NETWORK SIMULATION CRADLE

A thesis
submitted in partial fulfilment
of the requirements for the degree
of
Doctor of Philosophy in Computer Science
at the
University of Waikato
by
SAM JANSEN

The University of Waikato
2008

Abstract

This thesis proposes the use of real world network stacks instead of protocol abstractions in a network simulator, bringing the actual code used in computer systems inside the simulator and allowing for greater simulation accuracy. Specifically, a framework called the *Network Simulation Cradle* is created that supports the kernel source code from FreeBSD, OpenBSD and Linux to make the network stacks from these systems available to the popular network simulator ns-2.

Simulating with these real world network stacks reveals situations where the result differs significantly from ns-2's TCP models. The simulated network stacks are able to be directly compared to the same operating system running on an actual machine, making validation simple. When measuring the packet traces produced on a test network and in simulation the results are nearly identical, a level of accuracy previously unavailable using traditional TCP simulation models. The results of simulations run comparing ns-2 TCP models and our framework are presented in this dissertation along with validation studies of our framework showing how closely simulation resembles real world computers.

Using real world stacks to simulate TCP is a complementary approach to using the existing TCP models and provides an extra level of validation. This way of simulating TCP and other protocols provides the network researcher or engineer new possibilities. One example is using the framework as a protocol development environment, which allows user-level development of protocols with a standard set of reproducible tests, the ability to test scenarios which are costly or impossible to build physically, and being able to trace and debug the protocol code without affecting results.

Acknowledgements

Undoubtedly there is one person without whom this dissertation would never have come to be: my supervisor, colleague and friend Tony McGregor. Thank you for encouraging me, for believing in me, for giving me the chance. Thank you for the ever insightful words over the years.

I was lucky enough to be a part of a great research group during my PhD; it was in the labs of WAND network research group that the majority of the research was done, even though I spent large amounts of time overseas. Thanks to those who were a part of the sometimes late evenings, the discussions over coffee or on the way to the bakery, and the much needed time away from computers. I'm glad and proud to be a part of WAND.

A three-month visit to Intel Research Cambridge turned into moving to the United Kingdom for a much longer time. Thanks Andrew, Madeleine and Michael for making me feel a part of the place. While the research lab has sadly closed its doors, the memory of my time there will live on.

I'd like to acknowledge all of those in Cambridge who have endured tales of my writing and encouraged me. It would have been easy to lose sight of the final goal of finishing the dissertation without the constant positive feedback I received from you all. Last but certainly not least: Zoe, for being there, for putting up with my mind wandering off topics at hand to think about my thesis once again, for everything.

Contents

Abstract **iii**

Acknowledgements **v**

List of Figures **xiv**

List of Tables **xv**

List of Listings **xviii**

1 Introduction to network simulation **1**

1.1 Reasons for simulation 2

1.2 Validity of simulation results 5

1.3 The Transmission Control Protocol 5

1.4 The network simulator ns-2 6

1.5 Real world code 8

1.5.1 Multiple instances: the re-entrancy problem 9

1.5.2 Kernel code in user space 10

1.6 Problem statement and scope 10

2 Simulating the Transmission Control Protocol **13**

2.1 Research using TCP simulation 14

2.1.1 TCP under differing conditions 14

2.1.2 TCP modifications 16

2.1.3 Analytical model validation 18

2.1.4 Discussion 19

2.2 Network simulators used in TCP research 20

2.2.1 ns-2 20

2.2.2 ATM-TN 23
2.2.3 GloMoSim 25
2.2.4 OPNET 26
2.2.5 x-Sim 28
2.2.6 Discussion 28
2.3 Simulating with real world TCP code 29
2.3.1 Porting the TCP implementation 31
2.3.2 Modifying the operating system 32
2.3.3 Build a supporting framework 34
2.3.4 Discussion 37
2.4 Summary 39

3 The Network Simulation Cradle 41
3.1 Simulator integration 44
3.1.1 Shared libraries 44
3.1.2 Simulator agent 45
3.1.3 Interactions 49
3.2 Global parser 53
3.2.1 Modifying C global declarations 55
3.2.2 Modifying C global references 59
3.2.3 Implementation of the globaliser 61
3.3 Adding a new stack 66
3.3.1 Extracting the TCP code 66
3.3.2 Building a standalone TCP implementation 67
3.3.3 Incorporating with the Network Simulation Cradle 69
3.3.4 Configuration issues 70
3.3.5 Updating an NSC TCP implementation 70
3.3.6 Requirements of the NSC approach 71
3.4 Summary 72

4 Accuracy of TCP simulation with real code 75
4.1 Introduction to simulation and test bed comparisons 76
4.1.1 Emulating with a test bed network 77
4.1.2 WAND Emulation Network 77

4.1.3 Traffic shaping 78
4.1.4 Traffic generation and measurement 80
4.2 Packet trace comparisons 80
4.2.1 Connection establishment 82
4.2.2 Congestion 89
4.2.3 Summary 89
4.3 Simulated TCP performance 92
4.3.1 Performance over a complex topology 92
4.3.2 Uniform random loss 94
4.4 Summary 98

5 Variation between TCP implementations 99
5.1 On benchmarking TCP 100
5.2 New simulation scenarios 101
5.2.1 Packet reordering 101
5.2.2 Many TCP flows over a dumbbell topology 102
5.3 Reproduced simulations 108
5.3.1 TCP fairness on high-speed networks 108
5.3.2 Congestion control comparisons 112
5.3.3 Request latency for a SIP proxy 119
5.4 Summary 120

6 With more detail comes greater cost 123
6.1 Performance measures covered 124
6.2 CPU time 126
6.2.1 Time to simulate simple scenarios 126
6.2.2 Time required to simulate more complex scenarios 132
6.2.3 Profile 135
6.2.4 Discussion of CPU performance 138
6.3 Memory usage and scalability 140
6.3.1 Heap use with many TCP flows 141
6.3.2 The effect of increasing the TCP window size 142
6.3.3 Total memory use 144
6.4 The cost of the globaliser 145

6.4.1 CPU (online) cost 145
6.4.2 Offline cost 147
6.5 Discussion of performance results 149

7 Conclusions and future research 153
7.1 Accuracy of simulating with real world TCP code 153
7.1.1 Feasibility 153
7.1.2 Validity and accuracy 154
7.1.3 Applicability 155
7.1.4 Performance and scalability 156
7.1.5 Discussion 157
7.2 Future research 157
7.2.1 Simulating the application layer 158
7.2.2 Protocol development environment 158
7.2.3 Network stack additions 158
7.2.4 Automated protocol testing 159
7.3 Conclusions 160

Appendices 160

A Publications authored 161
A.1 Peer reviewed journal articles 161
A.2 Conference papers 161
A.3 Conference papers as a secondary author 162

B Network Simulation Cradle manual 163
B.1 Adding a new stack 163
B.1.1 Initial build process 163
B.1.2 Shared library creation 166
B.2 Testing and validation 169
B.2.1 Initial testing 170
B.2.2 Validation 170

C Network simulators 171
C.1 OMNeT++ 171

C.2 SSF 173
C.3 GTNeTS 174
C.4 J-Sim 175
C.5 JiST 175
C.6 IRLSim 175

References **177**

List of Figures

2.1 ns-2 simulation of TCP Tahoe responding to packet loss [5] 22
2.2 TCP dynamics on an ATM network [26] 24
2.3 TCP throughput during FTP measured from a router [67] 27
2.4 NCTUns simulation architecture (adapted from [104]) 33
2.5 Kernel network stack in user space (adapted from [113]) 35
2.6 FreeBSD network stack in the OMNeT++ simulator [106] 37

3.1 Simulator and Network Simulation Cradle interactions 43
3.2 Globaliser's parser flow 61

4.1 Topology used in the WAND Emulation Network 78
4.2 Example tcptrace time sequence graph 81
4.3 Simulated vs. measured connection establishment graphs: FreeBSD 83
4.4 Simulated vs. measured connection establishment graphs: Linux . . 84
4.5 Simulated vs. measured connection establishment graphs: OpenBSD 85
4.6 Time difference vs. packet number for FreeBSD traces 86
4.7 Simulated vs. measured TCP packet loss response for FreeBSD . . 90
4.8 Simulated vs. measured TCP packet loss response for Linux 91
4.9 Multi-bottleneck scenario (adapted from [155]) 93
4.10 TCP goodput over a multi-bottleneck topology 95
4.11 TCP goodput vs. loss rate 97

5.1 TCP goodput under packet reordering 102
5.2 Simulation scenario 103
5.3 TCP performance comparisons with cumulative graphs 105
5.4 Mean goodput difference as flows and bandwith is varied 107
5.5 Ratio of throughputs of competing TCP flows [175] 109
5.6 Fairness between two TCP flows as path propagation delay is varied 111

5.7 TCP over 10Mb/s bottleneck with reverse traffic [53] 113
5.8 TCP over 10Mb/s bottleneck with reverse traffic 114
5.9 TCP goodput vs. number of traversed hops [53] 117
5.10 TCP goodput vs. number of traversed hops 118
5.11 Simulation topology used for SIP simulations (adapted from [180]) . 119
5.12 Average SIP request latency for increasing loss rates 121

6.1 Simulated time vs. real time for a single TCP flow 128
6.2 CPU time to simulate many flows 129
6.3 Time to simulate and goodput with a varying MTU 131
6.4 CPU time to simulate increasing amounts of data 132
6.5 Simulation times for complex scenarios 133
6.6 Peak heap usage for increasing number of flows 142
6.7 Peak heap usage for increasing window size 143
6.8 Measured offline globaliser costs 148

List of Tables

1.1 Simulator popularity by citations 7

3.1 Number of declarations and references of global variables 53

3.2 Number of support functions in the NSC shared libraries 68

3.3 Number of lines of code used in the NSC stack support code 69

4.1 Emulation network RTT measurements 79

4.2 TCP performance during 5% bidirectional loss [162] 96

5.1 Simulation machines used 103

6.1 Performance testing setup 126

6.2 ns-2 profile using NSC: Linux 2.6 TCP 137

6.3 ns-2 profile using NSC: FreeBSD TCP 137

6.4 ns-2 profile using original TCP agents 138

6.5 Heap usage per TCP flow 142

6.6 Heap usage / window size for ns-2 and NSC 143

6.7 Memory footprint for ns-2 and NSC 144

6.8 Globaliser runtime CPU overhead 146

6.9 Codebases used for globaliser testing 147

Listings

1.1 Example ns-2 simulation script 6

3.1 ns-2 simulation script using a TCP model 48

3.2 Using NSC TCP models in a ns-2 simulation script 49

3.3 Example cradle code to connect a socket (from Linux 2.6 shared library) 51

3.4 Aggregating globals into a structure 54

3.5 Modifying a variable into an array 55

3.6 globaliser input and output for a single variable 55

3.7 Array initialisation 56

3.8 Corrected array initialisation 56

3.9 Unbounded array initialisation 57

3.10 Array initialisation solution 57

3.11 Indexing a modified array 58

3.12 Structure redefinition 58

3.13 typedef of array element type 58

3.14 Indexing a variable reference with a function 60

3.15 Self-referential initialisation 60

3.16 Self-referential initialisation error 60

3.17 Inline code block from Linux 63

3.18 gcc attribute use from FreeBSD 63

3.19 Additional gcc keywords 64

3.20 gcc inline assembly 64

3.21 Type name parsing problem 64

3.22 Section attribute object file placement 65

3.23 globaliser section support 66

3.24 Compiling a C file with gcc and the globaliser 69

4.1 Linux 2.6 `tcp_grow_window` code 88
5.1 Weka output for information gain attribute evaluator 105
5.2 Weka output for CFS attribute evaluator 106
5.3 Information gain attribute evaluator for differences data 106
B.1 FreeBSD kernel source directory 164
B.2 Stub function from the FreeBSD support code 166
B.3 Compiling a C file with gcc and the globaliser 169

Chapter 1

Introduction to network simulation

With the rise of the Internet over the past decades has come an increasing importance on design and testing of network protocols, the languages which allow computing components to communicate. Before a new protocol is accepted as a standard, sufficient testing must be done such that the protocol is believed to behave correctly under the varying conditions found on the Internet. Modifications to existing protocols also require testing for the same reason.

Network protocols are specified in documents often written in plain English. In the case of the Internet these are released by the Internet Engineering Task Force (IETF). Even with a controlling body and public feedback on protocols before they are finalised, protocol specifications include ambiguities which lead to differences in implementations. Some decisions are left up to the implementer as well, meaning two implementations that conform to the same specification can have quite different functionality.

An operating system such as a recent version of Linux contains many network protocol implementations: IP, UDP, TCP, SCTP, IPX, Appletalk and many others. These protocols are implemented as part of the operating system in standard C code. Other protocol implementations run as applications, examples include BGP, OSPF and DNS. Other operating systems contain similar lists of protocol implementations, though the implementations themselves differ.

There are two complementary avenues to testing network protocols. The network protocol can be analysed by building models, either mathematical or procedural in nature. These models can be used to simulate the protocol, allowing quick and reproducible tests. Further testing is provided by testing the protocol on real networks with real protocol implementations; either on an isolated test network or with limited tests on the Internet. Simulation is often used due to the difficulty of running many tests on real hardware.

Simulation requires support software: the simulator itself. The simulator must have models of the network elements to be simulated. This includes, but is not limited to, models of the physical aspects of the network (e.g., cables connecting computers), routing, hosts, applications and protocols. Each of these elements is an abstract model of what happens in the real world—program code (such as C++) or mathematical models designed to mimic the important aspects of the behaviour of the real entity.

The validity of simulation results is dependant on the models in the simulator used. Researchers validate simulation models by comparing them to real world measurements and known results. By showing the simulator to perform closely to real world measurements in various scenarios, it can be concluded that the simulator has a good chance of simulating further scenarios accurately. There is always the possibility that the models present in a simulator are not correct or are simplified too much to produce useful results for complex situations. The simulation models often do not capture the implementation differences found as they are either modelled after a single implementation, or the understanding of the authors of the model, which can differ from the specification.

The next sections in this chapter introduce the reasons for network simulation, discuss validity of simulation results, describe TCP and the network simulator ns-2 briefly, introduces using real world code as simulation models, and details the problem statement and scope of this thesis.

1.1 Reasons for simulation

Network simulation has many benefits over building test networks. Network simulation is reproducible, does not require a full protocol implementation, is able to scale to large and complex scenarios, is relatively easy to set up and simpler to record results from.

Simulation is reproducible

Network simulations are designed specifically to be reproducible. This means that for a given scenario, the result will be the same from run to run given the same input. An example implementation feature supporting this is the random number generator used in a simulator. This will be a deterministic pseudo random number generator that is seeded to a specific value.

Reproducible tests are important when modifying an existing model or debugging. They allow a developer to test under exactly the same conditions as a previous test. This allows bugs to be reproduced and specific interactions to be tested.

Simulation models are quick to build

Useful models of network elements can be built in simulation without modelling every aspect that is required in a real system. Many error cases need to be covered in a real implementation and compatibility with other independent implementations must be kept. Simulation models are instead built only to work within a simulator and many assumptions can be made over a real world implementation. This allows a model developer to build with a higher level of abstraction; the simulation models are therefore quicker and easier to build.

When developing new protocols, this can be important to show that the protocol interacts correctly in a variety of situations without having to spend a large amount of time developing a full implementation. There are fewer overheads involved in testing a modification to a protocol in simulation; there is no requirement to install a new kernel or program on each network host as is necessary with real world tests.

Simulation is scalable and adaptable

It is possible to simulate many thousands of entities in a network simulator on a modern desktop computer. The cost of testing an equivalent network in the real world is much higher: the equivalent is building a network of thousands of computers with many routers and switches, then controlling and measuring them all during testing. Simulations of many network elements involving complex interactions can be performed in reasonable time and effort.

Simulation is able to cope with network elements that are not yet found in the real world so is suitable for testing protocols for networks which are not yet physically possible. An example of this is optical network research where new network architectures are proposed to make use of very high data rates such as 100Gb/s [1], simulation is often used to test the networking ideas.

Scenarios can be developed rapidly

Network simulations have tools to facilitate the creation of arbitrary topologies and scenarios. Some simulators use special purpose programming languages while others have graphical user interfaces. It is possible for a network researcher to develop a complex scenario in a matter of hours, where creating such a situation in the real world would require significant resources or may not be possible. Once a scenario is created, many simulations can be run with varied parameters to view how different parameters affect the systems being tested. A rapid development and feedback cycle is possible with simulation.

Simulation has transparent access to data

Measuring performance metrics on real networks is difficult—protocol implementations are often in-kernel (making access to statistics for user-level processes hard) or otherwise inaccessible (such as running on external hardware). Measuring extra in-kernel statistics requires either additional software (for example, Web100 [2]) or custom modifications if there is access to the kernel source code.

Real networks require that data needs to be recorded from multiple points in the network, which is difficult. Ensuring the data from these elements is recorded at the same time is hard due to clock skew between the network elements the data is recorded from. It is also possible that recording extra information will change the results due to the processing overhead incurred.

In simulation, the simulator and all models are user-level software, which makes recording results easy. Recording and processing the statistics is generally convenient due to the predefined data collection routines available in the simulator being used. As the network runs on simulated time, recording data does not affect the result of the simulation and there are no problems with clock synchronisation.

1.2 Validity of simulation results

Simulation results are only believable if the simulation models being used are validated. Heidemann, Mills and Kumar [3] define validation as “the process of assuring that a model provides meaningful answers to the questions being asked.” Validation provides confidence that the approximations and abstractions used by simulation models do not substantially alter the answers to the questions being asked. The more accurate (and hence valid) simulation models are at predicting the systems they are modeled after, the more useful they are to simulation practitioners (though this must be balanced against performance, more accurate models will often be slower or have higher resource requirements).

Simulations should be backed up by laboratory and Internet experiments where applicable. Even with thoroughly validated simulation models there are many possible artefacts of simulation that can produce results inconsistent with reality (for example, phase effects[1] can be more common in simulations as shown by Floyd [5]).

Floyd also points out that even with the validation efforts put into the TCP models of ns-2 [5], the validation is not complete and there is less confidence in the models being valid for an arbitrary user of the simulator than the specific research carried out by Floyd. The models described are for the Transmission Control Protocol (TCP) in the network simulator ns-2, both of which are introduced in following sections.

1.3 The Transmission Control Protocol

TCP is the most common transport protocol used on the Internet, being used to transport web pages (with HTTP), email (with SMTP), files (with FTP) and much more. Internet measurements show TCP traffic as being prevalent; Cho, Mitsuya

[1]Periodicity and resonance in network traffic as described by Floyd and Jacobson [4].

and Kato [6] show that greater than 80% of all traffic measured on a backbone link between USA and Japan is TCP.

The importance of TCP is reflected by the wealth of research into it. It has been shown to interact poorly in some situations with several network types such as asynchronous transfer mode (ATM) networks [7], wireless networks [8] and fast long distance networks [9]. Many improvements have been suggested over the original specification [10], some of which have become standards [11, 12]. Generally, such modifications and extensions to TCP are tested in simulation before laboratory or Internet tests. A popular network simulator for TCP research is ns-2 [13].

1.4 The network simulator ns-2

ns-2 is a discrete event network simulator used widely in research of Internet protocols and mechanisms such as routing protocols, flow control and congestion control. ns-2 is written in C++ and uses the OTcl [14] language to control simulations. An example simulation script is shown in listing 1.1. In this example, a TCP flow transfers bulk data from the FTP application for 5 seconds over a full-duplex link with a bandwidth of 10Mb/s and a propagation delay of 34ms using the "drop tail" queuing strategy. A trace is written to the file "nam.tr" showing packet events such as enqueueing or dequeing at a router or host or packet drops.

Listing 1.1: Example ns-2 simulation script

```
set ns [new Simulator]
$ns trace-all [open "nam.tr" w]
# Create topology
set node1 [$ns node]
set node2 [$ns node]
$ns duplex-link $node1 $node2 10Mb 34ms DropTail
# Create TCP models
set src [new Agent/TCP/Newreno]
set sink [new Agent/TCPSink/DelAck]
# Attach models to the topology
$ns attach-agent $node1 $src
$ns attach-agent $node2 $sink
$ns connect $src $sink
set ftp [new Application/FTP]
$ftp start
$ns at 5.0 "$ftp␣stop"
$ns run
```

Simulator	Citations	
	Google Scholar	Citeseer
ns-2	1045	1275
GloMoSim	1042	97
OPNET	276	331
QualNet	62	25
x-Sim	38	9
ATM-TN	56	23

Table 1.1: Simulator popularity by citations

Listing 1.1 illustrates the level of abstraction provided by the models that are part of the ns-2 distribution. A researcher can specify the bandwidth and delay of a link and the queuing mechanism used, but no further detail is simulated: the physical aspects of the medium used to transmit data is not simulated, nor is the link layer. This is not a fundamental constraint imposed by ns-2, as several projects have extended the default models with, for example, models for a wireless radio channel. ns-2 has detailed models for routing, transport, and applications.

Though hard to quantify, it is probable that ns-2 is the most widely used network simulator for TCP research, and possibly the most widely used for general network research. Table 1.1 shows a rough measure of popularity: the number of citations for each network simulator. The two citation databases, Google Scholar [15] and Citeseer [16] were used to find the number of citations for each network simulator. The numbers here are only estimates because citations for the simulators are inconsistent; for example, ns-2 can be cited by its manual, website, or an early paper that describes it [17]. The citation systems are also incomplete, in some cases (such as GloMoSim) there is a large disparity between the two numbers. However, it can be seen that both citation systems report more citations for ns-2 than any other network simulator. Henderson *et. al* present similar findings, stating that over 50% of ACM and IEEE network simulation papers from 2000–2004 cite the use of ns-2 [18].

ns-2 contains a set of TCP models that can be used to simulate a variety of TCP features. The TCP congestion control algorithms of Tahoe, Reno, New Reno, Sack [11], and Fack [19] can be used. Each of the TCP models can be used in a one-way mode, in which data is transferred in only one direction (and acknowledgements flow in the opposite direction), or a mode which allows bidirectional communication like real TCP implementations.

The one-way TCP models have been studied in detail to validate them. Fall and Floyd [20] and Floyd [5] document a series of analysis based on looking at time-sequence graphs of TCP and checking packet traces manually for expected behaviour. Floyd [21] discusses how these tests are not comprehensive and that the validation tests are not necessarily effective for an arbitrary user of ns-2.

The TCP models are not designed to reproduce the behaviour of a specific TCP implementation and several aspects of the models do not match the behaviour of real implementations [5]. The one-way TCP models, which are the most often used due to their validation, deal in packets not bytes. The data and acknowledgement packet sizes can be specified, but as there is no two-way communication there is no provision for piggybacking of acknowledgements on data packets [5].

1.5 Real world code

In some cases part of the real system being studied can be integrated into simulation. This is possible if the original system is software and the code can be *directly executed* within the simulator. This provides for the possibility of very accurate simulation; the simulated model will respond to input using the same code as a real system.

There are several projects that use direct execution of real world code in network simulators. NsClick [22] allows Click Modular Router [23] routing protocols to run inside the network simulator ns-2 [13]—the same routing protocols that will run inside operating system kernels with Click support. An alternative way to simulate routing protocols accurately is proposed by Dimitropoulos and Riley [24] who integrate user-space routing software into the ns-2 simulator to form a feature full model of the Border Gateway Protocol (BGP). A similar project is InetQuagga [25], a port of the Quagga routing software to the OMNeT++ simulator.

Directly executing real world protocol implementations as models in network simulators has desirable benefits:

- the protocol model is likely to accurately match the real world implementation;
- the model may be more feature full than simplified simulation models; and
- the implementation has already been written and tested, the same process of building a simulation model and debugging it is not required.

There are difficulties involved in building simulation models from real implementations. The environment the code would originally execute in is changed to a network simulator, there are problems supporting multiple concurrent instances of the protocol, many protocol implementations are part of an operating system kernel and need to be removed from the operating system to be part of a network simulator. These issues are discussed in the following sections.

1.5.1 Multiple instances: the re-entrancy problem

A network simulation will create a number of instances of a model. For example, with a model of TCP, at least two instances of the model are needed for any sort of TCP communication—both endpoints of the TCP connection are required. Thousands or more instances of a model may be required in one simulation (see chapter 2 for a discussion of scale used in network simulation). However, real protocol implementations are generally not designed to work independently within a larger system. Network stacks are often only built to be part of the operating system kernel. In most cases, the code is not *re-entrant*: the code cannot be safely interrupted, re-entered to perform another task, and then resumed on its original task without side effects.

A mechanism is required to allow the code to run concurrently with many instances and no data shared between the instances. This process is sometimes known as virtualisation. There are a number of ways in which a code-base can be virtualised. For example, the real world code can be separated into different processes and inter-process communication used between the simulator and model instances. This means the operating system provides the virtualisation, each model instance is an independent operating system process. Another method of virtualisation is modifying the source code so all accesses to global data refer instead to data specific to the current thread of execution.

1.5.2 Kernel code in user space

Some network protocols are implemented in the operating system kernel. This is true of TCP: common operating systems such as Microsoft Windows, Linux, Solaris, FreeBSD and Mac OS X include TCP implementations within the operating system kernel.

To simulate with one of these protocol implementations the code must either run in user-space with the simulator or be modified to allow interactions with the simulator. Kernel code includes functionality that cannot run in user-space; for example, kernel memory management. The functions for these mechanisms need to be removed, and user-space implementations of these functions need to be written and used in their place. In some cases this is simple: the kernel memory management can be replaced with the call to the user-space function `malloc`. In other cases more detailed implementations are required.

1.6 Problem statement and scope

An examination of the following statement is presented in this dissertation:

> The accuracy of simulation of TCP can be improved by using real world TCP implementations instead of protocol abstractions.

Real world TCP implementations are software generally written in C. The implementations can be changed to work inside a network simulator instead of the normal environment, which is generally an operating system kernel. The implementations can then be used to produce very accurate results as the exact same code that runs on a real system runs as a simulation model.

Simulating with more than one real world TCP implementation is important. Each TCP implementation differs due to a number of reasons: the TCP specifications are ambiguous in places, there are many options left to implementers on what features are implemented, not all TCP implementations conform to specification, and the implementation may have bugs. We hypothesise that simulating with multiple different real world TCP implementations is required to answer the stated question.

This question is addressed by considering the following subcomponents:

Feasibility Using real world TCP implementations in a network simulator is theoretically possible, but it needs to be shown that doing so is practical. For simulation with such models to be useful, it is necessary that the models can be used to carry out research to the scale used by current TCP researchers and be run in reasonable time. This is investigated by reviewing current TCP simulation models and uses of TCP in research in chapter 2, describing an architecture and implementation that allows real world code to be used for TCP models in chapter 3, and using this implementation in TCP simulations. Chapters 4 and 5 show simulations carried out and an investigation of performance is shown in chapter 6.

Validity and accuracy For the results of simulation to be useful, they must be valid and accurate. This is investigated in chapter 4.

Applicability We hypothesise that simulating with multiple different real world TCP implementations is useful. This hypothesis and the idea that simulating with real world TCP implementations provides further knowledge to the simulation practitioner are analysed in chapter 5.

Performance and scalability The performance and scalability of using real world code as TCP models is discussed in chapter 6.

TCP research itself is discussed first in chapter 2. This is followed by descriptions of the simulators and their TCP models which are used to carry out TCP research. Existing approaches to real world code based simulation of TCP are also covered in this chapter. Following this chapter, the next chapters discuss and use the Network Simulation Cradle (NSC), a project created during this PhD. The NSC itself is described in detail in chapter 3. The accuracy of simulating with real world TCP implementations is analysed in the following chapter, chapter 4. Further experimental results are discussed in chapter 5, while performance and scalability is covered in chapter 6. The conclusions to the thesis are presented in chapter 7.

Chapter 2

Simulating the Transmission Control Protocol

This chapter presents a discussion on the types of research that use TCP simulation and the simulators used, including both simplified TCP models and real world code based TCP models.

Section 2.1 presents types of research that are carried out that use TCP simulation. This survey of research highlights the type and scale of problems which are investigated with network simulation and the network simulators used. This is followed by section 2.2, which describes the network simulators mentioned in the previous section in more detail. For each simulator, the TCP model is discussed and the validation performed covered. These two sections show how first the scope of simulation that a real world code based approach needs to address, and secondly the limitations of the state of the art in network simulation.

Using real world code for TCP simulation is covered in section 2.3, where approaches to using real world code are categorised and covered. The three categories—porting a TCP implementation to a simulator, modifying the host operating system, and building a supporting framework—are discussed in detail based upon the existing projects that use each approach. The chapter is summarised in section 2.4.

2.1 Research using TCP simulation

Much networking research involves studying TCP, or situations involving TCP, using network simulation. The following sections give examples of the types of research that are carried out. Research involving TCP simulation can be broken into three groups: studies of TCP under various conditions, modifications to the TCP algorithm, and simulation of TCP for analytical model validation. Each of these groups is reviewed with examples of research representative of the type of studies carried out in each area.

2.1.1 TCP under differing conditions

Many of the modifications to TCP are designed to improve performance in situations where TCP has been shown to be lacking. The initial research will involve a study of TCP in a specific set of scenarios or over a different network setup. Many of these studies are conducted partially or fully in simulation. In other cases the network itself is being studied and a TCP simulation model is used to generate realistic traffic on the network.

TCP performance over ATM networks is well studied [7, 26–28]. TCP can experience performance problems in ATM networks due to protocol conversion overheads, size mismatch between TCP segments and ATM cells, transmission errors and subtle interactions between the two protocols. A study of TCP over ATM on lossy ADSL networks [27] was carried out with the ATM-TN [29] network simulator. Another study carried out with the same simulator investigated the basic problems the two protocols have coexisting [26]; Gurski and Williamson showed how TCP was not able to utilise high bandwidth links effectively in their simulated network (a dumbbell topology with between 1 and 10 TCP flows).

Wireless networks are another common network in which some researchers have shown TCP to have performance problems. Many studies show how TCP performance degrades over wireless links and how TCP is unable to fully utilise the network resource available. One area of wireless network research is satellite systems; Obata, Ishida, Funasaka and Amano [30] present a performance analysis of TCP under such a system based on ns-2 simulation results. Another area is wireless protocols used by cellphone and other highly mobile technologies; Bai *et. al* [31] analyse TCP performance over CDMA-2000[1] wireless links using the OPNET simulator to simulate a single TCP flow between a wireless node and a base station.

TCP performance over multi-hop wireless networks is analysed by Gerla, Tang and Bagrodia [8] using results from simulations performed in the GloMoSim simulator with between 1 and 20 TCP flows over wireless networks ranging from 8 to 100 nodes. Further work on the same subject with GloMoSim is presented in [32]. An analysis of the TCP performance of a single flow over mobile ad hoc networks that uses the ns simulator is performed by Holland and Vaidya [33]. TCP throughput and loss is measured using up to 20 flows over multi-hop wireless networks in the ns-2 simulator and results are presented in [34]. Kuang and Williamson [35] develop a multi-channel MAC protocol for multihop ad hoc wireless networks and present results of simulations in ns-2. Simulations of TCP fairness using a custom queueing system for ad hoc wireless networks using the QualNet simulator and up to 6 TCP flows are reported on in [36].

Krishnan and Sterbenz [37] measure TCP throughput over load-reactive links: network links that have different properties as load increases (such as links controlled by dynamic quality of service scheduling). Simulations are performed in ns-2. Analysis of TCP undergoing denial of service attacks is presented in [38]. Simulations use the default TCP models in ns-2 and some measurements of the denial of service attacks are also performed on the Internet. Neglia and Falletta [39] mount the argument that packet reordering is not always harmful to overall TCP network performance based on results from ns-2 simulations and a theoretical justification.

TCP simulation is often used when investigating queueing mechanisms. Guo and Matta [40] present simulations in ns-2 of short and long-lived TCP flows through routers employing RED and ECN. Eddy and Allman [41] compare RED mechanisms using ns-2 TCP and FTP models using simulations on a dumbbell

[1]CDMA is Code Division Multiple Access, CDMA-2000 is a set of protocol standards for CDMA-based mobile communications.

network with 5 TCP flows. The adaptive RED algorithm is investigated with the ns-2 TCP models by Floyd [42] using simulations of up to 100 long lived TCP flows. ns-2 TCP models are used to test a new active queue management algorithm with simulations of up to 200 TCP flows in [43].

2.1.2 TCP modifications

Some research that uses TCP in simulation modifies the TCP protocol itself. Examples of the modifications possible include modifying the congestion control algorithm [19], changing the startup procedure [44, 45] or incremental improvements to address a problem TCP has in a particular scenario [46].

Packet reordering can be very harmful to the performance of TCP [47] and there are approaches to alleviate this that require modifications to TCP implementations. One approach is an extension to selective acknowledgements called DSACK [48], or duplicate selective acknowledgements. Blanton and Allman [47] modify the ns-2 TCP models to implement DSACK and study TCP performance with different retransmission strategies using simulations with a single TCP flow over a dumbbell topology. Another study uses DSACK in the ns-2 simulator and modifies the retransmission timer estimator algorithm and fast retransmit algorithm to attempt to avoid false retransmissions [46] using similar simulation scenarios to Blanton and Allman.

TCP adapts to network bandwidth by initially increasing the speed of packet transmissions exponentially in the slow start phase until packet loss is detected. The packet loss can be large and the time in slow start long on high bandwidth-delay paths. Hu and Steenkiste [44] present a method that they called Paced Start that uses active measurement algorithms to estimate the available bandwidth for the TCP stream and they modify the TCP algorithm to use this information. This potentially means a TCP flow transitions into congestion avoidance phase quickly with less packet loss while still making use of available network resources. The ns-2 simulator is used to carry out simulations with an unmodified TCP model and a TCP model modified to include the Paced Start algorithm. The scale of the simulation varies from a dumbbell network with one measured flow and one background flow, to a dumbbell topology with 102 flows and another topology they call a “parking lot” which features 11 routers each with 12 connected nodes and the routers connected serially for a total of 66 flows. Some measurements from real networks are collected from user-space and in-kernel implementations collected on Emulab[2] [49] and on the Internet respectively.

Williamson and Wu [50] study TCP performance with their version of TCP modified to include information from other network layers (such as web document size). The scenarios studied are first tested in ns-2 with modified TCP models then over a wide area network using a modified Linux TCP/IP stack. The simulation scenario uses 10 servers and 100 clients, with the clients making many short-lived HTTP requests to the servers during the simulation.

Modifications to the TCP congestion control algorithms are common. The Forward Acknowledgement algorithm [19] (FACK) modifies the congestion control algorithm to keep extra state when selective acknowledgements are used. This extra state allows the FACK algorithm to accurately regulate the amount of outstanding data in the network which means that in some situations TCP is less bursty and better able to recover from loss. The research was carried out with the ns simulator and modified ns TCP models using a dumbbell topology with one or two TCP flows.

Another modification to the TCP congestion control algorithm is presented in [51]. TCP is modified based on the idea of predicting traffic—a new TCP called TCP-TP (TCP with traffic prediction) is created. The new algorithm is tested in simulation with ns-2 and with a FreeBSD implementation. The simulation study uses up to 300 TCP flows in a topology with multiple bottlenecks.

TCP Westwood [52] is another TCP congestion control algorithm. It uses bandwidth estimation techniques and is shown to work well over wireless links in simulations with a custom Westwood simulation model built for the ns-2 simulator. The simulations use mixed wired and wireless nodes in a simple topology with one TCP flow. TCP Westwood later changed to Westwood+ [53] that includes a better bandwidth estimation algorithm, Westwood+ is tested with an updated ns-2

[2]Emulab is a freely available testbed network with full control over the machines used for testing.

simulation model and on the Internet with a Linux kernel implementation. The simulations vary, using dumbbell topologies with up to 210 flows, multiple bottleneck topologies with up to 21 flows and satellite scenarios using up to 30 flows.

Many other TCP congestion control algorithms have been proposed, some of which are initially investigated in simulation. Examples include TCP Vegas [54], TCP Hybla [55] and TCP Veno [56]. There are also many high-speed TCP variants such as H-TCP [57], Scalable TCP [58] and FAST TCP [59]. The research into TCP Vegas, TCP Hybla and H-TCP uses simulations with the x-Sim and ns-2 simulators with between 1 and 16 TCP flows. These are only short lists of the different TCP congestion control algorithms that have been suggested, there are many more suggested to aid TCP in various networking conditions such as high speed or high bandwidth-delay paths, wireless paths, paths which include reordering or asymmetric paths.

2.1.3 Analytical model validation

Various analytical models of TCP have been built, many of which are compared against simulations when validating the analytical model [60].

Anjum and Tassiulas [61] build analytical models that suggest that Tahoe TCP performs better than Newreno TCP on a wireless link with correlated losses while Sack TCP is better again. Simulations in ns-2 simulating a single TCP flow are performed to back up their analytical models.

Analytical models of long-lived TCP flows are presented in [62]. The models are derived directly from the TCP finite state machine. The authors use ns-2 simulation to validate the model results, using simulations of up to 500 flows.

Streaming multimedia over TCP is analysed in [63] where analytical models of TCP are updated to support video streaming. The models are validated using ns-2 simulations and Internet measurements. Simulations with ns-2 are performed in [64] to determine information for an analysis of TCP using game theory; a dumbbell topology with 10 flows is used in their simulations.

2.1.4 Discussion

The simulation studies covered use up to 500 TCP flows, with most studies using between 1 and 200 flows. For a TCP model to be useful in these situations, 1000 TCP model instances would need to be supported in a single simulation (one for each endpoint).

Other types of network simulation can require much larger scales. For example, simulating the Internet would require a much larger number of nodes. Some research uses these large scale network simulations, for example routing algorithm studies generally use very large scale simulations [65]. The scale of these simulations varies but examples include 20,000 routers [65] and 13,173 routers [66]. TCP can be important in these situations, as the Border Gateway Protocol (BGP) routing protocol uses TCP to transfer data between Autonomous Systems (AS).

Three groups of TCP research were covered in the previous sections: simulating TCP under differing network conditions, simulating with a modified TCP model, and using TCP simulations for analytical model validation. Real-world code based TCP models are applicable to all of these areas.

Simulations that modify the TCP algorithms are potentially more work for the person carrying out the simulation with a real-world code based TCP model, as implementing the modification to TCP is likely to be much easier with a simplified model. However, such modifications are often tested on real kernel implementations in addition to simulation. The real-world code based TCP model therefore is useful to allow developing and debugging the modification to the actual network stack code in simulation. This can then be used in the original operating system as well.

When TCP is used in a simulation scenario and no modifications are made to the original algorithm, using a real-world code based TCP model is potentially no more difficult than using a simplified model. In this case greater accuracy can be gained at low cost. The same applies for situations where TCP models are used to validate analytical models.

Six simulators were mentioned in the previous sections: ns-2 (and its predecessor, ns), ATM-TN, GloMoSim, QualNet, OPNET, and x-Sim. This is not a definitive list of simulators used in TCP research, as the studies covered are only a sample of the large amount of research that uses network simulation. ns-2 is the most common simulator we encountered in the literature which is consistent with the findings of several others [67–70]. Each of these simulators has a different set of features in its TCP models and has undergone a different amount of validation. These simulators are covered in the next section, section 2.2.

2.2 Network simulators used in TCP research

The previous section presented a sample of research using TCP simulation. A number of simulators were used to carry out this research: ns-2 (and its predecessor, ns), ATM-TN, GloMoSim, QualNet, OPNET, and x-Sim. These simulators and their TCP models are discussed next. Each simulator is introduced, followed by a discussion of the features of the TCP model, then information on the validation that has been performed.

The simulators discussed in the following sections are not an exhaustive list of network simulators with TCP models. Many simulators have been used to conduct network research and information on further simulators can be found in appendix C. The simulators reviewed in this section show different approaches to simulating TCP and cover a wide amount of research, as shown in section 2.1.

2.2.1 ns-2

ns [13] is an object oriented discrete event simulator designed for network research. ns provides support and models for TCP, routing, multicast, wireless and wired networks. The initial release of Ns version 2 was in 1996 and subsequent versions of the simulator have come to be known as *ns-2*. ns-2 is the evolution of the simulator called *tcpsim* that was a version of the REAL simulator [71] based on the NEST simulation software [72]. ns-2 has been used for a large body of networking research, much of it related to TCP:
see [30, 35, 37, 39–41, 47, 50, 51, 53, 63, 64, 73–76] for a set of examples.

ns-2 is built on a C++ simulation kernel heavily integrated with the OTcl [14] interpreted language. OTcl is an extension to the Tcl [77] language for object oriented programming. OTcl is used to describe simulation scenarios and implement parts of some models. C++ objects are created and interacted with in OTcl simulation scripts to create simulation topologies and scenarios programmatically.

TCP model

There are two types of TCP model available in ns-2: one-way TCP models, which allow only unidirectional transfer of data, and two-way TCP models which allow full bidirectional communication. The one-way models are more thoroughly validated and used most often in published research.

Both sets of models are feature full and allow a range of congestion control algorithms to be selected. One-way TCP data sources can use Tahoe [78], Reno [79], Newreno [80], Sack [11], Fack [19], Vegas [54] and other congestion control algorithms. The one-way TCP sink (endpoint which only sends acknowledgements) can use delayed acknowledgements, selective acknowledgements, or acknowledge every packet. Two-way TCP models can use Tahoe, Reno, Newreno or Sack congestion control algorithms.

Full segmentation is not performed by the one-way TCP models, data packets are always full sized. No receiver’s advertised window is used; the receiving application is assumed to consume data as fast as it arrives. Much configuration is possible: the MSS, window size, TCP/IP header size, timer granularity, minimum retransmission time, timestamps and other options can all be configured.

Validation

The ns-2 simulator has a large set of validation tests for many protocols including TCP. The TCP tests run simulations and record statistics such as sequence number over time. To validate ns-2 each simulation scenario was originally analysed by hand then subsequent tests are checked against previously saved output. Test scenarios exist to check a range of features such as slow start, fast retransmit and congestion avoidance algorithms under differing amounts of packet loss. Other

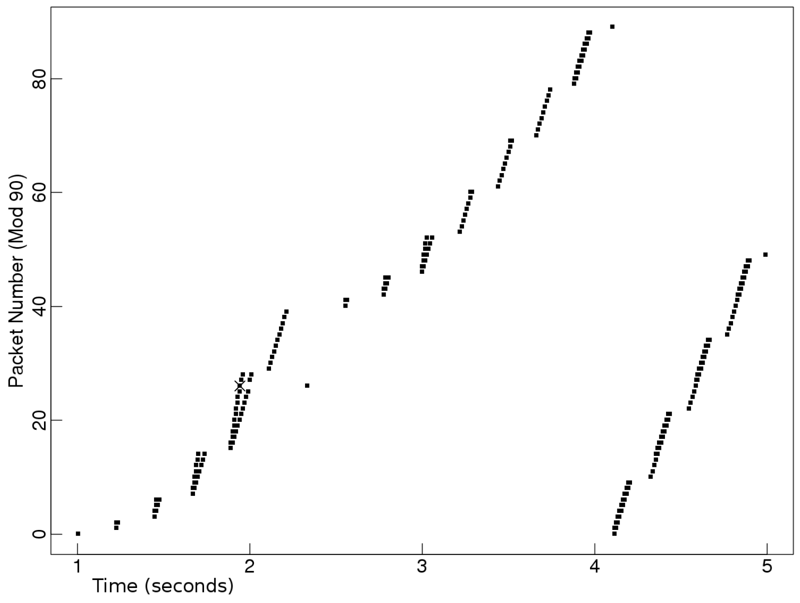


Figure 2.1: ns-2 simulation of TCP Tahoe responding to packet loss [5]

tests examine retransmit timers, delayed acknowledgements, fast recovery and selective acknowledgements [5]. The different congestion control algorithms are tested for both one-way and two-way TCP models under different amounts of loss in a similar fashion [20]. The two-way TCP models only have partial validation performed [81, 82], the suite of tests covering these models is not as comprehensive as the validation tests covering the one-way TCP models.

An example of an ns-2 validation graph is shown in figure 2.1. The graph is discussed by Floyd [5] when outlining the validation tests of TCP in ns-2. The graph "shows the Fast Retransmit, Slow Start, and Congestion Avoidance algorithms of Tahoe TCP". The cross on the graph indicates the single packet lost. The dots on the graph show each packet as it arrives and departs from the gateway.

This graph shows how initially TCP increases quickly until it gets a loss, then learns from this by adapting the slow start threshold after the loss. By analysing the graph it is evident that the TCP goes into slow start again after the loss, but quickly uses congestion avoidance due to the slow start threshold as the TCP stops increasing exponentially at approximately time=3.5 seconds. Graphs such as this one were initially verified by the TCP model author. The validation testsuite is then updated with the known correct packet trace, and the model is subsequently tested against this packet trace to ensure it stays correct.

2.2.2 ATM-TN

ATM-TN [29] is a network simulator originally designed to simulate ATM. The simulator is based on SimKit [83], a C++ library for high performance discrete event simulation.

TCP model

The TCP model used in ATM-TN is based on the Berkeley Unix BSD implementation known as Net/3 [84] developed by the University of California, Berkeley and released in April 1994. The TCP implementation is modified heavily from the original C implementation to port it into the C++ classes used by SimKit.

Gurski and Williamson [26] describe the TCP model in ATM-TN: it includes all the features of the Net/3 BSD TCP implementation including slow-start, fast retransmit, fast recovery, high-performance extensions (TCP window scaling and timestamps [85]) and full-duplex data communication.

Validation

The TCP model is validated by analysing graphs of TCP dynamics by hand and comparing results of a simulation with previously published research. Both methods are presented by Gurski and Williamson [26].

A validation experiment conducted by Gurski and Williamson showing analysis of TCP dynamics by hand is as follows. A single TCP source is configured to send data as fast as it can to a TCP sink. Between the source and sink an ATM switch is configured with a mismatch in link speeds between the incoming and outgoing links. It is expected from this scenario that the TCP source will exhibit cyclic behaviour: increasing its send window, filling the switch buffer, detecting a dropped segment, reducing the send window, and retransmitting. The TCP congestion window is graphed alongside transmitted cells and switch buffer occupancy. The graph of this experiment is shown in figure 2.2.

The graphs in figure 2.2 are analysed by Gurski and Williamson as follows. The TCP delayed acknowledgement option is evident due to acknowledgements being

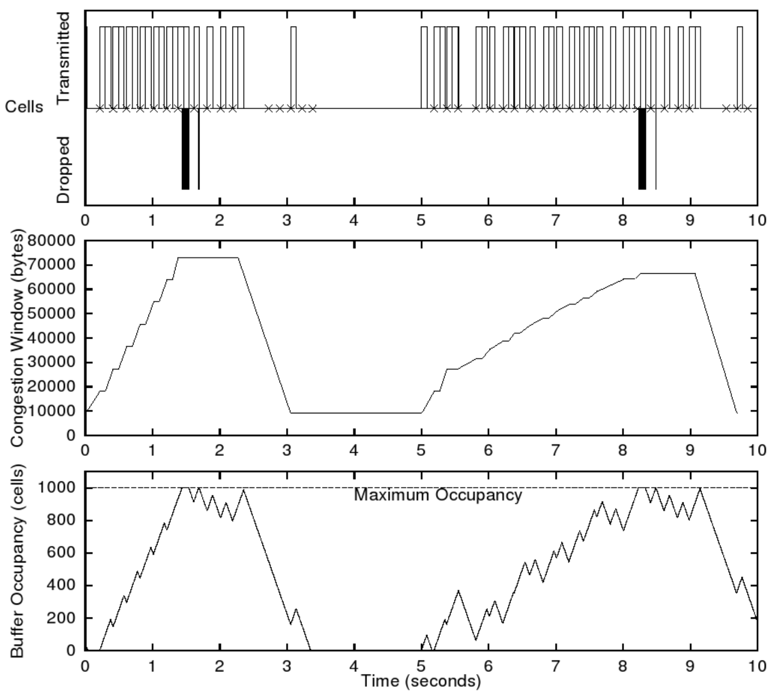


Figure 2.2: TCP dynamics on an ATM network [26]

spaced evenly. Slow-start is indicated by the fast growth of the congestion window initially where each acknowledgement results in one or two back-to-back data packets being sent. The cell sent at time=3 indicate fast retransmit in action: the '$\times$' symbol on the top graph indicates acknowledgements, this cell is retransmitted after three duplicate acknowledgements due to the fast retransmit algorithm (the authors use extra information along with the graphs to check that the acknowledgements were duplicates). The other cells are retransmitted at time=5 due to the TCP retransmission timer expiring. A similar analysis is provided by Gurski and Williamson [26] for TCP dynamics between two TCP streams.

A further validation experiment was conducted by replicating a simulation performed by Romanow and Floyd [7]. ATM switch buffer size is varied and the effective throughput of 10 TCP flows is measured. The results of the simulations did not agree exactly due to the complexities of reproducing the same simulation in a different simulator, though the general trends shown by Gurski and Williamson agree with those shown by Romanow and Floyd.

2.2.3 GloMoSim

GloMoSim [86], or Global Module system Simulator, is a library designed for parallel simulation of wireless networks. The library is implemented with the C-based Parsec parallel simulation language [87]. Models are written in this language, GloMoSim includes many wireless routing and MAC protocols as well as radio and mobility models. It also includes UDP, TCP and simple application models such as constant bit rate traffic generators.

GloMoSim has been used for much wireless research [8, 32, 88–90], some of it involving TCP [8, 32]. GloMoSim is no longer maintained and is succeeded by the commercial package QualNet [91, 92]. There is little information provided on the TCP simulation model[3] used in QualNet, the web-pages describe simple model features such as Reno and Newreno congestion control but do not mention the FreeBSD network stack that is used in GloMoSim.

TCP model

The FreeBSD 2.2.2 network stack is used as the TCP model in GloMoSim [8, 86, 93]. There is little information on the architecture used to incorporate the network stack into the simulation library. The TCP Tahoe model from ns-2 was ported to GloMoSim and is also available.

Validation

Bagrodia and Takai [93] state that the GloMoSim TCP model is validated against an operational prototype (a computer running FreeBSD 2.2.2) but do not describe this process in detail.

The TCP model ported from ns-2 was validated by comparing two similar simulation scenarios run in ns-2 with those run in GloMoSim: the results were not identical but "within appropriate statistical bounds" [93].

Further validation of the simulator and its models is provided by running scenarios of known results and checking the model output. A detailed event trace of the

[3]The company website provides some basic information (`http://www.scalable-networks.com/`).

model execution is analysed to ensure that it follows the expected path.

2.2.4 OPNET

OPNET Modeler [94] is a commercial object oriented network simulator. It is used in a large amount of networking research [67, 95] including many simulations involving TCP [31, 96–101]. OPNET provides an extensive Graphical User Interface that is used to build topologies and simulation scenarios, analyse data and create models.

OPNET Modeler is described as a high performance simulator capable of sequential, parallel, hybrid and analytical simulation. Many models are provided of protocols and applications including routing protocols, wired and wireless MAC protocols, transport protocols and others.

TCP model

The TCP model provided by OPNET Modeler is feature full. All basic RFC 793 [10] functionality is provided. Fast retransmit and recovery, selective acknowledgements, explicit congestion notification, Karn's algorithm, a receiver's advertised window and a persist timer are implemented. Window scaling is supported and TCP timestamps are used for RTT calculation.

Validation

The company that produces OPNET Modeler, OPNET Technologies Inc., does not provide validation or testing information on their simulator or models. However, there have been several independent tests of their simulator, including one comparing TCP dynamics between OPNET Modeler and ns-2 [67].

Lucio *et. al* [67] built a test bed network and designed several simulations in OPNET Modeler and ns-2. The scenarios tested used either constant bit rate traffic or FTP. Several simulation parameters were tuned for each simulator: the New Reno option, window scaling or window size, the TCP timestamp option and the maximum segment size. Each simulator supports a different set of options so these can not be configured to be identical for each simulator. The options were instead

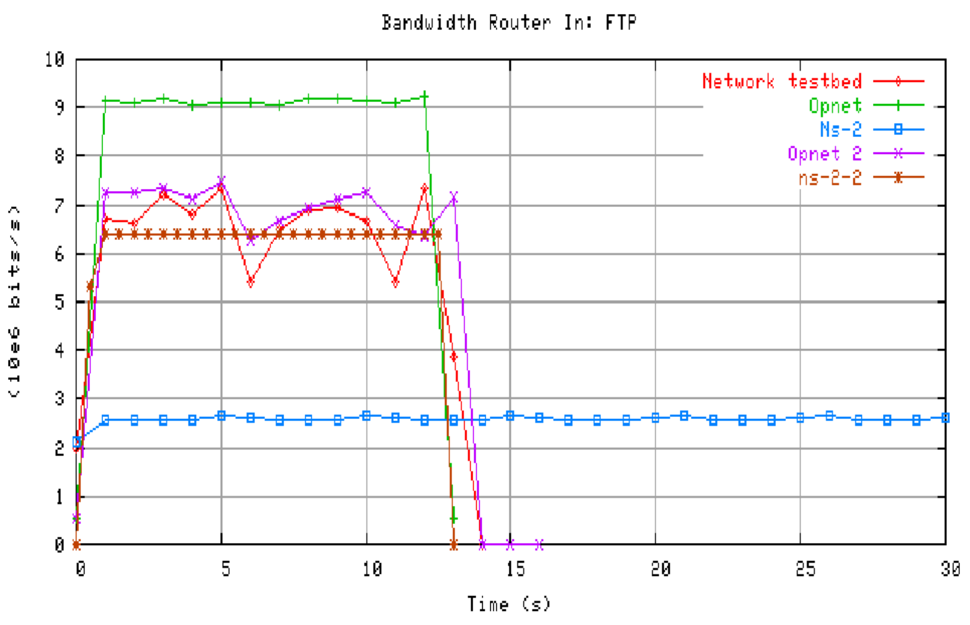


Figure 2.3: TCP throughput during FTP measured from a router [67]

tuned to be as close to the test bed network as the researchers could make them for each simulator. The network topology used is a dumbbell topology with two flows. The bandwidth is measured at several locations in the network and graphed over time. An example of a graph used for analysis is shown in figure 2.3.

The metric of throughput over time is used for all comparisons by Lucio *et. al*. Figure 2.3 shows one of the comparisons. The graph shows how the researchers tried different options for ns-2 (the lines marked as ”ns-2” and ”ns-2-2”) before they were able to produce results similar to the results from their test bed network. This result illustrates how the ns-2 TCP models do not necessarily model reality when used with their default configuration. A similar process was applied for the OPNET models, though the initial results without tuning show the correct trend, while the ns-2 results do not. In this case OPNET is a close match to the results from their test bed network.

Lucio *et. al* conclude that OPNET and ns-2 provide very close results but that both simulators did not model FTP well with their default parameters. OPNET produced results closer to measurements from a test bed network when fine-tuned to simulate the FTP scenario.

2.2.5 x-Sim

The x-kernel [102] is an operating system kernel designed to facilitate the implementation of efficient communication protocols. It provides an explicit structure and support for protocols. The x-kernel includes a large base of x-kernel protocol implementations such as TFTP, DNS, UDP, TCP, Sun RPC, IP, ARP, ICMP and more. The TCP implementation used in the x-kernel is a direct port of the 4.3 Berkeley Unix TCP/IP stack.

TCP model

x-Sim [54, 103] is a simulator that uses the x-kernel for protocol implementations. Simulations performed with x-Sim can therefore use the Unix TCP implementation that is part of the x-kernel.

The TCP stack is hand modified to fit into the new kernel architecture. The custom simulator is able to run many kernels and route messages between the kernels, enabling many instances of the TCP stack to be simulated. Any x-kernel protocol (everything is a protocol in the x-kernel, even applications) can be run in x-Sim, meaning real applications and a real network stack can be simulated, although the applications and network protocols must be ported or implemented for the x-kernel architecture first.

Validation

Some validation work has been done to make sure the results generated with x-Sim are consistent with expectations, though no information is given on the details of this, or whether porting the TCP implementation to the x-kernel changed the behaviour of the implementation at all. The one TCP implementation is available for simulation and is dated: the version used is from around 1990.

2.2.6 Discussion

Only a subset of network simulators is covered here, but all previous six simulators have been used for published research about TCP; each was used in research discussed in section 2.1. While many other network simulators exist, there is a large amount of network research which uses the simulators discussed. Lucio *et. al* [67] choose OPNET and ns-2 for their simulations "because of their popularity with academia, commercial and industrial communities". Breslau *et. al* [17] describe ns-2 and OPNET as prominent examples of network simulators.

Of the simulators covered, ns-2 has the most comprehensive validation suite. While validation studies have been performed with the other simulators, none was found to have as many tests or such a large framework in place to ensure the correctness of their models.

ATM-TN, GloMoSim and x-Sim all use real world TCP code. However, this alone does not guarantee that the TCP model is accurate. The approach of using real world code for a TCP model and its limitations are covered in the next section.

2.3 Simulating with real world TCP code

Three of the simulators introduced in section 2.2 use real world TCP code as a TCP simulation model. Other simulators not covered earlier also do this: NCTUns [104], dONE [105], OMNeT++ [106, 107] and IRLSim [108] all include, or have extensions for, real world TCP code.

The properties of such models are different to simplified models built specifically for simulation. The original TCP implementation will implement applicable RFCs by necessity: the TCP implementation must be able to communicate with other TCP implementations. The list of features available in a real TCP implementation is often quite different to a simulation model of TCP.

A full featured TCP stack is normally available in simulation if a real TCP implementation is used. However, there are limitations and problems inherent in taking this approach.

Hand modifying code To integrate the implementation into the simulation system, some amount of modification will be required. This can be a lengthy and difficult process, prone to error.

Keeping up to date TCP implementations that are part of operating systems are updated frequently and have bugs fixed and features added. Once a implementation has been added to simulation it can be difficult to keep up to date.

Validation While the original TCP implementation is known to work, it still needs to be tested that it works correctly in the simulator.

Multiple instances Network stacks are generally designed to be run as a single instance per computer. Simulation requires many instances of a model. A methodology is required to support multiple copies of a TCP implementation running concurrently and independently.

Multiple implementations Ideally a TCP researcher can choose which TCP implementation to simulate with, rather than be limited to one single TCP implementation that may be limited in features or known to include bugs. Both real world implementations and simplified models should be present as real world models are not always applicable: Floyd [21] points out that very detailed models can heavily skew results in some situations where only a coarse grained simulation is required.

In some cases using simplified models is preferred; initial development of a TCP modification is probably faster with a simplified model and when creating results for an analytical model only a coarse degree of accuracy might be required. In other cases a coarse-grained model is desired, Floyd [21] discusses how fine-grained models are not appropriate to all research due to their interactions possibly skewing results. It is therefore useful for a simulator to allow both simplified TCP models in addition to real-world code based TCP models.

The existing research into using real world code for a simulated TCP model can be categorised into three approaches:

- Porting the TCP implementation alone into a new framework. For example, an early BSD TCP implementation is ported to a C++ simulation library, SimKit [83], for the ATM-TN [29] simulator. BSD TCP implementations are ported to the Parsec [87] simulation language for the GloMoSim [86] and IRLSim [108] simulators. x-Sim [54] is a port of a BSD TCP implementation to the x-kernel [102], an operating system which can then be simulated.

- The existing operating system can be modified to allow the network stack on the simulation machine to be used for simulation. This approach is taken in the NCTUns [104] project.
- A framework can be built around a network stack that features as a bridge between the network stack environment and simulation environment. This approach is taken with dONE [105] and the FreeBSD extensions to OMNeT++ [106]. This approach is also taken by Wei and Cao [109], although they only include the TCP congestion control algorithms, not a full network stack.

The three different approaches are discussed next.

2.3.1 Porting the TCP implementation

The TCP implementations in this category have been modified to incorporate them into the new system. The projects covered in this category (ATM-TN, GloMoSim, IRLSim[4] and x-Sim) can be further arranged into those which incorporate the TCP implementation directly as a TCP model in a simulator, and those that incorporate the TCP implementation into a simulated operating system kernel.

ATM-TN, GloMoSim and IRLSim take the first approach: the TCP implementation is modified to make it compile as part of the simulator. For ATM-TN, an early BSD TCP implementation is modified from C source code to implementations of SimKit C++ classes. The basic structure of the TCP processing code stays the same, but many modifications are required to move functions and global variables into C++ classes and to modify the C code to compliant C++ code.

IRLSim and GloMoSim use the Parsec simulation language, which is a language similar to C. Terzis *et. al* [108] state that porting C to Parsec is simple but to fully port a TCP implementation, many modifications are required: global variables need to be modified and all interactions with the operating system need to be modelled with Parsec entities and messages. For example, this can be seen in version 2.03 of the GloMoSim source code [110] which includes modifications

[4]IRLSim is described in section C.6 on page 175.

throughout the TCP code. Many functions are modified to take an extra parameter identifying state specific to the particular GloMoSim node being simulated. In some places where functions were called previously, Parsec messages are instead constructed and sent.

The approach taken with x-Sim is somewhat different. The TCP implementation is ported to the x-kernel, a different operating system. The x-kernel operating system is a full operating system kernel like the original BSD kernel the TCP implementation was copied from. The x-kernel is able to be simulated and the simulation framework provides the facility to instantiate multiple independent instances of the kernel (and hence TCP implementation).

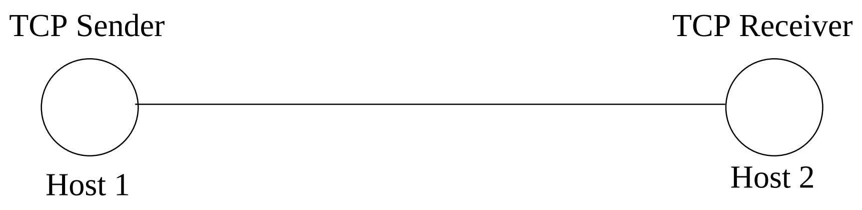
The code for version 3.3.1 of the x-kernel and associated x-Sim is available [111] and it is evident that a similar amount of source code needs to be changed to port a BSD TCP implementation to the x-kernel as it does to the other simulators discussed here. x-kernel specific functions are called when the code would interact with the operating system and extra code is added for event tracing.

The tight integration between network model and simulator has some benefits. Event tracing, statistics gathering and configuration are easy to integrate into software that was not originally designed to be used in a simulated environment. Gurski and Williamson [26] note this is true of ATM-TN, it allows simple modification of a range of parameters from all layers of the protocol stack: the socket, TCP/IP and ATM are all controlled by a set of options and parameters. Extra tracing function calls are added to the TCP implementations in x-Sim that provide, for example, in-depth information on the TCP control state during processing of TCP segments.

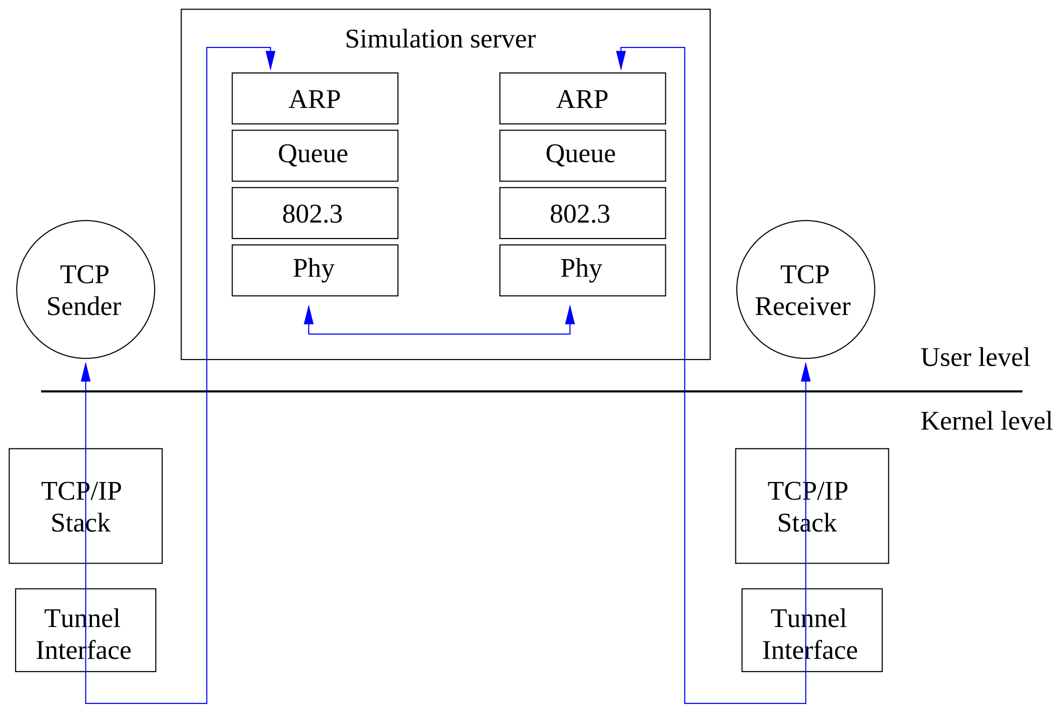
2.3.2 Modifying the operating system

It is possible to modify the operating system run on a computer such that a user-space simulation program can make use of the running kernel. This approach is used in the NCTUns [104] project.

Tunnel devices are available on most UNIX machines and allow packets to be



(a) Example simulation scenario



(b) NCTUns architecture used to simulate the scenario

Figure 2.4: NCTUns simulation architecture (adapted from [104])

written to and read from a special device file. NCTUns uses the local machine's network stack via a tunnel network interface. 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 interface's output queue is copied to the reading application. An example of this architecture being used for a simple simulation topology is shown in figure 2.4.

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—by NCTUns version 3.0 support for FreeBSD was dropped.

A separate computer is needed for every different version of every operating system that is to be simulated and the computer must be installed with that operating system. This means larger simulations could require many machines; the resource requirements are higher than a simulation run in network simulator using simplified models.

2.3.3 Build a supporting framework

Several projects aim to minimise code modifications to the TCP implementation, thereby reducing the chance of inadvertently changing the behaviour of the implementation. To integrate the TCP code with the simulator, a framework is built that bridges between the TCP implementation and the simulator.

The architectures used by the dONE [105] simulator's TCP model [112] and the FreeBSD TCP extensions [106] to the OMNeT++[5] simulator are based on the Alpine [113] project. Ely, Savage and Wetherall [113] describe moving a TCP implementation from kernel-space to user-space with minimal modifications to the original code. They moved the FreeBSD network stack into user-space to aid network protocol development and testing. Their architecture is depicted in figure 2.5.

This architecture shows a framework supporting a network stack that is unmodified. The framework was used to run the network stack in user space and to send packets out to the network in the Alpine project, although the same design can be used to support a network stack that is to communicate with a network simulator.

[5]OMNeT++ is described in section C.1 on page 171

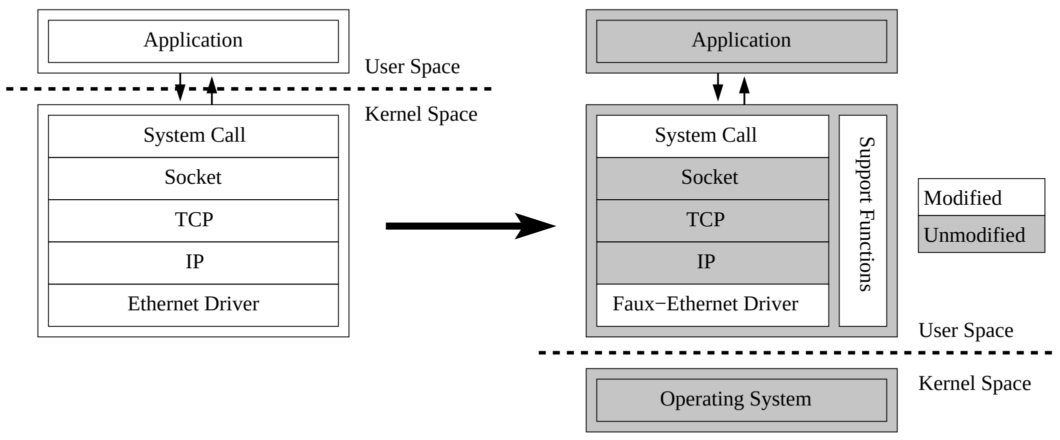


Figure 2.5: Kernel network stack in user space (adapted from [113])

Wei and Cao [109] take a different approach: only the code which implements the TCP congestion control algorithm is included in their framework. ns-2 TCP-Linux is a project where the TCP congestion control algorithms from recent versions of the Linux kernel are incorporated into the ns-2 TCP models. This approach allows scalable testing of Linux TCP congestion control algorithms without modifying the original real world code. The approach is also much more limited as the normal limitations of simplified models still apply: no extra functionality is added to the ns-2 TCP models.

dONE and Lunar

The Distributed Open Network Emulator [105], is a distributed hybrid emulation and simulation framework that includes the Lunar [112] software. Lunar is a project that ports the Linux 2.4.3 network stack to user space and makes it available to be linked in to a simulator. Lunar uses the Weaves [114] framework to support multiple instances.

The Linux 2.4.3 network stack in Lunar is moved to a user-space library by isolating the network stack from the kernel code, providing stub functions to implement missing identifiers and providing custom implementations for small amounts of kernel functionality. The stub functions do nothing, they are included to satisfy the linker. This is the basic methodology used in other projects such as Alpine [113].

Multiple instances of the Linux network stack in Lunar are supported by using the

Weaves framework. Weaves provides a multi-threaded environment in which many virtual hosts can run protocol stacks and applications as a single operating system process. Weaves provides each virtual host with a separate memory and namespace for its global and static variables by rewriting binaries. Bergstrom, Varadarajan and Back [105] note that the overhead of using Weaves is small.

Only basic validation testing of dONE and Lunar are described in [105] and [112]. dONE is shown to correctly simulate the trend of TCP goodput with increasing bandwidth-delay products. Some verification testing has been performed on Lunar. It has been shown to correctly transfer data by testing different reading and writing mechanisms of the network stack.

OMNeT++ extensions

The TCP models provided with OMNeT++ have limited features and are not thoroughly validated, as explained in section C.1 (which describes OMNeT++ in more detail). Bless and Doll [106] incorporate the FreeBSD network stack into OMNeT++ to solve this problem. They use a real world TCP/IP stack to avoid "possible implementation errors and costly validation tests". Figure 2.6 shows a view of the architecture used to incorporate the FreeBSD network stack into the simulator.

The FreeBSD 4.9 network stack is modified by hand to support multiple instances. The global variables in the source code are changed one-by-one; the authors found that a simple search and replace was not enough to handle the complexities of modifying global variables. They implement their own timer mechanisms to improve performance rather than rely on the kernel implementation that is based on a software interrupt mechanism. The routing table is also managed to allow using the FreeBSD network stack as a router as well as an end host for TCP connections (routes calculated in the simulator ar injected into the kernel routing table). They achieve scalability of around 1000 TCP connections transferring data concurrently.

The TCP code is not modified, apart from the global variable modifications, and because of this Bless and Doll conclude that they do not need to "test all potential error cases". Only minimal validation is performed. While it is true that the TCP

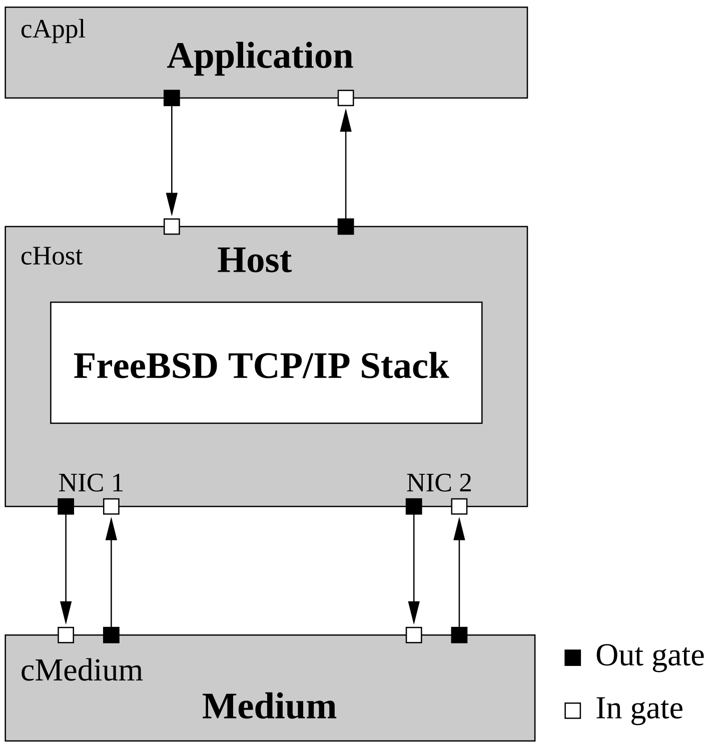


Figure 2.6: FreeBSD network stack in the OMNeT++ simulator [106]

model uses code from a real, well tested, TCP implementation, there is still possibility of introducing error into the model when modifying it to run in user space and in simulation.

Another project uses the NetBSD network stack in the OMNeT++ simulator [107]. At the time the work was carried out, the TCP models distributed in the networking framework of OMNeT++ were known to work incorrectly [115]. A validated TCP model was required to test the Message Queue Telemetry Transport protocol (MQTT) over lossy links, so Julio [107] used the NetBSD network stack for the TCP model in OMNeT++. Little information is provided in the process used to move the stack into simulation; there is no discussion of supporting multiple instances of the network stack. A small set of validation tests was run, comparing results on a testbed network to results from simulation.

2.3.4 Discussion

Three approaches to using real world TCP code in a network simulator are introduced at the start of section 2.3. Porting the TCP implementation to the simulator, modifying the operating system running on the simulator computer and building a supporting framework around a TCP implementation.

Porting a TCP implementation to a network simulator normally requires large changes to the TCP implementation. With many changes to the original system, it is difficult to add new TCP implementations or update the existing one. Confidence in the simulation model producing correct results is lower than the other two approaches as the changes to the original system are more substantial.

Making small modifications to the operating system to support simulation of the TCP stack is the approach taken in the NCTUns [104] project. This makes supporting different TCP implementations difficult, as a computer is required for each different version of each TCP implementation. Installing a modified operating system kernel is also required for each simulation machine.

A supporting framework can be built around a TCP implementation, allowing the implementation to be run in a new environment—a network simulator rather than an operating system kernel—with few code modifications. With little or no code changed and a framework in place, it is conceptually easy to update the TCP implementation supported and update the framework to add new TCP implementations. This is not the case in practice for the TCP model added to OMNeT++ by Bless and Doll [106]; to support multiple instances they make many hand modifications to the original code. dONE [105] uses the approach of binary rewriting (using the Weaves [114] project) to support multiple instances, but the approach is not extended to multiple TCP implementations or versions.

Five factors desirable for a real world TCP implementation used as a simulation model were introduced at the start of section 2.3. No hand modification of code, ability to keep the model up to date, validation, support for multiple instances and support for multiple implementations. The method of building a supporting framework is the closest to being able to satisfy all of the requirements but none of the projects covered do so. Only one simulator is able to support simplified TCP models and real world code; OMNeT++ with extensions real world code extensions by Bless and Doll [106].

2.4 Summary

This chapter reviews TCP research that uses network simulation, the network simulators that are used in this research, and using real world code for TCP simulation. The TCP research is broken down into three areas in section 2.1: simulations involving modifying the TCP algorithms, simulations that use TCP in a specific scenario, and simulations that used to validate analytical models. The scale of simulations used in this research ranges from simulations using under 100 TCP flows to simulations using 500 TCP flows. Other areas of network research such as routing research use network simulations of much larger scale.

Network simulators used to simulate TCP in the research covered include ns-2, ATM-TN, GloMoSim, OPNET and x-Sim. The amount of validation varies a lot between these simulators. ns-2 has the most comprehensive validation framework and feature full TCP models. Even ns-2 has some major limitations: for example, the TCP models generally used in ns-2 do not support bi-directional transfer of data. ATM-TN, GloMoSim and x-Sim all have TCP models built on real world code implementations. All are based on old BSD TCP implementations and have not been updated as the TCP implementations have evolved.

Approaches to using real world code are categorised in section 2.3: a TCP implementation can be ported into a simulator, the operating system can be modified to support interacting with a simulator, or a framework can be built that bridges between a real world TCP implementation and a network simulator. None of the projects which are covered in this section provide all of: multiple real world TCP implementations, integration with an existing simulator to allow ease of use and ability to use existing simplified TCP models, multiple TCP instances without much hand modification of code and thorough validation.

The next chapter describes the architecture and implementation of the Network Simulation Cradle (NSC), a project designed to take advantage of real-world code based TCP models while complementing the existing models in a network simulator. The NSC provides all of the features listed above and is scalable enough to simulate the research covered in section 2.1.

Chapter 3

The Network Simulation Cradle

The Network Simulation Cradle (NSC) is software designed to run real world TCP implementations in a network simulator. The NSC supports multiple versions of multiple different operating systems simulating many TCP connections simultaneously. This is achieved by a combination of the type of architecture presented in section 2.3.3, an approach that uses shared libraries to differentiate different TCP stacks, and programmatic modification of source code to support multiple independent instances of the TCP stacks. The world “cradle” is used to describe how the real world code is supported inside this framework: a cradle is built about the code that allows it to run in a different environment — a network simulator instead of an operating system kernel.

The construction of the Network Simulation Cradle shows that it is feasible to build software that accurately simulates multiple real world TCP implementations. This chapter presents the design that makes this possible and a discussion of how the detailed goals below are achieved. Chapters 4, 5 and 6 show results of this software providing accurate, applicable and scalable simulation of TCP respectively.

The Network Simulation Cradle is designed to meet goals set out in chapter 1: it needs to be valid, accurate, scalable, and able to carry out the sort of simulations TCP researchers perform (as discussed in chapter 2). These goals are discussed in detail below:

Simulate real world code real world code must be used as the code for TCP simulation models.

Utilise network simulators existing network simulators should be able to be utilised and support for at least one popular network simulator must be built. This means that a trusted network simulator can be used and should facilitate simulating previous simulation scenarios.

Perform and scale well the code must perform adequately to run simulations similar to existing research in reasonable time. Many instances of TCP endpoints need to be supported. This is required to simulate scenarios where, for example, background traffic is simulated with many TCP flows.

Produce accurate results the stacks being simulated must produce results which are very similar to real computers running the stacks—the NSC needs to be valid.

Be easy to update adding new stacks to the system should be possible and updating existing stacks to new versions should take a minimal amount of time. New versions of operating systems, and hence real world TCP stacks, are released over time and the versions that are installed by users of a network change to reflect this. Ease of update aids in supporting the versions of the real world code that are practically used.

Support different methods of statistics gathering different TCP variables should be able to be accessed and traced to view what TCP is doing internally. Transparent access to TCP internals is important for simulation researchers and this feature is available in existing simplified TCP models.

Allow a full range of TCP simulation scenarios the NSC should work in situations existing TCP models do, allowing a full range of simulations to be performed.

Complement simplified models the real world TCP stack should work alongside traditional simplified models in a simulator. This allows easy comparison of both models, which helps validation testing.

Two components form the basis of the NSC: a simulator model and a TCP implementation. The simulator model component routes simulation messages to and from the TCP implementation via a standard interface. The TCP implementation and supporting code is contained in a shared library.

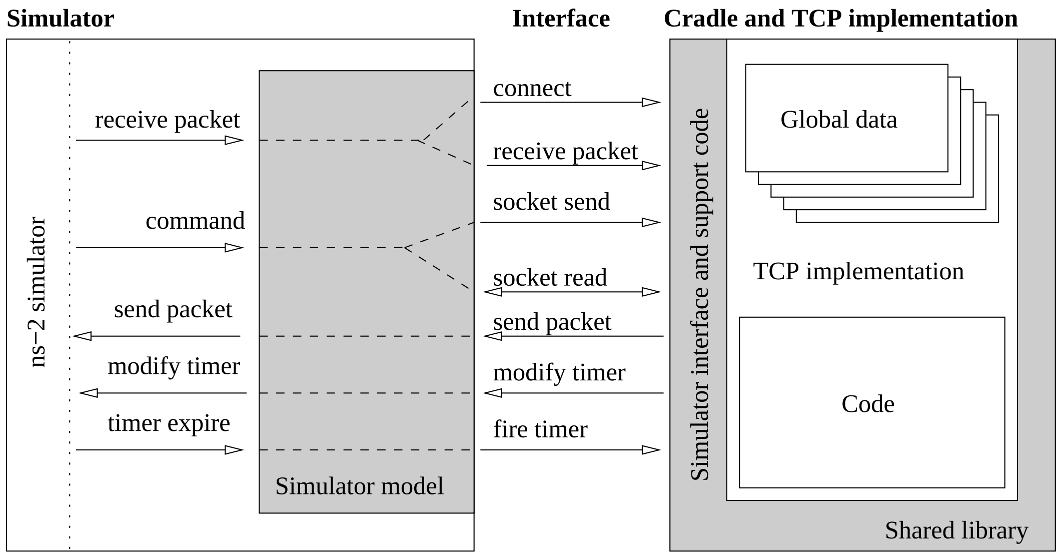


Figure 3.1: Simulator and Network Simulation Cradle interactions

Figure 3.1 outlines the interactions between the simulator model and each TCP implementation. The block on the left shows the network simulator and example interactions with the simulator model. On the right is the simulation cradle with a real world TCP implementation. The parts of the diagram coloured grey indicate areas where new code is written for the Network Simulation Cradle, the areas with white backgrounds indicate existing software. In between the two components some interactions are shown. The components communicate with a C++ interface exported from the shared library.

Only a subset of the actual interactions are shown in this figure for brevity. In this figure the network simulator ns-2 is used as an example simulator. The design of NSC allows for other network simulators to be used although this chapter describes only the integration with ns-2 in detail. The figure shows a standard set of interactions between the shared library and simulator model, the separation of code between simulator and library, and how the shared library contains support code and many copies of the global data. These ideas are discussed further in section 3.1.

The boxes labelled “global data” on the right of figure 3.1 indicate that there are multiple copies of the global data used by the network stack. This mechanism is used to support independent instances of the TCP implementation running within the same process. The implementation source code is modified programmatically by a program called the *globaliser*. This process is covered in section 3.2

NSC supports TCP implementations extracted from several operating systems (Linux, FreeBSD, and OpenBSD) along with a TCP implementation designed for use on embedded devices (lwIP). The process used to make each new stack available to simulation with NSC is described in section 3.3.

3.1 Simulator integration

The discussion of the architecture at the beginning of this chapter introduces the use of two components: a shared library and a simulator agent. Figure 3.1 shows the basic relationship between the two. This section discusses each component and details their interactions. Section 3.1.1 discusses how and why shared libraries are used in the Network Simulation Cradle. The integration with the network simulator is described in section 3.1.2 while the interactions between the simulator and shared libraries are covered in section 3.1.3. This section discusses in detail an architecture capable of supporting all the features introduced at the beginning of this chapter.

3.1.1 Shared libraries

Communication between the simulator and TCP implementation is required for a variety of interactions such as reading and writing data to sockets, sending and receiving packets, and configuring TCP endpoints. For the NSC to be efficient this communication needs to be efficient, as all interactions between simulator and TCP implementation use this mechanism. Simplified TCP models in simulators such as ns-2 are linked statically into the simulator executable. Functions calls within this executable are used for communication, making this approach very efficient. A real world TCP implementation can also be statically linked into the simulator in this way, examples of this approach can be seen in the addition of FreeBSD to the OMNeT++ simulator [106] and the early development of the NSC [116].

Statically linking TCP implementation code does not scale to multiple TCP implementations. For example, the OpenBSD and FreeBSD TCP implementations cannot be statically linked into one executable as there are many symbols (such as the function `tcp_input`) that clash. This is because all non-static functions and global variables share the same namespace when statically linking C code into a single executable. Individual namespaces are required for each TCP implementation.

The code needs to be separated in some way: either into different shared libraries or different processes. Shared libraries can be used if they are loaded at runtime with the POSIX `dlopen` function. Loading libraries in this way results in symbols that are only available to the executable if explicitly located with the `dlsym` function, meaning there are no symbol clashes between the libraries.

An alternative to using shared libraries would be to use a separate process to contain each network stack. The processes would use a form of inter-process communication (IPC) such as sockets to interact with the main simulator process. The overheads introduced by this approach are greater than if using shared libraries: IPC mechanisms need to perform extra functionality over function calls within a process and context switches are required to change from executing code in the simulator to executing code in the TCP implementations. The NSC uses shared libraries because they offer a scalable and efficient solution to communication between TCP implementations and the network simulator.

Each shared library implements a C++ interface so there is a generic way of handling each TCP implementation. The simulator agent decides which shared libraries to load based upon simulator input. The consistent interface allows the agent to handle each TCP implementation in the same way.

3.1.2 Simulator agent

The simulator agent (shown as in the grey area on the left side of figure 3.1) is responsible for routing messages between the simulator and the shared libraries containing the network stacks. There is one simulator agent per TCP endpoint. The simulator agent is integrated into the network simulator ns-2 [13] and forms a transport model with a user-facing interface compatible with the existing TCP models in ns-2. The agent is a standard ns-2 model; implemented as one source file linked into the ns-2 executable. Other ns-2 models interact with it via the ns-2 agent interface (inheritance and virtual functions are used). Configuration is exposed using standard ns-2 mechanisms to export variables and functions to OTCL simulation scripts.

The level of abstraction in the network simulator will be different to the mechanisms used in the TCP implementation, so the simulator agent must map between the two. An example is converting between addressing formats; the network simulator may abstract away IP addresses, while the real stacks will use version 4 or 6 IP addresses. Many network simulators do not use actual data in the packets, only a length is used instead. This is an example of another difference that the agent must support.

The NSC simulator agent for ns-2 performs these functions as well as managing TCP connections and the interactions between the simulated application and the socket functionality exposed by the network stack. This is required because ns-2 does not use a standard BSD sockets API for communication between an application and TCP model. These topics are covered next.

Stack instances

One independent instance of a TCP implementation is used per TCP endpoint (and therefore per simulator agent) in the NSC. A single network interface is configured and a default gateway is used to route all traffic through this interface. There is no support for using TCP/IP implementations to route packets with the NSC. This is not a limitation of the approach; rather the simulator performs the routing in a router node. The NSC could support multiple interfaces and routing, Bless and Doll [106] show this can be done by implementing this feature in their FreeBSD extensions for the OMNeT++ simulator. This is not a goal of the research presented in this dissertation, as the focus is on simulation of TCP, not IP routing.

An alternative to using a single stack instance per TCP connection endpoint is to share connections within one instance of a TCP implementation. Real-world TCP implementations generally support a large number of TCP connections efficiently. Simulating the connections for different simulation components with a single TCP implementation instance would mean that global data in the TCP implementation is shared. This will potentially affect results; one example of global data is the round-trip-time (RTT) cache utilised by some current operating systems. This cache stores TCP control data and RTTs to IP addresses that have been communicated to in the past and uses this information to initialise TCP variables to increase performance when a new connection is made to the IP address. If this data is shared between simulation nodes, then unrealistic defaults will be used for some TCP connections as they will be initialised from values that were cached from a connection on a different simulated computer.

TCP connection timers

TCP uses a number of timers, see [84] for a definitive list. These are managed in network stacks from a “soft” clock running many times a second, usually ranging from 10 to 1000. This value is often simply called hz. Once every $1/hz$ seconds, a timer is fired in the simulator agent that calls a function in the network stack to notify it of time passing. This allows the stack to manage its TCP timers.

This method has a performance impact for simulation because timers will have to fire every $1/hz$ seconds whether or not there is any activity on the TCP connection, or whether a TCP connection is established or not. For a performance analysis of the NSC, see chapter 6.

Application management

When configured to listen, the simulator agent configures its one managed socket to listen on a port, then attempts to accept a connection on the socket whenever a simulated packet is received. Once this is done, the new connection object is managed as described below. If configured to connect to another node, the simulator agent only has to call the connect function on the socket.

In ns-2, the receiving application is assumed to read data as fast as it can. ns-2 does not implement the receivers advertised window in its TCP models, flow control is not required when the receiving end consumes data as fast as it becomes available.

In the NSC this functionality is simulated by reading from the socket provided whenever a packet is read. This means that the underlying real world stack will implement flow control, but this facet of TCP will not be used due to the application modelling. This is a limit of ns-2's application design, other network simulators could make use of TCP flow control.

Sending data in ns-2 uses a simplified interface unlike the BSD sockets interface used on many real systems. A sending application model in ns-2 has no way of knowing if its request to send data will be enqueued in TCP buffers or not: there is no feedback or error response. The TCP agents in ns-2 have an effectively infinite buffer. When a request to send is made, an integer is increased by the number of packets to send.

The NSC simulator agent has two modes of operation: an infinite buffer mode, which works like the ns-2 TCP models do, or a limited buffer mode, where requests to send are ignored when the TCP buffers in the network stack are full. The latter is designed to mimic the pattern of writing to sockets that some real world applications take. This mode is used when performing validation tests which are covered in chapter 4.

Interchangeable use of TCP models

In ns-2, simulation scenarios are defined by OTcl simulation scripts. The NSC TCP agents support the interface used by the standard ns-2 TCP models so they can be used interchangeably in existing and new simulation scripts. Listing 3.1 shows an example of part of a simulation script that interacts with an ns-2 TCP model. To use the same script with an NSC TCP implementation, the script needs to be changed to listing 3.2: only the lines creating the TCP models need to change. The example shows a FreeBSD TCP implementation, which is one of the default TCP implementations, and an alternate way of loading a TCP implementation where the shared library is specified explicitly.

Listing 3.1: ns-2 simulation script using a TCP model

```
# Set defaults
Agent/TCP set packetSize_ 1500
Agent/TCP set window_ 40
# Create topology
set node1 [$ns node]
set node2 [$ns node]
$ns duplex-link $node1 $node2 10Mb 34ms DropTail
# Create TCP models
```

```
set src [new Agent/TCP/Newreno]
set sink [new Agent/TCPSink/DelAck]
# Attach models to the topology
$ns attach-agent $node1 $src
$ns attach-agent $node2 $sink
```

Listing 3.2: Using NSC TCP models in a ns-2 simulation script

```
# Set defaults
Agent/TCP set packetSize_ 1500
Agent/TCP set window_ 40
# Create topology
set node1 [$ns node]
set node2 [$ns node]
$ns duplex-link $node1 $node2 10Mb 34ms DropTail
# Create TCP models
set src [new Agent/TCP/NSC/FreeBSD5]
set sink [new Agent/TCP/NSC/Linux26]
# Attach models to the topology
$ns attach-agent $node1 $src
$ns attach-agent $node2 $sink
```

Variables such as `packetSize_` and `window_` are used by the NSC agent when initialising the TCP implementation. `packetSize_` is used to set the MTU while `window_` is used when setting the TCP buffer sizes. Initialisation is performed lazily by default. When the first packet arrives at a TCP model, or a TCP connection is created to send packets, the TCP implementation is initialised. An interface is added with an IP address based upon the ns-2 node address, which is automatically assigned by ns-2, and the required TCP socket is created. Further control is possible by explicitly calling initialisation functions.

3.1.3 Interactions

This section describes the interactions between the simulator agent (section 3.1.2) and the shared libraries (section 3.1.1): the arrows in the centre of figure 3.1. A C++ header file is included in the build of each shared library and the simulator. This header describes an interface the shared library must implement. All interactions between the agent and the network stack use this interface. Interactions will either be global, per-stack or per-TCP connection. For brevity, the interfaces shown below do not include debugging and similar functions.

As covered in section 2.3, previous approaches to using real world TCP code in simulation did not provide a way to support multiple TCP implementations transparently. The interface described in this section is generic and allows for this transparency of TCP implementation, making it simple for a user of the NSC to simulate with multiple different TCP implementations, a feature not previously available to network simulation practitioners.

Global interactions

Only one function is exported from the shared library: `nsc_create_stack()`. This function is known as a factory function: it creates a new instance of the stack contained in the shared library and returns a pointer to an object which manages this stack. This function is called with parameters that are opaque objects the shared library can use to call back into the simulator to send packets, ask for the time, or inform the simulator of activity on a socket (for example, there is new data waiting on that socket).

Per-stack interactions

The stack creation function returns an `INetStack` struct which has the following members that can be used to interact with that TCP stack:

- `void init()`
- `void if_receive_packet(int id, void *data, int len)`
- `void if_send_packet(void *data, int len)`
- `void if_attach(char *address, char *mask, int mtu)`
- `void add_default_gateway(char *address)`
- `int get_hz()`
- `void timer_interrupt()`
- `int sysctl(char *name, void *oldval, size_t *oldlen , void *newval, size_t newlen)`
- `bool set_var(char *name, char *value)`
- `struct INetStreamSocket *new_tcp_socket()`

After creating the stack with `nsc_create_stack()`, the NSC simulator agent calls the `init()` function, then uses the `get_hz` method to calculate how many times a second it must call the `timer_interrupt` routine. It initialises the stack with an interface with `if_attach()` and adds a default gateway. A TCP socket is created with `new_tcp_socket()`. The TCP socket creation function returns an interface like `nsc_create_stack()` does. Statistics reporting is supported with the `get_var` function, which parses the ”name” parameter and returns the result as a string. This is used to report global information that is not specific to a TCP connection. System-wide properties are configured by the `sysctl` function which works like the function of the same name on many Unix systems.

Per-TCP interactions

Each TCP socket includes the following interface:

- `void connect(char *addr, int port)`
- `void disconnect()`
- `void listen(int port)`
- `int accept(INetStreamSocket **socket)`
- `int send_data(void *data, int len)`
- `int read_data(void *buf, int *buflen)`
- `int setsockopt(char *name, void *val, size_t len)`
- `bool get_var(char *name, char *result, int len)`
- `bool set_var(char *name, char *val)`

The functions map onto the internal stack functions as the equivalent BSD sockets API functions would, except that the `listen()` function also internally calls `bind()`. All functions are non-blocking. Error returns must be handled specially in the cradle code: each operating system may use different error codes. The simulation cradle code specific to each stack must transform the error codes into the accepted standard for this interface.

TCP variables may be accessed via the `get_var()` function. Results are returned as strings like with the `INetStack` functions covered earlier. Variables that can be queried include but are not limited to the round trip time measurements, congestion window size (`cwnd`), window threshold (`ssthresh`), sequence and acknowledgement numbers and the current retransmission timer interval.

Listing 3.3: Example cradle code to connect a socket (from Linux 2.6 shared library)

```
/* Internal connection function */
void nsc_soconnect(void *so, unsigned int dest,
   unsigned short port)
{
    struct socket *sock = (struct socket *)so;
    struct sockaddr_in addr;
    int addrlen;

    addr.sin_family = AF_INET;
    addr.sin_port = port;
    addr.sin_addr.s_addr = dest;

    addrlen = sizeof(struct sockaddr_in);
    sock->ops->connect(sock, (struct sockaddr *)&addr,
       addrlen, O_NONBLOCK);
}
/* Interface implementation */
void LinuxStack::TCPSocket::connect(char *addr, int
   port)
{
    struct in_addr ip_dest;
    uint16_t ip_port;

    inet_aton(dest, &ip_dest);
    ip_port = htons(dest_port);

    set_stack_id(parent->stack_id);
    nsc_soconnect(so, ip_dest.s_addr, ip_port);
    set_stack_id(-1);
}
```

Each shared library needs code to bridge the general purpose interface functions to the internal TCP stack functions. Some extra management must be done in some cases to convert return values. Most functions have a straight forward mapping between the interface function and the internal stack function, as listing 3.3 shows. In this code, a typical example of mapping an interface function to an internal function is shown. Connecting a socket with Linux 2.6 requires some conversion of types then a call to the socket's connection function pointer. In other cases further management is required; setting the default gateway can be more complex: with FreeBSD, it requires management of a routing socket, with Linux a call to an `ioctl` function on a socket.

Table 3.1: Number of declarations and references of global variables

Network stack	Global variables	Number of references
FreeBSD 5.3	2418	11790
Linux 2.4.28	836	13794
Linux 2.6.14.2	792	10217
OpenBSD 3.5	735	6056

3.2 Global parser

Chapter 1 introduces the need for the network stacks supported to be re-entrant in section 1.5.1. Two important points are made in that section: multiple instances of simulation models are required and real code does not, in general, support multiple instances. The process of making existing code re-entrant is called *virtualisation* in the following sections.

The shared resources which need to be virtualised are global and static local variables (variables which have global linkage but local scope), herein referred to simply as global variables. These are placed in areas of memory which are not part of the call stack or heap; they are shared between different function calls in the source code. Global variables need to be modified such that multiple calls into the code can be made, each referencing a different set of global variables. Each reference to such a variable must be mapped to the real data in an implementation-dependent manner.

Other projects have modified the source by hand to support virtualisation. ENTRAPID [117] and ALPINE [113] are protocol development environments that modify the BSD network stack code by hand to virtualise it. Zec [118] modifies the FreeBSD network stack code by hand so it may be cloned. When integrating the FreeBSD TCP/IP stack into the OMNeT++ simulator, Bless and Doll [106] modify the code by hand to virtualise it. Modifying the source code by hand is not only error prone, but it makes updating the original source code harder.

Many lines of code need to be changed in a large project if all of the global variables and all references to global variables are modified. Table 3.2 shows the number of global variables that need to change in the network stacks that are used in the NSC and the number of times they are referenced. Ensuring that all the necessary changes are made is difficult: global variables need to be identified and all references and declarations changed. If some of the global variables that need to be modified are missed, subtle errors are possible. Any further additions or modifications to the original code (such as new releases or updates) must have the same manual process used to modify their code so it can be incorporated. This is also true of new projects that are to be supported (that is, any new TCP/IP stacks incorporated into the framework need to have this process applied).

This section introduces the global parser, also known as the *globaliser*. The globaliser created for this project programmatically modifies preprocessed C source code, changing global variable definitions and references as needed, making the code re-entrant.

There is a variety of ways the source code can be changed to support virtualisation. Zec [118] modifies each function to take a pointer to a structure which contains the previously global variables when making the FreeBSD network stack able to be cloned. An example of how the source is changed as shown in listing 3.4.

Listing 3.4: Aggregating globals into a structure

```c
int done = 0;

void process() {
    done = 1;
}
```

```c
struct globals {
    int done;
};

void init_globals(struct
   globals *g) {
    g->done = 0;
}

void process(struct
   globals *g) {
    g->done = 1;
}
```

On the left is some example input code, on the right is the sort of output that must be produced. The disadvantages to this approach are that all globals must be aggregated into one central structure, the initialisation for the globals must move into a separate function, and every function that refers to a global variable must be changed to include an extra parameter. It is difficult to aggregate all global variables into a central structure in a large base of code. Doing so means that the declarations of all global variables are moved into a central place, which causes potential variable and type name clashes.

A potentially simpler approach is to modify each global to be an array. An example follows in listing 3.5.

Listing 3.5: Modifying a variable into an array

```
int done = 0;

void process() {
    done = 1;
}
```

```
int nsc_current = 0;

int done[5] = { 0, 0, 0,
   0, 0 };

void process() {
    done[nsc_current] = 1;
}
```

In this case the global variable is changed into an array and an array index when referenced. One extra global variable is created to indicate which set of global variables is currently being accessed. The disadvantage of this approach is that the maximum number of independent instances supported must be specified in the array declaration. This means that to increase the number of instances supported the number must be changed and the code recompiled. The globaliser takes an approach based on this, how it modifies declarations and references to global variables is covered in the following sections: section 3.2.1 and section 3.2.2 respectively. Throughout these sections the examples shown are from the FreeBSD 5.3 TCP/IP source code unless otherwise mentioned and modified to be shorter in some cases for brevity. The examples are shown for an example situation that supports 2 network stack instances.

3.2.1 Modifying C global declarations

Variables in C are declared before they are used. They may be declared as an external symbol, “forward declared”, or declared in full. Once a symbol has been declared it may be used, or referenced in the source. This section describes how the globaliser modifies declarations of global variables. An example of a simple global variable from the FreeBSD source code follows in listing 3.6.

Listing 3.6: globaliser input and output for a single variable

```
static const int
   tcprexmtthresh = 3;
```

```
typedef const int
   _GLOBAL_307_T; static
   _GLOBAL_307_T
   global_tcprexmtthresh[
   NUM_STACKS] = {  3,  3,
    ...
```

This example is slightly more complex than the example shown earlier in section 3.2. The global variable is prefixed with `global_` for debugging reasons, and because it then means any reference to the old, non-modified variable, will produce a compiler error. `NUM_STACKS` is a macro that is defined by the user that specifies how many instances are supported. Arrays are particularly problematic and led to producing code differently when globalising array variables as detailed next. The reason for an additional **`typedef`** of `_GLOBAL_307_T` is described when structures as part of a type name are introduced.

Arrays

Adding an extra array dimension to the declaration of the global variable does not work when the variable is already an array. There are two reasons. The first is array ordering: in C, arrays are stored in row major order, so the compiler needs to know ahead of time the length of the rows. Only the number of rows, which corresponds to the innermost array dimension, may be left unbounded. This is illustrated by attempting to modify an initialised global array variable. Listing 3.7 is code that would be generated by adding an extra array dimension to the end of a global variable.

Listing 3.7: Array initialisation

```
int tcp_backoff[3] = { 1,
   2, 4 };
```

```
int global_tcp_backoff[3][
   NUM_STACKS] = { { 1, 2,
    4 }, { 1, 2, 4 } };
```

Listing 3.8: Corrected array initialisation

```
int global_tcp_backoff[NUM_STACKS][3] = { { 1, 2, 4 },
   { 1, 2, 4 } };
```

The code on the right of listing 3.7 is incorrect. This is because of the ordering of the array dimensions: the variable `global_tcp_backoff` is declared as an array with three rows of two elements, but is being assigned to an array of two rows of three elements. If the ordering of the array is reversed in the declaration the code is correct, as shown in listing 3.8.

Unbounded arrays can not be modified in this way correctly. The above method fails when an array bound is not specified. Listing 3.9 shows example output which will not compile once the code is changed due to the modification to the declaration of `tcp_backoff`. The innermost array bound can be left unspecified but all other bounds must be specified, only the number of rows will be deduced by the compiler.

Listing 3.9: Unbounded array initialisation

```
int tcp_backoff[] = { 1,
   2, 4 };
```

```
int global_tcp_backoff[
   NUM_STACKS][] = { { 1,
   2, 4 }, { 1, 2, 4 } };
```

The two problems illustrated above show that adding an array dimension, either before or after the original array definition, will not work in all cases. Adding an extra array bound in the correct way is mutually exclusive with supporting unbounded arrays. One solution involves an extra level of indirection.

The method used by the globaliser is shown in the example globaliser output in listing 3.10. The globaliser keeps the original declaration of the array variable intact and clones it `NUM_STACKS` times. Each time it creates a new unique variable name. A static array is created that contains pointers to each of the array variables that were cloned.

The array is static so the symbol is not exported outside the current C file, this makes sure the new variable does not clash with array variables created by the globaliser in other files. The symbol must not be global outside of the compilation unit because the same global variable may be defined in another C file. In C, variables may be defined multiple times but only initialised once. A new initialised array variable is being introduced at every definition of a global variable and therefore the array must be static so the same variable is not initialised in multiple source files.

The name of the array is created such that it is unique within the current file by appending a number which increments by one every time a new array is created by the globaliser. The gcc extension __`typeof`__ is used for convenience to declare the new array, if required the parser could be modified to find out this type itself. The reason for **`typedef`** is explained in the following section.

Listing 3.10: Array initialisation solution

```
typedef int _GLOBAL_0_T;
_GLOBAL_0_T _GLOBAL_0_tcp_backoff_I[] = { 1, 2, 4 };
_GLOBAL_0_T _GLOBAL_1_tcp_backoff_I[] = { 1, 2, 4 };
```

```
static __typeof__(_GLOBAL_0_tcp_backoff_I) *
   _GLOBAL_array_tcp_backoff_12_A[NUM_STACKS] = {
    &_GLOBAL_0_tcp_backoff_I, &_GLOBAL_1_tcp_backoff_I
};
```

Listing 3.11: Indexing a modified array

```
(*_GLOBAL_array_tcp_backoff_12_A[stack_index])[0] = 2;
```

References to the variable must also change into a more complex form as illustrated in listing 3.11.

Structures and types

The method described to modify arrays by cloning them several times produces erroneous code when a structure definition is involved in the variable declaration. If the type of the array variable being declared includes an entire structure definition, the structure will be defined many times, creating a namespace collision. The code in listing 3.12 shows example output in such a case if the global variable is cloned.

Listing 3.12: Structure redefinition

```
static struct ipqhead {
    struct ipq * tqh_first
        ;
    struct ipq ** tqh_last
        ;
} ipq[ 1 << 6 ];
```

```
static struct ipqhead {
    struct ipq * tqh_first
        ;
    struct ipq ** tqh_last
        ;
} _GLOBAL_0_ipq[ 1 << 6 ];
static struct ipqhead {
    struct ipq * tqh_first
        ;
    struct ipq ** tqh_last
        ;
} _GLOBAL_1_ipq[ 1 << 6 ];
```

The structure `ipqhead` is defined more than once in listing 3.12 which produces a compiler error. The original type of the variable is correctly cloned but the resulting code is erroneous. A solution to this is to **`typedef`** the type of the variable that is being modified and re-use the **`typedef`** in each cloned variable instance. This method is used by the globaliser and is illustrated in listing 3.13.

Listing 3.13: typedef of array element type

```
typedef struct ipqhead {
    struct ipq * tqh_first;
    struct ipq ** tqh_last;
} _GLOBAL_0_T;
static _GLOBAL_0_T _GLOBAL_0_ipq[ 1 << 6 ];
static _GLOBAL_0_T _GLOBAL_1_ipq[ 1 << 6 ];
```

The globaliser must create a unique type name for each `typedef` in a source file. The method shown above is to have a counter which increments with each `typedef`. `_GLOBAL_` is prefixed to this number and `_T` added appended. This does not guarantee uniqueness, as the original code could use such names in typedefs already, but the method has sufficed for hundreds of thousands of lines of code tested. While this `typedef` is not needed with a simple type like shown in earlier examples, the globaliser creates a `typedef` for all global variable declarations.

Potential limitations

The methods introduced in the previous section and used by the globaliser modify preprocessed C source code, adding extra code to support multiple copies of global variables. This increases the size of the code which results in slower compilation time. The extra symbols added by the process of modifying global array variables affects the performance of linking the compiled object files. Cloning the global variables also means that the object files and resulting binary are larger in size. There is a cost at runtime, as each access of a global variable is mapped through an indirection table. These potential limitations of the globaliser are covered in chapter 6 in detail.

3.2.2 Modifying C global references

Each reference to a modified global variable must be changed. Section 3.2.1 showed how an instance of a global variable is modified. While this is a simpler process than modifying the declarations of global variables, there are several cases that must be handled which are not immediately obvious: these include scoping and initialisation. Before these are discussed, the way a global variable reference is indexed is covered.

Indexing

Examples earlier in section 3.2 used a variable to index into an array of globals. This can potentially be a function call as well. In the Network Simulation Cradle

Report [116] we describe an approach that uses a thread for each stack. To retrieve the index of the stack for the running thread, a POSIX threads function (see e.g. [119] for an introduction to pthreads) is called that reads from thread specific storage for the current thread, allowing multiple threads of execution to be independently running the same code.

The globaliser outputs code to call the function `get_stack_id()` when indexing into a global variable array as illustrated in the code below. A function call provides maximum flexibility, as a potential user could perform actions such as calling pthreads functions or simply returning the value of an index variable. In the NSC TCP implementations, the approach of returning an index variable is used. The code produced by the globaliser to index variables is illustrated in listing 3.14.

Listing 3.14: Indexing a variable reference with a function

```
mtu = 1500;
tcp_backoff[0] = 2;
```

```
global_mtu[get_stack_id()]
    = 1500;
(*
   _GLOBAL_array_tcp_backoff_12_A
   [get_stack_id()])[0] =
   2;
```

Scope

The C language allows local variables to “shadow” global variables. This occurs when a local variable is declared with the same name as a global variable. Code within scope of the local variable will use the local variable not the global one. The globaliser needs to understand scoping so it correctly modified shadowed global variables and static local variables.

Self-referential initialisation

It is possible in the C language for a variable to reference itself in its own initialisation. The following source shows an example:

Listing 3.15: Self-referential initialisation

```
int header_len = sizeof(header_len);
```

The rules described so far would transform this to:

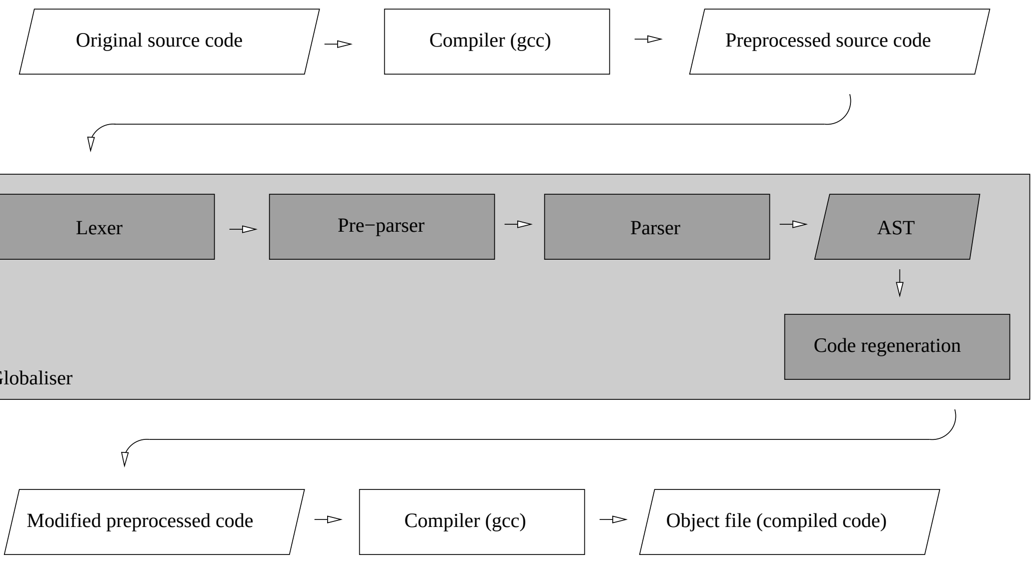


Figure 3.2: Globaliser's parser flow

Listing 3.16: Self-referential initialisation error

```
int global_header_len[NUM_STACKS] = { sizeof(
   global_header_len[get_stack_id()], sizeof(
   global_header_len[get_stack_id()] };
```

This code will not compile: a function cannot be called when global variables are initialised; only constant expressions are accepted because the value is calculated at compile time and put into the data section of the compiled object file. The globaliser outputs the `get_stack_id()` function call and therefore must solve this problem.

When parsing the expression on the right of a global variable assignment, the value `0` is used instead of `get_stack_id()`. All elements have the same size and value when the statement is computed, as the code for the elements is created by the globaliser.

3.2.3 Implementation of the globaliser

The globaliser's parser is implemented in C++ and uses the compiler-compiler tools Flex [120] and Bison [121]. The parsing is separated into two modules: a simple pre-parser and the Bison-generated parser. The flow of data through the globaliser is shown in figure 3.2.

Flex is a lexical analyser generator. It generates a lexical analyser, otherwise known as a lexer. This lexer is responsible for breaking the stream of characters read into tokens. For example, it will use regular expressions to recognise an expression like **`int`**, and then return an `INTEGER` token to the parser. A Flex input file is a list of regular expressions and the tokens they match.

The lexer returns information about whitespace and comments along with the C keywords, identifiers and symbols so the globaliser can regenerate the original code exactly. This allows easy verification with `diff` tools that the globaliser only modifies relevant sections of code.

The pre-parser stores whitespace and comment information in a buffer and passes other tokens to the Bison-generated parser. The Bison-generated parser reads the global buffer and copies it into the abstract syntax tree representation of the source.

Bison is a parser generator. It takes a context-free LALR grammar and generates a C program to parse that grammar. Bison produces quick parsers with one token of lookahead. The input format of Bison is similar to BNF. The ISO C standard [122] includes a BNF grammar for C, though none for Bison specifically.

A grammar compatible with Bison and Flex compatible lexer is freely available for the 1985 ANSI C standard (see [123]). The grammar used by the globaliser is based on this and has been updated to handle the features of the C89 and C99 [122] standards and gcc [124] extensions used by the operating system network stacks used in the NSC.

The globaliser builds an abstract syntax tree (AST) of the input source. When a declaration is found, it is processed to check whether it is a global. If it is, it is checked against the table of global variables to be modified. The node of the declaration in the AST will then be modified if the variable is to be changed. The node is changed based on the rules introduced earlier. The full AST representation of the source allows the name, type, and initialisation parts of the declaration to be extracted and modified with the rules introduced earlier. A similar process takes place whenever a variable reference is encountered. Once the input file is finished being read and the AST is fully built, the AST is printed out. This process reconstructs the source. Any nodes that were modified earlier to change global variable declarations or references are printed in their new, modified form. Other nodes of the AST that were not modified in the previous process are reconstructed so they produce the same output as input.

To handle scoping of variables, the globaliser maintains a stack of local variables. The contents of the stack are updated based upon the local variables encountered in each block of code found in the input. Whenever a variable is referenced, the stack is first scanned to see if the variable is a local variable. If not, it is then processed as a reference to a global variable.

gcc extensions

While the globaliser understands ANSI C, it is often run on source code designed to be compiled with gcc. In some cases this code uses gcc specific extensions to the C language which must be parsed correctly. Some examples from the Linux and FreeBSD kernels that make use of these extensions follow. A full list of gcc extensions to the C language can be found at [125].

The following example shows an inline code block which allows normal statements such as declarations.

Listing 3.17: Inline code block from Linux

```
return ({ int __x = (nbits); int __y = (find_first_bit(
   srcp->bits, nbits)); __x < __y ? __x: __y; });
```

gcc has many attributes which can apply to functions or variables. The example below shows the attributes associated with the `panic()` function in the FreeBSD 5.3 source code. The attributes tell the compiler that the function will not return and that it takes arguments like `printf`. This allows the compiler to perform extra analysis in blocks of code that use the `panic` function, such as producing warnings if the arguments passed in the variable argument list do not match up with the `printf` format string.

Listing 3.18: gcc attribute use from FreeBSD

```
void panic(const char *, ...) __attribute__((
   __noreturn__)) __attribute__((__format__ (__printf__
   , 1, 2)));
```

gcc allows the additional keywords `typeof` and `offsetof`. It also allows using alternate keywords by adding __ to the beginning and ending of a keyword, for example __asm__ instead of asm. The effect of this is that the lexer and parser must be aware of these new keywords.

Listing 3.19: Additional gcc keywords

```
typeof(int *);
offsetof(struct intf_t, iface);
```

Inline assembly is possible in gcc and needs to be parsed correctly because it can contain references to variables. The example in listing 3.20 shows the variables ptr and do_softirq within a gcc inline assembly statement. These are possible global variable references and must be understood correctly by the globaliser. The rules introduced for handling variable references can be used on the variables referenced in the asm listing.

Listing 3.20: gcc inline assembly

```
__asm__ __volatile__ ( "cmpl␣$0,␣-8(%0);"
"2:␣pushl␣%eax;␣pushl␣%ecx;␣pushl␣%edx;"
"call␣%c1;"
"popl␣%%edx;␣popl␣%%ecx;␣popl␣%%eax;"
: : "r" (ptr), "i" (do_softirq));
```

The globaliser parses all of the above gcc extensions and modifies references to variables found in them correctly.

Semantic support

The function of the globaliser is to virtualise C code, it is not required to check for well formed C code. The globaliser therefore does not need to perform semantic analysis of its input. It does, however, need some knowledge of the semantics of C to parse it correctly: it needs to understand the **typedef** keyword which defines a new type keyword. A set of all identifiers **typedef**-ed is kept, and whenever an identifier is found in the source, it is checked against the set of type names to see whether it is a type keyword or an identifier. This works for most C code, but there are valid C constructs which break this method.

Listing 3.21: Type name parsing problem

```
typedef int proc_handler (ctl_table *ctl, int write,
   struct file * filp,
     void *buffer, size_t *lenp, loff_t *ppos);
proc_handler *proc_handler;
```

The source in listing 3.21 shows where this method fails. It is legal in the C language to use an identifier which has been previously been **`typedef`**-ed as a normal identifier in some situations where it is not ambiguous. The globaliser uses an updated grammar which supports this feature by by handling **`typedef`** keywords differently to other type keywords such as **`int`**.

Section handling

One of the gcc attributes which can be set on a global variable is the "section" attribute. This allows the programmer to instruct the linker to place the variable in a particular section in the object file. The linker will then make two variables which point to the beginning and end of the section. This allows a programmer to place a set of variables in their own section in the object file, then iterate over them using the start and end pointers provided. The FreeBSD kernel uses this method for initialisation.

Listing 3.22: Section attribute object file placement

```
int var1 __attribute__ ((
   __section__ ("sysinit")
    ));
int var2 __attribute__ ((
   __section__ ("sysinit")
    ));
```

```
0x0014  __start_sysinit
0x0014  var1
0x0018  var2
0x001c  __stop_sysinit
```

Listing 3.22 shows a basic example of how sections work. On the left two variables are declared in a section called "sysinit". On the right example addresses of the variables in the object file are shown. Two extra variables are created which have memory addresses that bound the variables in the section.

The rules introduced so far would produce code that would compile but would not have the same functionality. Given a variable with a section attribute, the attribute would be retained but the variable definition modified, making many instances of the variable appear in the section. Any code which iterates over the section would then not work as expected.

To solve this problem the globaliser has support for modifying the section attribute. The output of the globaliser when using section support on the code in listing 3.22 is shown in listing 3.23. If configured to, the globaliser will see that var1 is in the section “sysinit” and instead of creating an array within the same section, will instead declare many instances of var1 in different sections. The __stop_sysinit declaration is not shown for brevity, it is equivalent to the method used for the __start_sysinit symbol.

Listing 3.23: globaliser section support

```
int _GLOBAL_0_var1 __attribute__ (( __section__ "
   global_section_0_" "sysinit" ));
int _GLOBAL_1_var1 __attribute__ (( __section__ "
   global_section_1_" "sysinit" ));
extern void * __start_global_section_0_sysinit, *
   __start_global_section_1_sysinit;
static void * * __start_sysinit[NUM_STACKS] = { &
   __start_global_section_0_sysinit, &
   __start_global_section_1_sysinit, };
```

3.3 Adding a new stack

TCP implementations of interest to researchers come and go. One of the goals of the NSC is to aid addition of future TCP implementations. The steps involved in adding support for a new TCP implementation are extracting the TCP code from its normal environment (usually an operating system kernel), compiling and linking the code into an executable, solving any undefined references, incorporating with the NSC and the globaliser and testing the new TCP code with ns-2. A guide through this process for someone wishing to support a new network stack in the NSC is found in appendix B. This section provides a discussion of the feasibility of adding new TCP implementations to the NSC.

3.3.1 Extracting the TCP code

The TCP code may need to be extracted from a larger base of code; this is true in the case of network stacks inside operating systems. Much operating system code does not make sense to run in user space. For example, the operating system manages access to the hardware, such functionality is not desired in a simulation model and the functionality would not work in user space.

The network code needs to be identified within a potentially very large project. In the operating systems studied this is simple due to a logical layout of files. In FreeBSD and OpenBSD, all code specific to Internet protocol implementations is found in the directory `/usr/src/sys/netinet`, where `/usr/src/sys` is the base directory for the kernel source code. The TCP implementation is contained in the files with the prefix `tcp_`. Information on where the TCP implementation is contained in a BSD derived kernel is also discussed by Wright and Stevens [84] and McKusick *et. al* [126]. Without books or obvious documentation this was still evident in Linux: TCP code is in files under the `net/ipv4/` directory inside the kernel sources that have names beginning with `tcp_`. While not part of the NSC, OpenSolaris has been studied as a potential addition to the NSC. The OpenSolaris TCP code is located in `uts/common/inet/tcp`. In all cases a search for files containing `tcp` in their name locates the TCP implementation source files.

3.3.2 Building a standalone TCP implementation

The TCP implementation extracted from an operating system needs enough support functionality to run independent of the original system it was part of. Compiling and linking the TCP implementation alone will show all undefined references to the host operating system. The type of support functions encountered include, but are not limited to, threading/locking primitives (mutexes, condition variables), time functions (time counters, management of timer callback functions), memory management, logging, error handling, cryptographic functions and IP networking. Some of these functions make sense to include in the library that will be loaded by the NSC while others will not work in user space.

Finding a suitable division

A suitable division is required such that the TCP implementation will operate like it did in its original environment while running in a simulated environment. Some of the support functions can be included to solve undefined references and will not have any further requirements on support from the NSC, cryptographic functions are often an example of this. No code other than the core TCP implementation needs to be included but including more code means that the model created is closer to the original system. Being closer to the original system makes it potentially easier to reproduce the behaviour of the original system which is the

Table 3.2: Number of support functions in the NSC shared libraries

Network stack	Stub functions	Implemented functions
FreeBSD 5.3	39	93
Linux 2.4.28	54	55
Linux 2.6.14.2	150	60
OpenBSD 3.5	43	24

goal of this process.

The division used in each of the TCP implementations extracted from operating systems in the NSC is similar. All contain at least TCP, IPv4, IPv6, ICMP, sockets, cryptographic functions, UDP, routing, packet buffer functions, some timer support, and global configuration support (i.e., `sysctl`). This division reduces undefined references and provides necessary support for the TCP code.

Building stub functions

Any undefined references that are not solved by introducing additional code into the build can be solved with stub functions. The term stub function is used here to mean a function that is created that performs no action other than to signal that it is not implemented. In the NSC these functions are implemented as assertion failures. If a stub function is called at runtime, the program is aborted with an error message indicating the function that is not implemented. This means the stub functions which are used are discovered during testing and must be implemented.

The number of stub functions and implemented functions in the NSC shared libraries are summarised in table 3.2. The implemented functions refer to the stub functions that are required to be implemented to allow the TCP code to run (the numbers reported in the table are independent, the number of implemented functions is not included in the number of stub functions). Each of these functions needs to be studied in detail to ensure it works in a way consistent with the original system. Some functions map to simple C library calls (e.g., memory allocation can use `malloc`) while others are not required to perform any action when being run in a simulated environment (e.g., mutexes, checking for user permissions).

Table 3.3 shows counts of the lines of code included in the NSC support code for each stack. The number of lines attributed to stub functions is made up mostly of a

Table 3.3: Number of lines of code used in the NSC stack support code

Network stack	Support lines of code	Stub lines of code
FreeBSD 5.3	3550	39
Linux 2.4.28	1603	571
Linux 2.6.14.2	1871	1205
OpenBSD 3.5	1540	311

lot of boiler plate code and could be reduced to only one line for each stub function. The NSC FreeBSD 5.3 stack has stub functions implemented in this way.

3.3.3 Incorporating with the Network Simulation Cradle

Once a TCP implementation has been built into a shared library it can be incorporated with the NSC. Doing so requires implementing the interface introduced in section 3.1.3. New code needs to be written that calls functions in the TCP implementation to perform actions such as connecting, reading and writing. Finding the correct functions to call can be achieved by tracing the code path between a user space application that uses the BSD Sockets API and the functions called inside the kernel.

The functions called in the TCP implementation need to return control to the simulator: they cannot block waiting for a resource (such as a packet arrival) because the simulator has a single thread of execution. Non-blocking versions of socket operations are used with the NSC implementations. If a TCP implementation did not support non-blocking operations then such functionality would need to be built. We have shown this to be feasible using threads to store different function call contexts in earlier work [116].

The globaliser needs to be incorporated into the build of the TCP implementation to support multiple independent TCP instances. It is used as part of the build tool chain after source code preprocessing but before compilation. An example of a build rule using the globaliser during compilation of a source file is shown in listing 3.24.

Listing 3.24: Compiling a C file with gcc and the globaliser

```
gcc ${CFLAGS} sample.c -E - |
    ./globaliser -vv ./globals.txt |
    gcc -xc ${CFLAGS} -c - -o sample.o
```

The shared library can be used with ns-2 once the interface is implemented. The ns-2 simulator agent for the NSC supports tracing packets in PCAP format so direct validation against packet traces from a equivalent network implemented with physical devices is possible once the interface is implemented. Validation of NSC TCP implementations is covered in detail in chapter 4. Because the NSC interface for all shared libraries is the same, existing simulation scripts can be used with the new TCP implementation by only changing the name of the shared library loaded.

3.3.4 Configuration issues

There are many configuration options in real world network stacks and these should be exposed to the simulation user. The NSC interface supports several types of configuration: sysctls, socket options, and general string-based commands. The ns-2 simulator agent integrates this configuration into the OTCL scripting language so a user may specify sysctls and other configuration in a natural format. Each stack then implements the interface to set such configuration.

The implementation of the configuration options is often simple: the input data from the simulation user is transformed into the format used by the stack, then kernel configuration functions can be called. This is true for example in the FreeBSD sysctl configuration, which is implemented in three lines of code (calling the FreeBSD function `kernel_sysctlbyname`). In other cases more support code needs to be written to support such configuration; in the Linux stack support code there is code to manually parse the sysctl name passed in.

3.3.5 Updating an NSC TCP implementation

One of the goals stated at the start of this chapter was for real world TCP models to be easy to update. Updating stacks to new versions should take a minimal amount of time, as new versions are often released regularly. The NSC makes this possible in most cases because the stack's source code used is not modified by hand. The code for the new version of the TCP implementation needs to be used in the built system in place of the previous code. The process of building and testing for undefined variables should be followed, like the initial integration of a network stack discussed in section 3.3.2.

A new version of a TCP implementation might add new files and require different compilation flags. The differences between the two versions should be inspected for additional files so they can be added into the build process. The build system should be verified against building the new TCP implementation code in its original environment to ensure the code is still compiled in the same way.

Once these steps are followed the new code can be tested. The process of updating is much simpler due to the work done to integrate the earlier version of the stack. The amount of work required to update to a new version is proportional to the size of the change in the TCP implementations. When upgrading from Linux 2.4.27 to Linux 2.4.28, no changes were required in the support code. The code was patched with the new version of Linux and tested to ensure behaviour consistent with a computer running Linux 2.4.28. Updating to support Linux 2.6.10 was more involved, as the 2.6 series kernel is a major update of the Linux kernel over the previous 2.4 kernels. Around 200 lines of support code changed during testing and many TCP implementation files were added and removed. The stub functions needed to be recreated as many internal kernel functions had changed or been added.

3.3.6 Requirements of the NSC approach

The Network Simulation Cradle requires the source code of the network stack to be simulated. This is available for open source TCP implementations such as those found in the operating systems of Linux, FreeBSD, OpenBSD, and OpenSolaris. However, the source code is not generally available for Microsoft Windows, and is therefore not available in the NSC. The approach used by the NSC applies to any TCP implementation with source available, so support for Microsoft Windows is conceptually possible if the source code were available.

The NSC is designed for incorporating code written in the C language (due to all TCP implementations studied being written in C), but other languages could be supported using processes similar to those discussed for the C language. A bridge between the two languages would be required in the support code in the shared library.

3.4 Summary

This chapter shows that it is feasible to use real world network stacks as a TCP model in a network simulator. That an implementation was created within the bounds of this project shows that simulating multiple real world TCP implementations can be achieved with reasonable cost: one researcher over the course of two years.

At the start of this chapter (page 41), a list of goals was set out for the design of NSC: it must simulate real world code, utilise existing network simulators, perform and scale well, produce accurate results, be easy to update, support different methods of statistics gathering, allow a full range of TCP simulation scenarios, and compliment existing simplified TCP simulation models. This list of goals is achieved with the implementation of the Network Simulation Cradle as the follows.

NSC supports simulation of real world code by providing a framework with which a real TCP implementation is connected to an existing network simulator. A new agent is built in the network simulator which loads shared libraries that contains the TCP implementations.

NSC is designed to perform well because of its use of shared libraries and support code. This allows a minimal overhead when an interaction between the simulator and network stack occurs. NSC is designed to scale well due to the globaliser statically altering code during the build process, allowing many instances to be created quickly during runtime. Both performance and scalability are analysed in depth in chapter 6.

Using real world stacks and not modifying code of the TCP implementation means that NSC can produce accurate results. Validation is supported by being able to produce packet traces that can be directly compared to real machines. The accuracy of results produced by NSC is covered in chapter 4.

Stack code within NSC can be updated easily because it does not need to be hand modified every time (like in other projects discussed in section 2.3). To update an existing stack in NSC, a patch should be created between the old and new versions of the stack to be updated, then the patch should be applied to the source code within NSC. The new version can then be tested and validated.

Chapter 4

Accuracy of TCP simulation with real code

For simulation results to be credible the simulation models in use must undergo *verification and validation*. Balci [127] 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 [128] and Sargent [129] define the two terms to be similar and both note that sufficient accuracy is achieved 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 to demonstrate that they conform to specification (verification of the model) and that the model implementation produces results consistent with a real system (validation of the model).

The ns-2 simulator has a test suite that tests many facets of the simulator including the one-way TCP agents [5]. 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 [81]. This type of testing is a verification that the models produce results consistent with specifications.

Floyd [5] points out that the TCP models in ns are not designed to model one specific real world TCP implementation but be a general model for experimenting with the underlying congestion control algorithms. When using real implementation code in a TCP model in simulation, a different sort of validation can be used. The simulation 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. This method of validation is used in this chapter to show the degree of accuracy attained using the real world TCP implementations in the Network Simulation Cradle for simulation of TCP.

The method of direct comparison is used by Bagrodia and Takai [93] where they 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 [86] simulator:

Direct incorporation of the implemented protocol into the model: this allows the protocol model to be validated against an operational prototype.

Comparison of independently developed models for a given protocol: compare with models from another simulator or models of the same protocol built by others.

Both methods are used in this chapter to show the validity and accuracy of the Network Simulation Cradle TCP implementations. The method of direct comparison is introduced first in section 4.1 and the results of these comparisons are described in section 4.2. Section 4.3 expands on these comparisons to show how simulating with real world TCP implementations and the abstracted models present in ns-2 differ.

4.1 Introduction to simulation and test bed comparisons

The Network Simulation Cradle can produce packet trace files in the format used by tcpdump [130]. Tcpdump captures packets from a network interface and can save them to file. A simulation can be modelled after a test network setup and 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 [131]. This method of comparison is used in section 4.2. The measurement and test bed setup is covered below.

4.1.1 Emulating with a test bed network

Building computer networks of varying topologies, varying link bandwidths and delays, possible packet loss, controlled router buffer sizes and differing TCP implementations is expensive and time consuming. This is one of the reasons simulation is performed; often it is impractical (or even impossible) to build networks to test a protocol or idea. Simulation of an entire network has many abstractions and needs to be validated against real systems, so a compromise often referred to as *emulation* is used. Network emulation is used here to mean a physical network which includes a device or set of devices that simulate part of the network. An example of this is a machine set to route packets between its network interfaces, delaying packets by 20ms. This machine would be simulating a long link in the network topology by adding the artificial delay.

4.1.2 WAND Emulation Network

The WAND Network Research Group [132] has a network of 24 machines available for testing. This network is called the WAND Emulation Network [133, 134]. The machines in the WAND Emulation Network have multiple network interfaces. One network interface card is connected to a central server to form a control network. The other network interface card is connected to a patch panel which in turn is connected to a switch. Some of the machines have four Ethernet ports on their second card, allowing them to be used as routers. The machines are configured with a topology by changing connections on the patch panel. All machines are also connected to a terminal server to allow administration without relying on networking.

Facilities for imaging machines with a new operating system are available, so changing operating system between tests can be automated. The operating system images are configured so a simple daemon program listens for connections on the control network once the system has started. This program accepts a string of text for a command to run and redirects the output of the command to the connection in a similar fashion to ssh or rsh. This allows the machines to be controlled easily and

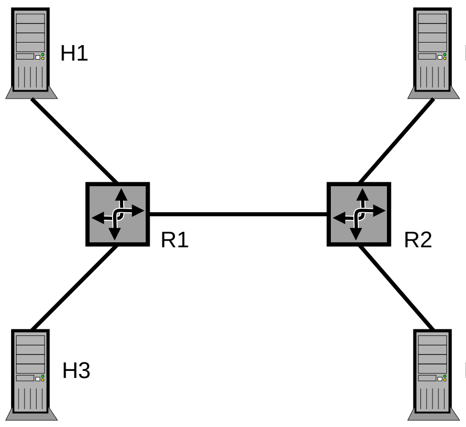


Figure 4.1: Topology used in the WAND Emulation Network

the output of commands to be viewed quickly.

Figure 4.1 shows the topology used for emulation network tests in this chapter. Routers R1 and R2 have four port network interface cards and run FreeBSD 5.3. These routers use Dummynet [135] to shape the traffic going through them as is discussed below. The hosts H1-H4 are imaged with different operating systems and tests are performed between (H1 and H2) and (H3 and H4).

4.1.3 Traffic shaping

Dummynet [135] is commonly used software for network emulation (e.g., [56, 136–139]). It is distributed with FreeBSD 3.4 and later and integrates with FreeBSD's IPFW firewall.

Packets are matched using FreeBSD's IPFW firewalling rules and sent to a Dummynet pipe. A pipe is configured with a bandwidth, delay and packet loss rate. On a tick of the software interrupt clock, Dummynet will check to see if there are any packets pending to be sent out at the current time and queue them for sending if so. FreeBSD 5.3 defaults to this clock ticking at $100hz$, meaning there is up to 10ms jitter for packet delay (there is potential for a larger jitter as the software interrupt will not be run in some situations when the machine is heavily loaded). This rate is determined by an option called `HZ` and can be changed by rebuilding the kernel. Higher values of `HZ` can result in instability and inaccuracy, while lower values result in greater jitter. `HZ` is set to 1000 in the tests in this

Table 4.1: 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

chapter as recommended in the documentation [140].

Experimentation has shown the RTT measured on the emulation network is similar to an equivalent scenario being simulated. Table 4.1 presents the results of running ping with two different packet sizes over the topology shown in figure 4.1 on the emulation network. Router R1 is configured to delay packets by 21ms in both directions and limit bandwidth to 2Mb/s. The ping is between hosts H1 and H2. RTT samples were taken with both packet sizes and with no other traffic running on the network. 1000 samples were used, enough to produce 95% confidence intervals with half lengths around 40μs. No ARP look-ups were performed in both tests as the IP addresses required were already in the ARP cache. Also shown is the RTT measured when simulating an equivalent network in ns-2. There is no jitter in simulation because packets are delayed by precise amounts; simulating an unloaded network will result in the delay being the same for every ping. The jitter shown is explained by the timer granularity of Dummynet and the standard deviation of the jitter and is within the expected range of approximately 1ms. See Vanhonacker [141] for further performance evaluation of Dummynet including delay jitter measurements.

Other emulation software is available. NIST net [142] is a Linux-based network emulation tool. Linux 2.6 contains NetEm [143], another network emulation tool. The ns simulator also has emulation capabilities [144]. Each has a similar featureset to Dummynet.

More precise traffic shaping could be provided by a hardware device. Research is ongoing in the WAND Network Group to produce a switch which can delay packets, limit bandwidth, introduce loss and organise topologies. At the time the research was carried out, no hardware was available for more accurate network emulation.

4.1.4 Traffic generation and measurement

Traffic is measured with tcpdump [130] on emulation network machines. The jitter introduced by the emulation network means that measurement devices with higher precision timing (such as the Dag [145] card) are not required to attain the accuracy needed to compare the emulation network and simulation results.

Traffic needs to be generated on the emulation network and in simulation in the same way. Different strategies used to write data to a TCP socket will result in slightly different TCP behaviours. For example, the size of the first write to a TCP socket will often determine the size of the first TCP packet carrying data. Another example is that the design of the application to use blocking or non-blocking socket IO will affect the resulting TCP stream. An application called Tcpperf [146] is used for fine grained control over the application behaviour to produce interactions with a TCP socket in a way consistent to how ns-2 application models work. Tcpperf allows specifying the size of each write to the TCP socket and which of two schemes to employ to write the data. The first scheme uses the `select` function call to wait until it is possible to write more data to the socket, then calls `send`. The second sets the socket to be non-blocking and calls `send` periodically. This method is similar to the way a constant bit rate traffic generator works in ns-2. Iperf [147–150] is used to generate traffic to be compared to tcpperf for validation purposes.

4.2 Packet trace comparisons

Packet traces produced in simulation and on the emulation testbed network are compared directly in this section. A three step process is used to analyse the traces for equivalence: traces are normalised, visualised and analysed by hand.

The variation in timing on the testbed network shown in table 4.1 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. Instead the traces are visualised with the tcptrace [131] utility and compared by analysing the textual output of tcpdump.

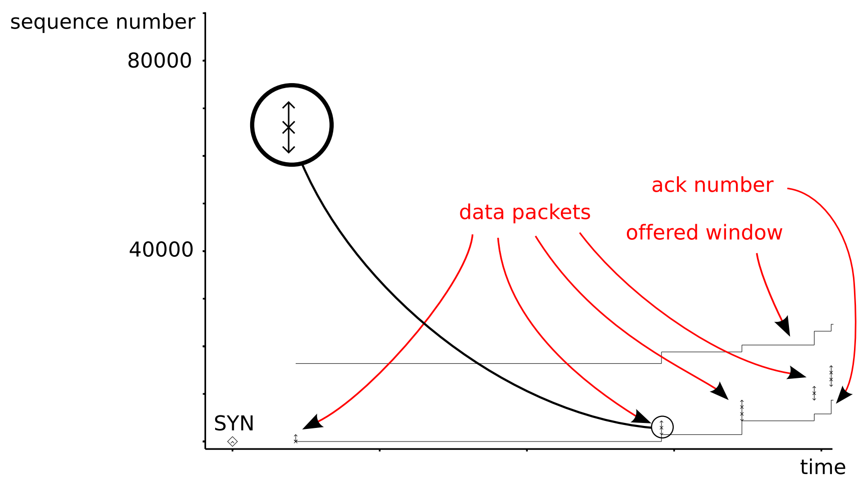


Figure 4.2: Example tcptrace time sequence graph

Tcptrace produces graphs of TCP connections from packet traces. The most useful graph produced by tcptrace for visualising a TCP connection is the time sequence graph. An example annotated time sequence graph is presented in figure 4.2.

The x-axis shows time and the y-axis shows the TCP sequence number. The bottom line on a tcptrace time sequence graph is the sequence number which has been acknowledged to. The top line is the acknowledgement number plus the receiver's advertised window. This shows the window in which the data packets should be sent. Data packets are indicated by small black double-ended arrows (also shown enlarged in a circle on the diagram). 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.

To compare two of these graphs it helps to normalise the time and sequence numbers of each packet in the trace. A utility called tcpnorm [151] was created for this purpose: it normalises a PCAP packet trace by making the PCAP time stamps and TCP sequence numbers start from 0. It handles the timestamp and selective acknowledgement TCP options. Tcptrace has an option to normalise when producing graphs, but this was found to be buggy and to produce inconsistent graphs.

4.2.1 Connection establishment

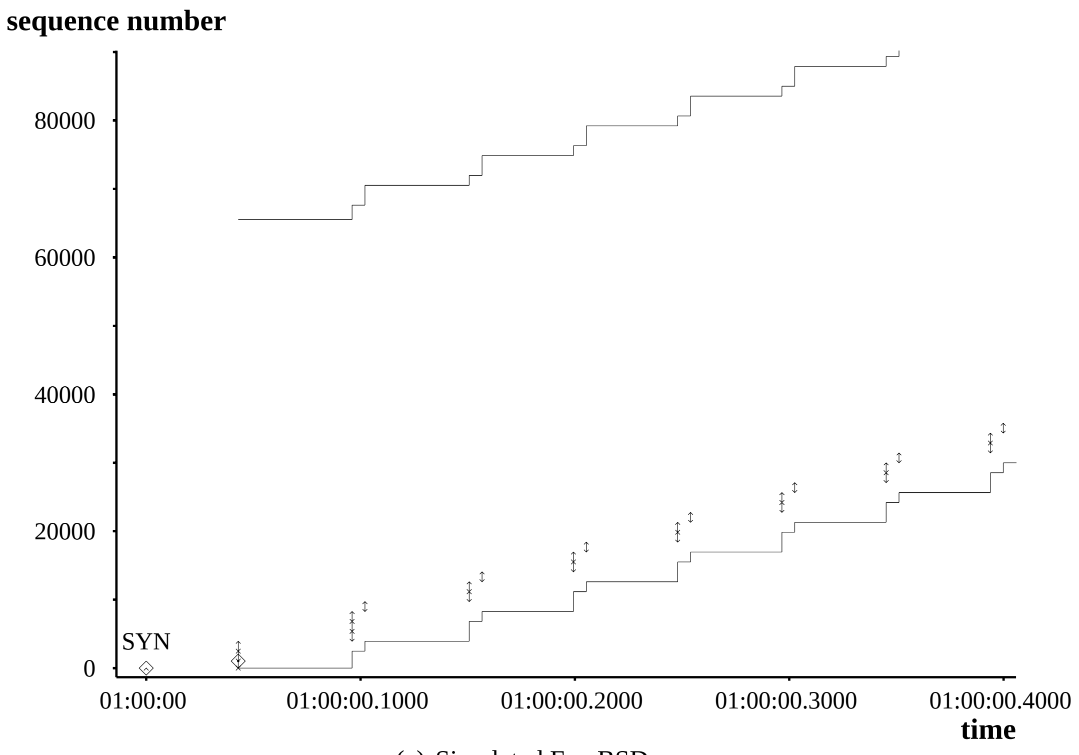
In figures 4.3, 4.4 and 4.5 time-sequence graphs of TCP during connection establishment are shown. These are produced from data collected with a topology as presented earlier in figure 4.1. Dummynet router R1 limits bandwidth to 2Mb/s, delays packets in both directions by 21ms and has a queue length of 10 packets. The simulation scenario is configured to be equivalent. For each operating system (FreeBSD, Linux and OpenBSD) a trace is captured on the testbed and created in simulation. The traces are normalised with tcpnorm then graphed with tcptrace. The two graphs for each operating system are shown side by side.

Each of the pairs of graphs in figures 4.3, 4.4 and 4.5 are very close visual 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.

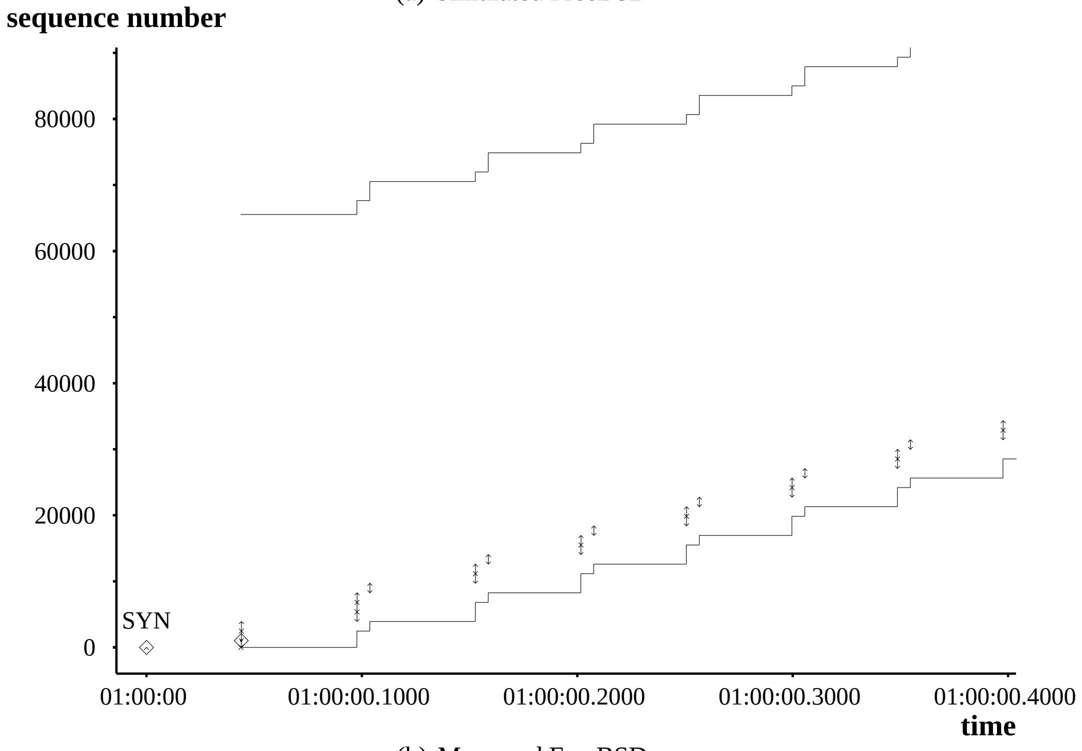
FreeBSD

The two traces for FreeBSD are very close. The textual output of tcpdump shows that the sequence and content of packets illustrated in figures 4.3(a) and 4.3(b) are nearly identical except for the TCP timestamp option. The throughput measured on the emulation network is within 2% of the throughput measured in the ns-2 simulation. The TCP timestamp option differs by one often 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 timing of packets. This is due to the difference in round trip time and variation in timing found in the emulation network, as described in section 4.1.3. The per-packet time difference is plotted in figure 4.6. This graph shows how, in this case, the time differences accumulates over time (this is not always the case for other network stacks tested). This eventually leads to a slightly different ordering of packets.

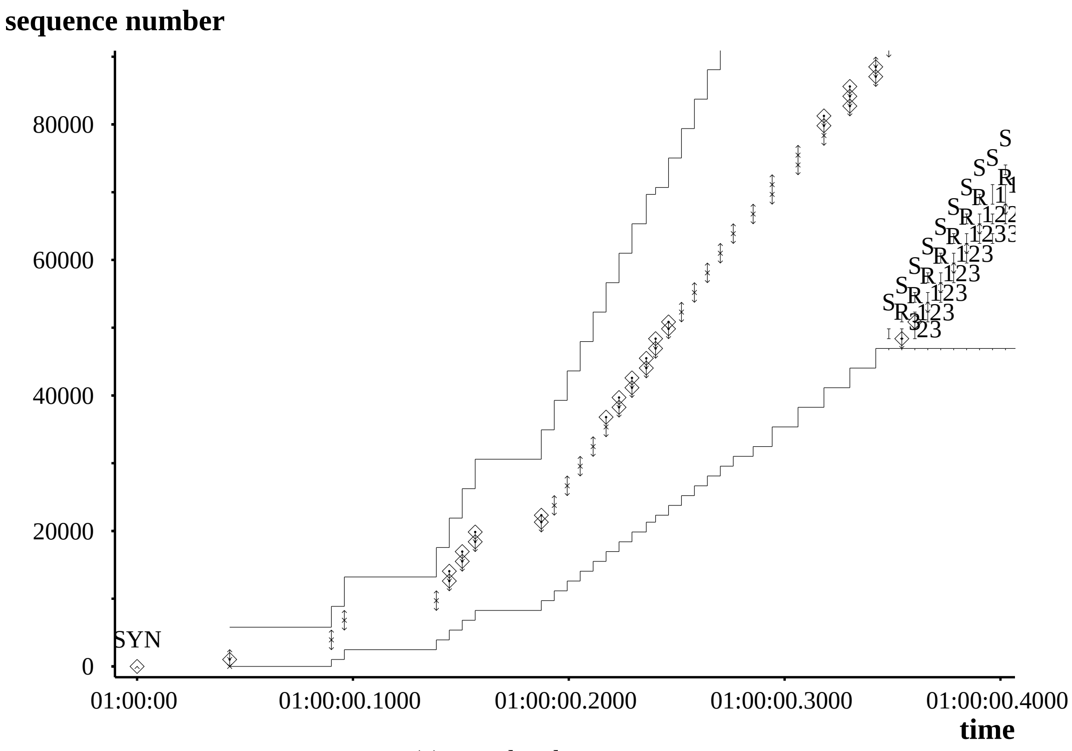


(a) Simulated FreeBSD

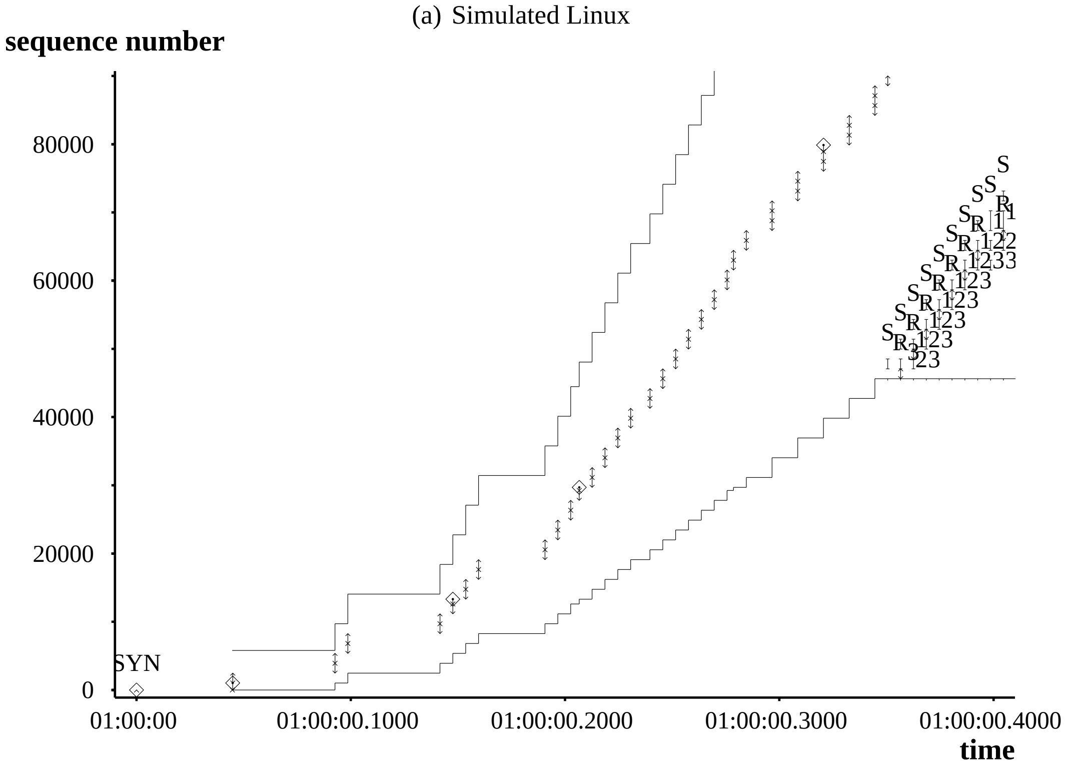


(b) Measured FreeBSD

Figure 4.3: Simulated vs. measured connection establishment graphs: FreeBSD

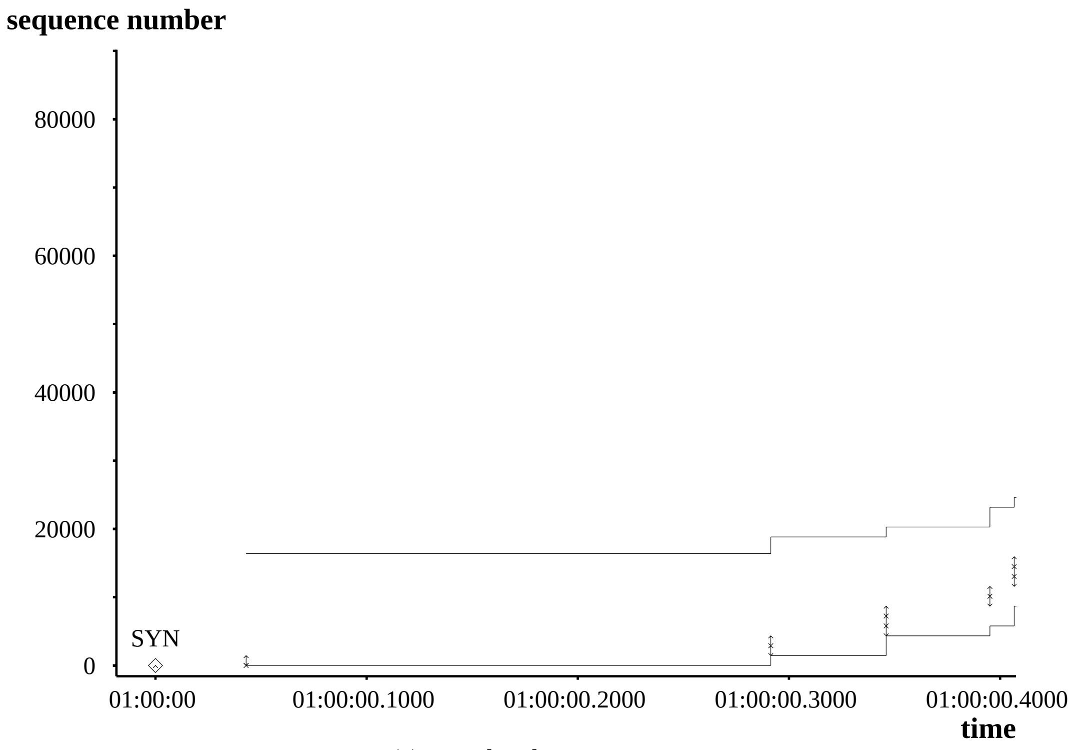


(a) Simulated Linux

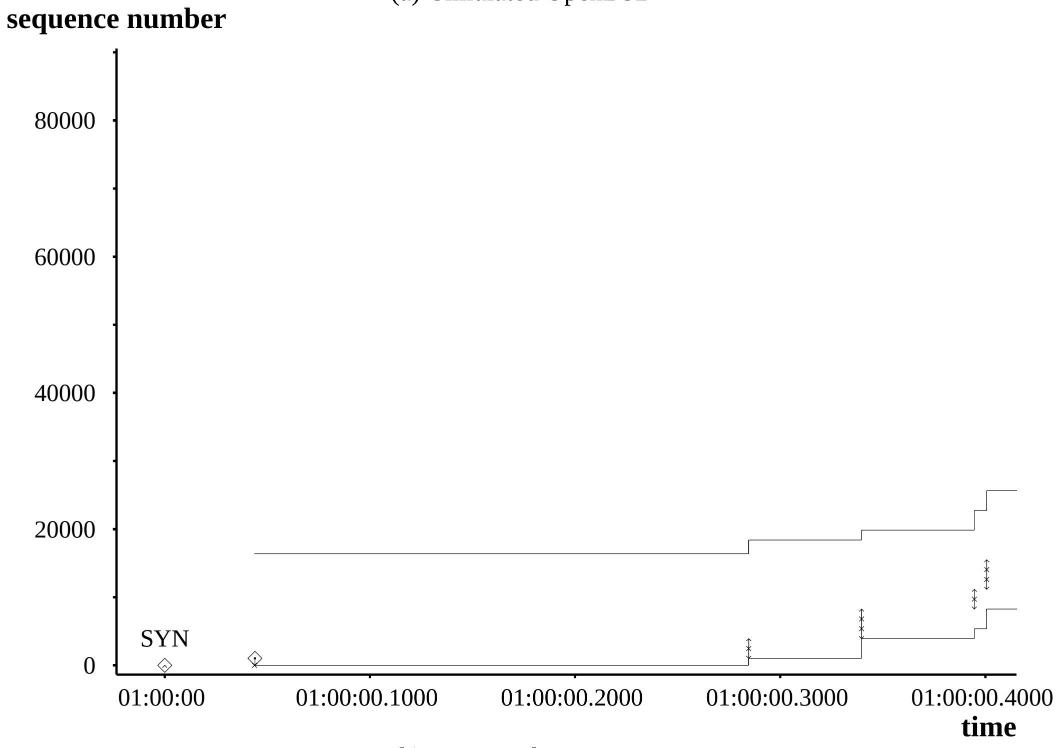


(b) Measured Linux

Figure 4.4: Simulated vs. measured connection establishment graphs: Linux



(a) Simulated OpenBSD



(b) Measured OpenBSD

Figure 4.5: Simulated vs. measured connection establishment graphs: OpenBSD

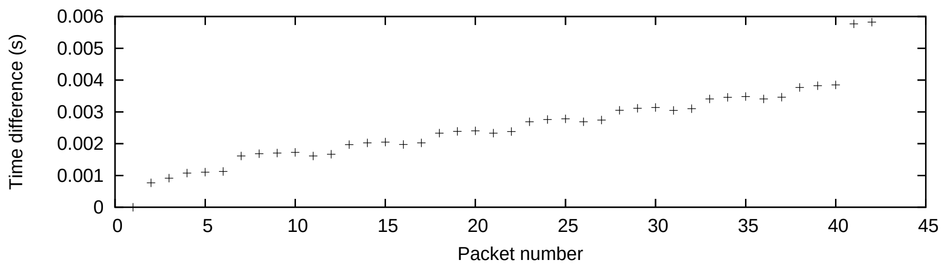


Figure 4.6: Time difference vs. packet number for FreeBSD traces

Linux

The traces for Linux 2.6 look similar in figures 4.4(a) and 4.4(b). 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 [10] to mean that when a receiving TCP sees the flag, it must not wait to receive more data before passing the data to the receiving process. In practice, 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 different in simulation with ns-2 and on a real machine, so the model of the application is not the same between the two. The captured trace from the emulation network shows how Linux sets the PUSH flag more aggressively than the other stacks measured, as the PUSH flag is set on packets after the first data packet with Linux and not with FreeBSD or OpenBSD. This functionality, when combined with the different application behaviour, results in the PUSH flag being set for extra packets in simulation when using the Linux TCP implementation. The PUSH flag is set based on when an application writes to a socket and how large the write is; the application model in ns-2 is not identical to the pattern of writing of the real world test application.

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 and the TCP timestamp option reflects this. This difference is due to both the timer granularity and the limitations of Dummynet introduced earlier in this chapter.

The traces are identical until the difference in PUSH flags save for the slight timing differences described above (approximately the first 20 packets are identical). 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 Linux traces are very similar when visualised with tcptrace and the throughput measured on the emulation network is within 2% of the throughput recorded in simulation.

OpenBSD

The sequence of packets shown in figures 4.5(a) and 4.5(b) are very close matches. When the traces are analysed further it is evident some TCP timestamps vary between the traces by one. This occurs for the same reason it does in the FreeBSD trace and is described earlier.

There are fewer data packets in the graphs showing OpenBSD (figures 4.5(a) and 4.5(b)) due to the OpenBSD sender only sending 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 [152] 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 4.5(a) and 4.5(b) show a timer firing with the same duration in emulation and simulation: the acknowledgement is received which results in further data packets prior to time 01:00:00.3000. The acknowledgement is received at this time due to the delayed acknowledgement timer being set to 200ms. This verifies that this TCP timer is firing at the correct time.

The timing difference of packets is similar to FreeBSD (see figure 4.6). This eventually leads to a different sequence of packets, although an overall tcptrace graph of the connection looks nearly identical and the throughput recorded in simulation is within 2% of the throughput measured on the emulation network.

Linux 2.6 setup

Linux 2.6 (used in the traces analysed here) has dynamic window size determination. This is supported in the Network Simulation Cradle as described below. Linux 2.6 tunes the windows used in TCP based on the amount of memory available in the machine. The cradle code uses memory size equivalent to the machines on the emulation network. The receiver's advertised window grows dynamically and is also affected by the size of the packet structure allocated in the Ethernet driver.

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 on to the network stack. When calculating the receivers advertised window, the size of the `skbuff` is checked as listing 4.1 shows.

Listing 4.1: Linux 2.6 `tcp_grow_window` code

```
int incr;
/*
 * Check #2. Increase window, if skb with such overhead
 * will fit to rcvbuf in future.
 */
if (tcp_win_from_space(skb->truesize) <= skb->len)
        incr = 2*tp->advmss;
else
        incr = __tcp_grow_window(sk, tp, skb);

if (incr) {
        tp->rcv_ssthresh = min(tp->rcv_ssthresh + incr,
            tp->window_clamp);
        tp->ack.quick |= 1;
}
```

In listing 4.1 `skb->truesize` refers to the size of the skbuff allocated in the driver. 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.

4.2.2 Congestion

Figures 4.7 and 4.8 are tcptrace graphs of TCP undergoing loss because it has overflowed the router queue size. The scenario simulated and measured is the same as presented previously in section 4.2.1; these graphs are produced from later in the connection.

FreeBSD

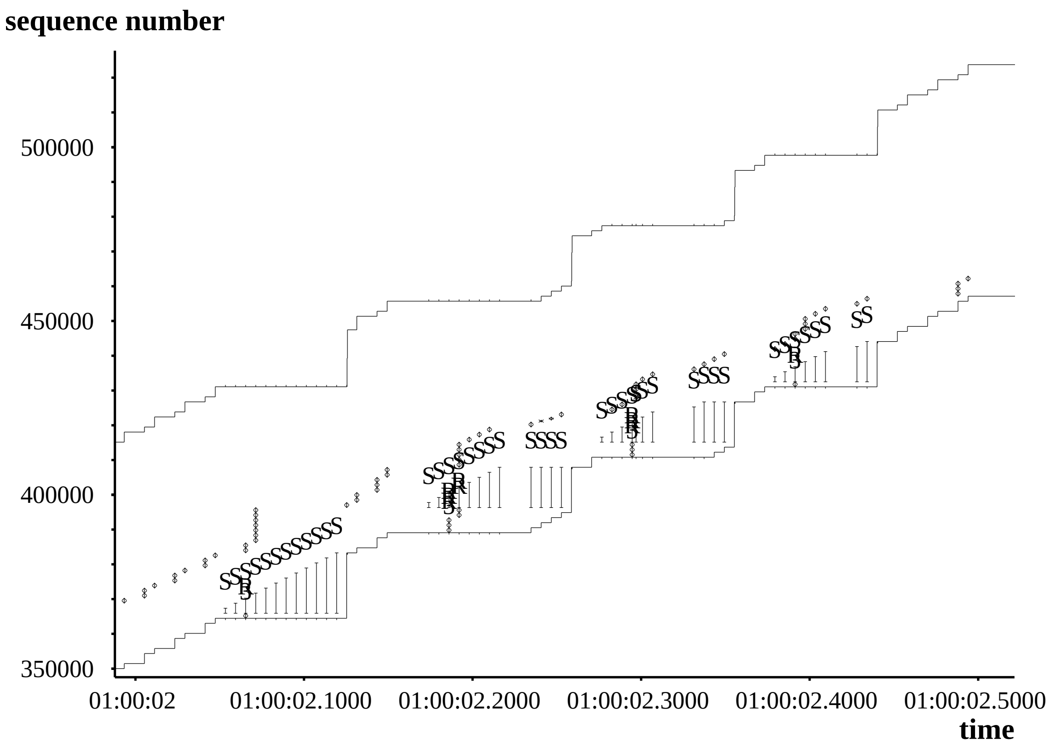
FreeBSD responds to the packet loss in the same manner in simulation and on the testbed network. Figure 4.7(a) and figure 4.7(b) show the same selective acknowledgement ranges and bursts of data packets due to the loss. The two graphs differ in the time and sequence numbers shown on the axis. This is due to loss occurring slightly earlier on the emulation network. This discrepancy is due to the timing difference noted in the earlier discussion of FreeBSD. FreeBSD's response to the different network conditions (due to Dummynet) means that the timing off the loss is different. The graphs show the algorithmic response of TCP is the same in simulation as it is on the real machines.

Linux

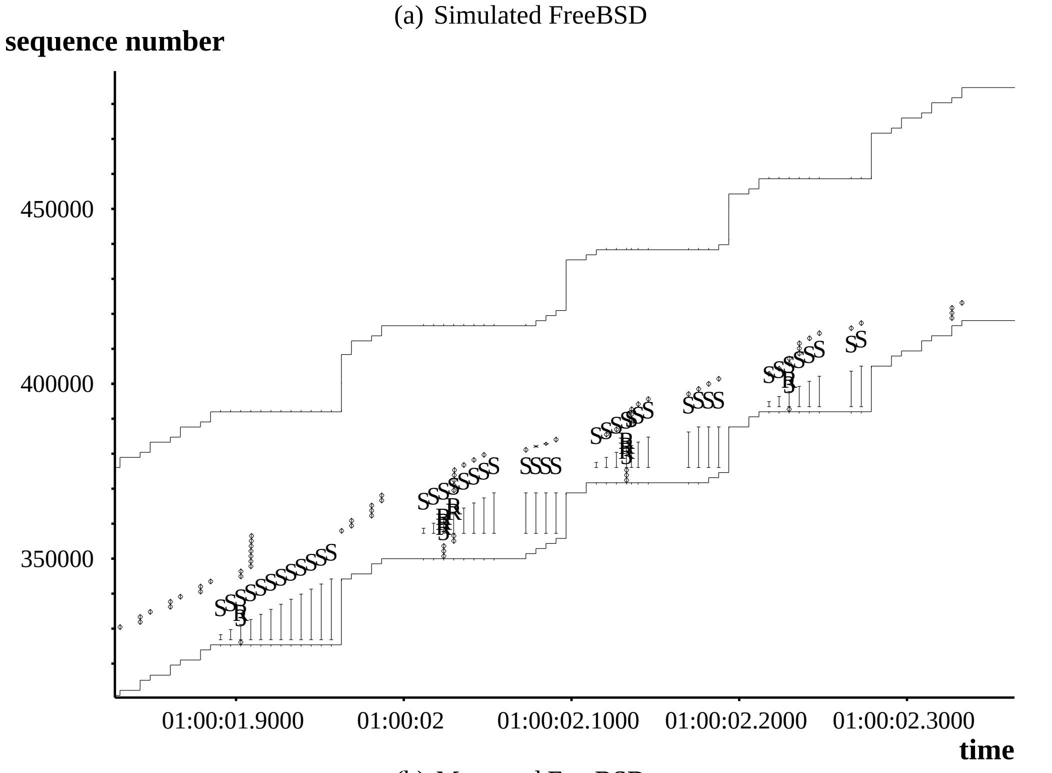
To make the graphs in figure 4.8 easier to understand and compare, the TCP PUSH flag has been omitted. Unlike figure 4.7, the two graphs have the same sequence numbers and time shown. The response to packet loss is identical with a simulated Linux TCP stack and one running on a real machine.

4.2.3 Summary

Comparing packet traces produced by a controlled real world network and equivalent simulation show that the results produced by the Network Simulation Cradle TCP implementations are able to achieve packet-level accuracy. Small differences in timing, TCP options (the timestamp option), and flags (the PUSH bit) occur, but otherwise the sequence of packets produced by the simulated versions of OpenBSD, FreeBSD and Linux are often identical. The exception to this is in some cases a slightly different sequence of packets is produced due to the

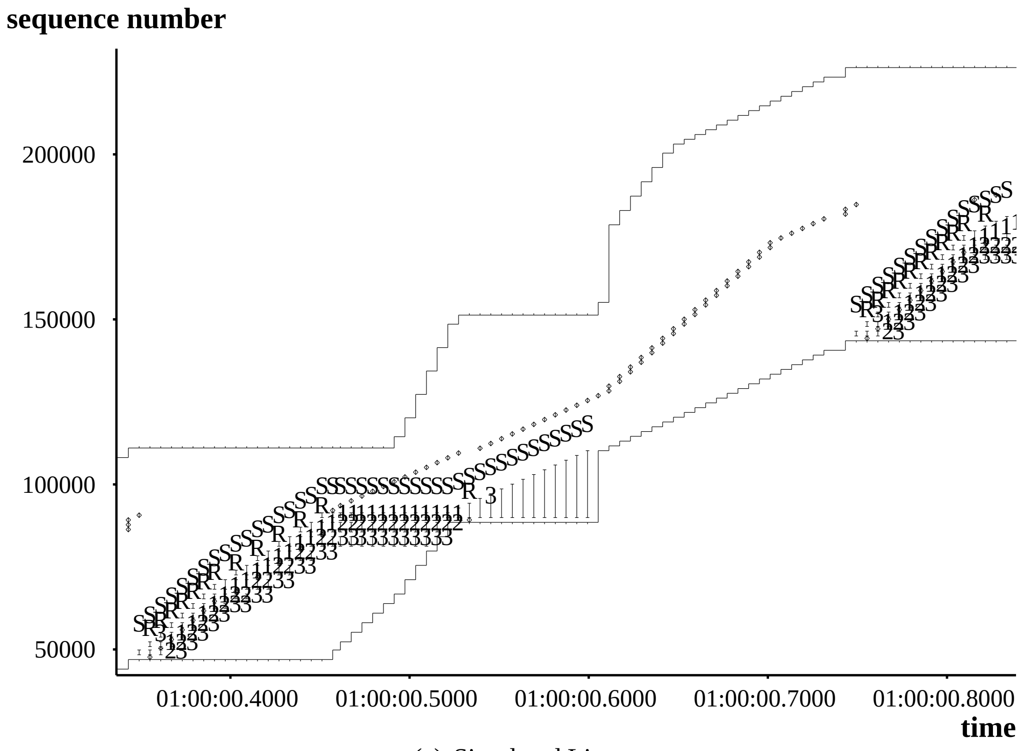


(a) Simulated FreeBSD

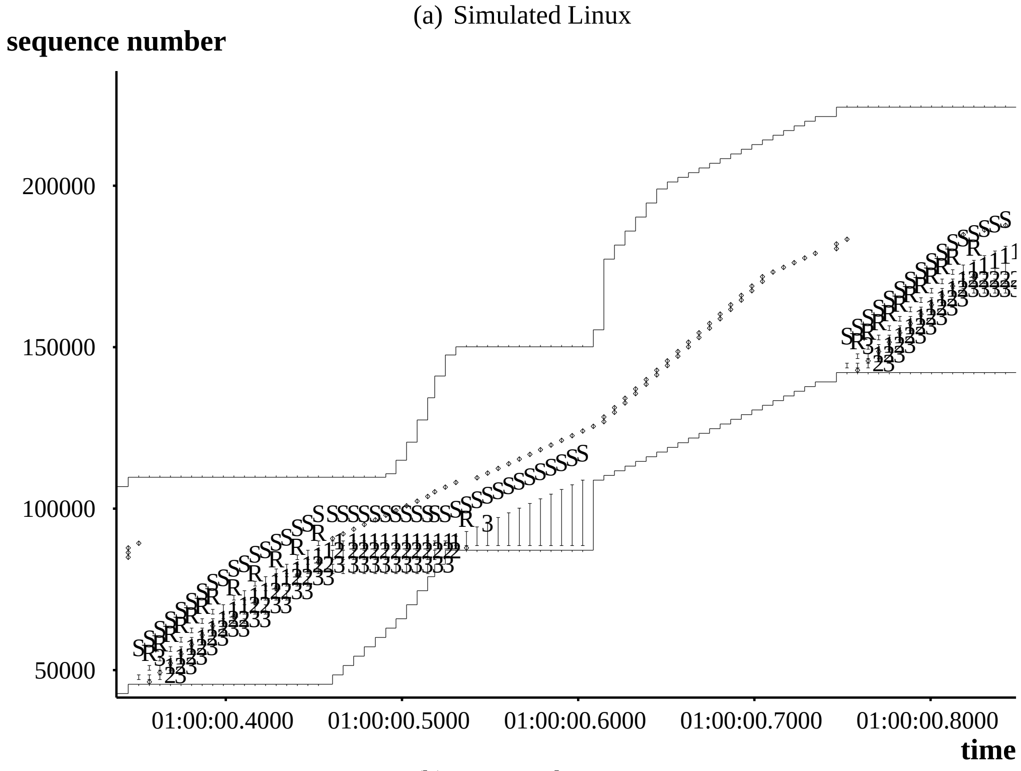


(b) Measured FreeBSD

Figure 4.7: Simulated vs. measured TCP packet loss response for FreeBSD



(a) Simulated Linux



(b) Measured Linux

Figure 4.8: Simulated vs. measured TCP packet loss response for Linux

difference in timing between the real world network and the simulated one.

4.3 Simulated TCP performance

Section 4.2 showed how a high degree of accuracy is attained when using real world TCP implementations in simulation by comparing packet traces at a micro level. This section describes comparisons of the TCP agents found in ns-2 with the NSC TCP implementations at a macro level, reproducing a previously published simulation scenario in section 4.3.1, and ties simulation back to measured results in section 4.3.2.

The results shown in this section compare the NSC TCP implementations to independently developed TCP models (those found in ns-2): a method of validation discussed by Bagrodia and Takai [93] and noted in the introduction of this chapter (page 75).

4.3.1 Performance over a complex topology

The simple dumbbell topology (also known as a *barbell* topology) is often used when conducting simulation based research [153] even though the research is often an Internet study and it is not clear such topologies represent Internet dynamics [153, 154]. The idea that dumbbell topologies are not sufficient to analyse Internet dynamics is discussed by Anagnostakis *et. al* [155] by analysis of Internet measurements and creation of a multiple-bottleneck simulation topology that presents results differing largely from a dumbbell topology.

A reproduction of one of the simulation scenarios studied by Anagnostakis *et. al* is presented here. The network topology and TCP flows that are discussed in [155] are shown in figure 4.9. This setup produces a result that is unlike results attained from using a dumbbell topology: as the number of TCP flows across the central link increases, the aggregate goodput[1] decreases. The reproduction presented here uses the original TCP models used in the study (the models from ns-2) and extends the simulation to use NSC TCP implementations as well. This allows direct comparison between results from simulations performed with the ns-2 TCP models

[1]Goodput is the rate of data received by the application layer from TCP.

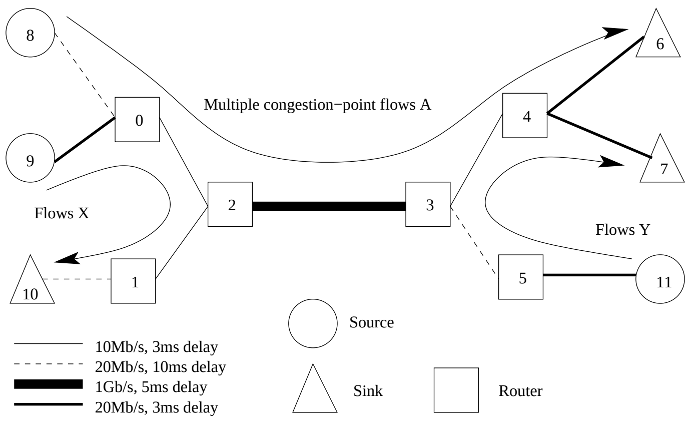


Figure 4.9: Multi-bottleneck scenario (adapted from [155])

and results from simulations performed with the NSC TCP implementations. This process provides further validation of the NSC TCP implementations.

The simulation topology used is shown in figure 4.9, which is equivalent to figure 3 in [155]. As in [155] the number of flows in X and Y of figure 4.9 are fixed at five each. The number of flows in A varies, as does the type of TCP source and sink used for the flows of A. The flows in X and Y use ns-2's Newreno TCP agent with delayed acknowledgements enabled. Simulations last 300 seconds. Start times for all TCP streams are randomly distributed in the interval $[0, 10.0]$, goodput is measured from when all flows have completed connection establishment and the application has received data.

Each set of simulation parameters is simulated 10 times with the random seed varied (the same methodology as Anagnostakis *et. al* [155]). The simulation output statistical analysis procedures discussed by Law [156] are used. The mean (μ) and variance (σ^2) are estimated by:

$$\overline{X}(n) = \frac{\sum_{i=1}^{n} X_i}{n}$$

And:

$$S^2(n) = \frac{\sum_{i=1}^{n} [X_i - \overline{X}(n)]^2}{n-1}$$

An approximate $100(1-\alpha)$ percent ($0 < \alpha < 1$) confidence interval for μ is given by:

$$\overline{X}(n) \pm t_{n-1,1-\alpha/2}\sqrt{S^2(n)/n}$$

Where $t_{n-1,1-\alpha/2}$ is the upper $1-\alpha/2$ critical point for a t distribution with $n-1$ degrees of freedom. Using these formulae the confidence interval for the mean is approximated for each set of simulation parameters. Comparing the half-length of the confidence interval to the point estimate of the mean gives a measure of the precision of the confidence intervals.

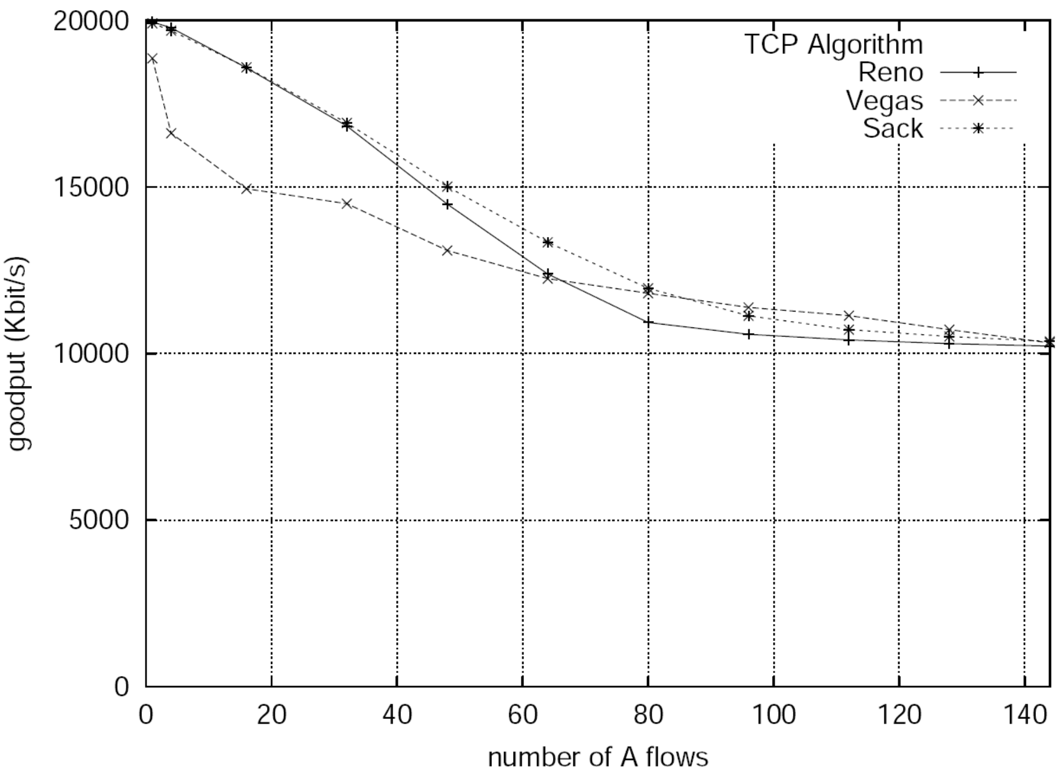
Figure 4.10 presents the results of reproducing the simulation (figure 3 in [155]). The point estimate of the mean is plotted with the confidence interval for each point. The confidence intervals are small enough that they are very hard to see on the graph: they vary between 0.01% and 2%. The results presented here agree with the original research in the case of using the original ns-2 TCP models and when using the NSC TCP implementations: as the number of flows in A increases the aggregate goodput decreases.

Anagnostakis *et. al.* [155] provide a thorough analysis of this result with different queueing mechanisms, queue sizes, TCP models and round trip times. Both the reproduced results from the ns-2 TCP models and the NSC TCP implementations agree with the original research. This result is further evidence that the NSC TCP implementations are valid.

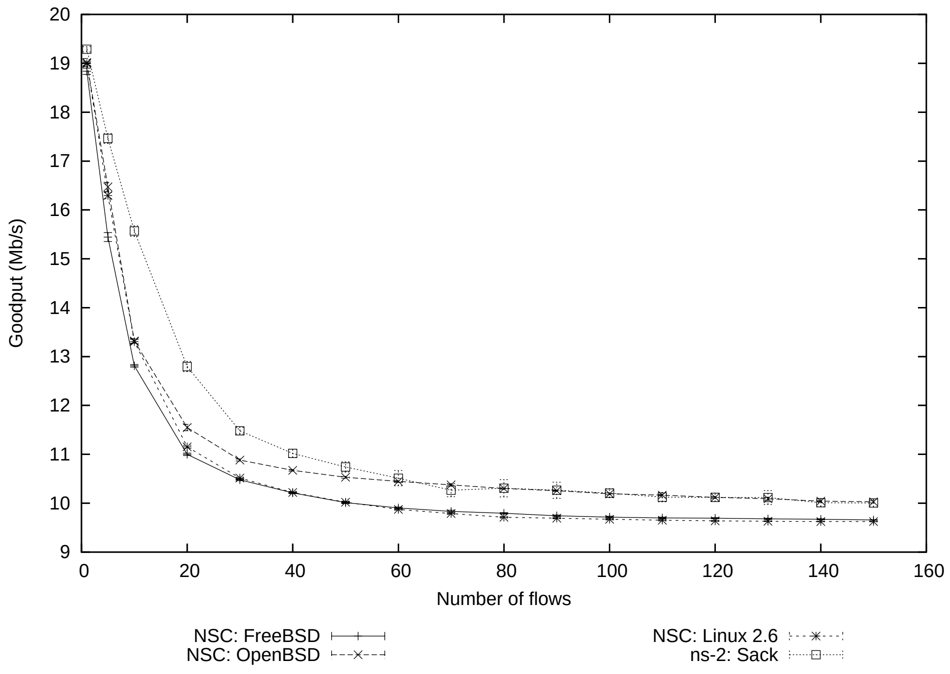
4.3.2 Uniform random loss

Measurement studies have found the presence of random loss on the Internet [157, 158] and uniform random loss is used as a simple model for loss encountered on the Internet [159–161] (or other networks, for example, ATM networks [7]) in many simulation studies. This section presents a study of TCP performance under uniform random packet loss showing comparisons between ns-2 TCP models, NSC TCP implementations and measurements from a test network to validate the NSC TCP implementations.

The performance of TCP during varying uniform random loss rates is presented in figure 4.11. Simulation results using ns-2 with its standard TCP models and with



(a) Original simulation [155]



(b) Reproduced simulation

Figure 4.10: TCP goodput over a multi-bottleneck topology

TCP Implementation	Min	Mean	Max	SD
Linux 2.6.10	164.38	213.98	287.67	22.75
Linux 2.4.27	153.82	207.42	248.70	22.86
FreeBSD 5.3	136.77	176.20	225.01	17.11
FreeBSD 5.2.1	128.74	162.81	219.01	19.56
Windows XP SP2	89.90	137.31	191.00	21.67
OpenBSD 3.5	63.84	117.98	166.82	22.11

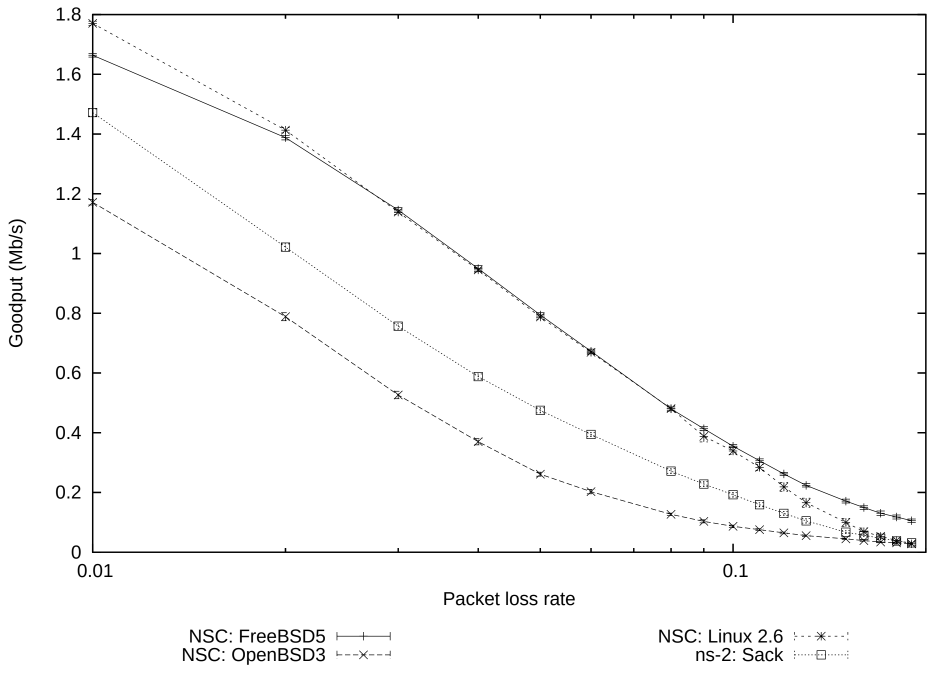
Table 4.2: TCP performance during 5% bidirectional loss [162]

NSC TCP implementations are shown in figure 4.11(a) and results measured from the WAND Emulation Network are shown in figure 4.11(b).

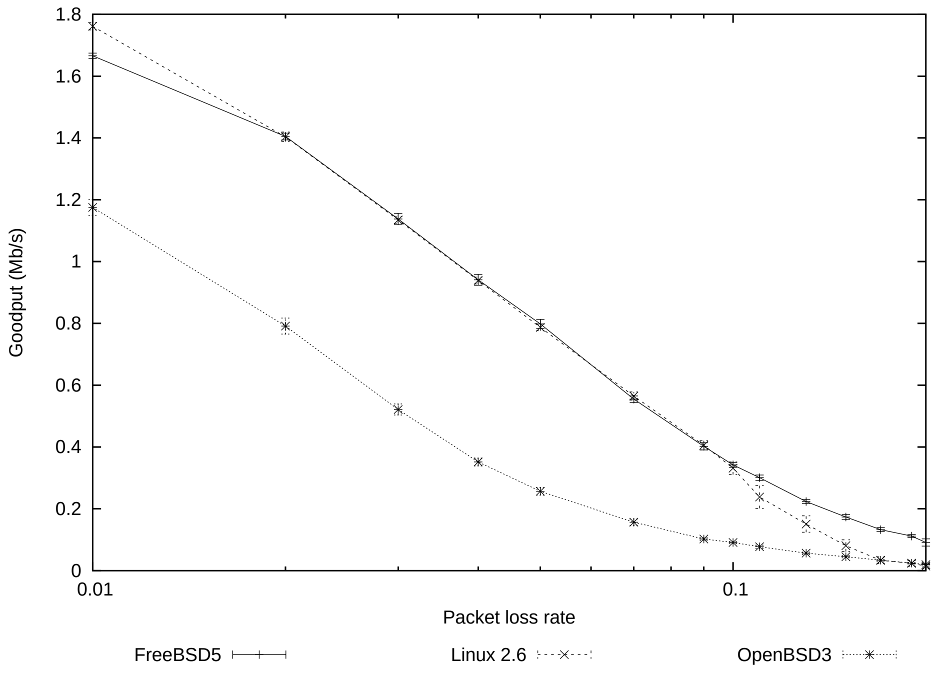
The TCP flow goes through a network with a round-trip time of 40ms and a bandwidth of 2Mb/s. Both graphs show 95% confidence intervals from a number of repetitions of each combination of packet loss rate and TCP variant used. The experiments on the emulation network are run 20 times for each point shown on the graph, while in simulation 50 runs are used to produce tighter confidence intervals. It is simple to run this extra number of experiments with simulation (total time to simulate with one computer was under 3 hours) while running as many tests on the emulation network would have taken many days.

The results in figure 4.11 expand on previous work where we show there is a large difference in performance between TCP implementations during random loss [162, 163]. Table 4.2 shows the results from measuring a set of TCP implementations on the WAND Emulation Network with 5% packet loss. The tests here show how the performance varies as the packet loss rate is increased and give greater insight into the relative performance of TCP implementations under random packet loss.

The simulation results for 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 10%), OpenBSD consistently recording the least goodput and Linux 2.6 dropping from the most 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.



(a) ns simulation



(b) Testbed measurements

Figure 4.11: TCP goodput vs. loss rate

4.4 Summary

At the start of this chapter two methods of validation used by Bagrodia and Takie [93] were described: direct comparison of simulation model and real system and comparison with independent models. This chapter used both of these methods to show that simulating TCP with real world code can produce very accurate results.

Section 4.2.2 showed how the response of two TCP implementations in the Network Simulation Cradle directly matched measurements from real machines on a testbed network. Such direct comparison can not be used for many TCP models, as the abstracted TCP models are often not designed to model one particular TCP implementation.

Section 4.3 presented simulations that compared against the existing ns-2 TCP models which are independently developed and validated [5, 21, 81]. The trends shown agree between the ns-2 TCP models and the NSC TCP implementations for the simple scenarios studied.

Throughout this chapter it is apparent that the different TCP implementations behave differently, even though they all implement the TCP protocol and can communicate between each other. Each implements TCP Sack, slow start, fast recovery and other mechanisms, yet they all differ. This is evident both on testbed measurements (see also our previous work in [162]) and with the simulated implementations. In the small set of simulations presented in section 4.3 the ns-2 TCP models are able to simulate the basic trend of TCP performance in a scenario but do not give any information on the variability between real TCP implementations. The NSC's ability to show the range of outcomes from different stacks and the value of this information is investigated in chapter 5.

Chapter 5
Variation between TCP implementations

The implementation of real world TCP stacks in a network simulator is described in chapter 3 and validity tests are presented in chapter 4. The simple set of tests in section 4.3 shows that the performance varies between TCP implementations and models. This chapter reports further on this variation by presenting a set of simulations that show the extent of performance difference between TCP implementations.

Section 5.2 shows simulations that were created to analyse the difference between TCP implementations and models. These simulations show how, even in very simple situations, there can be large differences in TCP performance between the various TCP stacks.

Following this section 5.3 presents simulation scenarios reproduced from previously published work. This gives further insight into the variation between TCP implementations in networking scenarios that are actually studied by researchers. These simulations, along with those presented in chapter 4, show that using real world code for TCP simulation models is feasible to carry out practical research and provides useful results.

5.1 On benchmarking TCP

TCP is a complex protocol that has evolved significantly from its original specification in 1981 [10]. Since that time it has been further specified to include many optional performance enhancements [11, 80, 85, 152, 164, 165]. The research community has published ideas to increase TCP performance in various scenarios, some of which are implemented in recent TCP stacks [9, 19, 46, 47] and some of which are not used [44, 45] in recent TCP implementations.

Often simulation models are built to test a single new TCP enhancement while TCP implementations may include the new idea as an option. The permutations of options and parameters for options can be enormous. For example, the Linux TCP/IP stack version 2.6.12 has 45 TCP options that can be modified via the `sysctl` program. Some options are boolean (e.g., whether selective acknowledgements are enabled, whether timestamps are enabled) while others can be tuned (e.g., the number of connection attempt retries, default window size for send and receive buffers).

In addition to the many TCP configuration options and algorithms available there are many types of network to test on and many metrics to test. The Internet Research Task Force Transport Modeling and Research Group lists 11 metrics for evaluating congestion control algorithms [166] and 17 tools and characteristics to test with simulation or test bed studies [167]. The range of possible networks and scenarios that can be simulated is infinite; attempts at benchmarking TCP do so by limiting the search space to a set of parameters which are designed to be representative of today's Internet [168].

The large parameter space for TCP performance evaluation means that designing thorough benchmarks of TCP variants is difficult. In this chapter, the simulations presented are not intended to provide a benchmark for TCP performance. Instead, they show that using real world code to simulate TCP is valuable because TCP implementations differ amongst themselves, even in simple scenarios (see section 5.2) and that using such TCP models in practical research provides additional insight (see section 5.3).

5.2 New simulation scenarios

This section presents simulations created to test the variation in performance between TCP implementations and models. TCP during severe packet reordering is studied in section 5.2.1. Linux TCP obtains very different results to the other TCP implementations and models studied in this scenario. Following this, a large number of simulation results over a dumbbell topology with different bandwidths, delays and queue sizes is presented in section 5.2.2.

5.2.1 Packet reordering

The results of a simulation scenario with substantial packet reordering due to packets being randomly delayed between a TCP source and sink are shown in figure 5.1. The TCP stream is limited by a bottleneck link of 4Mb/s and has a round trip time of 104ms. Data is transferred uni-directionally and packets travelling in the direction of the data are delayed by an exponential random variable. The scale factor (μ) of the exponential distribution is shown on the x-axis of the graph. At $\mu = 0$ no packet reordering occurs. Each point on the graph was generated from the mean of 20 simulation runs with differing random seeds. The confidence intervals calculated using the methodology outlined in section 4.3.1 are plotted on the graph but are too small to see, the half-lengths of the confidence intervals range from less than 1% to 3.2%: 20 simulation runs is enough to produce tight confidence intervals.

The ns-2 TCP models of an `Agent/TCP/Sack1` source and `Agent/TCPSink/Sack1/DelAck` sink have very similar results to the FreeBSD and OpenBSD network stacks simulated with the Network Simulation Cradle. There is however a large difference between these and the two versions of the Linux TCP stack tested.

The Linux TCP/IP stack has several mechanisms implemented to aid TCP performance during packet reordering [169]. Duplicate selective acknowledgements [48] (DSACK) [46] help distinguish between packet loss and packet reordering. The Linux TCP/IP stack uses TCP timestamps to help detect

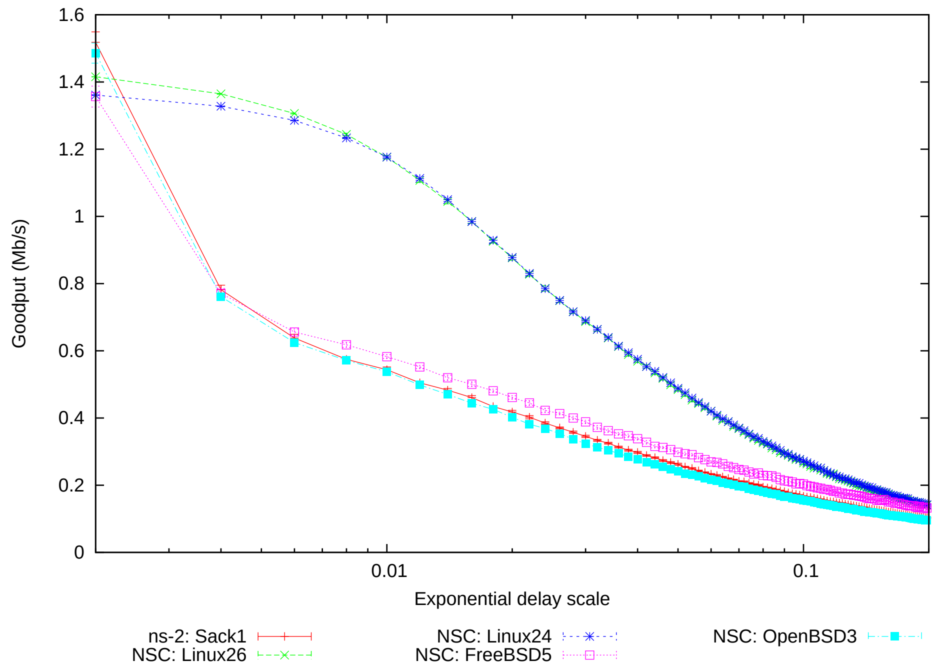


Figure 5.1: TCP goodput under packet reordering

spurious retransmissions similar to the TCP Eifel [170] algorithm. The forward acknowledgement algorithm [19] is also implemented and can help in this scenario. This shows how a range of values from real world implementations are possible in a given scenario; further insights into the situation being simulated are possible with a range of TCP implementations available for simulation.

5.2.2 Many TCP flows over a dumbbell topology

The simulation scenario presented in this section is an attempt to characterise how queue size, cross traffic, competing TCP type, bandwidth and delay affect TCP goodput in a dumbbell topology. Figure 5.2 shows the simulation scenario. The flows F and R have uniformly distributed RTTs in the interval $[8, 222]$ms. The number of flows in F is selected from $[0, 5, 20, 55, 100]$ and R is selected from $[0, 5]$. The routers on the bottleneck link have queue sizes selected from $[6, 10, 12, 15, 18, 30, 50]$ packets. The bandwidth of the bottleneck link is selected from $[0.512, 1, 2, 4, 6, 8, 10]$ Mb/s. Each set of parameters is simulated with 10 random seeds. Flow M is measured and has an RTT of 8ms. The TCP model is varied and goodput recorded after 200 seconds of simulation time.

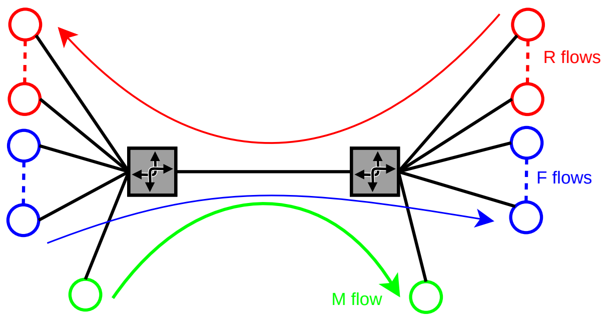


Figure 5.2: Simulation scenario

Table 5.1: Simulation machines used

CPU Type	Cache size	RAM	Number
Intel Pentium 4 2.60GHz	512 KB	512 MB	32
Intel Pentium 4 2.80GHz	1 MB	512 MB	47
AMD Athlon XP 2200+	256 KB	256 MB	19
AMD Opteron 250	1 MB	8 GB	1
Total			99

To simulate this range of parameters 112500 independent simulations were run. The simulations were spread over a set of 99 computers. The specifications of the simulation machines are summarised in table 5.1. A total of 4.98 CPU-years were spent simulating.

Direct TCP model performance comparisons

It is not trivial to compare TCP models from the results of the simulations due to the large parameter space and number of results. Due to the number of parameters being varied, 2 or 3 dimensional graphs do not have enough dimensions to display the parameter space.

Figure 5.3 shows comparisons of some of the TCP variants by plotting the difference in measured goodput. Only three sets of graphs are shown for brevity. Each point on the graphs on the left is the comparison of two simulations run with identical parameters apart from the TCP model used for the measured flow. A positive value on one of these graphs means the first TCP model attained more goodput than the second TCP model. For example, a point at $y = 1$ on the left graph of figure 5.3(a) means the Newreno ns-2 model was measured to have twice the goodput as the Sack ns-2 model. This data is presented as a cumulative percentage plot in the graphs on the right, where a positive value is counted on the right side of the graph.

Figure 5.3(a) shows the comparison of the ns-2 TCP models for Newreno and Sack. The comparison shows only a slight bias towards Newreno, both attain more goodput than the other a small percentage of the time. Greater differences are shown in figures 5.3(b) and 5.3(c). Linux 2.4 achieves more goodput than FreeBSD in many cases. This is evident on both graphs in figure 5.3(b). The difference between Linux 2.6 and FreeBSD is larger yet, with Linux 2.6 attaining more goodput than FreeBSD approximately $2/3$ of the time, only a very small percentage of the time is more goodput recorded for FreeBSD than Linux 2.6.

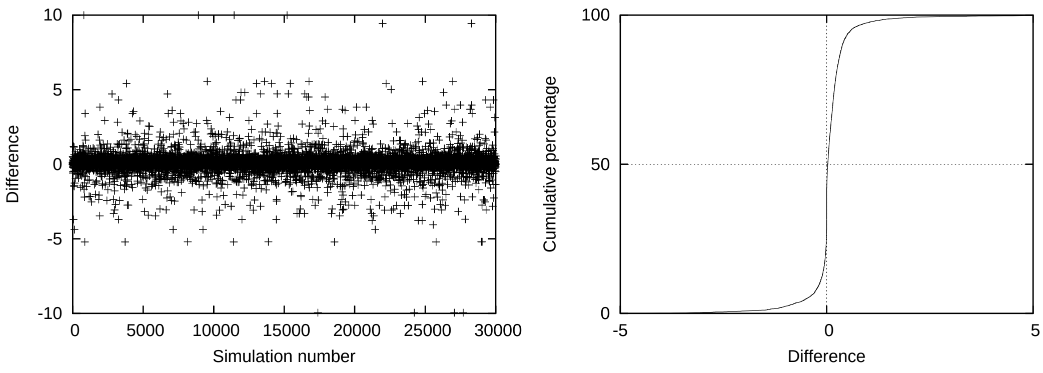
Not shown in figure 5.3 are graphs comparing the ns-2 models with the NSC models. These also show large differences, comparing ns-2's Sack with Linux 2.6 produces a graph similar to 5.3(c). These results show that even in a very simplistic scenario there can be large differences in goodput of the TCP implementations studied. The next section shows further analysis of the results, characterising some of the reasons for such differences in performance.

Characterising the differences

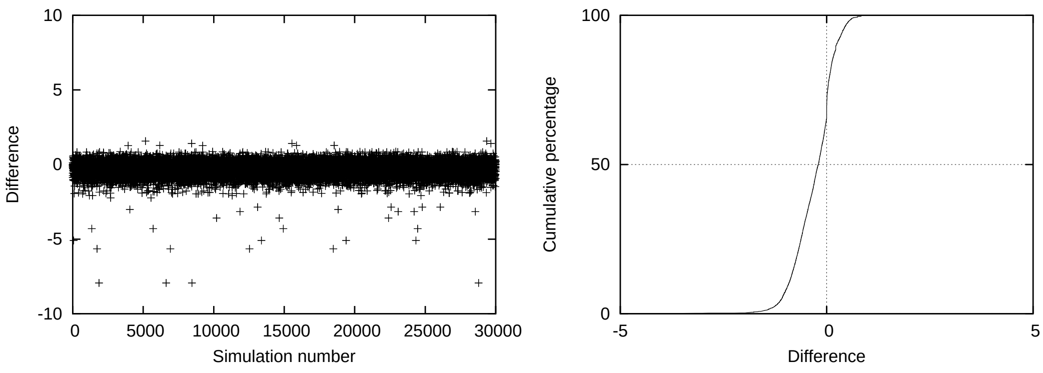
The graphs in figure 5.3 show that there is a difference between simulated TCP models but do not give any insight into which simulation parameters are causing the difference. The Weka [171] machine learning software was used to help analyse the large data-set. Weka implements many machine learning algorithms including classifiers (decision trees, rules, regression and Bayes), clustering algorithms, association and attribute selection.

Weka's attribute selection algorithms rank the attributes (parameters) on their importance in predicting a single parameter: the class value. The class is set to the goodput in all results presented. Some machine learning algorithms require the class value to be nominal, where goodput is numeric (continuous). In such cases the goodput is discretised into 10 bins of equal frequency.

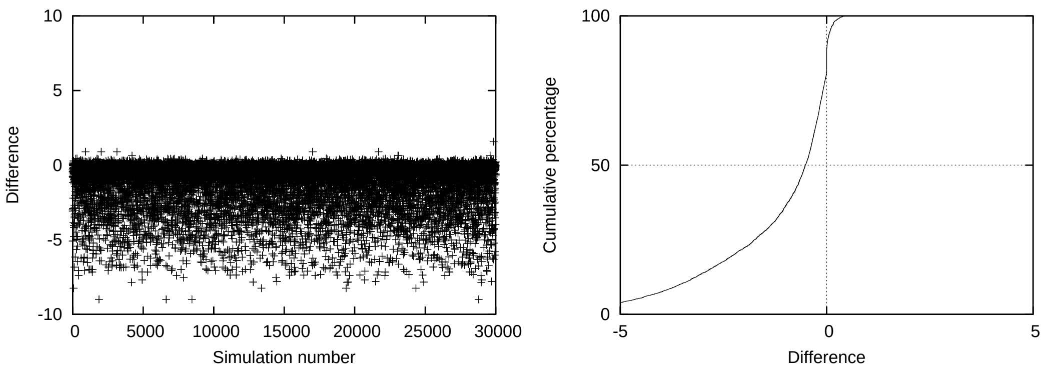
Listing 5.1 shows the output from Weka running attribute selection using the information gain attribute evaluator. This ranks the attributes in order and assigns an information gain value to each attribute. The information gain algorithm is a



(a) ns-2: Newreno vs. ns-2: Sack



(b) NSC: FreeBSD5 vs. NSC: Linux 2.4



(c) NSC: FreeBSD5 vs. NSC: Linux 2.6

Figure 5.3: TCP performance comparisons with cumulative graphs

Listing 5.1: Weka output for information gain attribute evaluator

```
Ranked attributes:
 0.60195565   Bandwidth
 0.60009471   Forward flows
 0.04666604   TCP type
 0.01071284   Reverse queue
 0.01071284   Forward queue
 0.00430134   Seed
 0.00293595   Reverse flows
 0.00000376   Cross TCP type
```

Listing 5.2: Weka output for CFS attribute evaluator

```
Forward queue
Reverse flows
TCP type
Seed
Bandwidth
Forward flows
```

Listing 5.3: Information gain attribute evaluator for differences data

```
Ranked attributes:
 0.805076263   Forward flows
 0.185627249   Forward queue
 0.185627249   Reverse queue
 0.082538220   Bandwidth
 0.044354906   Reverse flows
 0.000000192   Cross TCP type
 0             Seed
```

simple and fast ranking method that uses a measure of the change in entropy before and after observing an attribute [172]. This algorithm is often used in text categorisation applications where the dimensionality of the data is high [172].

Bandwidth and the number of flows in the forward direction are the greatest predictor of goodput. The size of the data set limits the potential machine learning algorithms that can be practically applied. Correlation-based Feature Selection [173] (CFS) is a sophisticated algorithm that is applicable to large data sets. CFS does not rank the attributes but computes a subset of the attributes that it considers to be the most important to predicting the class value. The attributes selected by CFS are shown in listing 5.2. The cross TCP type and reverse queue are not included in this output. The results from these attribute selection algorithms are consistent with the expected outcome as when bandwidth is low and there are many competing flows, the resulting goodput of the measured flow will be low.

Using the differences in goodput between TCP models run with the same set of parameters (the same data as presented in figure 5.3) results in some similar findings from attribute selection. The results of processing this data with the information gain algorithm are shown in listing 5.3. Forward flows is again a good predictor, though bandwidth much less so at predicting the difference in goodput between TCP models. CFS includes all attributes shown in listing 5.3 except cross TCP type and reverse queue.

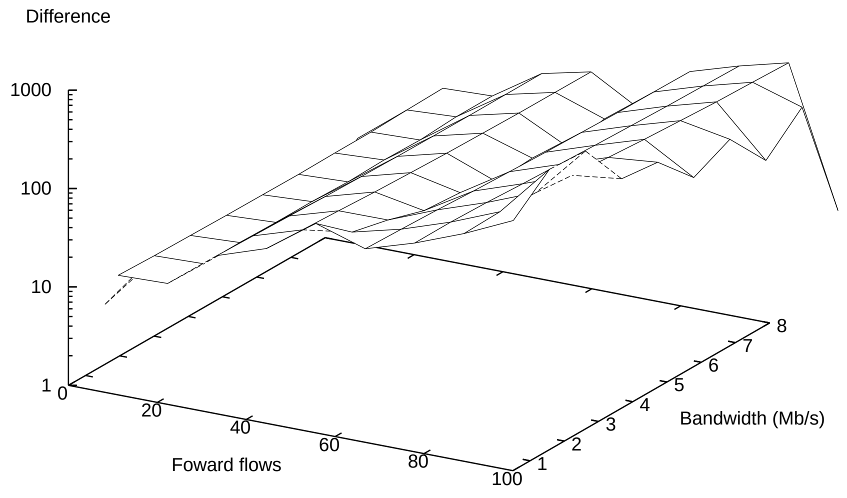


Figure 5.4: Mean goodput difference as flows and bandwith is varied

It is evident from viewing the raw data sorted by the difference in goodput that the largest differences are due to extreme circumstances: many flows with small queue sizes and small bandwidths. The results from Weka presented earlier support this conclusion. Often in such cases, no goodput is recorded for the ns-2 TCP models of Newreno and Sack, as their connection establishment fails, where the real world implementations are able to connect and send data. Figure 5.4 shows a visualisation of the mean difference encountered as flows and bandwidth are varied. This shows how at a low bandwidth and high number of flows the difference is the greatest and there is a general trend towards higher differences as the number of flows is increased.

The results of this study show large variations in recorded goodput for various parameter permutations of a very simple simulation scenario. The study is not intended to realistically model a particular real world configuration, rather it is designed to explore how different TCP models and implementations respond to similar situations. Generating large amounts of performance data from various real world TCP implementations is something that is difficult and resource intensive (over 100 computers would be required just to run the scenario presented here and it would take 280 days to perform with this amount of hardware) without a framework like the Network Simulation Cradle. The cradle, with its multiple TCP implementations, makes possible–and easy–comparative performance studies of TCP *implementations* over a range of networks and parameters with much less hardware and time requirements than testing on real networks.

5.3 Reproduced simulations

This section shows the use of the Network Simulation Cradle in reproductions of simulations and experiments conducted in a range of TCP based research. The results in this section show again that using real world TCP implementations in research is feasible for actual research undertaken with TCP simulation and more so that useful results and insights are possible from using such implementations.

During the course of the research many simulation scenarios were reproduced. Shown in the following sections are scenarios of interest to show the sorts of differences encountered when simulating with the NSC as well as the ns-2 TCP models. For some reproduced simulations, the results of using the NSC stacks is the same as using the ns-2 models, these simulations are not covered here. The following results do not show that ns-2 simulates TCP incorrectly; rather they show that in some situations the lack of detail in the ns-2 models is important to the result.

5.3.1 TCP fairness on high-speed networks

TCP over long distance fast networks is an active research area: TCP increases its window very slowly and is sensitive to packet loss, resulting in low link utilisation on many fast long distance networks. This is due to the combination of the round trip time being large and the congestion window size required to make full use of the bandwidth being large. It takes many round trip times for TCP to increase its window to be large enough to fill the network with packets; when packet loss is encountered this window is halved and the process needs to begin again

Various schemes have been invented to alleviate this problem, while remaining compatible with TCP. BIC-TCP [9], HSTCP [174], FAST TCP [59], H-TCP [57] and Scalable TCP [58] are examples. They also often have problems with fairness (sometimes exacerbating TCPs inherent RTT unfairness). Convergence times for these proposals vary [175].

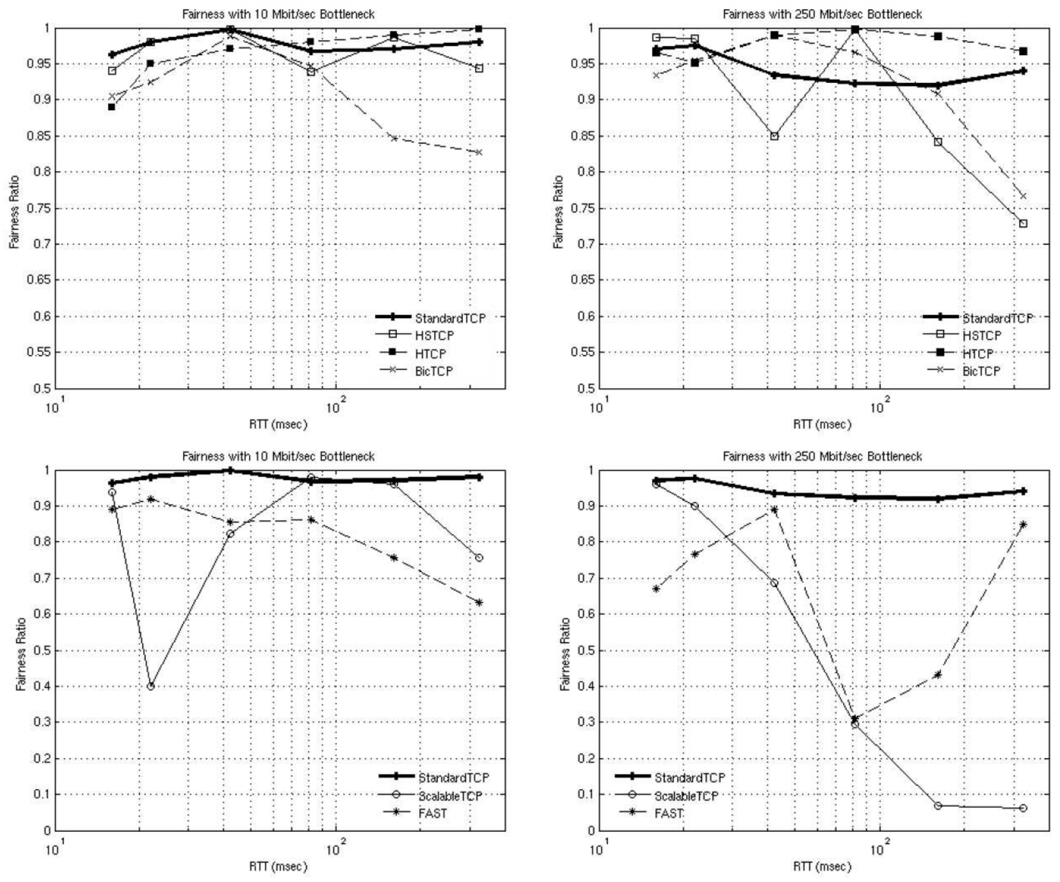
These TCP modifications have been tested both in simulation and on testbeds [176]. The simulations in this section reproduce experiments conducted on testbeds presented by the Hamilton Institute technical report [175]. This report has been widely cited in research since being published (e.g., [176, 177]) and is noted as a reference for the IETF Transport Modeling Research Group led by Sally Floyd [178].

Li *et. al* [175] aim to "compare the performance of competing TCP protocols in a systematic and repeatable manner." They define and use a set of benchmark tests to compare proposals to increase TCP's performance on high bandwidth-delay product networks. In this section a reproduction of one of their experiments is presented.

Figure 5.5 shows the original graphs presented as figure 4 in [175]. The ratio of the throughput of competing variants of TCP compared to "standard TCP" is shown for different bottleneck speeds. Figure 5.6 shows results for ns-2 and NSC TCP stacks in the same environment.

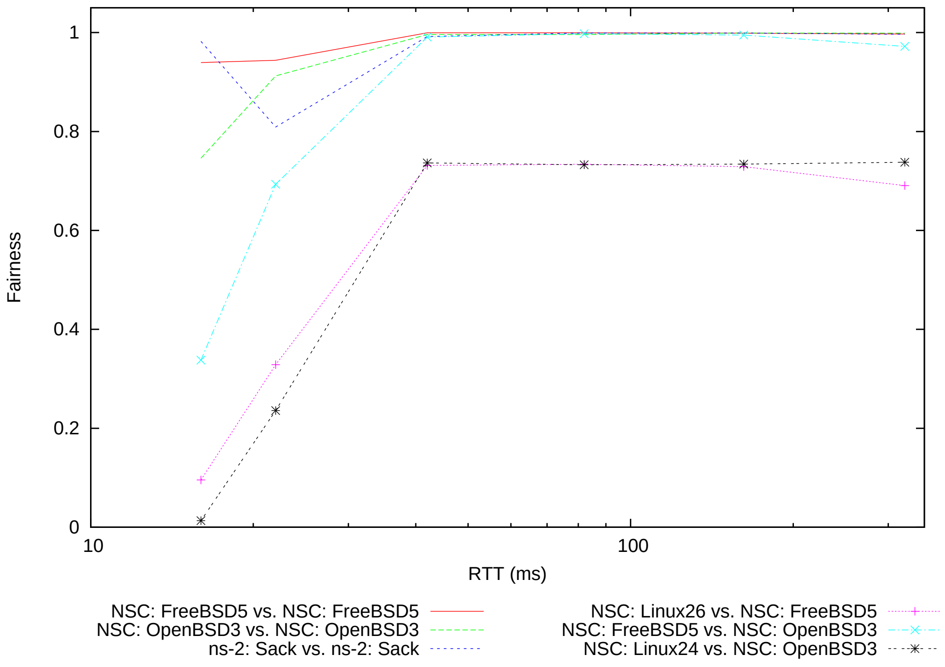
The topology used in the experiments is a dumbbell topology. Path propagation delay (and hence round trip time) is varied and the fairness between two TCP flows is measured. The fairness is defined as the ratio of goodput achieved by the two flows after 60 seconds. The queue size is set to 20% of the bandwidth-delay product. Flow start time is jittered by up to one RTT and each set of parameters is simulated with 5 random seeds. The graphs show the mean fairness over the 5 simulations for each data point.

The baseline or "standard TCP" cases shown in figure 5.5 are reproduced and presented in figure 5.6. The lines on the graphs without points show the results that are consistent between the original experiments and the reproduced simulations. The fairness measured is near to 1 for RTTs greater than or equal to 40ms, meaning the two TCP flows equally (fairly) share the link bandwidth. At low RTTs on figure 5.6(a) the fairness is less stable. As queue size is based on the bandwidth-delay product in this experiment, when both the RTT and bandwidth

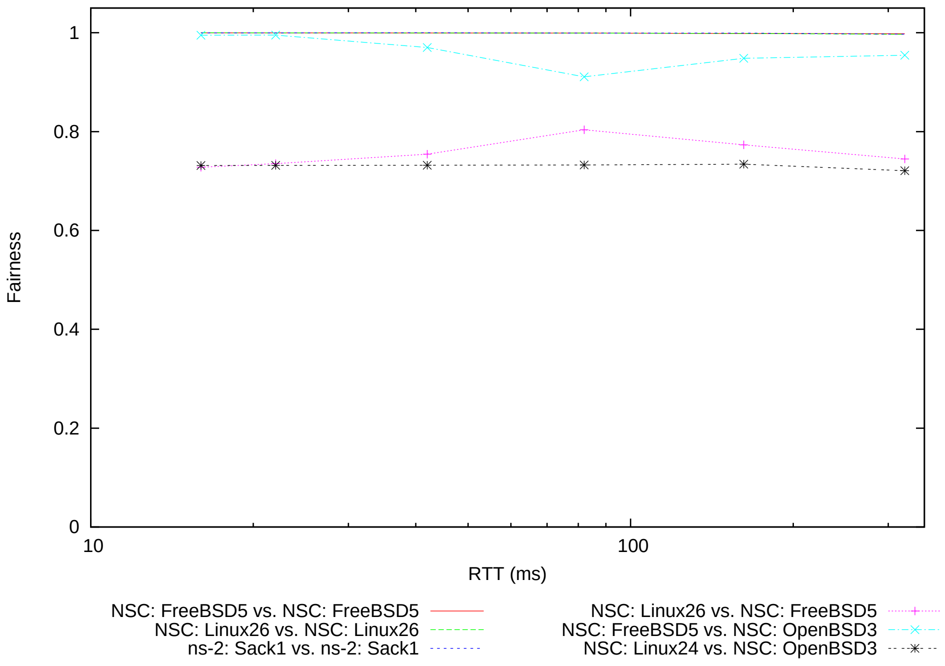


Ratio of throughputs of competing New-TCP and standard TCP flows as path propagation delay is varied. Results are shown for 10Mbit/sec (left) and 250 Mbit/sec (right) bottleneck bandwidths. Both flows have the same RTT. Queue size is 20% BDP.

Figure 5.5: Ratio of throughputs of competing TCP flows [175]



(a) 10Mb/s network



(b) 250Mb/s network

Figure 5.6: Fairness between two TCP flows as path propagation delay is varied

are relatively low the queue size on the bottleneck router is also very low (as low as 3 packets when RTT is 16ms). Very low queue sizes lead to unfairness as it is easy for one flow to occupy the entire queue, starving the other flow. With higher RTTs and/or a higher bandwidth the results are consistent.

These baseline results are extended by comparing different TCP implementations against each other with the Network Simulation Cradle. The lines shown on the graphs in figure 5.6 without points are for when both TCP flows are from the same implementation. The results on the graphs in figure 5.6 with lines and points show some of the combinations of standard TCP implementations compared against each other. In the context of [175] any of these results could be the “standard TCP”. Perhaps none of them should be used as a standard TCP reference point because no one implementation captures the range of results shown here. Instead, several TCP implementations need to be used to obtain this information.

Li, Leith and Shorten [175] compare new TCP variations such as BIC TCP and H-TCP against their standard TCP, which is the Linux 2.6.6 TCP implementation. The results in figure 5.6 show that there are large differences in fairness between standard TCP implementations, as much as between some of the high-speed TCP variants at 10Mb/s. Extending the scenario investigated to include further TCP implementations shows that the “standard TCP” presented in [175] does not cover the range of performance results that are encountered with multiple different TCP implementations. Using real world network stacks in simulation means evaluating the scenario discussed in [175] is easy and not the prohibitive amount of work it is without the NSC.

5.3.2 Congestion control comparisons

Grieco and Mascolo [53] compare the Westwood+ [52], New Reno and Vegas TCP congestion control algorithms using simulations with ns-2 and some real world measurements over the Internet. They analyse a series of scenarios: single and multiple bottleneck situations with various link capacities, buffer sizes, and traffic types, wireless links used with Medium Earth Orbit (MEO) and Geosynchronous Earth Orbit (GEO) satellites, and measurements from FTP transfers on the

Internet. Reproduced results from two of the simulation scenarios in [53] are presented in the following sections.

Single bottleneck scenario

A simulation scenario with a single bottleneck is used by Grieco and Mascolo [53] to evaluate goodput and fairness in bandwidth allocation between flows of the same TCP variant but differing RTTs. The topology used is a simple single-bottleneck or dumbbell topology. A varying number of TCP flows, named M henceforth, send data in the forward direction, while 10 TCP flows send data in the reverse direction. All flows in M use the same TCP congestion control mechanism. Round trip times are uniformly distributed in the interval $[20 + 230/M, 250]$ms. M ranges from 10 to 200. Simulations last 2000s of simulated time and the bottleneck link bandwidth is 10Mb/s. This scenario is reproduced with the TCP implementations available in the NSC used as the TCP model for the flows in M.

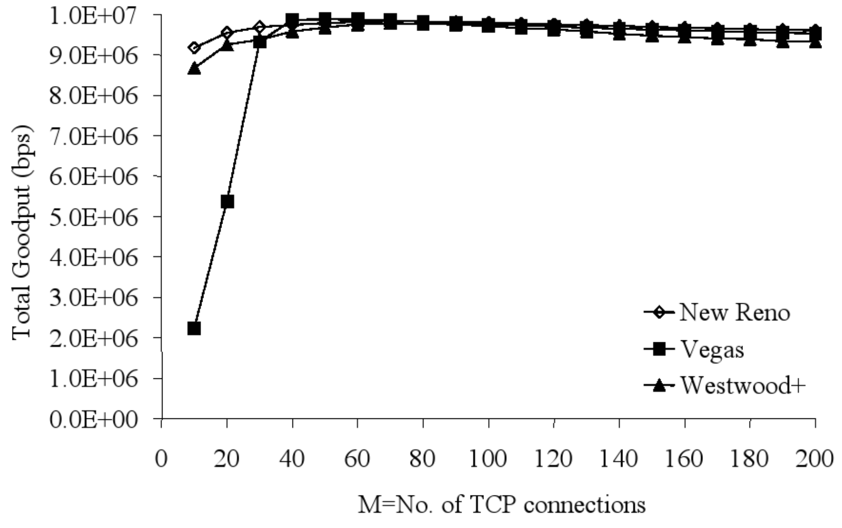
The results presented by Grieco and Mascolo [53] for this experiment are shown in figure 5.7. The results of the reproduction are shown in figure 5.8. Figure 5.8(a) shows the aggregate goodput for all M flows as M is increased. This result agrees with the results shown in figure 5.7(a) as once $M = 40$ the goodput levels out at approximately 9Mb/s. TCP Vegas is not included in the reproduced study.
Figure 5.8(b) provides further insight into this result by showing the variation in fairness between the TCP implementations.

Grieco and Mascolo use the Jain Fairness Index [179] to determine fairness between the flows in M. This index is defined in the following equation:

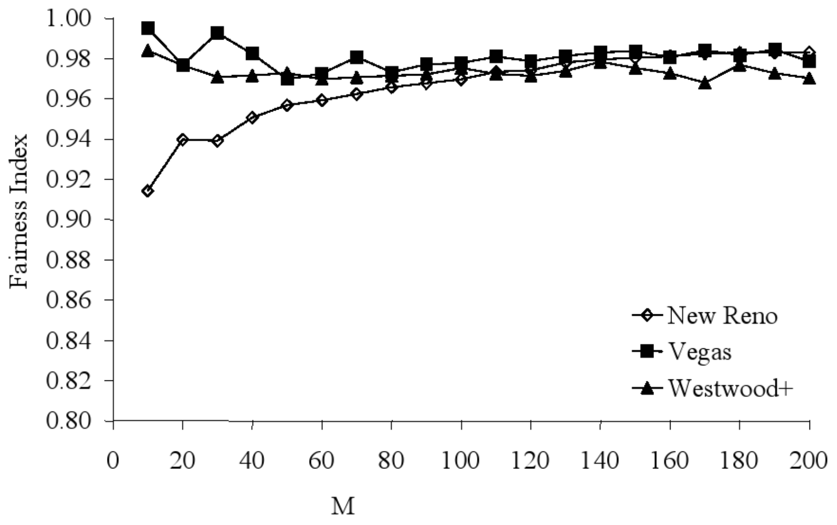
$$J_{FI} = \frac{(\Sigma_{i=1}^{M} b_i)^2}{M\Sigma_{i=1}^{M} b_i^2}$$

Where b_i is the goodput of the i^{th} connection and M are the connections sharing the bottleneck. The index belongs in the interval $(0, 1]$ where 1 is the fairest.

The Jain Fairness Index for the TCP models studied in figure 5.8(a) is plotted in figure 5.8(b). It is evident that while the TCP models achieve similar goodput, the fairness varies. The general trend of increasing fairness as M increases agrees with

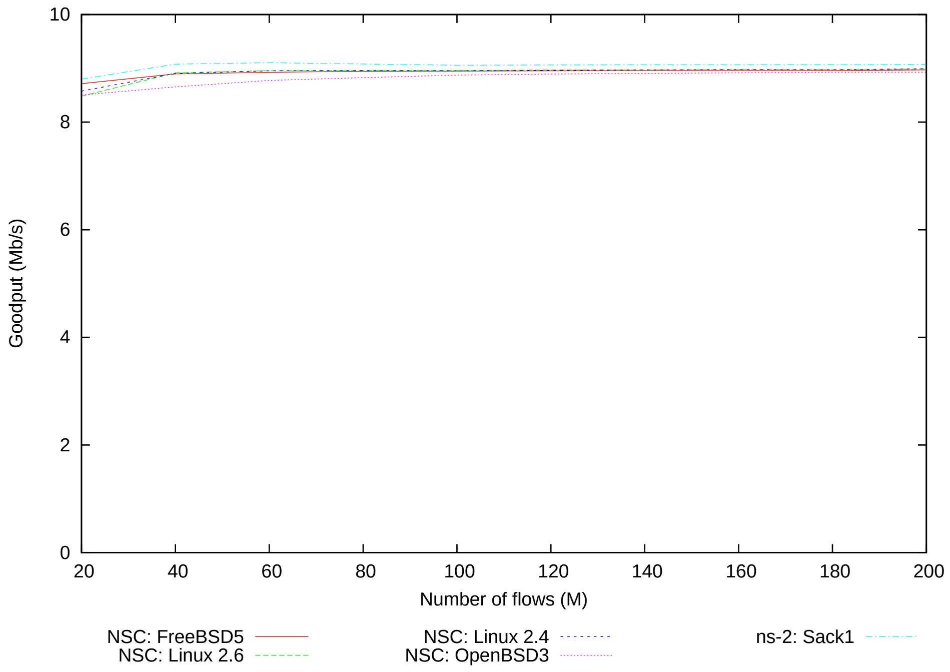


(a) Total goodput over M TCP connections

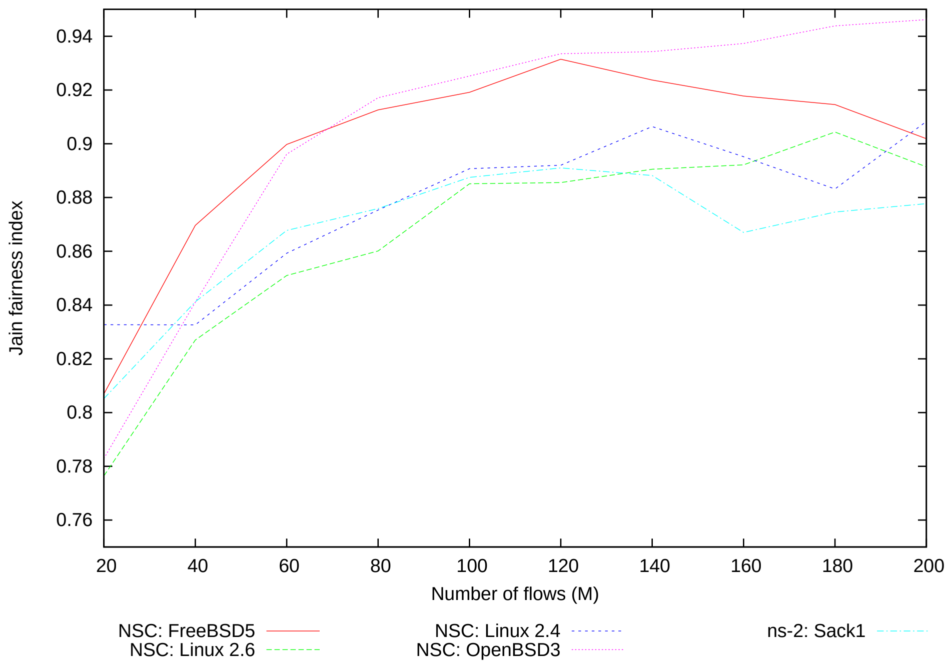


(b) Jain fairness index

Figure 5.7: TCP over 10Mb/s bottleneck with reverse traffic [53]



(a) Total goodput over M TCP connections



(b) Jain fairness index

Figure 5.8: TCP over 10Mb/s bottleneck with reverse traffic

the results presented in [53]. This trend occurs because at lower values of M there is a greater variation of RTTs which increases TCPs unfairness. This variation in RTTs is due to the setup of the experiment, where RTTs are uniformly distributed over an interval depending on the size of M.

The results in figure 5.8(b) also show the difference between TCP implementations. The ns-2 Sack TCP model creates results which have the same trend but using ns-2 abstracted models does not give any knowledge on the range of values the real TCP implementations produce.

Multiple bottleneck scenario

Figure 5.9(a) shows the simulation scenario used by Grieco and Mascolo to evaluate the effect of multiple congestion points on TCP congestion control. The figure shows the setup for 2 "hops" as they are described in [53]. Each hop consists of two routers and two flows transferring data in opposite directions. A single flow, from source $C1$ to sink *Sink 1,* traverses all of the hops. The number of hops is varied and the goodput of the flow $C1$ is measured. The capacity of the entry/exit links is 100Mb/s with 20ms propagation delay. The capacity of the links connecting the routers is 10Mb/s with 10ms propagation delay. Router queue sizes are set to 125 packets. Simulations last 1000 seconds where the cross traffic is active all the time. The measured flow $C1$ begins after 10 seconds of simulated time. A range of TCP variants are used for this flow. The flows generating cross traffic ($C2$ through $C5$ in the figure) are controlled by the ns-2 Newreno TCP model.

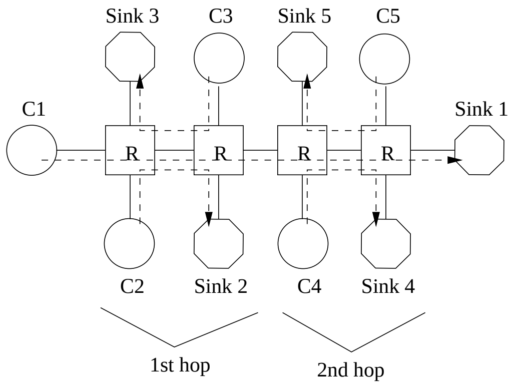
Two graphs are presented by Grieco and Mascolo for this simulation scenario. Copies of the originals are included in figure 5.9. Figure 5.9(b) shows the goodput of the flow $C1$ as the number of hops is varied for the TCP congestion control algorithms New Reno, Vegas and Westwood+. Figure 5.9(c) shows what Grieco and Mascolo refer to as the "total goodput", defined as the *"goodput of the C_1 connection + average of the $C_2, C_4 ... C_{2N}$ connection goodputs"* [53]. These scenarios are reproduced with the NSC TCP implementations in place of the TCP variants used in the original work. Grieco and Mascolo use New Reno TCP as the baseline TCP to compare Westwood+ to; the results of the reproductions shown in figure 5.10 show the variation between TCP implementations which could represent this baseline.

The graphs in figure 5.10 show only the FreeBSD 5 and Linux 2.6 NSC TCP implementations for brevity. Other TCP implementations produced results between these two implementations. The graphs include confidence intervals created based on 20 simulations with differing random seeds for each point on the graph. The confidence intervals are small enough that they are hard to see with the naked eye; the half-lengths range between 1% and 5% for figure 5.10(a) and 0.1% and 0.9% for figure 5.10(b). Only the ns-2 TCP Sack model is shown but results are consistent with those produced with the ns-2 NewReno TCP model.

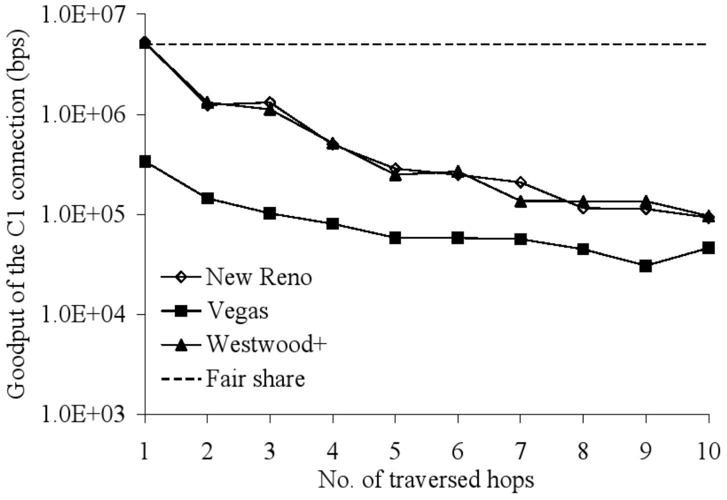
The reproduced results in figure 5.10 should be compared to New Reno in the original results, as the TCP stacks use the New Reno congestion control algorithm. The trends shown in the original research are reproduced here with the ns-2 TCP model. The goodput of the $C1$ connection starts at around the fair share and drops linearly to below 10^6 bps. The total goodput drops off quickly as hops increases then levels out near 7.10^6 bps. These trends are the same in the original research (figure 5.9) and the reproductions shown here (figure 5.10). The results from the NSC implementations show how there are a range of goodputs recorded; when the number of hops equals 10, the Linux 2.6 TCP implementation attains over three times the amount of goodput for the $C1$ connection than the FreeBSD TCP implementation or the ns-2 Sack model. The general trends are the same in the graphs in figure 5.10 but the gradients are different and there is a wide variation between the two TCP implementations shown.

5.3.3 Request latency for a SIP proxy

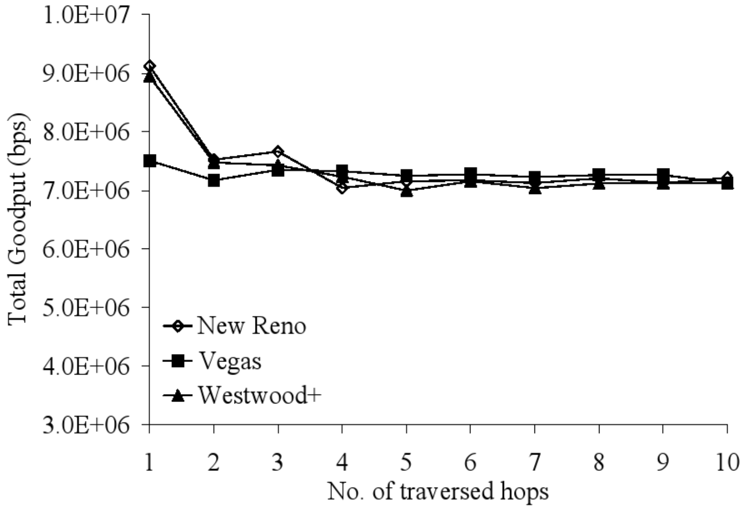
Lulling and Vaughan [180] simulated session initiation protocol (SIP) requests aggregated through a TCP proxy with different TCP variants. They compare Tahoe [181], Reno [182] and Sack [11] variants of TCP with the ns-2 simulator and show SIP request latency under unfavourable networking conditions such as that found on a best-effort network such as the Internet. The effect head of the line (HOL) blocking has on latency of SIP requests aggregated through one TCP



(a) Scenario

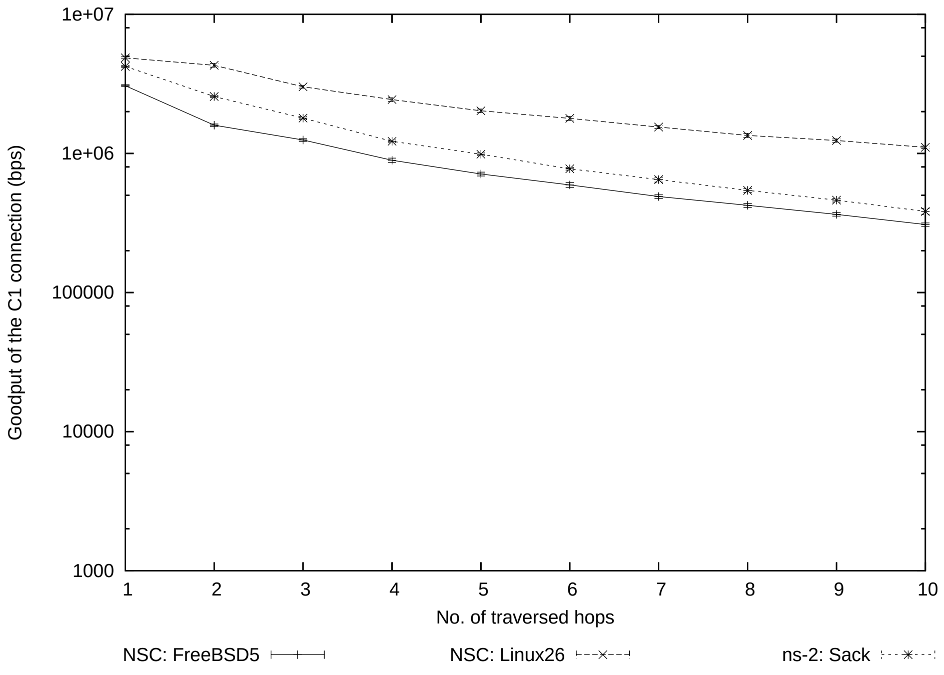


(b) Goodput for the $C1$ connection

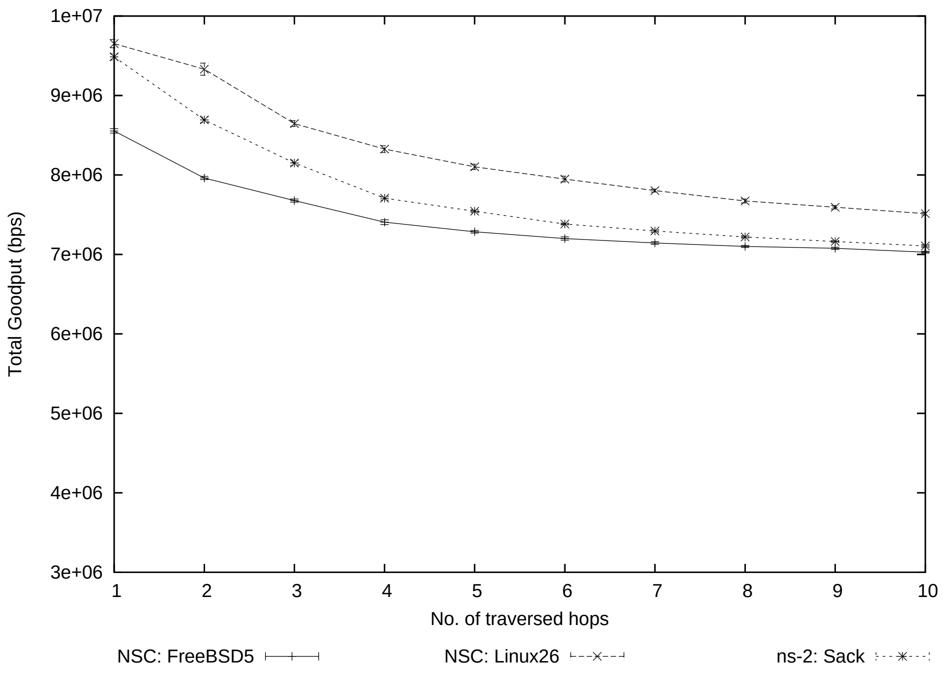


(c) Total goodput

Figure 5.9: TCP goodput vs. number of traversed hops [53]



(a) Goodput for the $C1$ connection



(b) Total goodput

Figure 5.10: TCP goodput vs. number of traversed hops

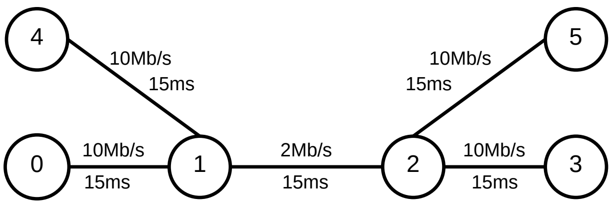


Figure 5.11: Simulation topology used for SIP simulations (adapted from [180])

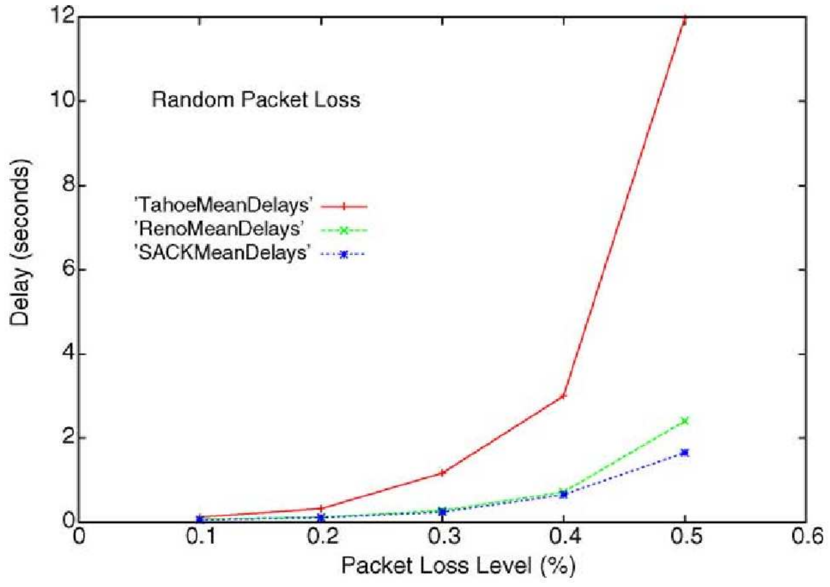
stream is analysed.

Figure 5.11 shows the simulation topology used in the SIP simulations. Nodes 0 and 3 are the SIP proxies using the TCP variants studied. Traffic is generated using a stationary Poisson model to generate the arrival times of 512-byte session establishment requests at node 0. This models a SIP session establishment “INVITE” request arriving from a user to a SIP proxy. The requests are immediately forwarded to the proxy at node 3 and the arrival time recorded. SIP would usually respond with a “100 Trying” response, though this is not modelled here. The TCP MSS is set so a SIP message occupies one TCP segment. TCP delayed acknowledgements are disabled.

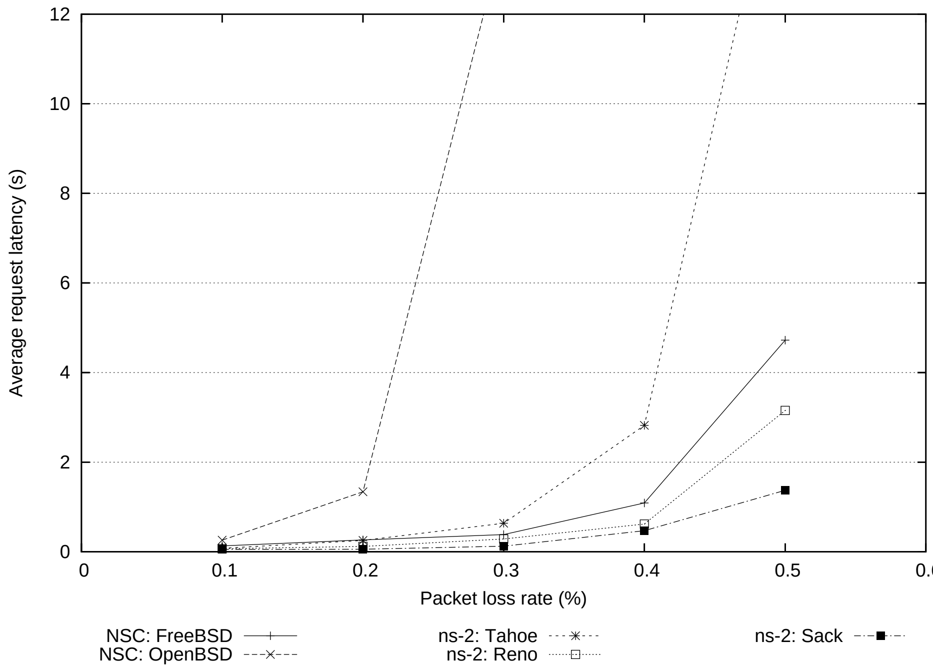
This simulation setup is used to test SIP request latency under varying loss conditions. Figure 5.12 shows the average request latency for increasing packet drop rates. The original results presented by Lulling and Vaughan [180] are shown in figure 5.12(a), these are from figure 9 of [180]. Figure 5.12(b) shows a reproduced graph with extra results from using the NSC FreeBSD and OpenBSD TCP implementations. Linux is not included because delayed acknowledgements cannot be disabled in the Linux TCP stack[1].

Lulling and Vaughan analysed the delays under these loss conditions and check whether the latency is within a 2 second bound. This bound is due to ISDN switches used to interconnect within the public switched telephone network (PSTN) which may abandon a call if a reply from a setup attempt is not received with 2 seconds [183]. They were able to conclude that TCP Sack is the only TCP variant that is able to satisfy this bound under all loss rates tested. As figure 5.12(b) shows, simulating with real world code provides extra insight into this scenario:

[1]Linux has a socket option called `QUICKACK` which disables delayed acks for only a short period, not the entire TCP connection duration.



(a) Original results [180]



(b) Reproduced results

Figure 5.12: Average SIP request latency for increasing loss rates

FreeBSD has an average latency of over 4 seconds at a loss rate of 0.5% where OpenBSD has a very large latency once the loss rate is greater than 0.3%.

It is unclear why such a low segment size was chosen for this simulation scenario as Internet MTUs are generally higher [6, 184, 185]. It is possible this was an attempt to reduce delay and jitter. When a higher MTU such as 1500 is used the request latency is much lower than presented in figure 5.12. Conversely, delayed acknowledgements are widely used in real TCP implementations [139] and increase the SIP request latency under random loss.

5.4 Summary

The simulation results shown here show that large differences can be found between TCP implementations and models in the same scenario. Using real world TCP code in simulation makes it practical to evaluate of how a range of TCP implementations react to a scenario, when such testing may have prohibitive cost without a system like the Network Simulation Cradle. This evaluation provides additional knowledge about a simulation scenario. It validates existing results and/or gives greater insight by showing the range of performance values recorded by real implementations which can be very different to the results from simulated abstractions.

Chapter 6

With more detail comes greater cost

Performance is important for a network simulator and its models: a researcher needs to be able to run a set of simulations in reasonable time to obtain results to analyse. Network simulators are often designed to address the question of performance. For example, the ns-2 implementation is split between C++ and OTcl; OTcl is only used for configuration where as time critical parts of the simulator and models are written in C++ for performance [17].

The Network Simulation Cradle provides TCP models that are intended to increase accuracy by using real world TCP code. This reduces the level of abstraction in the models, something which is generally noted to decrease performance [17, 129, 186, 187]. There are a variety of factors affecting the performance of the NSC. The NSC TCP models perform many operations that TCP models found in a simulator like ns-2 do not perform, such as:

- checksumming of all packets;
- full packet payload used;
- extra integrity checking;
- receive windows; and
- software interrupt clock handling.

However, there are some mitigating features. The TCP implementations are finely tuned for performance. The implementations that are part of the NSC are in some cases many years old; they have matured and been optimised over a long period of time. The same may not be true of a simulation model; in some situations a real implementation may be faster than a simplified model. This means the relative performance of real world TCP implementations and simplified TCP simulation models is not clear.

Overall, more work is performed by a real implementation, and because of this, we expect worse performance. This chapter presents quantitative measures of relative performance between the NSC TCP implementations and simplified TCP models. The NSC is compared to ns-2 TCP models. ns-2 is chosen because of its popularity for TCP simulation—it provides models that are scalable and fast enough for many practitioners of TCP simulation. As the underlying simulator used to run both the NSC models and the ns-2 models is the same, it is reasonable to compare the performance results of the models directly.

6.1 Performance measures covered

A basic measure of performance of a simulator is how many seconds of simulated time can be simulated per second of real time. This depends on the scenario simulated and the computer which runs the simulation. Graphing the relationship between the two measures of time shows whether each TCP model used has a linear increase in CPU time required as the simulation time is increased (this is shown later in figure 6.1). Each scenario we analyse to see if the scaling is linear for ns-2 and NSC TCP models; if this is so then the gradient of the lines on the graph provide a measure of relative performance.

The number of TCP flows in a simulation scenario can be varied to test how performance scales as the number of TCP flows that need to be simulated increases. Both the time taken and the memory required as the scale of the simulation increases are important as they will limit the size of simulations that can practically be run by simulation researchers with the NSC.

Simplified TCP simulation models often do not simulate packet payloads. Only packet header information is required to simulate TCP dynamics, so packet payloads are not used: this reduces memory usage and means that the packet payloads do not need to be copied between the application and TCP models which reduces the amount of work carried out per packet. The real TCP implementations in the NSC use full packet payloads. The cost of doing so is analysed by measuring CPU time and memory usage as the amount of data transferred is increased.

There is potentially a lot of processing per packet in a real world TCP/IP implementation. The implementation needs to ensure that packets are destined for the host the implementation is running on, it needs to check each packet for integrity with checksums and must match each packet to a TCP flow. Stevens [84] discusses the processing performed when a packet reaches the TCP input processing function in a BSD TCP implementation and shows that many operations are performed before the packet is fully processed. A TCP model designed for a simulator is able to ignore many of the requirements of a real implementation due to being run in an isolated, controlled environment. This processing overhead per packet is investigated due to this possible discrepancy in the amount of work required to process each packet.

A fine-grained analysis of performance is possible by profiling the simulator when simulating with either the NSC TCP models or the simplified TCP models. Profiling with tools such as OProfile [188, 189] and Valgrind [190] provide detailed information on where CPU time is spent and memory is used.

A view of performance encountered when running simulation scenarios used in published research is provided by CPU time measurements of the reproduced simulations presented in chapters 4 and 5. The time measurements from the simulation scenarios in previous chapters is reported in this chapter.

The globaliser (discussed in chapter 3, page 53) has a performance impact at compile time. The time to build the TCP implementations with the globaliser is measured to gain a quantitative measure of the time taken. The size of the shared libraries created during the build are checked, as this affects memory usage at runtime when the shared libraries are loaded. The output of the globaliser also impacts runtime performance because it creates indirect references to data. The cost is measured by comparing simulations run using the NSC TCP models with the globaliser enabled and disabled—this is possible if only one TCP endpoint per shared library is required in the simulation.

Table 6.1: Performance testing setup

CPU	AMD Athlon XP 2100+ (1730MHz)
CPU Cache	256KB
RAM	1.0GB
Simulator version	ns 2.29
NSC version	0.2.3
Operating system distribution	Ubuntu Edgy Eft (6.10) Linux
Operating system kernel	Linux 2.6.17-11-386
Compiler	gcc 4.1.2

Results from running experiments to analyse these issues are presented in the next three sections. Section 6.2 shows CPU-time measurements, section 6.3 shows memory usage measurements and section 6.4 shows results of experiments with the globaliser. The overall impact on performance is discussed at the end of this chapter, in section 6.5.

6.2 CPU time

The CPU performance of the Network Simulation Cradle for TCP simulation compared to simplified TCP models is presented in this section. Simulations are run with only the TCP model used changing. Run time is recorded with the Linux `time` command which reports real, user and system time spent running the process. To measure the run time the `time` command is configured to report the total number of CPU-seconds that the process used (wall clock time). The machine used to record the statistics is set to single user mode, where other user applications are not running. Results are reported in graphs with confidence intervals from a minimum of 20 runs. The simulation machine setup is summarised in table 6.1.

A simple dumbbell topology is used in the following experiments to show the basic performance of the Network Simulation Cradle. The bandwidth of the bottleneck link is 2Mb/s, the round trip time of all flows is 40ms and the MTU is 1500.

6.2.1 Time to simulate simple scenarios

Four simulation scenarios are covered in the following sections. In each scenario one simulation parameter is varied. The CPU time to simulate the scenario with each value of the varied parameter is plotted. The amount of simulated time, number of TCP flows simulated, packet size and amount of data transferred are the parameters varied. These scenarios are studied to see where any differences in simulation time are most prevalent. This information can then be used to decide on how to optimise simulation of real world TCP implementations if required.

Checking that the relationship between simulated time and real time is linear ensures that the simulation does not degrade over time. If it is linear, the gradient of the relationship provides baseline performance results of a simple simulation.

The scaling of TCP models is investigated by increasing the number of TCP flows simulated. The amount of work done per packet is analysed by varying the maximum transfer unit. The cost of using full packet payloads is presented by graphing the amount of data transferred versus the time to simulate for each TCP model.

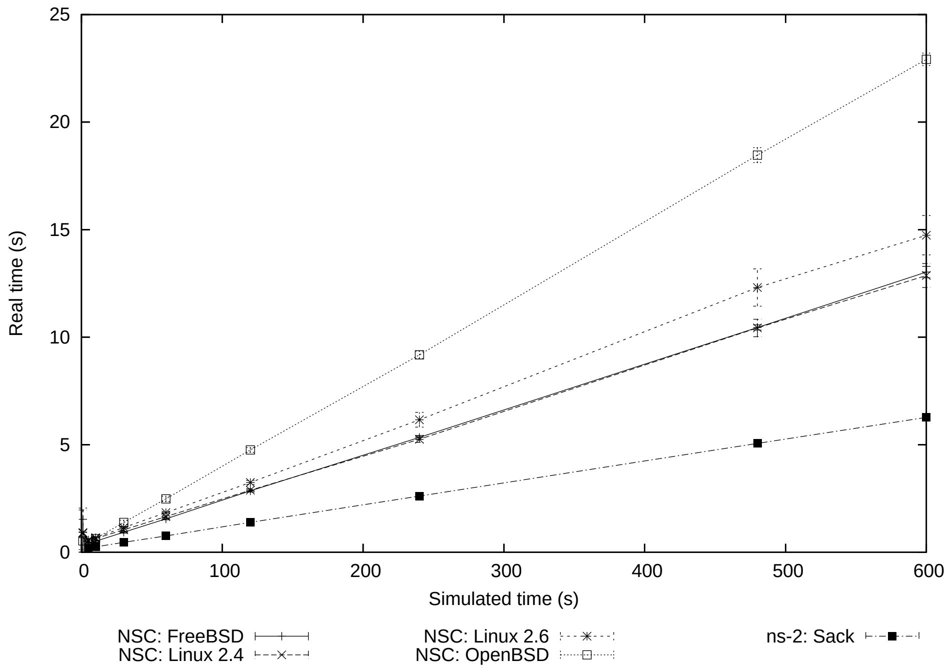
Simulation time vs. real time

Figure 6.1 shows the time to simulate when simulating a single TCP flow with a unidirectional bulk transfer. The length of the simulation is varied and the real (wall-clock) time to simulate recorded. Figure 6.1(a) shows the relationship between simulated time and real time for all TCP models studied. This relationship is linear. Other ns-2 TCP models have very similar results to those shown in the figure and are omitted for brevity. The gradient of the lines on the graph are shown in figure 6.1(b): this shows how many seconds are simulated for every real second.

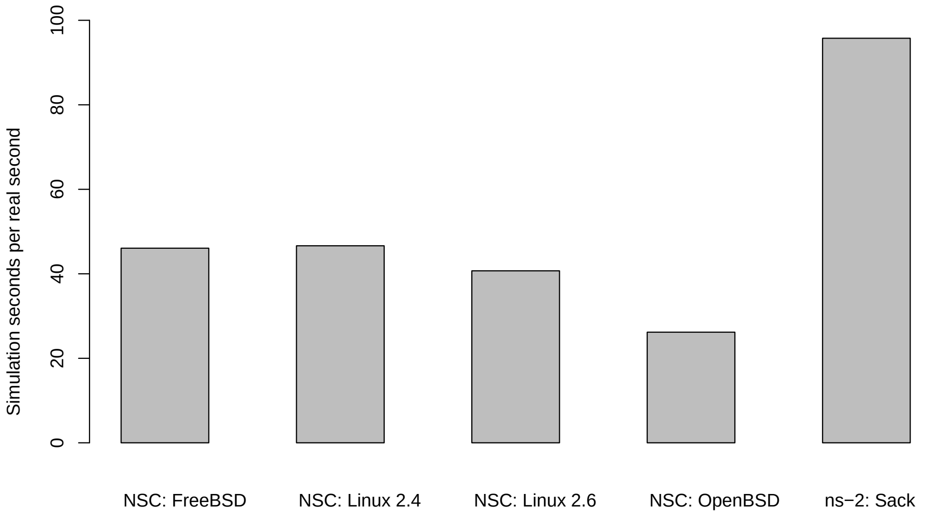
These results show that ns-2's simplified TCP models are almost 5 times faster than NSC with OpenBSD at simulating this scenario and roughly 2.4 times faster than NSC with FreeBSD.

Increasing TCP flows vs. real time

The time taken to simulate many flows over the topology described earlier is presented in figure 6.2. With many flows, the difference between simulating with NSC and ns-2 TCP models is less than with a single flow. With 200 flows, the worst case of NSC using the Linux 2.6 TCP stack takes roughly 2.5 times as long to simulate than ns-2's Sack TCP model.



(a) Simulation results



(b) Simulation speed

Figure 6.1: Simulated time vs. real time for a single TCP flow

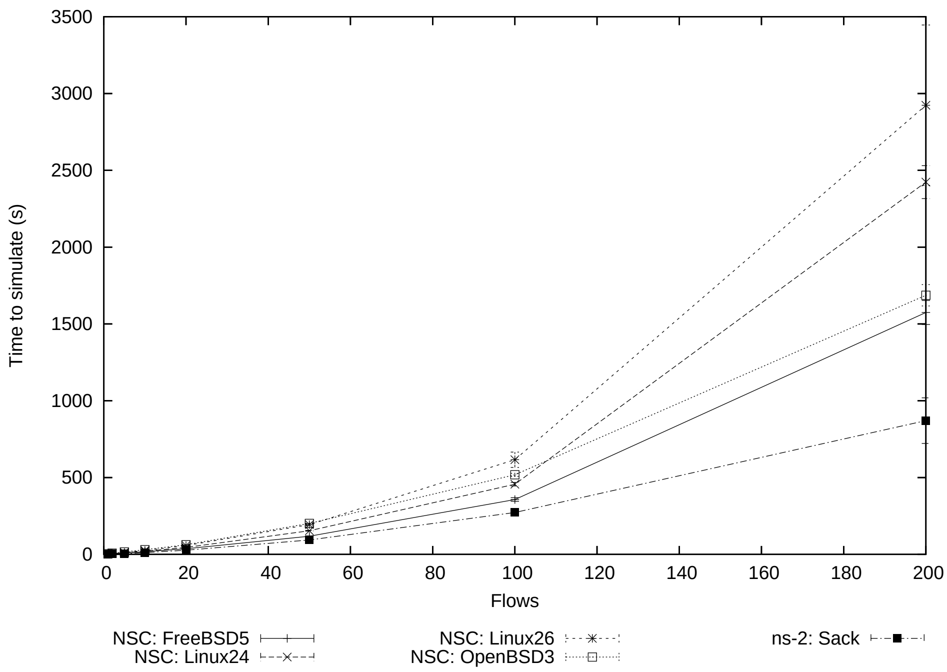


Figure 6.2: CPU time to simulate many flows

There is a smaller number of packets to process per TCP connection in this situation which is a likely reason for the difference in results compared to figure 6.1. The bottleneck link is still the same bandwidth, meaning the number of packets which get through will be similar in both simulations. With 200 flows, there will be a large amount of congestion, meaning there is possibly more interaction with retransmission timers, but as there is an overall decrease in the number of packets which must be processed per TCP stream, there is less work to do per TCP model.

The relationship between number of flows and time to simulate shown in figure 6.2 is non-linear: the time to simulate as the number of flows varies increases approximately exponentially for all models including the simplified ns-2 models and the NSC TCP implementations. The exponent is larger for the NSC TCP implementations.

Per-packet cost

Real TCP implementations must perform additional processing per packet sent or received than the simplified TCP models present in ns-2. For example, real TCP stack implementations must check incoming packets for integrity as, in the general cases, packets may be erroneous. Models built solely for simulation can make many assumptions about the data received, as the characteristics of the transmission are controlled entirely by the creator of the simulation scenario. Comparing the time to process many packets is of interest because the results show whether the difference in TCP models means that there are performance differences and how large the differences are.

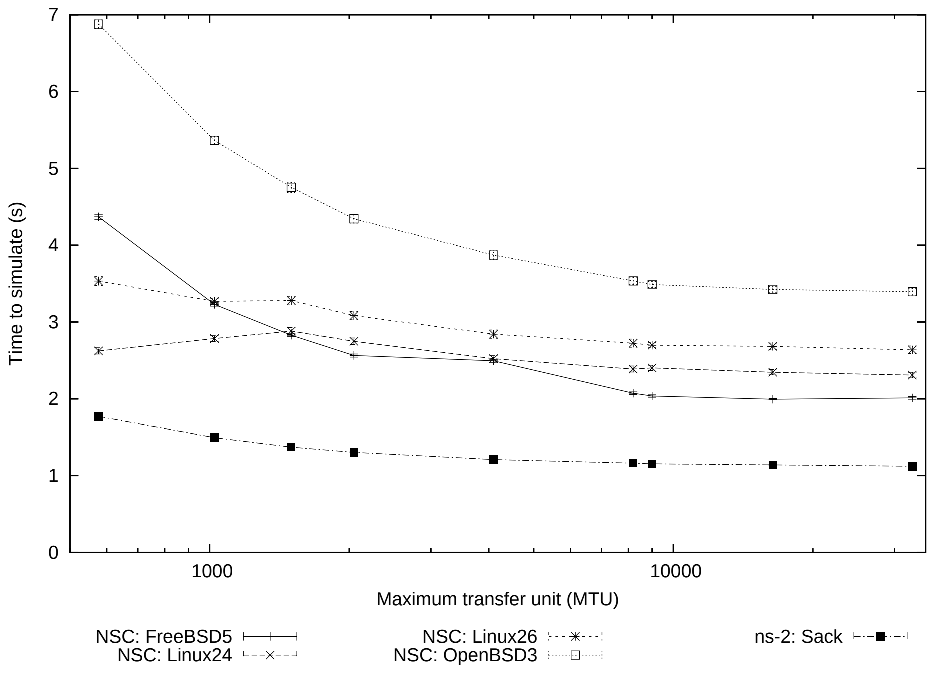
Varying the maximum transfer unit (MTU) in the same simulation scenario as shown previously, directly affects the number of packets processed. A single TCP flow transfers data as fast as possible for 120 seconds of simulated time. The MTU is varied in the following experiment. The way the time to simulate varies as packet size is changed is presented in figure 6.3(a).

The trends for NSC's FreeBSD and OpenBSD implementations and ns-2's Sack model are similar. The two Linux TCP implementations differ when the MTU is less than or equal to 1000 bytes. The goodput results are shown in figure 6.3(b). The Linux TCP implementations do not handle small MTUs well and achieve much less goodput than the other TCP implementations studied. With less goodput, there are less packet processing events to simulate and therefore the simulation takes less time.

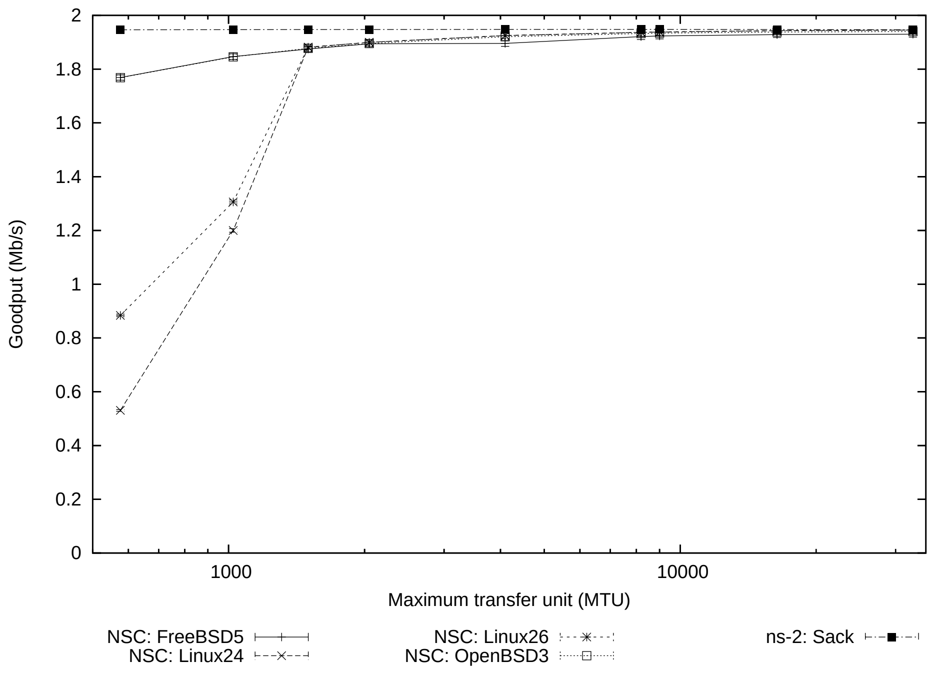
Simulating large data transfers

ns-2's TCP models do not include a full packet payloads, instead each simulated packet has a `size` field which is used to describe the packet size. There is no payload that must be copied when the packet enters and leaves a TCP model. The real world implementations in the NSC copy the packet data when data it sent or received by the simulated application and when packets are sent or received from the network stack. To analyse the effect of this difference, a simulation that increases the bandwidth of the bottleneck link while keeping bandwidth-delay product and simulation time constant is used.

The bandwidth-delay product (BDP) is fixed at 50kB. The bandwidth of the bottleneck link ranges from 56kb to 100Mb. The delay is calculated as follows to ensure the BDP is constant: $delay = bdp/(2.0 * bandwidth)$. The results of this



(a) Time to simulate



(b) Goodput

Figure 6.3: Time to simulate and goodput with a varying MTU

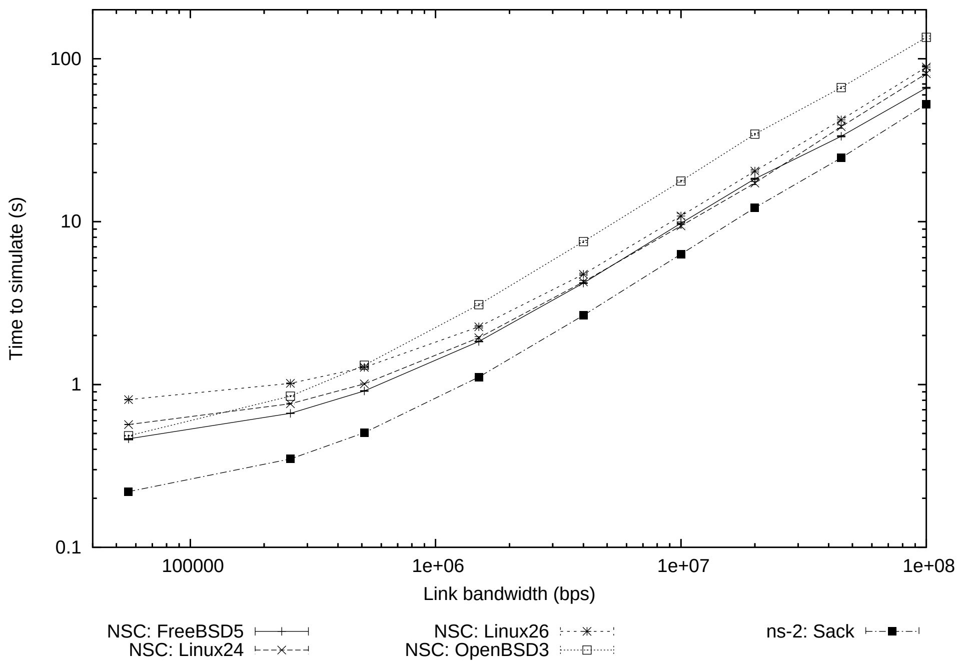


Figure 6.4: CPU time to simulate increasing amounts of data

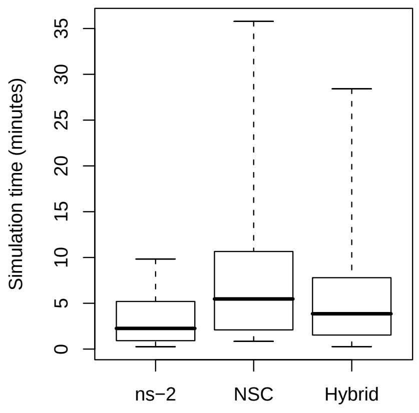
simulation are shown in figure 6.4.

The results show similar linear growth rates of time to simulate versus link bandwidth for all TCP models measured. At bandwidths of less than 1Mb the gradient is smaller due to the small amount of data transferred: the TCP models are not able to make full use of the bandwidth available in the simulation time due to the high latency. ns-2's TCP model is the fastest but slows down at the same rate the real TCP implementations do. This means that the performance of the simulation is not bounded by the extra copying of data due to real packet payloads in the NSC.

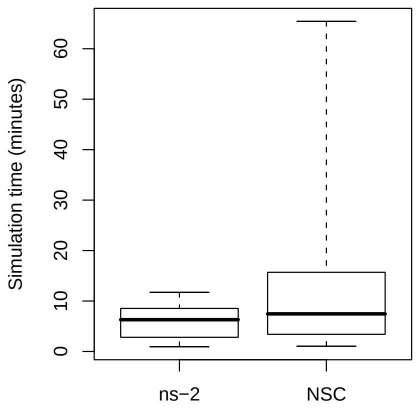
6.2.2 Time required to simulate more complex scenarios

The previous performance results show the time to simulate for constrained situations. Such simulations do not give a good understanding of this performance in the type of scenarios carried out by many researchers, as the pattern of usage is likely to be different. The results in this section are from simulation scenarios used in published research.

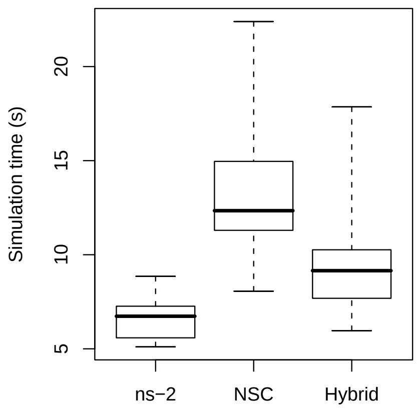
Results from simulations run for research replication studies presented in chapters 4 and 5 and in [163, 191] are presented in figure 6.5. In these studies a



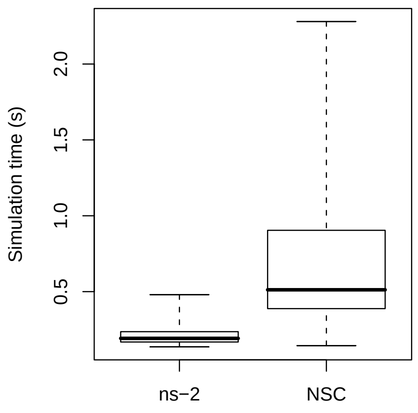
(a) A multiple bottleneck topology with up to 160 flows including cross traffic



(b) A dumbbell topology with up to 210 flows including reverse traffic



(c) Two TCP flows on a high bandwidth-delay product path



(d) One TCP flow over a network with random packet loss

Figure 6.5: Simulation times for complex scenarios

dual AMD Opteron 250 and a dual Dual-Core AMD Opteron 265 are used with the simulations randomly distributed across them. Some of the simulation scenarios use either ns-2 simplified TCP models or NSC TCP implementations, while others use both in a hybrid combination (for example, there is a scenario where 200 flows of two different TCP types are simulated using an NSC TCP implementation for 100 flows and an ns-2 TCP model for 100 flows).

Figure 6.5(a) shows the time to simulate the scenario described in section 4.3.1 (see page 92 for a complete description of this scenario). This simulation uses a complex topology consisting of multiple bottleneck links and up to 160 TCP flows. There are two different TCP variants used in this simulation, one is used to generate background traffic and used for 10 flows while the other is used for the rest of the flows in the simulation. Simulations last 300 seconds of simulated time. 2040 simulations are run in this scenario and the CPU time measurements of these are used to generate the data in this figure.

The graphs show the minimum, lower quartile, median, upper quartile and maximum values of time to simulate for each simulation scenario. Each data point included in the distribution is recorded from a single simulation run. A range of values is expected due to different sets of simulation parameters requiring a different amount of time to run. The ranges can be compared between ns-2, NSC and hybrid situations due to each group running simulations with the same set of parameters. These graphs give a simple view of the distribution of results which is useful: it is apparent that many of the results using NSC TCP implementations has a very large range but much smaller inter-quartile range, indicating that the simulations that take extreme lengths of time are exceptions rather than the norm.

The simulation times shown in figure 6.5(b) are recorded from simulating up to 210 TCP flows over a dumbbell topology. 10 flows are used to create background traffic and up to 200 flows are measured over a bottleneck link of 10Mb/s. This scenario is described in detail in section 5.3.2 (see page 112). CPU time measurements from 70 simulations are used to generate the data in this graph. The maximum time to simulate in this situation is almost six times greater when the NSC TCP implementations are used. However, the median and minimum time to simulate are very close, within 6% of each other. The upper quartile when using the NSC is higher than the maximum when using ns-2 TCP models, indicating that a large percentage of simulations are taking much longer with the NSC.

Figure 6.5(c) shows the time to simulate two TCP flows on a high bandwidth-delay product network: round trip time varies from 16ms to 162ms and bandwidth is 10Mb/s or 250Mb/s. This simulation scenario is covered in section 5.3.1 (see page 108). CPU time measurements from 2700 simulations are used to generate the data in this figure. The time to simulate when using the NSC is approximately twice the time to simulate with ns-2 TCP models. Of the graphs in figure 6.5, this is the only one that shows the minimum time to simulate when using the NSC is much higher than when using ns-2 TCP models. This could be due to the higher bandwidths used in this simulation; section 6.2.1 showed how there is a large offset in time to simulate at high bandwidths, though the percentage difference in time to simulate does not grow. This explains the results shown in figure 6.5(c): the minimum is higher, but the minimum, maximum and median are all around two to three times higher when using the NSC.

The results shown in figure 6.5(d) are CPU time measurements from the scenario presented in section 4.3.2 (see page 94): a single TCP flow undergoing random packet loss. The results shown are generated from analysing the runtime of 35000 simulations. There is much greater variability of runtime when using the NSC TCP implementations. There is approximately a difference of five times between using the simplified ns-2 TCP models and the NSC TCP implementations.

The simulation time for the set of four different simulation scenarios in figure 6.5 have some characteristics in common. The maximum time to simulate when using NSC TCP implementations is higher than the maximum when using ns-2 TCP models, while the minimum is often similar; the range is greater when simulating with NSC. Figure 6.5(b) shows a situation where the median time to simulate is very similar (though the maximum and inter-quartile range is larger). The difference in time to simulate in these complex scenarios is similar to the differences found in the simpler scenarios. The difference ranges from being very similar to differences of around five times.

6.2.3 Profile

The results shown in sections 6.2.1 and 6.2.2 show the NSC TCP implementations to be slower by up to 400% in almost all cases but give little insight into what causes the performance difference. Profiling the simulator shows where the most time was spent in the simulator for a specific simulation scenario. Profiling is used here to gain quantitative information on the difference in performance for a single simulation scenario.

The system-wide profiler OProfile [188, 189] provides profiling information with a low overhead by making use of a system's hardware performance counters. OProfile uses profiling support in the system's CPU if possible and falls back to using statistical profiling (where a timer interrupt is created by OProfile to periodically poll applications and record their state). In the results presented the performance counters present in the AMD Athlon XP CPU are used by OProfile.

The following results are gathered by OProfile when simulating the scenario described in section 5.3.1 (the performance results for this scenario are shown in figure 6.5(c)). Two TCP streams compete over a bottleneck link of 250Mb/s with an RTT of 82ms. The TCP model or implementation is the same for each TCP endpoint. This scenario was chosen due to the large difference in performance between simulations using ns-2 TCP models and simulating using the NSC TCP implementations.

Results from profiling this scenario are shown in tables 6.2, 6.3 and 6.4. The ten functions that the most amount of CPU time was spent in and the module they are located in are shown in each table. OProfile shows a detailed analysis with many more functions included but only the top ten are shown as the goal of this experiment is to compare where the majority of CPU time is spent. Results are shown for the NSC TCP implementations Linux 2.6 (table 6.2) and FreeBSD (table 6.3) and ns-2's Newreno TCP model (table 6.4). The results for the Newreno model are consistent with results measured from other ns-2 TCP models.

The *Samples* column in tables 6.2, 6.3 and 6.4 is the number of times a sample from OProfile found the CPU to be executing the function listed in the *Symbol* column inside the module listed in the *Image* column. The *Percent* column refers to the percentage of the total execution time of the process that was spent inside the function.

When simulating the Linux 2.6 stack, the most time is spent in the function `CalendarScheduler::insert`. This function is used to add a new simulation event to the global event list. The NSC agent for Linux 2.6 schedules a timer event every 1ms, where the other TCP implementations use a timer that fires every 10ms. Whenever the timer is rescheduled a new simulation event is added to the global event list. This explains why the insert function is higher in table 6.2 than in the other tables. Three functions from the Linux 2.6 shared library (`liblinux26.so`) also appear on the profile, although none of these functions

Table 6.2: ns-2 profile using NSC: Linux 2.6 TCP

Samples	Percent	Image	Symbol
17521	7.7733	ns	`CalendarScheduler::insert`
17339	7.6926	ns	`Scheduler::dispatch`
16214	7.1935	libm-2.3.5.so	(no symbols)
12256	5.4375	liblinux26.so	`sk_stream_wait_memory`
10561	4.6855	liblinux26.so	`get_stack_id`
9823	4.3580	ns	`Scheduler::schedule`
9029	4.0058	liblinux26.so	`tcp_sendmsg`
8899	3.9481	kernel	(no symbols)
8333	3.6970	ns	`CBR_Traffic::next_interval`

Table 6.3: ns-2 profile using NSC: FreeBSD TCP

Samples	Percent	Image	Symbol
15528	8.2930	libm-2.3.5.so	(no symbols)
14006	7.4801	ns	`Scheduler::schedule`
13823	7.3824	ns	`CalendarScheduler::insert`
12744	6.8061	ns	`Scheduler::dispatch`
10933	5.8389	kernel	(no symbols)
10137	5.4138	libc-2.3.5.so	(no symbols)
9315	4.9748	libfreebsd5.so	`sosend`
7928	4.2341	ns	`CBR_Traffic::next_interval`
6903	3.6867	ns	`NSCSimpleAgent::sendmsg`
6520	3.4821	ns	`TcpAgent::sendmsg`

uses more percentage time than the scheduler insert function. `libm-2.3.5.so` is the C mathematics library and is likely used when working with the floating point numbers used to express time in ns-2.

When the FreeBSD stack is used as the TCP implementation in the simulation scenario the profile data is similar to the profile for Linux 2.6, as seen in table 6.3. Inserting and managing the global event list via the `Scheduler` classes is again near the top of the table. The function `sosend` (a generic function used to send data through a socket) is the only function from the FreeBSD implementation that appears on the table.

Table 6.4 shows the profile data for the simulation scenario when using the ns-2 Newreno TCP model. Interaction with the scheduler and the mathematics library are again prevalent.

The results from using other TCP implementations with NSC for the test introduced at the start of this section (§6.2.3) are similar to those shown. No one

Table 6.4: ns-2 profile using original TCP agents

Samples	Percent	Image	Symbol
16119	13.0132	libm-2.3.5.so	(no symbols)
15042	12.1437	ns	`CalendarScheduler::insert`
12800	10.3337	ns	`TcpAgent::sendmsg`
9969	8.0481	ns	`Scheduler::dispatch`
9301	7.5089	ns	`Scheduler::schedule`
7434	6.0016	ns	`CBR_Traffic::next_interval`
6320	5.1022	ns	`RenoTcpAgent::window`
5461	4.4088	ns	`CalendarScheduler::head`
5360	4.3272	libc-2.3.5.so	(no symbols)
4098	3.3084	ns	`TrafficGenerator::timeout`

operation performed by the real world TCP implementations takes up any greater percentage of CPU time than existing interactions in the simulator.

Callgrind and KCacheGrind are part of the Valgrind [190] framework that records detailed performance results. KCachegrind is a visualisation tool for the data recorded by Callgrind. Profiling with these tools was also used and they show that the time spent in the mathematics library is due to the use of floating point functions in `CalendarScheduler::insert`. This insert function is called mostly from two timers: one is used to generate application traffic (`TrafficGenerator` and `CBR_Traffic` in the earlier tables) and the other is used by the NSC ns-2 simulation agent to send timer messages to the cradled TCP implementation. The time spent in `libc-2.3.5.so` is due to calls to `malloc` (memory allocate), `bzero` (set an area of memory to 0, used when the packet structure is allocated in ns-2) and `free` (free memory).

6.2.4 Discussion of CPU performance

The difference in performance between simulating TCP with a simplified ns-2 TCP model and simulating TCP with a full TCP implementation accessed via the Network Simulation Cradle varies between simulation scenarios. In almost all cases studied using a simplified model will take less time to simulate.

Trends

The way CPU time varies against the parameters tested against are similar for the real world TCP implementations and the simplified models. Figure 6.1, which shows the relationship between simulated time and real time for a simple scenario, shows linear scaling for all TCP models studied, with a simplified ns-2 TCP model being between 2.4 and 5 times faster than the NSC TCP implementations. The same scaling is also evident for all TCP models in figure 6.4 too, all TCP models slow down at the same rate as the amount of data to transfer is increased.

The per-packet cost simulations, shown in figure 6.3, show similar trends between the real world TCP implementations and the simplified TCP model. The trend is not linear and the ns-2 TCP model is consistently between two and three times faster, but the same pattern is followed by all TCP models when the MTU is greater than or equal to 1500.

Non-linearity is also presented in figure 6.2. This shows the time required to simulate as the number of TCP flows increases. The growth rate is roughly exponential and the rate of change of the gradient shown on the graph appears to follow the same pattern for all stacks measured.

In summary, the same trends are followed by real world TCP implementations and simplified TCP models in the tests covered.

Additional cost

It is slower to simulate using the real world TCP implementations in the NSC than using the ns-2 simplified TCP models in the scenarios studied. The percentage difference in time to simulate varies; in the reproduced simulations (covered in section 6.2.2) the difference in median time to simulate varies between 2% and 250%. The distribution of times varies more than this as the graphs in figure 6.5 show: the difference in maximum time to simulate in the graphs varies from between 170% to 450%.

The difference in time to simulate between real world TCP implementations and simplified TCP models in the simulations (covered in section 6.2.1) are within the same range. Figure 6.1(b) shows a direct comparison where the simplified ns-2 TCP model is between 205% and 365% percent faster than the TCP implementations.

Profiling data gathered using OProfile [188] shows an increase in the cost of the simulator scheduling algorithm. Virtual time is implemented in the NSC by calling the software interrupt function in the network stack. This function would normally be called from the hardware interrupt clock hz times a second, where hz varies but is often 100 or 1000. The NSC simulates this clock by enqueing a timer event hz times every virtual second. This happens for each instance of a stack (as their timers will not normally be synchronised).

The scheduling algorithm used by ns-2 is a Calendar Queue [192] by default. The Calendar Queue has been shown by several researchers to have performance problems in some situations. Tan and Thng [193] proposed the "SNOOPy Calendar Queue," while Ahn and Oh [194] presented the "Dynamic Calendar Queue." Yan and Eidenbenz [195] showed another calendar queue, the "Sluggish Calendar Queue." Each of these modifications to the original algorithm are designed to increase the chance of partitioning the calendar queue in an optimal way. Using a more efficient scheduling algorithm could increase the performance of scheduling with the NSC.

The implications of the impact in CPU performance due to using the NSC TCP implementations are discussed further at the end of the chapter in section 6.5.

6.3 Memory usage and scalability

The extra memory used by TCP implementations limits scalability. The simplified TCP models in ns-2 do not send actual data and therefore do not need to buffer any data. Each real world TCP connection endpoint has a socket buffer and network stacks store some amount of global state, all of which is not required for the ns-2 TCP models. Memory use is important because it limits the complexity of a simulation run on a computer: a simulation that requires too much memory cannot be run if the required memory resources are not available (while a slow simulation may be left running for a longer time).

Measuring the memory used by an application is not straightforward. Memory is shared between applications, paged in and out (between hard disk and RAM) and separated into several sections, making measurements of the amount of memory used by a process difficult. There are several metrics the Linux operating system can report that provide a measure of the memory used by a process, but none of them show the amount of stack and heap memory allocated by the program. The total memory an application can address is known as the applications Virtual Memory (VM) size. The amount of this that is resident in main memory is called the Resident Set Size (RSS). An executable and its shared libraries are mapped into the VM of the process. The code of the shared libraries might be shared with other applications. The code and data of executable and shared libraries are usually static; they do not grow over the lifetime of an application (except for the loading of more shared libraries if they are requested). Dynamic memory is allocated in two ways: the heap and the stack. The stack is fast and often limited to a small size[1] while the heap is used for general memory allocations and grows in size over the life of an application. The amount of memory allocated on the heap is a useful metric to show the amount of memory used by an application during runtime, but the metrics reported by the operating system do not provide this.

A tool named HeapProf [196] is used to measure heap usage. HeapProf is a shared library that is used as an `LD_PRELOAD`—the functions it exports override the standard C library functions of the same name. The heap allocation and management functions `malloc`, `calloc`, `realloc` and `free` are overridden by HeapProf. HeapProf keeps counters of the memory allocated and calls the original implementations of the heap functions in the standard C library functions after updating its statistics. This allows accurate tracking of heap usage over time with a very small performance overhead. HeapProf is used for course-grained heap usage information, for example the total amount of heap memory allocated at a point of time.

6.3.1 Heap use with many TCP flows

The results of the simulation study presented in section 6.2.1 show the time to simulate a simple scenario with a varying number of TCP flows. The peak heap usage over the execution of the ns-2 process as recorded by HeapProf for this scenario is shown in figure 6.6. The increase in peak heap usage as the number of

[1]For example, the maximum stack size defaults to a maximum of 8MB on Ubuntu 5.10 Linux

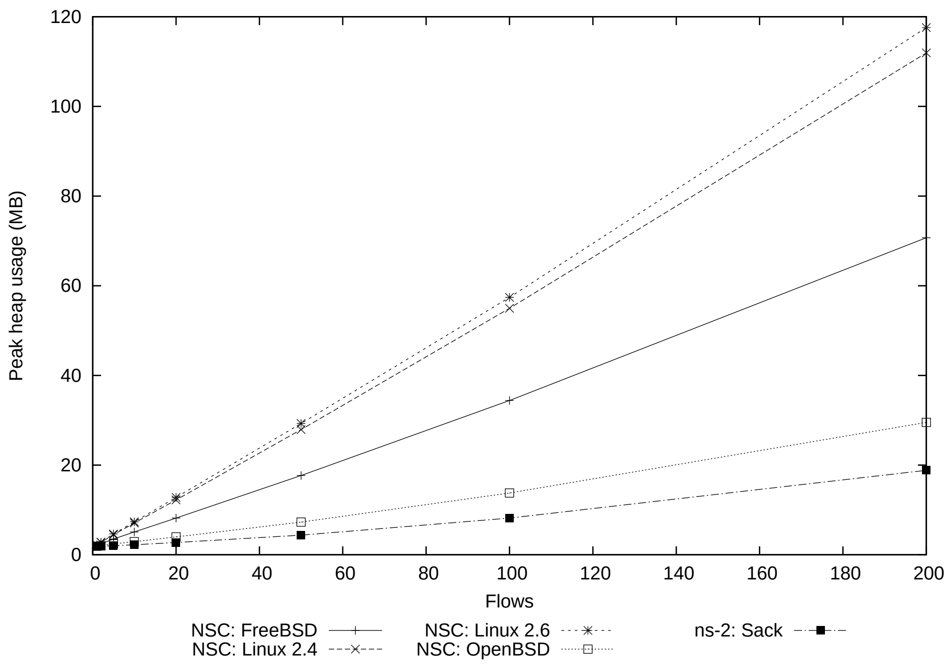


Figure 6.6: Peak heap usage for increasing number of flows

flows is increased is linear for all TCP models studied. The amount of memory used per-flow is summarised in table 6.5.

Table 6.5: Heap usage per TCP flow

TCP model	Memory used per TCP flow (kB)
NSC: FreeBSD	354
NSC: Linux 2.6	588
NSC: Linux 2.4	561
NSC: OpenBSD	147
ns-2: Sack1	94

Of the real world TCP implementations studied, OpenBSD scales the best with close to 7 flows supported per MB of RAM used. In comparison ns-2 TCP agents support close to 11 flows per MB of RAM. The other real world implementations use more RAM and the pattern shown follows the socket buffer sizes used in each of the stacks. OpenBSD defaults to very small socket buffer sizes (and therefore TCP send and receive windows), while versions of Linux have larger maximum defaults and FreeBSD falls between the two.

6.3.2 The effect of increasing the TCP window size

TCP implementations allocate memory to buffer sent and received data. The receive buffer size is explicitly advertised in TCP packets (known as the advertised

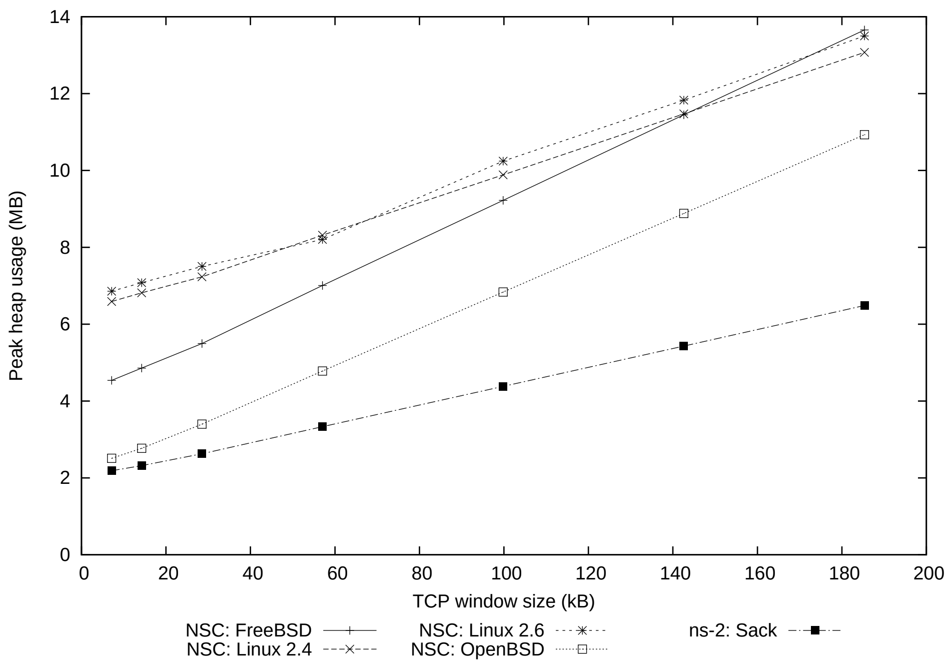


Figure 6.7: Peak heap usage for increasing window size

Table 6.6: Heap usage / window size for ns-2 and NSC

TCP model	Heap usage / window size
NSC: FreeBSD	52.40
NSC: Linux 2.4	37.23
NSC: Linux 2.6	38.15
NSC: OpenBSD	48.38
ns-2: Sack1	24.68

or receiver window) where the send buffer corresponds to the sender window. The simplified TCP models in ns-2 do not need to allocate these buffers because they do not use real data, only a size—simulated packets contain a field indicating their size, rather than a full packet payload.

The following simulation scenario was designed to explore the memory usage of varying TCP window sizes. The size of the TCP windows used by the TCP models/implementations is varied. The scenario used has an RTT of 142ms (corresponding approximately to an RTT from New Zealand to the west coast of the USA), a bandwidth of 10Mb/s and 10 TCP flows sending data at full rate in the same direction for the duration of the experiment. The router queue size on the dumbbell network is set to 2600 packets. These parameters were chosen because of the large bandwidth delay product. Figure 6.7 shows the peak heap usage in this scenario.

Table 6.7: Memory footprint for ns-2 and NSC

TCP model	Size (MB)				
	VM	RSS	Text	Data	BSS
ns-2: Sack1	8	5			
NSC: FreeBSD	144	46	32	19	87
NSC: OpenBSD	87	21	11	6	62
NSC: Linux 2.4	68	47	23	20	16
NSC: Linux 2.6	62	46	22	19	12

Heap usage increases faster with Network Simulation Cradle TCP implementations than with simplified ns-2 TCP models. The real TCP implementations used in NSC have to allocate socket buffers and use full packet payload. The difference is between roughly 150% and 215%. Table 6.6 shows the rate of change for the various TCP models and implementations shown in figure 6.7.

6.3.3 Total memory use

The previous sections discussed peak heap usage. Code and global data also take up memory. The globaliser clones global variables in source code, which increases the amount of memory required for the resulting shared libraries. The memory footprint for ns-2 and NSC built with `NUM_STACKS` set to 500 is summarised in table 6.7.

The reported virtual memory size in table 6.7 is the total VM size of an ns-2 process after simulating a single TCP connection and short data transfer with the indicated TCP model or implementation (the scenario discussed in section 6.3.1 with the number of flows set to one). The maximum Resident Set Size (RSS) is listed next, this is the amount of virtual memory which resides in main memory.

RSS is only a rough measurement of the memory that is actually in use by a process—the operating system can page out memory for a number of reasons. The numbers reported in table 6.7 were measured on a machine with 2.5GB of main RAM memory and show no variation across 5 tests. There was no other user activity on this machine during this time.

The other columns show the different sections of memory inside each shared library. The text section is where the executable code is stored, this can be shared between processes. The data section contains initialisation data for global variables. The BSS (Blocks Started by Symbol) section contains initialisation data for global variables which contain zero bytes initially.

The FreeBSD and OpenBSD shared libraries contain large BSS sections and this is reflected in their larger virtual memory sizes. This memory is not necessarily used during simulation. Simulations that do not use the full number of stacks supported by modifications made by the globaliser will not use all the global variables, hence not all of the BSS and Data sections will be used. The resident set size is much smaller than the virtual memory size in all cases.

The size of data and BSS in the shared libraries is dependent on the setting of `NUM_STACKS` when using the globaliser at build time. The next section analyses the affect of `NUM_STACKS` on the globaliser, including the resulting shared library size, as well as the CPU performance overhead of running code modified by the globaliser.

The total memory use when using the NSC TCP implementations is much higher than when using the ns-2 TCP models. The virtual memory size of an ns-2 process using the NSC TCP implementations is up to 18 times greater. This is largely due to needing to allocate memory for every TCP stack which is supported, even if that TCP stack instance is never used.

6.4 The cost of the globaliser

The globaliser (discussed in section 3.2) modifies the source code of the network stacks used in NSC before they are compiled. Global variables are modified to be arrays and are accessed with an array reference. This process has performance and scalability implications at runtime: there is extra CPU cost involved and more memory is used.

6.4.1 CPU (online) cost

When the globaliser has modified the code there is a performance degradation introduced when accessing global variables: whenever a global variable is

Table 6.8: Globaliser runtime CPU overhead

Scenario	Without globaliser		With globaliser		Difference
	μ	$\pm$	μ	$\pm$	%
FreeBSD	10.38	0.048	10.79	0.091	03.94
Linux 2.6	10.45	0.048	12.13	0.165	16.07
OpenBSD	16.39	0.057	17.02	0.110	03.84

accessed, a function is called and there is a mapping through an indirection table or, in the case of an array, multiple indirection tables. This slows down the access of a global variable that is modified by the globaliser, but it is not obvious how significant this effect is on the overall performance of the code.

To test this the NSC can be compiled with and without using the globaliser during the build. The shared library produced when not using the globaliser will only support a single TCP instance. The shared libraries can be compared by simulating a simple scenario which only requires a single TCP instance and comparing the time to simulate.

The test setup is one TCP flow with one endpoint an instance of FreeBSD, Linux 2.6, or OpenBSD and the other Linux 2.4. Only one instance of FreeBSD, OpenBSD, or Linux 2.6 is required. This allows compiling the shared library for each stack with and without the globaliser. The scenario used is the same as used earlier in this chapter (section 6.2.1)—a dumbbell topology with a bandwidth of 2Mb/s, a round trip time of 40ms and an MTU of 1500 bytes. A single flow transfers data as fast as it can for the duration of the simulation which is set to 500 seconds. Each test is run 20 times.

The performance will differ depending on how often global variables are accessed, so the characteristics will be different for each scenario it is used in. See table 6.8 for the CPU overhead introduced by the globaliser when measured from a simple TCP simulation. In this table the mean CPU usage (μ on the table) and 95% confidence interval ($\pm$ on the table) is shown for each of the TCP implementations used in this experiment.

The difference in time to simulate using shared libraries built with and without the globaliser varies between the TCP implementations. This difference is due to the

Table 6.9: Codebases used for globaliser testing

Project	Files	Lines of code	Global variables
FreeBSD	175	944621	2290
Linux 2.6	122	2687493	726
Linux 2.4	113	2398834	631
OpenBSD	155	683792	730
lwIP	13	12986	33

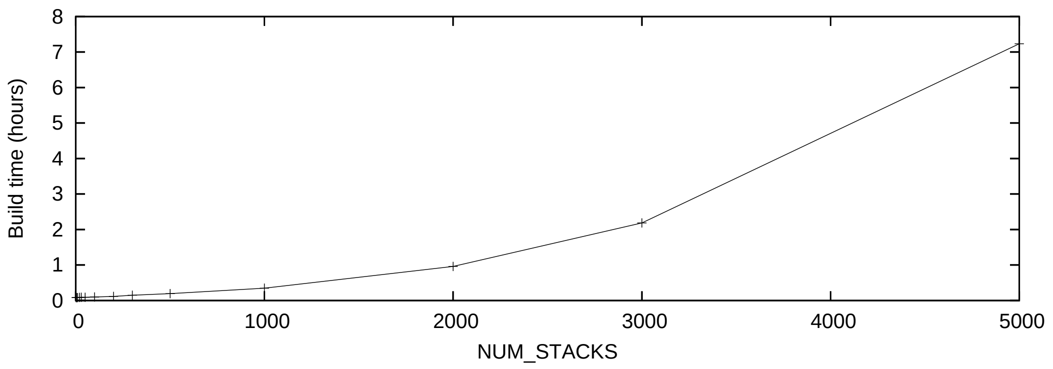
access pattern of global variables in each implementation. Table 6.8 shows that the greatest slowdown observed is when using Linux 2.6, running the simulation with code which has been modified by the globaliser takes a little over 16% longer than using unmodified code. The difference is below 4% for the FreeBSD and OpenBSD TCP implementations. This is comparable to other virtualisation approaches such as Xen [197], in which performance is within 10% of an unvirtualised system in most cases [198].

6.4.2 Offline cost

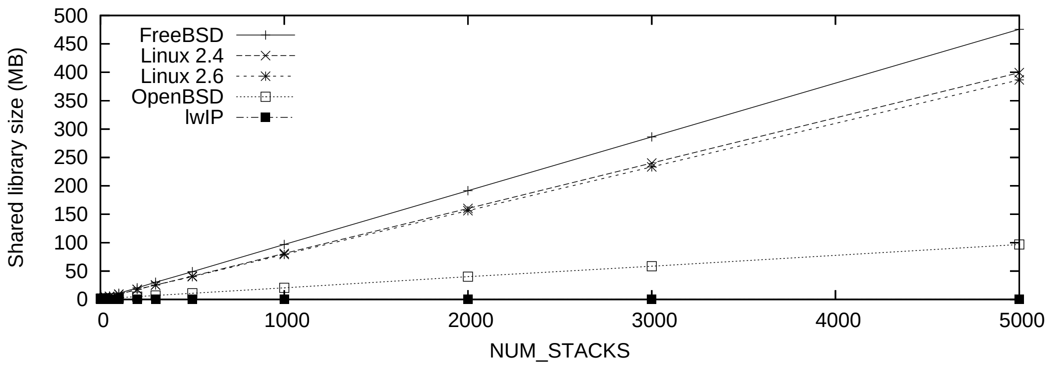
The globaliser adds extra code, potentially including many extra variables, to the project being globalised. This adds extra cost at build time. The extra time to build the shared libraries in the Network Simulation Cradle due to the globaliser is of less consequence than the runtime overhead, as the libraries only need to be built once and can then be used for all subsequent simulations. However, it is important to analyse the extra cost to understand any limitations to scalability of the shared library the globaliser might add, such as producing code that the compiler and/or linker are not able to consume.

To evaluate the offline cost of the globaliser, `NUM_STACKS` is increased from 2 to 5000 and three metrics are evaluated: total build time (preprocessing, running the globaliser, compiling and linking are all included), total file size of the source files output by the globaliser and the size of the shared library produced. Figure 6.8 shows these results for the codebases summarised in table 6.9. The build time was recorded on a computer with dual AMD Opteron 250 CPUs and 8GB of RAM.

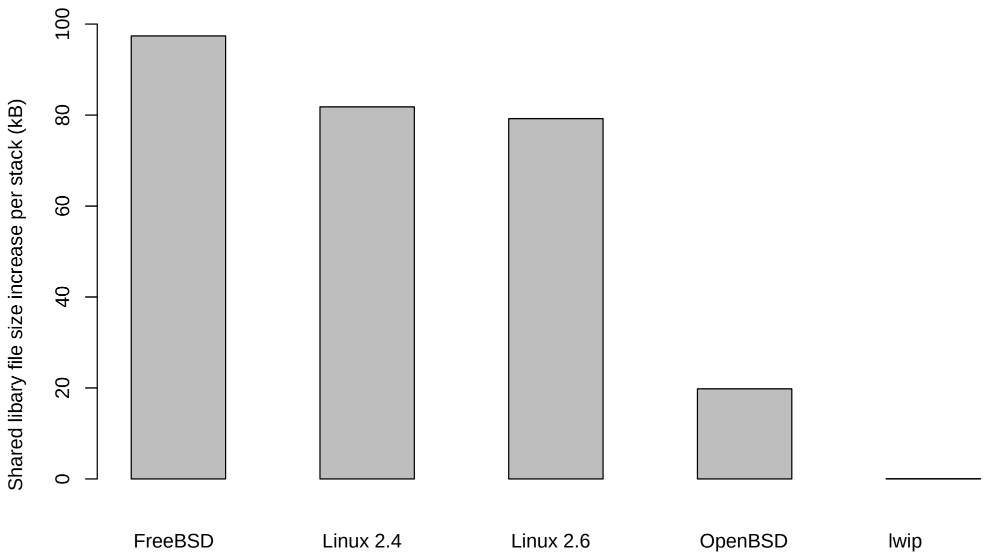
The build time shown in figure 6.8(a) shows that there is significant extra cost to build a project: the time taken rises from 5 minutes to over 7 hours to build the set



(a) Build time



(b) Library size



(c) Library size increase per stack instance

Figure 6.8: Measured offline globaliser costs

of 5 projects as `NUM_STACKS` increases from 2 to 5000. This is due to the large number of extra symbols created by the globaliser. The time to build the shared libraries increases roughly exponentially as `NUM_STACKS` increases. This is due to the increased linking time: the GNU linker exhibits almost exponential increase in time to link as the number of symbols to link increased. As this process is only required once to build the shared libraries, which can then be used for all subsequent simulations, the long build time does not prevent NSC from being useful for many types of simulation. It does prevent simulations of very large scales (for example, one million nodes) but does not prevent the simulations discussed in chapter 2 or reproduced in chapter 5 and chapter 4.

The object, library or executable files produced from the code output from the globaliser will be necessarily larger than code unmodified by the globaliser. If `NUM_STACKS` is set to 2, then every global that is modified is cloned once. This means that twice as much memory is used for these variables than was used originally. Figures 6.8(b) and 6.8(c) show the increase in shared library size. The size reported is after the debug information (such as variable names) has been removed. With higher values of `NUM_STACKS` (10000 or more) the build process fails because it hits a memory limit: the 32-bit linking process attempts to allocate more than 4GB of memory, which results in a memory allocation failure. The reason for the linker requiring more than 4GB of memory is the large number of new symbols that are created for array declarations. It would be possible to design a new algorithm that does not create so many extra symbols (other strategies for source modification are discussed in chapter 3), but this was left for future work; it was not needed to run the scenarios described in this thesis.

6.5 Discussion of performance results

The scenarios covered in this chapter show a difference in real world time to simulate of around 1.8–5.5 times between ns-2 TCP models and NSC TCP implementations. This range is consistent across simple scenarios and more complex scenarios reproduced from the descriptions given in research papers.

Importantly, similar trends were apparent for the simplified ns-2 models and the

NSC implementations for all of the scenarios. When the time to be simulated is increased and other simulation parameters remain constant, time to simulate increases linearly for all TCP models used. The same is true when the bottleneck bandwidth is increased and other parameters are constant. For these scenarios, both ns-2 TCP models and NSC TCP implementations are $\mathcal{O}(n)$ with the NSC TCP implementations having higher constants.

Non-linear trends are apparent when the number of TCP flows is increased: the time to simulate increases roughly exponentially. This can be expressed as $\mathcal{O}(n^k)$ where k differs for each model. k is higher for the NSC TCP implementations. In another scenario, as the packet size increases, the time to simulate decays exponentially ($\mathcal{O}(n^{-k})$ where k differs for each model). The non-linear trends are consistent across all the TCP models used.

The consequence of the difference in time to simulate is that the NSC is suitable for most simulations described in the literature, including those described in section 6.2.2. The higher CPU cost may make the NSC unsuitable for very large scale simulations. However, in these cases, the NSC may still have a role to play in validation: it can be used to simulate a subset of the cases that are to be investigated and its results compared to those produced by the simplified TCP models being used.

The difference in memory use between ns-2's simplified TCP models and the NSC TCP implementations is similar to that of CPU usage: the difference in heap memory used varies from 1.5 times difference to 6.3 times difference for the scenarios studied. The trends are $\mathcal{O}(n)$ where the NSC TCP implementations have higher constants. The total virtual memory (VM) size of the simulator process image is larger when the NSC shared libraries are loaded: the VM size increases from 8MB to 144MB in the case of loading the FreeBSD 5 shared library. This is due to the modifications the globaliser makes to the shared libraries.

The globaliser clones global variables which adds to the size of the shared library that is produced. As the number of TCP instances supported increases, the shared library size increases linearly. With 5000 TCP instances supported, the size of the NSC shared libraries ranges from 100MB to 470MB. This has repercussions other than memory use: the linker is unable to link all shared libraries with 10000 TCP instances supported, as it requires more memory space than is available for a 32-bit process. During runtime, the extra indirection added by the globaliser to references to global variables means that performance is reduced. This varies for each TCP implementation; in the scenario studied the slowdown for FreeBSD and OpenBSD was below 4%, but over 16% for Linux 2.6.

Overall, the NSC is capable of simulating many scenarios but is more limited in scale than the simplified models present in ns-2. Reproducing TCP research carried out by others with the NSC shows that the NSC has practical use and that the limitations to scalability do not inhibit its use by many simulation practitioners.

Chapter 7

Conclusions and future research

This thesis examines using the code from real world TCP implementations in the place of simplified models of TCP in a network simulator. The accuracy of simulating TCP is increased first by using more detailed TCP models and second by using a range of different TCP implementations. Simulating in this way can produce results which give a greater insight into the scenario being simulated than simulating with simplified models would.

7.1 Accuracy of simulating with real world TCP code

In chapter 1 the question “would using real world code to simulate TCP would increase the accuracy of the simulation” is introduced. To investigate this question, several sub-questions were identified: is it feasible to use real world TCP implementations in simulation, is using real world TCP implementations valid and accurate, is this approach to simulation applicable, and can the simulation be carried out with reasonable resource usage.

7.1.1 Feasibility

Previous attempts at producing more accurate TCP simulation results by using real implementations show that such simulation is possible but they are limited in one or more of the following ways:

- only a single TCP implementation is available in the simulator and this version is often not updated;

- the simulator itself is not popular or well tested, so previous simulations cannot easily be run and user experience of well-used simulators is ignored;
- little or no validation studies are undertaken to validate the TCP model;
- the code is modified by hand, increasing the chance of the model not replicating the original system in all details; and
- there is little or no support for simulating a large number of independent instances of the TCP model.

The creation of the Network Simulation Cradle shows that it is feasible to build software that addresses these issues. No other simulators or frameworks address all the issues listed above. The main characteristics of the NSC are:

- multiple TCP implementations (e.g., Linux, FreeBSD, OpenBSD and lwIP);
- multiple versions of a single implementation (Linux 2.4, Linux 2.6.10, Linux 2.6.14.2);
- all supported implementations within one simulator process on a single machine;
- programmatic support for multiple instances so code does not need to be modified by hand to allow TCP instances to run independently;
- scalability to thousands of TCP connections of differing implementations running on a single machine;
- thorough validation studies comparing measurements with simulations;
- support for multiple simulators including a well known simulator (ns-2); and
- support for both simplified TCP models and real world code based TCP models in the same simulator.

7.1.2 Validity and accuracy

The Network Simulation Cradle produces results very similar to the real systems studied. In chapter 4 several TCP packet traces recorded on a test network are compared to packet traces generated from simulations using the NSC. These traces are nearly identical, the same sequence of packets are produced by the NSC and real systems with only small differences in packet timing.

Further comparisons between results from tests on a controlled network and simulation, such as TCP goodput during random packet loss, show simulations with the NSC to provide a very high level of accuracy. The graphs on page 97 show this: the same trends are apparent for simulated and measured results and the absolute values, including confidence intervals, are nearly identical.

When compared to previously published results of a set of very thorough simulations, the NSC produced consistent results. This is shown from page 92 onwards, where the results of reproducing a simulation scenario shown by Anagnostakis *et. al* [155] of the goodput of TCP over a complex topology with multiple congestion points are discussed. Overall the results presented in chapter 4 show the NSC to be very accurate at both micro and macro levels. No previous work reviewed has shown such results.

7.1.3 Applicability

Others have noted the usefulness of simulating with real world code. Brakmo and Peterson [103], when describing perhaps the first work that uses a real BSD TCP/IP stack for network simulation, state:

> "Running the actual TCP code is preferred to running an abstract specification of the protocol; the latter is mostly useful for rapid experimentation."

More recently, Wang *et. al* [104] commented that they use real world TCP implementations in the NCTUns simulator to "generate more accurate simulation results than a traditional TCP/IP network simulator that abstracts a lot away from a real-life TCP/IP implementation." Another example of using real world code for greater accuracy is Julio [107], who used the NetBSD TCP implementation in the OMNeT++ simulator in place of the existing TCP models due to their incorrect behaviour [115].

Simulating with multiple different TCP implementations is also useful. This is due to TCP implementations differing substantially, implementations can produce very different results as chapter 5 shows. This is true not only in simple simulations created to show such differences (such as packet reordering, see section 5.2.1 on page 101), but in scenarios reproduced from published research. In one situation using multiple simulated TCP implementations varied more than using multiple simplified models showed, probably enough to change the conclusions made in [180] (see page 119).

A limitation of the approach used by the NSC and described in chapter 3 requires the TCP implementation source code. This means that closed source stacks, such as the Microsoft TCP stack, cannot be used in the NSC without the source code being released. If the source code were available, conceptually the Microsoft TCP stack could be incorporated into the NSC.

7.1.4 Performance and scalability

Performance tests covered in chapter 6 show that the NSC usually takes longer to simulate than ns-2's simplified TCP models and the difference ranges from less than a 100% difference in time to greater than 500% difference. When reproducing simulation scenarios published by others and in other work done with the NSC, this performance difference did not hinder carrying out the simulations; the time or resource costs were increased by manageable amounts.

The NSC implementation places some upper limits on scalability that would prevent some simulations being carried out that could be carried out with simplified models. Simulations using the NSC and current compilers and hardware are unable to scale past a few thousand TCP instances due largely to the mechanisms used by the globaliser (see page 145 and onwards for a discussion of this). Future work may increase the number of TCP stack instances possible.

The approach that allows real world TCP implementations to run in simulation used by the NSC is scalable and performs well enough for many simulations carried out by network researchers. In chapter 2 the types of simulations performed by users of TCP simulation is reviewed along with the scale of these simulations. All of the research shown in chapter 2 is of a scale that the NSC can simulate, although the simulations are likely to take longer.

7.1.5 Discussion

The implementation and results of using the NSC shows that simulating TCP with real world code is feasible and can be valid, accurate, applicable, fast and scalable. Simulating this way with the NSC provides very accurate results with very small changes needed to configuration: ns-2 simulation scripts only need to refer to a different TCP model name to use the NSC TCP implementations. This means simulating TCP with real world code is accessible and as easy to use as other ns-2 TCP models. The Network Simulation Cradle is freely available for download [199].

Using a range of TCP implementations gives much greater confidence that results of simulating with real world code based TCP models are not skewed by bugs or a single version of a single implementation. Simulating with a range of implementations is important in understanding the range of results possible when running on the Internet where many different versions of many different implementations of TCP exist.

7.2 Future research

There are many possible avenues of further research available with NSC. The comparative studies of TCP presented in this thesis only touch the surface of a large research area. NSC could be used to gain further knowledge on how TCP implementations perform and interact; even seemingly similar TCP implementations (FreeBSD and OpenBSD) were found to act surprisingly differently in some scenarios.

Reproducing existing simulations and using NSC in the place of the ns-2 TCP models shows promise as an interesting venture in validating previous research. There is a large body of simulation research which could be tested in this way, only a small set of these were reproduced and shown in this thesis. At the same time, the NSC is of use to current users of TCP simulation; it can be used as a primary set of models or to validate simplified models.

There are many areas of future research that the NSC could be further developed to support. A list of some of the possibilities follow in the next sections.

7.2.1 Simulating the application layer

If real applications were simulated as well as the real network stack, realistic application level protocols could be simulated. An example would be running the Quagga [200] routing software to realistically simulate BGP and other routing protocols. Simulating real world applications as well as the TCP stack would be a valuable extension to the NSC to capture application behaviour. x-Sim [103] shows one approach of doing so and Ely *et. al.* [113] discuss many of the issues of integrating a user-level network stack with an application in their project, Alpine. Integration with the NSC would expand on previous work like x-Sim by allowing applications to be simulated using all of the stacks in the NSC.

7.2.2 Protocol development environment

The NSC provides a powerful protocol development environment. Protocol code that is typically time consuming to install and test can be tested quickly and reproducibly in a simulator. The NSC can be recompiled and a simulation run rather than building and installing a new kernel or kernel module. The testing is then run in a simulator, which means that the tests will be reproducible. It is possible to debug code that is simulating one machine without fear of affecting results on another simulated machine with simulated time. This is of great benefit to debugging distributed systems code, as problems which can be hard to reproduce (such as race conditions) can be debugged easily in the simulator. It is also possible to test a much wider range of scenarios with the NSC, and also to automate the testing.

Some protocol development has been performed with the NSC to show that this is viable; an early DCCP implementation was tested [201]. The features described above are not common features of existing protocol development environment research [113, 202].

7.2.3 Network stack additions

There are other network stacks which would be interesting to simulate with the NSC:

- Recent versions of the Solaris operating system have been released as open source software in the form of OpenSolaris. This provides another network stack which could be supported in NSC. This is of potential interest because the network stack was independently architectured, not based on the BSD or Linux TCP stack.
- The Microsoft Windows network stack could also be able to be supported if the source code is available.
- Mac OS X uses an open source kernel called Darwin which is based on FreeBSD. Supporting this network stack should be straight forward given the existing support for FreeBSD in NSC.
- There are also non-TCP protocol stacks which are of potential interest. The Space Control Protocol Suite [203] is one, the reference implementation [204] of this could be ported to NSC. Many others are possible, for example the NSC could be extended to support the full set of protocols in the current network stacks such as UDP.

Having further TCP implementations would allow a greater range of comparative studies to be performed with NSC. It would also allow simulations which utilise a broader range of TCP implementations for background traffic. This is desirable, as measurement studies of the Internet show a wide range of TCP implementations in use [185].

7.2.4 Automated protocol testing

The NSC would fit in to an automated testing framework. No human interaction is required to run a test with NSC and produce a set of metrics. An automated test suite could report performance metrics for benchmarking or run compliance tests whenever the source code of a TCP implementation changes. This could provide knowledge on how changes to the source code practically affects the TCP implementation. Another possible route would be to run many previous versions of a TCP stack with NSC to view how TCP performance has changed over time. This would require further work, possibly not all of it automated, to support each version of the TCP implementation in NSC.

7.3 Conclusions

Simulating TCP with the code that is used in real TCP implementations increases simulation accuracy. This thesis has explored simulating TCP in this way. The software developed during this thesis, the Network Simulation Cradle, shows that simulating real world code is feasible. Simulating with this software shows a high level of accuracy. Reproducing TCP simulations from past research shows that the approach is of use and insights into TCP and the differences between implementations can be found. A wide range of future research is possible based on the software and ideas presented in this thesis.

With simple access to simulating with many different implementations and low enough resource costs for the simulations that use real world TCP implementations to be practical, there is little reason not to use such simulation techniques. Simulating with multiple TCP implementations with the NSC is as easy as using traditional simplified models in many scenarios. We believe that simulation practitioners *should* use real world code based TCP models and that simulation practitioners *should* use multiple TCP implementations to see the range of results possible. This is a new approach to TCP simulation and one which brings the benefit of more accurate and valid simulation results.

Appendix A

Publications authored

A.1 Peer reviewed journal articles

Sam Jansen and Anthony McGregor. Static virtualization of C source code. *Software: Practice and Experience*, 38(4):397–416, April 2008.

A.2 Conference papers

Sam Jansen and Anthony McGregor. Validation of simulated real world network stacks. In *Proceedings of the Winter Simulation Conference*, pages 2177–2186, Washington D.C., USA, December 2007. IEEE Press.

Sam Jansen and Anthony McGregor. Performance, validation and testing with the network simulation cradle. In *MASCOTS '06: Proceedings of the 14th IEEE International Symposium on Modeling, Analysis, and Simulation*, pages 355–362, Monterey, California, USA, 2006. IEEE Computer Society.

Sam Jansen and Anthony McGregor. Simulation with real world network stacks. In *WSC '05: Proceedings of the 37th Winter Simulation Conference*, pages 2454–2463, Orlando, Florida, USA, December 2005. Society for Computer Simulation International.

Sam Jansen and Anthony Mcgregor. Measured comparative performance of TCP stacks. In *Passive and Active Measurement Workshop*, volume 3431, pages 329–332, Boston, MA, USA, March 2005.

A.3 Conference papers as a secondary author

Adam Biltcliffe, Michael Dales, Sam Jansen, Thomas Ridge, and Peter Sewell. Rigorous protocol design in practice: An optical packet-switch MAC in HOL. In *14th IEEE International Conference on Network Protocols (ICNP)*, pages 117–126, Santa Barbara, CA, USA, November 2006. IEEE Computer Society.

Mark Apperley, Sam Jansen, Amos Jeffries, Masood Masoodian, Laurie McLeod, Lance Paine, Bill Rogers, Kirsten Thomson, and Tony Voyle. Lecture capture using large interactive display systems. In *ICCE ’02: Proceedings of the International Conference on Computers in Education*, page 143, Auckland, New Zealand, 2002. IEEE Computer Society.

Appendix B

Network Simulation Cradle manual

This appendix covers a step-by-step process for including a new TCP implementation in the Network Simulation Cradle. The first section covers the process of adding the new implementation, while section B.2 discusses the process of testing and validation.

B.1 Adding a new stack

Adding a new network stack to the Network Simulation Cradle involves building a new library with the network stack code in it. This library must implement the interface described in section 3.1.3. The process of building this library for a new network stack is covered in the following sections, presented in chronological order.

B.1.1 Initial build process

The initial build process refers to obtaining the code for the network stack to be simulated and building it into an executable. When a network stack is removed from an operating system, there are many references to operating system functions and variable which will be undefined if the new code is built alone. This section describes how the code is extracted and built, solving the problem of the undefined references.

Extract source code

If the network stack is part of a larger code base that cannot practically be simulated with reasonable resources in good time, such as an operating system, then the important source code needs to be extracted from the system.

Listing B.1: FreeBSD kernel source directory

```
# ls /usr/src/sys
Makefile   gdb        netatm     opencrypto
alpha      geom       netgraph   pc98
amd64      gnu        netinet    pccard
boot       i386       netinet6   pci
cam        i4b        netipsec   posix4
coda       ia64       netipx     rpc
compat     isa        netkey     security
conf       isofs      netnatm    sparc64
contrib    kern       netncp     sys
crypto     libkern    netsmb     tools
ddb        modules    nfs        ufs
dev        net        nfs4client vm
doc        net80211   nfsclient
fs         netatalk   nfsserver
```

Each of the entries in listing B.1 is a directory of source code in the FreeBSD kernel source directory, save the `Makefile`. When creating a TCP/IP simulation model, there are sections of the source code that obviously do not require to be part of the simulation model: device drivers (`dev`), the boot loader (`boot`), the Network File System (`nfs`, `nfs4client`, `nfsclient`, `nfsserver`) and some filesystem code (`geom`, `ufs`, `isofs`).

The important code is located in `netinet`: this contains the TCP and IPv4 protocol implementations. An in depth knowledge of the kernel source layout is not necessary for the user in this step, though it would help. Finding the TCP protocol implementation in a code base has not been a problem in any system studied.

The source code must be copied to a new directory where the project will later be built. A user might copy the entire set of source code, then narrow down which is used at a later date, or start by only copying the code they know to be important (such as the `netinet` directory in the example above). Using only a small amount of code initially is the route usually taken with NSC stacks.

Create build environment

The normal build environment of the system the network stack is extracted from needs to be recorded. In practice, this might mean building the original system with `make` and recording the output with `script` on a UNIX system. This allows the user to view the build flags and compiler invocation used to build the system. The same environment should be reproduced for the extracted code.

Once this new build system is created, replicating the original compiler flags, the system needs to be configured to produce an executable. That is, each source file should be compiled, then all source files should be linked together into an executable.

This may work on the original system the code is from, but not on other systems (for example, the FreeBSD kernel will compile on a FreeBSD system easily, but not on Linux at this stage). This is due to system header files. To isolate the build environment completely from the host system, the compiler is configured to not search standard include paths. All include files that are needed - including those from system include paths (such as `/usr/include/`) - are copied to the new build environment.

Link

Linking the code into an executable will provide a list of undefined references. If the original system is self-contained, such as lwIP [208] is, this list may be short or empty. If the networking code is extracted from a large monolithic kernel, such as the FreeBSD networking code is, the list will be longer.

The linker will output a verbose list of undefined references with many duplicates. The output will need to be processed into a readable form. It is possible to sort the undefined references by their frequency of occurrence, thereby understanding which are critical to the system.

Solve undefined references

If there are many undefined references that are referred to often, it is likely a good idea to include more code from the original system. The new executable should then be compiled and linked again, producing a new list of undefined references. This process can be repeated until the list of undefined references is of a small enough length to satisfy the user.

The rest of the undefined references must be implemented as stub functions and variable declarations. The stub functions should raise an error if called as shown in listing B.2. This means that when the model is being tested at a later date, the functions which are not implemented but need to be will fail, allowing the user to implement these functions.

Listing B.2: Stub function from the FreeBSD support code

```
int seltrue(dev, events, td)
        dev_t dev;
        int events;
        struct thread *td;
{
    assert(0 && "This␣function␣is␣intentionally␣
       unimplemented.");
    return 0;
}
```

There may be many functions which will not be called during the execution of the model that are suitable to have stub functions of the sort in listing B.2. If they are required, the user will need to either implement the function, or copy the original implementation, whichever is more appropriate to the function in question.

Undefined variable symbols are global variable declarations and should be copied exactly as they appear in the original code to preserve their default values. Once the undefined references have been solved, the build system can be configured to output a shared library. The creation of this shared library and cradling code follows.

B.1.2 Shared library creation

The shared library contains up to three parts: the untouched source code to be simulated that is extracted from the original system, possibly any stub functions created, and the cradle code that is described next.

Build cradle

The cradle code within a shared library implements the interface described in section 3.1.3. To implement this interface it needs to provide code to bridge between the network stack code and the simulation interface.

The amount of cradling code will vary depending on the interface the network stack provides. The FreeBSD code in the Network Simulation Cradle does not have the usual BSD sockets API available, only the lower level kernel primitive functions. To implement functionality such as creating a TCP socket, connecting a TCP socket, sending data over a socket and similar, the original syscall implementations were viewed and copied as much as possible. Other functionality will be even more complex, the method used to set the default gateway varies a lot between operating systems and is generally undocumented: FreeBSD uses a routing socket while Linux uses an `ioctl()` call or a Netlink socket.

Initialisation is important and often the most complex part of this process. FreeBSD uses “linker sets” to store initialisation information, the data is stored in different sections of the library or executable and the linker creates special symbols to signify the start and end of these sections. The code which normally sorts the section and executes the initialisation code is part of the kernel `main()` routine, which is not directly usable in simulation. This code, and other initialisation code, needs to be ported to user space. Detail of how the globaliser supports linker sets can be found in section 3.2.3.

Support non-blocking calls

Part of building the cradle will be to interface with functionality in the network stack which may take time to process. The function call may have the option of blocking: not returning until some external event has taken place.

When sending data over a socket using the popular BSD sockets API [209], the `send()` function has the option to block: until there is space in the TCP buffer to enqueue the data, for example. Some systems such as Linux allow the programmer to specify a `MSG_DONTWAIT` flag to prevent the call from blocking, else in many cases the file descriptor can be configured to be non-blocking via the use of an `fcntl()` function call.

When an application is blocked, the operating system pauses the application and resumes it when the event it is waiting for happens. When the application is blocked the operating system continues on and is able to awaken the application at a later time. In simulation with ns-2 there is only one thread of execution. If the application is blocked waiting for an event, the entire program is blocked and will not continue.

The cradle code must use non-blocking calls like those mentioned above for the BSD sockets API. Though it is possible a protocol is implemented to be only blocking, this is not the case for the systems studied and it is unlikely any major network stacks are implemented this way. The Network Simulation Cradle requires non-blocking protocol implementations, though a trial implementation that supports blocking protocols using threads[1] is introduced in earlier work [116] — non-blocking code is not a fundamental requirement of this approach.

Integrate with the simulator

Once the interface is implemented in the shared library, the stack can be used in the ns-2 network simulator. It can then be used in a large range of TCP simulations, though at this stage only a single instance of the stack will be supported.

With other network stacks a part of NSC, the new stack can be tested against one of these. Connecting a socket, sending data, listening, accepting and reading from a socket must all be tested. This process will show any mistakes in the stub functions used (see section B.1.1)—there will be assertion failures whenever a stub function is reached. Whenever this happens, the user must investigate the function and make a decision on how to implement the function. This process continues until a range of simulations can be performed without the program aborting due to an assertion failure.

[1]Another approach would be to use coroutines [210], for which several libraries exist for C/C++.

Add support for multiple instances

The globaliser must be added to the build process to allow support for multiple instances. There are three steps to integrating the globaliser into a build system. Adding the globaliser into the build process, setting it up to output any globals encountered, then building a list of global variables to be modified.

The globaliser described in section 3.2 takes preprocessed C source code as input on `stdin` and outputs modified C on `stdout`. When using the gcc compiler, the command to compile a C file called `sample.c` would be similar to listing B.3. The file `sample.c` is preprocessed. The preprocesed source is then passed through a UNIX pipe to the globaliser, which reads a list of global variables to modify from a file called `globals.txt`. The output of the globaliser is then passed through a UNIX pipe again to gcc, which compiles the file and saves the results in `sample.o`.

Listing B.3: Compiling a C file with gcc and the globaliser

```
gcc ${CFLAGS} sample.c -E - |
    ./globaliser -vv ./globals.txt |
    gcc -xc ${CFLAGS} -c - -o sample.o
```

The globaliser must read a list of globals to modify from a text file. This stops it from modifying variables the user does not want or need modified, for example, the variable used to denote which stack is currently running. The globaliser has a verbose option, `-v`, which makes it output any globals it encounters which it is not already modifying to standard error. This list of variables can be recorded and collated, and any variables that the user does not want modified can be removed. The remaining list can then be saved in the global list file and used in later builds.

B.2 Testing and validation

Once a shared library has been built and supports multiple instances it should be tested and validated. Initial testing was with a single instance to made sure the new stack would perform some basic operations in simulation. This testing neither looked at multiple instances of the stack, nor whether the results produced by the stack were accurate.

B.2.1 Initial testing

The first phase of testing should include the tests run earlier, to check for any regressions running the globaliser on the source. Then the simple tests should be expanded into tests of multiple instances communicating.

Specific features should be tested, such as timers working correctly, the correct amount of data being sent and received and whether the maximum transfer unit is correct. Writing tests is simple because the network simulator is already designed to be quick and easy to produce new simulation scenarios.

This further testing phase ensures that the earlier process of extracting code and then later using the globaliser has produced a simulation model that runs in a variety of scenarios and produces reasonable, if not yet validated, results. More stub functions may be encountered that need to be implemented, or the user may find that timers are not working as expected because of a user error earlier in the process. Once these issues are fixed the simulated network stack can undergo thorough validation.

B.2.2 Validation

The simulator agent of NSC has an option to turn libpcap [130] packet tracing on. The data from the packets created in the real stack code is saved to disk in tcpdump format. This can then be analysed later in the tcpdump program [130], tcptrace [131], Ethereal [211] and others. ns-2 trace format is also supported, allowing visualisation in Nam [212] and use of traditional ns-2 trace analysis.

An advantage of producing libpcap packet traces is that the packet traces can be directly compared to packet traces measured from real machines. A simulation can be modelled after a physical setup, perhaps in a laboratory test network, and packet traces can be measured on the real machines and in simulation. The packet traces can be directly compared on a packet-by-packet basis to see the differences between the two setups. This sort of validation using the Network Simulation Cradle is presented in [163] and in chapter 4.

Appendix C

Network simulators

This appendix covers prominent packet based network simulators encountered during research for this thesis. Only simulators not covered in chapter 2 are introduced here.

C.1 OMNeT++

The OMNeT++ [213, 214] framework is a discrete event simulation environment primarily used for simulating communication networks. It provides an object-oriented component architecture where components are written in C++ and assembled into larger models with a domain specific language called NED [215].

OMNeT++ is a popular simulator used in education [216], for wireless network research [217–221], optical network research [222] and TCP/IP research [107, 221, 223–225].

The simulator

The OMNeT++ simulation kernel supports sequential and parallel simulation [226]. Models are creating by deriving a *module* class which handles *messages* sent to it. Models create new messages and send them via *gates* or directly to other models. Gates are used to connect modules together, a application module might have an *out gate* connected to a TCP model used to send data. The simulation kernel provides utility functions to enable building models from these building blocks, for example, scheduling a message to arrive at a model after a specified amount of time (a timer).

Simple modules are combined into *compound modules* by grouping them in NED files, allowing unlimited compound module hierarchy levels. NED files also list module parameters and gates.

OMNeT++ includes a Graphical User Interface (GUI) that shows a graphical depiction of the simulation scenario and the events being generated. The simulation components and messages can be examined in detail and the speed of the simulation is directly controlled by the user. Simulations can also be run outside of the GUI. Several tools are included to aid visualisation and analysis of simulation results.

TCP model

OMNeT++ includes a basic TCP model in its INET framework which is documented [227] to support RFC 793 [10], RFC 1122 [228] and RFC 2001 [164]. The following mechanisms are listed as implemented: connection setup and tear down, segmentation, receive buffering for out of order data, delayed acknowledgements, Nagle's algorithm, Jacobson's and Karn's algorithms. TCP Tahoe and TCP Reno are both available.

The model does not include any more recent TCP advances that are used widely on the Internet today such as selective acknowledgements and TCP timestamps. Timers are not based on "fast" and "slow" timers as BSD implementations are, meaning timer granularity can be different to real implementations.

The TCP model has a basic testing suite. A series of test scenarios is run and traces from the simulations is checked to see whether it matches known good output. The tests check whether functionality such as Nagle's algorithm, delayed ACKs, retransmission and connection establishment work.

Due to the limited TCP model there are several attempts at providing TCP models based on real world protocol code [106, 107]. These are described in section 2.3.

C.2 SSF

SSF provides a single interface for discrete-event simulation known as the SSF API [229]. It is designed to support high-performance simulation by making it possible to build models that are efficient, scale well and utilise parallel processor resources. The API specifies the use of either Java or C++.

The SSF API is based around five base classes: `Entities`, which contain state, `Processes` that operate on that state, `inChannels` and `outChannels` that define the endpoints of communication channels. `Events` are objects that are passed between entities to communicate. On top of the basic simulation core classes domain specific component layers are built such as an IP networking layer. Models are composed and configured via a hierarchical attribute tree language known as Domain Modeling Language (DML).

There are several implementations including commercial and reference implementations. Two major implementations that are used in research are Dartmouth SSF [230] and SSFNet [231].

SSFNet

SSFNet [231] is a set of open source Java models of communication elements (such as TCP, UDP, BGP, routers and LANs) for SSF. Research using SSFNet includes large scale Internet simulations such as BGP simulations [232, 233], studies of TCP dynamics [234] and worm traffic [235].

The TCP model used in SSFNet implements basic RFC 793 [10] and RFC 2581 [182] congestion control including fast retransmit, fast recovery, duplicate acknowledgements and slow start. Segments are always the maximum size and data is immediately consumed by the receiver.

TCP model validation is based on the ns-2 validation tests. Traces produced by ns-2 and SSFNet are graphed and compared by hand. 14 tests are compared against ns version 2.1b4. ns-2 has changed in further versions, including bug fixes to TCP that changes the behaviour validated against. The comparison of traces is thorough but the results of the validation only show that the SSFNet TCP models are consistent with a specific version of another simulator, not real TCP implementations.

DaSSF

Dartmouth SSF, otherwise known as DaSSF, is implemented in C++ and has been used for simulations intended to model the global Internet [236, 237] and large scale sensor networks [238]. Dartmouth SSF is also known as iSSF in recent work [239]. There are several TCP/IP models for DaSSF: DaSSFNet, a port of the Java-based SSFNet, a custom TCP/IP model for DaSSF [240] and fluid models for TCP [241, 242].

The TCP model described in [240] implements basic TCP functionality (RFC 793 [10]) apart from buffering out of order packets that are within the receivers advertised window. The Nagle algorithm and the Silly Window Syndrome (SWS) fix are not implemented. Basic validation is performed by analysing time-sequence graphs in scenarios that test the features of TCP implemented in the simulation model.

Another approach to TCP simulation with DaSSF is using fluid models. Fluid modelling of TCP produces very fast and scalable simulations at the cost of accuracy. The model described by Nicol [241] combines discrete event and fluid modelling to simulate slow start, congestion avoidance, time-outs, lost data and fast retransmits. There is potential for very large speed-ups with such a model though there are still performance problems in some situations [242]. The accuracy of such modelling has been shown to be adequate in many scenarios TCP operates in [243], but there is question over the accuracy in general, further verification and validation work is required [241, 242].

C.3 GTNeTS

The Georgia Tech Network Simulator [244] (GTNeTS) is written in C++ and designed and used for large scale network simulation [245, 246] such as Internet worms [247]. GTNeTS includes TCP models for Reno, Newreno, Tahoe and Sack. There is no information on validation performed on these TCP models.

C.4 J-Sim

J-Sim [248, 249], formerly known as JavaSim, is a component-based simulator written in Java with an emphasis on network simulation. It is mostly used for wireless sensor network research [248, 250] but includes TCP models for the Reno, Tahoe and Vegas congestion control algorithms, which support delayed acknowledgements and understand ECN. No information is provided on verification and validation of the simulation models.

C.5 JiST

The JiST [251] simulator utilises the Java virtual machine to perform fast, scalable simulations. The SWANS framework uses this simulator to simulate wireless networks [252]. This framework includes a TCP model [253]. This model implements basic TCP functionality described in RFC 793 [10] and RFC 2581 [182].

C.6 IRLSim

IRLSim [108] is a general purpose packet level network simulator. It was originally designed to simulate the Resource Reservation Protocol (RSVP) [254] but expanded over time to be general purpose and include TCP/IP models. IRLSim is based on the Parsec [87] simulation language which allows sequential and parallel simulation. Parsec code is similar to C and porting code between the two languages is easy [108].

The TCP model used in IRLSim is a port of the BSD 4.4-Lite [126] TCP implementation. No validation information about the simulator or its models is provided. IRLSim is used in some RSVP [255] and routing [256] research.

References

[1] Michael Dales and Madeleine Glick. SWIFT: A high capacity wavelength-striped optically switched network with electronic control. In *INFOCOM Poster Session*, Miami, FL, USA, March 2005.

[2] Matt Mathis, John Heffner, and Raghu Reddy. Web100: extended TCP instrumentation for research, education and diagnosis. *SIGCOMM Comput. Commun. Rev.*, 33(3):69–79, July 2003.

[3] John Heidemann, Kevin Mills, and Sri Kumar. Expanding confidence in network simulation. *IEEE Network Magazine*, 15(5):58–63, Sept./Oct. 2001.

[4] Sally Floyd and Van Jacobson. Traffic phase effects in packet-switched gateways. *Journal of Internetworking:Practice and Experience*, 3(3):115–156, September 1992.

[5] Sally Floyd. Simulator tests. Technical report, Lawrence Berkeley Laboratory, May 1997.

[6] Kenjiro Cho, Koushirou Mitsuya, and Akira Kato. Traffic data repository at the WIDE project. In *USENIX, FREENIX Track*, pages 263–270, San Diego, CA, June 2000.

[7] A. Romanow and S. Floyd. The dynamics of TCP traffic over ATM networks. *IEEE Journal on Selected Areas In Communications*, May 1995.

[8] Mario Gerla, Ken Tang, and Rajive Bagrodia. TCP performance in wireless multi-hop networks. In *WMCSA '99: Proceedings of the Second IEEE Workshop on Mobile Computer Systems and Applications*, Washington, DC, USA, 1999. IEEE Computer Society.

[9] Lisong Xu, Khaled Harfoush, and Injong Rhee. Binary increase congestion control (BIC) for fast long-distance networks. In *IEEE Infocom*. IEEE, 2004.

[10] J. Postel. Transmission Control Protocol. RFC0793, September 1981.

[11] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP Selective Acknowledgement Options. RFC2018, October 1996.

[12] R. Ludwig and A. Gurtov. The Eifel Response Algorithm for TCP. RFC4015, February 2005.

[13] The network simulator - ns-2. `http://www.isi.edu/nsnam/ns/`, Accessed 2008.

[14] David Wetherall. Otcl: MIT Object Tcl. `http://otcl-tclcl.sourceforge.net/otcl/`, Accessed 2006.

[15] Google scholar. `http://scholar.google.com`, Accessed 2006.

[16] Citeseer scientific literature digital library. `http://citeseer.ist.psu.edu/`, Accessed 2006.

[17] 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. *IEEE Computer*, 33(5):59–67, 2000.

[18] Tom Henderson, Sumit Roy, Sally Floyd, and George Riley. ns-3 project plan. `http://www.icir.org/floyd/talks/ns3-Jun06.pdf`, June 2006.

[19] Matthew Mathis and Jamshid Mahdavi. Forward acknowledgement: refining TCP congestion control. In *SIGCOMM ’96: Conference proceedings on Applications, technologies, architectures, and protocols for computer communications*, volume 26, pages 281–291, New York, NY, USA, October 1996. ACM Press.

[20] Kevin Fall and Sally Floyd. Simulation-based comparisons of Tahoe, Reno and SACK TCP. *SIGCOMM Comput. Commun. Rev.*, 26(3):5–21, July 1996.

[21] Sally Floyd. Validation experiences with the ns simulator. Technical report, ACIRI, April 1999.

[22] Michael Neufeld, Ashish Jain, and Dirk Grunwald. Nsclick: bridging network simulation and deployment. In *MSWiM ’02: Proceedings of the 5th ACM international workshop on Modeling analysis and simulation of wireless and mobile systems*, pages 74–81, New York, NY, USA, 2002. ACM Press.

[23] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and Frans M. Kaashoek. The click modular router. *ACM Trans. Comput. Syst.*, 18(3):263–297, August 2000.

[24] X. A. Dimitropoulos and G. F. Riley. Creating realistic BGP models. In *Modeling, Analysis and Simulation of Computer Telecommunications Systems, 2003. MASCOTS 2003. 11th IEEE/ACM International Symposium on*, pages 64–70, 2003.

[25] Inetquagga. `http://www.omnetpp.org/pmwiki/index.php?n=Main.INETQuagga`, Accessed 2008.

[26] R. J. Gurski and C. L. Williamson. TCP over ATM: simulation model and performance results. In *Computers and Communications, 1996., Conference Proceedings of the 1996 IEEE Fifteenth Annual International Phoenix Conference on*, pages 328–335, 1996.

[27] Guang Lu, R. Simmonds, Xiao Zhonge, B. Unger, and C. Williamson. The performance of TCP over ATM on lossy ADSL networks. In *Local Computer Networks, 2000. LCN 2000. Proceedings. 25th Annual IEEE Conference on*, pages 418–427, 2000.

[28] D. Comer and J. Lin. TCP buffering and performance over an ATM network, March 1995.

[29] Brian W. Unger, Fabian Gomes, Xiao Zhonge, Pawel Gburzynski, Theodore Ono-Tesfaye, Srinivasan Ramaswamy, Carey Williamson, and Alan Covington. A high fidelity ATM traffic and network simulator. In *WSC '95: Proceedings of the 27th conference on Winter simulation*, pages 996–1003, New York, NY, USA, 1995. ACM Press.

[30] H. Obata, K. Ishida, J. Funasaka, and K. Amano. TCP performance analysis on asymmetric networks composed of satellite and terrestrial links. In *ICNP '00: Proceedings of the 2000 International Conference on Network Protocols*, Washington, DC, USA, 2000. IEEE Computer Society.

[31] Yong Bai, Pengfei Zhu, A. Rudrapatna, and A. T. Ogielski. Performance of TCP/IP over IS-2000 based CDMA radio links. In *IEEE Vehicular Technology Conference, 2000*, volume 3, pages 1036–1040, 2000.

[32] M. Gerla, R. Bagrodia, L. Zhang, K. Tang, and L. Wang. TCP over wireless multihop protocols: Simulation and experiments. In *IEEE International Conference on Communications (ICC)*, June 1999.

[33] Gavin Holland and Nitin H. Vaidya. Analysis of tcp performance over mobile ad hoc networks. In *Proceedings of IEEE/ACM MOBICOM '99*, pages 219–230, August 1999.

[34] Z. Fu, P. Zerfos, H. Luo, S. Lu, L. Zhang, and M. Gerla. The impact of multihop wireless channel on TCP throughput and loss. In *INFOCOM*, April 2003.

[35] Tianbo Kuang and Carey Williamson. A bidirectional multi-channel MAC protocol for improving TCP performance on multihop wireless ad hoc networks. In *MSWiM '04: Proceedings of the 7th ACM international symposium on Modeling, analysis and simulation of wireless and mobile systems*, pages 301–310, New York, NY, USA, 2004. ACM Press.

[36] Kaixin Xu, Mario Gerla, Lantao Qi, and Yantai Shu. Enhancing TCP fairness in ad hoc wireless networks using neighborhood RED. In *MobiCom '03: Proceedings of the 9th annual international conference on Mobile computing and networking*, pages 16–28, New York, NY, USA, 2003. ACM Press.

[37] Rajesh Krishnan and James P. Sterbenz. TCP over load-reactive links. In *International Conference on Networking Protocols*, Washington, DC, USA, 2001. IEEE Computer Society.

[38] Aleksandar Kuzmanovic and Edward W. Knightly. Low-rate TCP-targeted denial of service attacks: the shrew vs. the mice and elephants. In *SIGCOMM '03: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 75–86, New York, NY, USA, 2003. ACM Press.

[39] G. Neglia and V. Falletta. Is TCP packet reordering always harmful? In *Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, 2004. (MASCOTS 2004). Proceedings. The IEEE Computer Society's 12th Annual International Symposium on*, pages 87–94, 2004.

[40] Liang Guo and Ibrahim Matta. The war between mice and elephants. In *International Conference on Network Protocols*, July 2001.

[41] Wesley M. Eddy and Mark Allman. A comparison of RED's byte and packet modes. *Comput. Networks*, 42(2):261–280, June 2003.

[42] Sally Floyd, Ramakrishna Gummadi, and Scott Shenker. Adaptive RED: An algorithm for increasing the robustness of RED's active queue management. Available http://www.icir.org/floyd/papers/adaptiveRed.pdf, August 2001.

[43] Zhang Heying, Liu Baohong, and Dou Wenhua. Design of a robust active queue management algorithm based on feedback compensation. In *SIGCOMM '03: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 277–285, New York, NY, USA, 2003. ACM Press.

[44] Ningning Hu and Peter Steenkiste. Improving TCP startup performance using active measurements: Algorithm and evaluation. In *ICNP '03: Proceedings of the 11th IEEE International Conference on Network Protocols*, Washington, DC, USA, 2003. IEEE Computer Society.

[45] H. Wang and C. Williamson. A new scheme for TCP congestion control: Smooth-start and dynamic recovery. In *MASCOTS '98: Proceedings of the 6th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, Washington, DC, USA, 1998. IEEE Computer Society.

[46] Ming Zhang, Brad Karp, Sally Floyd, and Larry Peterson. RR-TCP: A reordering-robust TCP with DSACK. In *ICNP '03: Proceedings of the 11th IEEE International Conference on Network Protocols*, Washington, DC, USA, 2003. IEEE Computer Society.

[47] Ethan Blanton and Mark Allman. On making TCP more robust to packet reordering. *SIGCOMM Comput. Commun. Rev.*, 32(1):20–30, January 2002.

[48] S. Floyd, J. Mahdavi, M. Mathis, and M. Podolsky. An Extension to the Selective Acknowledgement (SACK) Option for TCP. RFC2883, July 2000.

[49] Emulab. `http://www.emulab.net/`, Accessed 2006.

[50] Carey Williamson and Qian Wu. A case for context-aware TCP/IP. *SIGMETRICS Perform. Eval. Rev.*, 29(4):11–23, March 2002.

[51] Guanghui He, Yuan Gao, Jennifer C. Hou, and Kihong Park. A case for exploiting self-similarity of network traffic in tcp congestion control. *Comput. Networks*, 45(6):743–766, August 2004.

[52] Saverio Mascolo, Claudio Casetti, Mario Gerla, M. Y. Sanadidi, and Ren Wang. TCP westwood: Bandwidth estimation for enhanced transport over wireless links. In *MobiCom '01: Proceedings of the 7th annual international conference on Mobile computing and networking*, pages 287–297, New York, NY, USA, 2001. ACM Press.

[53] Luigi A. Grieco and Saverio Mascolo. Performance evaluation and comparison of Westwood+, New Reno, and Vegas TCP congestion control. *SIGCOMM Comput. Commun. Rev.*, 34(2):25–38, April 2004.

[54] L. S. Brakmo and L. L. Peterson. TCP vegas: end to end congestion avoidance on a global internet. *IEEE Journal on Selected Areas in Communications*, 13(8):1465–1480, 1995.

[55] Carlo Caini and Rosario Firrincieli. TCP Hybla: a TCP enhancement for heterogeneous networks. *International Journal of Satellite Communications and Networking*, 22(5):547–566, August 2004.

[56] Cheng P. Fu and Soung C. Liew. TCP Veno: TCP enhancement for transmission over wireless access networks. *IEEE Journal on Selected Areas in Communications*, 21(2):216–228, February 2003.

[57] D. Leith and R. Shorten. H-TCP: TCP for high-speed and long-distance networks. In *Proceedings of the 2nd Workshop on Protocols for Fast Long Distance Networks*, Argonne, Canada, 2004.

[58] Tom Kelly. Scalable TCP: improving performance in highspeed wide area networks. *SIGCOMM Comput. Commun. Rev.*, 33(2):83–91, April 2003.

[59] C. Jin, D. Wei, and S. Low. FAST TCP: Motivation, architecture, algorithms, performance. In *INFOCOM*, 2004.

[60] I. Khalifa and L. Trajkovic. An overview and comparison of analytical TCP models. In *Proceedings of the 2004 International Symposium on Circuits and Systems*, volume 5, pages V–469–V–472 Vol.5, 2004.

[61] Farooq Anjum and Leandros Tassiulas. Comparative study of various TCP versions over a wireless link with correlated losses. *IEEE/ACM Trans. Netw.*, 11(3):370–383, June 2003.

[62] Michele Garetto, Renato Lo Cigno, Michela Meo, and Marco A. Marsan. Closed queueing network models of interacting long-lived tcp flows. *IEEE/ACM Trans. Netw.*, 12(2):300–311, April 2004.

[63] Bing Wang, Jim Kurose, Prashant Shenoy, and Don Towsley. Multimedia streaming via TCP: an analytic performance study. *SIGMETRICS Perform. Eval. Rev.*, 32(1):406–407, June 2004.

[64] Aditya Akella, Srinivasan Seshan, Richard Karp, Scott Shenker, and Christos Papadimitriou. Selfish behavior and stability of the internet: a game-theoretic analysis of TCP. *SIGCOMM Comput. Commun. Rev.*, 32(4):117–130, October 2002.

[65] Xin Liu and Andrew A. Chien. Realistic large-scale online network simulation. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, Washington, DC, USA, 2004. IEEE Computer Society.

[66] Garrett R. Yaun, David Bauer, Harshad L. Bhutada, Christopher D. Carothers, Murat Yuksel, and Shivkumar Kalyanaraman. Large-scale network simulation techniques: examples of TCP and OSPF models. *SIGCOMM Comput. Commun. Rev.*, 33(3):27–41, July 2003.

[67] G. F. Lucio, M. Paredes-Farrera, E; F. Jammeh, and M. J. Reed. OPNET modeler and ns-2 - comparing the accuracy of network simulators for packet-level analysis using a network testbed. *WSEAS Transactions on Computers*, 2(3):700–707, July 2003.

[68] Johan Garcia, Stefan Alfredsson, and Anna Brunstrom. The impact of loss generation on emulation-based protocol evaluation. In *PDCN'06: Proceedings of the 24th IASTED international conference on Parallel and distributed computing and networks*, pages 231–237, Anaheim, CA, USA, 2006. ACTA Press.

[69] S. Naicken, B. Livingston, A. Basu, S. Rodhetbhai, I. Wakeman, and D. Chalmers. The state of peer-to-peer simulators and simulations. *SIGCOMM Comput. Commun. Rev.*, 37(2):95–98, April 2007.

[70] Roberto Canonico, Donato Emma, and Giorgio Ventre. Extended Nam: An ns2-compatible network topology editor for simulation of web caching systems on large network topologies, October 2003.

[71] Srinivasan Keshav. REAL: A network simulator. Technical report, University of California at Berkeley, Berkeley, CA, USA, 1988.

[72] Alexander Dupuy, Jed Schwartz, Yechiam Yemini, and David Bacon. NEST: a network simulation and prototyping testbed. *Commun. ACM*, 33(10):63–74, October 1990.

[73] M. Handley, J. Padhye, and S. Floyd. TCP congestion window validation. Technical report, University of Massachusetts, Amherst, MA, USA, 1999.

[74] G. Hasegawa, K. Kurata, and M. Murata. Analysis and improvement of fairness between TCP Reno and Vegas for deployment of TCP Vegas to the internet. In *ICNP '00: Proceedings of the 2000 International Conference on Network Protocols*, Washington, DC, USA, 2000. IEEE Computer Society.

[75] M. C. Weigle, K. Jeffay, and F. D. Smith. Quantifying the effects of recent protocol improvements to standards-track TCP. In *11th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer Telecommunications Systems*, pages 226–229, 2003.

[76] Qian Wu and Carey Williamson. Improving ensemble-TCP performance on asymmetric networks. In *Proceedings of the Ninth International Symposium in Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, Washington, DC, USA, 2001. IEEE Computer Society.

[77] John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley Professional, March 1994.

[78] V. Jacobson. Congestion avoidance and control. *SIGCOMM Comput. Commun. Rev.*, 18(4):314–329, August 1988.

[79] V. Jacobson. Modified TCP congestion control and avoidance algorithms. end2end-interest mailing list, April 1990.

[80] S. Floyd and T. Henderson. The NewReno Modification to TCP's Fast Recovery Algorithm. RFC2582, April 1999.

[81] K. Fall, S. Floyd, and T. Henderson. Ns simulator tests for Reno FullTcp, 1997.

[82] ns-2 validation tests. `http://www.isi.edu/nsnam/ns/ns-tests.html`, Accessed 2006.

[83] Fabian Gomes, John Cleary, Alan Covington, Steve Franks, Brian Unger, and Zhong-E Ziao. SimKit: a high performance logical process simulation class library in c++. In *WSC '95: Proceedings of the 27th conference on Winter simulation*, pages 706–713, New York, NY, USA, 1995. ACM Press.

[84] Gary R. Wright and Richard W. Stevens. *The Implementation (TCP/IP Illustrated, Volume 2)*. Addison-Wesley Professional, January 1995.

[85] V. Jacobson, R. Braden, and D. Borman. TCP Extensions for High Performance. RFC1323, May 1992.

[86] Xiang Zeng, Rajive Bagrodia, and Mario Gerla. GloMoSim: A library for parallel simulation of large-scale wireless networks. In *Workshop on Parallel and Distributed Simulation*, pages 154–161, 1998.

[87] R. Bagrodia, R. Meyer, M. Takai, Yu-An Chen, Xiang Zeng, J. Martin, and Ha Y. Song. Parsec: a parallel simulation environment for complex systems. *IEEE Computer*, 31(10):77–85, 1998.

[88] Gang Zhou, Tian He, Sudha Krishnamurthy, and John A. Stankovic. Impact of radio irregularity on wireless sensor networks. In *MobiSys '04: Proceedings of the 2nd international conference on Mobile systems, applications, and services*, pages 125–138, New York, NY, USA, 2004. ACM Press.

[89] K. Tang and M. Gerla. MAC reliable broadcast in ad hoc networks. In *Communications for Network-Centric Operations: Creating the Information Force. IEEE Military Communications Conference*, volume 2, pages 1008–1013 vol.2, 2001.

[90] Sonja Buchegger and Jean-Yves Le Boudec. Performance analysis of the CONFIDANT protocol. In *MobiHoc '02: Proceedings of the 3rd ACM international symposium on Mobile ad hoc networking & computing*, pages 226–236, New York, NY, USA, 2002. ACM Press.

[91] Scalable Network Technologies. Qualnet network simulator. `http://www.scalable-networks.com/`, Accessed 2006.

[92] Haejung Lim, Kaixin Xu, and M. Gerla. TCP performance over multipath routing in mobile ad hoc networks. In *Communications, 2003. ICC '03. IEEE International Conference on*, volume 2, pages 1064–1068 vol.2, 2003.

[93] Rajive Bagrodia and Mineo Takai. Position paper on validation of network simulation models. In *DARPA/NIST Network Simulation Validation Workshop*, May 1999.

[94] Inc. OPNET Technologies. Opnet modeler. `ttp://www.opnet.com/products/modeler/`, Accessed 2006.

[95] Xinjie Chang. Network simulations with opnet. In *WSC '99: Proceedings of the 31st conference on Winter simulation*, pages 307–314, New York, NY, USA, 1999. ACM Press.

[96] Chengyu Zhu, O. W. W. Yang, J. Aweya, M. Ouellette, and D. Y. Montuno. A comparison of active queue management algorithms using the OPNET Modeler. *IEEE Communications Magazine*, 40(6):158–167, 2002.

[97] J. W. K. Wong and V. C. M. Leung. Improving end-to-end performance of TCP using link-layer retransmissions over mobile internetworks. In *IEEE International Conference on Communications*, volume 1, pages 324–328, 1999.

[98] Jing Wu, Peng Zhang, Tao Du, Jian Ma, and Shiduan Cheng. Improving TCP performance in ATM network by the fast TCP flow control. In *International Conference on Communication Technology*, volume vol.2, pages 5 pp. vol.2+, 1998.

[99] J. B. Pippas and I. S. Venieris. A RED variation for delay control. In *IEEE International Conference on Communications*, volume 1, pages 475–479, 2000.

[100] Wan G. Zeng, Meihua Zhan, Zhiwen Lin, and Ljiljana Trajkovic. Improving TCP performance with periodic disconnections over wireless links. In *OPNETWORK*, Washington D.C., August 2003.

[101] F. Baccelli, D. R. Mcdonald, and J. Reynier. A mean-field model for multiple TCP connections through a buffer implementing RED. *Perform. Eval.*, 49(1-4):77–97, 2002.

[102] Norman C. Hutchinson and Larry L. Peterson. The X-Kernel: An architecture for implementing network protocols. *IEEE Trans. Softw. Eng.*, 17(1):64–76, January 1991.

[103] Lawrence S. Brakmo and Larry L. Peterson. Experiences with network simulation. In *SIGMETRICS '96: Proceedings of the 1996 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, volume 24, pages 80–90, New York, NY, USA, May 1996. ACM Press.

[104] 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.

[105] Craig Bergstrom, Srinidhi Varadarajan, and Godmar Back. The distributed open network emulator: Using relativistic time for distributed scalable simulation. In *20th Workshop on Principles of Advanced and Distributed Simulation*, pages 19–28, 2006.

[106] Roland Bless and Mark Doll. Integration of the FreeBSD TCP/IP-stack into the discrete event simulator OMNeT++. In *Winter Simulation Conference*, pages 1556–1561, December 2004.

[107] Perez Julio. MQTT performance analysis with OMNeT++. Master's thesis, Networking Insitut Eurecom, September 2005.

[108] A. Terzis, K. Nikoloudakis, Lan Wang, and Lixia Zhang. IRLSim: a general purpose packet level network simulator. In *33rd Annual Simulation Symposium*, pages 109–120, 2000.

[109] David X. Wei and Pei Cao. Ns-2 TCP-Linux: an ns-2 TCP implementation with congestion control algorithms from Linux. In *WNS2 '06: Proceeding from the 2006 workshop on ns-2: the IP network simulator*, New York, NY, USA, 2006. ACM Press.

[110] Scalable Network Technologies. GloMoSim and parsec source code distribution. `http://pcl.cs.ucla.edu/projects/glomosim/obtaining_glomosim.html`, Accessed 2007.

[111] University of Arizona. x-kernel and x-sim source code distribution. `http://www.cs.arizona.edu/projects/xkernel/software.html`, Accessed 2007.

[112] Christopher C. Knestrick. Lunar: A user-level stack library for network emulation. Master's thesis, Virginia Tech, February 2004.

[113] 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.

[114] Srinidhi Varadarajan. The Weaves runtime framework. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, pages 197+, 2004.

[115] Jeroen Idserda. TCP/IP modeling in OMNeT++. B-Assignment, July 2004.

[116] Sam Jansen. Network simulation cradle report. Technical report, Department of Computer Science, The University of Waikato, 2003.

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

[118] Marko Zec. Implementing a clonable network stack in the freebsd kernel. In *USENIX Annual Technical Conference*, pages 137–150, 2003.

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

[120] GNU Project - Free Software Foundation (FSF). Flex. `http://www.gnu.org/software/flex/`, Accessed 2006.

[121] GNU Project - Free Software Foundation (FSF). Bison. `http://www.gnu.org/software/bison/`, Accessed 2006.

[122] International Organization for Standardization. *ISO/IEC 9899:1999: Programming Languages — C*. International Organization for Standardization, Geneva, Switzerland, December 1999.

[123] Jeff Lee. ANSI C grammar. `ftp://ftp.uu.net/usenet/net.sources/ansi.c.grammar.Z`, Accessed 2006.

[124] GNU Project - Free Software Foundation (FSF). GNU compiler collection: C compiler. `http://www.gnu.org/software/gcc/`, Accessed 2005.

[125] GNU Project - Free Software Foundation (FSF). Using the GNU compiler collection: C extensions. `http://gcc.gnu.org/onlinedocs/gcc-4.0.2/gcc/index.html#toc_C-Extensions`, Accessed 2005.

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

[127] Osman Balci. Verification, validation and accreditation of simulation models. In *Proceedings of the Winter Simulation Conference*, 1997.

[128] John S. Carson. Verification validation: model verification and validation. In *WSC '02: Proceedings of the 34th conference on Winter simulation*, pages 52–58. Winter Simulation Conference, 2002.

[129] Robert G. Sargent. Verification and validation of simulation models. In *WSC '03: Proceedings of the 35th conference on Winter simulation*, pages 37–48. Winter Simulation Conference, 2003.

[130] Van Jacobson, Craig Leres, and Steven Mccanne. tcpdump. `http://www.tcpdump.org`, Accessed 2005.

[131] Shaun Ostermann. tcptrace. `http://www.tcptrace.org`, Accessed 2006.

[132] WAND network research group. `http://www.wand.net.nz/`, Accessed 2008.

[133] Brendon Jones. WAND emulation network. `http://www.wand.net.nz/~bcj3/emulation/`, Accessed 2006.

[134] Brendon Jones. Architecture and trial implementation of a performance testing framework. Technical report, Waikato University, 2004.

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

[136] Yasushi Saito, Christos Karamanolis, Magnus Karlsson, and Mallik Mahalingam. Taming aggressive replication in the pangaea wide-area file system. *SIGOPS Oper. Syst. Rev.*, 36(SI):15–30, 2002.

[137] Yin Zhang, Lee Breslau, Vern Paxson, and Scott Shenker. On the characteristics and origins of internet flow rates. In *SIGCOMM '02: Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, volume 32, pages 309–322, New York, NY, USA, October 2002. ACM Press.

[138] Luigi Rizzo. pgmcc: a TCP-friendly single-rate multicast congestion control scheme. In *SIGCOMM '00: Proceedings of the conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, volume 30, pages 17–28, New York, NY, USA, October 2000. ACM Press.

[139] Mark Allman and Aaron Falk. On the effective evaluation of TCP. *SIGCOMM Comput. Commun. Rev.*, 29(5):59–70, October 1999.

[140] Luigi Rizzo. IP_DUMMYNET documentation. `http://info.iet.unipi.it/~luigi/ip_dummynet/`, Accessed 2006.

[141] W. A. Vanhonacker. Evaluation of the FreeBSD dummynet network performance simulation tool on a pentium-4 based ethernet bridge. Technical report, Center for Advanced Internet Architectures, Swinburne University of Technology, Melbourne, Australia, December 2003.

[142] Mark Carson and Darrin Santay. NIST net: a linux-based network emulation tool. *SIGCOMM Computer Communications Revue*, 33(3):111–126, July 2003.

[143] S. Hemminger. Network emulation with netem. In *Linux Conf Au*, April 2005.

[144] K. Fall. Network emulation in the VINT/NS simulator. In *Proceedings of the fourth IEEE Symposium on Computers and Communications*, 1999.

[145] J. Cleary, S. Donnelly, I. Graham, A. Mcgregor, and M. Pearson. Design principles for accurate passive measurement. In *The First Passive and Active Measurement Workshop*, pages 1–7, Hamilton, New Zealand, April 2000.

[146] Sam Jansen. tcpperf - tcp performance tool. `http://www.wand.net.nz/~stj2/nsc/software.html`, Accessed 2006.

[147] Ajay Tirumala, Feng Qin, Jon Dugan, Jim Ferguson, and Kevin Gibbs. Iperf version 1.7.0. `http://dast.nlanr.net/Projects/Iperf/`, Accessed 2006.

[148] Ajay Tirumala, Les Cottrell, and Tom Dunigan. Measuring end-to-end bandwidth with iperf using web100. In *Passive and Active Monitoring Workshop*, San Diego, CA, USA, April 2003.

[149] S. Tao, L. K. Jacob, and A. Ananda. A TCP socket buffer auto-tuning daemon. In *Proceedings of the 12th International Conference on Computer Communications and Networks*, pages 299–304, 2003.

[150] Ross Mcillroy. Network router resource virtualisation. Master's thesis, University of Glasgow, 2005.

[151] Sam Jansen. tcpnorm. `http://www.wand.net.nz/~stj2/nsc/software.html`, Accessed 2006.

[152] M. Allman, S. Floyd, and C. Partridge. Increasing TCP's Initial Window. RFC3390, October 2002.

[153] Sally Floyd and Eddie Kohler. Internet research needs better models. *SIGCOMM Comput. Commun. Rev.*, 33(1):29–34, January 2003.

[154] Vern Paxson and Sally Floyd. Why we don't know how to simulate the internet. In *WSC '97: Proceedings of the 29th conference on Winter simulation*, pages 1037–1044, New York, NY, USA, 1997. ACM Press.

[155] K. G. Anagnostakis, M. B. Greenwald, and R. S. Ryger. On the sensitivity of network simulation to topology. In *MASCOTS '02: Proceedings of the 10th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS'02)*, Washington, DC, USA, 2002. IEEE Computer Society.

[156] Averill M. Law. Practical statistical analysis of simulation output data: the state of the art. In R. G. Ingalls, M. D. Rossetti, J. S. Smith, and B. A. Peters, editors, *Winter Simulation Conference*, pages 67–72. Winter Simulation Conference, December 2004.

[157] J. Bolot. Characterizing end-to-end packet delay and loss in the internet. *Journal of High-Speed Networks*, 2, 1993.

[158] Y. Zhang, V. Paxson, and S. Shenker. The stationarity of internet path properties: Routing, loss, and throughput. Technical report, ACIRI, 2000.

[159] T. V. Lakshman and Upamanyu Madhow. The performance of TCP/IP for networks with high bandwidth-delay products and random loss. *IEEE/ACM Trans. Netw.*, 5(3):336–350, June 1997.

[160] T. V. Lakshman, Upamanyu Madhow, and Bernhard Suter. TCP/IP performance with random loss and bidirectional congestion. *IEEE/ACM Trans. Netw.*, 8(5):541–555, October 2000.

[161] Jitendra Padhye, Victor Firoiu, Donald F. Towsley, and James F. Kurose. Modeling TCP Reno performance: a simple model and its empirical validation. *IEEE/ACM Trans. Netw.*, 8(2):133–145, April 2000.

[162] Sam Jansen and Anthony Mcgregor. Measured comparative performance of TCP stacks. In *Passive and Active Measurement Workshop*, volume 3431, pages 329–332, Boston, MA, USA, March 2005.

[163] Sam Jansen and Anthony McGregor. Simulation with real world network stacks. In *WSC '05: Proceedings of the 37th Winter Simulation Conference*, pages 2454–2463, Orlando, Florida, USA, December 2005. Society for Computer Simulation International.

[164] W. Stevens. TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms. RFC2001, January 1997.

[165] M. Allman, S. Floyd, and C. Partridge. Increasing TCP's Initial Window. RFC2414, September 1998.

[166] Sally Floyd. Metrics for the evaluation of congestion control mechanisms. Internet Draft, October 2005.

[167] Sally Floyd. Tools for the evaluation of simulation and testbed scenarios. Internet Draft, October 2005.

[168] David X. Wei, Pei Cao, and Steven H. Low. Time for a TCP benchmark suite? Technical report, Caltech, 2005.

[169] Pasi Sarolahti and Alexey Kuznetsov. Congestion control in linux TCP. In *Proceedings of the FREENIX Track: 2002 USENIX Annual Technical Conference*, pages 49–62, Berkeley, CA, USA, 2002. USENIX Association.

[170] A. Gurtov and R. Ludwig. Responding to spurious timeouts in TCP. In *INFOCOM 2003. Twenty-Second Annual Joint Conference of the IEEE Computer and Communications Societies. IEEE*, volume 3, pages 2312–2322, 2003.

[171] Ian H. Witten and Eibe Frank. *Data Mining: Practical Machine Learning Tools and Techniques, Second Edition*. Morgan Kaufmann, June 2005.

[172] Mark A. Hall and Geoffrey Holmes. Benchmarking attribute selection techniques for discrete class data mining. *IEEE Transactions on Knowledge and Data Engineering*, 15(6):1437–1447, November 2003.

[173] Mark A. Hall. Correlation-based feature selection for discrete and numeric class machine learning. In *ICML '00: Proceedings of the Seventeenth International Conference on Machine Learning*, pages 359–366, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[174] S. Floyd. HighSpeed TCP for Large Congestion Windows. RFC3649, December 2003.

[175] Yee-Ting Li, Douglas Leith, and Robert N. Shorten. Experimental evaluation of TCP protocols for high-speed networks. Technical report, Hamilton Institute, NUI Maynooth, 2005.

[176] Sangtae Ha, Yusung Kim, Long Le, Injong Rhee, and Lisong Xu. A step toward realistic performance evaluation of high-speed TCP variants. In *Fourth International Workshop on Protocols for Fast Long-Distance Networks*, 2006.

[177] Kazumi Kumazoe, Katsushi Kouyama, Yoshiaki Hori, Masato Tsuru, and Yuji Oie. Can high-speed transport protocols be deployed on the internet? : Evaluation through experiments on jgnii. In *Fourth International Workshop on Protocols for Fast Long-Distance Networks*, 2006.

[178] Sally Floyd. The transport modeling research group (tmrg). `http://www.icir.org/tmrg/`, Accessed 2006.

[179] Raj Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling.* John Wiley and Sons, 1991.

[180] M. Lulling and J. Vaughan. A simulation-based performance evaluation of Tahoe, Reno and Sack TCP as appropriate transport protocols for SIP. *Computer Communications*, 27(16):1585–1593, October 2004.

[181] V. Jacobson. Congestion avoidance and control. In *SIGCOMM '88: Symposium proceedings on Communications architectures and protocols*, volume 18, pages 314–329, New York, NY, USA, August 1988. ACM Press.

[182] M. Allman, V. Paxson, and W. Stevens. TCP Congestion Control . RFC2581, April 1999.

[183] Bellcore. LSGRR: Switching system generic requirements for call control using the integrated services digital network user part (isdnup). Technical report, GR-317-CORE, Bellcore, Morristown, New Jersey, December 1997.

[184] Rishi Sinha, Christos Papadopoulos, and John Heidemann. Internet packet size distributions: Some observations. `http://netweb.usc.edu/~rsinha/pkt-sizes/`, October 2005, Accessed 2006.

[185] Alberto Medina, Mark Allman, and Sally Floyd. Measuring the evolution of transport protocols in the internet. *SIGCOMM Comput. Commun. Rev.*, 35(2):37–52, April 2005.

[186] Doug Burger and David A. Wood. Accuracy vs. performance in parallel simulation of interconnection networks. In *IPPS '95: Proceedings of the 9th International Symposium on Parallel Processing*, pages 22–31, Washington, DC, USA, 1995. IEEE Computer Society.

[187] Susan J. Eggers. Simplicity versus accuracy in a model of cache coherency overhead. *IEEE Trans. Comput.*, 40(8):893–906, August 1991.

[188] John Levon. OProfile - a system profiler for Linux. `http://oprofile.sourceforge.net/`, Accessed 2006.

[189] Will Cohen. Multiple architecture characterization of the build process with oprofile. Submitted to Workshop on Workload Characterization 2003, 2003.

[190] Julien Seward, Nicholas Nethercote, Cerion Armour-Brown, Jeremy Fitzhardinge, Tom Hughes, Paul Mackerras, Dirk Mueller, and Robert Walsh. Valgrind. `http://valgrind.org`, Accessed 2006.

[191] Sam Jansen and Anthony McGregor. Performance, validation and testing with the network simulation cradle. In *MASCOTS '06: Proceedings of the 14th IEEE International Symposium on Modeling, Analysis, and Simulation*, pages 355–362, Monterey, California, USA, 2006. IEEE Computer Society.

[192] R. Brown. Calendar queues: a fast 0(1) priority queue implementation for the simulation event set problem. *Commun. ACM*, 31(10):1220–1227, October 1988.

[193] Kah L. Tan and Li-Jin Thng. Snoopy calendar queue. In *Proceedings of the 32nd conference on Winter simulation*, pages 487–495, San Diego, CA, USA, 2000. Society for Computer Simulation International.

[194] Jongsuk Ahn and Seunghyun Oh. Dynamic calendar queue. In *Proceedings of the Thirty-Second Annual Simulation Symposium*, Washington, DC, USA, 1999. IEEE Computer Society.

[195] Guanhua Yan and Stephan Eidenbenz. Sluggish calendar queues for network simulation. In *Modeling and Simulation of Computer and Telecommunication Systems*, pages 127–136, Monterey, CA, 2006. IEEE Computer Society.

[196] Sam Jansen. Heapprof heap profiling tool. `http://www.wand.net.nz/~stj2/nsc/software.html`, Accessed 2006.

[197] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM Press.

[198] Sam Jansen and Anthony McGregor. Static virtualization of C source code. *Software: Practice and Experience*, 38(4):397–416, April 2008.

[199] Sam Jansen. Network simulation cradle software. `http://research.wand.net.nz/software/nsc.php`, Accessed 2008.

[200] Quagga routing suite. `http://www.quagga.net/`, Accessed 2006.

[201] Bjrn Hggqvist. High quality video conferencing. Master's thesis, Lulea Technical University, 2005.

[202] J. Dike. A user-mode port of the linux kernel. In *Proceedings of the 4th Annual Linux Showcase and Conference*. USENIX, October 2000.

[203] R. Durst, G. Miller, and E. Travis. TCP extensions for space communications. In *Proceedings of the second annual international conference on Mobile computing and networking*, White Plains, NY USA, 1996.

[204] Scps reference software. `http://www.openchannelsoftware.com/projects/SCPS`, Accessed 2006.

[205] Sam Jansen and Anthony McGregor. Validation of simulated real world network stacks. In *Proceedings of the Winter Simulation Conference*, pages 2177–2186, Washington D.C., USA, December 2007. IEEE Press.

[206] Adam Biltcliffe, Michael Dales, Sam Jansen, Thomas Ridge, and Peter Sewell. Rigorous protocol design in practice: An optical packet-switch MAC in HOL. In *14th IEEE International Conference on Network Protocols (ICNP)*, pages 117–126, Santa Barbara, CA, USA, November 2006. IEEE Computer Society.

[207] Mark Apperley, Sam Jansen, Amos Jeffries, Masood Masoodian, Laurie McLeod, Lance Paine, Bill Rogers, Kirsten Thomson, and Tony Voyle. Lecture capture using large interactive display systems. In *ICCE '02: Proceedings of the International Conference on Computers in Education*, page 143, Auckland, New Zealand, 2002. IEEE Computer Society.

[208] Adam Dunkels, Leon Woestenberg, Kieran Mansley, and Jani Monoses. lwIP embedded TCP/IP stack. `http://savannah.nongnu.org/projects/lwip/`, Accessed 2006.

[209] Richard W. Stevens. *Unix Network Programming*. Prentice Hall PTR, January 1990.

[210] Chris D. Marlin. *Coroutines: A Programming Methodology, a Language Design and an Implementation*, volume 95 of *Lecture Notes in Computer Science*. Springer, 1980.

[211] Angela D. Orebaugh and Gilbert and Ramirez. *Ethereal Packet Sniffing*. Syngress, February 2004.

[212] Deborah Estrin, Mark Handley, John Heidemann, Steven Mccanne, Ya Xu, and Haobo Yu. Network visualization with the VINT network animator nam. Technical Report 99-703b, University of Southern California, 1999.

[213] Omnet++ community site. `http://www.omnetpp.org/`, Accessed 2006.

[214] Andras Varga. The OMNET++ discrete event simulation system. In *Proceedings of the European Simulation Multiconference*, pages 319–324, Prague, Czech Republic, June 2001. SCS – European Publishing House.

[215] A. Varga and G. Pongor. Flexible topology description language for simulation programs. In *Proceedings of the 9th European Simulation Symposium*, Passau, Germany, October 1997.

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

[217] Chris Savarese, Jan M. Rabaey, and Koen Langendoen. Robust positioning algorithms for distributed ad-hoc wireless sensor networks. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 317–327, Berkeley, CA, USA, 2002. USENIX Association.

[218] Tijs van Dam and Koen Langendoen. An adaptive energy-efficient MAC protocol for wireless sensor networks. In *SenSys ’03: Proceedings of the 1st international conference on Embedded networked sensor systems*, pages 171–180, New York, NY, USA, 2003. ACM Press.

[219] Koen Langendoen and Niels Reijers. Distributed localization in wireless sensor networks: a quantitative comparison. *Comput. Networks*, 43(4):499–518, November 2003.

[220] Mirco Musolesi, Stephen Hailes, and Cecilia Mascolo. Adaptive routing for intermittently connected mobile ad hoc networks. In *WOWMOM ’05: Proceedings of the Sixth IEEE International Symposium on a World of Wireless Mobile and Multimedia Networks (WoWMoM’05)*, pages 183–189, Washington, DC, USA, 2005. IEEE Computer Society.

[221] Adam Dunkels, Thiemo Voigt, Juan Alonso, and Hartmut Ritter. Distributed TCP caching for wireless sensor networks. In *Proceedings of the Third Annual Mediterranean Ad Hoc Networking Workshop (MedHocNet 2004)*, June 2004.

[222] Fu-Tai An, Kyeong S. Kim, D. Gutierrez, S. Yam, E. Hu, K. Shrikhande, and L. G. Kazovsky. SUCCESS: a next-generation hybrid WDM/TDM optical access network architecture. *Lightwave Technology, Journal of*, 22(11):2557–2569, 2004.

[223] K. Wehrle, J. Reber, and V. Kahmann. A simulation suite for internet nodes with the ability to integrate arbitrary quality of service behavior, 2001.

[224] Ulrich Kaage, Verena Kahmann, and Friedrich Jondral. An OMNET++ TCP model. In *Proceedings of the European Simulation Multiconference*, June 2001.

[225] Johnny Lai, Eric Wu, Andras Varga, Ahmet Y. Sekercioglu, and Gregory K. Egan. A simulation suite for accurate modeling of ipv6 protocols. In *Proceedings of the 2nd International OMNeT++ Workshop*, Berlin, Germany, January 2002.

[226] Ahmet Y. Sekercioglu, Andras Varga, and Gregory K. Egan. Parallel simulation made easy with OMNeT++. In *Proceedings of the European Simulation Symposium*, Delft, The Netherlands, October 2003.

[227] Omnet++ model documentation. `http://www.omnetpp.org/doc/INET/neddoc/index.html`, Accessed 2006.

[228] R. Braden. Requirements for Internet Hosts - Communication Layers. RFC1122, October 1989.

[229] *Scalable Simulation Framework API Reference Manual*, March 1999.

[230] Dartmouth ssf implementation. `http://www.crhc.uiuc.edu/~jasonliu/projects/ssf/intro.html`, Accessed 2006.

[231] Scalable simulation framework. `http://www.ssfnet.org/`, Accessed 2006.

[232] T. G. Griffin and B. J. Premore. An experimental analysis of BGP convergence time. In *Ninth International Conference on Network Protocols*, pages 53–61, 2001.

[233] David M. Nicol, Brian Premore, and Andy Ogielski. Using simulation to understand dynamic connectivity at the core of the internet. In *Proceedings of UKSim*, Cambridge University, England, April 2003.

[234] Michael Liljenstam and Andy T. Ogielski. Crossover scaling effects in aggregated TCP traffic with congestion losses. *SIGCOMM Comput. Commun. Rev.*, 32(5):89–100, November 2002.

[235] Michael Liljenstam, David M. Nicol, Vincent H. Berk, and Robert S. Gray. Simulating realistic network worm traffic for worm warning system design and testing. In *WORM '03: Proceedings of the 2003 ACM workshop on Rapid malcode*, pages 24–33, New York, NY, USA, 2003. ACM Press.

[236] James H. Cowie, David M. Nicol, and Andy T. Ogielski. Modeling the global internet. *Computing in Science and Engg.*, 1(1):42–50, January 1999.

[237] J. Cowie and H. Liu. Towards realistic million-node internet simulations. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, 1999.

[238] Jason Liu, Felipe L. Perrone, David M. Nicol, Michael Liljenstam, Chip Elliott, and David Pearson. Simulation modeling of large-scale ad-hoc sensor networks. In *European Simulation Interoperability Workshop*, 2001.

[239] David M. Nicol, Jason Liu, Michael Liljenstam, and Guanhua Yan. Simulation of large-scale networks using ssf. In *Proceedings of the 35th Winter Simulation Conference*, pages 650–657. Winter Simulation Conference, 2003.

[240] Michael G. Khankin. TCP/IP implementation within the dartmouth scalable simulation framework. Technical report, Dartmouth College, June 2001.

[241] D. M. Nicol. Discrete event fluid modeling of TCP. In *Simulation Conference, 2001. Proceedings of the Winter*, volume 2, pages 1291–1299 vol.2, 2001.

[242] David Nicol, Michael Goldsby, and Michael Johnson. Fluid-based simulation of communication networks using ssf. In *Proceedings of the European Simulation Symposium*, Erlangen-Nuremberg, Germany, October 1999.

[243] David M. Nicol and Guanhua Yan. Discrete event fluid modeling of background TCP traffic. *ACM Trans. Model. Comput. Simul.*, 14(3):211–250, July 2004.

[244] George F. Riley. The Georgia Tech network simulator. In *MoMeTools '03: Proceedings of the ACM SIGCOMM workshop on Models, methods and tools for reproducible network research*, pages 5–12, New York, NY, USA, 2003. ACM Press.

[245] R. M. Fujimoto, K. Perumalla, A. Park, H. Wu, M. H. Ammar, and G. F. Riley. Large-scale network simulation: how big? how fast? In *11th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer Telecommunications Systems*, pages 116–123, 2003.

[246] G. F. Riley. Large-scale network simulations with gtnets. In *Proceedings of the 2003 Winter Simulation Conference*, volume 1, pages 676–684 Vol.1, 2003.

[247] G. E. Riley, M. L. Sharif, and Wenke Lee. Simulating internet worms. In *The IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*, pages 268–274, 2004.

[248] Hung-Ying Tyan, A. Sobeih, and J. C. Hou. Towards composable and extensible network simulation. In *19th IEEE International Parallel and Distributed Processing Symposium*, pages 225a–225a, 2005.

[249] Hung-Ying Tyan. *Design, Realization and Evaluation of a Component-Based Compositional Software Architecture for Network Simulation*. PhD thesis, The Ohio State University, 2002.

[250] Ahmed Sobeih, Wei P. Chen, Jennifer C. Hou, Lu C. Kung, Ning Li, Hyuk Lim, Hung Y. Tyan, and Honghai Zhang. J-Sim: A simulation environment for wireless sensor networks. In *Annual Simulation Symposium*, pages 175–187, 2005.

[251] Rimon Barr, Zygmunt J. Haas, and Robbert van Renesse. Jist: an efficient approach to simulation using virtual machines: Research articles. *Softw. Pract. Exper.*, 35(6):539–576, May 2005.

[252] Zygmunt J. Haas and Rimon Barr. Density-independent, scalable search in ad hoc networks. In *Proceedings of IEEE International Symposium on Personal Indoor and Mobile Radio Communications*, September 2005.

[253] Kelwin Tamtoro. TCP implementation for SWANS. Technical report, Cornell University, January 2004.

[254] R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin. Resource ReSerVation Protocol (RSVP) – Version 1 Functional Specification. RFC2205, September 1997.

[255] A. Terzis, L. Wang, J. Ogawa, and L. Zhang. A two-tier resource management model for the internet. In *Global Telecommunications Conference*, volume 3, pages 1779–1791 vol.3, 1999.

[256] D. Pei, L. Wang, D. Massey, S. F. Wu, and L. Zhang. A study of packet delivery performance during routing convergence. In *International Conference on Dependable Systems and Networks*, pages 183–192, 2003.