Proceedings of the 2005 Winter Simulation Conference
M. E. Kuhl, N. M. Steiger, F. B. Armstrong, and J. A. Joines, eds.

SIMULATION WITH REAL WORLD NETWORK STACKS

Sam Jansen
Anthony McGregor

Gate 1 Knighton Road
University of Waikato
Hamilton, NEW ZEALAND

ABSTRACT

Network simulation is used widely in network research to test new protocols, modifications to existing protocols and new ideas. The tool used in many cases is ns-2. The nature of the ns-2 protocols means that they are often based on theoretical models that might not behave in the same way as real networks. This paper presents the Network Simulation Cradle which allows real world network stacks to be used in a wrapper that allows the stacks protocols to be used in the ns-2 network simulator. The network stacks from the open source operating systems Linux, FreeBSD and OpenBSD are included in the simulation cradle as well as a stack designed for embedded systems, lwIP. Our results show that ns-2's TCP implementations do not match observed behaviour from real machines in some respects and using the Network Simulation Cradle produces results closer to real world network stacks.

1 INTRODUCTION

Testing, developing and evaluating network protocols is often done with the help of network simulators. Simulation is used because it allows experiments to be undertaken even if the network hardware is not available or physically cannot be constructed in the real world. For example there is practically no limit to the bandwidth and delay which can be specified for a simulated link. As networks continue to grow more complex, the need for network simulation increases. To predict the expected performance of complex networks and to understand the interactions of protocols, network designers and researchers use simulation.

The network simulator ns-2 is widely used in the network research field and has had its models validated by its creators (Information Science Institute (ISI) 2004). However the models of protocols used by ns-2, specifically Transmission Control Protocol (TCP), are heavily abstracted. Heidemann et al. (2001) report that the one-way TCP models included in ns-2 model have a simplified protocol supporting unidirectional data transfer without message fragmentation and do not attempt to model any particular TCP implementation or specification. These models suffice for many situations but do not reflect how real TCP implementations perform.

To illustrate the difference between real TCP implementations and ns-2 TCP models, consider table 1 which shows the goodput recorded for varying operating systems running on a test network, compared to the same situation simulated with ns-2 TCP agents. The data was collected by running throughput tests on an isolated test network which used FreeBSD Dummynet (Rizzo 1997) to limit bandwidth and introduce loss. A simple dumbbell topology is used with uniform random packet loss of 5% set on one of the bottleneck routers. The bottleneck link has a latency of 100ms and a bandwidth of 2Mb/s. The queue size of both routers is set to 10 packets. All tests were run 100 times. The mean goodput is displayed in the table. *Goodput* is the amount of data successfully read from a socket at the end of a TCP connection.

The wide variation of goodputs recorded on the test network shows how real world TCP implementations differ under the same circumstances. All the machines tested had delayed acknowledgements turned on. FreeBSD should be compared to the Newreno agent with delayed acknowledgements: ns-2 gets 60% of the goodput recorded with FreeBSD 5.2.1 machines. Both Linux and OpenBSD use TCP selective acknowledgements (SACK) and delayed acknowledgements and both get very different goodput. The measured goodput for comparible ns-2 TCP implementations are 43% of Linux's goodput and 79% of OpenBSD's. Further discussion of this experiment can be found in section 4.

Table 1 shows that while ns-2 stack models produce results that are close to the same range as actual stacks they do not match particular stacks closely. If accurate simulations matching real world stacks is required, new models are necessary.

Table 1: Mean Goodput with 5% Bidirectional Loss

TCP Implementation	Goodput (kb/s)
ns-2: Sack1/Sack1-DelAck	88.9
Linux 2.4.27	208.6
ns-2: Newreno/DelAck	96.9
Linux 2.4.27 No SACK	193.7
FreeBSD 5.2.1	162.8
OpenBSD 3.5	113.2

This paper addresses the need for more accurate network simulation. Abstraction of protocol implementations are used in network simulators for a few reasons. Efficiency is important. Writing complex models of protocols is time consuming and prone to error. Even with a thorough implementation, validation of the model is still required. Each real world network stack exhibits differing behaviour as the results in table 1 show. Manually writing code to model each stack in simulation, possibly each version of each stack, is not feasible.

We present a way to use the protocol implementations from open source operating systems in ns-2. Although we present our approach for ns-2, it is largely independent of the simulator and should be able to be integrated into other network simulators without any changes to the core system. The network stacks of the operating systems Linux 2.4.26 to 2.4.28, FreeBSD 5.1 to FreeBSD 5.3 and OpenBSD 3.5 are included in what is known as the *Network Simulation Cradle*. The embedded TCP/IP stack lwIP (Dunkels et al. 2004) is also included. The Network Simulation Cradle, or NSC, includes code for each stack to support the stack outside of its native environment or operating system. To ease integration and reduce error, the only modifications made to the stack are done programmatically.

This approach has several benefits. NSC can be used for its TCP implementations that perform similar to real machines. The NSC network stacks are a near drop-in replacement for the current TCP implementations in ns-2. For many simulations it is possible to substitute the original ns-2 TCP agent with an NSC TCP agent without any other changes required. The extra functionality available in a full TCP implementation can be used; for example simulations such as GnutellaSim (He et al. 2003) require actual data sent over the TCP connection which is not available in the TCP models present in ns-2. ns-2's TCP models also do not include some basic TCP features such as a dynamic receivers advertised window.

Validation of TCP using NSC is covered in section 4. An overview of the architecture and design of NSC is described in section 3. CPU and memory performace is dicussed in section 5. Approaches using real world network code in simulation and in capacities similar to this project are discussed in the next section.

2 RELATED WORK

We use real world network code in simulation to increase accuracy. Other approaches exist that use real network stacks. Alpine (Ely et al. 2001) and ENTRAPID (Huang et al. 1999) use BSD network stacks in user space to facilitate protocol development. The BSD Network Stack Virtualization (Zec and Mikuc 2003) project allows high-bandwidth real-time simulation. NCTUns (Wang et al. 2003) allows real network stacks to be used for simulation. Bless and Doll (2004) integrate the FreeBSD network stack into the simulator OMNET++. These projects are discussed in the following sections.

2.1 ENTRAPID

ENTRAPID (Huang et al. 1999) is a protocol development environment designed to provide some of the features of general-purpose network simulation. ENTRAPID is a process running in user space supporting multiple virtualised networking kernels. This is a set of network stacks that are managed to provide an abstraction of a real-time simulator. Each stack is modified 4.4 BSD network code.

Modifying the network stacks to allow multiple stacks to run in the same user space process independently is performed by hand modifications to the stack. All non-local references are found and mapped through a indirection table to an appropriate shared resource.

ENTRAPID is a powerful user-space development environment but suffers from the problem of keeping it up to date, due to the extensive hand modifications necessary for virtualisation. ENTRAPID is now part of a commercial project and is not publicly available.

2.2 Alpine

Alpine (Ely et al. 2001) is aimed at protocol development in user space like ENTRAPID, but is a simpler system. The Alpine project moved the FreeBSD 3.3 network stack into user space with few hand modifications and created a compatibility layer that allowed the stack to be used as a user-space network stack providing the traditional BSD sockets API. The process Alpine used to move the network stack to user space is very similar to that used in this project. It does not provide any way for multiple instances of the stack to run concurrently. Alpine does not provide any way to be used with simulation, it is only used for protocol development. Alpine has not been updated from FreeBSD 3.3.

2.3 BSD Network Stack Virtualization

Zec (2003) modified the FreeBSD 4.7 operating system kernel to allow multiple network stacks to run alongside

each other on the same system. This work was then extended to allow simulation to use the virtual network stacks. Zec and Mikuc (2003) present a real-time network simulator capable of operating at gigabit speeds. Their work allows applications to run unmodified and interact with a particular network stack instance correctly. They also modify the kernel so each virtual network stack can be limited in CPU usage to stop runaway processes from starving the system of resources.

The host system is able to be configured in such a way that packets that enter from an outside network are routed through the simulated network in any way. This means using the existing FreeBSD emulation capabilities of Dummynet (Rizzo 1997) is possible. The simulator can be combined with a real network seamlessly.

Zec and Mikuc do not address the possibility of adding error into the network stack during their extensive modifications. Nor do they validate the system, though they do note that testing is required.

Detailed statistics gathering with this project is hard. Obtaining information other than goodput over time generally involves modifying the kernel and installing the new kernel. This is not specific to the virtualisation simulator, most real time simulators or emulators face this same problem.

2.4 NCTUns

NCTUns 1.0 (Wang et al. 2003) is a simulator which attempts to make use of a real world network stack for simulation, much like this project. NCTUns uses the local machines network stack via a tunnel network interface. 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 the normal TCP/IP processing. When a packet is read from the tunnel device, the first packet in the tunnel interfaces output queue is copied to the reading application.

One of the advantages of this approach is that it allows real-life UNIX application programs to run on simulated nodes in the network because the system default UNIX POSIX API is available.

However, NCTUns has some disadvantages. First, it needs kernel modifications for all machines it runs on. The kernel needs to be patched to support changes to timing, the scheduler, and other facilities. This has three major ramifications: hand changes to the protocol code means that results produced are less convincing, as it is hard to know whether these changes will affect results. To use NCTUns, the user needs full administrative privileges to install the new patched kernel, which is not always an option, especially in a student laboratory setting where access may be restricted. The code also needs to be maintained for all operating systems it runs on. Statistics gathering faces the same problems as detailed in section 2.3.

A separate computer is needed for every different version of every operating system that is to be simulated. While simulating, computers cannot be used for other activities for fear of affecting the simulation results. This means larger simulations could require many machines; the resource requirements are higher than a simulation run in ns-2 would be.

2.5 OMNET++ and FreeBSD

Bless and Doll (2004) describe integration of the FreeBSD TCP/IP stack into the simulator OMNET++ (Varga 1999). Their reasons for using a real world TCP/IP stack is that they wished to avoid "possible implementation errors and costly validation tests" and OMNET++ lacked a validated TCP model.

Bless and Doll (2004) describe how the many timer events generated by a real stack are a performance bottleneck and propose a solution to solve this problem. They also manage the routing table in the FreeBSD stack and allow routing packets. Neither the timer performance changes or the routing table updates are implemented in NSC.

To allow multiple instances of the FreeBSD TCP/IP stack to work in OMNET++, the global variables in the code were changed by hand. The authors found that a simple search and replace was not enough to handle the complexities of modifying global variables. They conclude a Perl script to modify the source programmatically is an area of further research and modify the global variables and their references by hand. Further discussion of the problem of multiple instances, and our general solution to solve it, can be found in section 3.3.

Manually changing source code means that keeping this project up to date with future FreeBSD releases is a time consuimg process. The project integrates FreeBSD 4.9 which is already a major version behind the current stable version of FreeBSD in 2005. Also, the authors only perform a minimal amount of validation because "we did not modify the TCP code of FreeBSD, so we did not have to test all potential error cases." While it is true that the vast amount of validation that should go into an independant TCP model is not necessary, it is our experience that there is a possibility of many bugs even when the network stack code does not appear to be modified. The manual approach used to modify global variables and their references means there is room for human error; a variable or a reference to a variable might not be changed correctly or may be missed altogether, resulting in unforseen and possibly hard to predict behaviour. We found many of these bugs before we had finished the global parser (see section 3.3).

The FreeBSD/OMNET++ project did not investigate using different real world network stacks in simulation. The other projects discussed in this section save NCTUns were the same in this regard: the approach was specific to one network stack or operating system. This paper presents a general way of using real world network stacks in simulation and validates the existing stacks. The architecture and design of our approach is described next, in section 3.

3 ARCHITECTURE AND DESIGN

NSC is designed as two distinct objects that communicate through a well defined interface. There is an ns-2 *agent* implementing an ns-2 API which forms the transport protocol in the simulator: this means the agent will be connected to another agent and instructed to send data. The agent is responsible for instantising and interacting with the other part of NSC, the shared library. The shared library contains the network stack in question as well as supporting code. NSC also has one other component which is used during the build process. The global parser programmatically changes references to global variables.

A high level diagram of ns-2 with NSC appears in Figure 1. Further details of the components in the diagram and the global parser are described in the following sections.

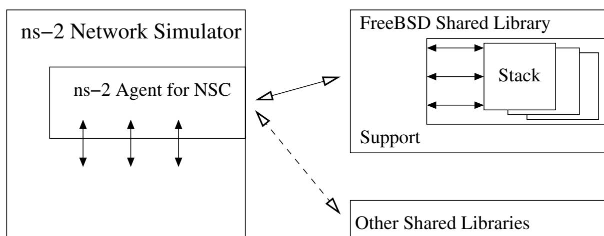

Figure 1: Interaction with Multiple Network Stacks

3.1 Simulator Integration: an ns-2 Agent

The ns-2 agent first loads the correct shared library containing the requested network stack and initialises it. The agent is responsible for routing messages between the simulator and the network stack that resides in a shared library. The simulation script and simulated application perform actions that require communicating with the network stack. The network stack will communicate with the agent by informing the agent to send packets or set timers.

Existing TCP agents in ns-2 model a single TCP connection. They implicitly connect when the first data is written. No actual data is sent over the simulated connection; only a number of bytes is specified in each application send.

This simple model means an underlying network stack needs special management. The interface to the network stacks allows creation of an arbitrary number of sockets which perform the usual socket operations: connect, listen, accept, write, read and close. To work the same way as the existing ns-2 agents the NSC agent initially creates one socket to listen on. Another is created to connect to a remote host if the agent is ever instructed to connect. Whenever an attempt is made to read data, an attempt is made to accept a connection from the listening socket. If this succeeds, another socket is created that will be used for reading data in the future.

An API is exposed to scripts that allows setting specific IP addresses and netmasks and listening on specific ports. However, to keep maximum compatibility with existing scripts, the NSC agent can be created such that IP addresses are automatically allocated based on the ns-2 node identifier. Port numbers are fixed and connection is triggered when the first data is sent.

3.2 The Shared Library

The shared library contains the network stack along with supporting code that implements the interface necessary to communicate with the simulator. Shared libraries must be used instead of static libraries because shared libraries provide namespaces for the symbols contained within. For example, FreeBSD and OpenBSD have a routine called `tcp_input` used during TCP processing. If the network stacks were statically linked into the simulator, this symbol name along with others would clash and cause errors.

The shared library is made of three parts. The C++ simulator interface is required to communicate with the simulator. This is a C++ class that is created via the only function exported to the simulator; a `create_stack` function. The network stack itself is contained in the shared library. Support routines are also created to expose functionality to the C++ interface.

There is an important separation of the C++ interface and the support routines. The support routines only include headers from the network stack and nothing from the system C library. This is required because when compiling the FreeBSD network stack on Linux, for example, there are clashes between the Linux C library include files and the include files in the FreeBSD kernel.

There is a shared library for each different stack. Each shared library has its own set of code to create and manage sockets. To interact with a network stack, applications generally communicate through the BSD sockets API. This is not available at a kernel level, so a sockets API needs to be implemented on a per-stack basis. The socket operations all need to be non-blocking because the simulator is completely single threaded. This has not been a problem in practice as every protocol implementation in kernel space encountered does not require blocking.

3.3 The Global Parser

Using real world network stacks means that handling multiple instances of each network stack is potentially a large hurdle. Network stacks are designed to be used on their own, there is normally no allowance for multiple instances of network stacks. An exception to this rule is the FreeBSD Network Stack Virtualization project (Zec 2003), but this is not part of core FreeBSD; it is distributed as kernel patches.

The need to support multiple network stacks can be solved in two ways. Forking the process or loading a completely new shared library for each network stack is one solution. Another is to change global variables so each network stack has it's own copy of them. The first solution is simple to implement and does not require changing existing source code, meaning the chance of inadvertently adding an error is small. It suffers from a scalability problem however, as forking or loading shared libraries has a large memory and CPU cost. Changing global variables is more efficient but also more error prone. Doing so by hand is a large amount of work and requires significant effort to keep the network stack up to date.

The FreeBSD Network Stack Virtualization project solves the issue by manually changing the network stack to aggregate the global variables into one "virtualization" structure. A pointer to this structure can then be passed through the kernel as a function parameter which specifies which network stack is active. The method which it took is neither robust nor maintainable enough for NSC.

It is possible to change global variable definitions and references programmatically. Because network stacks are written in C code, some filter which runs over the C code and understands the global variables and how to change them is possible. Originally attempts at multiple search and replaces or creating Perl scripts failed because of the many complexities that arise when programmatically analysing C source code. A program that uses the compiler-compiler tools Bison and Flex was created for use in NSC. Referred to as the "global parser", the program understands C well enough to replace global variables and global variable definitions correctly while leaving the surrounding code the same. It understands local variables shadowing global ones as well as static local variables.

The global parser reads in a list of variables to change on startup. This means there must be some way of selecting which global variables need to be replaced without missing any which are important. The global parser solves this problem by having a mode of operation where it outputs every global and static local variable encountered. It is then possible to go over the list and manually select the symbols needed.

4 VALIDATION

The amount of code developed for the Network Simulation Cradle is small. One of the reasons for this was to minimise the chance of inadvertently making the network stack perform abnormally. This section presents a comparison of traces produced from simulation and from a test network. Because NSC produces traces from a real network stack, it is possible to compare traces between simulation and a test network of real computers on a packet-by-packet basis to look for differences; section 4.1 details this approach. A higher level analysis of the difference in goodput found is then presented in section 4.3. The following sections concentrate on Linux and FreeBSD, OpenBSD was found to have very similar results to that of FreeBSD.

4.1 Packet Trace Comparisons

NSC can output packet traces in `tcpdump` format (Jacobson et al. 2004) which can be compared to traces captured on a test network at the same location in the topology. By normalising the traces with `tcpnorm` (Jansen 2004) the traces can be compared directly by comparing the output of `tcpdump` or by graphs generated by `tcptrace` (Ostermann 2004).

Figure 2 shows examples of time sequence graphs comparing TCP traces in simulation using FreeBSD in the Network Simulation Cradle and a trace captured from the test network. The bottom line on the graphs shows the sequence number that has been acknowledged to. The top line is the bottom line plus the size of the receivers advertised window. The small vertical lines with arrows at both ends indicate data segments. Each graph shows the first second of slow start over a link with a round trip time (RTT) of 200ms and a bandwidth of 2Mb/s. There is only one difference in the two graphs: in the simulated version the first data packet sent out has the PUSH flag set, indicated by a diamond shape on the graph. This difference is created by the applications that write to the TCP socket; the PUSH flag indicates that there is no more data to send at the time the packet was created in BSD derived TCP stacks such as FreeBSD and OpenBSD.

The same graphs can be shown for Linux to illustrate that differences between TCP implementations. Figure 3 shows same same scenario but using Linux 2.4.27 instead of FreeBSD. The TCP PUSH flag again is different during this trace due to differences between the simulated application and the application used on the test network. There is also a slight difference in timing of packets that is created by the processing delay on the real machine which is not present in simulation.

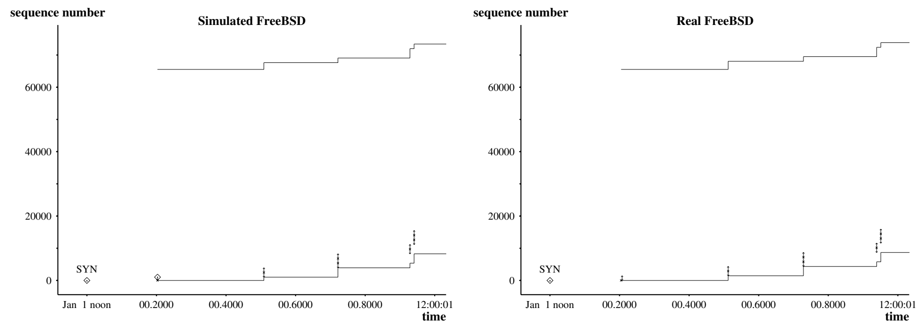

Figure 2: Time Sequence Graphs of Simulated and Real Freebsd

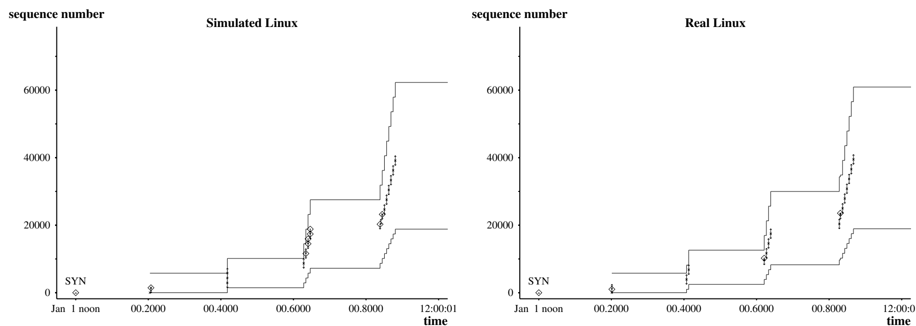

Figure 3: Time Sequence Graphs of Simulated and Real Linux

4.2 Comparison of Packet Traces with tcpdump

The graphical views presented in the previous section do not show all the necessary detail needed to accurately compare two packet traces. By looking at the differences in the output of the `tcpdump` command changes in packet sizes, TCP options, and timing can be observed. We compare traces for both Linux and FreeBSD in the same scenario as presented in figures 2 and 3.

4.2.1 Linux

The first five packets generated are exactly the same expect for a small difference in the time they are received. Differences of one millisecond are found which are due to the model of the physical network; there is variation in the laboratory test network that is not present in simulation. This makes a small difference to the packets generated: the TCP time stamp options sometimes differ by one. The unit used for time stamps in the case of Linux is *jiffies*, which has a granularity of 10 milliseconds. The difference in time stamp is sporadic; for packets six to eight it differs, but is in sync for the next three packets. Measuring ping times on the test network showed that due to the use of software routers and end hosts, there was a standard deviation of 2.2 ms over 150 ping packets of 64 bytes each. Such variation is not present in simulation.

During periods of no loss the traces are very similar. Packet timings are within 2 ms of each other. Though there is the occasional small difference in packet flags, the traces stay in sync.

4.2.2 FreeBSD

The exact same packet sequence can be seen in FreeBSD for the first 76 packets. The packets are identical apart from some having the TCP time stamp slightly different, which only ever differs by one. A difference occurs when an acknowledgement is sent sooner in the laboratory test which makes the two streams somewhat out of sync.

4.2.3 Conclusions

There are many variables which affect the exact sequence of packets that are output by both stacks. How the network driver works makes a difference in some cases. It was found in Linux that allocating different amounts of memory for the packet container in the kernel (*skbuff*) meant the initial window size offered in TCP packets changed. The modelling of the application is also important, how applications make use of blocking versus non-blocking IO makes a difference. Effort was made to make the simulation the same as the laboratory machines, but there are still some differences outstanding. It is unlikely there is anything wrong with the inner workings of the network stacks; all evidence encountered shows the differences to come from external factors: the simple network driver model and application model in particular.

4.3 Higher Level Analysis

Higher level analysis shows the differences in packet traces are indicative of only a small difference in measured goodput. The comparisons in section 4.1 do not analyse many facets of TCP. Performing more complex tests and comparing total goodput over a long period gives a higher level view on whether the TCP implementations in NSC are performing as they do on a real network.

Table 1 in section 1 showed the differences between ns-2's TCP in simulation and measurements from TCP implementations on a test network. A similar table is presented here, but with the results of simulations performed with NSC included.

The NSC results in Table 2 are run in the same conditions as their ns-2 counterparts and are again averaged over 100 runs with only the random seed differing from run to run. The largest difference found between NSC and laboratory tests is 4%. In the case of Linux with SACK disabled, there is a difference of less than 1%. This is an example of a scenario that ns-2's TCP implementations do not model correctly yet the Network Simulation Cradle is able to report performance very close to what is measured on a real network. The differences found here are because of the slight variations between a real and simulated network and differences in a real and simulated application. Each stack responds to these variations differently, explaining the range of one to four percent changes in measured goodput.

Table 2: Comparative Mean TCP Throughput with 5% Bidirectional Loss

TCP Implementation	Goodput (kb/s)
OpenBSD 3.5	113.2
NSC: OpenBSD 3.5	109.0
Linux 2.4.27	208.6
NSC: Linux 2.4.27	204.9
Linux 2.4.27, No SACK	193.7
NSC: Linux 2.4.27 No SACK	192.7
FreeBSD 5.2.1	162.8
NSC: FreeBSD 5.2.1	156.5

5 PERFORMANCE

This section reports on measurements of the performance of the Network Simulation Cradle, both in running time and memory usage. We compare simulations run with ns-2's FullTcp agent with network stacks in the Network Simulation Cradle. The measurements were run on an Athlon XP 1800+ with 512MB SDRAM running FreeBSD 5.3. ns-2's New Reno agent was also tested for performance. The results were largely similar to FullTcp, varying by 5% at most. For the sake of brevity, only the results of FullTcp are presented in the following sections, as its performance results are representative of ns-2's TCP implementations.

Compiler optimisations are enabled for building ns-2 and the simulation cradle network stacks. Linker optimisations are also enabled for the shared libraries in NSC.

5.1 Time Performance

Various factors could influence how long it takes to simulate a specific situation. Initially, we look at varying the number of nodes in a simulation, then look at increasing the simulated time for a small number of nodes. To test the per-packet overhead this simulation is then reproduced with a smaller segment size. An analysis of these results follows in section 5.1.4.

5.1.1 Time to Simulate *n* Nodes

In this particular situation, we look at simulating a growing number of nodes communicating over a bottleneck link in a classic barbell topology. Each simulation has 60 seconds of traffic. At the start of a simulation, half of the nodes connect to the other half. The connection goes through a single, limited link. There is a unidirectional TCP transfer of data over each connection until the end of the simulation. The bottleneck link is 2Mb with 10ms latency. Anagnostakis et al. (2002) report this type of simulation scenario to be by far the most common scenario simulated.

The results of these simulations are summarised in Figure 4. The difference in time between ns-2 and most of the NSC agents in this graph is largely due to initialisation. Figure 4 shows that simulating a large number of NSC stacks takes time, but the overhead over ns-2 doing the same with it's TCP implementations is small. NSC requires extra initialisation which explains the extra time in this scenario. There is a one-off cost of loading a shared library which can be seen by the initial increase in real time when the number of nodes is around five. The initialisation of each NSC stack takes some time; the cost of this differs between each network stack. lwIP is simplistic and requires very little initialisation, while FreeBSD, OpenBSD and Linux have more complicated processes. The slowdown by two to three times is probably acceptable in many cases.

There is only a small data transfer in the scenario reported by figure 4. How quickly the Network Simulation Cradle can process data is examined next.

5.1.2 Larger Amounts of Traffic

Rather than increasing the number of nodes, this test has a static number of nodes but increases the length of the time simulated. The simulation setup is otherwise the same as in the previous section. This shows how the simulator scales with respect to the volume of data going through it. Figure 5 shows the results of this set of simulations.

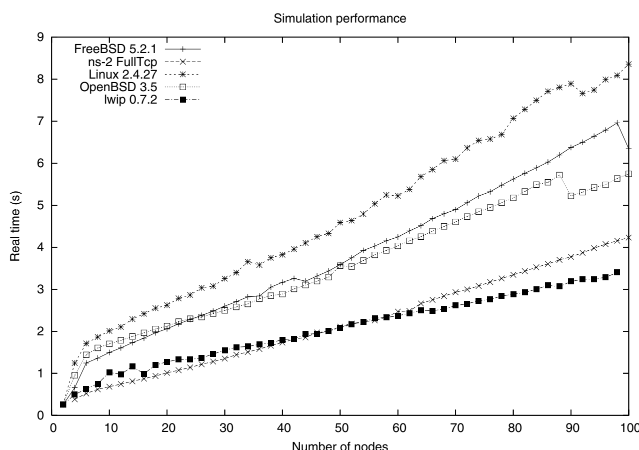

Figure 4: Time Taken to Simulate an Increasing Number of Nodes

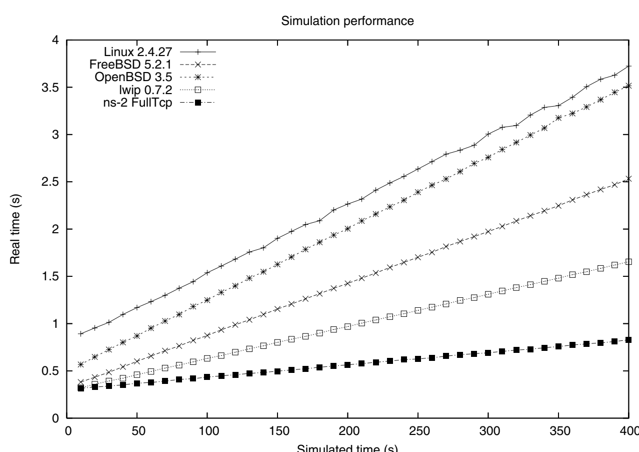

Figure 5: Simulation Times for Longer Simulations

The difference in time found when the simulated time is small is due to the overhead of loading the different shared libraries. The shared libraries containing the OpenBSD and Linux network stacks have many symbols, up to ten times as many as the libraries containing FreeBSD and lwIP. This is due to the operation of the parser, in some cases it will create many new symbol names which need to be linked by the dynamic linker when first loading the shared library.

Figure 5 shows that ns-2's TCP implementations can be a lot faster than NSC stacks used for simulation. OpenBSD is nearly six times slower than FullTcp in this scenario. The next section tests how much of this slowdown is a result of the per-packet overhead.

5.1.3 Per-Packet Overhead

To test per-packet overhead, the simulation described in section 5.1.2 was re-run with one modification: a smaller maximum segment size (MSS) was used. For the NSC network stacks, the maximum transfer unit of the interface is set to a lower value, while for ns-2's FullTcp the segment size is reduced. They are set so the maximum transfer unit (MTU) is reduced from 1500 to 576 in both cases. By comparing this simulation to the previous one per-packet overhead can be seen.

Table 3 shows the results of this simulation. Plotting a graph shows linear trends like figure 5. It is interesting to compare the ratio of real time to cpu time for the two MTU sizes simulated. The difference between simulating with the different MTUs gives an idea of the overhead due to packet processing. An interesting result here is that FullTcp suffers the greatest slowdown in this case.

Table 3: Per-Packet Simulation Performance

TCP Implementation	Time ratio		Change
	MTU 576	MTU 1500	
NSC: Linux 2.4.27	70.906	144.64	2.04x
NSC: OpenBSD 3.5	62.711	139.14	2.21x
NSC: FreeBSD 5.2.1	97.951	190.88	1.94x
ns-2: FullTcp	327.97	837.60	2.55x

5.1.4 Analysis

The CPU performance differs between ns-2's TCP implementations and using the Network Simulation Cradles TCP implementations in ns-2. In the situations studied here, we find the ns-2 FullTcp agent to be up to six times faster than simulating with an NSC network stack. While this is a significant slowdown, for many simulations this will be acceptable. The results in figure 4 and table 3 show that in some situations the difference in performance between native ns-2 simulations and using NSC is closer than reported in figure 5. When creating ns-2 trace files such as Nam traces (Estrin et al. 1999), the difference is small due to the large slowdown created by writing the trace to disk.

NSC network stacks work as a drop-in replacement for the existing ns-2 TCP agents. This means it is possible to design and run simulations using the existing agents then use the NSC stacks for validation purposes if the simulation scenario is too large to be run using the NSC stacks.

The previous sections showed that using NSC slows down simulations with ns-2 but do not analyse the impact on memory usage. When using NSC, the real packets are passed through the simulator which has the potential for much greater use of memory. Each network stack also allocates some memory for internal buffers and structures. The amount of memory used when simulating with ns-2 and NSC is reported in the next section.

5.2 Memory Performance

The virtual memory size of ns-2 when simulating a simple scenario similar to that described in section 5.1.1 grows linearly as the number of stacks instantiated grows. There

is a large difference in size between OpenBSD and Linux and the rest of the NSC stacks and ns-2 agents due to a large `.bss` (uninitialised data) section in the OpenBSD and Linux shared libraries. The parser described in section 3.3 will in some circumstances create many uninitialised variables. Much of this will never be paged in and will not use physical memory; in the case of OpenBSD only around 25 of 90 megabytes of memory was in the resident set when 20 stacks were used in this test. In larger tests this memory is more likely to be used.

The data structure used in ns-2 to describe packets does not contain packet payload during normal operation of the ns-2 TCP agents. The Network Simulation Cradle stacks require the real packet data so when NSC stacks are being simulated the packet data is attached to ns-2's packet structure. This means that when there is a complex situation with many router buffers the memory usage will grow due to the extra overhead required per packet in the simulator. To test this situation a larger simulation was run and the memory usage recorded.

Table 4 displays the virtual memory size at the end of the simulation when there are 300 stacks instantiated and sending and receiving data in a complex topology. The memory usage is a lot higher when using NSC than with the base ns-2 TCP agents. Using different ns-2 TCP implementations did not change the amount of memory used from that reported in Table 4 significantly.

Table 4: Memory Usage During Larger Simulation

TCP Implementation	Virtual memory size (MB)
NSC: Linux 2.4.27	134.224
NSC: OpenBSD 3.5	113.032
NSC: FreeBSD 5.2.1	60.064
NSC: lwIP 0.7.2	39.400
ns-2: Newreno	26.524

The performance figures presented in this section report on a largely unoptimised use of real world network stacks in simulation. Memory usage could be reduced by using only enough memory per packet to contain the packet headers as the data payload is unused. This would also mean that only the header needs to be copied, currently there is a large amount of redundant memory copying done due to copying the entire packet contents into a buffer during transmission and receipt of a packet. The stacks need to be profiled to gain further insight into their performance characteristics. This is part of the ongoing development of the Network Simulation Cradle.

6 CONCLUSIONS

This paper describes the Network Simulation Cradle which makes network stacks from real world operating systems available to simulation. While we describe integration of the cradle with the network simulator ns-2, the approach is general and the current framework could be integrated with other network simulators. NSC currently includes the network stacks from the operating systems Linux, FreeBSD and OpenBSD and the network stack designed for embedded systems, lwIP.

NSC has been validated by directly comparing with results from a test network. Results are very close to those measured on the real network, while in some cases ns-2's TCP models produce significantly different results. Comparing packet traces by hand from the test network to traces generated from simulation shows that the network stacks used in NSC produce different results from each other but are very close to the same stack used on the test network.

We show that there is a performance impact of using real network stacks in simulation, though large simulations are still able to be run in reasonable time. Using NSC with ns-2 requires little change to current simulation scripts meaning this approach is complementary to existing simulations performed with ns-2. It is possible to design simulation scenarios using the current protocol implementations in ns-2 then replace them with the more accurate protocol implementations found in the Network Simulation Cradle.

Future work will involve adding further network stacks to the simulation cradle such as newer Linux versions (the 2.6 series of the Linux kernel) and any other open source network stacks of note. Performance can also be increased by reducing the amount of memory used by the packets in the simulator and reducing the amount of copying done.

REFERENCES

Anagnostakis, K., M. B. Greenwald, and R. S. Ryger. 2002. On the sensitivity of network simulation to topology. In *Modeling, Analysis, and Simulation of Computer and Telecommunications*.

Bless, R., and M. Doll. 2004, Dec. Integration of the FreeBSD TCP/IP-stack Into the Discrete Event Simulator OMNeT++. In *Winter Simulation Conference*, 1556–1561.

Dunkels, A., L. Woestenberg, K. Mansley, and J. Monoses. Accessed 2004. lwIP embedded TCP/IP stack. <http://savannah.nongnu.org/projects/lwip/>.

Ely, D., S. Savage, and D. Wetherall. 2001, March. Alpine: A User-Level infrastructure for network protocol development. In *3rd USENIX Symposium on Internet Technologies and Systems*, 171–184.

Estrin, D., M. Handley, J. Heidemann, S. McCanne, Y. Xu, and H. Yu. 1999, March. Network visualization with the VINT network animator nam. Technical Report 99-703b, University of Southern California. revised November 1999, to appear in IEEE Computer.

He, Q., M. Ammar, G. Riley, H. Raj, and R. Fujimoto. 2003, Octobor. Mapping peer behavior to packet-level details: a framework for packet-level simulation of peer-to-peer systems. In *Modeling, Analysis and Simulation of Computer Telecommunications Systems*. Georgia Institute of Technology, Atlanta.

Heidemann, J., K. Mills, and S. Kumar. 2001, Sept./Oct.. Expanding confidence in network simulation. *IEEE Network Magazine* 15 (5): 58–63.

Huang, X. W., R. Sharma, and S. Keshav. 1999. The ENTRAPID protocol development environment. In *INFOCOM (3)*, 1107–1115.

Information Science Institute (ISI) Accessed 2004. The network simulator - ns-2. <`http://www.isi.edu/nsnam/ns/`>.

Jacobson, V., C. Leres, and S. McCanne. 2004. tcpdump. <`http://www.tcpdump.org`>.

Jansen, S. 2004. tcpnorm. <`http://www.wand.net.nz/~stj2/nsc/software.html`>.

Ostermann, S. Accessed 2004. tcptrace. <`http://www.tcptrace.org`>.

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

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

Wang, S. Y., C. L. Chou, C. H. Huang, C. C. Hwang, Z. M. Yang, C. C. Chiou, and C. C. Lin. 2003. 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.

Zec, M. 2003. Implementing a clonable network stack in the FreeBSD kernel. In *USENIX Annual Technical Conference*, 137–150.

Zec, M., and M. Mikuc. 2003, June. Real-time ip network simulation at gigabit data rates. In *7th Intl. Conference on Telecommunications*.

AUTHOR BIOGRAPHIES

SAM JANSEN is a PhD student at The University of Waikato. The work described in part in this paper forms a significant portion of his PhD research. He started in the WAND Network Research group in 2003. His email address is <`sam@wand.net.nz`> and his website is <`http://www.wand.net.nz/~stj2/nsc`>.

TONY MCGREGOR is an Associate Professor with the Department of Computer Science of The University of Waikato in New Zealand. His resaerch areas are network measurement and simulation. He is a senior member of the Waikato University WAND network measurement group and leads the NLANR AMP active monitoring project. He teaches operating systems and computer networks.