Validation of simulated real world TCP stacks

Sam Jansen and Anthony McGregor
WAND Network Research Group
University of Waikato
Hamilton, New Zealand
Email: sam@wand.net.nz, tonym@cs.waikato.ac.nz

This is an early version of the paper (June 13, 2006) that is under submission.

***Abstract*— The TCP models in the network simulator ns-2 are widely used in network research. They are not aimed at producing results consistent with a TCP implementation, they are rather designed to be a general model for TCP congestion control. The Network Simulation Cradle makes real world TCP implementations available to ns-2: Linux, FreeBSD and OpenBSD can all be simulated as easily as using the original simplified models. These simulated TCP implementations can be validated by directly comparing packet traces from simulations to traces measured from a real network. We describe the Network Simulation Cradle, present packet trace comparison results showing the high degree of accuracy possible when simulating with real TCP implementations and briefly show how this is reflected in a simulation study of TCP throughput.**

I. Introduction

For simulation results to be credible the simulation models in use must undergo *verification and validation*. Balci [1] defines verification as substantiating that a model is built from a problem formulation accurately, where validation is substantiating that the model behaves with satisfactory accuracy within its domain. Carson [2] and Sargent [3] define the two terms to be similar and both note that sufficient accuracy is when a model can be used instead of a real system for purposes of experimentation and analysis.

In the context of simulation models for TCP, the models should be tested that they conform to the TCP specification (verification of the model) and that the model implementation produces results consistent with a real system (validation of the model).

Bagrodia and Takai [4] raise the question of whether a TCP model is correct with respect to actual TCP implementations and list two cases where validation was quite successful in their work with the GloMoSim [5] simulator. Direct incorporation of the implemented protocol into the model allows the protocol model to be validated against an operational prototype. Comparison of independently developed models for a given protocol provide further validation information.

The ns [6] simulator has a test suite that tests many facets of the simulator including the one-way TCP agents [7]. The TCP tests cover a range of situations designed to provoke certain behaviour for each TCP variant. For example, the fast recovery mechanism of TCP Reno is tested with differing amounts of packet loss. A similar, though less thorough, set of tests exists for the bidirectional TCP agents [8]. This type of testing is a verification that the models produce results consistent with specifications.

Floyd [7] points out that the TCP models in the ns simulator are not designed to model one specific real world TCP implementation but be a general model for experimenting with the underlying congestion control algorithms. The Network Simulation Cradle [9] (NSC) uses real TCP implementation code in TCP models in simulation, allowing a different sort of validation to be used. The simulation model can be *directly compared* to a real network: the output of the simulation model should be very close to that of a real machine given the same input. Using real TCP implementations makes a full range of features available; simulated TCP models are often limited in the features they support. Protocol development is also possible with real world code in a simulated environment. Protocol development within kernel code but in user-space and a simulated environment is an ideal process before testing the implementation in-kernel on machines. With more than one TCP implementation available, NSC allows a researcher to view a range of results recorded from TCP implementations in a simulated scenario.

The Network Simulation Cradle makes the OpenBSD, Linux and FreeBSD network stacks available as ns-2 TCP agents. These agents have the same interface as the original ns-2 TCP models, making it very easy to use

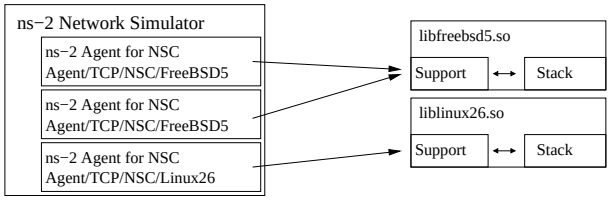


Fig. 1. Interaction with multiple network stacks

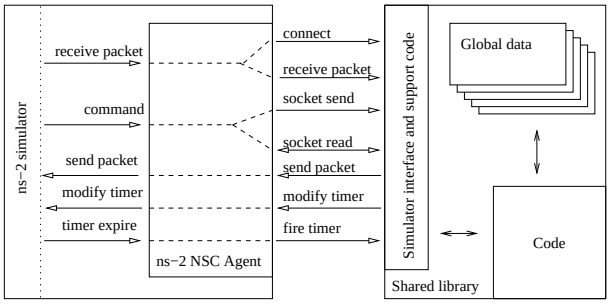


Fig. 2. Per-network stack interactions

NSC models in the place of or in addition to the existing models in a simulation script. This paper presents validation work done with the Network Simulation Cradle and ns-2 and shows the degree of accuracy attained when using real world code for simulation of TCP.

Both methods are used in this paper to show the validity and accuracy of the Network Simulation Cradle TCP implementations. The results of directly comparing traces with real networks are presented in section III. Comparing simulation results with ns-2 TCP models follows in section IV. In section II the architecture of the Network Simulation Cradle is described.

II. Architecture of the Network Simulation Cradle

NSC is designed as two distinct objects that communicate through a well defined interface. There is an ns-2 *agent* that provides a 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 creating an instance of and interacting with the other part of NSC, the shared library. The shared library contains the network stack as well as supporting code. NSC also uses another component during the build process: the global parser programmatically changes references to global variables while building the shared library.

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

The Network Simulation Cradle version 0.2.2[1] is used with ns-2 version 2.29. NSC 0.2.2 includes the network stacks Linux 2.4.28, Linux 2.6.10, FreeBSD 5.3 and OpenBSD 3.5.

A. Simulator integration: an ns-2 agent

The ns-2 agent is responsible for routing messages between the simulator and the network stack that resides in a shared library. The agent first loads the correct shared library containing the requested network stack and initialises it. 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. A view of some these interactions is shown in Fig. 2.

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 for NSC is exposed to simulation 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 in this mode.

B. The shared library

The shared library is made of three parts. These are the C++ simulator interface, required to communicate with the simulator, the network stack itself and support routines. The interface is a C++ class that is

[1]Available from `http://research.wand.net.nz/software/nsc.php`

created via the only function exported to the simulator; a `create_stack` function. Support routines form a bridge between the network stack and the C++ interface.

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 practise as every protocol implementation in kernel space encountered does not require blocking.

The shared library contains the network stack along with supporting code that implements the interface necessary to communicate with the simulator. Shared libraries are used instead of static libraries because shared libraries provide namespaces for the symbols contained within them. 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.

There is an important distinction between 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.

C. The global parser

Network stacks are designed to run inside an operating system, there is normally no allowance for multiple instances of network stacks[2]. 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 its 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. This approach does not scale well, 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 a large effort to update the network stack to future versions.

It is possible to change global variable definitions and references programmatically. Because network stacks are written in C code, creating a filter which processes the C code and understands the global variables and how to change them is possible. 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 implements a full C grammar and supports C well enough to replace global variables and global variable definitions correctly while leaving the surrounding code the same. The ANSI C and C99 specifications are mostly implemented and the parser has been tested on the networks stacks of the Linux, FreeBSD and OpenBSD kernels.

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[3]. 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.

III. Trace comparisons

The Network Simulation Cradle can produce packet trace files in the format used by tcpdump [11]. Tcpdump captures packets from a network interface and optionally saves them to a file. This facility can be used to validate a simulation that is modelled of the same network setup. Tcpdump traces can be recorded at the same logical points in the two networks: the network trace from NSC and from a real machine can then be directly compared using trace analysis tools such as tcptrace [12]. Such a method of comparison is used in this section.

An example of such validation comparisons follows. A testbed network called the WAND Emulation Network [13] is used to generate real tcpdump traces. Six machines are configured in a simple dumb-bell topology. Two machines run FreeBSD 5.3 and use ipfw Dummynet [14] to shape traffic. Due to the scheduling of packet delays with Dummynet[4], the round trip time (RTT) on this network has some noticeable variation as presented in Table I. The RTT of an equivalent

[2]An exception to this rule is the FreeBSD Network Stack Virtualization project [10], but this is not part of core FreeBSD; it is distributed as kernel patches.

[3]Not all global variables should be modified. For example, a global variable is used to select which network stack instance is currently being used.

[4]Dummynet processes delays on a software interrupt clock, which fires once every $1/HZ$ seconds. HZ is set to 1000 in these tests.

TABLE I
EMULATION NETWORK RTT MEASUREMENTS

	Round trip time (ms)				
Packet size	Min	Median	Max	Std. Dev.	Simulated
84	43.0	43.6	49.9	0.588	43.1
1500	53.3	53.8	61.1	0.653	54.4

network simulated with ns-2 is also shown. The results are gathered from 1000 pings on an otherwise unloaded network. A FreeBSD Dummynet router is configured to delay packets by 21ms in both directions and limit bandwidth to 2Mb/s.

The variation in timing on the testbed network shown in Table I means that there will be some small variation in timing between the simulated trace and the measured trace. A direct binary comparison of the traces is therefore not useful. The sequence numbers and time used in the traces are also not synchronised. The tcpnorm [15] utility is used to normalise the traces by changing the time and sequence numbers in a trace to start from 0 and tcptrace [12] is used to visualise them by making time-sequence graphs.

The bottom line on a tcptrace time sequence graph (see e.g., Fig. 3(a)) is the sequence number which has been acknowledged to. The top line is the acknowledgement number plus the receivers advertised window. This shows visually the window in which the data packets should be sent. Data packets are indicated by small black double-ended arrows. If the packet is a retransmission, it will have an "R" next to it. Selective acknowledgement blocks are shown by lines within the advertised window with an "S" next to them. If a data packet has the PUSH flag set a diamond will be drawn around the packet.

A. Connection establishment

Fig. 3 shows tcptrace graphs of TCP during connection establishment and slow start. For each operating system a trace is measured on the testbed and created in simulation. A Dummynet router limits bandwidth to 2Mb/s, delays packets in both directions by 21ms and has a queue length of 10 packets. The simulation scenario is configured to be equivalent. The two graphs for each operating system are shown side by side.

Each of the pairs of graphs in Fig. 3 are very close matches for each other. In addition to these graphs, each situation is analysed in detail using the textual output of tcpdump in the following sections.

1) FreeBSD: The two traces for FreeBSD are very close. The sequence of packets shown in Figures 3(a) and 3(b) are identical save for the TCP timestamp option. This often differs by one in the traces. The reason for this is that the timestamp counter is based on the `ticks` variable in the network stack which in this situation occurs once every 10ms. This timer starts counting when the machine boots, so synchronising it between simulation and the real machine is not practical.

There is a small difference in the timing of packets. This is due to the difference in round trip time and variation in timing found in the emulation network. This eventually leads to a slightly different ordering of packets, although the tcptrace graphs look similar. This is analysed further in section III-B.

2) Linux: The traces for Linux look similar in Figures 3(c) and 3(d). The one notable difference is some of the data packets have diamonds around them meaning they have the PUSH flag set.

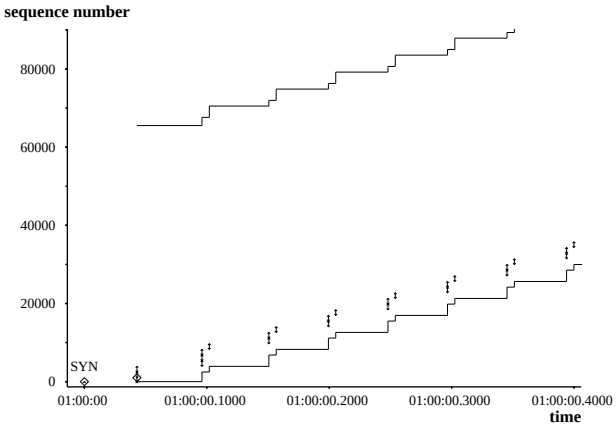
The PUSH flag in TCP was originally specified in RFC 793 to mean that when a receiving TCP sees the flag, it must not wait to receive any more data before passing the data to the receiving process. In practise, data is passed to the application as soon as possible irrespective of the PUSH flag and it is set by the sending network stack, rather than the application, in most recent TCP implementations.

The interface between application and network stack is very different in simulation with ns-2 and on a real machine[5], so the model of the application is slightly different between the two. These differences result in the PUSH flag being set for extra packets in simulation.

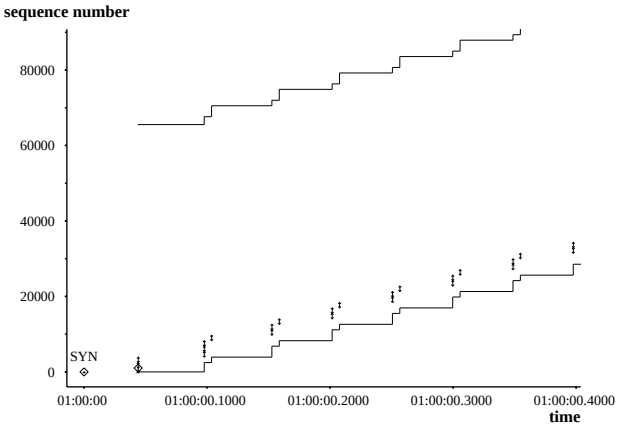
The TCP timestamp option differs between the traces. The counter used for the timestamp is increased once every millisecond in the version of Linux studied. The packets are consistently between 0 and 3 milliseconds different in their timings due to the effects of the real network and Dummynet and the TCP timestamp option reflects this.

An example of subtle trace differences occuring: Linux 2.6 tunes the windows used in TCP based on the amount of memory available in the machine. The cradle code attempts to have reasonable defaults set that match the machines on the emulation network. The receivers advertised window grows dynamically and is additionally affected by the size of the packet structure allocated in the Ethernet driver. This is an example of subtle interactions coming from seemingly unimportant supporting code. We believe that validating real world code in simulation is important, because there is the

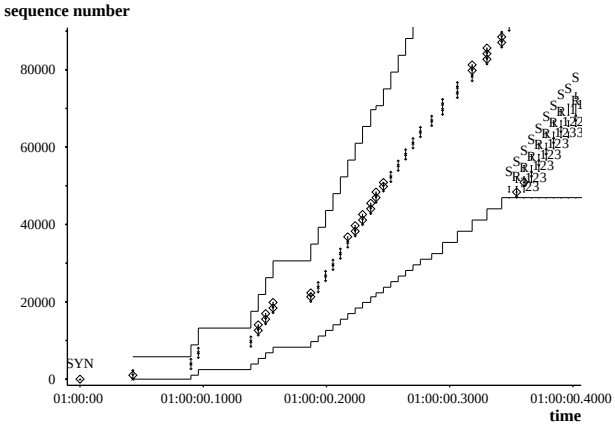
[5]ns-2 does not provide a sockets API like real operating system do.



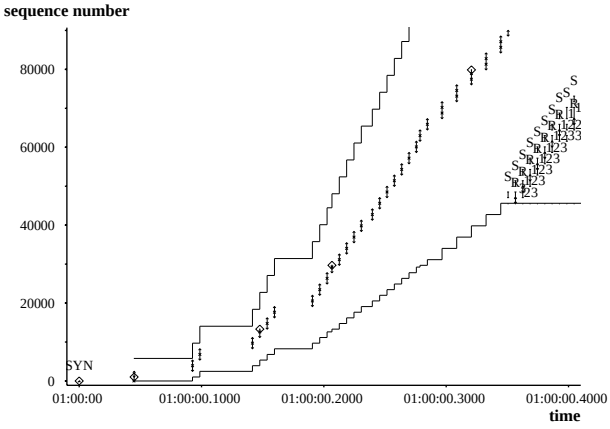
(a) Simulated FreeBSD



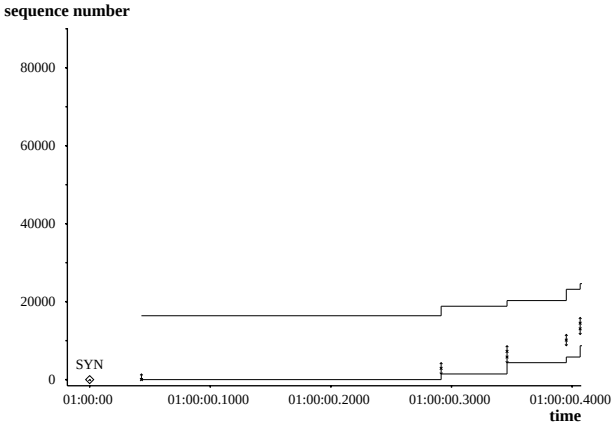
(b) Measured FreeBSD



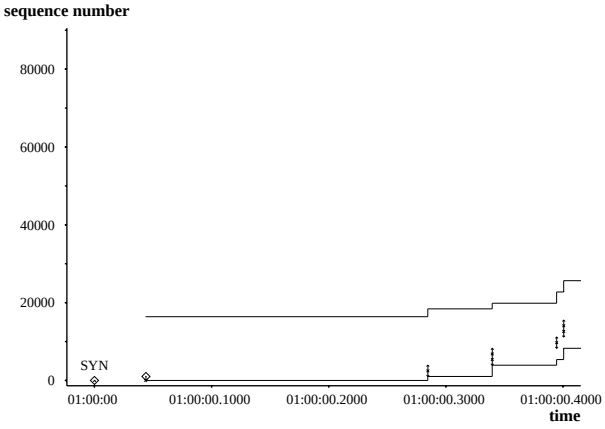
(c) Simulated Linux



(d) Measured Linux



(e) Simulated OpenBSD



(f) Measured OpenBSD

Fig. 3. Simulated vs. measured connection establishment graphs

possibility of many interactions such as this affecting results.

When a packet is received in a network driver, the driver allocates a structure called an `skbuff` with enough space to hold the packet. It is up to the driver to select the space for the packet received, often there is extra slack space that is unused by the driver (but possibly used later by other sections of the network stack). This packet is then sent to the network stack. When calculating the receivers advertised window, the size of the `skbuff` is checked.

To obtain the same traces on real machines and in simulation, the simulation driver code needs to allocate `skbuff` sizes in the same manner as the driver used on the real machine. The simulation driver allocates `skbuff`s similar to the *eepro100* driver used on the emulation network machines and is able to produce the same offered window sizes as those measured on the emulation network.

The traces are identical until the difference in PUSH flags save for the slight timing differences described above. On the real machines some data packets are generated later in the trace that are smaller than the MTU. This is due to application differences; the timing of when data is written to the TCP socket by the application is different between simulation and the real machine which results in this behaviour. The traces are very similar when visualised with tcptrace and the goodput[6] measured on the emulation network is within 2% of the goodput recorded in simulation.

3) OpenBSD: The sequence of packets shown in Figures 3(e) and 3(f) are very close matches. When the traces are analysed further it is evident there is an off by one difference in the TCP timestamp option. This occurs for the same reason it does in the FreeBSD trace and is described earlier.

The OpenBSD sender only sends one initial data packet after the three-way handshake of TCP. The acknowledgement for this packet is not sent straight away by the other end of the connection due to the delayed acknowledgement mechanism: either the delayed acknowledgement timer must fire or two packets must arrive. This is one of the reasons for RFC 3390 which increases the initial TCP window size. The version of OpenBSD tested does not implement RFC 3390 while the versions of Linux and FreeBSD studied here do.

Figures 3(e) and 3(f) show a timer firing with the same duration in emulation and simulation. This is an example of a test validating the timer mechanism.

The timing difference of packets is similar to that found for FreeBSD. This eventually leads to a different sequence of packets, though an overall tcptrace graph of the connection looks nearly identical and the goodput recorded in simulation is within 2% of the goodput measured on the emulation network.

B. Congestion

Figures 4 and 5 are tcptrace graphs of TCP undergoing loss because it has overflowed the router queue size. The scenario simulated and measured is the same as earlier, these graphs are produced from later in the connection. Only FreeBSD and Linux are covered here due to space considerations, OpenBSD also shows similar results between simulation and emulation.

1) FreeBSD: FreeBSD responds to the packet loss in the same manner in simulation and on the testbed network. Fig. 4(a) and Fig. 4(a) show the same selective acknowledgement ranges and bursts of data packets due to the loss. The difference in the two graphs is the time and sequence numbers shown on the axis. This is due to loss occurring slightly earlier on the emulation network. While the difference in network produces this discrepancy, the graphs show the algorithmic response of TCP is the same in simulation as it is on the real machines.

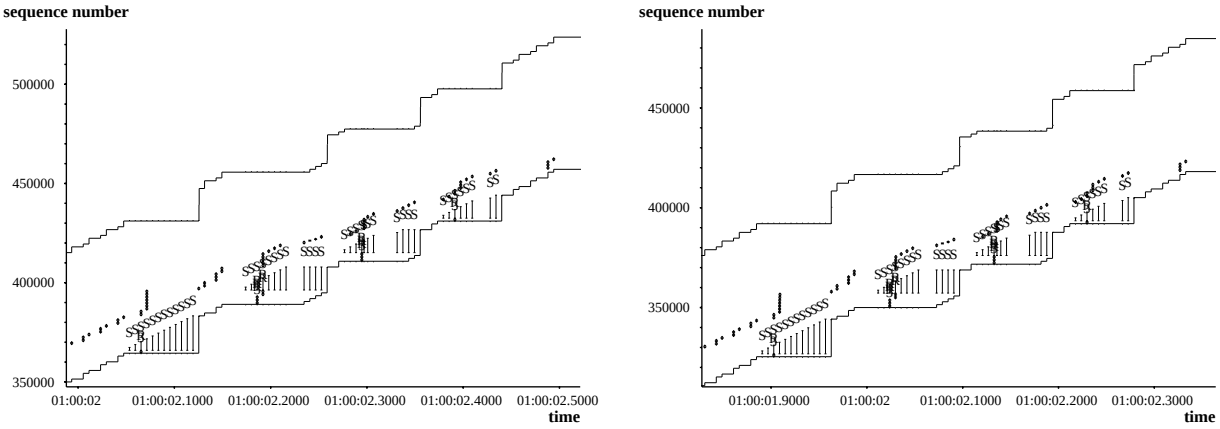
2) Linux: The graphs in Fig. 5 do not show the TCP PUSH flag as previous graphs have. This makes the graphs easier to follow and compare. The two graphs have the same sequence numbers and time shown, unlike Fig. 4, and are almost an exact match. The response to packet loss is the same with a simulated Linux TCP stack and one running on a real machine.

IV. Simulation comparisons

The performance of TCP during varying uniform random loss rates is presented in Fig. 6. Simulation results using ns-2 with its standard TCP models and with NSC TCP implementations are shown in Fig. 6(a) and results measured from the WAND Emulation Network are shown in Fig. 6(b). The TCP flow goes through a network with a RTT of 200ms and a bandwidth of 2Mb/s. Each point on the graphs is the mean of six runs of the same test.

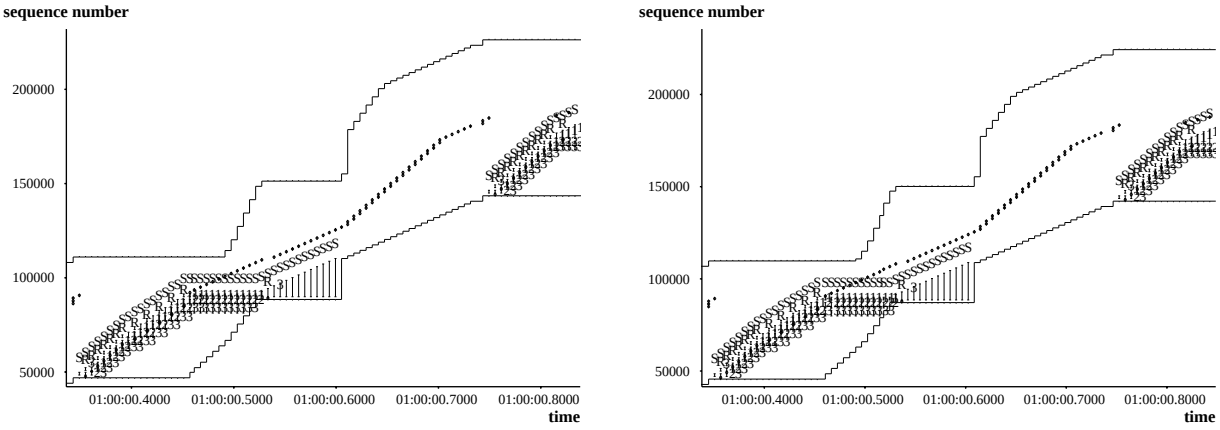
The goodput of Linux 2.6 is much higher during low loss than the other TCP implementations in simulation and measured on the testbed. A likely explanation are the auto-tuning buffer sizes with large maximums combined with the use of BIC-TCP [16] by default [17].

[6]Data received by the application layer from TCP.



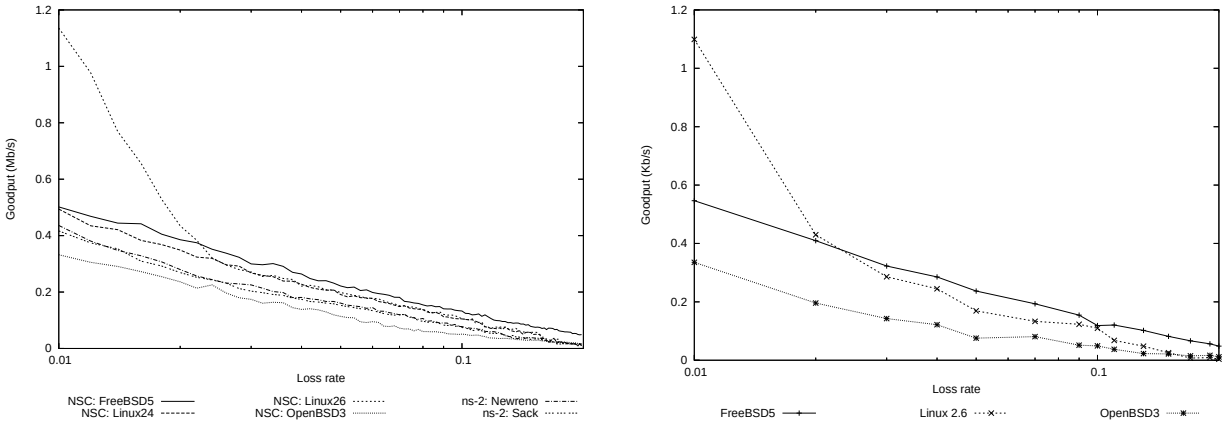
(a) Simulated FreeBSD (b) Measured FreeBSD

Fig. 4. Simulated vs. measured TCP packet loss response for FreeBSD



(a) Simulated Linux (b) Measured Linux

Fig. 5. Simulated vs. measured TCP packet loss response for Linux



(a) ns simulation (b) Testbed measurements

Fig. 6. TCP goodput vs. loss rate

BIC-TCP is a TCP enhancement design to make TCP more scalable so it performs better on high bandwidth delay product links. As the delay is very large in this experiment the use of this congestion control algorithm makes a larger difference than in previous tests.

The simulation results of TCP implementations using the Network Simulation Cradle are consistent with measurements of the same implementations on the testbed network. All follow the same trend, with FreeBSD attaining the most goodput when the loss rate is higher (greater than or equal to 3%), OpenBSD consistently recording the least goodput and Linux 2.6 dropping from substantially more goodput at low loss rates to between the two BSD variants at higher loss rates. The ns-2 TCP models have the same general trend as the real TCP implementations studied and fall in between the measurements for the real implementations.

V. Conclusion

Using real world network stacks in simulation can produce very accurate results. The Network Simulation Cradle, a project that uses open source network stacks as TCP simulation models in ns-2, is able to produce packet traces that are nearly identical to traces collected from real machines.

Though other projects (e.g., [18], [19]) use real TCP implementations in simulation to have a full, valid, TCP implementation in simulation, we believe that the process used to extract the network stack in these projects and make it available to simulate can introduce error. Even though the network stacks in the Network Simulation Cradle have not had any lines of code modified and the process of modifying global variables is automated, we have still found subtle bugs that come from the support code that is required to allow the previously kernel-mode code to run in user-mode and in a simulator. Validation of the Network Simulation Cradle by comparing traces measured on a testbed network to traces generated in simulation has help identify and fix these bugs.

Visualising traces with tcptrace and comparing between a testbed network and equivalent simulation is a very helpful validation mechanism. The traces produced in both scenarios match to the naked eye and only small timing differences exist, save for the TCP PUSH flag in some situations. Checking traces during connection establishment, congestion and showing timers fire at the correct time gives a high level of confidence in the simulated TCP implementation. We go on to show how TCP goodput under uniform random packet loss differs between TCP implementations and TCP models and how the results again agree between a measured test network and simulation with NSC. Combining fine-grained comparisons, both visually via tcptrace and by comparing textual packet traces, with macro-level comparisons gives strong evidence that the Network Simulation Cradle TCP implementations produce correct behaviour in simulation.

References

[1] O. Balci, "Verification, validation and accreditation of simulation models," in *Proceedings of the Winter Simulation Conference*, 1997.

[2] J. S. Carson, "Verification validation: model verification and validation," in *Proceedings of the Winter Simulation Conference*, 2002, pp. 52–58.

[3] R. G. Sargent, "Verification and validation of simulation models," in *Proceedings of the Winter Simulation Conference*, 2003, pp. 37–48.

[4] R. Bagrodia and M. Takai, "Position paper on validation of network simulation models," in *DARPA/NIST Network Simulation Validation Workshop*, May 1999.

[5] X. Zeng, R. Bagrodia, and M. Gerla, "GloMoSim: A library for parallel simulation of large-scale wireless networks," in *Workshop on Parallel and Distributed Simulation*, 1998, pp. 154–161.

[6] "The network simulator - ns-2," Accessed 2006. [Online]. Available: http://www.isi.edu/nsnam/ns/

[7] S. Floyd, "Simulator tests," Lawrence Berkeley Laboratory, Tech. Rep., May 1997.

[8] K. Fall, S. Floyd, and T. Henderson, "Ns simulator tests for Reno FullTCP," 1997. [Online]. Available: http://citeseer.ist.psu.edu/247784.html

[9] S. Jansen and A. McGregor, "Simulation with real world network stacks," in *Proceedings of the Winter Simulation Conference*, December 2005.

[10] M. Zec, "Implementing a clonable network stack in the FreeBSD kernel," in *USENIX Annual Technical Conference*, 2003, pp. 137–150.

[11] V. Jacobson, C. Leres, and S. Mccanne, "tcpdump," http://www.tcpdump.org, Accessed 2005.

[12] S. Ostermann, "tcptrace," Accessed 2006. [Online]. Available: http://www.tcptrace.org

[13] "WAND emulation network," Accessed 2006. [Online]. Available: http://www.wand.net.nz/~bcj3/emulation/

[14] L. Rizzo, "Dummynet: a simple approach to the evaluation of network protocols," *ACM Computer Communication Review*, vol. 27, no. 1, pp. 31–41, 1997.

[15] S. Jansen, "tcpnorm," Accessed 2006. [Online]. Available: http://www.wand.net.nz/~stj2/nsc/software.html

[16] L. Xu, K. Harfoush, and I. Rhee, "Binary Increase Congestion Control (BIC) for fast long-distance networks," in *IEEE Infocom*, 2004.

[17] S. Hemminger, "Network performance improvements in linux 2.6," in *Linux World Expo*, February 2005. [Online]. Available: http://developer.osdl.org/shemminger/LWE2005_TCP.pdf

[18] R. Bless and M. Doll, "Integration of the FreeBSD TCP/IP-stack into the discrete event simulator OMNeT++," in *Winter Simulation Conference*, December 2004, pp. 1556–1561.

[19] C. Bergstrom, S. Varadarajan, and G. Back, “The distributed open network emulator: Using relativistic time for distributed scalable simulation,” in *20th Workshop on Principles of Advanced and Distributed Simulation*, 2006, pp. 19–28.