Network Simulation Cradle

Final Report for COMP420Y

Sam Jansen

Department of Computer Science

Hamilton, New Zealand

October 17, 2003

Abstract

Network simulators commonly use abstractions of network protocol implementations. Results from simulations using these abstractions are not as convincing as they would be if real world network code were used.

This report describes the creation of the *Network Simulation Cradle* that is used to allow a real network stack to work with a network simulator. The process of moving the FreeBSD 5 network stack from kernel specific code into a user space simulation library is discussed, as is the integration within the network simulator NS-2.

Also a preliminary analysis of differences between simulations using NS-2's and FreeBSD's TCP/IP implementation is presented along with preliminary testing of the performance of a cradle incorporating real world code.

Acknowledgements

I would like to acknowledge the WAND group for their help and support. My supervisor, Tony McGregor, should also be acknowledged. He has managed to keep me going in the right direction this past year.

Contents

List of Figures

Chapter 1

Introduction

Network simulation is an important tool in developing, testing and evaluating network protocols. Simulation can be used without the target physical hardware, making it economical and practical for almost any scale of network topology and setup. It is possible to simulate a link of *any* bandwidth and delay, even if such a link is currently impossible in the real world. With simulation, it is possible to set each simulated node to use any desired software. This means that meaning deploying software is not an issue. Results are also easier to obtain and analyse, because extracting information from important points in the simulated network is as done by simply parsing the generated trace files.

Simulation is only of use if the results are accurate, an inaccurate simulator is not useful at all. The *Network Simulation Cradle* aims to increase the accuracy of network simulation by using a real life network stack in a simulator. Most network simulators use abstractions of network protocols, rather than the real thing, making their results less convincing. S.Y. Wang reports that the simulator OPNET uses a simplified finite state machine to model complex TCP protocol processing. [19] NS-2 uses a model based on BSD TCP, it is implemented as a set of classes using inheritance. Neither uses protocol code that is used in real world networking.

1.1 Improving Network Simulation

Wang states that “Simulation results are not as convincing as those produced by real hardware and software equipment.”[19] This statement is followed by an explanation of the fact that most existing network simulators can only simulate real life network protocol implementations with limited detail, which can lead to incorrect results. Another paper includes a similar statement, “running the actual TCP code is preferred to running an abstract specification of the protocol.”[2] Brakmo and Peterson go on to discuss how the BSD implementations of TCP are quite important with respect to timers. Simulators often use more accurate round trip time measurements than those used in the BSD implementation, making results differ. [2]

Using real world network stacks in a simulator should make results more accurate, but it is not clear how such stacks should be integrated with a simulator. The network simulator NCTUns shows how it is possible to use the network stack of the simulators machine. NCTUns also has many limitations that reduces its usefulness as a simulator, as section 2.1.3 explains.

1.2 Goals

The goals of this project are broader than just integrating a single network stack into a simulator. To make the project useful in the future, there should be provision to allow future similar network stacks to be used for simulation. Also, work should be done towards the goal of having different operating system's network stacks available for simulation use. To realise this goal, the idea of a *simulation cradle* is proposed. The cradle provides facilities which the environment the stack would normally run in would usually provide. Generally, the network stack is a part of the kernel, so the cradle needs to provide those kernel specific functions the network stack needs. The basic idea of this cradle is expressed in figure 1.1.

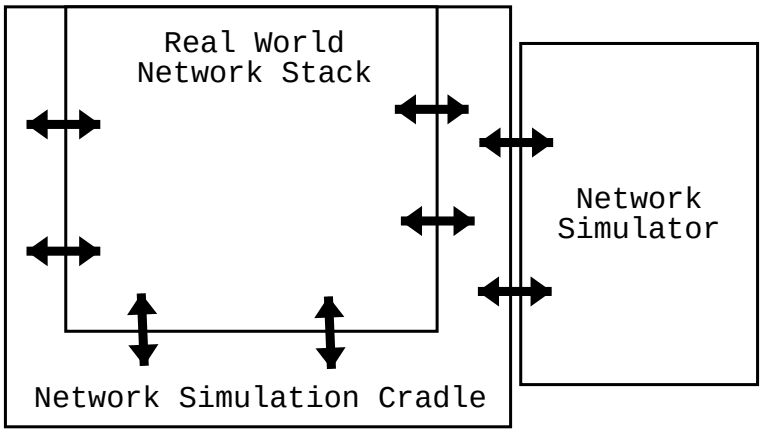

Figure 1.1: Simulation cradle diagram

To make it easy to support the current and future versions of a network stack inside the cradle, the network stack source code should not be changed by hand. Changing the source would mean such changes would need to be done on every version of network stack used with the cradle; this is something that should be avoided.

The first step is evaluating the feasibility of a cradle, and assuming such an approach can be used, the second step would be to build a generic cradle. Lastly, the cradle should be integrated with a network simulator.

There are some likely issues, such as performance. A real world network stack needs to examine the real data and will likely involve copying data. Abstractions of network stacks used in simulators do not need to use the actual data. This limitation, and the fact that the real world network stack could possibly do a lot of extra processing, might mean simulations done with such a stack are very slow.

Other problems include the possibly large amount of programming required. Moving kernel code into

user space is not often done, and complex even for a small piece of code. Moving an entire system such as a network stack into user space possibly includes a substantial programming effort.

Whether there is any difference between a real network stack and the abstraction is a question worth asking. One of the goals of the project is to show such a difference, but this is not needed. Using a real network stack gives a greater confidence in results, even if testing has not shown any differences before.

There is no guarantee that moving a network stack into user space can be done without extensive hand modification. With a large amount of hand modification, the question of whether the stack operates like it would without the modification also becomes an issue.

The goals include solving all the above issues by building and evaluating a *Network Simulation Cradle*.

The next chapter, background, goes over existing approaches to network simulation and describes some related material. The methodology chapter details the entire development cycle of the cradle and looks at the design decisions made. The next chapter shows results of the methodology, while the last chapter makes some brief conclusions about the *Network Simulation Cradle* project.

Chapter 2

Background

Using real world network code for simulation is not a new concept, nor is making the FreeBSD network stack work in user space with minimal code modification. Building a cradle to allow recent network stacks to work with an existing event simulator, is a new development, however. This chapter discusses background information and work related to the *Network Simulation Cradle*.

Current network simulation is covered first. The concept of discrete event simulators is discussed in section 2.1.1. Network emulation follows, where a subsection on the simulator NCTUns 1.0 is last in the current network simulation section.

Two projects are discussed in the real world network code section: Alpine, a project which ports the FreeBSD 3.3 network stack to user space is one. The other is the BSD Network Stack Virtualisation project which modified the FreeBSD kernel to allow multiple instances of the network stack to run at the same time. These are covered in sections 2.2.1 and 2.2.2, respectively.

2.1 Current Network Simulation

Most network simulation is done using discrete event simulators, OMNeT++ [17], REAL [16], NS-2 [15] and SSFnet [14] are all examples of such simulators. The way these work is discussed in the next section, 2.1.1.

2.1.1 Discrete Event Simulators

The core of a discrete event simulation is very simple. It involves a loop which pops the next event off an event queue, advances time to that of the event, then executes the event. This event may or may not add new events somewhere in the queue, which is sorted in temporal order. Generally there is some way of getting information out of the simulator, such as a trace file.

Time is an important concept in discrete event simulation. Simulations are not often run in real time. Instead a simulated time is used. The simulator will keep a global reference to the simulated time. Before an

event is executed, the simulated time is advanced to the time at which this event is scheduled to happen. This way the time advances in uneven steps and time for which no events are queued does not take up any real time. NS-2's implementation of this uses a `double` to store the absolute time, which the application programmer can access via the `Scheduler` object.

Though the core is simple, the simulator as a whole is complex. The supporting code is where the complexity arrives. For example, NS-2 contains an entire scripting language interpreter inside it (OTcl) which allows easy set up of a simulations topology. It also contains facilities to trace events and to handle timing. It has an extensible packet header class, a custom seeded random number generator, facilities to allow the simulator to be used as an emulator and many different protocol implementations and supporting code for these protocols. NS-2 has roughly 330,000 lines of C++ and OTcl code. NS-2 is just an example, all discrete event simulators need to provide similar facilities.

2.1.2 Network Emulation

Network emulation refers to actual network traffic passing through some software which might do some analysis or perhaps modify the traffic in some way. The Emulation Network in the WAND group is used for testing and evaluation of networking software and hardware. The scale is limited; it is made up 24 emulation machines and one central controlling computer. Setup of such a network is time consuming and expensive: in addition to the aforementioned 25 computers, a Cisco 2950 switch and a Cyclades 32 port terminal server are included in the network. Each emulation machine also has a 4 port network interface controller. The controlling machine includes special capture cards (known as DAG [6] cards) to allow easier capture and processing of network traffic. This network has no easy way of adding latency and bandwidth bottlenecks, which means creating adverse conditions on the network is difficult. It is possible to use Dummynet [13] to add latency, but this is a lot of work. There is a project to solve this issue; a non blocking crossbar Ethernet switch is being created for the network, but the cost involved is large.

Other network emulation done in the WAND group include validating the WAND simulator [7]. This was done by setting up a physical network with FreeBSD machines using Dummynet to add latency. Dummynet is one example of network emulation software, NIST Net [10] is another, it claims to "allow controlled, reproducible experiments with network performance sensitive/adaptive applications and control protocols in a simple laboratory setting".

NS-2 also provides some network emulation functionality, it is able to capture packets from the live network and drop, delay, re-order, or duplicate them. Emulation, especially in the case of a network simulator like NS-2, is interesting because it is using *real world* data from *real world* network stacks.

Emulation offers something simulation never can: it is performed on a real network, using actual equipment and real software. However, it is very limited compared to simulation in other areas; for example scale.

The WAND Emulation Network described earlier requires a lot of setup and is expensive, yet only contains 24 emulation machines. There is no theoretical limit to the number of nodes a simulator can handle, and increasing the size of a simulation does not cost anything. The factors to consider are RAM, disk space and the small amount of time taken to change a simulation script. In general, changing the simulation is a simple step, though it would be complex in the case a huge amount of nodes being required (a million, for example). Also, network emulation must of course be run in real time, where simulation can sometimes simulate large time periods in a small amount of time. In the case of a million nodes, the simulation might run in greater than real time because the hardware it is run on would limit performance.

2.1.3 NCTUns 1.0

NCTUns 1.0 is a simulator which attempts to make use of a real world network stack for simulation. NCTUns [19] uses a tunnel network interface to use the local machines network stack. Tunnel devices are available on most UNIX machines and allow packets to be written to and read from a special device file. To the kernel, it appears as though packets have arrived from the link layer when data is written to the device file. This means the packet will go through all the normal processing routines: the TCP/IP stack. When a packet is read from the tunnel device, the first packet in the tunnel interfaces output queue is copied to the reading application.

The first obvious problem with this is that timing is still handled by the kernel, and hence there is no simulated time. This is solved in NCTUns by modifying the kernel to use a special virtual timer. The virtual time resides in a memory mapped region accessible both to kernel and user space. The kernel code is modified, which has negative repercussions. First of all, each kernel NCTUns might run on needs to be patched. This means that for every differing version of each operating system, a patch needs to be written. Secondly, the kernel needs to be patched on all simulation machines. This is not always an option, especially in a laboratory setting where the users do not often have full control of the computers available.

One of the problems discussed in the NCTUns 1.0 paper [19] is that of performance. Because of the real network code being executed, abstractions of stacks cannot be executed: there is more processing necessary in a real stack. Real data is transferred throughout the system and also needs to be copied. To some extent this problem would apply to a real network stack used inside the *Network Simulation Cradle*, as it would need the actual data to do packet processing on too. The *Network Simulation Cradle* only needs the packet headers, the entire packets do not need to be used. A lot of system calls are needed in NCTUns's case, which also reduces performance. Even with these issues Wang concludes that "the performance of the NCTUns 1.0 in its current form is still satisfactory".

NCTUns achieved reasonably scalability - more than 256 nodes - by working around an initial limitation imposed by BSD UNIX, due to 8-bit integers being used to identify devices.

Other problems with NCTUns include the fact that it is difficult to get statistics out of the stack code

running inside the kernel, and only one stack type is used in simulation. Other stacks could be used in NCTUns by a distributed simulation approach, but this requires at least one machine per stack simulated, each running the specific operating system version wanting to be simulated. This means the simulation is more like an emulation, which loses some of the benefits of simulation. Also, because the simulation network stacks run inside the kernel, the simulation computer cannot be used for other activities while simulating, such activity could make the kernel behave quite differently.

2.2 Real World Network Code

NCTUns 1.0 (section 2.1.3) showed one way of using an actual network stack within simulation, though it introduced some limitations. Another approach, taken by this project, is to make use of a network stack in user space. This section looks into how a large section of kernel code can run in user space.

2.2.1 Alpine

Alpine [4], or Application-Level Protocol Infrastructure for Network Experimentation, is a "user-level infrastructure for network protocol development". The FreeBSD 3.3 network stack was taken out of the kernel and made to run in user space in this project. The motivation for this project was rapid protocol development, developing a protocol in user space is a lot easier than developing in kernel space. The network stack was almost unmodified in its transition to user space and the basic BSD socket API was kept the same. The web page [1] shows the amount of modification done to the FreeBSD 3.3 network stack. The amount of code changed is fairly minimal, but the creators of Alpine did have to modify the code by hand to get the stack working in user space. Although Alpine is not directly related to network simulation, it does give an example of an existing FreeBSD network stack in user space. Alpine is no longer maintained and uses a version of FreeBSD from late 1998; so the code is often not applicable to moving the current FreeBSD 5.0 network stack to user space.

The design used by Alpine to move the FreeBSD 3.3 network stack in to user space is very similar to the idea of a cradle. Figure 2.1 shows the image from the Alpine web page showing the design used. The top two layers of this diagram are the only ones which are different to the *Network Simulation Cradle*. Alpine re-implemented many system call functions so applications linked with Alpine would call the new versions of the system calls, and hence use the network stack of Alpine rather than the system one. Alpine was designed to be used as a shared library loaded before libc, to allow overriding of network related system calls. Such design is not a goal of this project, but it does show a method of being able to use existing applications as part of simulation.

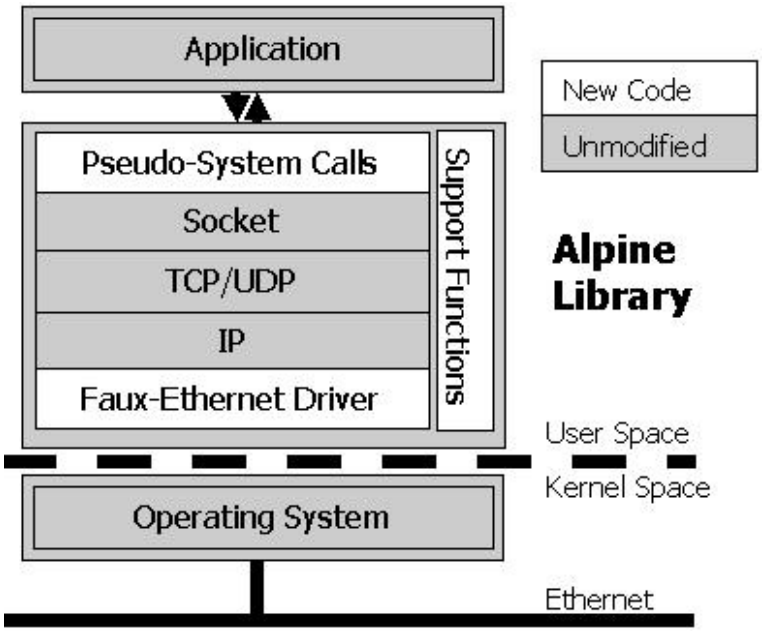

Figure 2.1: Alpine Design [4]

2.2.2 BSD Network Stack Virtualisation

The BSD Network Stack Virtualisation [20] project took a recent FreeBSD kernel and modified it to allow multiple virtual network stack images to run at once. The code still runs inside the kernel, but this project showed how to make multiple stacks by replacing global variables and their references by hand. Patches for the kernel were released initially for FreeBSD 4.7, and later for FreeBSD 4.8 (May 2003).

This project, like Alpine, is not directly related to network simulation but is relevant to the *Network Simulation Cradle*. Simulation requires the use of many nodes and connections, something not easily simulated with only one network stack. This project showed how it was possible to run multiple network stacks. However, the code was modified by hand and designed for kernel mode, two important facts which go against some of the goals of this project.

Chapter 3

Methodology

The *Network Simulation Cradle* was developed in two distinct parts: moving the FreeBSD network stack into user space, and integrating the cradle and stack into the NS-2 network simulator.

Compiling and linking is covered fist, followed by a section on initialisation. How packets are sent and received is covered in section 3.3, while section 3.4 describes how multiple instances of the network stack are handled. Section 3.5 is devoted to initial testing. Finally, section 3.6 describes integration with NS-2.

3.1 Compilation and Linking

The FreeBSD kernel is a large piece of software, it is made up of roughly two million lines of C source code. Because of its size and nature, it uses a complex build system. This section describes how the code can be compiled and linked outside of the kernel.

3.1.1 Compiling

The source code cannot merely be moved into another directory and built without further further changes to the build scripts. It is fairly simple to identify the network code and copy it to a directory outside the kernel tree, but not so simple to find the compiler flags necessary to build this code.

One of the less obvious features here was the forcing of extra include files on the command line passed to `gcc`. The easiest way to find out how the kernel was compiled was to observe an actual kernel build and find the required flags from there. The method used to compile a FreeBSD kernel is documented in the FreeBSD Hankbook [5]. The UNIX `script` command logs a terminal session, it allows the output of the build process to be written to a text file for later analysis. This process allows easy observation of the flags passed to `gcc` during the kernel compile. The other way to solve this problem is to understand the makefile system used. This system is large and complex, it was decided that this approach it a lot more work than inspecting the build process.

3.1.2 Linking

Linking the source code means all function dependencies need to be satisfied before an executable is able to be created. When the network stack code is taken outside of the kernel source tree and compiled, the many functions referenced in the networking code that are defined elsewhere need to also exist. When an executable is compiled and linked, errors appear as "undefined references" in the compiler output.

A simple makefile to build the network stack outside the kernel coupled with a Perl script to parse the error output results in 192 symbols that are not defined. Each one of these symbols needs to be defined before an executable can be linked together.

Implementing 192 functions by hand is not realistic in the time frame on a COMP420 project. These are all sorts of internal kernel functions; most have no documentation at all. It is important to realise that while linking an executable might require all 192 functions to be implemented, many of these functions may not be called during normal operation.

The common way to find out what functions are being called and how often during the running of a program is to profile the program. This is often more complex in the case where the program is the operating system kernel. However, in the case of FreeBSD, there already exists a way to easily produce profiling information from the kernel. The existing kernel was built with profiling information enabled and rebooted. Profiling was turned on for the duration of a single ping to a computer on the same network. Profiling was then turned off and the information dumped to a file. Processing the information both by hand and with the help of Perl scripts found that about 50 of the undefined functions were called during the ping.

In an attempt to visualise the complexity involved, the kernel profile output was processed by some simple scripts and used as input to a program called VCG [18]. VCG stands for Visualisation of Compiler Graphs and is a free program that will create visual call graphs. Figure 3.1 shows the graph created. Each line on the graph represents one function calling another. At each end of each line there would be the function name printed if there were room. The mass of black shown is many functions interacting, the black is a result of one line per function call. It is obvious this graph is of no practical use, it is included only to show the complexity involved even when only a simple ping is performed.

Given these findings it makes sense to attempt to satisfy the external dependencies by using kernel code from outside the network stack first. A lot of kernel code is not suitable to be run in user space, but creating an executable that links is an important step. Once that step is complete replacing code is simple.

Finding important kernel files to include in the build was a case of trial and error. After including the IPv6 related source files as well as some cryptographic source files, the dependencies had only increased slightly. Some scripts were built to copy kernel files from the kernel source tree and count the number of undefined references. Only files that reduced the number of errors were kept for future builds. This reduced the number of undefined references to below 100.

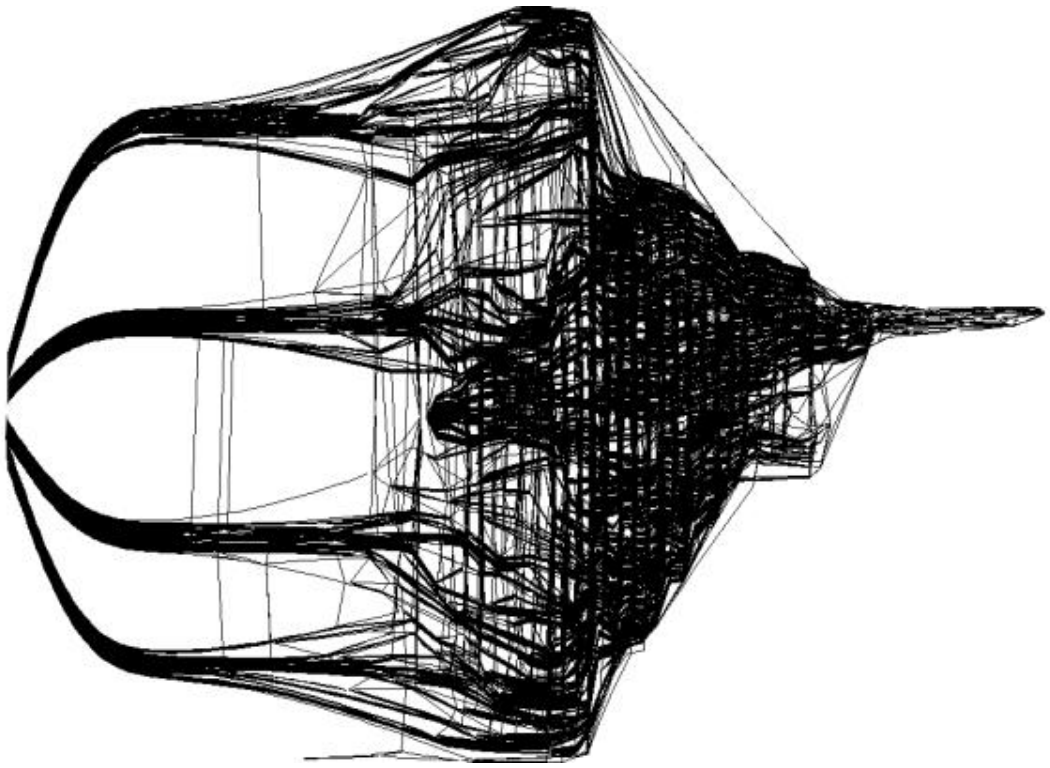

Figure 3.1: Kernel call graph

The remaining functions were implemented as stubs; partly by hand and partly via the use of another Perl script. Each of these functions only had a single line, `assert(0);`. This allowed the stack to be compiled and made it obvious which functions were being called; if any of the stub functions were called the assert would fail.

3.2 Initialisation

Before executing the network stack and supporting kernel code, the global data structures must be initialised. The initialisation functions need to be called in the same order as they would have been during boot process of a FreeBSD kernel. The Alpine project (described in section 2.2.1) initialised the FreeBSD 3.3 network stack code, and looked to be a good starting point. However, kernel initialisation mechanisms have changed between FreeBSD 3.3 and 5.0 significantly, the work in Alpine was of no use. This section discusses how the issues involved in initialisation are solved in the *Network Simulation Cradle*.

3.2.1 Linker Sets

FreeBSD 5.0 uses something called a “Linker Set” to manage initialisation objects. These are created by a set of macros in various C files to set up some information, including a callback function, that is processed during kernel initialisation.

Each macro creates a global structure that includes 2 fields used to decide the order in which the callback function will be called. These structures are organised in a global array and sorted depending upon the defined precedence parameters. During normal kernel operation the sorted array would then be traversed, each callback function being called in turn. The very last function would be the scheduler, which does not return.

There is no scheduler which needs to be called in user space of course, this function will return. That is fine, but the complexity arrives when considering how the array is set up. Each macro defines a single global structure. These structure definitions are distributed arbitrarily among source files. The question here is: how do these structures instantiated in different source files form an array? The answer lies in the use of a special compiler-specific attribute. The `section` attribute allows data to be put within a specific section of the generated object files and final executable. By putting all the structures in the same `section`, they are effectively organised as an array once the executable is loaded into memory. This array is known as a linker set.

The routine used to sort and traverse the linker set is called `mi_startup` in the kernel. This function is included in the source file `kern/init_main.c`, a file dealing with low-level startup procedures. Most of the functions in this file deal with code that does not make sense in user space. Initialisation of low level kernel systems such as the virtual memory system, creation of the process which handles swapping and building of the file descriptor table all make little sense for a user space library. The cradle provides the functionality which `mi_startup` normally would, the implementation of the function is very similar to the original. For pragmatic reasons the sort routine has changed from the original bubble sort to a merge sort.

Particular care must be taken to avoid bus and segmentation errors when implementing this function because of the nature of the memory used. Because the linker set is defined in a different section in the object file, it is loaded into a different section in memory. Bus errors are especially prevalent here because they will be generated from a write to an invalid memory page in this region.

3.2.2 Per CPU Information

FreeBSD uses a structure to hold information about each CPU. This is called per CPU information, and is held in an instance `struct pcpu` for each CPU. This structure keeps pointers to the current thread the CPU is executing, statistics and counters, and other information. A global list is kept of pointers to these structures.

In the normal operation of the FreeBSD kernel, it makes sense to have such structures, but there is little reason for their existence when the network stack is ported to user space. There is a lot of code that relies upon per CPU information, however, meaning that such a facility cannot be removed without network stack code modification. The most obvious reliance upon per CPU information are the functions which take pointers to a thread structure and the references to `curthread`. The functions used to send and receive data from the kernel socket (`sosend` and `soreceive`) both take pointers to a thread structure. The obvious candidate here

is the currently running thread; the global `curthread` seems to provide this. Unfortunately, `curthread` is not a simple global variable. It is instead a preprocessor macro; on an Intel i386 machine it expands out to something similar to figure 3.2.

```
#define __PCPU_GET(name) ({                                             \
        __pcpu_type(name) __result;                                     \
 ... omitted for brevity ...
        } else if (sizeof(__result) == 4) {                             \
                u_int __i;                                              \
                __asm __volatile("movl %%fs:%1,%0"                      \
                    : "=r" (__i)                                        \
                    : "m" (*(u_int *)(__pcpu_offset(name))));           \
                __result = *(__pcpu_type(name) *)&__i;                  \
        }                                                               \
 ... omitted for brevity ...
        __result;                                                       \
})
```

Figure 3.2: Example per CPU source code

There is obviously a complex low-level interaction involved in finding `curthread`, it is not as simple as returning the current thread pointer from the currently running CPU's `struct pcpu`. The source code in figure 3.2 made the executable segmentation fault every time during testing. The use of segments and offsets in the assembler code suggest that some trick is being done with memory. It is likely that the code attempts to reference a memory location that is not accessible to the application when it runs in user space. This will be memory that is protected by the kernel: if the application were running in kernel space, this protection would not apply. The segmentation fault encountered will be because of the attempted read of protected memory.

There is no need for such per CPU information to be kept for a user space version of the network stack, this code could be replaced with a simple variable. To make this change another problem needs to be solved: `curthread` is defined in a system header file. System headers should not be changed to allow the *Network Simulation Cradle* to compile and run. The solution used here was to create an `override/` directory which is first in the include path. This allows any header to be put in the right subdirectory and it will be included instead of the usual header. The single header `pcpu.h` lives in the subdirectory `override/sys/` to allow changing the definition of `curthread`. The header file is only slightly modified, the following lines were added:

```
#define curthread __pcpu.pc_curthread
extern struct pcpu __pcpu;
```

The structure `__pcpu` is normally set up by a low level initialisation routine. Such functions had to be modified to work in user space; mainly by removing code that did not work.

This is a section of the project that requires some amount of effort to update for other versions of the

FreeBSD stack. Some of the initialisation done in low level routines such as `init386` changed noticeably between FreeBSD 5.0 and 5.1. Also, the overriding of the `pcpu.h` header means the header needs to be kept up to date with the network stack being used alongside the cradle.

3.3 Sending and Receiving Packets

Once the stack was compiled and initialised, the next step was to set up and test functionality by queueing packets and checking that they are being processed correctly. Normally the link layer will pass the packets onto the next layer, the Internet Protocol in our case. The most common case is for the link layer to be an Ethernet driver. There is no Ethernet driver in this project; but there needs to be code to perform the same functions as one. An interface needs to be created that is capable of sending and receiving packets. The interface created is described in the next section, followed by a section detailing the process used to queue packets using this interface.

3.3.1 Fake Ethernet Driver

D. Ely in the Alpine project (see section 2.2.1) created a fake Ethernet driver that had rather simple functions used to send and receive packets. The same approach was used, creating a fake Ethernet driver and attaching it as an interface. The interface to an Ethernet driver is actually very simple. A structure of type `ifnet` is set up with three callback functions: init, output and ioctl. Other attributes include maximum transfer unit (MTU), name and maximum queue length. The structure is then passed to the network stack function `if_attach`. The callback functions were very easy to write: the `init` function does not need to do anything, the output function did nothing initially but was later attached into a simulator to queue the packet in the simulator, and the `ioctl` function again does not need to do anything.

The interface was attached easily, but giving it an IP address did not prove so simple. Because of the number of stub functions implemented earlier and also the amount of supporting kernel code used initially, a lot of changes were needed before the interface was given an IP address.

3.3.2 Queueing Packets

One other Ethernet driver function needed to be implemented, an input function. This function is responsible for queueing a packet. Initially, this code was based upon the code Alpine used: it got a handle on the IP packet queue and calls a function to push the packet onto the queue. At that point the stack must be told there are new packets waiting, so `schednetisr` must be called. After upgrading to FreeBSD 5.1 the method of queueing packets changed slightly. The method above was changed to a single function call, `netisr_dispatch`.

When data is queued by an interface driver the data must be copied into a buffer the network stack owns. This process is more complex than normal memory allocation (`malloc`) and copy (`memcpy`) done in user space applications. Network buffers are handled by structures called mbufs in the FreeBSD network stack. Unfortunately, the file which implements them was left out of the build because it relied upon too many kernel-specific functions. Implementing the mbuf functions turned out to be fairly simple; they are well documented in both man pages and books [8] and the kernel source code is fairly easy to read.

3.4 Multiple Network Stacks

The FreeBSD network stack was written as part of an operating system for one computer, it was not written to allow multiple instances. Simulating a single computer is not very useful. Simulations are often done of a large number of computers. The following section discusses three different approaches to solving this problem. Section 3.4.2 describes the primary issue faced with these approaches.

This chapter then describes the implementation of a C parser to replace global variables, from a naive implementation (section 3.4.3) to more intelligent designs discussed in section 3.4.4 and section 3.4.5. How the parser replaces the global variables is explored in section 3.4.6. Lastly, problems faced due to initialising multiple network stacks is discussed in section 3.4.7.

3.4.1 Threads and Fork

The `fork` system call creates an exact replica of the current process; making another independent process with the exact same data and executable code. This would be the simplest way of allowing multiple stacks: call `fork` for every computer simulated. Each process would then need some way of communicating, Unix domain sockets would probably be a good choice here. The problem with this choice is that it has a high cost per stack. Forking for every new host in a simulation does not scale very well, for example the maximum processes a user may have at any one time is normally limited, 4096 is probably an upper limit on many systems. Even if the administrator of the computer allows users to have large numbers of processes open, the overhead of creating all the processes and the memory requirements would both be significant. The real advantage of using `fork` is the fact that a complete copy of the executable is made, global data and all. This means there are no problems with re-entrancy.

A more efficient way would be to have a thread for every simulated network stack. Creating threads is a lot faster [12] than forking and threads do not require as much memory overhead.

If the simulator is single-threaded, threads are not necessary in the cradle at all. In a single-threaded application there only needs to be some way of knowing which virtual stack is currently doing any processing in the network code at any one time.

Without using `fork`, a copy of all the data in the executable is not done, meaning potential problems arrive with global variables.

3.4.2 Global Variables

Threaded or not, there is one major hurdle standing in the way of multiple instances of the stack running together: global variables. The problem with the network stack is that it is not reentrant. The Oxford English Dictionary defines reentrant as the following:

> *re-entrant*: Of, pertaining to, or designating a program or subprogram which may be called or entered many times concurrently from one or several programs without alteration of the results obtained from any one execution. [11]

The only memory that will be accessed concurrently by multiple threads or in multiple calls to the network stack is that which contains the global variables. All other information is local to each thread of execution. There are many global variables defined in the network stack, the utility `nm` was used to give an initial count: 429 global variable symbols were found.

Many of these are important, but not all of them. The BSD Network Stack Virtualisation (section 2.2.2) project has already gone through the FreeBSD 4.8 kernel by hand finding the important global variables used by the network stack. This project released a large patch to the kernel which wraps these variables up in a single structure. This structure gives a good idea of what variables are important; but it was not conclusive for this project because of the different FreeBSD versions: 4.8 versus 5.1. FreeBSD 5.1 makes many architectural changes from 4.8, hence the change in the major version number. One of the bigger changes includes fine grained kernel locking, and though this does not effect the global variables a lot, other changes to the network stack do.

Global symbols need to be changed by a program which is able to handle the current network stack code and also code from future network stacks. Each definition of a global needs to be changed to an array to accommodate multiple stacks. References to the same global variables need to then be changed to array references. The array needs to be referenced by the index to the currently running network stack. This could possibly be modelled by a global variable. However, using a variable is not very flexible and does not work in the case of multiple threads. Using a function, such as `get_stack_id()` means there is great flexibility in the possible implementation, it would be possible to use either thread specific data (which requires a function call to retrieve the data) or to return a simple variable.

3.4.3 Naive Approach to Replacing Global Variables

There needs to be a programmatic way of changing all global variables and all references to global variables. The first approach to this problem might be to create some simple text replacement scripts in Sed or Perl which would be able to find specified globals and change them appropriately. There are problems with this approach. The first becomes apparent with macros. Following is a line of source that is not possible to parse with a simple script. There is no obvious way to change this global variable into an array without knowing exactly how the macro expands. It would be possible to account for every special macro case; but this is perhaps too specific to the source code being changed.

```
LIST_HEAD(, gif_softc) gif_softc_list;
```

Getting rid of macros is not hard; it involves only changing the build process slightly so the `-E` option is passed to `gcc` so it outputs preprocessed C files. However, further complexities arise for a text based approach to modification of global variables. Without understanding the C language, it is hard for a script to differentiate types from variables, and it will not know anything about scoping. Without a fairly thorough understanding of C, a simplistic text substitution would be unable to correctly distinguish globals and their references from local variables. Consider the following case:

```
struct ifnet ifnet;

void function()
{
    struct ifnet ifnet;
```

If the global in question is named `ifnet` it is hard to know exactly what should be replaced in the above code. Of the four appearances of `ifnet`, only one needs to be changed (the first occurance is the global).

3.4.4 Initial C Parser

Rather than write scripts to do substitution, it is possibly to write a program that can make informed decisions because it understands (at least some of) the C grammar. There are many different ways to create such a program, normally called a parser. Writing them by hand is generally not a good choice because of the large amount of work involved, so Perl or Sed are not the best choices to write a C parser in. Instead the commonly used open source compiler compiler tools were used: Flex and Bison. It is fairly easy to find example lexical analysers and Bison grammars for C, though harder for the current International Standards Organisation (ISO) C standard.

Initially it seemed a better idea to write as minimal a parser as possible. Rather than basing the grammar on an actual C grammar, write a parser which only knew a small amount of C; just enough to find global

variables and references to the global variables. Getting a parser like this to understand a majority of source files was easy, however it was unable to identify all the quirks of C and was liable to get confused on some of the more obscure C syntax, such as `typedef`'d function pointers.

3.4.5 Design of a C Parser

By basing the Bison grammar on an existing C grammar written for a slightly older standard it was possible to create a C parser in a short time. To parse C correctly, the parser needs to know the type names defined with the `typedef` keyword. The Bison grammar first had to be extended to have a set of type names which was updated whenever a typedef was encountered. This set is then checked whenever a non-reserved identifier is found.

Though this parser can accept standard C, it still lacked a few features in practice. Preprocessed source code still has lines that look like preprocessor directives. The preprocessor outputs `#line` directives to allow the compiler to add the correct debugging information. It was easier to ignore these directives in the lexical analyser at the start, but ideally they should be passed through to the compiler so the correct debugging information is created. Even with the `#line` directives removed there were two more situations to account for above the basic C grammar. First, there are extensions to the language that evolved in latter standards. These are catered for by extending the grammar.

Second, there are compiler specific options. These are common in the FreeBSD kernel code, section 3.2 describes the use of one `gcc` specific attribute. These are not simple to parse, adding them to the grammar would be a large amount of work. The lexical analyser also cannot account for them easily. The reason for this is because they require matching of left and right brackets, not something a lexical analyser can do. To confound this situation further, there needs to be a way to pass the whitespace through from the lexical analyser to the parser. The whitespace is needed so a file without globals, for example, would not be modified at all after being parsed. Ideally, the actual whitespace would be passed through along with the compiler directive with an actual token that is important. So the lexical analyser returns the token and allows the parser to access the text including the token and all effective whitespace following it. To implement this, a modification of the traditional lexical analyser/parser mechanism was needed.

Figure 3.3 shows the flow of data through the parser. The interesting step here is the intermediate parser. The design of this step of the parser is simple but it reduces the complexity of the overall process significantly. This intermediate parser allows the same tokens to be passed from lexical analyser to parser but buffers all whitespace and compiler directives and adds the text from these tokens into a global variable. Once a non-whitespace token is found, this token is returned to the parser. The parser then is able to put the text buffer in its own data structure. The lexical analyser was modified so it has enough knowledge to parse compiler directives; even though this type of operation is not normally a function of a lexical analyser. Most compiler

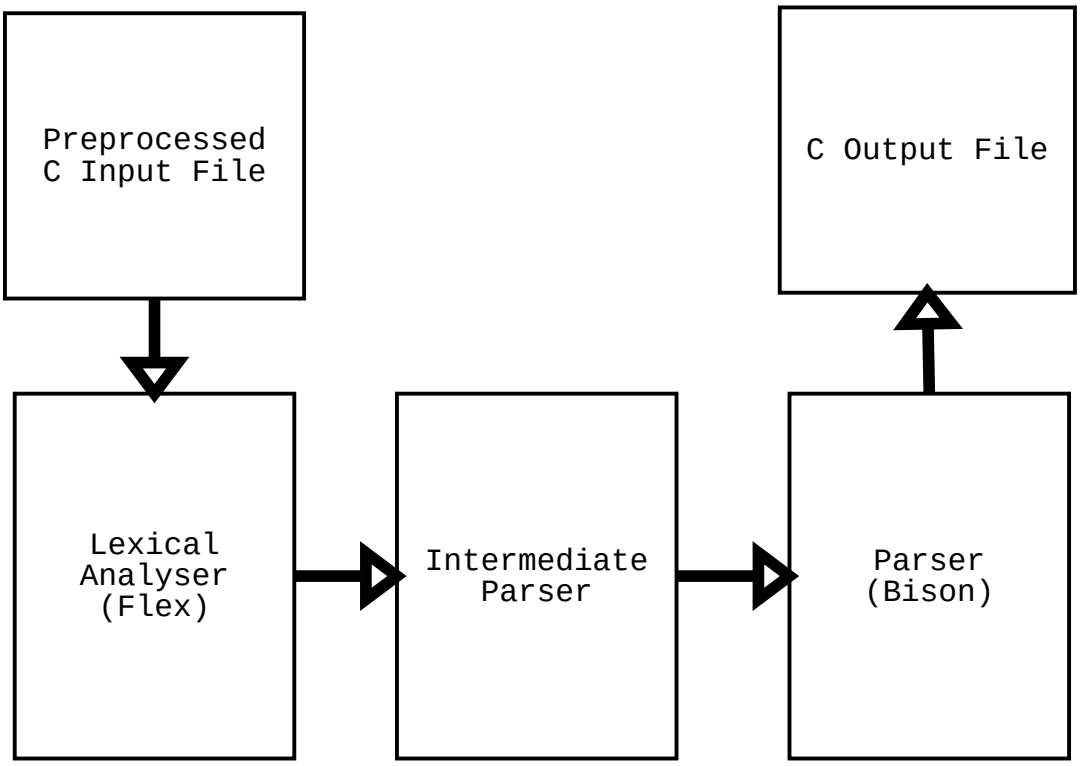

Figure 3.3: Parser design

directives are returned to the parser as whitespace, though there are a couple that need to be treated as types such as `__builtin_va_arg`, which is used for variable argument lists.

3.4.6 Replacing Global Variables

Whenever a global variable definition is found, the function `handle_global` is called. This function first checks whether the variable is one that needs to be changed; a hash set of important globals contains this information. This set is initialised from a text file upon parser startup. If the global variable is one which needs to be changed, the symbol name has `global_` prepended to it and is turned into an array. For example; the global variable `struct ifnet ifaddr;` would be transformed to `struct ifnet global_ifaddr[NUM_STACKS];`. The first time a global variable is found in an input file, the line `#include "support/thread.h"` is added. This file adds the definition of the `NUM_STACKS` constant and the `get_stack_id()` function. Though the source file has already been preprocessed, it will again be preprocessed when it is compiled, so adding a `#include` line is allowed.

References to global variables are handled in a similar way, although this time the function `handle_global_reference` is used. The variable identifier has `global_` prepended and `[get_stack_id()]` appended. For example, `ifaddr` would be replaced with `global_ifaddr[get_stack_id()]`. This only happens in the case the global is an one that is included in the aforementioned hash set.

If the global variable definition also includes assignment, each variable in the array needs to be assigned.

This is done with an initialiser list. If NUM_STACKS were equal to 4, `int ipforwarding = 0;` would be transformed to `int global_forwarding[NUM_STACKS] = { 0, 0, 0, 0 };`.

The above situation can create erroneous code in one situation. If the assignment relies upon another global variable, a function call will be introduced (`get_stack_id()`), which cannot happen in a top level declaration. Consider the following source code:

```
int ipforwarding = 0;
static struct sysctl_oid sysctl___net_inet_ip_forwarding = {
    &sysctl__net_inet_ip_children,
    { 0 }, 1, 2|((0x80000000|0x40000000)),
    &ipforwarding,
    0, "forwarding", sysctl_handle_int, "I", 0,
    "Enable IP forwarding between interfaces"
};
static void const * const __set_sysctl_set_sym_sysctl___net_inet_ip_forwarding
    __attribute__((__section__("set_" "sysctl_set")))
    __attribute__((__unused__)) = &sysctl___net_inet_ip_forwarding;
```

The definition of the first structure relies upon the address of the global `ipforwarding`. If the behaviour was as described above, this would be replaced with `global_ipforwarding[get_stack_id()]` inside the structure initialiser. This is illegal C code, function calls cannot exist in initialisers at the top level, because the initialisers are resolved at compile time. This is handled by a special case in the parser code, in this case a 0 is inserted instead of the function call. The reasoning behind this is that all indices in the global array will have the same value at program startup and the first array element is guaranteed to exist. The following is the output of the parser when the above code is used as input. This is actual source code from the file `ip_input.c`.

```
int  global_ipforwarding[NUM_STACKS] = { 0,  0,  0,  0,  0,  0,  0,  0,
    0,  0};
static struct sysctl_oid sysctl___net_inet_ip_forwarding = {
    &sysctl__net_inet_ip_children,
    { 0 }, 1, 2|((0x80000000|0x40000000)),
    &global_ipforwarding[0],
    0, "forwarding", sysctl_handle_int, "I", 0,
    "Enable IP forwarding between interfaces"
};
static void const * const __set_sysctl_set_sym_sysctl___net_inet_ip_forwarding
    __attribute__((__section__("set_" "sysctl_set")))
    __attribute__((__unused__)) = &sysctl___net_inet_ip_forwarding;
```

Testing the parser is easy, it just involves running it on the FreeBSD network stack. With 130,000 lines of un-preprocessed C code, the network stack provides ample testing input.

3.4.7 Initialisation

Section 3.2 describes initialising a single network stack. The same process can be followed for multiple stacks to some extent. The basic process of initialisation is to traverse the array of initialisation objects, calling a callback function then marking the object as done so it will not be reinitialised. However initialisation objects are not processed by the parser. These objects are put in a special section in the object file. The items in this section are sorted, iterated over and counted assuming they are in adjacent memory locations. Changing each to be an array would require a reworking of the entire initialisation section; if each individual object was an array, the sorting would need to be improved so each array did not get shuffled into other arrays. The traversal would then need to be modified so only the objects specific to the current stack would be traversed. This is possible, but there would need to be a lot of extra code to handle the extra cases. A simpler solution is to handle when an object is marked as done. For subsystems that are specific to the network stack, the object is not marked as done so the object will be visited on every call to the initialisation function. This confuses some sanity checking code which needed to be worked around: one specific initialisation routine would check the values of some variables and call `panic` if they were already initialised. This function was modified to return instead. Otherwise this approach worked smoothly.

3.5 Initial Testing

Before attempting to integrate the network simulation cradle and associated network stack with a simulator, tests need to be done to make sure it was capable of sending and receiving packets. The first testing was to send a simple ping packet, but as the project progressed further and more complex testing was needed, this was done by sending UDP and later TCP packets.

3.5.1 An ICMP Echo Request

The first test of any actual packet processing was queueing a single ICMP (Internet Control Message Protocol) echo request packet (otherwise known as a ping) in the stack. The expected behaviour here is the packet to be processed and a response created and sent out through the same interface. The fake Ethernet drivers output function should then be called with this reply packet. This test was chosen because it was the simplest available that would still test the ability to send and receive packets.

Although this is a relatively simple test, the packet goes through the major network related functions such as `ip_input` and `ip_output`. These functions are integral to the operation of the network stack, all Internet

Protocol packets pass through them. The successful completion of the test required more of the stub functions to be implemented, although, in general, these functions were quite simple.

Once this was working it was possible to queue a hand-crafted ping packet and see the network stack attempting to send a response out the attached interface. The output of running the executable at this point in the project is shown below. The executable is hard coded to receive one ping packet and abort when attempting to output a packet.

```
Copyright (c) 1992-2003 The FreeBSD Project.
Copyright (c) 1979, 1980, 1983, 1986, 1988, 1989, 1991, 1992, 1993, 1994
        The Regents of the University of California. All rights reserved.
        Timecounters tick every 10.000 msec
        IPv6 packet filtering initialized, default to accept, logging
        limited to 100 packets/entry
        ipfw2 initialized, divert disabled, rule-based forwarding enabled,
        default to accept, logging limited to 100 packets/entry by default
        BRIDGE 020214 loaded
        DUMMYNET initialized (011031)
        lo0 XXX: driver didn't initialize queue mtx
        IPsec: Initialized Security Association Processing.

        Init finished.
        Fake interface attached.
        attempting to output a packet:
        destination: 192.168.0.1:0 interface:sa
        packet: len:84
        Assertion failed: (0), function fake_ether_output, file
        support/interface.c, line 262.
```

3.5.2 UDP

Testing UDP was achieved by writing support code that would instantiate and initialise two stacks within the cradle and allow packets to be sent from one to the other. UDP is simpler than TCP because it does not require any time or state information. It is a more extensive test than ping because it requires data through the stack: when the stack was in the kernel, the data would be moving from kernel space to user space, and vice versa.

There are various system calls that allow a user level process running on FreeBSD to communicate with another process, whether on the local machine or over a network. These system calls work with file descriptors which are an abstraction of files, sockets and other input-output structures. With most of the kernel stripped from the network stack, there are no file descriptors. There is also no system call mechanism.

Because there are no file descriptors, socket creation involves kernel socket handling functions. A `struct socket` is created by the call `socreate`. From there, data can be sent by calling a callback function on the created socket. To do this two structures need to be set up: an IO vector (`struct iovec`) and a structure usually used to copy between user space and kernel space (`struct uio`). The method used to do this was taken and adapted from the system call implementations in the kernel.

One of the problems with IO is that it often involves *blocking*. Blocking is where a path of execution must wait for something to happen: usually data to arrive or leave. This means the current thread is completely halted; not an option for a discrete event simulator. For any data to arrive or leave, an event will need to be processed. Obviously, events cannot be processed if the thread is halted.

A calling thread will only be halted in the send and receive socket calls. This is done by calling the `msleep` function, normally the kernel would put the caller into a sleeping state and schedule another process to run. To implement this functionality, threads were used. A thread would act as an "application" process; attempting to send and receive data, and being blocked appropriately.

Implementing `msleep` involved blocking on a condition variable. Condition variables let threads synchronise on the value of data, they provide a notification system among threads [9]. Normally a process is woken via the `sowakeup` call, but without the process infrastructure that is present in the kernel, it is impossible to use. When data is received via the `fake_ether_input` function, the condition is signalled and woken. `msleep` would then return. The caller of msleep is responsible of checking whether it still needs to block; if it does, it calls `msleep` again. Figure 3.4 shows the design of this feature in the context of the simulation cradle.

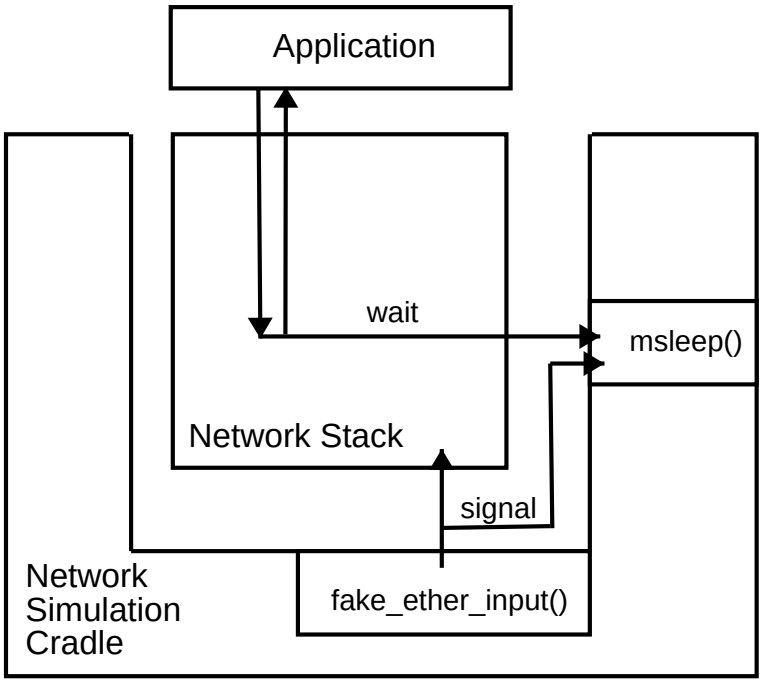

Figure 3.4: Condition variable diagram

Testing UDP involved two stacks communicating, one sending UDP packets containing some debugging information (such as “Hello World”) and the other receiving these and printing them out. Each had an application thread to do the actual reading and writing from sockets, while a simple event loop was devised to allow sending and receiving packets between each stack. At this point there is still a lot missing from the cradle; for example sending and receiving UDP requires no interaction with time at all. However, it does show that each stack is capable of communication.

3.5.3 TCP

TCP, or Transmission Control Protocol, is much more complex than UDP. It requires more functions of the cradle, for example `read_random`, a function used to create random numbers, this is called during sequence and acknowledgement number creation. Also, TCP requires around six timers per connection, these need to be created, stopped, and reset. The timers need to work in simulated time, not real time. This means there needs to be a framework in place to support such time. See section 2.1.1 for a discussion of simulated time.

It is possible to get a small amount of TCP working for testing without implementing timers at all. By adapting the framework set up to test UDP, it is possible to establish a connection with TCP. The socket function `soconnect` is called on the TCP socket and a three way handshake is begun. When the handshake is completed a packet of data is sent. This shows that TCP is basically going, though it still needed a lot of work before it would have all of its functionality.

To allow TCP to work completely, there needs to be a full simulation framework in place to support features such as timing and an application layer. It is possible to write code to allow this testing, but there is little point. All these features and more are supported by actual network simulators, it makes more sense to couple the cradle and a simulator and perform more testing once integrating the two is complete.

3.6 Simulator Integration

The goal of the *Network Simulation Cradle* project is to use real network stacks in a simulator. As such, a simulator needs to be chosen for integration. There are many options; a few are mentioned in section 2.1. One of the design goals is not to tie the choice of simulator too closely into the network simulation cradle. As such, the simulator chosen should be fairly easy to extend and provide facilities that would ease the integration of a real network stack into a simulator. Ideally this simulator should be commonly used and have TCP implementations to compare against.

Network Simulator 2, usually known as NS-2 [15], fulfils all the above criteria. It boasts varying implementations of TCP, is easily extended by implementing the `Agent` class, and even provides other useful programs such as the network simulation visualiser, Nam. L. Breslau states “NS is an ideal virtual testbed for

comparing protocols because it offers a publicly available simulator with a large protocol library." [3]

3.6.1 An NS-2 Agent

Adding protocol implementations into NS-2 is done by extending the `Agent` class. This class has several functions which need to be implemented, such as `send`, `recv` and `command`. The `command` function is the most interesting initially: it is passed a list of string arguments given in the simulation script. These can be interpreted in any way to allow the application programmer great flexibility.

Simulation scripts are written in the language OTcl, an object-oriented form of Tcl. OTcl is the language NS-2 uses to set up and manage simulations. OTcl is almost seamlessly integrated with the C++ that forms the foundation of NS-2, so referring to the new agent type created for the *Network Simulation Cradle* in a OTcl script is trivial. By implementing the `command` function mentioned above, it is easy to interact with the C++ object via a OTcl script.

An agent represents a single connection or node which can send or receive data. Hence, each agent is not a complete instantiation of a stack, but instead a socket. When creating an agent, a unique stack identifier must be assigned to the agent. This identifier is simply an index into the array of network stacks available, it should start from 0 and increase by 1 for each new stack needed. Each of these stacks also needs to be initialised once with a call to `init-stack`. Once this is done an interface needs to be added to the stack with `add-interface`. An example of OTcl code seting up an agent follows:

```
set bsd1 [new Agent/BSD]
$bsd1 set stack_id_ 1
$bsd1 init-stack
$bsd1 add-interface 192.168.7.1 255.255.255.0
```

An object, `$bsd1` is created here of type `Agent/BSD`. This is the name given to an instance of the FreeBSD network stack inside the simulation cradle. This object is then given a stack identifier of 1. This identifier is the global identifier used to uniquely identify the stack, the next stack would be assigned a number of 2, then 3 and so on. The stack then is initialised, and finally an interface is added. The interface is assigned an IP address and given a netmask.

The object `$bsd1` can then be used like any other transport agent in NS-2. For example, a traffic creation application can be attached to it such as a simulated FTP or telnet session.

Implementing the above functionality involved calling existing cradle code from the `command` function implemented in the agent. The more involved work came when implementing actual network functionality.

3.6.2 Cradle and NS-2 Interaction

Perhaps the most obvious function the simulator needs to provide to the network stack is timing. Timing changes a lot when integrating with a simulator, the clock no longer increases at a constant rate. Timing is important in TCP, which relies upon a set of timers (for example the retransmission timer) and also absolute time (for round trip time calculations). In FreeBSD absolute time is measured in *ticks*. Ticks are a unit of time that increment once every $1,000,000/hz$ microseconds. The *hz* value is set to 100 by default, which means ticks increases once every 10,000 microseconds, or 100 times a second.

The actual functions called and the basic flow of data is shown in figure 3.5. This diagram is simplified somewhat, as there can be multiple instances of both the `BSDAgent` class and the `BSDStack` class. `fake_ether_output` checks a hash table of interfaces to find the specific `BSDStack` needed, and calls the send function `if_send_packet`. This in turn calls a callback function of `BSDAgent`, which deals with the data and hands it off to NS-2. The `recv` function of `BSDAgent` is called when a packet is received at a BSDAgent node. This is synonymous with a packet being received at the physical network card. Therefore the data is eventually queued in the network stack by the fake Ethernet driver, `fake_ether_input`. Data is sent through the socket classes, which write data into the "top" of the FreeBSD network stack.

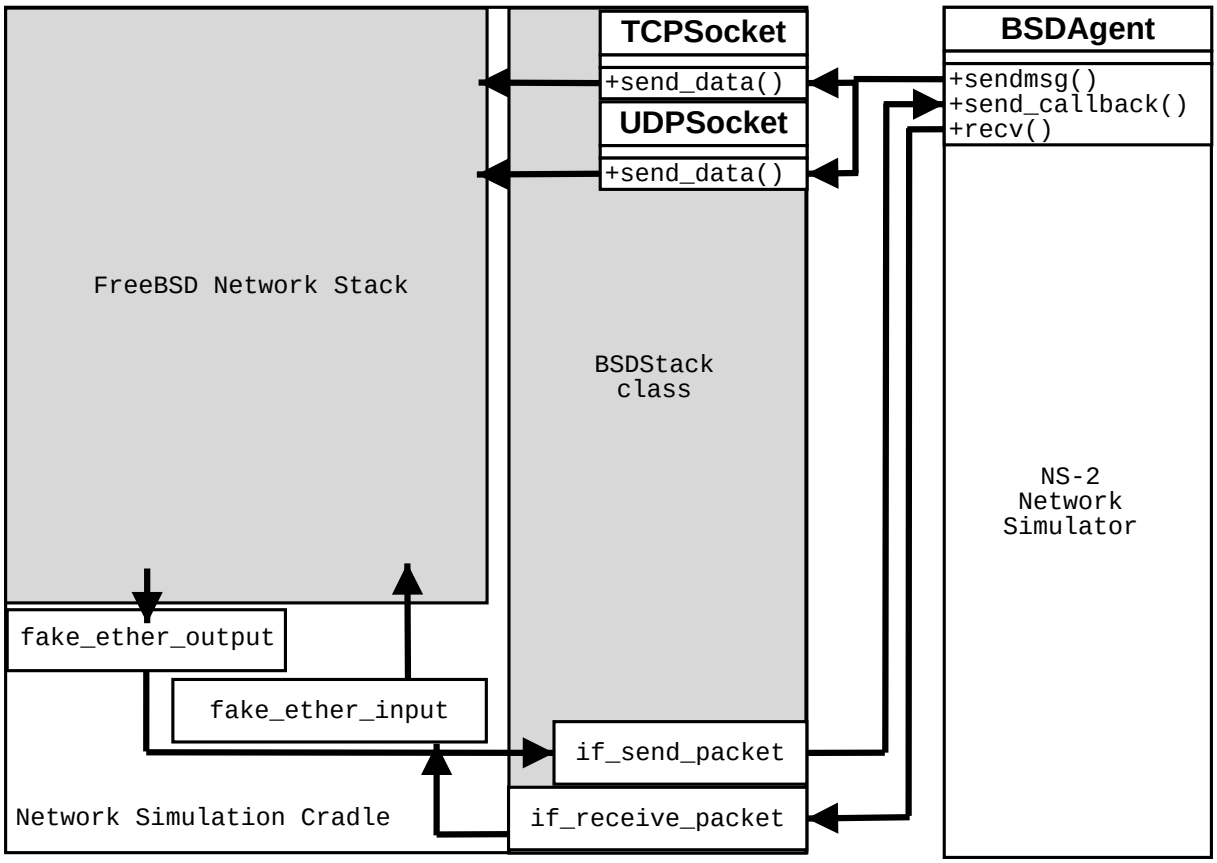

Figure 3.5: Cradle and NS-2 Interaction

Timers used by TCP are handled by the callout and timeout functions. These functions create, reset and

destroy the various timers needed for a TCP connection. The basic form of a TCP timer is to call a callback function after a specified interval. This is implemented in the NS-2 agent by extending the existing `Timer` base class. When a timer function is called in the network stack code such as `callout_reset`, the call eventually reaches the NS-2 agent, which attempts to find the timer to be reset. If it does not exist, a new timer is created, otherwise the current one is reset. The timers created are handled by NS-2, a function in the `Timer` class is called when the timer expires. This function then calls the specified network callback function.

There is one very important timer that is used frequently. This is the software interrupt timer which expires every $1,000,000/hz$ microseconds. The FreeBSD network stack checks for queued packets in the interrupt function and calls functions for protocols to process packets if any lie in wait. Without implementing this software interrupt timer no packet processing is done. This timer is implemented slightly differently to the FreeBSD kernel implementation, this time it calls a specific network stack function to do the packet processing, increases the *ticks* global variable, and reschedules the timer for the next interrupt.

NS-2 does not use actual packet data by default, in general the data sent in a packet is not important to a simulator, only the length of that data will be recorded. However, NS-2 does provide a method of sending real data in a `PacketData` class. The data is ignored by all of the NS-2 agents, such as those which do routing. This means there are some limitations here, for example the time to live (TTL) field of the packet header never gets modified. Because routing is not done by the FreeBSD stacks, each stack does not have any routes other than the route created when the interface is brought up. NS-2 provides routing transparent to the FreeBSD stacks, each node must be given an IP address in the same subnet.

NS-2 is a single-threaded application, including threads in the stack code would only complicate things. As such, threads were changed to non-blocking sockets. This requires the NS-2 agent to repeatedly poll a socket for data. Also, when sending data it is up to the NS-2 agent to deal with problems, an `ENOBUFS` error is returned if the socket buffer is full.

The stack does not exhibit its full functionality when integrated with NS-2. Currently there is no way to use a FreeBSD node as a router in the simulation; code was never written to allow this, though of course the stack itself supports being used to route packets.

Chapter 4

Results

A cradle which allowed the FreeBSD 5 network stack to run inside the network simulator NS-2 was one of the end results of this project. This chapter discusses this and other results the *Network Simulation Cradle* project produced. This chapter includes a discussion of this result in the following section, followed by an analysis of the performance of the cradle in section 4.1.1. The sacrifices neccessary to achieve a working cradle are detailed in section 4.1.2. Lastly, some initial simulations were run and are analysed in section 4.2.

4.1 The Cradle

The most obvious result of this project is a working cradle around on the FreeBSD network stack which allows it to run in the NS-2 network simulator. Though the cradle has its limitations (see section 4.1.2), its design supports the integration of network stacks. This is evidenced by the fact that there are little changes by hand required to the source of the FreeBSD network stack.

The creation of the cradle means that current researchers using NS-2 can now use a real network stack in their simulations. They can re-run previous simulations, this time using the FreeBSD network stack, with a minimum of effort. It also allows new research to be done comparing the existing protocol implementations in NS-2 against the real ones in FreeBSD. Also, the nature of the FreeBSD stack means that it is feature complete: features such as IPSec and IPv6 can all be simulated. If no more work was done on the project, in its current form it would still be a very useful extension to NS-2.

At the outset of this project it was not clear that a recent real network stack could be integrated into a discrete event simulator without extensive modification by hand. The creation of this cradle has demonstrated that such is possible, and more. It also demonstrates that such can be done with little code modification by hand, meaning future stacks can be integrated with the cradle easily.

The Alpine project (see section 2.2.1) is of interest to this project because it achieved a similar result but for a different reason: rapid protocol development. The idea behind Alpine is have the networking code runing in user space so adding and debugging code is much easier for the developer. It is possible the cradle could

be used for something similar, only with a more recent version of the network stack. Currently, the cradle cannot be invoked in the same way as Alpine. Alpine works as a shared library that overrides certain libc functionality. Adding such features would be very easy, however.

4.1.1 Performance

The performance of a simulator using a real network stack is worth investigation. The nature of the implementation of many simulators means that many shortcuts can be made, such shortcuts often mean the the code should run a lot quicker. For example, the NS-2 protocol implementations do not pass actual packet data about the simulated network, only abstractions of the data. The FreeBSD network stack uses actual data, it does a full copy of all that data whenever a packet enters or leaves the network stack.

There are two variables that are interesting with respect to speed of simulations: the amount of nodes in the simulation, and the amount of traffic that is simulated. How the speed of simulation scales to the input size is important, anything worse than a linear scaling is potentially very limiting the usefullness of the simulations that can be run. How the cradle and associated FreeBSD network stack perform in comparison to NS-2 is shown in figures 4.1 and 4.2.

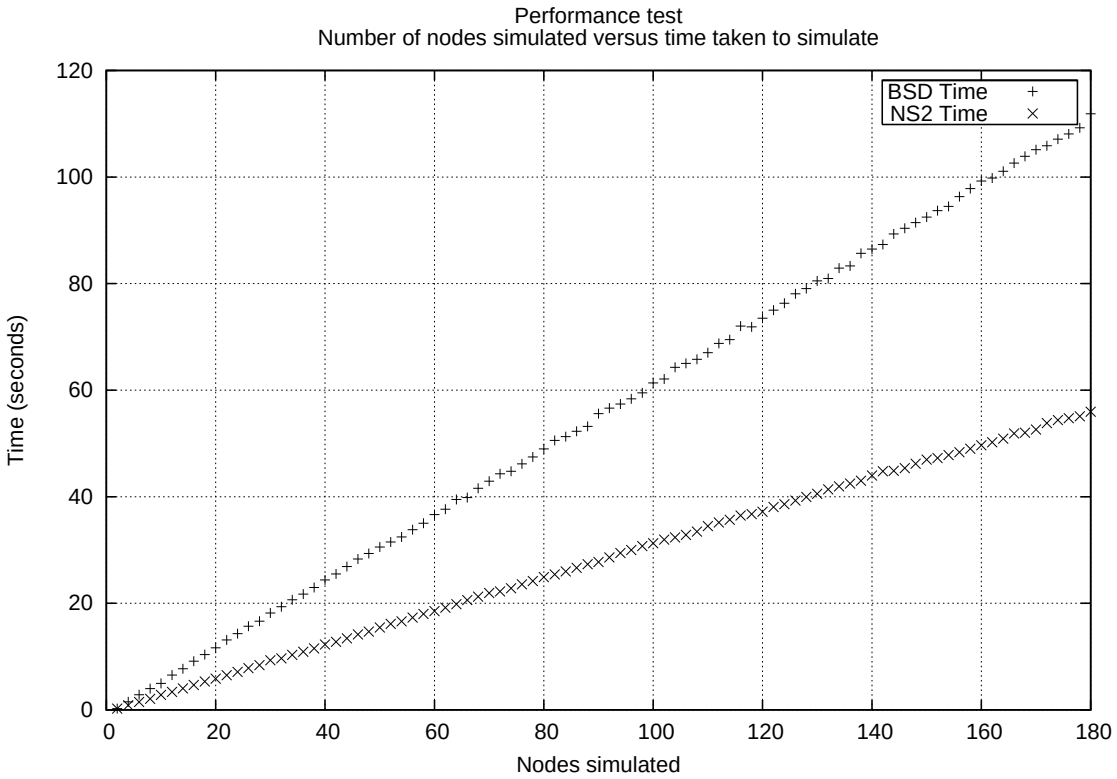

Figure 4.1: Time versus number of simulated nodes

Figure 4.1 shows a set of simulations where the number of nodes varies. Each set of two nodes forms a TCP connection. The simulation is set up to transmit traffic for 100 simulated seconds for each connection

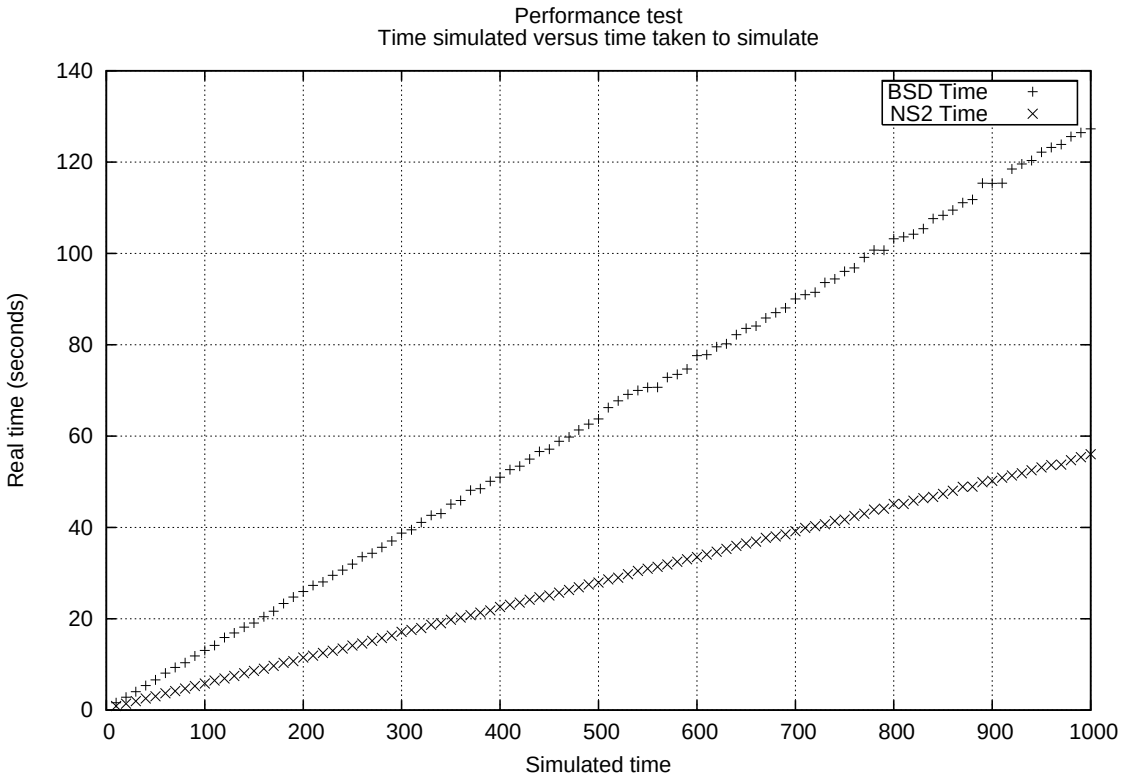

Figure 4.2: Actual time versus simulated time

before stopping. The total amount of user time, recorded using the `time` command, shows on the Y axis. This graph shows linear increases in time as the amount of nodes grows both in the case of NS-2 the cradle. The FreeBSD simulations take about 200% of the time the NS-2 simulations do. This simulation is set up such that congestion does not occur. When congestion does occur, the results vary somewhat more. Because TCP slows down in the case of congestion, not as much data is sent. When less data is sent, less data needs to be copied into and out of the FreeBSD stack, which is the probable bottleneck of the cradle.

Figure 4.2 shows almost identical results. The simulations done using NS-2s TCP implementation are consistently 230% faster than those which use FreeBSD. This graph shows a set of simulations with only one node sending a constrant stream of traffic to the other. The only variable that changes here is the duration of the transfer.

These results show that while NS-2s TCP implementation may be faster than what the *Network Simulation Cradle* offers, they both run in linear time and the percentage difference between the two is not large. The performance tests run also suggest that copying data in and out of the FreeBSD network stack is a major performance bottleneck. The entire packets do not need to be copied as they currently are, only the packet headers are important. It would be possible to reduce the amount of data copied greatly by only copying the packet headers.

4.1.2 Sacrifices Made

Making the cradle as generic as possible was a goal of this project but turned out to be very hard. Code from a kernel is both complex and unportable, this is especially true in the case of the FreeBSD network stack. Because of this, making the cradle generic enough to support other network stacks turned out to be a secondary goal. Newer FreeBSD network stacks should integrate with the cradle with a small amount of work, but currently there is little support for any other operating systems. Most of the cradle code is FreeBSD specific, this was unavoidable given the time constraints. The parser discussed in section 3.4.5 *is* written to be generic enough to cope with any C code, however, and has so far been used for the upgrade from FreeBSD 5.0 to 5.1.

At times decisions have had to be made that mean there will be extra maintenance needed when upgrading. Examples of this include the override directory used to resolve per CPU problems (section 3.2.2). These have been kept to a minimum, but they do exist. Upgrading to newer version of the stack does require some amount of work, the change from FreeBSD 5.0 to FreeBSD 5.1 took around two hours to resolve all problems. It is likely that this time will get smaller, during the upgrade the cradle only became more generic and easier to change.

4.2 Differences Between Stacks

This section discusses some simulations done to show the differences using a real network stack makes to simulation. In many cases results are quite similar, appendix A shows additional simulation results not discussed in this section.

4.2.1 TCP Performance on a Congested Link

Figure 4.4 shows the performance of a single TCP stream on a congested link. The simulation is set up as shown in figure 4.3. Nodes 0 and 3 are both TCP nodes: node 0 is attempting to send a constant 1 Mb to node 3. Nodes 4 and 5 create congestion. Node 4 sends some amount of UDP data to node 5. As the amount of this traffic increases congestion will occur on the link between node 1 and node 2. This link was generally limited to 2Mb/s, but the amount was specific to the simulation being run. Each graph notes the bandwidth and latency of this link.

Each point on the graph in figure 4.4 is a simulation. Each of these points is the average traffic rate over the entire simulation. The expected result is for the UDP data to take up a majority of the bandwidth and for the router to eventually discard packets once congestion occurs. The queing algorithm on the router comes into play at this point, namely Random Early Discard in this case. This results in slightly curved trends in the TCP performance once the link is congested as shown in figure 4.4. Note the differences between the NS-2

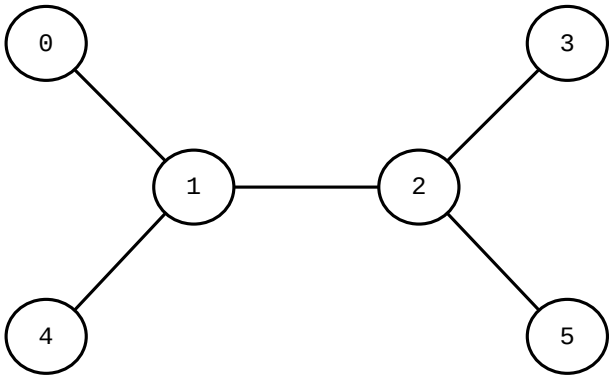

Figure 4.3: Simulation setup

TCP performance and that shown by the FreeBSD TCP implementation. They follow each other quite closely in the graphs but show some difference once the link becomes congested. NS-2 shows a fairly idealised curve that FreeBSD does not always follow, this is especially evident between 1 and 1.2Mbit/s presented UDP load.

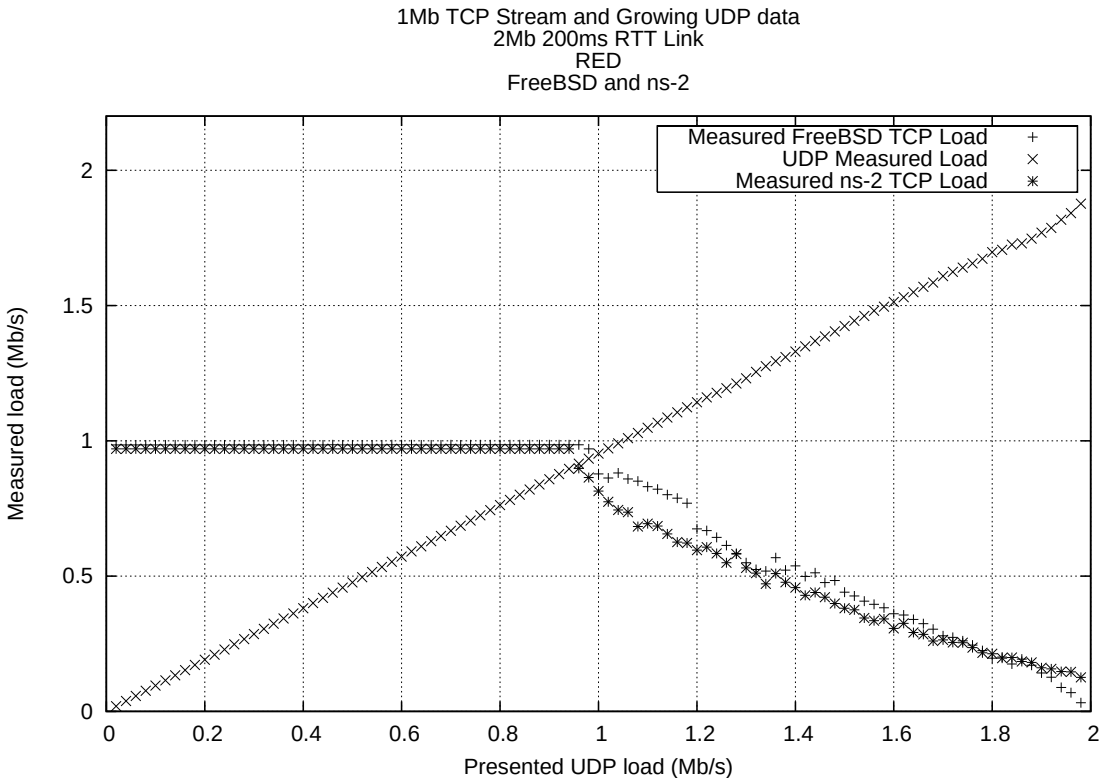

Figure 4.4: TCP Performance: RED

Figure 4.4 can be compared with figure 4.5 to show the difference RED makes over drop tail as router policies in such a simple situation. That difference is not what is interesting here, however. Figure 4.5 shows a noticeable difference in the graphs between 1.3Mb/s and 1.7Mb/s on the x-axis. There is quite a difference between the NS-2 and FreeBSD TCP implementations here. At this point the reason for these differences is not clear, but again, the differences between the FreeBSD and NS-2 stacks indicate that using a real world

stack is valuable.

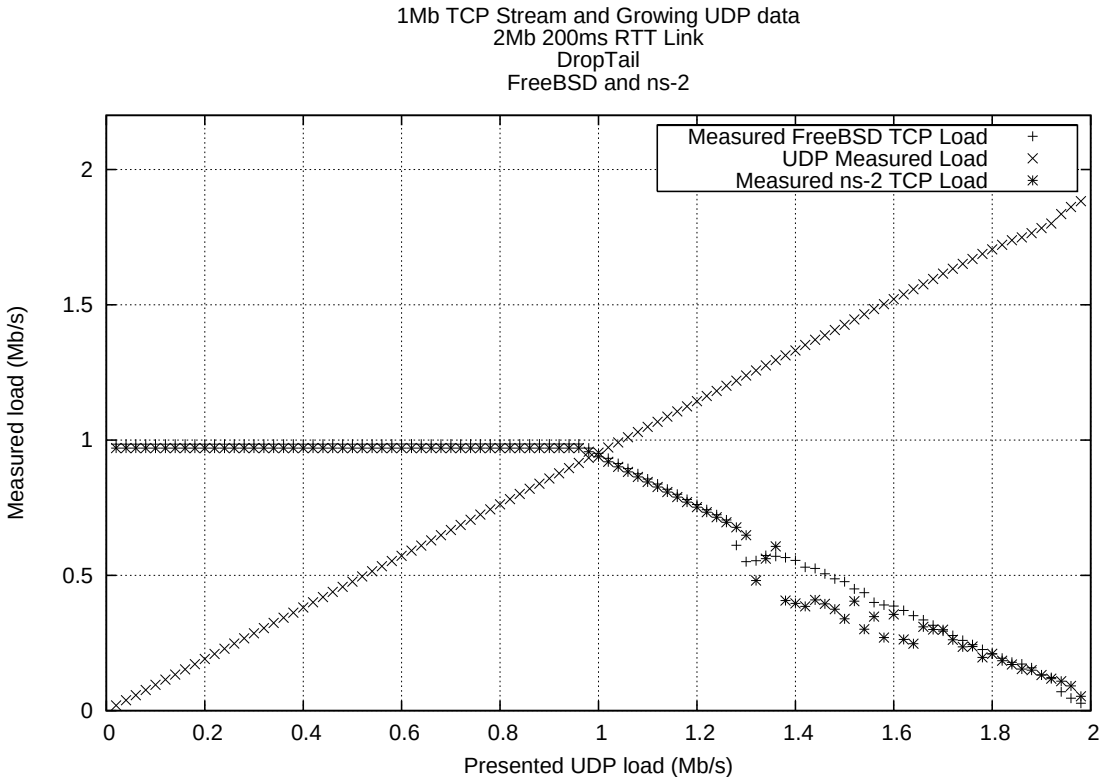

Figure 4.5: TCP Performance: Drop Tail

Larger differences between NS-2 and FreeBSD were found when combining two TCP streams and attempting to write a large amount of data, enough to overfill the pipe. At this point TCP congestion control comes into play, this time between two TCP streams, rather than one stream attempting to cope with a UDP stream. There is one other important change to this set of simulations, the latencies are different for the two streams. So though both TCP streams go through the same congested link, the latencies for each is different. For the constant 1Mb/s stream the latency is 40ms (therefore a 80ms round trip time). For the growing stream the latency is 100ms. This difference was created to change another variable that effects TCP in an attempt to find a difference between the two implementations.

This difference in latencies means that the TCP streams should not use 50% of the bandwidth each, the stream with a lower latency should use more of the available bandwidth than the other. Both NS-2 and FreeBSD show this property, but the amount of change differs. The difference is obvious in figure 4.6; it is around 5% for each of the two streams.

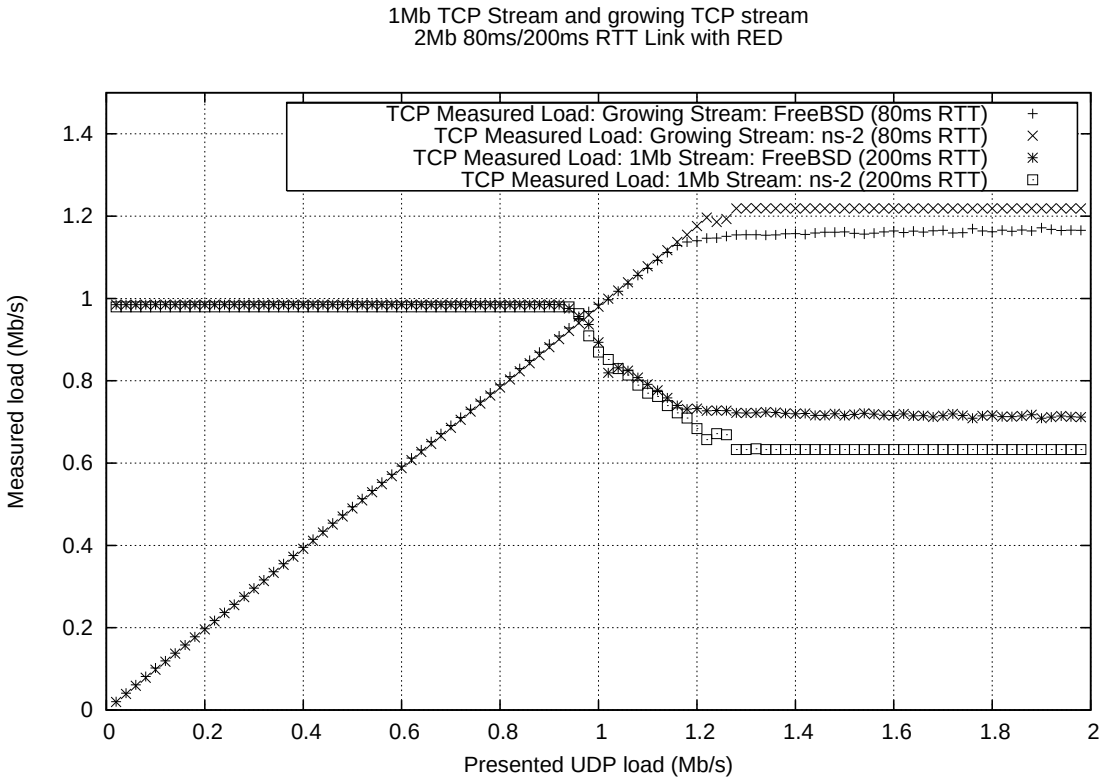

Figure 4.6: TCP Congestion Control With Differing Latency

Chapter 5

Conclusions and Future Work

The *Network Simulation Cradle* project set out to create a framework which allowed real network stacks be used for simulation. The cradle allows a current FreeBSD network stack to be used in the popular network simulator NS-2, while its design is generic enough to cope with future versions of the FreeBSDs network stack. Its creation demonstrates the correctness of the fundamental goal of the project, a cradle which allows a real world network stack to be used for simulation.

A generic cradle has not been fully realised at this point. It is possible that this will be much more difficult to achieve than was originally envisaged.

Preliminary simulations run show some different results comparing the NS-2 TCP implementation against the FreeBSD implementation. These simulations show some differences in performance between the two, but it is too early to draw any concrete conclusions about the differences, but indications are that these differences are important.

The *Network Simulation Cradle* is currently a feature full target for simulation with NS-2, it allows the use of a real network stack within a simulated environment without the usual problems encountered in network emulation.

Future work involves looking at the differences encountered in simulation. The first step necessary in evaluating this behaviour would be validating the FreeBSD stack used. This would involve network emulation, a situation would need to be set up where FreeBSD computers were configured to do the same things the simulation nodes were configured to do in the results shown above. Results from this emulation would need to be recorded and compared against the simulation results.

Assuming the emulation produced favourable results, specific results would need to be analysed for differences. This would involve looking at the packet traces produced by the simulation and going through them by hand to figure out why such differences are occurring between NS-2 and FreeBSD. The simulator visualisation tool Nam is also useful here, it could possibly help highlight any difference.

Other work to be done in the future includes porting other network stacks to use the *Network Simulation Cradle*. Linux is the obvious candidate here, there are no other obvious candidates. Including future versions

of the FreeBSD stack in the cradle is also part of the future work that could be done on this project. Improving performance can likely be done with the current cradle, investigating copying less data in and out of the stack is another area of work. More extensive performance testing and more simulations should also be run in the future.

Appendix A

Simulation Results

Not all simulations were discussed in the main report, other simulation results are presented in this appendix. The results discussed here are for completeness, they do not show the variations between FreeBSD and NS-2 as the graphs discussed in section 4.2 do.

A.1 TCP Congestion Control

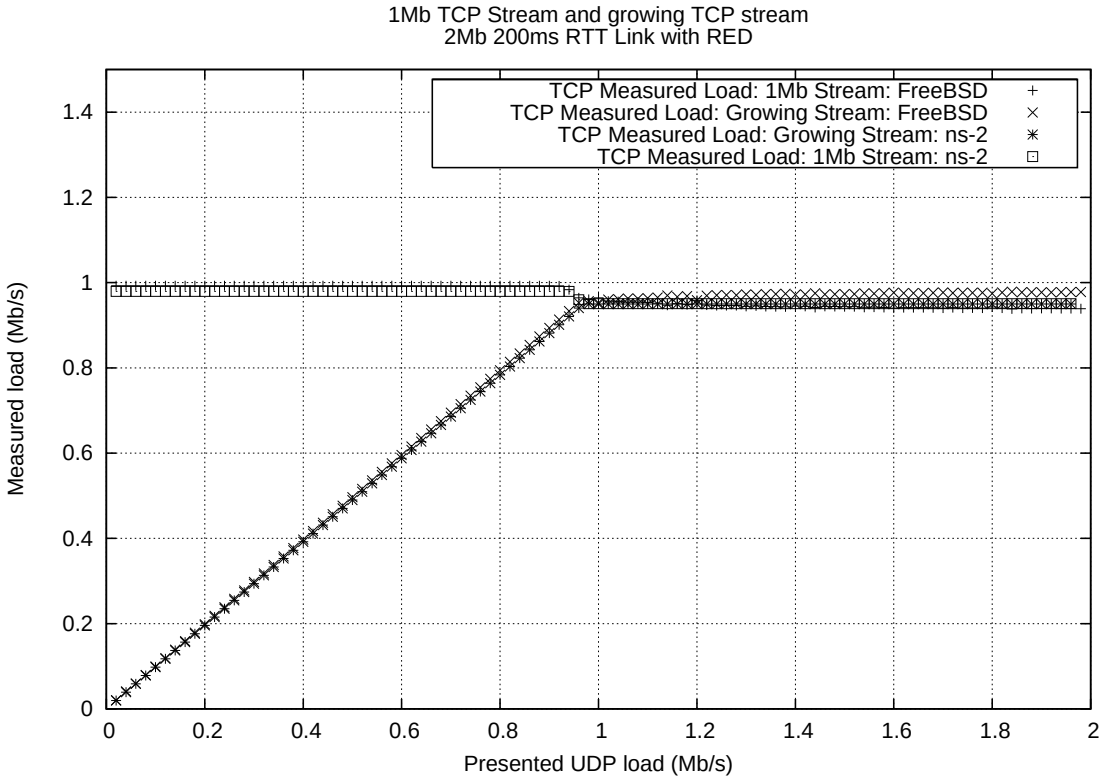

Figure A.1: TCP Congestion Control

Figure A.1 shows a set of simulations performed to test TCP congestion control. The simulation setup here includes two TCP streams. One attempts to send a constant 1Mb/s, while the amount is varied for the

second stream. The simulation involves a 2Mb link with 100ms latency, while the routers use RED. The graph results are as expected: as the link between the senders and receivers is only 2Mb, each stream uses roughly half the link bandwidth. Both NS-2 and FreeBSD show the same results.

A.2 TCP Congestion Control With Different Latency

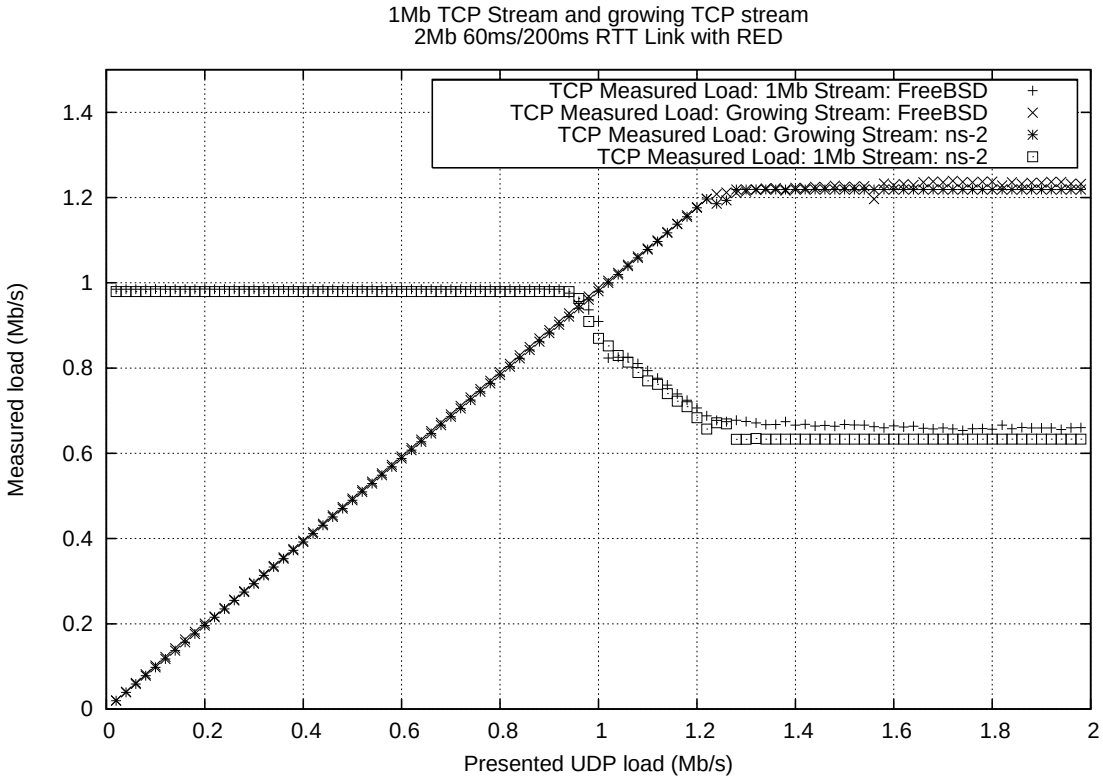

Figure A.2: TCP Congestion Control With Different Latency

By changing the latencies involved, it is possible for TCPs congestion control to share the link unequally between the streams. In this simulation the link bandwidth is 2Mb, while the latencies encountered are 30ms and 100ms with a router using RED. Figure A.2 is a graph showing a set of simulations which illustrate this particular facet of TCP.

Appendix B

Memory Leak Debugger

The result of combining the FreeBSD network stack and NS-2 was an enormous amount of code, a combination of over 400,000 lines of C, C++ and OTcl, only the tiniest percentage of the final product was cradle specific code. Memory leaks were a continuing problem when developing the cradle code to integrate the two. Tracing memory leaks that occur somewhere in one of the large sections of other peoples code is extremely hard, so memory debuggers were investigated.

Very few memory debuggers exist that work without any code modification needed. Of those that did, the most complete found was Valgrind. This debugger only works for Linux, though, where the cradle only compiles on FreeBSD. Another common memory debugger in Electric Fence, but this was unable to cope with the large project and ended up stopping with cryptic error messages.

To solve this problem, a simple memory debugger was built. The requirements were to report memory leaks without any code needing to be changed, it is not realistic to go through every source file in the project to add a header in or change how memory allocation routines are called. Also, the debugger should work on FreeBSD and also function properly with an executable of the size faced in this project.

It is possible to override the calls to the libc functions `malloc` and `free` by creating a shared library with these functions defined and putting that shared library into the `LD_PRELOAD` environment variable. This method allows a simple memory debugger to be written in a shared library and this library can be linked at run time without any code modification in the original project.

Whenever memory is allocated with `malloc`, the a record of the allocated memory is prepended to a linked list, as well as the actual memory being returned. When memory is freed with `free`, the memory is removed from the list. Because allocations are added at the start of the list, and memory allocations are most often short lived, the lookup time when freeing memory is quite small. When the program finishes, the remaining elements of the list can be printed out for later analysis.

The information printed out includes the last three functions on the call stack when the memory was allocated, the address of the memory allocated, and the size of the block. The functions are printed out only as

addresses, these addresses are obtained by the use of the gcc-specific builtin function, `__builtin_return_address`, which returns the address of a particular function on the call stack. The function addresses can be transformed into useful information by using the `addr2line` GNU utility.

Overriding `malloc` means that the shared library, also, does not have access to the libc version of `malloc`. The libc functions need to be called to do the actual memory allocation and freeing. To get access to the functions, `/usr/lib/libc.so` is dynamically loaded using the `dlopen` function. The real `malloc` and free functions are then loaded using `dlfunc`.

The basic output is fairly unreadable and looks like the following:

```
0x8208823 0x81b3c86 0x81f828a : 32
0x8224f3b 0x820b97c 0x820b94c : 128
0x8224f3b 0x821834e 0x8268045 : 448
0x8224f3b 0x8219934 0x82316fe : 32
0x8224f3b 0x821f725 0x826b2ab : 108
0x8224f3b 0x822360b 0x826ba14 : 512
0x8224f3b 0x822502f 0x8276fc6 : 256
```

A Python script written by Perry Lorier cleans this output up and uses `addr2line` to obtain symbol names and such, the formatted output looks like:

```
TclpAlloc(??:0) Tcl_Alloc(??:0) NewVar(??:0) : 32
new_malloc(support/malloc.c:38) cblock_alloc_cblocks(kern/tty_subr.c:135)
    clist_init(kern/tty_subr.c:89) : 128
new_malloc(support/malloc.c:38) nd6_ifattach(netinet6/nd6.c:172)
    in6_update_ifa(netinet6/in6.c:1139) : 448
new_malloc(support/malloc.c:38) nd6_rtrequest(netinet6/nd6.c:1218)
    rtrequest1(net/route.c:739) : 32
new_malloc(support/malloc.c:38) nd6_prelist_add(netinet6/nd6_rtr.c:826)
    in6_ifattach_linklocal(netinet6/in6_ifattach.c:563) : 108
new_malloc(support/malloc.c:38) scope6_ifattach(netinet6/scope6.c:78)
    in6_ifattach(netinet6/in6_ifattach.c:801) : 512
new_malloc(support/malloc.c:38) m_get(support/mbuf.c:39)
    igmp_init(netinet/igmp.parsed.c:4774) : 256
```

This shows the three functions from the call stack, along with their source file and line number information if it exists, a colon, then the size of the block allocated.

This memory debugger was used to find several major memory leaks in the cradle code.

Bibliography

[1] Alpine. http://alpine.cs.washington.edu/, Accessed 2003.

[2] Lawrence S. Brakmo and Larry L. Peterson. Experiences with network simulation. In *Measurement and Modeling of Computer Systems*, pages 80–90, 1996.

[3] Lee Breslau, Deborah Estrin, Kevin Fall, Sally Floyd, John Heidemann, Ahmed Helmy, Polly Huang, Steven McCanne, Kannan Varadhan, Ya Xu, and Haobo Yu. Advances in network simulation. *Computer*, 33(5):59–67, 2000.

[4] David Ely, Stefan Savage, and David Wetherall. Alpine: A User-Level infrastructure for network protocol development. In *3rd USENIX Symposium on Internet Technologies and Systems*, pages 171–184, 2001.

[5] Freebsd handbook. http://www.nz.freebsd.org/doc/en_US.ISO8859-1/books/handbook/index.html, Accessed 2003.

[6] Ian Graham, Murray Pearson, Jed Martens, and Stephen Donnelly. Dag - a cell capture board for atm measurement systems. Technical report, 1997.

[7] Daniel Lawson. Validating the simulator. Directed study, Department of Computer Science, The University of Waikato, Hamilton, New Zealand, 2002.

[8] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and Josn S. Quarterman. *The Design and Implementation of the 4.4BSD Operating System*. Addison-Wesley, 1996.

[9] Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell. *Pthreads Programming*. O'Reilly, 101 Morris Street, Sebastopol, CA 95472, 1998.

[10] Nist net. http://snad.ncsl.nist.gov/itg/nistnet/, Accessed 2003.

[11] Oxford english dictionary online. http://dictionary.oed.com, Accessed 2003.

[12] Posix threads programming. http://www.llnl.gov/computing/tutorials/workshops/workshop/pthreads/-MAIN.html#WhyPthreads, Accessed 2003.

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

[14] Scalable simulation framework. http://www.ssfnet.org/, Accessed 2003.

[15] The network simulator - ns-2. http://www.isi.edu/nsnam/ns/, Accessed 2003.

[16] The real network simulator. http://minnie.tuhs.org/REAL/, Accessed 2003.

[17] Andras Varga. Using the omnet++ discrete event simulation system in education. *IEEE Transactions on Education*, 42(4), 1999.

[18] Visualisation of compiler graphs. http://rw4.cs.uni-sb.de/users/sander/html/gsvcg1.html, Accessed 2003.

[19] S. Y. Wang, C. L. Chou, C. H. Huang, C. C. Hwang, Z. M. Yang, C. C. Chiou, and C. C. Lin. The design and implementation of the nctuns 1.0 network simulator. *Computer Networks: The International Journal of Computer and Telecommunications Networking*, 42(2):175–197, 2003.

[20] Marko Zec. Implementing a clonable network stack in the freebsd kernel, 2003.