Lunar: A User-Level Stack Library for Network Emulation

Christopher C. Knestrick

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
In
Computer Science

Dr. Srinidhi Varadarajan (Chair)
Dr. James D. Arthur
Dr. Scott F. Midkiff

February 24, 2004
Blacksburg, Virginia

Keywords: Networking, Protocol Testing, Direct Code Execution, Simulation

Lunar: A User-Level Stack Library for Network Emulation

Christopher C. Knestrick

(ABSTRACT)

The primary issue with developing new networking protocols is testing how the protocol will behave when deployed on a large scale; of particular interest is how it will interact with existing protocols. Testing a protocol using a network simulator has drawbacks. First, the protocol must be written for the simulator and then rewritten for actual deployment. Aside from the additional work, this allows for software bugs to be introduced between testing and implementation. More importantly, there are correctness issues. Since both the new and existing protocols must be specially written for the simulator, and not actual real-world implementations, the question remains if the behavior observed and, specifically, the interactions between the protocols are valid. Direct code execution environments solve the correctness problem, but there is the loss of control that a simulator provides. Our solution is to create an environment that allows direct code execution to occur on top of a network simulator. This thesis presents the primary component of that solution: Lunar (Linux User-level Network Architecture), a user-level library that is created from the network stack portion of the Linux operating system. This allows real-world applications to link against a simulator, with Lunar serving as the bridge.

For this work, an implementation of Lunar was constructed using the 2.4.3 version of the Linux kernel. Verification testing was performed to demonstrate correct functioning of the library using both TCP (including TCP with loss) and UDP. Performance testing was done to measure the overhead that Lunar adds to a running application. Overhead was measured as the percent increase in the runtime of an application with Lunar as compared to the application running without it, and ranged from approximately 2% (running over 100 Mbps switched Ethernet) to approximately 39% (1 Gbps Myrinet).

Acknowledgements

I would like to thank my committee members, Dr. Varadarajan, Dr. Arthur and Dr.. Midkiff for all their help, both with regards to this project, as well as other aspects of my graduate life. In particular, I'd like to especially thank my advisor, Dr Varadarajan, for providing me with a project that has enabled me to accomplish all the objectives that established for myself when I decided to return to school.

Thanks to my parents, who gave me love and support over the years, as well as a place to stay and wonderful meals when I absolutely had to get out of Blacksburg for a few days.

Thanks to Kate McDevitt, who gave me constant support and encouragement. An extra special thanks for the wonderful help proofreading the rough draft of this document (and helping get rid of all those semicolons!)

Thanks to the wonderful office staff in the Computer Science department, especially Carol Roop, Tess Sentelle and Ginger Clayton-Allen. They have helped me in more ways than I can list (or even remember!). Without them, I'd still be wondering what happened to my paychecks!

Thanks to my fellow students who have helped me on this thesis. Thanks to Joy Mukherjee who helped with the section on Weaves, and Kumar Thirunavukkarasu, who proofread my initial draft of Chapter 1.

Most of all, I want to thank Rob and Miranda Capra, for their unlimited patience, encouragement and support (as well as loads of help with the bibliography!). They kept my spirits up when I thought that I would never be finished and made sure that I finally did. They also taught me an invaluable lesson: cats can be wonderful pets!

This research was partially funded as part of the Integrated Research and Education in Advanced Networking (IREAN) program at Virginia Tech, an Integrated Graduate Education and Research Training (IGERT) program of the National Science Foundation (Award DGE-9987585).

Table of Contents

List of Figures

Chapter 1. Introduction

The primary problem with developing new networking protocols is testing how the protocol will behave when deployed on a large scale. A protocol that works well on a small scale may have unforeseen consequences when it is released on a much larger scale; the occurrence of an unlikely event becomes a certainty as the size of the network grows [1].

A key area of concern when testing is how a given protocol will interact with other existing protocols. This may include both "vertical" interactions, in which one protocol utilizes the second as a lower layer (e.g. DNS over UDP), as well as "horizontal" ones, either between two distinct protocols (e.g. SSH and FTP) or between two instances of itself (e.g. TCP interacting with another TCP).

The classic example of poor protocol interaction (a "vertical" interaction in this case) is that of HTTP 1.0 and TCP [2]. Originally, the HTTP protocol would create a new TCP connection for each item that it requested from a server. For example, if a web page contains three images, the client would make four HTTP requests - one for the HTML document and one for each image – each requiring a separate connection. This method results in several inefficiencies in terms of network utilization. First, each connection incurs the overhead of connection setup and shutdown. TCP employs a three-way handshaking protocol to open a connection, as well as an additional two packets to close it. While it is possible to piggyback data along with two of the five packets needed for connection management (the final packet of the startup and the first packet of the shutdown), even then the setup and shutdown requires three packets that carry no data. If the amount of data that each connection transmits is small (for example, the HTML text file), the percentage overhead of connection management can be considerable.

In addition to the overhead incurred by opening and closing each connection, the short-lived connections created by HTTP suffer from the added problem of poor bandwidth utilization. This is due to the fact that TCP employs a slow-start algorithm [3] as a congestion control mechanism. Rather than transmitting packets as fast as possible, TCP "tests the waters" by setting the initial window size (that is, the number of packets that will be transmitted before an acknowledgement is required) to one[1]. The size of the window is doubled each time a complete window is transmitted successfully. When a threshold window size is reached, slow-start is terminated and the window size is increased linearly by one packet for each successive window. A dropped packet implies network congestion and consequently TCP will reduce its window size by half.

A short-lived HTTP connection will never exit the slow-start phase, and consequently will not be fully utilizing available network bandwidth. Each new connection that is created to retrieve the remaining items on a page will then begin the slow-start phase anew, with a window size of one, further wasting bandwidth.

[1] Technically, TCP works with the number of bytes sent; a packet can be thought of as a specified number of bytes.

The root cause of these problems is a poor interaction between HTTP and the underlying transport protocol, TCP. HTTP was designed to use numerous short-lived transactions, while TCP favors long-duration connections for better resource utilization.

An example of "horizontal" interference can occur when a large number of TCP streams are created nearly simultaneously and all attempt to acquire bandwidth via slow-start. An example of this is a large parallel application where each node may create a number of interconnections with other nodes at startup. Each TCP stream is doubling its window size as part of slow-start, which results in the available bandwidth quickly being consumed. In reaction, all of the streams immediately half their window sizes, freeing up a large amount of bandwidth (bandwidth which is supposedly unavailable, which is why the window size was cut in the first place). The streams are synchronized, so this process will continue; they will increase their window sizes, experience congestion, and cut them again. This problem is analogous to Ethernet retransmissions due to collisions, in which it is possible for two hosts to wind up synchronized such that they continue to retransmit at the same time, resulting in an endless series of collisions. Ethernet provides a mechanism to add randomness to the amount of time it waits between collisions to prevent this; TCP does not provide an equivalent mechanism to stagger connection starts. The result of this interference between the different TCP streams is that the network bandwidth is poorly utilized.

1.1 Simulation and Direct Code Execution

In general, the approaches to protocol testing can be classified into two categories: simulation and direct code execution (sometimes referred to as emulation, since the environment emulates the properties of the underlying network). Although in practice the boundaries are not quite as clear-cut, this dichotomy provides a useful classification (as well as one that is commonly used).

Simulation models discrete events, such as the transmission or reception of a data packet, over a virtual network topology. Within a simulation, time is also a virtual quantity. Each event has an associated time, and the flow of time advances as each event is processed. Simulators often take advantage of the inherent parallelism that exists within networks, resulting in parallel discrete event simulators (PDES) that run on workstation clusters. This greatly improves the scalability of such simulators. Current network simulators include REAL [4], ns-2 [5], and OPNET Modeler [6].

The key problem with simulation is that its implementation of a protocol is not a "real-world" implementation. Rather, it is one specifically written for that particular simulator in a discrete event model and consequently may have behavior that differs from the actual protocols that are deployed across the Internet. This problem is discussed by Brakmo and Peterson [7], and some specific instances of this that the authors encountered are described. For example, the clock on the real-world implementation that was used (BSD) ticked twice a second. This meant that round-trip time (RTT) measurements had a granularity of only 500 ms, resulting in retransmit timeouts that were considerably greater than the real RTT. On the other hand, the simulators often had a much finer RTT

granularity, and consequently much smaller and more accurate retransmit timeout intervals. Because of this, packet loss on the simulator would yield a lower loss of throughput (since the retransmit timer would expire much sooner) than would be experienced by the real-world implementation.

It is also possible that a given implementation may deviate – intentionally or otherwise – from the protocol specification. In the case of the Linux kernel, an example of this is the amount of time that a TCP connection will remain in the CLOSE-WAIT state. The TCP standard [8] specifies that the operating system must retain the connection information for four minutes after the connection has been closed in order to catch any packets that may arrive late. The problem with this is that the port used for the connection cannot be reused until after this time has expired, preventing an application (specifically a server which uses a predefined port) from binding to it for the prescribed time. Consequently Linux, like many other operating systems, violates the specification and shortens this length to only 60 seconds.

While these examples are specific to particular implementations, the problem is a general one. Each operating system's implementation of the various networking protocols can be expected to contain its own set of idiosyncrasies. While some may be inconsequential, others may have a significant impact on network behavior.

The issue of real versus simulator implementation leads to another problem associated with network simulation, the transfer of a protocol from a simulated environment to an actual operating system. There is the additional work that is required in order to rewrite the simulator code into the operating system's native language (in the case of Linux, C). But with this additional coding comes the possibility of the introduction of new bugs into the final implementation. Validation issues may still remain: is the deployed protocol equivalent to the one that was simulated?

The second of the two modeling approaches addresses both of these problems. Direct code execution environments (DCEEs) provide a testbed in which the network stack from a running operating system is used instead of a simulated model. Such environments, which function in real time, are attached to physical networks and operate by introducing and manipulating such parameters as delay and jitter to model the desired network conditions. This is usually accomplished by adding hooks into the kernel to control these conditions. Some examples of DCEEs are NIST Net [9], dummynet [10], and ENTRAPID [11].

While DCEEs solve correctness issues, they have problems of their own. Since they operate in real time, the temporal control that a virtual time simulator provides is lost. Of particular importance is the ability to replicate a series of events, which is often not possible when dealing with an underlying physical network. Scalability becomes a concern since each host (or at least, each end host) must be associated with a physical machine. Another problem is that when using a physical network, the highest link capacity that can be emulated is restricted by what is available on that network. Additionally, debugging can be troublesome when dealing with a running stack. Under

Linux, for example, it is possible to debug a running kernel using *gdb*, but this only provides the ability to view the current value of variables. It does not allow the user to step through each line of code.

Our goal is to create a protocol testbed that provides the correctness of direct code execution while allowing the control provided by network simulation. This thesis presents Lunar (Linux User-level Networking Architecture), a user-level network stack library designed to help achieve this goal by allowing real-world networking code to execute on top of an underlying simulated environment.

1.2 Our Work

Lunar consists of the networking portions of the Linux operating system (kernel version 2.4.3) that have been extracted from the kernel and compiled as a user-level library, which can then be linked against application code. To the application code, the library provides the interface functions necessary for network communication (the BSD sockets API as well as other related functions). From the standpoint of the code, it is unaware that these system calls are being serviced by anything other than the host operating system. This provides the desired functionality of a direct code execution environment:, application code utilizing real-world networking protocols.

While a user application is linked "on top" of Lunar, "underneath" it lies a network simulator, which interacts with the library via its own set of API functions. While traffic is generated by real-world code, the flow of time is regulated by the simulation layer. This provides the control benefits associated with network simulation, which is absent in pure direct code execution environments.

There are several additional benefits of this method over existing DCEEs which use running kernels to emulate a network. First, when modifying a protocol, it is easier to modify and recompile a user library than a full kernel; if the protocol is built into the core kernel rather than as a module, it will require a reboot in order for the changes to take affect.

Second, it is possible to separate the kernel being tested from the machine's operating system. If a modification "breaks" the stack, the host machine will be unaffected. This can be especially important if the machine is part of a cluster where console access may be difficult.

Finally, it allows for easier debugging since the library resides in user-space and does not require attempting to debug a running kernel.

The next section provides a more detailed description of context in which Lunar operates. In particular, the relationship between the application, library, and simulator is explored.

1.3 Architecture Overview

Lunar is designed as one component of a larger project, the Open Network Emulator (ONE), a large-scale network emulation testbed. The library acts as a bridge between the real-world application code executing on top of it and the simulation layer beneath it.

1.3.1 Open Network Emulator (ONE)

As stated in [13], the goals of the ONE are:

1. To provide a protocol development environment that closely models *real-world* networks. This requires scalability to the order of tens of thousands of virtual network nodes, where a network node is an end-host, a router or a network switching device.
2. To support execution of unmodified protocol and application code.
3. To integrate direct code execution and protocols simulation within a single framework. In the traditional model, direct code execution operates in real time and hence lacks the controllability of virtual time simulation.

Figure 1.1 shows the architecture of the ONE (based on Figure 1 of [13]). The principle logical unit of the ONE is the *virtual host*, which consists of one or more instantiations of a networking application, such as telnet, FTP, or HTTP, and an IP stack in the form of our library. Multiple virtual hosts reside on a single *physical host*, which is, in turn, part of a cluster of workstations. The underlying application which ties all the virtual hosts

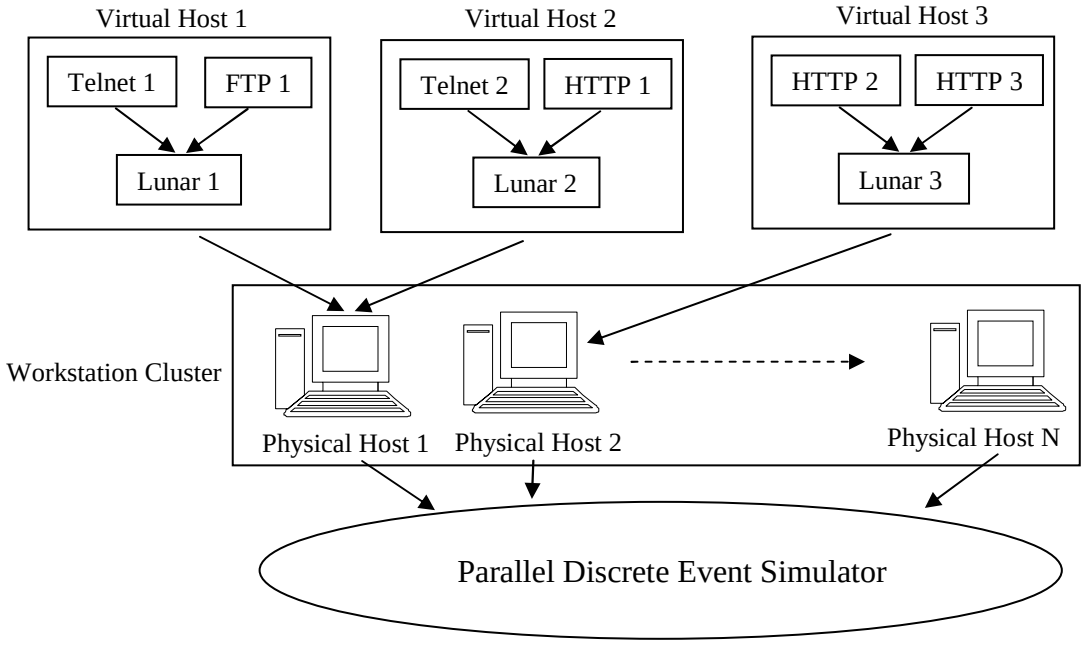

Figure 1.1: Architecture of the ONE

together is a parallel discrete event simulator (PDES), which includes facilities for statistics gathering, fault generation, and traffic visualization.

Since scalability is critical, the goal is to minimize the overhead associated with each virtual host. One of the key mechanisms to achieve this is to limit the number of processes on the host machine, since context switching between them is expensive. While Figure 1.1 shows the logical representation of the virtual hosts, it does not provide any details of their physical representation on the host machines. At first glance, it may appear that each virtual host constitutes a single process, but even that incurs too much of a performance penalty. Instead, for a physical host, all virtual hosts – including applications, Lunar, and simulator code – are combined together in a single process. To accomplish this, a new programming model, the Weaves Framework, is introduced.

1.3.2 Weaves Framework

While the inclusion of a user-level stack library removes correctness issues, it does nothing in and of itself to address scalability concerns. For that, we require the addition of another component, the Weaves Framework, a new generalized programming model. Within the context of this model, the two primary existing models, *processes* and *threads*, exist at the extremes. The benefits of this model are exploited by working between these two extremes. A complete discussion of the Weaves Framework can be found in [14] . This section provides a brief overview of the relevant aspects of Weaves as they relate to Lunar.

In the *process* model, each process is self-contained, having its own code and data context; neither is shared between separate processes. One of the drawbacks of this model is the high cost of context switching between different processes. Additionally, each process requires its own set of operating system data structures, consuming valuable memory, as well as the time required to create and maintain these structures. These problems lead to the creation of a new programming model, the *thread* model, also known as *lightweight processes*.

In the *thread* model, each process contains multiple flows of execution, which can be switched with much lower overhead than is required when switching between separate processes. In contrast with processes, each thread shares both code and global data context with every other thread. The downside to this feature is that global data must be protected via mutexes to prevent race conditions and maintain consistency. This means that code that utilizes threads must be “thread-safe,” i.e. written such that it contains the necessary protections of global data. Making existing code thread-safe can often times be a non-trivial exercise.

The Weaves Framework provides a third programming model in which code context is shared, while data is maintained independently for each flow of execution. This provides the low context switching overhead of the threads model, while eliminating the problems associated with shared global data. Consequently, code does not need to be written with

any knowledge of Weaves; there is no need to write "Weaves-safe" code[2]. The low overhead of the Weaves model is what provides the scalability that is required by the ONE.

The simplest way to describe the Weaves Framework is to use the example application that provides the motivation for this work, the ONE. We will consider one physical node which consists of three logical elements:

- Application code that implements the protocols (telnet, FTP, etc) of interest
- Simulator code that implements the simulation layer
- Lunar library that links the application with the underlying simulator

The basic element of the Weaves Framework is the *module*. A module is a piece of logically related object code. In our example, the simulator code, Lunar, and each application would each be a module. Each module provides an application-specific API used to interface to any other modules that it wishes to communicate with at runtime. For example, Lunar has two APIs defined, one for the simulator and one for the application (both of these are discussed in detail in later sections). At runtime, each module is instantiated one or more times, with each instantiation called a *bead*. Logically related beads are then grouped together to form a *weave*, representing a program with its own

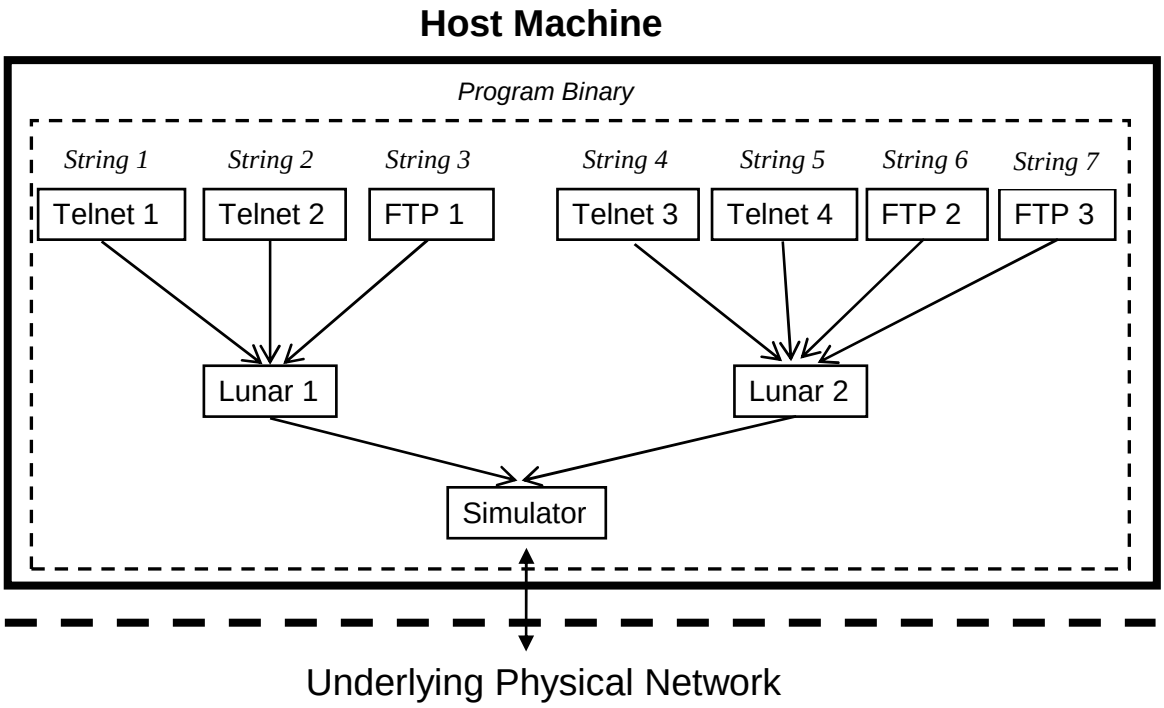

Figure 1.2: Example ONE Tapestry

[2] This description is somewhat simplified. In reality, Weaves allows global data to be shared selectively. Code written specifically for the Weaves framework can utilize this feature, but the default implementation uses independent data spaces. This feature is beyond the scope of this work and is not discussed further.

complete namespace (requiring no external references). A *string* is a flow of control which exists within a weave. A weave may have more than one string, but a string may not span more than one weave. The complete collection of weaves forms a *tapestry*, which exists on the host machine as a single process. Figure 1.2 shows an example tapestry. The architecture of the tapestry – the beads that are instantiated and the weaves (and their associated strings) formed by those beads – is specified at runtime. By a simple modification of the configuration file, the architecture can be changed between runs, without requiring any code changes or recompilation. In the example tapestry, each application-Lunar-simulator set forms a weave, each containing a flow of execution, which can be labeled, left to right, as Strings 1 through 7. Weaves provides a scheduler that serves as an executive authority for the tapestry and controls which string is executing at a given time. This is accomplished using a combination of both preemptive and cooperative scheduling.

Using preemptive scheduling, each string is allotted a timeslice during which to run. When that time expires, the Weaves Scheduler switches to the next string. The selection of the next string to execute is subject to certain restrictions based upon the code that is currently executing when the context switch takes place. Using a combination of the string ID and the current instruction pointer, the Scheduler is able to determine the bead currently executing. The string that is selected next must be one whose weave does not also consist of that bead. For example, if the initial string that is executing is String 1 (Telnet 1–Lunar 1–Simulator), and execution stops while within the Lunar 1 bead, the next string to execute must be one that uses Lunar 2 (Strings 4–7). The Scheduler can not switch to String 2 or 3 since they both use Lunar 1, which is currently executing under String 1. If, on the other hand, String 1 was executing within Telnet 1 when it was switched out, the Scheduler would be free to switch to any other string, since none of them make use of Telnet 1.

The Scheduler also makes use of cooperative scheduling, in which the application explicitly yields control while waiting for an event. In the case of a networking application, this is usually waiting for a data packet to arrive. In doing so, the Scheduler is signaled that the bead is now “safe” to use, removing the restrictions described above.

1.4 Thesis Contribution

The contribution of this thesis is the creation of a functional user-level stack library that, in conjunction with the Weaves Framework, will allow direct code execution to occur in a simulation environment. The result of this is the ability to conduct large-scale protocol development and testing in an environment which provides both the benefits of a network simulator as well as a direct code execution environment. This work provides a blueprint, in the form of a catalog of required changes, for the creation of a similar library in future versions of the Linux kernel. Additionally, an API is defined that will allow the incorporation of the network stacks of other operating systems to be added in the future.

1.5 Organization

The remaining sections of this thesis are arranged as follows. Chapter 2 provides a review of related work. Chapter 3 describes the implementation of Lunar, including the library's APIs as well as its configuration mechanisms. Chapter 4 details the testing process and associated results. Included as part of this is an overview of the experimental architecture used as a substitute for the PDES. Chapter 5 consists of a brief conclusion and a discussion of future work. Chapter 6 contains a list of references used in this work. Appendix A provides a list of Lunar's API functions exported to the application level. Appendix B provides a series of tables documenting each of the changes made to the original kernel source files as part of the implementation. Appendix C contains a list and brief description of additional files used by the library.

Chapter 2. Survey of Related Work

This chapter provides an overview of related work. As discussed in the introduction, existing solutions can be broadly classified as either simulation or direct code execution. Since our work falls into the category of direct code execution (the simulation layer of the ONE is a separate component), those approaches will be the primary focus of this chapter. For completeness, a brief survey of the more common network simulators will also be presented.

Since classification is never as clear-cut in practice as it is in theory, some work may not fit directly into either category as described in the previous chapter. For example, in many cases, systems attempt to take advantage of the benefits of simulation underneath direct code execution (the ONE falls into this category). The defining criterion used in this section is that direct code execution environments use protocol implementations from an operating system to generate packets.

2.1 Simulation

REAL [4] is a network simulator designed by Srinivasan Keshav at Cornell Univeristy. Users provide the simulator with a scenario file that describes simulation parameters such as network topology and the protocols used. The simulator's output consists of relevant statistics organized on a per-node basis. Statistics for each node include the number of packets transmitted, dropped, and retransmitted, queuing delays (minimum, average, and maximum), and round-trip-times (minimum, average, and maximum).

Protocols are modeled as modules, and include TCP (Reno and Tahoe), DEC-bit, telnet, FTP, as well as various other types of generic schemes to generate traffic. The modules are written in C and can be modified by the user (the complete source code is included). Additionally, this modular design enables users to write new protocol modules and add them to the simulator.

A descendant of REAL is ns-2 [5]. The simulator itself is written in C++, but provides a Tcl interpreter that is used as an interface. The interpreter allows for easy configuration of simulations, while much of the computationally intensive code is implemented in C++. Like REAL, ns-2 provides a built-in set of protocols such as TCP and UDP, as well as application-level protocols such as FTP and telnet that are used to generate traffic. More generic traffic types, such as constant-bit-rate and Pareto, are also supported.

One key advantage ns-2 has over REAL (and other simulators) is the ability to incorporate a physical network into the simulation. This allows packets generated by physical hosts on a network to be read into the simulator, and packets from the simulator to be written to the network. In this way, ns-2 could also conceivably be classified as a direct code execution environment. However, ns-2 is clearly first and foremost a network simulator, providing an extensive and detailed simulation framework. By contrast, its direct code execution capabilities are much less developed (currently, the documentation

describes them as experimental) and constitute an addition rather than an integral component.

The Scalable Simulation Framework Network (SSFNet) [15] is a Java-based network simulator. It is built using the Scalable Simulation Framework (SSF), which provides a generic object-oriented model for discrete-event simulators. Like the other simulators, SSFNet provides its own implementations of various protocols (Level 3 and above) including OSPF, BGP, and HTTP. It also includes a BSD socket-like interface to allow the development of application level protocols.

From the commercial arena, there is OPNET Modeler [6]. Like other simulators, it provides a set of modules that model protocols at all networking layers, including routing protocols as well as specific applications such as e-mail and video conferencing. Even though it is a commercial application, protocol model code is open, allowing users to modify protocols or add new protocols of their own.

While each of the simulators described provides implementations of a wide variety of networking protocols, each is written specifically for that simulator. Developers that wish to modify existing protocols, or to develop new ones, must do so based on the given simulator's API, and not that of the system on which it will be deployed. This is the key difference between our work – and direct code execution in general – and simulators.

2.2 Direct Code Execution

The Harvard simulator [16] provides a combination of simulation and direct code execution, using a modified BSD kernel to provide a virtual-time simulation framework on which real-world networking applications execute. Links are created as tunnel interfaces, pseudo-network interface devices which can be accessed via a file in the /dev directory. Applications transmit packets using these devices; after the appropriate amount of time (propagation delay plus transmission time), the kernel writes the packet to the tunnel device of the receiving process. By performing this procedure numerous times, multiple-hop paths can be modeled. Time within the system is virtual; since the operating system's own network stack is used, the kernel's TCP timers must also function in virtual time. To prevent synchronization, each node maintains its own virtual clock that is randomly offset.

A similar system is dummynet [10], which also uses a modified BSD kernel to emulate the properties of a physical network. dummynet traps calls to a given protocol's (typically TCP) input and output functions and, using a set of internal queues, emulates a link with a specified bandwidth, propagation delay, and router queue size and queuing policy. In contrast to the Harvard Simulator, applications use the machine's loopback device to communicate rather using than tunnels. Additionally, time is measured in real time, not virtual.

The Internet Protocol Traffic and Network Emulator (IP-TNE) [17] incorporates packets generated by a real host into a parallel discrete event simulator environment. Each physical host on a LAN is mapped to a virtual host inside the simulator. When a host

transmits a packet, it is intercepted by the simulator and inserted into the virtual network at the appropriate virtual host. When the packet has been routed through the simulator to the destination host, the simulator transmits the packet to the physical destination host on the LAN. Within the simulator, there are also purely virtual hosts (those that have no corresponding physical host) which implement a minimal version of IP. While this approach is similar to the emulation abilities of the ns-2 simulator, IP-TNE is not classified as a simulator since the virtual topology exists only to emulate a network for physical hosts.

NIST Net [9] allows a Linux-based (kernel 2.4.xx) PC to act as a router which emulates the properties of an entire network between two end hosts. This is accomplished using a kernel module which intercepts packets at the IP layer, introduces the desired parameters such as delay, loss, reordering, or jitter, and then returns them to the kernel stack to be forwarded to their destination. The actions taken on a packet are defined in a table of *emulator entries*, each of which specifies a set of rules (similar to firewall rules) that are used to match a packet and the resulting action to be performed on a packet that matches. Matching rules are based on such properties as source or destination address, source or destination port, or protocol type.

A similar, though much simpler, system is the Ohio Network Emulator [18]. Like NIST Net, it acts as a router between hosts, emulating an intervening network with the prescribed properties. The properties that are modeled, however, are much more limited than NIST Net: transmission delay, queuing delay, and propagation delay. The internal queue size as well as the queue buffer size (the amount of buffer allocated for each queue entry) are also configurable. These properties are set on a per-network interface basis, providing a much coarser granularity of control than NIST Net.

In many of these models, the network is considered a "black box" between end hosts. It introduces certain properties into the data flow, but the details are obscured. This reduces the granularity of the emulation and may miss certain interactions that occur within the interior of the network. Those that model multiple links are generally simple and on a small scale. ModelNet [19] attempts to overcome this by providing finer-grained emulation aimed at large scale topologies. Rather than a single hop emulated on a single machine, ModelNet emulates each individual hop in a network using a core cluster of workstations connected by gigabit links. Each workstation, which uses a modified version of FreeBSD, models multiple links. This architecture allows ModelNet to represent much larger topologies than those using a single host. Traffic is generated by host machines outside of the core (so-called *edge nodes*) running any operating system of choice.

The previous systems all suffer from the same problem, namely, the use of kernel-level implementations of the network stack and, in some cases, modified kernels. The distinction of our work is that it moves the stack to user-level to allow for easier and safer modification and debugging. There are several direct code execution solutions that also use operating system code ported to user-level. One common difference with our work is

their choice of FreeBSD instead of Linux. Since protocol implementations differ, even between closely related operating systems, this is a relevant distinction.

Alpine (Application Level Protocol Infrastructure for Network Experimentation) [20] is a user-level library that implements the FreeBSD 3.3 network stack. It is designed to be integrated with the host machine's stack, using the host's network devices to access a physical (rather than a simulated) network. This requires that Alpine shares both the host's IP address as well as port space.

Packets are sent and received via what is referred to as a *faux-Ethernet driver*, which serves as the interface between Alpine and the host machine. Sending is accomplished using a raw socket, which writes the packets generated by Alpine to the network unmodified. The fact that these packets are generated by Alpine and not the host machine is invisible to the receiving machine.

Receiving data is more complicated since packets destined for Alpine must be separated from those destined for the host machine. The filtering is done based on the local port number. Since the port space is shared, this is guaranteed to be a unique identifier. To synchronize the port allocation between Alpine and the host machine, an additional process acts as a port server. When a port is bound by Alpine, the server binds a dummy socket on the host machine to that port, preventing it from being allocated again. The sever also updates a firewall on the host machine with this port information, allowing Alpine packets to be filtered out prior to being passed up the host's network stack.

Packets are received by Alpine using the *libpcap* packet capture library, which provides a copy of all the incoming packets on the network interface. Like the host's firewall, a packet filter is used to remove those destined to non-Alpine ports.

This architecture tightly ties each Alpine process to its host machine; the library can be thought of as a user-level extension of the host's network stack. This coupling is in sharp contrast to our work, which regards each stack as an independent entity. Each stack contains its own set of network interfaces, and their associated IP addresses, and port space separate from that of the machine on which it runs. This independence, along with the use of a simulator rather than a physical network, provides an increased flexibility not available under Alpine. Furthermore, Alpine is not designed with large-scale protocol testing in mind. Since each emulated host requires its own physical host, scalability is a problem.

The final example of a direct code execution environment is ENTRAPID [11]. Like the ONE, ENTRAPID is a protocol development environment which combines direct code execution with a simulated network. Its goal is to provide exact emulation of real-world protocols, but to do so in a controlled and scalable environment.

The heart of ENTRAPID is the Virtualized Networking Kernel (VNK), an implementation of the 4.4 BSD network stack. An ENTRAPID process can support several hundred VNKs, each representing a separate machine, running simultaneously.

Each VNK can have multiple virtual (running inside the ENTRAPID process) or external (running outside ENTRAPID, and connected via TCP to an internal proxy process) processes associated with it. When a process makes a system call to its VNK, a message is passed to the ENTRAPID task scheduler, which generates a new thread to handle the request. At most one thread can be in a given VNK at a time. This approach to the application-system transition is more intensive than that used in our work. System calls (described in more detail in Section 3.3) are treated as normal function calls within a single flow of control. There is no need for the additional overhead required to dynamically signal, generate and schedule a new thread. Additionally, the use of threads still requires "thread-safe" code.

The VNKs are linked together to form a user-defined virtual topology. It is also possible to send packets from the virtual network to a physical network via the hosts network interface. In this way, the virtual ENTRAID network can interact with other physical hosts, or with other ENTRAPID processes on separate machines.

From a practical side, the problem with ENTRAPID appears to be its availability. It was previously distributed by Ensim [12], which provided a free version to educational institutions. Currently, Ensim is no longer distributing it and it appears that development on it has ceased.

2.3 Chapter Summary

Existing solutions to the problem of network protocol testing can be classified as either simulation or direct code execution. This chapter has examined specific systems from each category, as well as the associated problems with each. The next chapter describes the design and implementation of our solution, Lunar.

Chapter 3. Implementation

In this chapter, the implementation details of Lunar and the methodology used to create it are discussed. The different kernel sub-systems that are required by the network stack for correct functioning are described, along with an overview of the methods used to incorporate them into the library while striving to keep the level of non-networking code to a minimum. Next, Lunar's APIs are discussed. The library has two separate interfaces, one between itself and the application layer, and a second with the underlying simulation layer. Finally, Lunar initialization and configuration is covered.

3.1 Implementation Methodology Overview

This section describes the methods used to build Lunar. The goal is to provide an overview of the "thought process" that was used during this project, and to provide a flavor of some of the implementation issues that were faced and how they were overcome.

In order to work with the Linux kernel, it is first necessary to have an understanding of its operation. This, in many respects, was the most time consuming portion of this work. An excellent reference for this is "Linux Core Kernel Commentary" by Scott Maxwell [21]. The understanding of the kernel that was necessary to complete this project was provided in large part by this source.

The primary issue with building the library was to extract only the relevant portions of the operating system to include as part of the library. The problem with a monolithic operating system design is that an operating system is a complex collection of interrelated parts, and no part exists in isolation of the others. This is particularly true of the network stack. Figure 3.1 shows a partial list of the additional sub-systems that the stack is dependant upon in one form or another:

- Memory Management
- Filesystem
- Process Management
- Interrupts
- Time
- Network Devices
- Low-Level Calls

A further description of each of these subsystems is provided in Section 3.2.

The goal is to provide the services necessary for the networking portion of the code to function as if it were still inside a fully-featured kernel without needing to include the entire kernel code base as part of the library. Invariably, more code will be included than is strictly necessary to perform the required functions. The tradeoff is between the size of the binary and the ease and cleanness of implementation; in this project, the ease of implementation always wins.

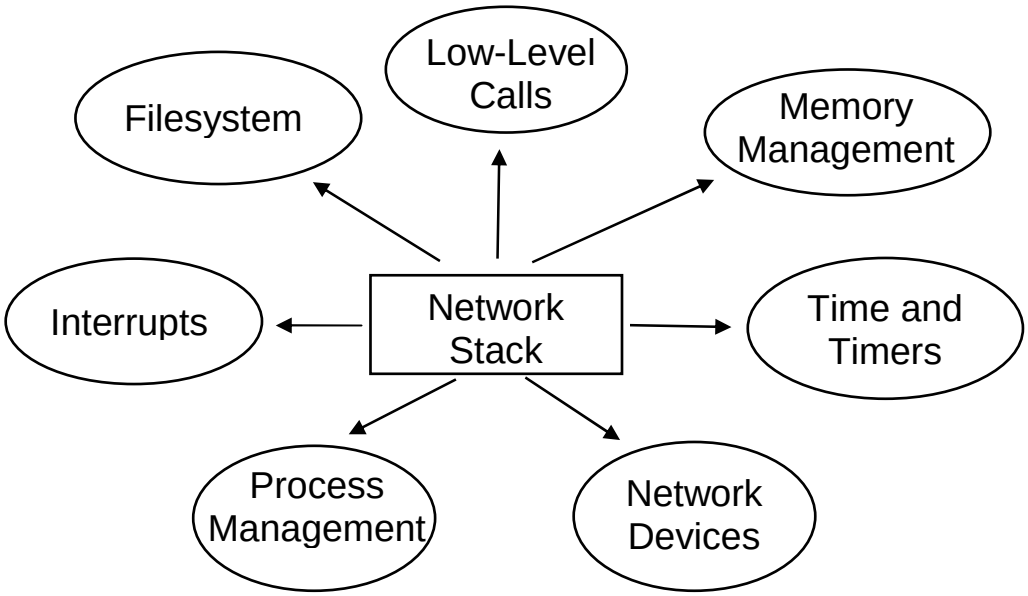

Figure 3.1: Kernel Sub-system Interactions With the Network Stack

The general strategy used to extract the necessary kernel code was one of trial-and-error. There are obvious portions of code that must be included, specifically those that form the core of the TCP/IP stack. It then becomes an iterative process of compiling the library and attempting to resolve all the symbols (variables, functions, macros, etc.) that are not present. This process was far from straight-forward. The naive approach would be to simply identify the files that are necessary to resolve each symbol and include them in the library. The problem with this approach is that it results in a cascade of files that wind up being included. While some of the files are necessary, eventually a point is reached where files are included for a single, isolated symbol, but generate multiple unresolved symbols that are extraneous to the library's operation. The end result is an explosion of code that is added unnecessarily.

At a certain point, decisions must be made about which files to include and how best to take care of the remaining symbols. For a given unresolved symbol, there are three possible options: it can be included, it can be removed, or a workaround can be found.

It may be determined that the symbol is necessary for the library to function correctly, in which case, the necessary code must be added. There are several possible scenarios, depending on the particular code. First, it might prove necessary to include the entire file that contains the function/macro definition or variable declaration. This is the case when a particular function relies on numerous ancillary functions which are all contained in the file. Second, it may be determined that the particular function exists in isolation, in which case it can be extracted and placed in a "stub" file to be used as part of the library. The third and much less frequent possibility is that the entire file is included, but a significant amount of other symbols can be removed. An example of this is the file namei.c, which contains routines that are part of the core of the filesystem implementation, and hence necessary for the library to function properly. However, the

file also includes the system call handler functions for numerous other filesystem-related commands that do not apply to sockets, but only to physical files (`mknod()`, `mkdir()`, etc.). These functions are troublesome because they require numerous other functions, found in other files, to compile correctly. The result is that while the file is included, these unnecessary functions are removed.

A symbol which is unnecessary is one which can be removed entirely. Usually, this situation occurs when a symbol – usually a function or macro – is used to provide a feature which is not supported within Lunar.

There are two methods of removing unneeded symbols; the first is to simply comment[3] out the symbols where they are used. The problem with this approach is that it may require changes in numerous places, depending on the frequency of the particular identifier. The preferred method is to create stub functions that will satisfy the linker. The function bodies themselves are usually empty, provided that no return value is expected; in non-void functions, an appropriate return value is selected. An example of this is the function that tests the capability of the user to perform a certain action. Since security is not a concern and adding the additional kernel code required to implement this feature is superfluous, a return value of true (the integer 1) is always returned.

The final option for handling unresolved symbols is the most interesting from an implementation standpoint. In certain instances, a function is critical to the proper operation of the library, but that function's implementation within the kernel is problematic. In some instances, it may simply be that the dependencies that it generates are numerous and, aside from playing a minor role in this single function, are unneeded. More likely, however, the implementation requires operations that simply can not be accomplished running as a user-level application. The key examples of this are the kernel's memory management functions. In these cases, the solution is to write our own implementation of the affected functions. While this approach seems contradictory to the basic methodology of this project, in practice the code that is written is both small (usually only two or three lines) and obvious in intent. Additionally, it is sometimes the only option.

3.1.1 Stub Files

"Stub" files are additional files (i.e. non-kernel files) that are included as part of the Lunar code base and are used to provide additional code necessary for the proper functioning of the library. They are used to implement the strategies discussed in the previous section. The code in these files includes:

- Stub functions that are needed to satisfy the linker, but which do no useful work
- Kernel code that is needed, but exists in isolation such that the entire file need not, and usually should not, be included
- Library-specific implementations of kernel functions

[3] Commenting out code is used in place of deletion. This allows for easy identification of code portions that have been modified. In some instances, code is removed because a particular feature is not currently supported; future additions may provide support for these features. Uncommenting code is much simpler than attempting to find code that was removed in the original source and re-add it.

The files are named based on the type of code that they contain, with the word "stub" appended e.g. memory_stub.c and timer_stub.c. A description of each of these files can be found in Appendix C.

3.2 Kernel Sub-systems

The networking portion of the Linux kernel interacts with numerous other kernel sub-systems. This section presents an overview of those interactions, and describes how they are implemented in the library. It is not meant to serve as a comprehensive listing of all the changes made integrate these components with the rest of the network stack. Instead, it is meant to highlight some of the key implementation issues of Lunar as well as provide background information that will be used later in this document.

3.2.1 Memory Management

The basic memory management functions include `kmalloc()` and `kfree()`, which are in turn built upon `get_free_pages()` and `free_pages()`. The functions `get_free_pages()` and `free_pages()` access the kernel's low-level memory allocator and provide *physical* memory, which means that this code can not be ported into user level. However, the required memory management functionality is also provided in the user-level by Glibc's `malloc()` and `free()` functions. Although these functions deal in virtual memory, from the standpoint the code, this fact is irrelevant. Therefore, `kmalloc()` and `kfree()` are implemented in Lunar as wrapper functions to `malloc()` and `free()` (the kernel also includes the virtual memory management functions `vmalloc()` and `vfree()` which are implemented in the same manner). The functions `get_free_pages()` and `free_pages()` are handled in an identical manner, with `get_free_pages()` returning memory in multiples of the page size (4096 bytes).

The kernel also utilizes a set of cache functions for commonly used data structures such as sockets and packet buffers. These functions implement a slab allocator memory management scheme (based on a system described in [22]). Since they also rely on the kernel's low-level page allocation routines, they must be implemented in similar manner as the other memory functions.

The function `kmem_cache_create()` is used to create a cache, and returns a pointer to it. The function takes numerous parameters, but the only two of interest in our version are the size of the cache's object (e.g. the socket), and the constructor function that is called when an object is allocated from the cache. The cache data structure is allocated using `malloc()`, the size and constructor function pointer are stored in the appropriate fields, and the cache pointer is returned. Now, allocating an object from the cache in `kmem_cache_alloc()` is simply a matter of using `malloc()` with the prescribed size, calling the constructor function for the object, and then returning the pointer to the newly allocated object.

The functions to free the cache objects are even simpler to implement, as they are nothing more than wrappers to `free()`.

3.2.2 Filesystem

Linux, like all versions of UNIX, handle sockets as a special type of file. Consequently, the networking portion of the kernel is tightly coupled with that of the filesystem. The result is that the core portions of the filesystem must be included as part of the library. These core portions form the Virtual File System (VFS), the generic interface that Linux provides for filesystem operations. Those parts of the VFS that are not used by sockets are eligible for removal if they contain unresolved symbols, such as was described earlier.

3.2.3 Process Management

File descriptors (and hence, socket descriptors) are created on a per-process basis. Internally, file information is stored in the task structure (`task_struct`) associated with each process. Because of this, certain portions of the process management sub-system of the kernel must also be included as part of the library. The code that is incorporated into the library is kept at a minimum, and covers only the basic functions necessary for creating and maintaining a run queue of tasks.

In a production system, the job of creating processes will be managed by Weaves, rather than by the standard methods of `fork()` and `exec()`. Because of this, the functions used to create tasks do not need to be implantations of existing kernel functions (like memory management), but simply functional equivalents that perform whatever work is required.

Of interest are the functions `create_task()` and `switch_to_task()`. The function `create_task()` creates and initializes a `task_struct` with a given PID. It then inserts it into the run queue (a circular, doubly-linked list) as well as the hash table that the kernel uses as an alternate method of finding the task. Given a PID, `switch_to_task()` finds the appropriate task in the run queue and sets it as the current task.

The kernel identifies the running task using the macro `current`, which returns a pointer to the current process' `task_struct`. This is accomplished using some "assembly magic" which is not supported at the user level. Instead, the macro `current` is replaced with a variable that is a pointer to the current task.

A second function of the process management code is scheduling. Scheduling is not a feature explicitly supported by Lunar, but, rather, will be taken care of by the Weaves scheduler. The one exception is when a process blocks waiting for data. In this event, the kernel function `sched_timeout()` is called, which in turn calls schedule`()`, the

kernel's scheduling function which causes the process to yield the CPU[4]. In Lunar, `schedule()` calls the Weaves' scheduler to signify that it is ready to be scheduled out. In this way, Lunar and the Weaves' scheduler utilize both cooperative and preemptive scheduling. The section on Experimental Setup describes the current, non-Weaves solution that is being used for testing.

3.2.4 Interrupts

On a host machine, the hardware can signal the operating system of an event, such as data being ready on an input device, via a hardware interrupt. The operating system registers a handler function that is called to service the interrupt when it is received. Lunar supports two interrupts: the timer interrupt and a "data received" interrupt (equivalent to what would be generated by a network card). Although these interrupts are generated as software calls, rather than via the host machine's hardware, their use is the same. The interrupts serve as a method for the underlying simulator to signal Lunar of an event; put another way, Lunar's interrupt handlers form the API which the library exports to the simulation layer. Specific details on the two interrupts are provided in subsequent sections.

There is a second type of interrupt, called a *soft interrupt*, which is also used. A soft interrupt is raised as part of the processing of the interrupt handler (referred to as a *hard interrupt*). When the handler completes, `do_softirq()` is called to process any soft interrupts. The main difference between the two types of interrupts is that during a hard interrupt, additional interrupts are disabled. For performance reasons the time spent in this state is kept at a minimum, resulting in the use of soft interrupts (which can themselves be interrupted) for processing that doesn't require disabled interrupts.

Within Lunar, soft interrupts are raised in both interrupt handlers. During the timer interrupt, the soft interrupt is used to processes kernel timers. In the "data received" interrupt, the kernel's network receive function, `netif_rx()`, enqueues incoming buffers in a processor queue. The associated soft interrupt handles sending the packet up through the main portion of the network stack and placing it in the queue for the appropriate socket.

3.2.5 Time and Timers

Time is tracked by the Linux kernel using the timer interrupt, which is generated at a regular interval by the system's hardware clock (on an Intel machine, the default value is 10 ms). At boot time, the kernel installs an interrupt handler, `do_timer()`[5], that performs the necessary time-related processing for the kernel. One of the primary responsibilities of the interrupt handler is to update the variable `jiffies`. This variable is simply a counter of the number of timer interrupts that have occurred since the

[4] The Weaves' scheduler, in turn, uses the function `switch_to_task()` to enable to stack library to switch task.

[5] Technically, `do_timer_interrupt()` is the handler. However, `do_timer()` is the function that performs all the useful work. Therefore, for our purposes, we will consider `do_timer()` as the interrupt handler.

installation of the interrupt handler at system start. Since the interrupts occur at a known interval, elapsed time can then be calculated.

The next critical responsibility of `do_timer()` is to examine all the kernel timers to see if any have expired. Timers are added, removed and modified by kernel functions using `add_timer()`, `del_timer()`, and `mod_timer()`, respectively. When a timer is added, it is time stamped with the `jiffies` value at which it will expire. When that time is reached, the callback function registered with the timer is called.

It is also possible to get "wall clock" time from the Linux kernel. The wall clock functions are built on top of the Glibc time functions (`time()` and `gettimeofday()`) in the similar manner as that of the memory management functions.

Lunar implements the time and timer functionality using the kernel code itself, with only minor modifications.

3.2.6 Network Devices

The Linux kernel interacts with the underlying network via network interface devices (e.g. Ethernet or Token Ring cards). The kernel internally represents each of these hardware devices as a `struct net_device`. This data structure functions as a "virtual device" that provides an interface between Lunar and the simulator.

The current implementation of Lunar maintains a device chain that consists of a standard loopback device, as well as three "Ethernet" devices which interact with the simulator[6]. These devices are configured at Lunar initialization based on the desired configuration; it is not necessary that every device be used in any given scenario.

The Ethernet devices are, in fact, based on the loopback device. The two critical functions, `eth_init()` and `eth_xmit()`, are simply copies of the equivalent loopback functions, with `eth_xmit()` calling the simulator instead of the kernel's receive routine.

3.2.7 Low-Level Calls

While not specifically a sub-system of the kernel, low-level calls compose the final group of kernel functions that must be implemented by Lunar. Low-level calls are code – often platform-specific and written in assembly – that perform operations related to hardware. In some case, these calls are unable to be performed by user-level programs. An example of this are the macros `cli()` and `sti()`, which are used to disable and enable hardware interrupts, respectively. These calls are made to protect sections of code which, for reasons of consistency, can not be interrupted. Because of the dire consequences that can result from improper usage, the processor does not allow programs running in user mode to execute these commands.

[6] The number of devices is easily increased; future versions may dynamically create devices rather than relying upon a static list.

In order to provide the same functionality as `cli()` and `sti()`, a mutex, `cli_sti_mutex`, is created that protects the library. When `cli()` is called, the mutex is grabbed, and it is released upon a call to `sti()`. The interrupt handler checks this mutex when it is called, and only proceeds if it is not held. If the interrupt handler finds the mutex held, it sets a flag variable (there is one for each interrupt) to 1 (true). The call to `sti()` checks these variables after releasing the mutex, and calls the appropriate interrupt handler if necessary. This is identical to the manner in which the actual hardware behaves; when interrupts are re-enabled after a call to `cli()`, interrupts that occurred during the period will be trapped, but only the specific interrupt, not the total number.

A second set of macros that are related to `cli()/sti()` are `local_irq_save()` and `local_irq_restore()`. The macro `local_irq_save()` first saves the local CPU's flags into a user-supplied variable and then calls `cli()` to disable interrupts. The semantics of `local_irq_restore()` are slightly more subtle. It restores the stored CPU's flags, but does not explicitly re-enable interrupts. The reason is that the interrupt flag is one of the flags that is restored, which will re-enable interrupts **only if** they were enabled prior to the call to `local_irq_save()`. This pair is used in place of `cli()/sti()` in code that may be called with interrupts either enabled or disabled and prevents the re-enabling of interrupts prematurely [23]. In the case of Lunar, these macros simply consist of storing and restoring the value of the `cli_sti_mutex`.

3.3 Application API Implementation

The goal of Lunar's API is to provide a transparent interface to application programs. The objective to be able to run application code without the need for any modification within the source code. In the ideal case, the only additional work necessary is to recompile the application and link it against the library. In order to do this, Lunar must provide an API that is identical to that of the operating system's system calls (specifically, those provided by the Glibc library). It is not necessary that the library provides all the functionality of Glibc, just those functions necessary for network communications. Specifically, it must provide its own implementation of the BSD socket interface, as well as any related functions that can operate upon sockets. Appendix A provides a complete list of all the functions that are supported by Lunar's API.

Understanding how the Lunar API is implemented first requires a brief discussion of how system calls are handled in Linux. The user-level portion of a system call is handled by the Glibc libraries. A user application invokes a system call through a function (e.g. `read()` or `write()`), the definition of which is part of Glibc. The Glibc routines format the function parameters for use by the operating system. This involves storing the system call number in the *eax* register, and the parameters (or pointers to them) in the remaining registers. Finally, the system call software interrupt (0x80) is raised and control is passed to the operating system.

Inside the operating system, there is a function associated with every system call that serves as the handler for that call. The naming convention is that for a system call `foo`,

the handler function is called `sys_foo()` e.g. `read()` is handled by `sys_read()`[7]. When the system call is complete, control returns to the application via Glibc.

The entry point into Lunar for each system call is its associated `sys_*` function. In order for an application to function correctly with Lunar, each of the C functions associated with a system call must link against the system handler functions in Lunar, not those that in Glibc (which would call the real operating system). This is accomplished through the use of wrapper functions that are defined locally, and therefore will link prior to the point at which the Glibc libraries are searched to resolve any remaining unresolved symbols.

The wrapper functions (which are defined in the file stack_socket.c) simply call the necessary Lunar handler function to process the associated system call. For example, the wrapper for the function `socket()`:

```
int socket(int family, int type, int protocol)
{
     int res = sys_socket(family, type, protocol);

     if (res < 0) {
          errno = -res;
          return -1;
     }
     return res;
}
```

The additional processing of the return value is due to the way that errors are returned from the Linux kernel. The return value of a system call on error is the negative of the appropriate error number (defined in /usr/include/asm/errno.h). This additional processing is normally taken care of by Glibc, and consists of returning a -1 and setting the global variable `errno` to the error number. This keeps the return values from Lunar in compliance with the Glibc function specifications.

There is, however, an inherent problem with this arrangement: what if we actually **want** to make a system call to the operating system? This solution does not provide us with any alternative; we use either Lunar or the operating system. Consider the function call `read(fd, buf, len)`, where the parameter `fd` is the file descriptor from which to read. Since Linux treats sockets as files, `fd` may in fact be a socket file descriptor, requiring Lunar to handle the call and consequently, for the function `read()` to be

[7] The BSD socket functions are actually handled as a single system call, `socketcall`, with an additional parameter specifying the particular socket function. The handler `sys_socketcall()` simply calls appropriate function (i.e. `sys_socket()`, `sys_bind()`, etc) based on this parameter. For the purposes of Lunar, this intermediate multiplexing/demultiplexing is ignored, and each of the socket functions is treated as a separate system call.

supported by the API. However, if the file descriptor represents a physical file on disk (such as a configuration file), the operating system must handle the call. In this case (and others which will be discussed later), there are situations where system calls that are supported by Lunar must be bypassed in order to be serviced the host machine's operating system.

The solution to this problem lies in the fact that Glibc is provided as both a static and shared library. At Lunar initialization, `dlopen()`and `dlsym()` are used to acquire function pointers to the dynamic library versions of each of the API functions, which can then be used to access the Glibc versions when necessary.

Still, an additional problem remains. In the above example of the `read()` function, how is it possible to tell the two cases apart? There must be a way to determine which of the two functions to call, that of Lunar or of Glibc. A simple solution can be found by offsetting the file descriptors returned by Lunar by some large amount, such that it will be larger than any file descriptor returned by the operating system. The value of 1024 is used because it is the largest number of open file descriptors that Linux allows by default.

This scheme requires the addition of three functions:

```
#define SIM_MAX_FD 1024

static int inline to_sim_fd(int fd)
{
     return (fd += SIM_MAX_FD);
}

static int inline from_sim_fd(int fd)
{
     return (fd -= SIM_MAX_FD);
}

static int is_sim_fd(int fd)
{
     return (fd >= SIM_MAX_FD);
}
```

The first two functions are used to convert between the two types of file descriptors – the original and the offset – while the third is used to determine the type of file descriptor.

The wrapper function to create a socket now becomes:

```
int socket(int family, int type, int protocol)
{
     int res;

     res = sys_socket(family, type, protocol);

     if (res < 0) {
          errno = -res;
          return -1;
     }

     return (to_sim_fd(res));
}
```

Wrapper functions that take sockets (or, more generally, file descriptors) as a parameter look like:

```
ssize_t read(unsigned int fd, char * buf, size_t count)
{
     int res;

     if (!is_sim_fd(fd)) {
          res =  (*glibc_read)(fd, buf, count);
          return res;
     }

     res = sys_read(from_sim_fd(fd), buf, count);

     if (res < 0) {
          errno = -res;
          return -1;
     }

     return (res);
}
```

The first thing that must be done is to determine if the file descriptor is for a Lunar socket or not. If it is not, the Glibc version is called (note that this is a function pointer, hence the unusual syntax of the call) to handle it. If the file descriptor is for a Lunar-generated socket, the library's system call handler is called.

Note that Lunar initially returns a file descriptor in the "normal" range. It is only when the `socket()` wrapper function returns the descriptor to the application that the offset is applied. Similarly, when the file descriptor is sent to Lunar's system call handler, the offset is removed. In this way, only the application sees the modified descriptor. This

method isolates all the code necessary to provide an offset descriptor from the actual kernel code, simplifying the implementation. Additionally, it eliminates the problem of the kernel code rejecting a file descriptor for being outside of the acceptable range.

3.4 Simulator API Implementation

The API between Lunar and the underlying simulator consists of three functions:

- `int write_msg(int id, void *data, int len)`
- `int read_msg(int id, void *data, int len)`
- `void timer()`

The simulator exports `write_msg()` to Lunar, while the Lunar in turn exports `read_msg()`, and `timer()` (Figure 3.2). The functions `read_msg()` and `timer()` represent Lunar's interrupt handlers.

The use of `write_msg()` and `read_msg()` are straightforward. Lunar calls `write_msg()` when it has data to transmit over the simulator. The parameter `id` indicates the specific link on which to transmit (since a single host may have more than one link), `data` is a pointer to the data to be transmitted, and `len` is its length.

The function `read_msg()` operates in a similar manner, and is called by the simulator when it has data available for Lunar. It performs the functions of a network device driver, taking the raw message, inserting it into a kernel socket buffer, and finally passing it to the kernel's receive function, `netif_rx()`.

The third function, `timer()`, represents Lunar's timer interrupt handler. The simulator invokes the function `timer()` every 10 ms of simulated time, which in turn calls `do_timer()`, the kernel's timer interrupt handler.

At the completion of both `read_msg()` and `timer()`, the function `do_softirq()` is called to process the soft interrupts raised.

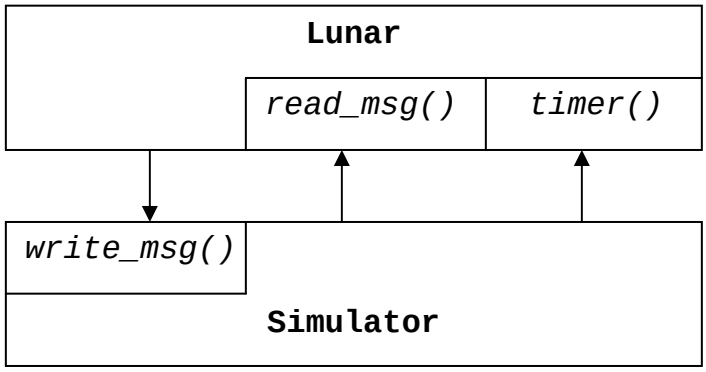

Figure 3.2: Lunar/Simulator Interface Functions

3.5 Lunar Initialization

Lunar is initialized at startup with a call to the function `stack_init()`. This function call is the only additional code that must be added to the application in order for it to function correctly with the library. With the integration of Weaves, this will no longer be required, as it will be called as part of the Weaves initialization procedure. Until then, the simple addition of the function invocation at the beginning of the application's `main()` function is all that is needed.

The processing performed by `stack_init()` is essentially that which occurs at system boot. Specifically, the initialization functions for the included portions of the kernel are called. Additionally, the function pointers for the Glibc versions of Lunar functions are initialized. Finally, the Lunar's networking devices, routing table, and (if required) firewall are configured.

3.5.1 Configuration of Lunar

The Linux TCP/IP stack is configured via the `ioctl()` system call by several programs that are distributed as part of the *net-tools* [24] package. In a typical system, these programs are invoked by the operating system's configuration scripts at boot time and include:

- ifconfig – configures the network interface devices
- route – configures the stack's routing table
- iptables – configures the firewall rules (distributed in a separate package[25])[8]

In order to allow for the configuration of Lunar, the code for these programs is linked in as part of the library itself. However, since these are standalone programs, they each contain a `main()` function which must be renamed. In each case, the function is simply given the name of the original program i.e. `ifconfig()`, `route()`, or `iptables()`.

The specific ifconfig, route, and iptables commands are passed to Lunar in a configuration file that is read at Lunar initialization. Each command is listed as it would be entered on the command line or in a startup script (Figure 3.3). Lunar's initialization routines parse each command and create an argv-type array with them. When the command has been parsed, the appropriate function is called with the number of parameters and the array; from the standpoint of the function, it as if it is called from the operating system with argc and argv. The net-tools/iptables functions then process each command and call the required functions directly.

[8] Support for the firewall is a configurable option, with the default set to off

```
ifconfig lo 127.0.0.1 netmask 255.0.0.0
ifconfig eth0 192.168.200.1 netmask 255.255.0.0
ifconfig eth0 mtu 1500
ifconfig eth0 -arp

route add -host 192.168.200.2 dev eth0

iptables –-policy INPUT ACCEPT
```

Figure 3.3 Sample Lunar Configuration File

Of note is the command:

```
ifconfig eth0 -arp
```

This specifies that the device is not to use the Address Resolution Protocol (ARP) in order to resolve MAC addresses. This is done because MAC addresses are not supported in the virtual devices of Lunar; consequently, any ARP request would go unanswered. All the packets sent by Lunar have the required space for the MAC information allocated, but this is simply done as padding, with no useful data stored.

The benefit of this approach to Lunar configuration is two-fold. First, in keeping with the ideology of the rest of this project, it utilizes existing real-world code that is known to work, rather than attempting to write new code from scratch. Second, it provides a well know interface to the user, so there is no need to learn a new syntax for configuration commands. The configuration files themselves can be easily generated automatically using a scripting language.

3.5.2 Additional Configuration Using *sysctl*

It is also possible to perform additional configuration on Lunar using *sysctl*, a feature of the Linux kernel that allows for runtime configuration of specified kernel parameters. There are two separates methods of utilizing this feature. The *sysctl* program operates on the kernel parameters via the /proc/sys subdirectory. However, the program requires that the /proc filesystem be supported in the kernel; Lunar does not provide this support.

The method used by Lunar is the C function, `_sysctl()`, which allows for the reading and writing of the same parameters as are found in /proc/sys. Specifically, the library supports those parameters that are found in the /proc/sys/net/core and /proc/sys/net/ipv4 directories. The downside is that this approach does not lend itself to using a configuration file; all `_sysctl()` commands must be hard-coded in Lunar's initialization routines. The upside is that in most situations, *sysctl* is not needed; it is simply provided as a "nice to have" tool for fine tuning the library.

3.6 Chapter Summary

This chapter has described the implementation of Lunar, a user-level library built from the Linux kernel. The implementation methodology used to construct the library was described. The API functions between Lunar and the application and simulation layers were defined. Finally, the tools used for the runtime configuration of Lunar were examined. The next chapter details verification and performance testing performed on Lunar.

Chapter 4. Testing and Evaluation

The first portion of this chapter is devoted to describing the experimental architecture used as part of the testing environment. This consists of an additional layer of code added to Lunar in order to provide the functionality of the yet-as-incomplete, simulator. The second half describes the various functional and performance tests used, and their associated results.

4.1 Experimental Setup

Since Weaves/ONE is currently in the development phase, an alternative setup is necessary for Lunar testing. In particular, a replacement is needed for the underlying simulator and interface functions that it exposes to Lunar.

The solution that was developed was to associate each network device (*eth0*, *eth1*, etc) with a real UDP socket (by "real", we mean that it is a socket on the host machine, not Lunar). These sockets are linked to form a set of point-to-point connections that represent the simulated network (Figure 4.1). Note that this is another situation where it is necessary to still have access to the operating system's system calls that would otherwise be provided by Lunar.

Each UDP socket is configured via an entry in the configuration file, for example:

```
eth0 8192 8192 192.168.200.2
```

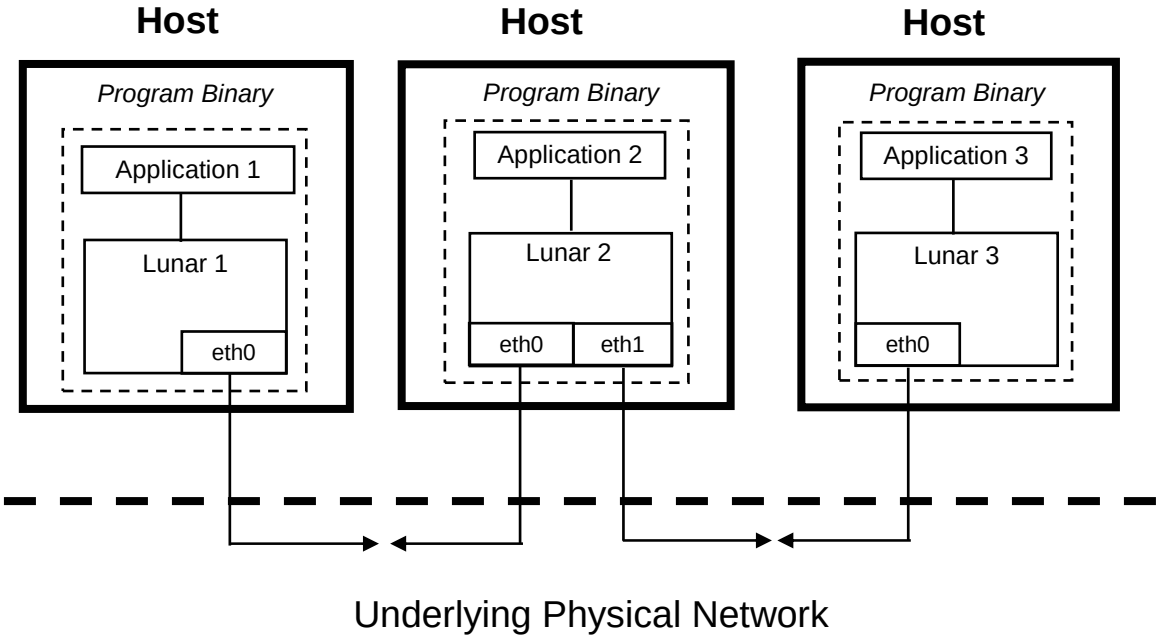

Figure 4.1: Example Experimental Architecture

The first parameter, *eth0*, indicates which network device inside Lunar with which this socket is associated. The next parameter specifies what port the UDP socket should bind to – in this case, 8192. The next two parameters specify the other endpoint of the socket "connection" – namely that the socket should transmit its data to port 8192 on host 192.168.200.2. The related configuration file entry on second host would be:

```
eth0 8192 8192 192.168.200.1
```

There are a few items of note here. First, this scheme is not the typical client-sever model often seen in UDP, but rather peer-to-peer. In the standard model, one socket functions as a server by binding to a specific port which is known to the client. The client socket does not bind to a specified port, but rather has a free port assigned to it by the host machine's stack when it begins transmitting data. When the server reads the data send from the client, it also receives the "return address" of the datagram – the client's IP address and port number – which it can use to send data back to the client. In this model, both sockets act as servers in that they each bind to a specified port of which the other end is aware. The benefit of this is that either can send data first; there is no need to wait for the server to receive the client's IP address and port.

The second item of note is that in this example, the port numbers are both the same. This is perfectly acceptable since each port is on a separate host machine. It would be just as acceptable to assign a different port to each socket. Using the same port for each point-to-point connection, however, is much simpler and easier to manage. There is a one-to-one relationship between links and port numbers, and it eliminates the problem of accidentally reversing the local and remote port in the configuration files (which would prevent communication between the sockets).

There is a scenario, however, in which having different local and remote port numbers is required. It is possible to have both applications, each with their own copy of Lunar, running on the same machine (note that without Weaves, a copy of Lunar for each application is a necessity). It is for this reason that two entries are provided in the configuration file, rather than just one. In such a case, the two configuration files would look like:

```
eth0 8192 8193 127.0.0.1

eth0 8193 8192 127.0.0.1
```

4.1.1 Simulator API

The experimental setup must support the same API functions that are used between Lunar and a simulator: `write_msg()`, `read_msg()` and `timer()`. Of the three, the `write_msg()` is the simplest, requiring only that the data to be transmitted over the appropriate UDP socket using `sendto()`. This implementation of `write_msg()` (as well as `read_msg()`, described below)uses the UDP socket's file descriptor as the link ID.

The two remaining functions, `read_msg()` and `timer()`, are implemented using signals. This makes sense, since these functions are in essence Lunar's interrupt handlers, and signals function like software generated interrupts.

For `read_msg()`, asynchronous notification is set for each UDP socket using `fcntl()` and providing the O_ASYNC flag. The result is that every time data is received on the socket, the operating system will generate a SIGIO, which is then caught by Lunar. The function registered as the signal handler is not, however, `read_msg()`, but rather the wrapper function `recv_msg()`. The reason is that `read_msg()` takes as its parameters the data, its length, and the link ID, while the signal simply indicates data is available. It is up to signal handler `recv_msg()` to read the data from the socket and pass it to `read_msg()` in the correct format.

Since each network device is represented by a UDP socket, and most hosts will have more than one device, `recv_msg()` must first determine which socket (or sockets) has data (the signal only indicates that data is available on some socket, not which one). To do this `recv_msg()` uses `select()`, which takes a list of file descriptors (in this case, for those of sockets) and indicates which, if any, have data on them to be read. For each socket that has data, `recvfrom()` can then be called, and the data then passed up the stack by normal means.

A pseudocode representation of `recv_msg()` looks like:

```
if cli_sti_mutex held {
     int_pending_recv_mesg = 1;
     return;
}

while (1) {
     if select(fdlist) has data {
          for each fd in fdlist with data {
               recvfrom(fd, data);
               read_msg(fd, data, len);
          }
     } else {
          break;
     }
}
```

The loop will continue to call `select()` until no data remains. This will handle the case where there are multiple datagrams to be read on a single socket.

The timer mechanism is implemented using Linux's *itimer* functions, which generate a SIGALRM at a specified interval (in this case, 10 ms). The function `timer()` is registered as its respective the signal handler (since `timer()` takes no parameters, no wrapper function is needed).

The pseudocode representation of `timer()` is similar to that of `recv_msg()`:

```
if cli_sti_mutex held {
     int_pending_timer = 1;
     return;
}

do_timer();
```

Note that both functions only test the `cli_sti_mutex`, but do not attempt to grab it. Normally, interrupt handlers (or at least large portions of them) are non-reentrant, and must not themselves be interrupted. What prevents another signal, either a SIGIO or SIGALRM, from interrupting these functions? The answer is that the Linux signal mechanism blocks the current signal (as well as any other signals specified by the programmer) while inside a signal handler. Upon exit from the handler, the blocked signals will be delivered. If the signal is blocked because the `cli_sti_mutex` is held, when the mutex is released by `sti()`, the two interrupt pending flags will be checked and the appropriate signal(s) will be raised manually (using `raise()`).

4.1.2 Scheduling

The final area of discussion with regards to the experimental setup relates to scheduling. As described earlier, scheduling is handled by the Weaves' Scheduler, which is usually transparent to the process. The one exception is when Lunar must wait for an event – usually the arrival of a data packet – and explicitly calls `schedule()` to yield the CPU (which in turns call the Weaves' scheduling function). In the non-Weaves environment, `schedule()` simply calls `nanosleep()`, which puts the process to sleep until either the prescribed time expires, or a signal is received. Since the event that the process is waiting to occur will be associated with a signal, it will immediately be awakened to check if the condition that originally caused it to sleep has been satisfied. If not, `schedule()` will be called again and the process put back to sleep. The timeout used for the call to `nanosleep()` is 10 ms, the time between timer interrupts, guaranteeing that the process will sleep until an interrupt occurs.

4.2 Testing and Results

Testing of Lunar consisted of two types: verification and performance. The first series of tests were designed to demonstrate the correct functioning of the library with regards to the various socket functions, as well as to routing. The second series were performance-related, where performance is measured in terms of the additional overhead that is incurred by the addition of the library to an application.

Testing was performed on a cluster of PCs running Linux kernel version 2.4.9-21. Each machine has a 1 GHz Athlon processor, 1 GB of RAM, and is connected by both 100 Mbps switched Ethernet and 1Gbps Myrinet. Unless otherwise specified, all testing was performed over the Ethernet interface.

4.2.1 Verification Testing

The primary test consisted of a simple client/server file transfer using a 1 MB file. The output is then compared (using `diff`) to verify that it matches the input file. Several different scenarios were tested:

- TCP using `send()`/`recv()`
- TCP using `write()`/`read()`
- TCP using `writev()`/`readv()`
- TCP using `select()`
- TCP over an unreliable link (5% packet loss)
- UDP using `sendto()`/`recvfrom()`

In all cases, Lunar correctly transferred the data.

A second test was done to verify that routing was being performed correctly. The original experiment was expanded to include a second client/server pair transferring a different 1 MB file using TCP. Rather than each client and server being directly linked, all four were connected to an additional machine, which functioned as a router. The resulting star topology is shown in Figure 4.2. Each client and server resides on its own network (each link has a netmask of 255.255.255.0). Additionally note that the link between the router and the second server consists of two different networks (192.168.2.0 and 192.168.3.0), indicating one or more intermediate routers that are not represented. The purpose of this is to verify that the library is able to correctly route to a network address that is not directly connected, but one for which a static route exists[9]. The library was able to correctly route both connections.

4.2.2 Performance Testing

The critical performance issue involving the library is the amount of overhead that it adds to an application. The overhead is measured in terms of time; specifically, the percent increase in time for an application running with Lunar as compared to the application

[9] All the static routes provided to the router are done so as network, rather than host, routes. For example: `route add –net 192.168.3.0 netmask 255.255.255.0 dev eth3`

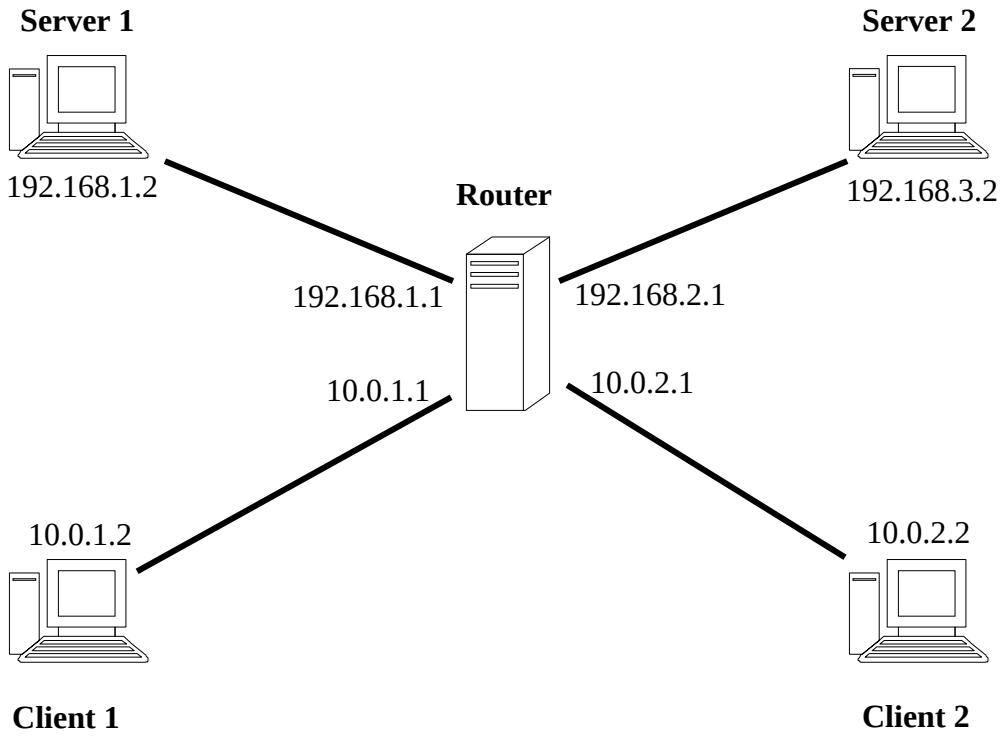

Figure 4.2: Topology for Routing Test

running without it. We examine the effect of the library on both the total running time of an application as well as latency.

4.2.2.1 Runtime Overhead

The first of the performance tests measure the effect of the library on the total running time of an application. Again, the test case used was a simple client/server file transfer using TCP. The advantage of a file transfer application is that it is almost an entirely communication-oriented operation; virtually no time is spent in computation outside of the stack. To further this, the "file" that was used was not a physical file, which would require time to read from disk, but rather an empty array that was repeatedly transmitted until the desired file size was reached. Three files sizes – 10, 20 and 30 MB – were used to test how an increased load would affect Lunar. For each file size, complete runs were made with an ever increasing transmission size (the size of the data sent in each `write()` call), from 1 to 55 KB i.e. the first run transfers the file in 1 KB chunks, the second in 2 KB chunks, etc. The purpose of this was to see what, if any, effect fragmentation had on performance. For each file size/transmission size pair, five runs were made and the average was taken.

In order to prevent unnecessary fragmentation at the UDP level, the MTU of Lunar's Ethernet devices was reduced from 1500 to 1458 bytes. This 42 byte difference represents the TCP, IP, and MAC headers that the library adds to each data packet. If the MTU of the virtual devices are set to 1500, the total amount of data sent over the UDP

socket will be 1542. Since this is greater than the MTU of the underlying Ethernet, the message will be fragmented. While reducing the MTU does cause a small decrease in performance, this is still preferable to an increased number of packets that would need to be transmitted.

The complete test suite (10, 20, and 30 MB) was run a second time using the 1 Gbps Myrinet[10] as the underlying physical/link layer for an additional performance measurement. In order to correctly emulate a Myrinet device, Lunar's virtual network devices were set with the Myrinet MTU value of 9000 (actually, 8958 to compensate for the header).

The goal of these tests is not to accurately *quantify* the amount of overhead that Lunar will add to a running simulation; the values here can not be directly applied in order to calculate the increase in simulation runtimes with the addition of the library. Rather, the goal is to verify that, with regards to a general application, the library adds only a marginal increase in runtime. If, for example, testing revealed that Lunar doubled or even tripled the running time of the test application, that would raise serious concern as to the effect it would have on simulation times. Such results would warrant much more extensive – and specific – testing. However, if the overhead is low, this implies that the library will have a manageable impact on the simulator. Of course, this assumption may still prove false when the two are eventually integrated, but the test does provide confidence enough to move on to the next step.

Runtimes for the unmodified code versus the library code for each of the file sizes are in Figures 4.5 – 4.7, with the percent overhead for each size in Figure 4.3 (Ethernet) and Figure 4.4 (Myrinet). For Ethernet, the overhead that is added by the library is both minimal as well as reasonably consistent over varying file and buffer sizes. The overhead is independent of buffer size and appears to be only minimally affected by the amount of data transmitted, with the value converging towards a constant percentage as the data size increases.

Over Myrinet, the percent overhead is higher than over Ethernet. This can be accounted for by the increased transmission rate of Myrinet, which in turn results in computation (as opposed to communication) accounting for a greater percentage of the total running time. Over Ethernet, the additional computation produces a smaller footprint since much more time is spent in packet transmission, which impacts both library and unmodified code equally.

[10] 1 Gbps is the *effective* rate; Myrinet is capable for delivering data rates of 1.28 Gbps (half duplex) or 2.56 (full duplex). This discrepancy is due to the fact that the 64-bit/66 MHZ Myrinet card is operating in a 32-bit/33 MHZ slot.

Additionally, the overhead is considerably more variable over Myrinet than Ethernet, though still within only a few percent over most of the buffer size range. This variability can be explained in terms of the shorter total running times that are achieved with Myrinet, which causes small differences in timing to have a greater effect; over a longer time, these variations are smoothed out. Even in light of the increased overhead and variance over Myrinet, Lunar still incurs an acceptable performance penalty.

4.2.2.2 Latency Overhead

The final test measures the increase in latency that is added by Lunar. The test consists of sending a single 4-byte message from a client to a server, which then echoes the message back to the client. The client measures the time required to send the message and receive the reply. This procedure, done both with and without the library, was repeated 100 times and the average taken. Like the previous tests, the percent overhead is then calculated.

The average time for the application without the library was 100.3 μs. The inclusion of Lunar raised that value to 157.64 μs, yielding an overhead of 57.2%. As noted in the previous section, this increase is due to the fact that with the latency test, a large percentage of the measured time represents computation rather than communication costs, and falls within acceptable values. Additionally, Myrinet natively uses a zero-copy implementation. Since Lunar performs a data copy, this adds to the latency.

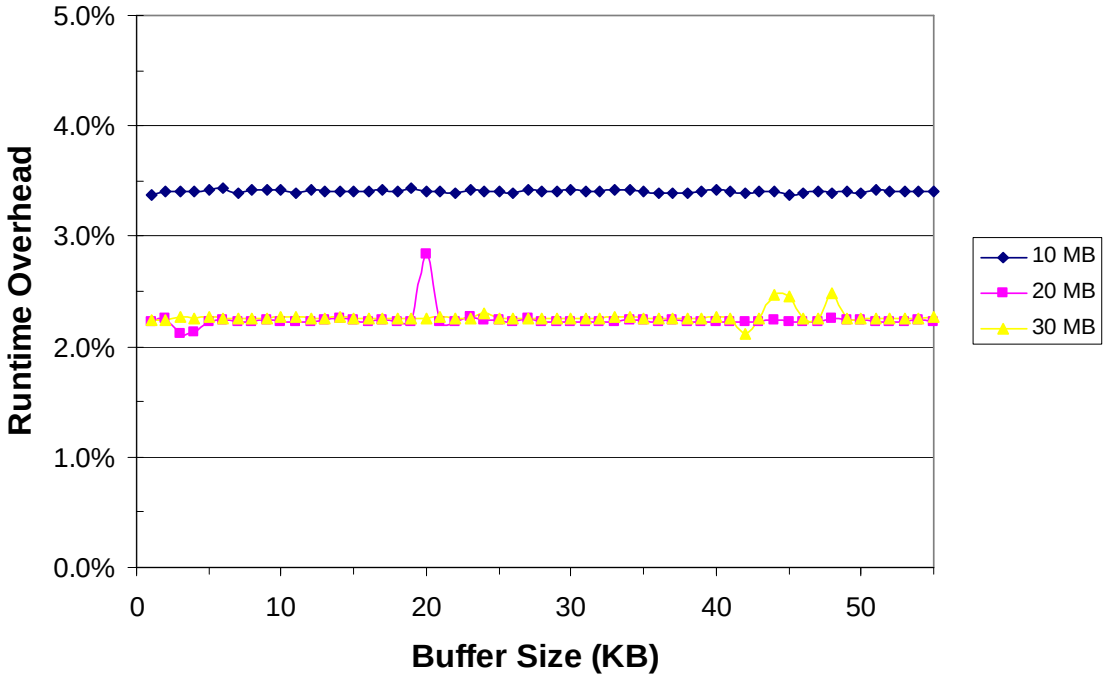

Figure 4.3: Percent Overhead of Lunar vs. Unmodified Code (Ethernet)

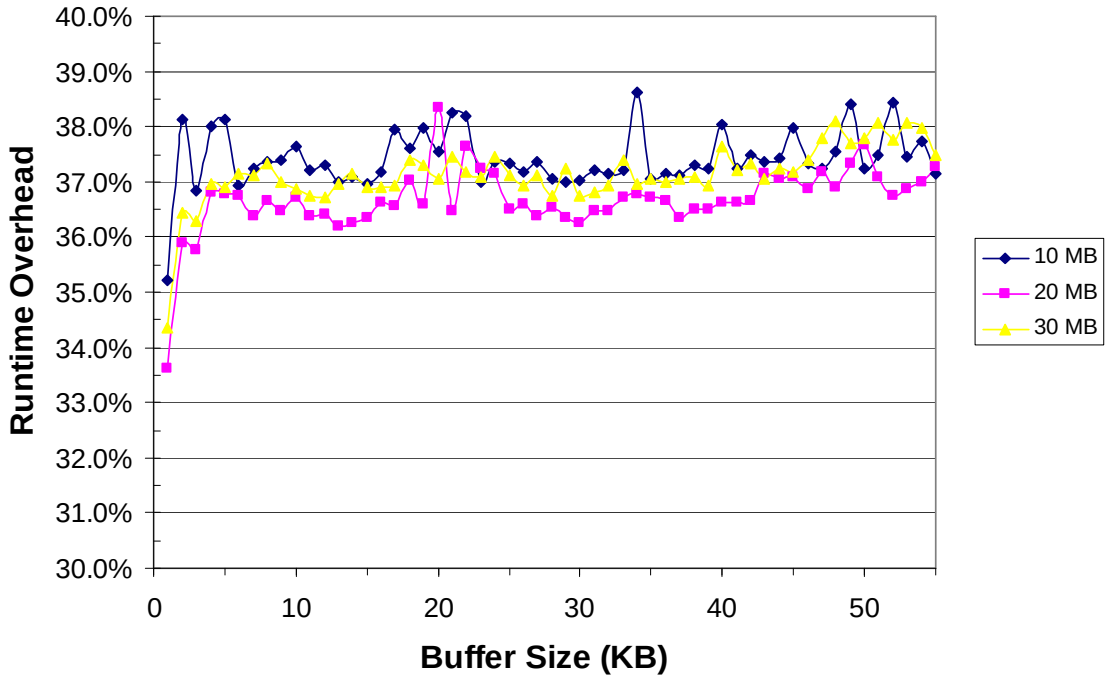

Figure 4.4: Percent Overhead of Lunar vs. Unmodified Code (Myrinet)

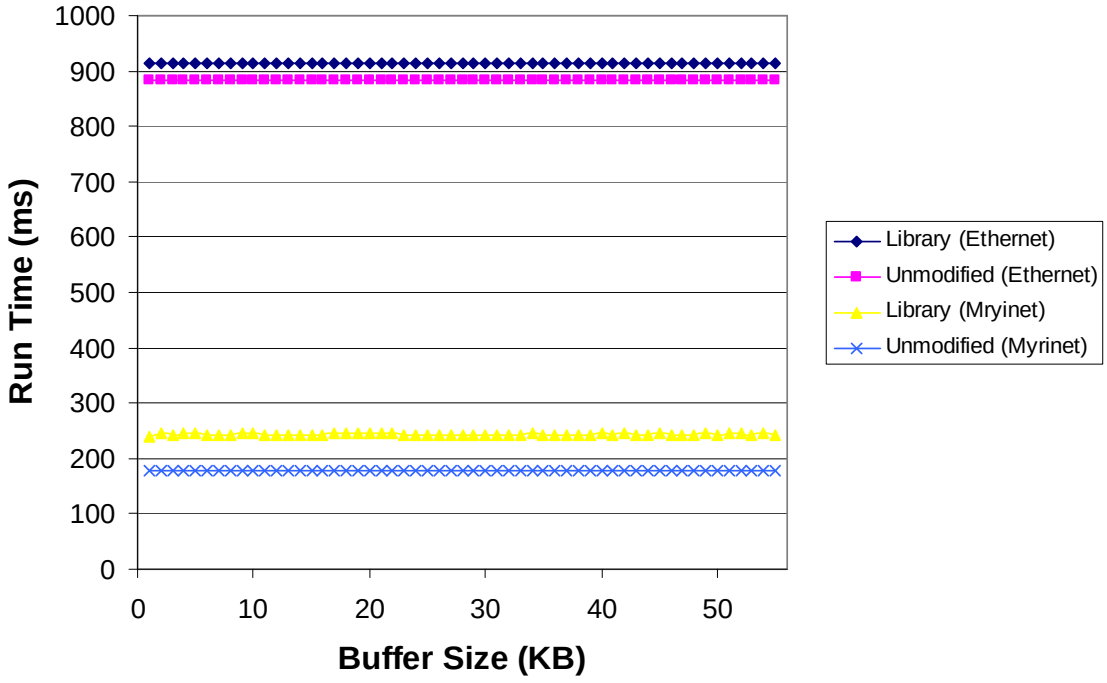

Figure 4.5: Runtime Of Lunar vs. Unmodified Code With 10 MB File

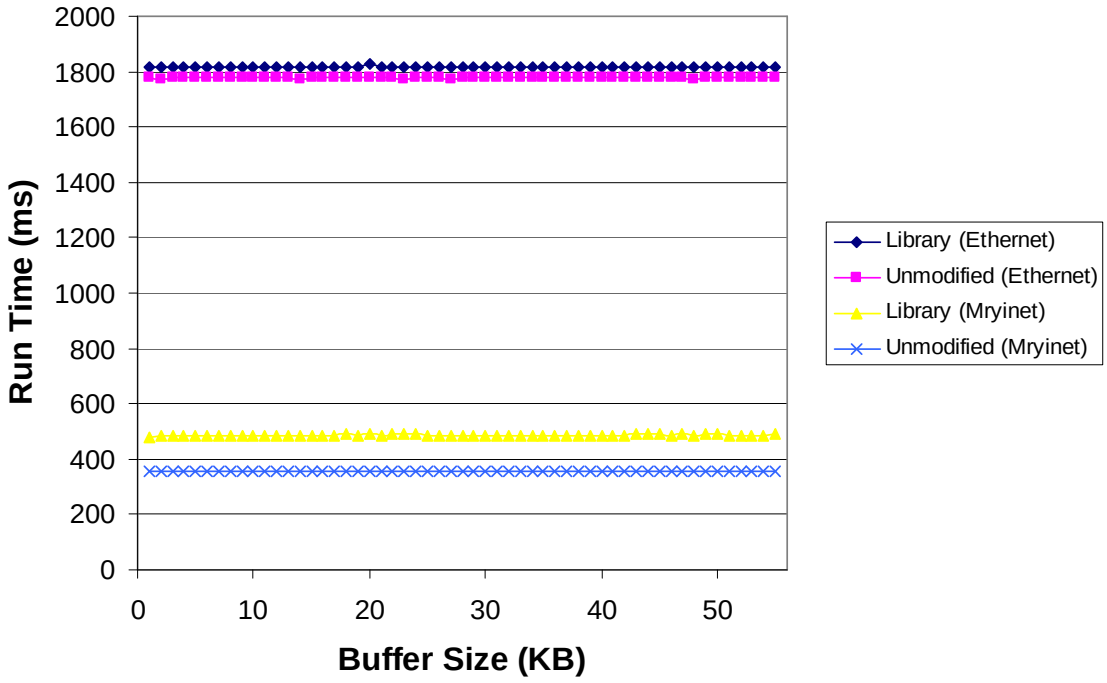

Figure 4.6: Runtime Of Lunar vs. Unmodified Code With 20 MB File

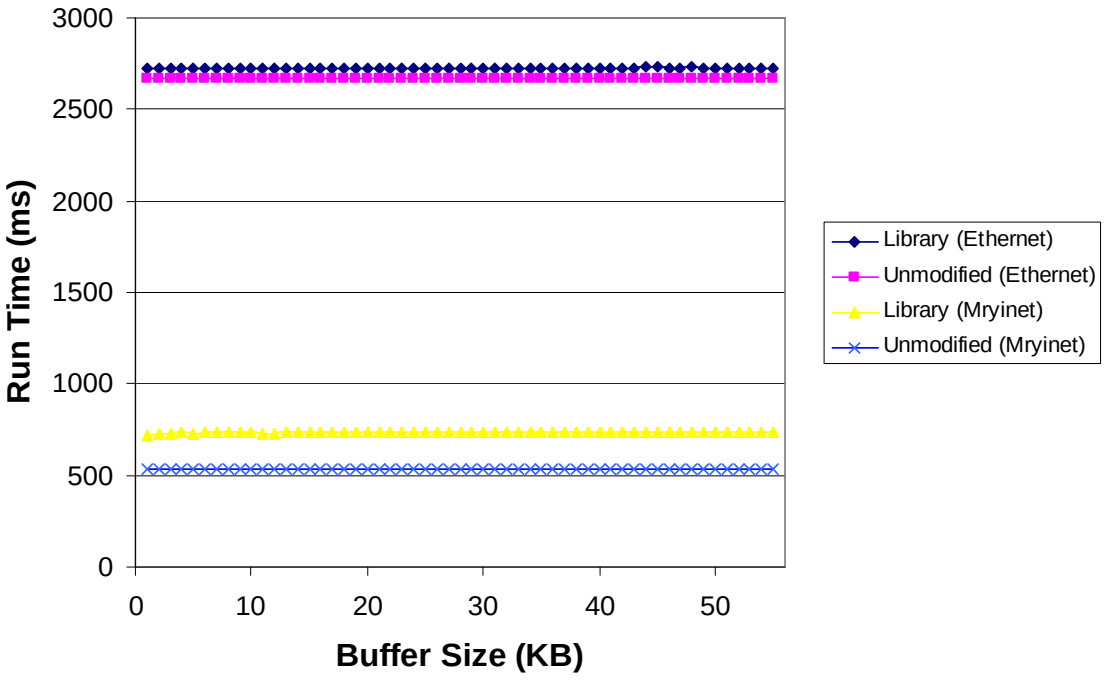

Figure 4.7: Runtime Of Lunar vs. Unmodified Code With 30 MB File

4.3 Chapter Summary

This chapter has described the testing performed on Lunar. Since the simulator for the ONE has not yet been built, an alternative architecture for testing was devised in which links were constructed using UDP sockets. Verification testing was performed to demonstrate correct functioning of the library. Performance testing was done to measure the overhead that Lunar adds to an application and was within acceptable limits

Chapter 5. Conclusions and Future Work

This chapter provides a brief summary of the work described in this thesis, as well as a description of future work with Lunar.

5.1 Summary

This thesis has presented a user-level library built from the network stack of the Linux operating system. The library serves as a key component in the Open Network Emulator, a large-scale network emulation testbed designed to allow the testing and development of networking protocols. It allows for networking applications to execute on top of a simulator architecture, combining the benefits of direct code execution (correctness of behavior and ease of deployment) with the control and scalability afforded by a parallel network simulator.

The library transparently integrates with applications by providing a system call interface identical to that provided by Glibc (e.g. BSD sockets). This interface redirects calls for networking services to the library, rather than to the host's operating system. The library interacts with the underlying simulator using virtual, rather than physical, network devices, where each device is associated with a link in the simulator's virtual topology. Messages are transmitted and received by the library using a simple set of interface functions between it and the simulator. Additionally, the simulator also provides the library with its notion of time – in this case, virtual time – by calling the library's timer function every 10 ms.

Correctness and performance testing show that the library functions correctly with an acceptable amount of overhead.

This thesis provides the following contributions:

1) A functioning library which can be used as part of the Open Network Emulator for protocol testing and development
2) A formal catalog of changes made to the kernel source code, enabling future kernel versions to be converted with considerably less work
3) An API for the inclusion of libraries built from additional operating systems

5.2 Future Work

The obvious next step is to fully integrate the library with Weaves, and finally with the parallel simulator. As part of the development of Weaves, preliminary testing with an older version of Lunar was performed successfully [14]. In it, two separate stacks were instantiated, each with a client and server application that transferred a 1 MB file between them. The applications were connected via each stack's loopback device; there was no inter-stack communication. Though this test demonstrated basic functionality, additional work still needs to be done. In this scenario, for example, data was not being received via interrupts, since the loopback device bypasses the lower layers and calls the kernel's receive function directly. Additionally, more thorough testing is required of the Weaves'

Scheduler and its performance when there are a large numbers of applications, many of which may need to block while awaiting data.

The final step in the construction of the ONE will be the integration of Lunar with the underlying parallel discrete event simulator. Currently, the design of the simulator has not been finalized, and consequently has not yet been built. From the standpoint of Lunar, however, the design details are irrelevant; the only requirement is that the API described in Section 3.4 is used. This means that it is possible to begin testing with any simulator design, during which time the final simulator for the ONE is being constructed.

As for the library, there are a few additions and improvements that can be made. The first is to rebuild the library using a more recent version of the kernel. During development, attempting to keep up with the latest kernel version would have been a futile task. The primary concern was having a kernel from the most current family (in this case, 2.4.x), but not necessarily the most recent version[11].

A second addition to Lunar that needs to be added is signal handling. Currently, signaling of application programs is not supported. This means that asynchronous notification as well as correct handling urgent TCP data is not performed. Future work will involve incorporating the kernel's signal handling routines into the library.

A final addition would be the inclusion of the file stream functions (`fread()`, `fwrite()`, etc). Networking applications can use these functions to communicate over sockets by first converting the socket file descriptor to a stream using `fdopen()`. These functions are provided as part of Glibc, and their incorporation was considered as an option during the development of the current version of the library. However, that idea was abandoned due to the fact that it would require a large amount of time devoted to what is essentially a tangential project (interestingly, the Glibc source code is **much** more complicated than that of the kernel!). However, these functions are valuable to have as part of the library, and can be included either by porting the Glibc code or by writing a separate implementation.

In the long term, future work should include similar libraries being created for other operating systems since the use of a single operating system as part of the ONE is clearly not representative of real-world networks. The behavior resulting from the interactions of different operating systems is of great importance when testing protocols. It is the goal of this work to serve as the first piece in what will be a much larger system and to provide a template for future work.

[11] Work on the library was originally begun using a 2.2 kernel. Shortly thereafter, the 2.4 family became the standard and the library was rebuilt. This additional work took very little time, boding well for future portings of the library.

Chapter 6. References

[1] Sally Floyd and Vern Paxson, "Difficulties in Simulating the Internet," *IEEE/ACM Transactions on Networking*, 9(4): 392-403, 2001.

[2] Jeffery C. Mogul, "The Case for Persistent-Connection HTTP," *ACM SIGCOMM Computer Communications Review*, 25(4): 299-313, 1995.

[3] Van Jacobson, "Congestion Avoidance and Control," *ACM SIGCOMM Computer Communications Review*, 18(4): 314-329, 1988.

[4] S. Keshav, REAL 5.0 Homepage, http://www.cs.cornell.edu/skeshav/real/overview.html , 1997. Accessed January 8, 2004.

[5] The Network Simulator – ns-2 Homepage, http://www.isi.edu/nsnam/ns/index.html . Accessed January 8, 2004.

[6] OPNET Modler Hompage, http://www.mil3.com/products/modeler/home.html , OPNET Technologies Inc., Bethesda, MD. Acessed January 8, 2003.

[7] Lawence S. Brakmo and Larry L. Peterson, "Experiences with Network Simulation," *SIGMETRICS '96*, pp. 80-90, 1996.

[8] RFC 793, "Transmission Control Protocol," Information Sciences Institute, University of Southern California, Marina del Rey, CA, 1981. Accessed from http://www.faqs.org/rfcs/rfc793.html , January 16, 2004.

[9] Mark Carson and Darrin Santay, "NIST Net – A Linux-based Network Emulation Tool," *ACM SIGCOMM Computer Communications Review*, 33(3):111-126, 2003.

[10] Luigi Rizzo, "Dummynet: a simple approach to the evaluation of network protocols," *ACM SIGCOMM Computer Communication Review*, 27(1): 31-41, 1997.

[11] X.W. Haung, R. Sharma and S. Keshav, "The ENTRAPID Protocol Development Environment," in *Proceedings of IEEE INFOCOM '99*, pp 1107-1115, 1999.

[12] S. Keshav, "Ensim Entrapid Educational Release," Press Release from Ensim, Sunnyvale, CA, February 14, 1999. Accessed on January 12, 2004 from http://linuxtoday.com/developer/1999021401305NWSW .

[13] Srinidhi Varadarajan, "Weaving a Code Tapestry: A Framework for Reconfigurable Programming," DOE Early Career Proposal, 2000.

[14] Joy Mukherjee, "A Compiler Direct Framework for Parallel Compositional Systems," Unpublished Thesis, Virginia Tech, Blacksburg, VA. Available at http://scholar.lib.vt.edu/theses/available/etd-12312002-181024/unrestricted/Joy_Mukherjee_MSThesis.pdf .

[15] Scalable Simulation Framework Homepage, http://www.ssfnet.org/homePage.html . Accessed January 8, 2003.

[16] S. Y. Wang and H.T. Kung, "A Simple Methodology for Constructing an Extensible and High-Fidelity TCP/IP Network Simulator," in *Proceedings of IEEE INFOCOM '99*, pp. 1134-1143, 1999.

[17] Rob Simmonds, Russell Bradford and Brian Unger, "Applying Parallel Discrete Event Simulation to Network Emulation," in *Proceedings of the 14th Workshop on Parallel and Distributed Simulation*, 2000.

[18] Mark Allman, Adam Caldwell and Shawn Ostermann, "ONE: The Ohio Network Emulator," Ohio University, School of Electrical Engineering and Computer Science TR-19972, August 18, 1997.

[19] Amin Vahdat, Ken Yocum, Kevin Walsh, Priya Mahadevan, Dejan Kostić, Jeff Chase and David Becker, "Scalability and Accuracy in a Large-Scale Network Emulator," in *Proceedings of USENIX 5th Symposium on Operating System Design and Implementation*, pp. 271-284, 2002.

[20] David Ely, Stefan Savage and David Wetherall, "Alpine: A User-Level Infrastructure for Network Protocol Development," in *Proceedings of 3rd USENIX Symposium on Internet Technologies and Systems*, 2001. Retrieved from http://alpine.cs.washington.edu/alpineUsits01.pdf , Decmber 31, 2003.

[21] Scott Maxwell, "Linux Core Kernel Commentary," The Coreolis Group, Scottsdale, AZ, 1999.

[22] Jeff Bonwick, "The Slab Allocator: An Object-Caching Kernel Memory Allocator," USENIX Summer 1994 Technical Conference, 1994.

[23] The Linux-kernel mailing list FAQ, http://www.kernel.org/pub/linux/docs/lkml/ . Accessed January 11, 2004.

[24] Phillip Blundell, *net-tools* Software Package, Version 1.60, April 2001. Available at http://www.tazenda.demon.co.uk/phil/net-tools/net-tools-1.60.tar.bz2 .

[25] netfilter/iptables Hompage, *iptables* Software Package, Version 1.2.1a, March 2001. Available at http://www.iptables.org/files/iptables-1.2.1a.tar.bz2 .

Appendix A - Lunar API Functions

The following is a list of all the functions which are implemented as part of Lunar's interface.

BSD Socket Functions:

```
int socket(int family, int type, int protocol)
int bind(int fd, struct sockaddr *umyaddr, int addrlen)
int sendto(int fd, void * buff, size_t len, unsigned flags,
                struct sockaddr *addr, int addr_len)
long send(int fd, void *buff, size_t len, unsigned flags)
int recv(int fd, void * ubuf, size_t size, unsigned flags)
int recvfrom(int fd, void *buf, size_t len, int flags,
               struct sockaddr *from, socklen_t *fromlen)
int recvmsg(int fd, struct msghdr *msg, int flags)
long sendmsg(int fd, struct msghdr *msg, unsigned flags)
long socketpair(int family, int type, int protocol,
                  int usockvec[2])
long listen(int fd, int backlog)
long accept(int fd, struct sockaddr *upeer_sockaddr,
                 int *upeer_addrlen)
long connect(int fd, struct sockaddr *uservaddr,
           int addrlen)
long getsockname(int fd, struct sockaddr *usockaddr,
               int *usockaddr_len)
long getpeername(int fd, struct sockaddr *usockaddr,
               int *usockaddr_len)
long setsockopt(int fd, int level, int optname,
               char *optval, int optlen)
long getsockopt(int fd, int level, int optname,
              char *optval, int *optlen)
long shutdown(int fd, int how)
```

Non-socket Specific Functions:

```
ssize_t read(unsigned int fd, char * buf, size_t count)
ssize_t write(unsigned int fd, const char * buf,
               size_t count)
ssize_t readv(unsigned long fd,
               const struct iovec * vector,
               unsigned long count)
ssize_t writev(unsigned long fd,
               const struct iovec * vector,
               unsigned long count)
int close(int fd)
int ioctl(unsigned int fd, unsigned int cmd,
          unsigned long arg)
```

```
int fcntl(unsigned int fd, unsigned int cmd,
          unsigned long arg)
long select(int n, fd_set *inp, fd_set *outp, fd_set *exp,
               struct timeval *tvp)
int _sysctl(struct __sysctl_args *args)
```

Appendix B – Modifications To Kernel Files

The following tables present a list of all the changes made to the kernel files that are used as part of Lunar, not including the *_stub.c files. The first column contains the file name, the second contains the function or macro in which the change is made, or the global variable effected. The final two columns contain a description of the change made and its associated reason.

File	Function/Variable	Description	Reason
af_inet.c	init_inet	Removed static keyword from function declaration	Function is now called from stack_init.c
dcache.c	dcache_init	Replaced `for` loop that sets the variable `order` based on the number of pages of memory to a constant vale of 0.	Only sockets are used and there is no need for more than a page of memory for cache entries
fcntl.c	do_fcntl	Removed case F_DIRNOTIFY	Unused for sockets
fcntl.c	sys_fcntl64	Removed function	Unused
fcntl.c	send_sigio_to_task	Removed function	Unused
fcntl.c	send_sigio	Removed function body	Signals are not currently supported
inetpeer.c	inet_initpeers	Replaced si_meminfo() with sysinfo()	sysinfo() is the user-level version of si_meminfo()
iptables-standalone.c	iptables	Replaced exit() with return()	Function is no longer part of an independent program
namei.c	vfs_create	Removed function	Unused
namei.c	open_namei	Removed function	Unused
namei.c	lookup_create	Removed function	Unused
namei.c	vfs_mknod	Removed function	Unused
namei.c	sys_mknod	Removed function	Unused
namei.c	vfs_mkdir	Removed function	Unused
namei.c	sys_mkdir	Removed function	Unused
namei.c	vfs_rmdir	Removed function	Unused
namei.c	sys_rmdir	Removed function	Unused
namei.c	vfs_unlink	Removed function	Unused
namei.c	sys_unlink	Removed function	Unused
namei.c	vfs_symlink	Removed function	Unused
namei.c	sys_symlink	Removed function	Unused
namei.c	vfs_link	Removed function	Unused
namei.c	sys_link	Removed function	Unused

namei.c	vfs_rename_dir	Removed function	Unused
namei.c	vfs_rename_other	Removed function	Unused
namei.c	vfs_rename	Removed function	Unused
namei.c	do_rename	Removed function	Unused
namei.c	sys_rename	Removed function	Unused
namei.c	vfs_readlink	Removed function	Unused
namei.c	__vfs_follow_link	Removed function	Unused
namei.c	vfs_follow_link	Removed function	Unused
namei.c	page_getlink	Removed function	Unused
namei.c	page_readlink	Removed function	Unused
namei.c	page_follow_link	Removed function	Unused
namei.c	page_syslink_inode..	Removed variable declaration	Unused
neighbour.c	neigh_table_clear	Removed call to kill_tasklet()	Tasklets not supported by Lunar
open.c	filp_close	Removed call to fcntl_dirnotify()	Unused for sockets
read_write.c	generic_ro_fops	Removed struct declaration	Unused
read_write.c	default_llseek	Removed function	Seek is not supported on sockets
read_write.c	llseek	Removed function	Seek is not supported on sockets
read_write.c	sys_lseek	Removed function	Seek is not supported on sockets
read_write.c	sys_llseek	Removed function	Seek is not supported on sockets
read_write.c	sys_read	Removed call to inode_dir_notify()	Unused for sockets
read_write.c	sys_write	Removed call to inode_dir_notify()	Unused for sockets
read_write.c	do_readv_writev	Removed call to inode_dir_notify()	Unused for sockets
read_write.c	sys_pread	Removed function	Seek is not supported on sockets
read_write.c	sys_pwrite	Removed function	Seek is not supported on sockets
socket.c	move_addr_to_user	Replaced call to __put_user() with put_user()	The result is the same, but this simplifies having to implement all the internal macros for memory copies

softirq.c	do_softirq()	Added call to local_irq_enable()	Interrupts can be enabled; this prevents adding this call in multiple places throughout the code
super.c	kill_super	Removed implementation	Function is used in unmounting file systems, which will not occur in Lunar
sysctl.c	root_table	Removed entries for all tables, with the exception of CTL_NET	Unsupported by Lunar
sysctl.c	kern_table	Removed table	Unsupported by Lunar
sysctl.c	vm_table	Removed table	Unsupported by Lunar
sysctl.c	proc_table	Removed table	Unsupported by Lunar
sysctl.c	fs_table	Removed table	Unsupported by Lunar
sysctl.c	debug_table	Removed table	Unsupported by Lunar
sysctl.c	dev_table	Removed table	Unsupported by Lunar
sysctl_net.c	net_table	Removed entry for NET_802	Unsupported by Lunar
sysctl_net.c	net_table	Removed entry for NET_ETHER	Unsupported by Lunar

From the *include* directory:

File	Function/Variable	Description	Reason
asm/bitops.h	find_first_zero_bit	Changed *ebx* to *esi* register	Weaves requires the use of the frame pointer, which is stored in *ebx* (the kernel is normally compiled with the option *–fomit-frame-pointer*
asm/current.h	get_current	Removed function	Unsupported method to access current task
asm/current.h	current	Replaced macro with an externally declared struct task_struct pointer	Alternate method to access current task (see Section 3.2.3)

asm/processor.h	init_task	Replaced macro defining init_task with an external variable declaration (variable is defined in process_stub.c)	Necessary to support Lunar's implementation of processes
asm/system.h	__cli	Replaced implementation with a call to __libstack_cli()	User-level code is unable to disable interrupts via the cli command (see Section 3.2.7)
asm/system.h	__sti	Replaced implementation with a call to __libstack_sti()	User-level code is unable to enable interrupts via the sti command (see Section 3.2.7)
asm/system.h	local_irq_save	Replaced x86 assembly code with an equivalent implementation for Lunar	Lunar provides the same functionality using a different method
asm/system.h	local_irq_restore	Replaced x86 assembly code with an equivalent implementation for Lunar	Lunar provides the same functionality using a different method
asm/uaccess.h	access_ok	Removed implementation; macro now always returns 1 (true)	Feature is not applicable in Lunar
asm/uaccess.h	get_user	Replaced implementation with a memcopy	No distinction between user/kernel space in a user-level application
asm/uaccess.h	put_user	Replaced implementation with a basic assignment	No distinction between user/kernel space in a user-level application
linux/mm.h	alloc_pages	Replaced implementation with one that uses virtual memory	Unable to use kernel's low-level memory functions in Lunar
linux/proc_fs.h	proc_root_driver	Removed external variable declaration	This was a bug in the kernel was fixed in subsequent releases

linux/sched.h	root_user	Removed external variable declaration	Unused in Lunar
linux/sched.h	INIT_USER	Remove macro	Unused in Lunar
linux/sched.h	INIT_TASK	Initialized field exec_domain of the task_struct to a NULL pointer	This field of the task_struct is unused in Lunar
linux/sched.h	INIT_TASK	Initialized field active_mm of the task_struct to a NULL pointer	This field of the task_struct is unused in Lunar
linux/sched.h	INIT_TASK	Initialized field real_timer.function of the task_struct to a NULL pointer	This field of the task_struct is unused in Lunar
linux/sched.h	INIT_TASK	Initialized field user of the task_struct to a NULL pointer	This field of the task_struct is unused in Lunar
linux/sched.h	INIT_TASK	Initialized field sig of the task_struct to a NULL pointer	This field of the task_struct is unused in Lunar
linux/sched.h	capable	Removed implementation; function always returns 1 (true)	User capabilities are unnecessary in the context of Lunar and aren't supported

From the *net-tools/include* directory:

File	Function/ Variable	Description	Reason
interface.h	if_port_text	Added #define to replace if_port_text with net_tools_if_port_text	The symbol if_port_text is declared in the kernel

From the *iptables* directory:

File	Function/ Variable	Description	Reason
iptables-standalone.c	iptables	Replaced exit() with return()	Function is no longer part of an independent program

The netfilter (firewall) code is implemented as a set of kernel modules. Since Lunar does not support modules, the code must be modified to be supported being linked directly as part of the library. All the changes listed below are designed to achieve that goal.

From the *netfilter* directory:

File	Function/ Variable	Description	Reason
ip_conntrack_standalone.c	init	Renamed function to ip_conntrack_standalone_init()	Function requires a unique name
ip_conntrack_standalone.c	module_init	Removed macro call	No longer implemented as a module
ip_nat_standalone.c	init	Renamed function to ip_nat_standalone_init()	Function requires a unique name
ip_nat_standalone.c	module_init	Renamed function to ip_nat_standalone_init()	No longer implemented as a module
ip_tables.c	init	Renamed function to ip_tables_init()	Function requires a unique name
ip_tables.c	module_init	Removed macro call	No longer implemented as a module
ipt_MASQUERADE.c	init	Renamed function to ipt_MASQUERADE_init()	Function requires a unique name
ipt_MASQUERADE.c	module_init	Removed macro call	No longer implemented as a module
ipt_MIRROR.c	init	Renamed function to ipt_MIRROR_init()	Function requires a unique name
ipt_MIRROR.c	module_init	Removed macro call	No longer implemented as a module
ipt_REDIRECT.c	init	Renamed function to ipt_REDIRECT_init()	Function requires a unique name
ipt_REDIRECT.c	module_init	Removed macro call	No longer implemented as a module
ipt_REJECT.c	init	Renamed function to ipt_REJECT_init()	Function requires a unique name

ipt_REJECT.c	module_init	Removed macro call	No longer implemented as a module
ipt_limit.c	init	Renamed function to ipt_limit_init()	Function requires a unique name
ipt_limit.c	module_init	Removed macro call	No longer implemented as a module
ipt_mark.c	init	Renamed function to ipt_mark_init()	Function requires a unique name
ipt_mark.c	module_init	Removed macro call	No longer implemented as a module
ipt_multiport.c	init	Renamed function to ipt_multiport_init()	Function requires a unique name
ipt_multiport.c	module_init	Removed macro call	No longer implemented as a module
ipt_state.c	init	Renamed function to ipt_state_init()	Function requires a unique name
ipt_state.c	module_init	Removed macro call	No longer implemented as a module
ipt_tcpmss.c	init	Renamed function to ipt_tcmpmss_init()	Function requires a unique name
ipt_tcpmss.c	module_init	Removed macro call	No longer implemented as a module
ipt_tos.c	init	Renamed function to ipt_tos_init()	Function requires a unique name
ipt_tos.c	module_init	Removed macro call	No longer implemented as a module
iptables_filter.c	init	Renamed function to iptables_filter_init()	Function requires a unique name
iptables_filter.c	module_init	Removed macro call	No longer implemented as a module

Appendix C – Non-kernel Files

This appendix contains a list of all the non-kernel files that are part of Lunar, including the *_stub.c files.

dev_stub.c
The device chain for Lunar is declared in this file, as well as `eth_init()` and `eth_xmit()`.

device_init.c
This file contains the code to process the `ifconfig` and `route` commands from Lunar's configuration file.

file_interface.h
The code to manipulate the file descriptors returned by Lunar, as described in Section 3.3, is implemented here. This header file is included by stack_socket.c.

inode_stub.c
This file is a stripped down version of *fs/inode.c*, most of which contains functions which are not used by sockets.

kernel_stub.c
Provides implementations of the kernel functions `printk()` (the kernel's version of `printf()`) and `panic()`.

lib_stub.c
This file contains various routines from files in the kernel's lib directory that are used to format option lists (converting a string containing a list of comma separated integers into an integer array).

libnet/libnet.c (libnet/libnet.h)
These files contain portions of the experimental simulator level, as well as the simulator API function `timer()`.

memory_stub.c
This file contains the wrapper functions that provide memory management functionally using Glibc's `malloc()` and `free()`, rather than the kernel's own memory management routines.

misc_stub.c
Miscellaneous functions that are don't fit elsewhere.

netfilter_init.c
This file contains the code to process the `iptables` commands from Lunar's configuration file.

notifier_stub.c
The necessary functions that support the kernel's notifier chain are extracted into this file. The notifier chain, in the context of Lunar, is used when initializing interface devices.

process_stub.c
The process management code (`create_task()`, `task_switch()`, etc) is implemented here.

random_stub.c
This file implements the random number generator used by TCP to generate a random starting sequence number for connections. The kernel can access this generator via the function `get_random_bytes()`, and user-level programs can access it via the device /dev/urandom. This file includes the function `get_random_bytes()`, which is rewritten to function at user-level by reading from the host machine's device.

sched_stub.c
This file contains the scheduling code (`schedule_timeout()`, `schedule()`, etc) described in the section on process management.

signal_stub.c
This file contains the stub functions for signal-related calls. Signals will be handled in a future version.

sim_interface.c
A portion of the simulator API is implemented here, in particular, the function `read_msg()`.

sim_layer.c
This file provides the majority of the functionality for the experimental simulation layer detailed in Section 4.1 and its sub-sections. The API function `write_msg()` is implemented here.

stack_debug.c (stack_debug.h)
These files contain several debugging functions used as part of the library's development.

stack_init.c
This file provides the entry point to the Lunar's initialization functions.

stack_socket.c
The application level API described in Section 3.3 is implemented here.

time_stub.c
Lunar's "wall clock" functions are built on top of Glibc's time functions; this file contains those implementations.

timer_stub.c
The code for handling the timer interrupt and the kernel timers are stored in this file. The code here is all kernel code that has been extracted, with minor modifications made.

wait_stub.c
Wait queues are not implemented in the experimental version, since they require interaction with the Weaves' scheduler. This file contains the stub functions that are need to satisfy the linker.

write_parse.c
Lunar contains a set of functions that parse and print in a human-readable format each packet received or transmitted (this feature is a compile-time option). This file contains the relevant code.

Vita

Christopher C. Knestrick

Education

- **Virginia Polytechnic Institute and State University**, Blacksburg, VA
  - August 1999 - Present
  - Current GPA: 3.53/4.00
  - Includes courses in Computer Networks, Operating Systems, Programming Languages, Compiler Design, Theory of Algorithms and Genetic Algorithms, as well as a multidisciplinary course on commercializing new technology

- **Middle Tennessee State University,** Murfreesboro, TN
  - August 1994 – December 1997
  - BS, Aerospace with a Computer Science minor
  - GPA: 3.75/4.00, graduated Cum Laude

- **Georgia Institute of Technology,** Atlanta, GA
  - September 1993 – July 1994
  - GPA upon transfer: 3.4/4.00

Work Experience

- **Graduate Research Assistant**, Virginia Polytechnic Institute and State University
  - August 2002 – Present
  - Integrated Research and Education in Advanced Networking (IREAN) Fellowship, funded by the National Science Foundation

- **Graduate Teaching Assistant**, Virginia Polytechnic Institute and State University
  - August 1999 – May 2002
  - Assisted with introductory C++ courses as well as graduate computer networking

- **Instructor,** Virginia Polytechnic Institute and State University
  - Summer 2000, Summer 2001, Fall 2001
  - Instructor for introductory C++ course
  - Received Computer Science department's "Outstanding Graduate Teaching Assistant" award for teaching by a graduate student

- **Integration/Interface Specialist,** Valley Health System, Winchester, VA
  - March 1998 - July 1999
  - Promoted to Senior Interface Programmer, March 1999
  - Responsible for designing, implementing and maintaining software interfaces between the primary hospital computer and departmental ancillary systems using a DataGate Interface Engine