Architecture and Trial Implementation of a Performance Testing Framework

Final Report for COMP420Y

Brendon Jones

Department of Computer Science

Hamilton, New Zealand

October 22, 2004

Abstract

The WAND Emulation Network is used by researchers to undertake performance evaluation of computer network software and hardware. While the network provides a powerful environment for testing, many of the procedures are ad hoc. Researchers must invest time in developing their own measurement system. Further, these systems are disjoint from one another, so results are not necessarily comparable.

This report details the development of a testing framework that will allow easy emulation setup and measurement. Amongst the more important of these components is the automatic configuration of the network and the generation of traffic to create situations that fully test the capabilities of a device.

Acknowledgements

I would like to acknowledge the members of the WAND group for their support during this project. Extra thanks should go to my supervisor Richard Nelson for all of his help throughout the year.

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Goals 2

2 Background **3**

2.1 Network Performance Metrics 3

2.1.1 Current Performance Metric Research 4

2.1.2 Current Performance Tools 4

2.2 Network Emulation 5

2.2.1 Emulab 5

2.2.2 WAND Emulation Network 5

2.3 Network Simulation 6

2.3.1 ns-2 6

3 Implementation **7**

3.1 Configuration Language 7

3.2 Language Choice 7

3.2.1 Parsing 8

3.2.2 Overriding `ns-2` Functions 9

3.3 Custom `ns-2` Agent 10

3.3.1 Emulation Interface 10

3.4 Emulation Daemon 11

3.4.1 Transport Protocol 12

3.4.2 Command Set 12

3.4.3 Application Testing 13

3.4.4 Configuration 14

3.4.5 Traffic Generation . . . 14
3.4.6 Security . . . 15
3.5 Network Configuration . . . 15
3.5.1 Hosts . . . 16
3.5.2 Topology . . . 16
3.5.3 NIST Net . . . 17
3.6 Disk Imaging . . . 19
3.6.1 Creating a Disk Image . . . 19
3.6.2 Boot Procedure . . . 20
3.6.3 Custom RAM Disk . . . 22

4 Results **23**
4.1 Accurate Configuration . . . 23
4.2 Performance Testing . . . 23

5 Conclusions and Future Work **26**

A Emulation Machine Specifications **28**

B Abridged User Guide **30**
B.1 Global Settings . . . 30
B.2 Creating and Configuring Nodes . . . 31
B.3 Creating and Configuring Links . . . 31
B.4 Creating and Configuring Agents . . . 32
B.5 Creating and Configuring Applications . . . 32
B.6 Final Points . . . 33

Bibliography **34**

List of Figures

3.1 Available Configuration Commands 9
3.2 Default ns-2 Structure 10
3.3 ns-2 Integrated with Emulation System 11
3.4 Emulation Daemon Protocol Specification 13
3.5 Example Topology Diagram Generated by `dot` 17
3.6 Simplified Boot Procedure 21

4.1 Competing UDP Flows 24

A.1 Machine Specifications 29
A.2 Overview of Emulation Network 29

Chapter 1

Introduction

Networking is a very active area of research. New protocols are always under development, and existing ones are being tweaked or added to as well. New devices are being created to put in networks to improve security, reliability or performance. Before these are deployed, it is important to understand how they will behave in a real network, and interact with existing protocols and devices. It is not sensible for these tests to take place in a production network because of the possibility of damage being caused or existing traffic being interrupted. Also, it is such a random environment that results are not always able to be trusted. Instead, what is needed is a closed system over which a researcher has full and total control, and in which they can conduct experiments.

1.1 Motivation

Network simulation is becoming increasingly prevalent as the way to test new ideas and research, and so it is important that we are able to accurately validate the results gained through it. In this case, simulation is the process of experimenting with the diverse attributes of networking in a software environment. The physical devices, computers, wiring, operating system and user behaviour are represented by models based on their specifications and observed behaviour. Situations that would otherwise be hypothetical or simply difficult to create can be explored, repeated and tweaked, with constraints on time and scale imposed only by the speed of the hardware the simulation is running on. Because these models are necessarily abstracted in nature to make their performance tractable, simulation cannot always accurately predict the subtle interactions that occur between different implementations of network stacks, operating systems and the various network devices along a path. Despite this, good simulators that are consistent are useful for comparative studies and are a valuable resource for improving the state of networking. Exactly how much it differs from the "reality" of a situation, and how consistent it is need to be known. Heidermann, Mills and Kumar[6] acknowledge that the "growing reliance on simulation raises the stakes with regard to establishing the correctness and predictive merits of specific simulation models", and that methods of validation need to be improved.

Emulation is one of the preferred methods for validating simulation (and to a lesser extent, testing in its own right), but despite this, tools for emulation lag behind in terms of quality and availability. It is perhaps understandable that this is so, because emulation brings together many different areas of computing in order to provide a closed, controllable environment constructed with real equipment. The one-to-one correlation between nodes used in emulation and the physical machines (rather than the one physical machine, many nodes approach simulation is capable of) means a sizable network of computers is required to be able to emulate a complex topology. All these need to be managed, configured and updated in order to keep them going, and there is the added issue of how testing is performed and how it is made to produce results. What is needed is a way to perform emulation in a manner similar to simulation, with the advantages of both.

1.2 Goals

The overall goal of this project is to create a system for users to interact with that removes the complexity of current network emulation. If the user is completely unable to spot any difference in the process when they conduct simulations and emulations and they can apply the same skill set to both, then this goal will have been realised and surpassed. More practically, this implementation will likely function as a precursor to such a system, and will identify key points that need to be understood while itself developing into a full featured but less streamlined interface to emulation.

The first step is taking care of configuring machines and lowering the cost of maintaining them. To do so it should remove the need for users to be involved with specific configuration, and instead allow them to focus on their experiments (while the system takes care of the details). This configuration should take place quickly, so that experiments can be rapidly modified to suit changing conditions, and so that it can be time shared between users with low overhead.

Secondly, experimentation needs to be controllable at a higher level than it is currently. At present the method tends to revolve around logging into each machine involved, starting and stopping test software and copying around results files. Doing so manually is time consuming and scripting it takes effort and is usually non portable between different situations. The system should provide alternatives to this so that testing can be simply a matter of the user specifying how and where to test, and all the work is taken care of behind the scenes. Whether it is possible to cover all options will need to be investigated, and a way to allow the more complex tests to be run found, should this be necessary.

Chapter 2

Background

Interest in the performance of computer networks has grown alongside the networks themselves, to the point where it is no longer a small part of network administration, but a fully fledged area of research. Modern networks, network devices and protocols are constantly being evaluated to determine how they interact with other parts of the network, and how they can be changed to work better. New ideas are not put into production until their impact is fully understood.

This chapter discusses some of the background work in network performance that this project builds on. In particular, the efforts of the Internet Engineering Task Force (IETF) to create standardised tests and measures, and the range of approaches that other groups have taken in order to understand how new systems will behave – more specifically, different ways that are commonly used to determine the state of a network, and methods of simulating and emulating networks. The ideas put forward by the IETF with respect to performance testing are important ones and should at least be considered as part of the process. Also, investigating how the current batch of simulators behave and how other network emulators have dealt with the problems they have encountered can give useful insights.

2.1 Network Performance Metrics

An essential part of the project is ensuring that it is possible to test as many metrics as possible. This means that it is important to understand what aspects of network performance are considered most important and useful, in addition to how the tests are commonly performed. Standardised metrics for measuring performance allow comparison across results obtained by different researchers. These should guide development of the whole performance testing framework being developed as part of this project, in such a way that the architecture is capable of supporting them.

Of interest here are the IP Performance Metrics[9] and Benchmarking Methodology[1] IETF Working Groups, who are responsible for publications detailing their recommendations for network measurement, and methodology for testing. Another source of useful data on metrics is the available toolset for performance

testing. Such software is written to fill a perceived need, and so its existence can show which aspects are of importance to the networking community as a whole. Often the results of commonly used software of this type develop into a de facto standard, or correspond with a more formal metric.

2.1.1 Current Performance Metric Research

Most of the relevant documentation available is dominated by the standards published by the IP Performance Metrics, and Benchmarking Methodology working groups, which are all aimed at developing metrics and tests that can give an accurate measure of how well devices within a network function, though each group has a different focus.

The IP Performance Metrics group is aimed at metrics that any user, privileged or not can perform. As such they are mostly measuring end to end performance from the current location in the network. Regular users are most interested in the characteristics of the path between themselves and the remote machine they are dealing with, and how these directly affect the performance they observe. This means that the tests are based around connectivity, throughput and latency along the path, and are all very simple to perform.

The Benchmarking Methodology Working Group is at the opposite end of the spectrum, being interested in intense testing of many aspects of both equipment and services within a closed laboratory environment, with a focus on network devices rather than end host or end to end performance. They explain in very precise terms their methods for testing classes of system, and what metrics are tested. In particular they have published documents that standardise terminology and describe techniques for testing such things as routers, firewalls, and switches.

One of the original plans for the project was to decide on which metrics were important, and to include as many of them as possible. This was not possible however, as the sheer number and variability of the different metrics precluded catering to all of them, and choosing just a subset was not desired. Instead, design moved towards making the system generic and flexible enough that any of the tests would be able to be performed.

2.1.2 Current Performance Tools

In order to understand their networks, administrators and users employ a wide variety of tools. Many of these have become standard programs that any operating system will include, and most are publicly available to anyone who wishes to make use of them. For example, it would be unusual to be without the `ping` and `traceroute` utilities that are useful for diagnosing network connectivity issues. Of perhaps more relevance to this project are lesser known tools such as `pathchar`[15] and `iperf`[10] that are used for exploring such characteristics of a link as bandwidth and latency. These tend to focus on user observable behaviour of network paths rather than the specifics of what what is happening internally at each hop – which makes sense in that the tools are designed to be used in a live network, where user access is not normally given to the

internals of routers.

Currently available tools can be useful in the design of the project in a couple of ways. They can be used as a source of data to load the network and create cross traffic during experiments, while measurements are taken elsewhere. Alternatively, they can be used to make measurements while different circumstances are experimented with.

2.2 Network Emulation

As mentioned, network emulation is one approach to providing an environment for testing. It involves using real machines to represent the problem space, and using them to recreate situations that could be found in any live network. This means tests can be run with accurate implementations of protocols and timers for high confidence in the results. The cost it takes to set up and run tests are the major disadvantages to emulation.

This section discusses the approach taken previously by large emulation projects, as well as detailing the original network upon which this project is built.

2.2.1 Emulab

Emulab[21] is one of the largest emulation projects currently underway, involving more than 150 PCs, and is made available for research use. The motivation behind Emulab is to provide an easy to use system that takes care of configuring a network topology for a researcher to use as a platform for conducting their own experiments. Simple scripts are used to describe nodes and the links between them, and then the system installs the machines afresh, and configures them to use the new topology. Researchers are then free to run their own tests and collect their own measurements on a private network.

The Emulab project is a successful one in that it does provide everything it claims too, and is well used. However, it only goes part of the way towards fully integrating simulation and emulation. It is merely a configuration tool, and it is up to the individual to create their own traffic models. This means that when the results of tests run on Emulab are compared with those of tests run in a simulator, there are two major differences instead of one – the traffic models as well as the underlying network (real or simulated) are likely to be different. If the same model could be used for both this would help remove and inconsistency and make results more comparable.

2.2.2 WAND Emulation Network

The WAND Emulation Network[20] is a network of 24 PCs around a central server. With multiple interfaces per machine, it allows for flexible topology configuration, making it useful as a test-bed for any sort of network research. The problem with it is that it is almost completely manually configured. Machines need to be

installed and configured separately, and conducting experiments involves manually logging in and running commands, unless time is spent on developing scripts to help automate the process for a particular set of tests. This is less than desirable, as it takes a researchers focus away from the purpose of their experiment, and forces them to deal with the logistics of it instead. The configuration framework this report describes replaces this with automated systems that remove the need for detailed and specific knowledge about network management.

Also available in the Emulation Network are a number of DAG[4] 3.5E network capture cards. They are passive devices that are transparent to the rest of the network, and record all packets that pass through them. Their clocks are synchronised to a GPS pulse, and so are capable of timestamping packets with an accuracy in the order of 100 nanoseconds. These can be inserted into any point of the network to get highly accurate records of network traffic.

Appendix A discusses the physical details of the hardware used within the WAND Emulation Network in greater detail.

2.3 Network Simulation

The other approach to testing is through simulation. More common than emulation because it tends to be easier, faster and in many cases free, simulation represents the problem space entirely in software. This often increases the speed at which it can be run, and makes for great scalability. These points offset the slight inaccuracy that tends to be present due to the abstractions of network protocols and interactions they use.

2.3.1 ns-2

`ns-2`[19] is a free and open source discrete event simulator designed for network research. Under constant development for many years, it is feature-rich, and contains implementations of most protocols that are commonly used. Because of its usefulness as a platform to develop protocols in, there also exist third party modules that provide support for most protocols that have been produced and experimented with in the last few years. The ease with which it can be extended like this by anyone is one of its strong points – the whole backend is written in extremely modular C++ for speed of execution, with the front end written in in OTcl[12] for speed of modification.

Chapter 3

Implementation

The construction of the configuration and testing framework went through a number of distinct processes. At a very high level, this involved a stage where machines are prepared and put into a ready state, followed by setting up and actually performing the tests, though each of these is itself more complex than a single step.

Initially work was done on developing a language to describe the network design and how testing should be done before deciding on leveraging that of the `ns-2` simulator, this is detailed in Section 3.1. Once that had been resolved, it was possible to start work on the architecture that would allow the configuration to be put into action. The custom `ns-2` agent that acts as a gateway between scripts and the live network is covered in Section 3.3, followed by a description of the software that runs on the network hosts to perform these commands in Section 3.4. Specifics of the configuration process are described next in Section 3.5. Section 3.6 contains a detailed description of the low level work required to have machines running in a state that allows all the rest of the configuration to be possible.

3.1 Configuration Language

There needs to be a way for users to specify what sort of network they would like created, how hosts communicate, and what sorts of tests should be performed, and this has to be on a more abstract level than if they were to manually configure it themselves in order to have value. The process of selecting a language and its format is covered in Section 3.2, followed by the initial approach taken to integrate it in Section 3.2.1. The change in approach that led to the accepted solution is then in Section 3.2.2.

3.2 Language Choice

The first step was to decide on the format of the language, and how users would communicate their ideas through it. This could be done in a number of ways such as through a configuration script, command line parameters to some program, or interactively. It was decided to use a configuration script for this purpose.

Using a configuration script provides a way to see historical emulations by being persistent even when configuration is not taking place. Both interactive and command line options are transient unless saved elsewhere, though that is not greatly useful unless the settings can be reloaded and reused – in which case they essentially become a configuration script.

Writing configuration files in an XML based language was a possibility in order to make it portable and easy to understand, but this was decided against. Rather than write a new language, employing one that is already in use in a similar domain allows for transference of the skills, knowledge and tools. Because of this, the configuration language is based directly on that of the `ns-2` network simulator. Its popularity and widespread use made it an obvious choice, and it had been seen to be successful when used for the same purpose in the Emulab project. Emulab have greatly expanded on the language vocabulary in order to allow many aspects of their network to be separately configured, but a conscious decision was made to use it here without modification except in the case where not doing so would harm functionality.

If most `ns-2` scripts can be run without major changes, then this makes available a large body of prior research that can be easily validated in an emulation environment. OTcl is the language of `ns-2` configuration scripts, and is now the language of emulation configuration scripts, which helps to lessen the distinction between the two somewhat.

3.2.1 Parsing

Example `ns-2` scripts from the website and tutorial portrayed them to be straightforward and linear configuration files. Essentially, these were lists of commands and their arguments. This made it appear possible that the information could be harvested using a simple parser, and one was implemented using the standard Unix tools Flex[3] and Bison[2]. Rules were built up to cover all the commands that provided information useful to setting up the emulation system, and to extract it into a usable form. This reached the stage where it understood a basic set of commands, and could hold a global view of the network described, knowing what machines existed and how they were connected. Development using this approach quickly stopped however when further investigation into example `ns-2` scripts showed something that was not obvious before – it is very common to make full use of the language features that OTcl provides. This means that real scripts often contain flow control structures such as *if* and *while* statements, and use variables extensively. The simple parser was designed to handle scripts that were simply attribute value pairs and would not work with anything more complex. Attempting to extend the parser was dropped because the task was too large, especially in the time frame that is available, and so other options needed to be looked at.

Command	Purpose	Notes
node	Create a new node	Extra parameter added
modifyImage	Specify the disk image to use	New command
simplex-link	Create a one way link between nodes	Interface unchanged
duplex-link	Create a two way link between nodes	Interface unchanged
lossmodel	Add a loss model to a link between nodes	Interface unchanged
All ns-2 agents	Responsible for packet transport	Four new additions, interface unchanged
All ns-2 applications	Responsible for traffic generation	Interface unchanged
attach-agent	Adds an agent to a node / application to an agent	Interface unchanged
connect	Links two agents together	Interface unchanged

Figure 3.1: Available Configuration Commands

3.2.2 Overriding ns-2 Functions

Because of the possibility that variables will contain useful information, and the various flow of control options, the next approach was to get inside ns-2 and use the information that it had available. Here the strength of using an open source project was seen as it was a simple matter to begin replacing key internal functions with new code to discover how they worked. Tracing through the execution of the code and displaying debug output meant that the execution path could be marked for each function call, and thus which functions needed changing in order to incorporate the new system. This was explored in a top down manner, beginning with the obvious functions that are called within all ns-2 scripts, and exploring the functions they called to understand their interactions with other parts of the system. It turns out that most of the top level functions follow the principle of only performing a single task, and so were easy to trace and replace with new code. An important point in favour of this approach is that it still essentially matched commands and caused code to run – instead of the parser making the match it was now being done by the Tcl interpreter. This meant that all the configuration code was still useful, and only needed bindings added so that it could be called from within Tcl, saving much time that would have been wasted re-implementing it all.

In order to work with older ns-2 scripts, it is vital that there are as few changes away from the original specification as possible, preferably none. All the modified functions have been made to work with exactly the same number and type of arguments as the originals, except in a single case where change was necessary to fit in with the physical system. To access configuration options that are not normally a part of ns-2, but are required to set up extra features of the Emulation Network, a number of extra commands were added to the set. These new commands and the ways in which the interface to existing ones have been modified can be seen in Figure 3.1.

3.3 Custom ns-2 Agent

In order to allow ns-2 to control the traffic model, it needs a way to communicate with the traffic generation process running on the emulation nodes. The traffic models normally interact directly with each transport agent running within ns-2, and so in order to have them interact with something external, a custom agent had to be created. Although there are already options available for having ns-2 act as an emulator and communicate with other machines (and act as a network of arbitrary size, topology and speed while doing so), this functionality is incomplete and progress on this appears to have halted.

There are four agents that are involved in this system, though they are all very similar. Two each have been created for the common protocols in the IP suite, the User Datagram Protocol (UDP), and the Transmission Control Protocol (TCP). These are described in Section 3.3.1, along with their output and interactions with the Emulation Network. They link very closely with a software daemon running on each emulation machine, which is described in Section 3.4.

3.3.1 Emulation Interface

These new agents were created as drop in replacements for the existing agents in ns-2 that would work with the emulation network instead of with the rest of the simulator. The standard UDP agent was used as the base step from which to build the UDPEmu and UDPEmuSink agents. Instead of passing messages to other sections of ns-2, they are now responsible for communicating configuration to the process running on each emulation node.

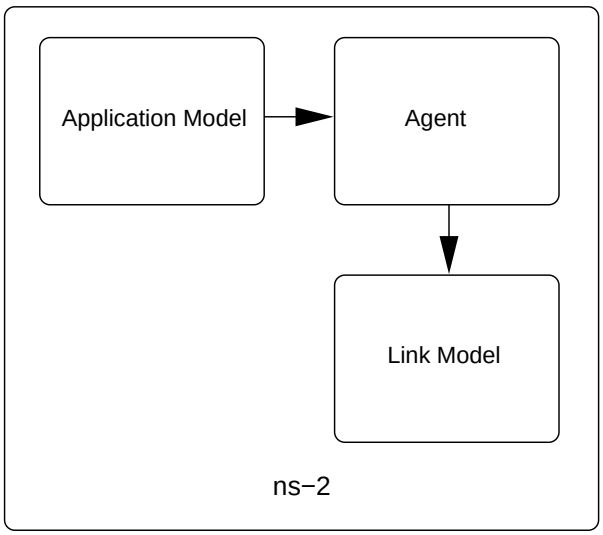

Figure 3.2: Default ns-2 Structure

Figure 3.2 shows the way traffic is normally handled inside of ns-2, and where the agents fit into it. Each element in the diagram is software. The generation of packets uses one of a variety of prebuilt functions

that attempts to mimic the distributions of packet sizes and rates that common networking protocols tend to exhibit, and this is passed on to the agent for a simulated transfer. Using an agent that exchanges information with a real network removes the need for the simulation aspect as seen in Figure 3.3.

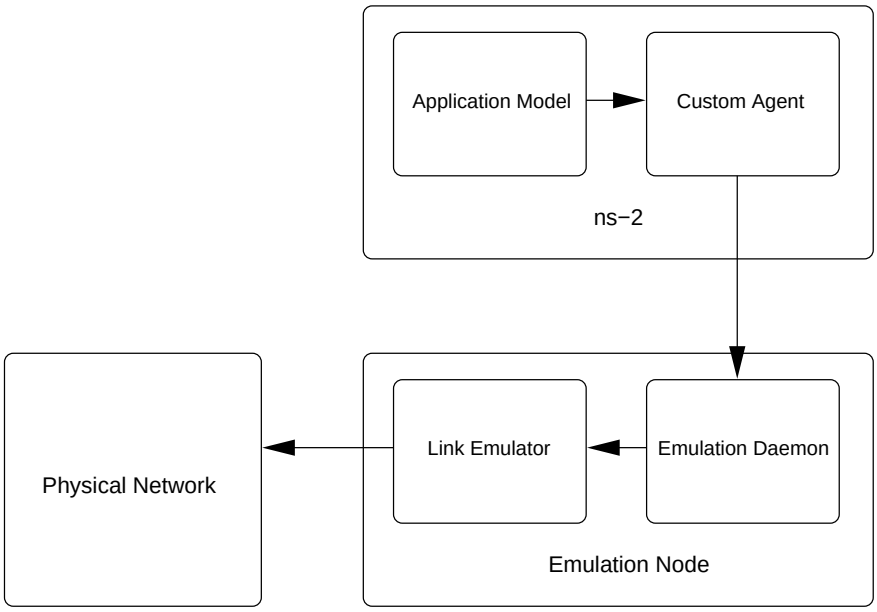

Figure 3.3: ns-2 Integrated with Emulation System

The idea behind these new agents is to give the emulation nodes access to the traffic generation models within `ns-2`. To do this, it needs to communicate to the machines in the network information about when to send packets, who to, and how big they should be. When the agent is initialised it opens a control connection to the host to which the agent belongs (as specified in the configuration script). Through commands described in Section 3.4, the two end points for each connection for experimental data are made aware of each other. Information such as user configured packet sizes are also sent.

Apart from the configuration options they now send, the UDP and TCP traffic sources are essentially the same as their original `ns-2` counterparts, but with most of the functionality stripped out. There is no need for them to fragment traffic or implement congestion control algorithms as this is all taken care of by the host operating systems. Instead, the agents act as a relay between the `ns-2` model and the emulation network – when the model tells them to send a packet, the pass the appropriate command through the network to the emulation nodes who do so.

3.4 Emulation Daemon

The other half of the control system with the `ns-2` agents is a daemon running on each of the emulation nodes, whose purpose is to receive and act on control data, generate traffic, and execute commands. This program is the key to everything that happens on the machines, and as such it is important that it will work on all of

them. Currently it has been tested on all operating systems for which there are images, and it works on all of them – Linux, FreeBSD and OpenBSD. Section describes the early decisions made regarding the transport protocol to use for control data. Next, Section 3.4.2 gives an outline of the commands that are recognised, followed by a number of issues encountered implementing them in Section 3.4.3. How these commands relate to configuration and traffic generation within the network is covered in Sections 3.4.4 and 3.4.5 respectively. A brief discussion of the security implications is in Section 3.4.6.

3.4.1 Transport Protocol

In order to perform its duties, the Emulation Daemon must communicate with a control source, and any users. The daemon uses TCP to receive information. In order to lower delay this was initially planned to use the more lightweight but unreliable UDP, as it was reasoned that the quality of the link would prevent the possibility of any packets being lost. However, the final decision was made to use TCP. This was for a number of reasons – firstly reliability is a big issue, as should a control packet setting up one end of a connection be dropped, then the whole test will completely fail. In most cases for anything useful to come of sending packets in an experiment, both ends of a connection need to be aware. Secondly, the delay in sending TCP packets can be minimised by setting the `TCP_NODELAY` socket option to disable the Nagle Algorithm[13], and issues of TCP being in slow start are mostly resolved by initial control messages that are not time critical being used to open congestion windows. This means that the only advantage of UDP has been matched by TCP in this situation, and because of the extra advantages TCP gives in terms of reliability, TCP is the best choice.

3.4.2 Command Set

Each command is a single line of ASCII characters, starting with an identifier and followed by a variable number of arguments, each argument contained within quotation marks. A line ends with a line feed character. A plain text protocol was chosen over a binary one as it is much easier to debug, and building code to match strings is generally more self documenting than code written to match binary formats. All the possible commands can be seen in Figure 3.4, and are described below.

The `EXEC` command is designed to run specified commands that are a part of the host operating system. It takes as an argument a string containing a command line, which is then passed to and run by a shell in a new process created with the `fork` system call. Any output from the command is sent back to the source – this is discarded in the case of the `ns-2` agent controlling it, or possibly saved for later use if it is human controlled. The idea behind this is that it allows for configuration of the host without the hassle of login names, passwords, and the appropriate programs being present to allow access, as well as allowing users to control emulation nodes from the central server.

The commands `SETUDP` and `UDPSINK` are used to configure the Emulation Daemon as the source and

<Command> ⟶ <Execute Command> | <Set UDP Target> | <Set UDP Sink> | <Set TCP Target> | <Set TCP Sink>
<Execute Command> ⟶ `EXEC "`<Command String>`"`<LF>
<Command String> ⟶ Any printable string
<Set UDP Target> ⟶ `SETUDP "` <Address>`" "` <Port>`" "` <Packetsize>`"` <LF>
<Set UDP Sink> ⟶ `UDPSINK "` <Port>`"` <LF>
<Set TCP Target> ⟶ `SETTCP "` <Address>`" "` <Port>`" "` <Packetsize>`"` <LF>
<Set TCP Sink> ⟶ `TCPSINK "` <Port>`"` <LF>
<Port> ⟶ <Digit>*
<Packetsize> ⟶ <Digit>*
<Address> ⟶ 1*3<Digit> . 1*3<Digit> . 1*3<Digit> . 1*3<Digit>
<Digit> ⟶ `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `9`
<LF> ⟶ `0x0A`

Figure 3.4: Emulation Daemon Protocol Specification

sink ends of a UDP connection respectively. As a sink, the daemon only needs to know which UDP port to listen on for a connection, but as a source of traffic it needs to know the destination address, port and packet size that should be used.

`SETTCP` and `TCPSINK` follow the same reasoning as the equivalent UDP commands, but use the TCP protocol. TCP is a connection oriented protocol, but making the initial connection is deferred until data is ready to be sent – receiving the `SETTCP` message simply configures and prepares the daemon for the connection, without actually making it.

3.4.3 Application Testing

During its development, the standard `telnet` program was used to act as a source of control data. This, combined with the textual nature of the communications protocol had the advantage of simplifying the testing process – commands could be entered arbitrarily and the output immediately observed. Because `telnet` ends every line sent with a carriage-return and line feed, this was initially adopted as the standard ending to a command. Working directly with input from `telnet` gave the option of leaving it as a method for user configuration and execution of commands.

When it came time to integrate the Emulation Daemon with current users of the Emulation Network, a design flaw became apparent that was missed during (or perhaps because of) the method of testing. Communicating with the daemon independently of the `ns-2` agent worked well when tested interactively in `telnet`, but would fail every time it was used as part of an automated process. A scripted `telnet` session that was fed data from a pipe would connect and appear to transfer everything successfully, but it would then terminate as the daemon replied with the output of the command. Studying logs of network traffic and debug output from the daemon confirmed that the connection and transfer were completed, and that the clients `telnet` session

was terminating early. The command lines used in the tests that were failing were of the form:

```
echo EXEC \"ping -c 1 localhost\" | telnet localhost 5000
```

This redirects the output of the `echo` command to the connection opened by `telnet`, and is the simplest case of what is necessary to properly automate tests through the use of scripts. For some reason, `telnet` was behaving differently when it was used as part of a pipe, compared to when it was used interactively. Looking closer at the network traffic showed that it was `telnet` closing the connection, not the Emulation Daemon, and that it closed it as soon as it had sent all its data. This was the problem - it had done what was expected of it, and terminated. The solution here was very simple once the problem was known. Replacing the `telnet` utility with `netcat`, which would hold the network connection open after sending all its data and only close when the remote end did, worked. The only change required to the original code was to change the requirement for lines to end in a carriage-return and line feed to simply a line feed.

3.4.4 Configuration

The first function the Emulation Daemon performs is configuration. It is available from machine startup, it is able to run commands on the host system, and so is a good choice for setting up each machine. Other possibilities exist for doing the same thing, but almost all require some sort of authentication and/or transferring of files, which takes time and adds complexity. Once the disk imaging process is complete, and determines that the hosts have successfully booted, the already generated configuration information is sent over the network to the Emulation Daemon. This is already in the form of shell commands, and all that needs to be done is for them to be run on the appropriate machine. This is done by sending them to the daemon as part of an `EXEC` command, which is treated no differently to any other use of the command.

3.4.5 Traffic Generation

Once the network is configured and working, a researcher could use it as an environment in which to perform their tests, but the Emulation Daemon has facilities for taking care of this also – in conjunction with the new `ns-2` agents, each daemon is capable of acting as both a source and a sink for network traffic simultaneously. This was not the case originally, instead it was planned to use an already developed traffic generator called (C)RUDE[17]. Free, open source and with very clear code, but with the failing being that it only provided UDP support, and added to the number of separate programs that needed to be running. Once it became apparent that the Emulation Daemon would be needed for configuration and would be running anyway, and after investigating other emulation facilities this changed. Rather than spend time updating the code of (C)RUDE to work with TCP and working on models for traffic generation, the Emulation Daemon was updated to communicate directly with `ns-2`.

A node is configured by to act as a sink when it receives a `UDPSINK` or `TCPSINK` command from an `ns-2` agent during its initialisation. The immediate action this causes the daemon to take is to begin listening on the port assigned to it in the received message. This functionality is very simple, and only requires the Emulation Daemon to receive packets and output cumulative statistics. Every packet received prints a line containing a timestamp and the current total number of bytes received. There is no difference between the UDP and TCP sinks. When in source mode because of receiving a `SETUDP` or `SETTCP` command, the daemon listens for control data from its attached `ns-2` agent that describe when packets should be sent. As a source of network traffic, the daemon is reliant on the information it receives from one of the `ns-2` agents discussed in Section 3.3. If the traffic models of `ns-2` are not appropriate, the user is able to use the Emulation Daemons `EXEC` functionality to run their own traffic generation or testing software.

3.4.6 Security

The issue of security is an important one, especially when dealing with network related software that, should it be exploited could allow attackers access to machines or services. This is such a large and overwhelming field that it was determined best to place aside and ignore completely. The Emulation Daemon has no authentication mechanisms and allows any connections, it runs with root privileges and will run any program it is told to run. As security is not taken into account at all, this development is completely unfettered by the restrictions being secure would impose – no concessions or compromises have had to be made in the way the system operates and so it is a powerful tool for machine configuration and testing. Having to enter passwords or deal with keys would partially defeat the purpose of the daemon, and using a Secure Shell connection would have been more practical. If security had been an issue, then that would have grown to fill the available time frame.

In this circumstance however, the security of the software is not as vital as it might seem, despite giving remote root access to every machine in the network. The WAND Emulation Network is a private network that is inaccessible from the global Internet. There is only a single machine from which users may interact with the network, and that is the central server. The server is already protected by appropriate security precautions, and user accounts are only issued to people who require them. All of these users are part of a cooperative academic environment, and are trusted to have no malicious intent. Should this not be the case, it is a simple matter to revoke their access and re-image any compromised machines.

3.5 Network Configuration

When the modified `ns-2` examines its configuration file there is a lot of information that needs to be stored for later use. Much of the configuration relies on having full knowledge of the network, and so cannot take place until the entire script has been read. At the end of the script when all this information is complete, it is

saved to disk while the machines are imaged. Once they have booted, this configuration is made live.

3.5.1 Hosts

Hosts on the network have a number of features – machine number, IP address (of each interface), operating system, routing, hostname and user accounts to name some of the more visible ones. All of these need to be configured in order to have a working machine. However, every one of these is set up to be automatically configured within the disk image itself, or inferred from other commands in the script. The only two commands that directly refer to host configuration are the node creation and node image selection ones.

The `node` command has been slightly modified to include as a parameter the number of the physical machine that should represent the node. This change is important because of the limited number of hosts available on the Emulation Network, and because there are two distinct types of host (refer Appendix A for details). In order to efficiently make use of these resources and not disadvantage other users, the machine allocation scheme is manual – an automated way to determine the best physical machine to use for each node representation would be a useful extension to the system. The second host configuration command `modifyImage` is a wholly new command that was created in order to specify the disk image a machine should load. This defaults to a standard Debian Linux install running a 2.4.20 kernel, but can be changed to any of the available disk images. Changing any of these options affects the processes that perform the disk imaging of nodes.

Those options that are unique per machine and isolated from outside influences are set as soon as enough information is available. Hostnames are created based on the number of the physical machine, and others such as user accounts are already a part of the disk image itself. The rest of the changeable host configuration, in particular addressing and routing is decided upon later based on what links are created between nodes. The node information structure within the system is created blank, and values are filled in as other options are set.

3.5.2 Topology

Once the nodes have been added, they need to be linked together in order to be able to send data. This is a two part process firstly involving configuring the hosts in order to use these links, and secondly, physically changing the network to create these links.

Each `simplex-link` and `duplex-link` in the configuration file creates an entry in an adjacency matrix. Every link is equally weighted on a boolean scale that simply determines if a link exists between two machines or not. Once all desired links have been created and this matrix is completed, an All Pairs Shortest Path algorithm is run over it in order to determine the shortest route from each pair of machines. This information is used to statically configure the routing tables of emulation machines. It is possible that a dynamic routing protocol such as OSPF could be used to do this, but it introduces extra traffic onto the

network, which is not always desirable.

The next step is to connect the machines together physically with cabling. Emulab has every single machine always connected to each other, and uses Virtual Private Networks to isolate the different test networks. This requires enough switch ports to cater for every interface the machines have and high bandwidth between switches if multiple ones are used, both of which have high monetary costs associated with them. This project uses a set of passive patch panels to which are routed cables coming from the non-control interfaces of machines in the network. Connections can be made between machines simply by running a crossover cable between them, or various devices such as hubs, switches and routers can be inserted at this point to change the topology. This needs to be done manually any time there is a routing change, but this is acceptable in the scheme of things as it takes place in another room outside of the server room, where there are less issues with granting physical access. To aid in changing cables and for reference during testing, a diagram of the network topology is created using the tool `dot`[5] from a dump of the adjacency matrix and interface addresses.

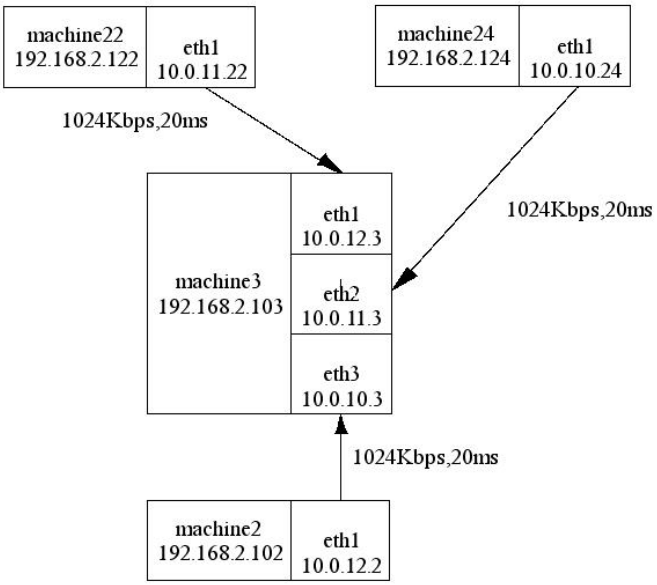

Figure 3.5: Example Topology Diagram Generated by `dot`

Currently under construction is an electronic patch panel that will allow connections to be made internally without the need to physically move cables. This means that the system itself will be able to assume responsibility for the correct machines being connected, and will do so much faster than any manual method.

3.5.3 NIST Net

NIST Net[14] is a tool for artificially adding latency and loss to a network, or constraining bandwidth. This is useful because the emulation network has only 100 megabit per second Ethernet links, but often it is needed to emulate links that are slower and of poorer quality. Rather than use physically different networking technolo-

gies in an attempt to cater for all possible circumstances, a more flexible solution is to use software to recreate the attributes of a certain link type. NIST Net allows each link to be individually configured with different parameters for bandwidth, delay, and chance of being dropped or duplicated, meaning any combination of link quality and speed can be created up to the actual maximum speed of the real links.

NIST Net is easy to install and run, and performs its intended task well. Before deciding to use it, tests were run to check the accuracy of any imposed limits and it was found to be close enough for most practical purposes. When used to introduce latency, the mean observed delay was approximately 0.2ms greater than the expected value, and deviated at most 0.1ms in each direction. Latency is often measured with a granularity at the millisecond level, and so this difference is so small that it is not an important factor. Its bandwidth limiting aspect was also tested and also found to be satisfactory. With low limits, the observed and expected possible bandwidth were almost identical, (the observed value was very slightly less). As limits rise the observed and expected values meet at about the 1 megabit mark, after which the observed value is slightly higher than it should be. At its highest point, this difference is still less than 2%, and if it is an issue it can be worked around by setting values slightly lower.

Using NIST Net is not without problems however. Because of the way that it intercepts incoming packets, it causes some strange interactions with other programs that are trying to use the packets at the same time. As part of the evaluation process of NIST Net, packets sent as part of a bulk transfer were observed with `tcpdump`[18]. Every incoming packet was duplicated in the output from `tcpdump`, as if the sending machine had sent everything twice. Packets observed leaving a machine were unaffected. Unfortunately no solution was found for this other than two work arounds – edit trace files to remove the duplicate packets, or perform captures on the receiving host of any transfer.

Another problem encountered with NIST Net is that on occasion the scheduler breaks down, and packets are no longer allowed through it as they should be. No cause was found for this either, though it only ever happened after long periods of use. Methods are included within NIST Net to restart the clock and to flush out packets, which helps to correct the problem when it occurs, but the fact that it happens at all is a rather large negative.

NIST Net only works with Linux kernel versions in the 2.0.xx, 2.2.xxx and 2.4.xx ranges, which limits its use to machines running Linux with a somewhat outdated kernel. This is because it is a kernel module, and is written to work within a very specific environment. This problem, and those discussed above make it necessary that at some time it either be fixed, or replaced. For the purposes of showing how this whole system could be made to work it is suitable, but alternatives will need to be found to enable it to work successfully with other operating systems, and into the future with newer kernel versions. Under FreeBSD, Dummynet[16] is the most commonly used tool for modifying link characteristics and works with firewall rules to select packets and delay/limit them. It has been a part of FreeBSD since 1998, and should continue to be a part of it for

the foreseeable future. Linux kernels in the new 2.6 series are starting to contain more possibilities for traffic shaping and should also be investigated as a possible replacement for NIST Net running on old kernels. If all other options fail, extra nodes running an operating system that works with this software can be inserted into links specifically to modify links.

3.6 Disk Imaging

Disk imaging involves the creation of a complete copy of a hard drive which can later be used to overwrite the contents of another drive. If the copy was originally made of a fully configured and working operating system, then the recipient now also has a fully working operating system installed. A library of disk images can be created so that any machine could have one of a variety of operating systems deployed on it easily. This is useful for a couple of reasons. Firstly it makes keeping large numbers of machines up to date much easier – changes can be made to a single machine of which an image is then made and sent out to all the others. This takes a fraction of the time it would take to replicate the changes manually and eliminates the need to individually perform installs and updates. In terms of its use here, disk imaging is a way to ensure that identical software configurations can be run, and that installing machines takes a minimum of time and effort. The process of copying images around can often be done remotely, and in the case of this project it *must* be possible to perform without having physical access as the machines are in a locked server room. Granting access to this room with many other critical machines is not taken lightly, and so other ways have to be found to create and install images.

Secondly, it is important that the environment be in a known state before performing any experiments. If previous tests have modified system settings, and someone is unaware of this, there can be little validity to any results they get. Re-imaging all machines involved beforehand removes this uncertainty. Changing between completely different environments is also easily achieved through the use of disk imaging.

Writing a disk imaging program is no small task, and so it was decided to make use of an existing application in order to save development time. This meant it was important that any tool selected be free and open source so that it could be modified to be more suitable for the specific needs of the project. Of the open source options, Frisbee[7] (created to work within Emulab, but also freely available to anyone who wishes to use it) looked to do what was needed best. It uses multicast for distribution of disk images which is fast and efficient, is generally well documented.

3.6.1 Creating a Disk Image

The `imagezip` program within Frisbee is used for the creation of compressed disk images. It understands the format of NTFS, EXT2 and FFS filesystems (amongst others), and so is able to selectively copy only the

sectors of a disk/partition that contain data, which it then compresses. Should a filesystem be of an unknown type, it is still capable of making a "raw" copy in which everything is copied verbatim, including areas of the disk that are unused. On the nodes in the WAND Emulation Network, `imagezip` takes in the order of 10 minutes to copy and compress a basic install of Debian Linux, FreeBSD or OpenBSD, which are the three operating systems currently in the disk image library.

Before this can be done however, there needs to exist a base install that will work in the environment of the emulation network, and contain enough tools such that it is useful as soon as it is installed. The first step towards creating an image is to have an understanding of how it will be used and thus what it will need to have installed in order to do so. Using Debian Linux as an example, the very first thing to be configured was the package management system `apt` so that it would be able to retrieve packages from the local server rather than more official sources (as access to the Internet is blocked). With this in place, any shortcomings in the toolset can be corrected at the users convenience. The small programs that are often taken for granted but are necessary to actually use a machine such as text editors and common network tools were also installed, and standard user accounts created. The most important changes made were networking related – each machine is configured to use the Dynamic Host Configuration Protocol (DHCP), and to synchronise clocks with the server using the Network Time Protocol (NTP). Having each clock in the network set to the same time means that timestamped events can be compared across machines, which is likely to be a common occurrence when testing.

It is important that the harddisk of the machine being imaged is not written to at any point during the process, otherwise inconsistencies can occur between the the original machine and the image, as well as cause errors within the filesystem structure of the image. The first attempt at imaging discovered this, as every time the image was reloaded, the machine spent a while repairing disk structures. The next approach involved the use of the Linux distribution Knoppix[11] on a live CD to create a working system that did not rely on disk access – it was reasoned that imaging is not a common activity, and so the physical access required to use a live CD could perhaps be allowed. It soon became apparent that images were going to be created more frequently than first thought, and the process needed to be completely remote. Further investigation showed that the Unix command `mount(8)` could be used to selectively change the way that filesystems were mounted, and could force them into a read-only state. With this, the contents of the harddisk can be frozen and made unchangeable while the system is still running, and `imagezip` can be run through a remote login, passing the disk image data back through the network. The read-only state is not permanent however, as the system files that determine how devices are mounted on boot is unchanged, only the current, running configuration.

3.6.2 Boot Procedure

As well as creating images remotely, it is just as important that they can be installed remotely as well.

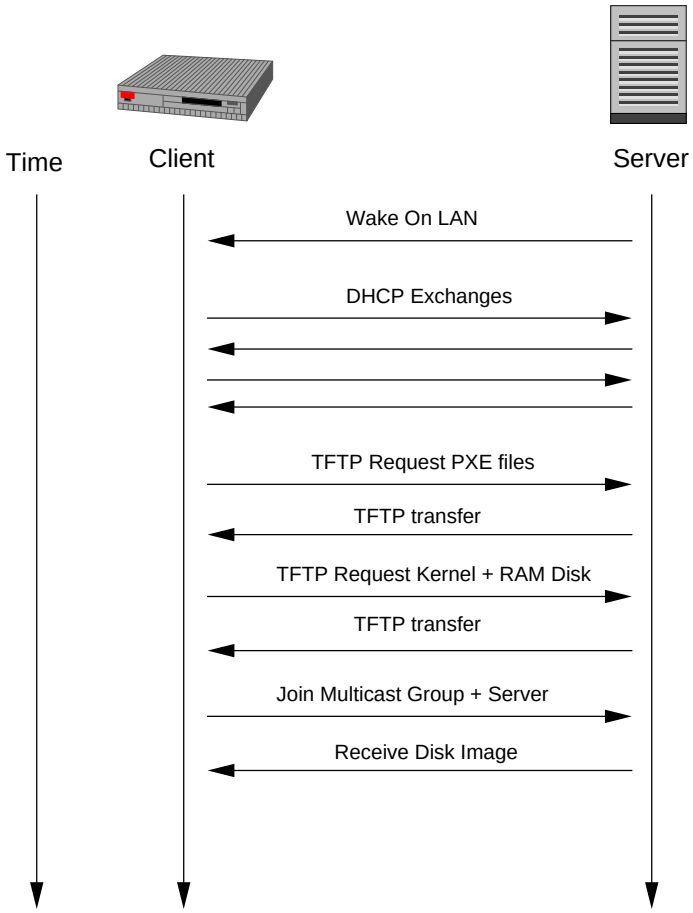

Figure 3.6: Simplified Boot Procedure

The boot procedure when installing a disk image makes use of a number of the Wired for Management (WfM)[8] features available on the machines in the network. Each machine can be powered on remotely using a “magic packet” based around the MAC address of the NIC connected to the control network, (this is known as *Wake on LAN* in the WfM specification) which then enters the Preboot eXecution Environment (PXE). This involves using extension options to the Dynamic Host Configuration Protocol to retrieve configuration information from a central server, which includes the specification of a bootable image that should be downloaded and run. This image is the custom RAM disk described above and is downloaded from the server using the Trivial File Transfer Protocol (TFTP), then executed within the PXE environment.

Once the RAM disk is booted and running, the imaging process can begin. The client connects to the central machine that is running the Frisbee server, which begins to send the disk image to the multicast group they all belong to. As more clients boot they join the group, and start receiving the file at whatever point it happens to be at, while at the same time requesting any parts they may have missed due to joining late, or loss within the network. As data is received, it is extracted onto disk, and once the entire image is extracted the client leaves the multicast group and shuts down to wait for the rest to finish. Wake on LAN is used again to power on all the machines, which will skip the PXE boot step and instead boot from the newly extracted

image on their disk.

3.6.3 Custom RAM Disk

Disk imaging has the problem that overwriting the contents of the hard drive currently in use can have unpredictable results. The solution offered by Frisbee to overcome this involves the use of a FreeBSD live CD, which loads using a RAM disk and is based in memory. The hard drive does not get used, and so can be imaged with no side effects. This does not fit in with the WAND Emulation Network however, as imaging needs to be automated and require no physical access to the machines involved – putting CDs in drives for the purpose of installing disk images violates this just as much as it does when creating the images. The whole process of using the CD also includes manually entering the commands to run the imaging software. A custom RAM disk was created that would allow the imaging to happen with no user intervention.

The RAM disk uses a Minix filesystem because of its low overhead requirements, and contains a minimal Linux kernel, an `init` program, and a number of supporting programs that it needs to do its job – a DHCP client and the Frisbee client. These last two are used straight out of the box, as-is, but the `init` needed to be created to fit this particular need. It runs the DHCP client, then collects information about the available interfaces on that particular machine which it uses to build an appropriate Frisbee command line. This is necessary because it has to be generic enough to run on any machine in the network, it's not practical to hard code addresses for each host. It runs Frisbee, and once imaging is complete, it tidies up and then shuts down the machine ready for the next step.

Chapter 4

Results

The end result of the project is a system that works for configuring the WAND Emulation Network, and for providing traffic generation for experimental purposes. Its can be best judged in two ways – that it does indeed configure all aspects of the network such that it can be used for testing, and that it makes proper use of the application models within `ns-2` when performing the tests. Section 4.1 briefly discusses what has come from the configuration and automation aspect, while 4.2 looks at how well `ns-2` fits into the framework and generates traffic.

4.1 Accurate Configuration

Deciding whether the configuration produced by the system is correct is obvious. The inputs are clear and unambiguous, and describe exactly what the end result should be, if this matches with the network that is given to the user after configuration, then it is a success. Through the course of development, this has been proven correct as test networks consistently match their specifications. The best way to determine its accuracy is perhaps through the way it is treated by users.

At the time of writing, the system is being used regularly by Sam Jansen for experiments related to his Ph.D involving integrating code from real network stacks into the `ns-2` simulator. It is important to him that he has a method of validating that the behaviour of his code in the simulator is the same as it is when running normally, and this is provided in the form of an automatically configured network. Changing topologies and requirements means that machines are often re-imaged, and they are always in the desired state.

4.2 Performance Testing

The second criteria is more complex to test, and requires using the system to its full extent. The real test is that the emulation network can properly apply the application models within `ns-2` in a physical environment to be useful for measuring performance. The quality of the models is not the issue, it is the faithfulness with

which they are reproduced now that more interactions are involved with many different devices. Because a real network stack is now responsible for sending and receiving traffic rather than the abstracted model within `ns-2`, the chances of results from simulation and emulation matching are slim. Instead, the same trends should be visible within both sets of results because of similar network and protocol artifacts, and because the traffic is being generated from the same source.

Though the Emulation Daemon had been tested during its development, that was at the packet level – individual packets were sent and observed to ensure that they were correctly formed and sent at the right time. Once the emulation agents for `ns-2` had been completed, it needed to be seen how well they would integrate. They had been written to work together, but the number of different situations the Emulation Daemon must cater for meant that there were still plenty of cases that had not yet been fully explored.

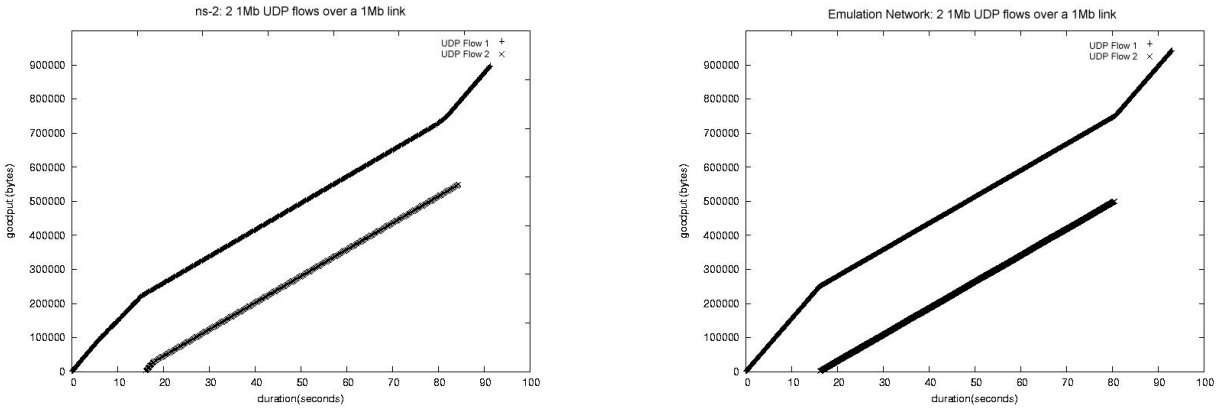

Figure 4.1: Competing UDP Flows

The most simple test was the generation of UDP traffic in the emulation network based off a simple constant bit-rate model within `ns-2` running on the server. A configuration file was created to set up two directly connected machines, and to run two UDP streams from one machine to the other. Both of these were set up to send a constant 1Mb/s of data, which was also the maximum capacity of the link. One stream was started ahead of the other and allowed the whole network to itself, before starting the second stream. After some time the second stream was stopped and the first was allowed unrestricted access again. Figure 4.1 shows this situation in the `ns-2` simulator on the left, and then again in the emulation network on the right, and both generally look exactly as it is expected they should. The first stream initially uses the entire available bandwidth until the second stream starts, from which point the bandwidth is shared equally between them. Once the second stream completes, then the amount of data being transmitted by the first stream once again fills the link.

Of note is that the second stream in the simulator transmits at the same rate as the first stream for a short period, before they both settle into an even amount, while in emulation the second flow never gets a chance

to send at more than half the available bandwidth. This illustrates the differences that are common between emulation and simulation, and the importance of validation. Discovering what these differences are is the first step towards understanding why they exist and working towards removing them.

While this section only demonstrates UDP traffic, it shows that the emulation framework is in place to extend a traffic model from `ns-2` onto the network. The interaction between the custom agents and the Emulation Daemon is consistent with the way the tested `ns-2` model works in its native environment, and so should extend to other similar models. Other protocols and more complex scenarios could be tested with little modification to the system.

Chapter 5

Conclusions and Future Work

The development of this framework has shown that network emulation is viable, and does not need to be as complex to use as it currently is. It is quite possible to have a system that wraps around the physical network and provides methods for configuration that do not require users to have specialist knowledge. The framework described is based on the infrastructure and resources available and is specific to this situation, but there is enough flexibility in the interface and supporting code base that it could be modified to support other simulators or physical networks.

Deciding to base the configuration language on `ns-2` gave firm direction in terms of what needed to be implemented to have a working system. Following the most commonly used functions in example `ns-2` scripts was useful in providing an understanding of what was important, which was needed to successfully rewrite them all. The challenge was to deal with the internal `ns-2` representations of the network and convert them into a format from which real configuration could be made.

Creating custom agents to transfer the configuration and traffic model data to the network from `ns-2` required a remote agent on each emulation node – the Emulation Daemon. Although more difficult than reusing an existing system for remote command execution, and traffic generation, this allowed more flexibility and freedom to design a protocol that was simple, with low overhead.

Making use of the *Frisbee* tools for disk imaging allowed more time for other tasks. However, in order to work in this environment, it needed to be automated. Adding extra configuration options to `ns-2` was necessary, which means configuration scripts can no longer be completely identical to those normally used with `ns-2`, but the gains far outweigh any complications due to this. Improving it to run remotely using a RAM disk and PXE was very important to the overall usability of the system.

The decision to ignore security concerns is one that cannot be ignored forever. It was the most practical approach to take during the development of this framework, but will need to be addressed at a later date.

The project was developed with the expectation that it would be used by other networking researchers, and work planned for the future should support this. A brief user guide has been written and is included as

Appendix B, but it will be necessary to work with users for a period to ensure they fully understand how the system operates. Extensions may be necessary to support more complex tests researchers wish to perform, or to add functionality that they require. The use the network has seen already has shown that users have vastly different requirements and levels of complexity in their experiments. Based on their needs the disk image library is likely to expand, and graphical elements could be built to make the entire process easier.

Because this system works very closely with `ns-2` and allows simulations to be reproduced on a real network, it enables validation of the simulator in a way that was not possible before. Being able to compare the differences in emulation and simulation directly makes any differences all the more obvious, and the implications this will have with regard to simulation quality can only be positive.

Appendix A

Emulation Machine Specifications

The WAND Emulation Network consists of 24 machines that are physically identical in every way apart from the number of network interfaces they posses. 14 of the nodes have 5 network interfaces, and the remaining 10 have 2. One of these interfaces on each machine is used to transfer control data, and so connects to the much more powerful server machine. The other interface(s) are available for configuring into any topology and transmitting experimental data. Figure A.1 shows the specifications of all nodes that are involved in the operation of the network.

All the their interfaces on the control network are terminated at a Cisco 2950T switch. It has 24 10/100 ports, and 2 10/100/1000 ports, one of which is connected to the central server. In a similar configuration, all emulation nodes are connected via serial to a Cyclades TS2000 terminal server for "out of band" management, or recovery in the case a user misconfigures network settings. The terminal server also provides access to configure the switch, as well as controlling a Cyclades AlterPath PM8i-10A power management strip for remote power cycling. Figure A.2 shows how all these components fit together.

The object labelled *Patch Panel* is a set of 3 patch panels in a separate room to which access is much less restricted than the server room that holds the machines themselves. Each of the remaining non-control interfaces on the emulation nodes is terminated here in such a fashion that the topology can be set up by changing the way interfaces are connected. Extra devices such as switches or routers can be inserted into the network at this point, along with DAG cards for performing packet capture. There is currently a research project underway that aims to build a s special patch panel that can be configured to make connections electronically, removing the need for manual patching.

	Type 1 Node (Dell Optiplex GX1)	Type 2 Node (Dell Optiplex GX1)	Server Node
Quantity	14	10	1
Processor	Pentium2 350MHz	Pentium2 350MHz	Pentium4 1.8GHz
Memory	64MB	64MB	1GB
NICs	1x 3Com 3c905B 1x MikroTik 4 port w/ Intel Chipset 82559	1x 3Com 3c905B 1x Realtek RTL 8139	2x Intel E1000 1x Intel E100

Figure A.1: Machine Specifications

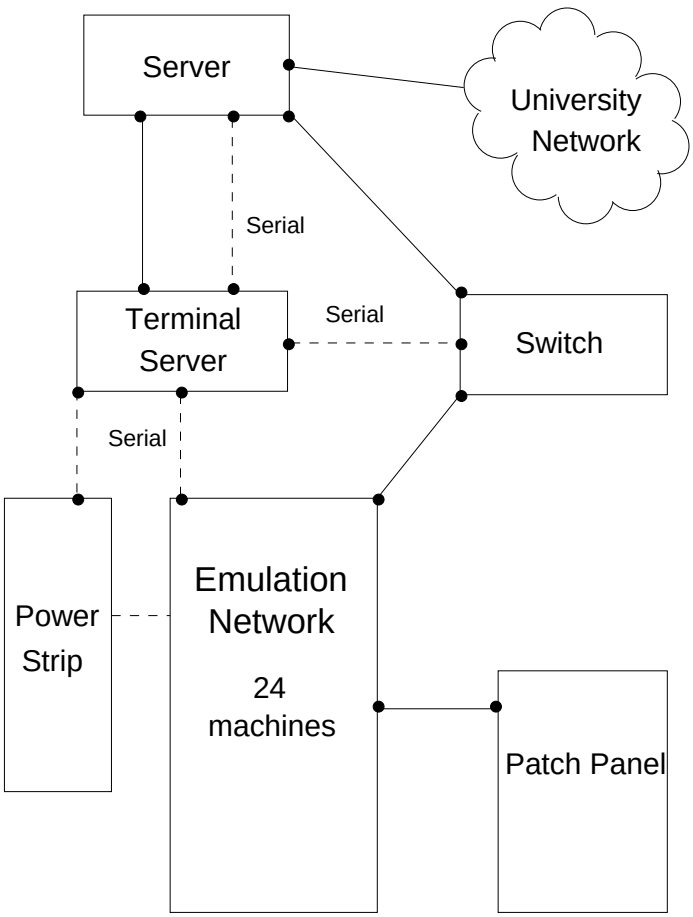

Figure A.2: Overview of Emulation Network

Appendix B

Abridged User Guide

The configuration file used in this system is very similar to that used by the `ns-2` simulator, and knowledge of their format can be transferred almost directly to this domain. They are written in OTcl, and so have available all the language constructs it possesses.

B.1 Global Settings

Every script should begin with the following line, in order to properly set up all parts of the simulator and provide access to its functions:

```
set ns [new Simulator]
```

Depending on what the desired use of the system is, one of the next two commands may be useful. Calling `config-only` will skip the disk imaging stage and proceed directly to configuring the machines themselves. The `image-only` command is the inverse of that, only performing disk imaging and not doing any configuration at all. Of course, if both of these are omitted the system will do the full set of tasks.

```
$ns config-only
$ns image-only
```

If it is only being used for configuration and/or imaging, then that is the only prerequisite. If the traffic generation aspects are also going to be used, then it is important to set the internal event scheduler within `ns-2` to act in real time with the following command:

```
$ns use-scheduler RealTime
```

Once the required commands are taken care of, the rest can be performed. The only restriction on the ordering is that each node, link, agent and application must be created before attempting to set any of its attributes.

B.2 Creating and Configuring Nodes

In simulation, users have the freedom of creating almost as many nodes as they wish, with the only tradeoff being in terms of performance. With emulation however, the number of nodes is limited by the physical number of machines available, in the case of the WAND Emulation Network at this point in time, this is 24. Each created node must be linked to a particular machine at the time it is created:

```
set <node_name> [$ns node <machine_number>]
```

This will create the node with the name <node_name>, on the physical machine with the number <machine_number>. It will have the default image installed on it (currently a Debian Woody install with a 2.4.20 kernel), although this can be changed with the command:

```
modifyImage <machine_number> "/path/to/image/file"
```

The machine numbered <machine_number> will instead be imaged with the file at the specified path, assuming that it is indeed a valid disk image, and that it exists.

B.3 Creating and Configuring Links

Links come in two types – simplex (one way) and duplex (two way). Simplex links are useful in pairs to create asymmetric links, while duplex links create a link that is the same in both directions.

```
$ns simplex-link <source_node> <dest_node> <bw>Mb/Kb <delay> <queue_model>
$ns duplex-link <source_node> <dest_node> <bw>Mb/Kb <delay> <queue_model>
```

Two node names need to specified first, as the source and destination nodes for that link. <bw> is the bandwidth of the link in Megabytes or Kilobytes, and the units should be expressed as either “Mb” or “Kb” respectively. There should be no whitespace between the value and the units. <delay> is measured in milliseconds, with the units “ms” following the value. If either bandwidth or delay are given the value “0” then they are untouched and the link will run at full speed. The only legal value for the <queue_model> currently is ”DropTail”.

Loss can be added to a link with the creation of a loss model. The only one currently supported is a uniform loss model based on packets.

```
set <loss_model> [new ErrorModel/Uniform <drop_rate> pkt]
$ns lossmodel <loss_model> <source_node> <dest_node>
```

These commands will create a loss model called <loss_model>, that drops <drop_rate> of all packets (on a scale from 0 to 1). This loss model can then be attached to the link between <source_node> and <dest_node>.

Currently there is the limitation that each lossmodel may only be used for one link, reusing it will overwrite the old one.

B.4 Creating and Configuring Agents

Agents are responsible for getting the data sent across the network. The type of agent you create determines what protocol will be used. Currently the emulation network supports UDP and TCP, and there is a pair of agents for each protocol. In each pair, one agent is a source of traffic and the other is a sink. The source sends packets to its associated sink, which outputs statistics and acknowledges them if required.

```
set <agent_name> [new Agent/UDPEmu]
set <agent_name> [new Agent/TCPEmu]
set <agent_name> [new Agent/UDPEmuSink]
set <agent_name> [new Agent/TCPEmuSink]
$ns attach-agent <node_name> <agent_name>
$ns connect <source_agent_name> <destination_agent_name>
```

Each experimental connection must consist of a UDPEmu or TCPEmu agent and a corresponding UDPEmuSink or TCPEmuSink agent. Once each agent is created, the node that it relates to must be specified by "attaching" the agent to the node with the `attach-agent` command. Finally, to show which pairs of agents should interact the `connect` command links agents with agents.

B.5 Creating and Configuring Applications

Applications decide at what rate data should be written to the agents for sending onto the network. There are two main types of these - basic traffic generators, and simulated application models. The traffic generators follow a simple function, such as the pareto distribution, or are simply constant bit rate streams. The application models include FTP and Telnet (and others such as various Internet worms, and a lot of custom models written by researchers which are not covered in this outline).

```
set <app_name> [new Application/Traffic/CBR]
set <app_name> [new Application/Traffic/Exponential]
set <app_name> [new Application/Traffic/Pareto]
set <app_name> [new Application/FTP]
set <app_name> [new Application/Telnet]
<app_name> attach-agent <agent_name>
```

Once an application model has been instantiated, it has to be attached to the transport agent that it should run over using a different version of the `attach-agent` command. Each application has their own configuration options but almost all support a basic set of commands to change packet sizes and data transfer rates.

```
<app_name> set packetSize_ <packet_size>
<app_name> set rate_ <rate>Mb/Kb
```

Packet size is specified in bytes, and rate in megabits or kilobits per second. Other attributes that can be modified are explained further in the `ns-2` manual. Applications also need to be configured to start and end at certain times with commands of the following form:

```
$ns at <time> "<application_name> start"
$ns at <time> "<application_name> stop"
```

This will start the specified application sending data <time> seconds after the experiment has started running. This is specified as a floating point value.

B.6 Final Points

There are two more things that must happen before a script is complete – the end time must be specified in a similar style to how timing is configured for applications:

```
$ns at <time> "finish"
```

and finally the command to actually start the experiment must be included. If it is not, then nothing at all will happen. This should be the last line in the file:

```
$ns run
```

In order to actually run the emulation, `ns-2` must be invoked with the name of the file containing the extra functions for use in emulation, and the name of the configuration script. The shell script `/usr/local/bin/emu.sh` runs `ns-2` with all the appropriate files, and if given the name of a configuration file will have it executed. Use it as follows:

```
/usr/local/bin/emu.sh <configuration_file>
```

Bibliography

[1] Benchmarking Methodology Working Group. http://www.ietf.org/html.charters/bmwg-charter.html, Accessed 2004.

[2] GNU Bison. http://www.gnu.org/software/bison/, Accessed 2004.

[3] GNU Flex. http://www.gnu.org/software/flex/, Accessed 2004.

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

[5] Graphviz - open source graph drawing software. http://www.research.att.com/sw/tools/graphviz/, Accessed 2004.

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

[7] Mike Hibler, Leigh Stoller, Jay Lepreau, Robert Ricci, and Chad Barb. Fast, scalable disk imaging with frisbee. In *Proc. of the 2003 USENIX Annual Technical Conf.*, pages 283–296, San Antonio, TX, June 2003. USENIX Association.

[8] Intel Wired for Management (WfM). http://www.intel.com/labs/manage/wfm/, Accessed 2004.

[9] IP Performance Metrics Working Group. http://www.ietf.org/html.charters/ippm-charter.html, Accessed 2004.

[10] Iperf - The TCP/UDP Bandwidth Measurement Tool. http://dast.nlanr.net/Projects/Iperf/, Accessed 2004.

[11] Knoppix Linux. http://www.knoppix.net/, Accessed 2004.

[12] MIT Object Tcl. http://sourceforge.net/projects/otcl-tclcl/, Accessed 2004.

[13] John Nagle. Congestion Control in IP/TCP Internetworks. RFC-896, January 1984.

[14] NIST Net. http://snad.ncsl.nist.gov/itg/nistnet/, Accessed 2004.

[15] Pathchar. http://www.caida.org/tools/utilities/others/pathchar/, Accessed 2004.

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

[17] RUDE & CRUDE. http://rude.sourceforge.net/, Accessed 2004.

[18] tcpdump. http://www.tcpdump.org, Accessed 2004.

[19] The network simulator - ns-2. http://www.isi.edu/nsnam/ns/, Accessed 2004.

[20] WAND Emulation Network. http://www.wand.net.nz/projectDetail.php?id=3, Accessed 2004.

[21] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, pages 255–270, Boston, MA, December 2002. USENIX Association.