboratory of Network Technologies, Samara Academy of Transport Engineering, Samara. Department of Electrical and Computer Engineering, Ohio State University, Ohio USA. HEANet Ltd, Dublin, Ireland. Russian Institute for Public Network, Moscow, Russia.
[17] R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Tayler, I. Tytina, M. Kalla, L. Zhang, V. Paxson. Stream Control Transmission Protocol, RFC2960. Network Working Group. http://www.rfc-editor.org/rfc/rfc2960.txt.
[18] The Open Group, The Single UNIX Specification, Version 3. Read/Download IEEE Std 1003.1, 2004 Edition, Single UNIX Specification Version 3. Website http://www.unix.org/version3
[19] Richard M Stallman and the GCC developer community. GCC 4.1.1 Manual, Using the GNU Compiler Collection (GCC). Website http://gcc.gnu.org/onlinedocs/.
[20] SearchNetworking.com. SearchNetworking.com Definitions. Website: http://searchnetworking.techtarget.com/sDefinition/0,,sid7_gci212614,00.html.
[21] The GNU C Library Reference Manual, for Version 2.3.x. website: http://www.gnu.org/software/libc/manual/
[22] UNIX Network Programming, volume1, Third Edition Source Code. Website: http://www.unpbook.com/src.html.
[23] Daniel Mahrenholz, Svilen Ivanov. Howto: Wireless network emulation using ns2 and Distributed Applications. Version 1.0. University of Magdeburg, Germany. 15th December 2004.
[24] Daniel Mahrenholz, Svilen Ivanov. Real-time Emulation with ns-2. University of Magdeburg Germany.

Appendix A
Function List of
BSD Socket API for Simulator

```
int socket(int domain, int type, int protocol);
int bind(int sockfd, struct sockaddr *my_addr, socklen_t addrlen);
int connect(int    sockfd,    const    struct sockaddr *serv_addr,
                    socklen_t addrlen);
int listen(int s, int backlog);
int accept(int s, struct sockaddr *addr, socklen_t *addrlen);
int socketpair(int d, int type, int protocol, int sv[2]);

ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
int readv(int filedes, const struct iovec *vector,
                              size_t count);
int writev(int filedes, const struct iovec *vector,
                              size_t count);
ssize_t send(int s, const void *buf, size_t len, int flags);
ssize_t sendto(int    s,    const    void *buf, size_t len, int flags, const
                                 struct sockaddr *to, socklen_t tolen);
ssize_t sendmsg(int s, const struct msghdr *msg, int flags);
ssize_t recv(int s, void *buf, size_t len, int flags);
ssize_t    recvfrom(int s, void *buf, size_t len, int flags, struct sock-
                   addr *from, socklen_t *fromlen);
ssize_t recvmsg(int s, struct msghdr *msg, int flags);

int close(int fd);
int shutdown(int s, int how);

int dup(int oldfd);
int dup2(int oldfd, int newfd);

int ioctl(int d, int request, ...);
int    getsockopt(int    s, int level, int optname, void *optval,
                    socklen_t *optlen);
int setsockopt(int s, int    level,    int    optname,    const    void    *optval,
```

```
                    socklen_t optlen);

int getsockname(int s, struct sockaddr *name, socklen_t *namelen);
int getpeername(int s, struct sockaddr *name, socklen_t *namelen);
int fcntl(int fd, int cmd, ...);
int select(int n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
                    struct timeval *timeout);
int pselect(int      n,      fd_set      *readfds,    fd_set    *writefds,    fd_set
           *exceptfds, const struct timespec *timeout, const sigset_t *sigmask);
int poll(struct pollfd *ufds, unsigned int nfds, int timeout);
```

Appendix B
Kernel Data Structures

```
/*
 *The following stuff is taken from
 * /usr/include/bits/socket.h
 * /usr/include/sys/socket.h
 * /usr/include/netinet/in.h
 * /usr/include/netdb.h
 * /usr/include/sys/uio.h
 * /usr/include/linux/socket.h
 * /usr/include/sys/poll.h
 * due to the name conflicts, a number of data structures have to be defined here.
 */
struct sockaddr
  {
    __SOCKADDR_COMMON (sa_);    /* Common data: address family and length. */
    char sa_data[14];                   /* Address data.    */
  };
/* Structure describing messages sent by
   `sendmsg' and received by `recvmsg'.    */
struct msghdr
  {
    void *msg_name;                 /* Address to send to/receive from.    */
    socklen_t msg_namelen;          /* Length of address data.    */
    struct iovec *msg_iov;          /* Vector of data to send/receive into.    */
    size_t msg_iovlen;              /* Number of elements in the vector.    */
    void *msg_control;              /* Ancillary data (eg BSD filedesc passing). */
    size_t msg_controllen;          /* Ancillary data buffer length.    */
    int msg_flags;                  /* Flags on received message.    */
  };
/* Structure used for storage of ancillary data object information.    */
struct cmsghdr
  {
    size_t cmsg_len;            /* Length of data in cmsg_data plus length of cmsghdr
structure.*/
    int cmsg_level;             /* Originating protocol.    */
    int cmsg_type;              /* Protocol specific type.    */
```

```
#if (!defined __STRICT_ANSI__ && __GNUC__ >= 2) || __STDC_VERSION__ >=
199901L
      __extension__ unsigned char __cmsg_data __flexarr; /* Ancillary data.    */
#endif
   };
struct ucred
{
   pid_t pid;                                   /* PID of sending process.    */
   uid_t uid;                                   /* UID of sending process.    */
   gid_t gid;                                   /* GID of sending process.    */
};
/* Structure used to manipulate the SO_LINGER option.    */
struct linger
   {
      int l_onoff;                          /* Nonzero to linger on close.    */
      int l_linger;                          /* Time to linger.    */
   };
/*
   *The following data structures are from
   *    /usr/include/netdb.h
 */
struct hostent
{
   char *h_name;                                    /* Official name of host.    */
   char **h_aliases;                                /* Alias list.    */
   int h_addrtype;                                   /* Host address type.    */
   int h_length;                                     /* Length of address.    */
   char **h_addr_list;                               /* List of addresses from name server.    */
#define              h_addr              h_addr_list[0] /* Address, for backward compatibility.    */
};
// Internet address (a structure for historical reasons)
struct in_addr {
      unsigned long s_addr; // that's a 32-bit long, or 4 bytes
};
struct sockaddr_in {
      short                sin_family;      // Address family
      unsigned short    sin_port;           // Port number
      struct in_addr       sin_addr;        // Internet address
      unsigned char          sin_zero[8]; // Same size as struct sockaddr
};

//struct iovec from sys/uio.h
struct iovec {
   void *iov_base;      /* Starting address */
```

```
   size_t iov_len;        /* Number of bytes */
};
struct pollfd
   {
      int fd;                          /* File descriptor to poll.    */
      short int events;                /* Types of events poller cares about.    */
      short int revents;                /* Types of events that actually occurred.    */
   } ;
```

Appendix C
Modifications to Code from "UNIX Network Programming"

The "UNIX Network Programming, The Sockets Networking API, Volume 1, Third Edition" is accompanied by examples. The source code of these examples can be found at its official website: http://www.unpbook.com/src.html. The source code is downloaded as a tarfile: unpv13e.tar.gz. Files are extracted from the tarfile simply by:

```
tar –xvzf unpv13e.tar.gz
```

Change directory to the root directory of examples by:

```
cd unpv13e
```

The file README can be found there. It contains instructions to configure and compile these examples.

The first step is to figure out all implementation differences by issuing:

```
./configure
```

And then build the basic library that all programs need.

```
cd lib
make
cd ../libfree
  make
```

If your system supports BSD 4.4/UNIX 98 address structure:

```
cd ../libroute
make
```

It is very important to check if your system is using the socket address structure as:

```
struct sockaddr {
      uint8_t sa_len;
      sa_family_t sa_family;
      char sa_sa_data[14]
};
```

Sockets use the sockaddr address structure to pass and to receive addresses. The structure above is defined in UNIX 98/BSD 4.4. But in BSD 4.3 a different socket address structure is defined as:

```
struct sockaddr {
      u_short sa_family;
      char sa_data[14]
};
```

So, if compiling files in “libroute” directory on BSD 4.3, there will be several errors due to the missing structure member “sa_len”.
XTI (X/Open Transport Interface) is an extension to and refinement of TLI (Transport Layer Interface). Both the TLI and XTI library interfaces are available as part of the UNIX system transport interface.

```
cd ../libxti
make
```

If all of the instructions above work, net step is to compile individual programs. The following sections demonstrate compilation of programs used in the functional tests of BSD Socket API for Simulator.

C.1 Compilation of Servers

There are several versions of servers in director “server/”, to compile the servers:

```
cd ../server
make
```

Unfortunately, the compilation of the servers ended with an error shows:

lock_pthread.c:20: undefined reference to ‘Pthread_mutexattr_setpshared’

The name “Pthread_mutexattr_setpshared” does not exit in pthread.h, it should be “pthread_mutexattr_setpshared”. It is a function to set the current pshared attribute for the mutex attribute object. After changing the function name, the servers compile successfully.

C.2 Compilation of clients using select ()

Three TCP clients exist in directory “select/”, these clients make use of the select () function to achieve parallelism. To compile the clients:

```
cd ../select
make
```

C.3 Compilation of TCP clients and servers

In directory “tcpcliserv/”, there are more TCP client/server examples, these examples cover every aspects of TCP client/server communication. When compiling these programs in LINUX, there are two undeclared identifier found: “OPEN_MAX” and “POLLRDNORM”. In UNIX, OPEN_MAX is the maximum number of file descriptors can be open per process. In LINUX, to check this number:

cat /proc/sys/fs/file-max

The number in the test machine is 789304. It can be set only by system administrator.

The other undeclared identifier “POLLRDNORM”, defined in <sys/poll.h>, is a bit mask.

To solve the two missing symbols, a simple way is taken here: two symbols are simply defined as:

```
#define MAX_OPEN 789304
#define POLLRDNORM 0x0040
```

The two definitions above are added to “lib/unp.h”.

C.4 Compilation of UDP Clients and Servers

The UDP servers and clients are in directory “udpcliserv/”, these programs can be compiled as:

```
cd ../udpcliserv
make
```

C.5 Socket Options

There are socket options test programs in directory “sockopt/”. When compiling these programs, there are lots of errors. These errors are actually caused by an undeclared “SO_USELOOPBACK”. A new definition is added to “lib/unp.h”.

```
#define SO_USELOOPBACK 0x0040
```