An Investigation into Network Emulation and the Development of a Custom Network

Submitted in partial fulfilment
of the requirements of the degree of

Bachelor of Science (Honours)

of Rhodes University

Glenn Wilkinson

Grahamstown, South Africa
November 2007

Abstract

Network emulators allow for testing of network protocols and applications in a controlled environment which may otherwise be difficult or even impossible to do. Emulators manipulate traffic between hosts in a physical network, in a bridged, transparent manner, modelling real world network conditions and configurations. Such a system is very appealing in the field of academia where network dependent systems are developed on a regular basis. In this paper we firstly examine the fundamentals and varying aspects within network emulation. We compare and contrast emulation to other feasible options for testing network applications (simulation, testbeds, hybrids). Lastly, we describe how we built a custom emulator from existing tools and technologies and provide performance results.

Contents

List of Figures

List of Tables

Acknowledgements

I would like to extend a big thank you to my supervisors, Barry Irwin and Ingrid Siebörger. Your help and support this year has been invaluable. Ingird, your proof reading skills in the later stage of this project were amazing and are very much appreciated. I would also like to acknowledge the lecturers and members of staff of the Computer Science Department for helping me to grow and learn over the past few years.

I would like to acknowledge and thank Mr. Tony Long of the finance department who assisted in the sourcing of funds for my undergraduate degree, and a very big thank you to Dr. Hamilton who made it possible for me to finish my undergraduate degree. In addition, thank you to the finance department who allowed me to study towards my Honours degree with the aid of a bursary.

Finally, I acknowledge the financial and technical support of Telkom SA, Business Connexion, Comverse SA, Verso Technologies, Stortech, Tellabs, Amatole telecommunications, Mars Technologies, Bright Ideas Projects 39 and THRIP through the Telkom Centre of Excellence at Rhodes University.

Chapter 1

Introduction

1.1 Background

Modern networks are comprised of many network nodes connected via many network links. At each of these nodes there is the possibility of packets being dropped, delayed or corrupted. The Internet runs as a "best effort" network [9]; websurfing, email and other such activities tolerate large variations in such conditions but the activation of real time applications such as VoIP and streaming media require stable and predictable conditions. Packets not arriving, or being modified in structure or sequence are generally not as a result of a fault on the network path. Rather, this is a result of switching infrastructure coming under heavy load and having to drop packets or the nature of the TCP protocol and its congestion control [36] mechanisms. TCP is a protocol that can exhibit complex behavior, especially when considered in the context of the current Internet, where the traffic conditions themselves can be quite complicated and subtle [37].

The Internet's infrastructure consists of a series of routers connected by many links. As each packet arrives at a router it is examined in order to determine the next hop (which will either be another router or the destination computer) to which they should forward the packet such that it ultimately reaches its destination. Routers have physical limitations on the number and rate of packets they can forward. Sometimes they receive more packets than can immediately be forwarded in which case the packets are momentarily queued in a buffer which increases the delay of packets traversing the network. Sometimes the packets are dropped altogether if the load is insurmountable. This group of conditions are by specification and occur intentionally.

On the other hand, packets may be delayed, dropped or malformed due to a failure between the two end points. This may be at the software level such as a misconfigured router. On the hardware, or 'wire' level, packet collisions and interference from external sources may result in packets being dropped or inadvertently manipulated.

1.2 Problem Statement

The problem which we are addressing in this project is to produce a system that tests network dependent applications' performance with real world network phenomena. It is of no use to conclude that an application or service is worthy of deployment over a large scale network such as the Internet after having only tested it on a high speed local network. Therefore, using a network emulator to recreate such traffic impairments as mentioned in section 1.1 allows manufacturers, service providers, developers and, most importantly in our context, academic institutions to test the robustness of their product.

1.3 Research Goals

Our goal in this project was to build a cheap, open source based emulator from existing and readily available tools to run on the FreeBSD operating system, utilizing common hardware. It was a requirement for the emulator to be light weight with a small overhead and to be able to mimic network behaviour such as limited bandwidth, delay (constant and variable), random dropping of packets, multipath effects and enforce varying queue management policies. Furthermore, we required a usable graphical interface to the end user allowing him or her to create network nodes, save them and link them together.

This goal was achieved by building a web interface to a webserver running on a FreeBSD operating system which was deployed in several contexts; virtual machines, standard computers, and an image suitable for deployment on embedded machines such as the Soekris net4801 [11] machine.

1.4 Document Structure and Outline

This dissertation begins, in chapter 2, by briefly discussing modern network principles. Next, characteristics which manifest in the deployment of networking infrastructure as a result of real world limitations (such as packet delay, packet loss, jitter, bandwidth limitations) are investigated. We examine how network conditions affect network traffic and then look at techniques which can be used to recreate the conditions in a controlled manner. There are three broad categories or methods to test network protocols and applications. Each category shares the common goal of examining how the application or protocol handles the effects of network conditions such as delay, jitter, bandwidth limitations, packet loss and.

The first of the categories is network emulation which is the core direction of this paper. The second technique examined is network simulation. Next, the differences between hardware and software network emulators, as well as descriptions of hardware testbeds [20] and hybrid models of the above technologies are examined. The final section of chapter 2 involves a discussion on embedded machines.

In chapter 3 we introduce and present the network emulator, which we have called "Network in A Box" (NIAB). This is a custom built network emulator created and tested from existing tools and technologies. We discuss design principles and implementation details and delve into the choices and deployments of operating system, hardware architecture, programming languages and data storage. NIAB was built primarily using PHP and Perl, and deployed on a FreeBSD system in a bridged environment. The emulator exists in three forms; a Virtual Machine image, a standard computer form and an image suitable for deployment on embedded machines.

In chapter 4 we discuss the emulator test results in manipulating traffic. Statistics and graphs demonstrating the capability of the system are presented in this chapter. Key test areas included bandwidth manipulations, delays (constant and variable) on traffic flow, packet loss and multipath effects. Both single, and multiple link options are examined.

The final chapter revisits the problem statement, as well as discusses possible future extensions. The appendices contain the source code, scripts for building the embedded image, and the custom NIAB FreeBSD kernel.

Chapter 2

Terminology and Related Work

In this chapter we discuss relevant background information which is pertinent in designing a network emulator. We briefly examine some network terminology, and then discuss, compare and contrast the technologies of network emulation, network simulation and testbeds. From this discussion it is possible to understand the requirements of building a custom network emulator.

2.1 Real World Networks

The reader needs a basic understanding of the seven layer Open System Interconnection (OSI) model [44], as well as an understanding of network fundamentals because network emulators are inserted into this OSI model. As a recap to the reader, the OSI model is discussed together with an examination of network conditions that affect and alter packets as they traverse a network.

2.1.1 Network Terminology

According to [45], the OSI [44] reference model describes how information from a software application in one computer moves through a network medium to a software application in another computer. The OSI reference model is a conceptual model composed of seven layers, each specifying particular network functions. The model was developed by the International Organization for Standardization (ISO) in 1984, and it is now considered the primary architectural model for intercomputer communications. The OSI model divides the tasks involved with moving information between networked computers into seven smaller, more manageable task groups. A task or group of tasks is then assigned to each of the seven OSI layers. Each layer is reasonably self-contained so that the tasks assigned to each layer can be implemented independently. This enables the solutions offered by one layer to be updated without adversely affecting the other layers. The seven layers are depicted in Table 2.1.

7	Application Layer
6	Presentation Layer
5	Session Layer
4	Transport Layer
3	Network Layer
2	Data link Layer
1	Physical

Table 2.1: The Seven OSI Model Layers [44]

2.1.2 How Network Conditions Affect Packets

This section discusses some of the most common network conditions that affect traffic as it flows across a network. These common conditions have been deemed as the most significant conditions to be replicated in a network emulation system.

Within the bounds of TCP/IP there is no guarantee that every packet sent will be received in a timely manner, in sequence state, or even at all. However, a stream of data sent on a TCP connection is delivered reliably and in order at the destination [27]. That is to say that TCP is a reliable service but individual packets may become corrupted, delayed or lost which may require retransmission. We examine some of the conditions which may affect traffic flow, which are ultimately what an emulator replicates. However, it should be noted that an emulator is a model of reality, and with all models there are varying degrees of granularity. The complexity of the individual model is dependent on the application.

Bandwidth is a finite commodity. Hosts and routers limit the amount of traffic passing through them either as a function of saturation or by only allowing a certain bit rate.

Packets may be delayed en route to their destination. The two main sources of delay [9] on the Internet are the actual transmit time to go from the source to the destination and at a router whose input rate is greater than the output rate. This rate difference causes packets to wait in memory while the router processes previous packets. The typical distribution of packet delays through the Internet demonstrates a “heavy tail” which is a skewing toward the right when compared to normal distribution [34]. Delays may be intentionally implemented by throttling throughput. This is a common event with most ISPs and educational institutions. Delays can also be caused by packet loss and retransmission

The delay experienced by multiple packets in a traffic flow may vary. This variation in delay is known as jitter [13, 25]. It is the measure of the variability over time of the latency across a network. Real time communications (for example VoIP) usually have quality problems due to this effect. In general, it is a problem in slow-speed links or with congestion.

Packets may also be lost whilst traversing networks. The reasons for packet loss are numerous and include equipment failure, overflowed buffers, and over capacity routers. Failure with a transmission link or router causes bursts of packets to be dropped. Queuing algorithms [21] drop packets on purpose to avoid router buffers from reaching capacity. If a router is over capacity it will drop bursts of incoming packets.

Other more general impairments include out of order packets, packet fragmentation and packet duplication. Traffic engineering techniques such as Multiprotocol Label Switching (MPLS) alter packet paths to avoid congestion which can result in different packets arriving at different times at the final destination. The Internet Protocol allows IP fragmentation so that datagrams can be fragmented into pieces small enough to pass over a link with a smaller MTU than the original datagram size. [27] describes the procedure for IP fragmentation, transmission and reassembly of datagrams. [16] describes a simplified reassembly algorithm which can easily be implemented in hosts. Packet duplication may occur for many reasons, including faulty routers or hosts as well as packets thought to be lost eventually reaching their destination.

2.2 Network Emulation

In most situations the aim of experiments on network protocols is to determine their behaviour in a complex network consisting of many nodes, routers and links with different queuing policies, queue sizes, bandwidth and propagation delays [41]. According to [34] an emulator is a specialised router which emulates the behaviour of an entire network in a single hop, as is depicted in Figure 2.1. It can also be said that the purpose of a network emulator is to *mimic* the behaviour of a specific network scenario in order to analyze its impact on the software communicating over the network [26]. To accomplish this, emulators provide interfaces through which hosts communicate. A general setup is to have a transparently bridged device between two hosts which manipulates the traffic as it passes through. The transparency implies that the hosts on either end are unaware of the presence of a system in between them, and require no additional reconfiguration. Furthermore, these interfaces connect at layers within the network model protocol stack,

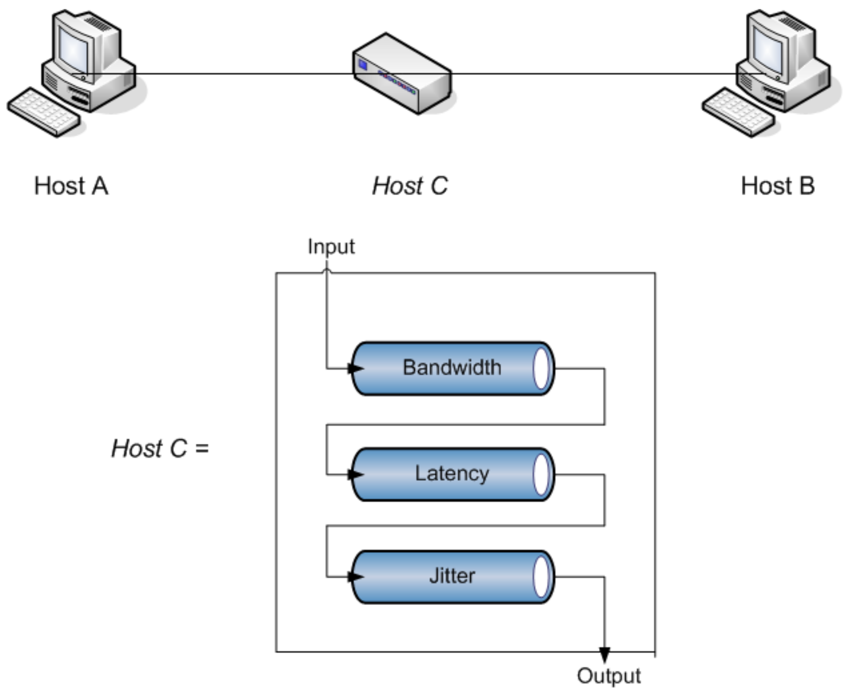


Figure 2.1: Host C intercepts traffic between hosts A and B and manipulates it

providing varying levels of manipulation. Connections at different layers are made depending on the emulator. This configuration is portrayed in Figure 2.2.

The typical approach taken by existing network emulators [12, 26, 35] is to intercept communication at the kernel level between two protocol layers, and approximate the presence of a real network with finite queue sizes, bandwidth limitations, communication delays and lossy links [41]. For example, the original Dummynet [41] catches calls from TCP to IP in the FreeBSD stack and can introduce fixed delays and packet loss. This is known as the emulation abstraction layer [26] and an emulator offers different services on different layers. The three most common emulation abstraction layers used by emulators are [26]:

1. Transport Layer Emulation - emulator reproduces communication channel characteristics such as the performance of the TCP channel. At this abstraction layer performance may be analysed to measure the impacts of channel characteristics on applications.

2. Network Layer Emulation - emulator reproduces the end-to-end characteristics if connecting network hosts. Examples in this layer include packet delays, congestion and losses.

3. Link Layer Emulation - The emulator reproduces the behaviour of single network links such as bandwidth and frame delay.

It should be noted that the above are the common insertion points for an emulator, but there are multiple possible options as shown in Figure 2.2. It is common to say that the emulator operates on the lower of the two layers that the packets are transitioning between when they are captured [26].

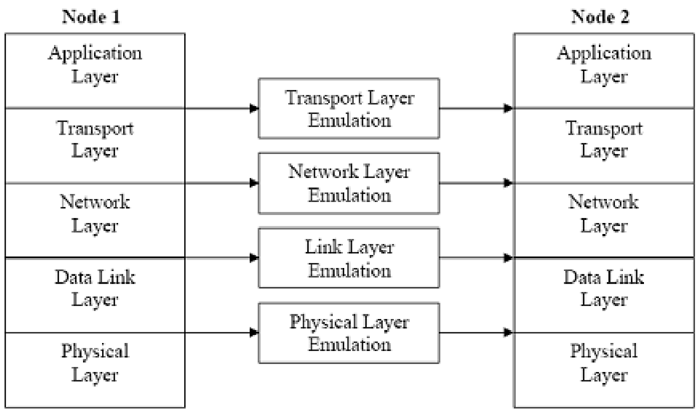


Figure 2.2: Depiction of emulators working between different layers [19].

Realistic emulation on higher abstraction layers is somewhat more difficult to achieve than on the lower layers because of the increase in the number of variables to consider. For example network layer emulation has to deal with the effects of network layer issues such as dynamic routing and queuing to achieve a realistic outcome whilst emulation of links may use actual implementations of network layer protocols to achieve the desired effects. Emulation on the link layer is the lowest possible emulation abstraction that is feasible using purely software methods, and not specialized hardware [26].

Furthermore, according to [41], in order to simulate the presence of a network between two peers two elements are required to be inserted in the flow of data; routers with bounded queue size and a given queuing policy, and communication links or pipes with given bandwidth and delay.

Several functions of emulators noted by [15, 26] are as follows:

- Abstraction - Varying the simulation granularity allows an emulator to accommodate high level and low level testing of protocols, ranging from the detail of an individual protocol to the aggregation of many data flows.

- Emulation Parameters - Network emulation has to mimic the behaviour of an actual network link as closely as possible. All parameters affecting performance have to be considered. (Parameters, as discussed, include bandwidth limitation, propagation delay, jitter and dropped packets).

- Transparency - The connection of the software being tested and the emulated network, through the respective interface, must be completely transparent to the software. This facilitates the traffic to be evaluated in its original, unmodified form.

- Minimal Side Effects - There will be some inevitable delay when transferring a datastream through an emulator, caused by overhead. Since this overhead is not part of the original specified scenario it must be minimized.

- Scenario Generation - This relates to an emulators ability to save predefined templates of network design. Automatic recreation of complex traffic patterns, topologies and events can help generate such scenarios. For example, a template emulating the effect of a data flow going through a Telkom DSL connection, followed by an ATM link, followed by a satellite link, followed by a T1 trunk and finally another DSL link.

- Visualization - Analyzing output as performance numbers does not make for easy evaluation of the effects of the emulator. Network animating tools [3] and graph software give valuable output to users.

- Extensibility - Several authors note that a modular design to emulators is an important function as it allows it not only to be extended, but also to be dynamically loaded.

In its most simplistic form a network emulator emulates a single link between the two hosts, focusing on some network properties at a single point. *Centralized* emulators can work with dynamic scenarios but constitute a bottleneck to the emulated system and therefore limit the scenario in size and bandwidth. It is possible to setup several emulators on a number of hosts to overcome this problem but few emulators of this sort support a central dynamic model [26, 19]. Most emulators work with centralized real time simulation components which limits the scenario size and maximum traffic, or focus on the emulation of some network properties at a single point. A more desirable approach as put forward by [26] is to emulate more realistic links, showing how several emulated links can be combined to reproduce a comprehensive network model. This *distributed* option as discussed by [19] deploys traffic manipulation capabilities on every node. Distributed emulations can be made dynamic by having a central coordinator update the manipulations to be applied to the traffic. Furthermore this option allows for the emulator to be more scalable, and allows for higher bandwidth between nodes. This is because with a centralized model all packets flow through a central packet manipulator. With a high volume of traffic it may have trouble keeping up with the real time manipulation.

Combining simulators and emulators also allow for a greater number of links to be created in the simulated environment. We will discuss this idea in further detail in the coming sections, along with a brief look at testbeds and hardware emulators.

2.2.1 Emulation Techniques for Network Conditions

There are numerous approaches and models to manipulate traffic as it passes through an emulator system. We will now examine some of the more common methods to emulate packet delay, packet loss, bandwidth limiting, and jitter. The routines used to generate the impairments in the emulator don't need to be realistic models of the actual internal mechanisms of networks and routers, but should rather be computationally simple whilst still being able to imitate a wide range of network behaviours. It is important that the computations required for manipulating traffic are able to keep up with the packet receive rates.

Packet delay may either be fixed or random with the the shape of the random distribution curve being settable. From real world results [34] a heavy-tail (a skewing towards the right) distribution is observed for packet delays on a graph of packets versus time. [34] explains that this curve makes sense because there are far more ways for things to go wrong for a packet (worse than average) than there are for things to go right (better than average). Simulations can not be run fast enough to generate individual packet delays in real time resulting in most emulators adopting a simpler approximation approach. A table of values is generated before hand and a random lookup is made at run time from such a model. It is useful to be able to set the mean, standard deviation and linear correlation when using this model; setting a large standard deviation and a small linear correlation, packet reordering can be maximized. Finally it can be noted that, in terms of emulation, the delay times are dependent on the granularity of the system clock. This heavy-tail skew is depicted by [34] in Figure 2.3.

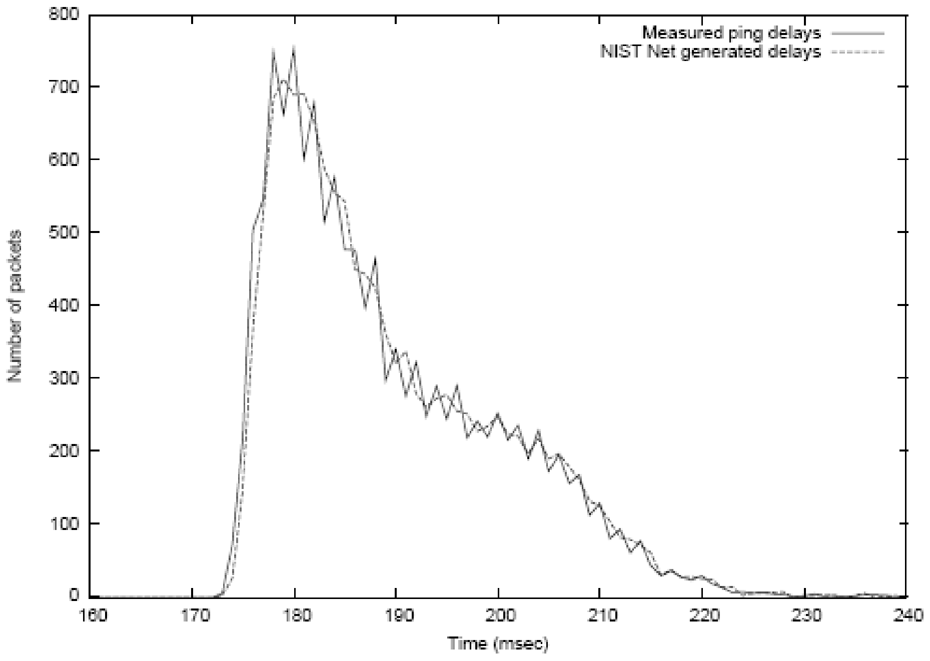


Figure 2.3: Real vs NIST Net-synthesized delay distributions [34]

Packet loss may be modelled either statistically as in [26] or by a simple random dropping approach used by [28]. The statistical approach may be followed by using the Random Early Detection (RED) method [21] or the Derivative Random Drop (DRD) method [22]. An alternative to RED is GRED (Gentle Random Early Detection) which decreases the packet drop rate in a slower manner. DRD drops packets with a probability that increases in a linear fashion with the instantaneous queue length. The queue length is measured in the number of packets. For example, in [34] when the queue length reaches the configured minimum queue length, DRD starts dropping 10% of packets and the loss percentage continually increases until the actual queue length reaches the configured maximum queue length. At the configured maximum queue length, DRD loses 95% of packets [33]. RED is a more complex approach which can result in coordination of packet drops and retransmissions across multiple flows after certain types of instantaneous traffic bursts. The less complex approach to packet loss is to simply generate a random number between 0 and 1 for each packet and compare it to the predefined drop probability. If this number is greater than the drop probability it is dropped, otherwise it is sent on. The statistical approach will model a more realistic scenario but is somewhat more difficult to implement.

Bandwidth limitations may be computed on an instantaneous basis. When a packet arrives, the theoretical amount of time the packet would take to transmit at the set bandwidth limitation is calculated and the packet is held in the queue for the duration. Because bandwidth limit-related delays are cumulative, it is advisable to impose limits on queue lengths. Each packet should be delayed by the packet size divided by the bit rate.

Jitter, as mentioned earlier, is defined as a random delay specified with an average and standard deviation. Each link may be given a jitter probability, average and maximum deviation. Then the time a packet is delayed for may be calculated with a random number R as follows [28]:

$$D_{jitter} = AVG + R * AVG * STD$$

Packet modifications in a random or probabilistic manner is another feature of many network emulators. Fields in the packet header such as checksum source and destination IP addresses and/or port numbers and TTL values are likely candidates for manipulation. Such modifications are useful for testing responses to unexpected situations.

2.3 Network Simulation

Network simulators are typically discrete event-based systems [29]; they facilitate testing to a great extent, but they have a major limitation in that they rely mainly on models of both the physical infrastructure and the networking protocols. The problem being that models are only representations of reality and are never entirely accurate. They may ignore unaccounted-for but important factors. This is especially true in complex situations in which obtaining an accurate model is not possible. Network simulators are also unable to replicate the view of real-time for end users.

Key events are stored in a sorted list, sorted by the time at which the event takes place. This list is traversed and each event is sequentially executed. An event's execution may result in additional events being created. For example when a packet is dequeued from a router, it will either be enqueued into another router or dropped [19].

One important component that has been consistently simplified or ignored in network models and simulations is the processing cost or processing delay on a network node. Traditionally, this delay has been considered negligible as only simple packet forwarding functions needed to be implemented [40].

2.4 Network Simulation/Emulation Hybrid

Since the simulation of complete networks has been addressed in detail already in [15], it makes sense to reuse existing simulators for emulation purposes. According to [26], simulators can work with complex network models, and their simulator core can compute the effects that specified network properties have on network traffic traversing the model which makes up the main part of an emulation facility.

Two problems are identified with linking a discrete simulator to an emulator. Network simulators work with discrete event schedulers, with the events being processed in a non-real time manner. The scheduler needs to be modified to work in real time to combat this first problem. The second problem is that there needs to be an interface between the emulator and the simulator. There are numerous examples of such hybrid systems available [20, 30].

As another author notes [34], emulation is a combination of two common techniques, simulation and emulation. We can define simulation as a synthetic environment for running representations of code, whilst emulation relates to live testing in a real environment for running real code. Emulation can be seen as a semi synthetic environment for running real code in the sense that it is a real network implementation (such as [28][34][46]) with an additional ability to create synthetic delays and faults [34]. It can therefore be said that emulation offers advantages of both discrete simulation and live testing. That is to say a controlled, reproducible environment offered by simulation versus a real environment in which the representation is obvious [34].

2.5 Hardware Emulation

Hardware network emulators are physical routers which manipulate traffic as it passes through them. The advantage of using hardware to manipulate traffic is a higher level of precision and repeatability. In software based systems there is a dependence on the clock cycle of the CPU [41] which is countered by using specialized hardware. Furthermore, a dedicated hardware device does not have the operating system overhead which would be experienced with software based solutions. Such accurate hardware solutions are essentially only required to meet the needs of organizations that are designing, building, testing, deploying and using high-speed fiber optic data networks. There are several options available for hardware emulation such as products designed by Anue [1] and Packetstorm [9] which are capable of processing traffic up to 10Gbit/s.

2.6 Testbeds

Several nodes running individual network emulators can be used to emulate multiple network links. The combination of a central network model and the use of link layer emulation tools make up a complete network emulation facility working on link layer abstraction. Having multiple nodes allows for the load to be distributed, enabling higher complexity network situations to be modelled. For example, the The University of Utah Network Emulation Facility [32] consists of over 200 nodes with over two miles of cabling. Their goal is to create a unique type of experimental environment: a universally-available "Internet in a room" which will provide a new, much anticipated balance between control and realism.

2.7 Comparisons of Techniques

This section outlines the advantages and disadvantages of the different techniques discussed in sections 2.2 to 2.6 and concludes that for our purposes in this project the hybrid emulator/simulator is the best choice.

2.7.1 Emulation vs Simulation

One of the key differences between these two approaches is that simulation creates an artificial representation of time [19]. The simulator can skip ahead until an event is due. Some events might take a long time to be evaluated, but this does not matter to the simulator, since the representation of time is artificial and does not have to be synchronized with an outside clock. This is a useful characteristic over emulation when the situation may involve network conditions which take a relatively long period of time. However, as already mentioned, simulators have a major drawback of not being an accurate representation of reality. It is difficult to decompose certain situations into a modelled representation suitable for the simulator. In the academic situation in which we wish to modify traffic, the simulator option is not of use as it would be significantly difficult to represent our situations to a form which could be inserted into the simulator. The emulator option is superior in this situation as no modification of the software under test or additional work is required.

However, in some situations, simulators may be a useful, cost effective option such as when the problem domain is limited and well defined.

2.7.2 Emulation vs Hybrid

The key problem to notice with a purely emulated system is the difficulty in replicating multiple links. As noted in section 2.2 the approach with emulation is to make single, atomic, manipulations to packets as they pass through the system. Emulating a single link with this method is feasible as a single operation of delay, jitter, random loss etc is applied to the data stream as data flows. However, culminating effects of multiple links is difficult. Some systems attempt to achieve this by sumating effects of each condition over each link. For example, summing the delay effect of several links and applying the total delay value.

The hybrid option is a more desirable approach to emulating multiple links as put forward by [26] and [34].

2.7.3 Hardware Emulation vs Software Emulation

This category comes down to a price/performance trade off. Hardware emulators [1, 9] can provide for largely more complex scenarios with finer granularity and increased precision and the ability to pass traffic at much higher rates (up to 10Gb/s). This is achieved through specialized design and low overheads as compared to regular PCs. Modern end users computers have the processing capacity to handle complex network scenarios. It can be noted that for most intents and purposes a software solution is acceptable. The hardware option is only desirable in situations involving large amounts of traffic flowing at high speed, where absolute accuracy of results is essential.

2.7.4 Testbeds vs Emulation

A testbed is of course a largely expensive approach. Deploying many machines with complex network configurations involves cost in the form of currency as well as administration. Such a solution is viable over emulation in situations where a very complex, very large network needs to be emulated (such as the Utah testbed which is attempting to emulate the entire Internet).

2.7.5 Conclusion of techniques

The most viable, modern, small to medium scale option is the hybrid model incorporating emulation with aspects of simulation. This can be run on a single machine and relatively accurately mimic impairments over multiple links. Simulators have the obvious drawback of being based purely on a model therefore lacking aspects of realism. Testbeds are expensive and impractical for all but large scale scenarios. Hardware emulation has its niche but for the most part is overly expensive.

However, that being said, a combination of techniques during the development of an application or protocol is desirable, as noted by [19]. Each method has advantages and disadvantages, and is appropriate at a different time in the development cycle. In the early stages of development only general ideas exist on how the protocol or application would behave in different situations. Using these ideas, a model of the behaviour can be created enabling simulation to test the model, in order to evaluate the performance of the new idea. If the performance is promising, an actual implementation may be developed which would detail how the protocol would respond in any given situation. Emulation and network testbeds can be used to test implementations to see if they perform as well as the model suggested.

For the purposes of this investigation, within the academic domain, a hybrid model of emulation and simulation has been concluded to be the most viable and desirable solution. Under the constraints of cost and administration, as well as the low anticipated traffic volume and no need for absolutely precise manipulations this option is acceptable.

2.8 Limitations of Emulation

As with all models of reality, network emulation can only approximate the behaviour of a real system with given features. Most approximations derive from the granularity and precision of the system clock of the device they are running on. The granularity limits the resolution in all timing related measurements but in practise this may only pose a problem in emulating many network links or fast networks with short pipes resulting in an overall packet delay comparable with the granularity.

A second limitation or problem is that of a task being run late or even being missed depending on system load. However for small scale projects all the emulator products examined claim to handle load on obsolete machines (NISTNet [34] claims to be able to process 100Mb/s on a 300Mhz machine).

2.9 Deployment Options

A Virtual Machine Manager [6] allows the deployment of an operating system on a Virtual Machine. One version of the network emulator was created inside a FreeBSD Virtual Machine. This Virtual Machine image may be deployed on any operating system running VMWare.

An embedded machine is a computer with a small form factor, with lower specifications and power consumption than a PC. It has limited I/O in the form of serial and/or network port interfaces, as opposed to video, keyboard and mouse I/O. Embedded machines are designed to be deployed and left unattended, or have a specific task with a well defined simple interface. In our context we have chosen one of the deployment options of our network emulator to be on an embedded machine allowing it to perform the well defined task of manipulating network traffic.

Lastly, the emulator has a standard computer deployment option.

2.10 Summary

In this chapter we defined several networking terms and phenomena. Network emulation was then examined in depth and compared to other techniques in this area (network simulation, testbeds). We concluded that network emulation was a superior choice over the alternatives in our domain; small quantities of traffic in an academic environment. Furthermore three options were mentioned for the deployment of the emulator; a Virtual Machine, an embedded machine or a standard computer. Based on the findings from the literature review the design and construction of a custom network emulator is described in Chapter 3.

Chapter 3

Design and Implementation of Custom Built Emulator

This chapter discusses the details of how the emulator was built. It was constructed in two phases. In the initial phase it was developed and tested on a virtual machine and then on a computer. In the second stage an image was produced suitable for deployment on a Soekris [11] embedded machine. Traffic flows into the emulator and is “picked up” by ipfw firewall rules, which pass the select traffic onto a set of dummynet pipes, which in turn create the desired manipulation effects. The dummynet pipes within the emulator have been created prior to this and contain properties to manipulate the traffic in the desired manner. Traffic leaves the dummynet pipes, through the ipfw rules and back through the opposite interface. The properties of the pipes are set through the web interface of the emulator. We have implemented the ability for the emulator to throttle bandwidth, to delay traffic in a constant or variable manner (jitter) and to drop random packets on a link. It is also possible to create a multipath effect.

3.1 Design

No formal design approach was used to construct NIAB. However, the principles and ideals of the Agile [31] software development framework may be construed to be the driving force behind the project. Small incremental deliverables with testing at each stage of development highlights this approach. NIAB was designed to be lightweight and have small overhead. The project focus has been on deploying a simple network emulator with a minimalist front end and a small backend footprint. Part of the design specification was for the system to be non obtrusive and transparent to users.

3.2 Implementation

The emulator was deployed on a machine which has three network interfaces. One network interface is used for the control of the system via ssh or a web based user interface. Traffic passes through the other two interfaces transparently in a bridged manner whilst network conditions are applied to the traffic flow.

3.2.1 Phase 1 - Virtual Machines and PC

The emulator was initially constructed and deployed using a FreeBSD installation running inside VMWare's "VMWare Server" [6]. From here it was moved onto a standard x86 computer for further development and testing.

3.2.1.1 Operating System

The FreeBSD operating system was chosen as the platform on which to deploy our network emulator. It was seen as an appropriate option as it can be stripped down to a small, lightweight size which is suitable to a system which has only to pass traffic between interfaces and apply simple manipulations to them. Furthermore, the FreeBSD ipfw firewall system along with dummynet and bridge components provide a solid infrastructure for the emulator. Custom FreeBSD kernels were built for this system and are included in Appendix A.3. Several sysctl [24] variables were set to allow processes such as IP forwarding between interfaces.

A minimal install of FreeBSD was deployed. No source code, manual pages or windowing system were used. Additional packages installed include the Apache webserver, PHP, Perl and several CPAN and PHP packages for manipulating XML data structures.

3.2.1.2 Use of Existing Tools and Structures

IP firewall, or "**ipfw**", is FreeBSD's IP packet filter and traffic accounting facility [24]. It processes access rules for the FreeBSD firewall. Each of these rules relates to specific kinds of packets and describes what to do with them. When ipfw receives a packet it checks each of the rules in a predetermined order until it finds one which matches the packet in question. After a match is found an action is performed. By default once a match is found processing of the rule list terminates, as it is normally an accept or reject. We will see, however, that it is useful in certain situations to continue processing through the rule

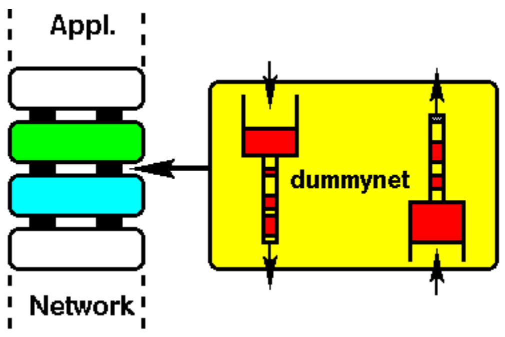


Figure 3.1: The dummynet subsystem [41]

list. In addition, the ipfw rules can match on select interfaces and on select directions. This was utilized in the NIAB system to match traffic only on the two selected traffic interfaces, and not the control interface. It also allows the emulation of non uniform bidirectional links with varying up/down stream characteristics. These two properties are demonstrated in Listing 1 where we create a typical ADSL connection.

Within the FreeBSD kernel, setting "options IPFIREWALL" adds the ipfw firewall functionality. "options IPFIREWALL_VERBOSE" adds the logging of packets to the syslog facility. "options IPFIREWALL_VERBOSE_LIMIT=100" specifies that a packet that is logged should only be logged up to 100 times. "options IPFIREWALL_DEFAULT_TO_ACCEPT" specifies that the default status of ipfw, when enabled with sysctl, is to allow all packets. By default, without this kernel option, when ipfw is started it blocks all traffic. Subsequent firewall rules will, of course, change this behavior, however this distinction is important in the interim between booting the NIAB system and running the rule file. Within the NIAB system we chose to assign all ipfw rules to the ruleset number 13 to differentiate them from standard rules, which are assigned to set 0 (ipfw had 32 sets available for rules).

Ipfw utilizes the **dummynet** tool. Luigi Rizzo's dummynet [41] works on the kernel level. It is a flexible tool originally designed for testing networking protocols, and since then (mis)used for bandwidth management. According to the creator of dummynet, it simulates/enforces queue and bandwidth limitations, delays, packet losses, and can be used for multipath effects. It also implements a variant of Weighted Fair Queuing called WF2Q+ as well as RED and GRED active queue management. Dummynet works by intercepting packets in their way through the protocol stack as displayed in Figure 3.1. They are then passed through one or more pipes, which create the effects of bandwidth limitations, delays and packet losses [41].

Dummynet is able to implement variable delay times (jitter) when manipulating traffic in one of three ways, either by editing and recompiling dummynet.c source code; by dy-

Listing 1 Typical ADSL Connection

```
#ipfw add set 13 pipe 1 ip from lnc0 to lnc1 out
#ipfw add set 13 pipe 2 ip from lnc1 to lnc0 in
#ipfw pipe 1 config bw 384Kbit/s delay 2ms plr 0.01
#ipfw pipe 2 config bw 512Kbit/s delay 1ms plr 0.01
```

namically editing the pipes from an external program; or by using probabilistic matching of ipfw rules to dummynet pipes. Within the NIAB system we chose to use probabilistic matching of pipes as it is the least complicated method whilst giving the required outcome. Multipath effects can be emulated using a similar approach, by creating different pipes with the same ipfw rule number, and giving each a probability of matching.

It should be noted that dummynet can be loaded as a kernel module and not compiled into the kernel, but since the dummynet system is of such integral importance to the NIAB system it has been included in the kernel.

Therefore, we can use ipfw in conjunction with the creation of the backend of our emulator. Network traffic of the order we wish to manipulate can be identified with ipfw's firewall ruleset (by interface, packet type, IP address etc). The ipfw ruleset therefore acts as the insertion point from the network interface to the emulator. From here traffic may be passed through dummynet pipes which add the effects of delay, bandwidth limitations and packet loss. As an example, a typical ADSL connection is presented in Listing 1.

In the demonstrated typical ADSL link in Listing 1 we create two pipes, one for each direction of traffic. From here we add an upstream bandwidth delay of 384Kbit/s and a downstream one of 512Kbit/s. Furthermore, there is a packet loss of 1% created in each direction.

When considering **queue management** and queuing policies with respect to multiple flows, the simplest form of congestion control is the "drop tail" [14]. As discussed in section 2.2.1 "drop tail" is a simple queue management algorithm; traffic is not differentiated and therefore treated equally. When the queue is filled to its maximum capacity, the newly arriving packets are dropped until the queue is freed to accept incoming traffic. Drop Tail distributes buffer space unfairly among traffic flows which can lead to "global synchronization" ([14], [38]) as all TCP connections hold back at the same time, and then step forward at the same time. Networks become under-utilized and flooded by this turn based behaviour of stepping forward and back. The more sophisticated options which counter global synchronization are RED and GRED. They are used in many modern routers and network devices which in effect implements congestion management, and are not designed to operate with any specific protocol in mind but perform with protocols which perceive packet loss as congestion. TCP is one such protocol. RED monitors the average queue size and drops packets based on statistical probabilities. If the buffer is almost empty, all incoming packets are accepted. As the queue grows, the probability for dropping an incoming packet grows. When the buffer is full, the probability has reached 1 and all incoming packets are dropped. RED is considered more fair than Drop Tail. The more a host transmits, the more likely it is that its packets are dropped. The RED algorithm is depicted in flowchart and code in Figure 3.2 and Algorithm 1.

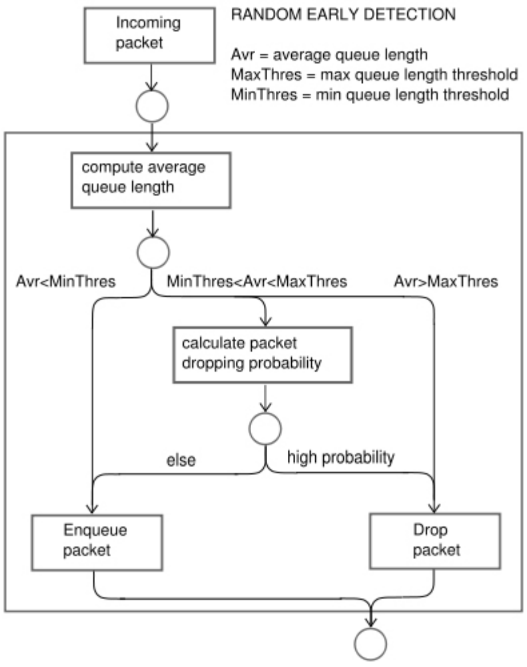

Incoming packet
RANDOM EARLY DETECTION
Avr = average queue length
MaxThres = max queue length threshold
MinThres = min queue length threshold
compute average queue length
Avr<MinThres
MinThres<Avr<MaxThres
Avr>MaxThres
calculate packet dropping probability
else
high probability
Enqueue packet
Drop packet

Figure 3.2: Random Early Detection Depiction [47]

Algorithm 1 Random Early Detection Pseudo Code [21]

```
for each packet arrival
 calculate the average queue size avg
 if minth <= avg < maxth
  calculate probability pa
  with probability pa:
   mark the arriving packet
 else if maxth <= avg
  mark the arriving packet
```

Listing 2 Calculation of average queue size

$$avg_i = (1 - wq) \times avg_{i-1} + wq \times q$$

The process starts when the average queue size is greater than the minimum queue length threshold. RED was specifically designed to use the average queue size, instead of the current queue size, as a measure of incipient congestion, because the latter proves to be rather intolerant of packet bursts [38]. The NIAB system calculates the average queue size (in which wq is recommended to be set to 0.002[17]) and q is the individual queue size as per the formula in Listing 2.

The NIAB system is able to use both RED and GRED active queue management policies within the ipfw and dummynet subsystem, and has a default queue size of 50 slots. The user is not given the ability to change the size of the queue as it is not seen as necessary. RED/GRED queuing is handled via the insertion of a traffic shaping delay node, in much the same way that bandwidth, delay, and packet loss is handled. Within the NIAB system there are four variables associated with setting the (G)RED policy as listed in Table 3.1.

Options	Flags	Values
Weight to calculate average queue size	w_q	(0..1]
Minimum Queue Length Threshold	min_th	Integer
Maximum Queue Length Threshold	max_th	Integer
Maximum Dropping Probability	max_p	(0..1]

Table 3.1: (G)RED Queue Management Options

Finally let us consider the **bridge** functionality of ipfw. A bridging firewall is a firewall that does not perform routing but allows transparent passing of layer two (see Table 2.1) traffic. In theory, the bridging firewall could be replaced at any time with a simple Ethernet hub without affecting network operation on either end. As previously mentioned, the interfaces involved in the emulation do not have IP addresses. This is extremely useful and powerful as it can be inserted onto any Ethernet connection at any point on the network without any further configuration of the network. Within FreeBSD the bridge driver is a kernel module and will be automatically loaded by ifconfig [24] when creating a bridge interface. It is possible to compile the bridge in to the kernel by adding "device if_bridge" to the kernel configuration file. Listing 3 lists the required options appended to the NIAB system's /etc/rc.conf file to enable the bridge, and ensure it starts up with the system.

Listing 3 /etc/rc.conf options for the NIAB bridge

```
cloned_interfaces="bridge0"
ifconfig_bridge0="addm bge0 addm fxp0 up"
ifconfig_bge0="up"
ifconfig_fxp0="up"
```

Listing 4 Kernel options required for basic emulator setup

```
options IPFIREWALL
options IPFIREWALL_VERBOSE
options IPFIREWALL_VERBOSE_LIMIT=100
options IPFIREWALL_DEFAULT_TO_ACCEPT
options dummynet
device if_bridge
options IPSTEALTH
options HZ=10000
```

However, as traffic passes through the NIAB system the Time To Live (TTL) IP header will still be decremented making the bridge noticeable. This problem was corrected by using **stealth forwarding** which does not decrement the TTL, making the NIAB system completely invisible. Stealth forwarding is used by including in the kernel “options IPSTEALTH”.

The FreeBSD kernel is by default set to 100Hz which means a granularity of 10ms, and the NIAB system performs its task once per timer tick. For accurate simulation of high data rates the kernel was updated to have a frequency of 10000Hz (as per the recommendation of [41]) by altering the kernel configuration file, thus increasing the granularity. A rate that is set too extreme may cause some interfaces using programmed I/O to take a considerable time to output packets. So, reducing the granularity too much might actually cause ticks to be missed thus reducing the accuracy of operations [7].

The additional kernel options required as per the discussion in this section are presented in Listing 4. See Appendix A.3 for the full kernel listing. Furthermore, specific systcl kernel values were set; “sysctl net.link.bridge.ipfw =1” to forward packets over the bridge, “net.net.ip.fw.one_pass=1” to match packets to each firewall rule, thereby passing traffic through every dummynet pipe as opposed to just the first one and “net.inet.ip.fw.enable=1” to forward IP packets over the system.

To sum up what has been discussed in this section with respect to the backend of the emulator; ipfw matches packets to its ruleset and passes them onto several existing dummynet pipes. These dummynet pipes manipulate traffic (from select interfaces and directions) adding delay, packet loss, bandwidth limitations and active queue management. All this is achieved through a transparent bridged arrangement, making NIAB invisible to systems plugged into its inbound and outbound traffic manipulation interfaces.

3.2.1.3 Back End Scripts

Perl is a general-purpose programming language [4] originally developed for text manipulation. It is now used for a wide range of tasks including system administration, web development and network programming. We have developed numerous scripts to manage our emulator with the Perl scripting language.

Furthermore a small module was developed to manage manipulating ipfw rules. The alternatives to this are either to use Perl's "backticks" functionality ('ipfw <cmd>') to directly access the ipfw program, which is not a safe option, or to use the existing ipfw manipulation module which is cumbersome and verbose; beyond our simple requirements. In general backticks should be avoided, as they impose needless security, portability, and maintainability problems. It is possible to submit non expected input to a Perl program via the Common Gateway Interface (CGI) gateway resulting in the running of arbitrary shell commands.

The functionality of the scripts will be made apparent in section 3.2.1.5 where we examine the user interface. For now it is suffice to say that scripts exist to setup the systems network interfaces, to create and delete link nodes (with properties of throughput, delay, packet drop etc), to set which nodes are currently active and several utility scripts. The representation of the link nodes is discussed in section 3.2.1.4. The link from the front end to these back end scripts is done using CGI and is discussed in section 3.2.1.5.

The specific scripts constituting the NIAB system can be viewed in Appendix A.2.

3.2.1.4 Data Store

Several considerations were required with respect to storing data. The settings of the emulator, as well as custom nodes to be emulated would have to be stored and retrieved. Three main possibilities were considered for data store; flat file, database or XML files. Initially a SQL database was deployed and used to store settings as well as program state and node information. This was found to be an excessive solution and an XML file was decided on for such purpose. An example of a nodes.xml file is presented in Listing 5.

Each node in the file has bandwidth, delay, drop rate and queue management characteristics which are parsed by Perl scripts (section 3.2.1.3) to be instantiated as ipfw dummynet

Listing 5 Example of a nodes.xml portraying several emulated nodes

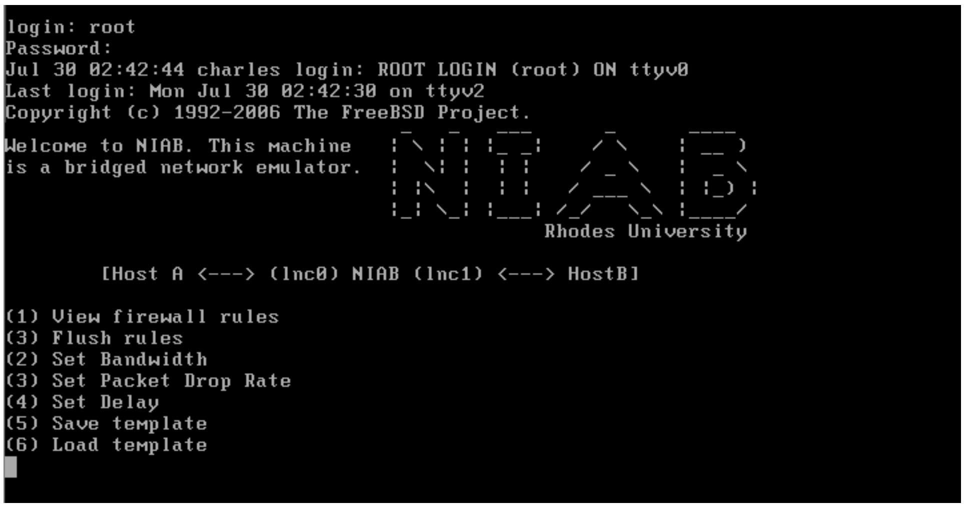
```
<?xml version = "1.0"?>
<nodes>
 <node name="Satelite" inUse="Y">
    <bwUp>1024</bwUp>
    <bwDown>2048</bwDown>
    <delay>60</delay>
    <drop>0.01</drop>
    <qMgt type="GRED">
       <w_q>1.1</w_q>
       <min_th>2.2</min_th>
       <max_th>3.3</max_th>
       <max_p>4.4</max_p>
     </qMgt>
 </node>
<node name="ADSL" inUse="N">
    <bwUp>384</bwUp>
    <bwDown>512</bwDown>
    <delay>1</delay>
    <drop>0.012</drop>
    <qMgt type="RED">
       <w_q>1.1</w_q>
       <min_th>2.2</min_th>
       <max_th>3.3</max_th>
       <max_p>4.4</max_p>
     </qMgt>
 </node>
</nodes>
```

pipes. The “inUse” attribute denotes that the pipe exists, as opposed to being stored for the potential of being selected and used. It was decided that all dummynet pipes are stored in set 13 of the ipfw rule list. Each ipfw rule belongs to one of 32 different sets, numbered 0 to 31. Set 31 is reserved for the default rule. By default, rules are put in set 0, unless specified when creating a new rule. Sets can be individually and atomically enabled or disabled, so this mechanism permits an easy way to store multiple configurations of the firewall and quickly (and atomically) switch between them. Set 0 remains in use for general firewall rules which may be required in the context in which the emulator is deployed.

3.2.1.5 User Interface

With the backend Perl scripts to manipulate ipfw and dummynet, as well as manipulating file structures for storing settings we move onto a user interface.

Several options were considered for the user interface section. The first option which was deployed during testing was a simple bash script which was executed upon connecting to the machine via ssh as shown in Figure 3.3. This, however, was seen as not intuitive enough and a graphical interface was developed. Options in this realm were to either create a custom message passing system with server and client sockets, or to create a web interface after deploying a web server. It was decided that using a web server was the best option, namely Apache 2.



```
login: root
Password:
Jul 30 02:42:44 charles login: ROOT LOGIN (root) ON ttyv0
Last login: Mon Jul 30 02:42:30 on ttyv2
Copyright (c) 1992-2006 The FreeBSD Project.
                                 _   _   ___       _     ____
Welcome to NIAB. This machine   | \ | | |_ _|     / \    | __ )
is a bridged network emulator.  |  \| |  | |    / _ \   |  _ \
                                | |\  |  | |   / ___  \  | |_) |
                                |_| \_| |___| /_/   \_\  |____/
                                             Rhodes University

        [Host A <---> (lnc0) NIAB (lnc1) <---> HostB]

(1) View firewall rules
(3) Flush rules
(2) Set Bandwidth
(3) Set Packet Drop Rate
(4) Set Delay
(5) Save template
(6) Load template
█
```

Figure 3.3: Initial User Interface

Initially the graphical user interface (GUI) was developed with Flash and Google Web Toolkits, but these came across as too cumbersome so a more simple GUI was created using

HTML, CSS and PHP. A number of screenshots (Figures 3.5 to 3.13) help to facilitate discussion regarding the role of the web interface. The system is accessed via a web browser and all configuration and manipulation is done through this medium. The web interface communicates to the underlying Perl scripts via the CGI technology through a process of GET and POST HTTP requests [42]. The CGI.pm module is viewed as the standard tool for creating CGI scripts in Perl as it provides a simple interface for most common CGI tasks. However this module is very large. It is considered bloated [42] and it was decided to handle HTML output and the processing of GET and POST requests manually. The custom algorithm for processing these requests is presented in Algorithms 2 and 3. These algorithms parse HTTP headers and save the request parameters into a hash table for easy access.

Algorithm 2 Perl CGI GET request parser

```
sub populateQueryFields {
  %queryString = ();
  my $tmpStr = $ENV{ "QUERY_STRING" };
  @parts = split( /\&/, $tmpStr );
  foreach $part (@parts) {
    ( $name, $value ) = split( /\=/, $part );
    $queryString{ "$name" } = $value;
  }
}
```

Algorithm 3 Perl CGI POST request parser

```
sub populatePostFields {
  %postFields = ();
  read( STDIN, $tmpStr, $ENV{ "CONTENT_LENGTH" } );
  @parts = split( /\&/, $tmpStr );
  foreach $part (@parts) {
    ( $name, $value ) = split( /\=/, $part );
    $value =~ ( s/%23/\#/g );
    $value =~ ( s/%2F/\//g );
    $postFields{ "$name" } = $value;
  }
}
```

The NIAB system structure is presented in Figure 3.4. We can see that users interact with the web page user interface which communicates with the underlying Perl scripts via CGI. The Perl scripts manipulate the ipfw rulesets and dummynet pipes which in turn manipulate traffic passing through the kernel subsystem.

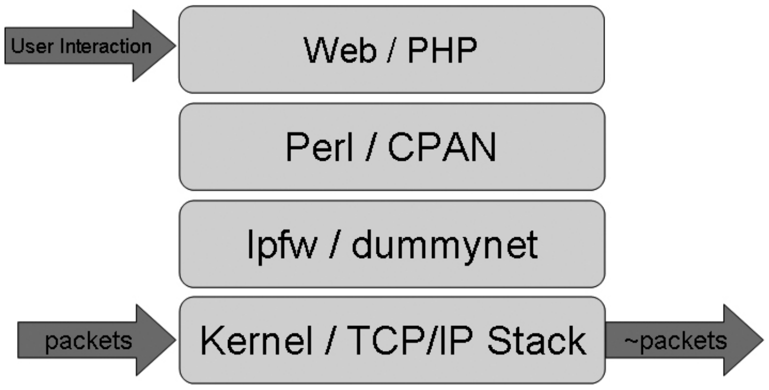


Figure 3.4: NIAB Structure

To connect to the NIAB web interface the user enters the IP address of the control interface into the web browser (in this example 146.231.121.140) and enters the password at the prompt. The main NIAB screen is displayed in Figure 3.5. From here the options available can be seen in the left hand section of the page.

3.2.1.6 User Account

From the "User Account" section the user can change the current login password. The NIAB system has only one user, who has control of the entire system. It is not seen as necessary to have multiple users. The user is required to enter the old password, and then the new one twice. This is displayed in Figure 3.6.

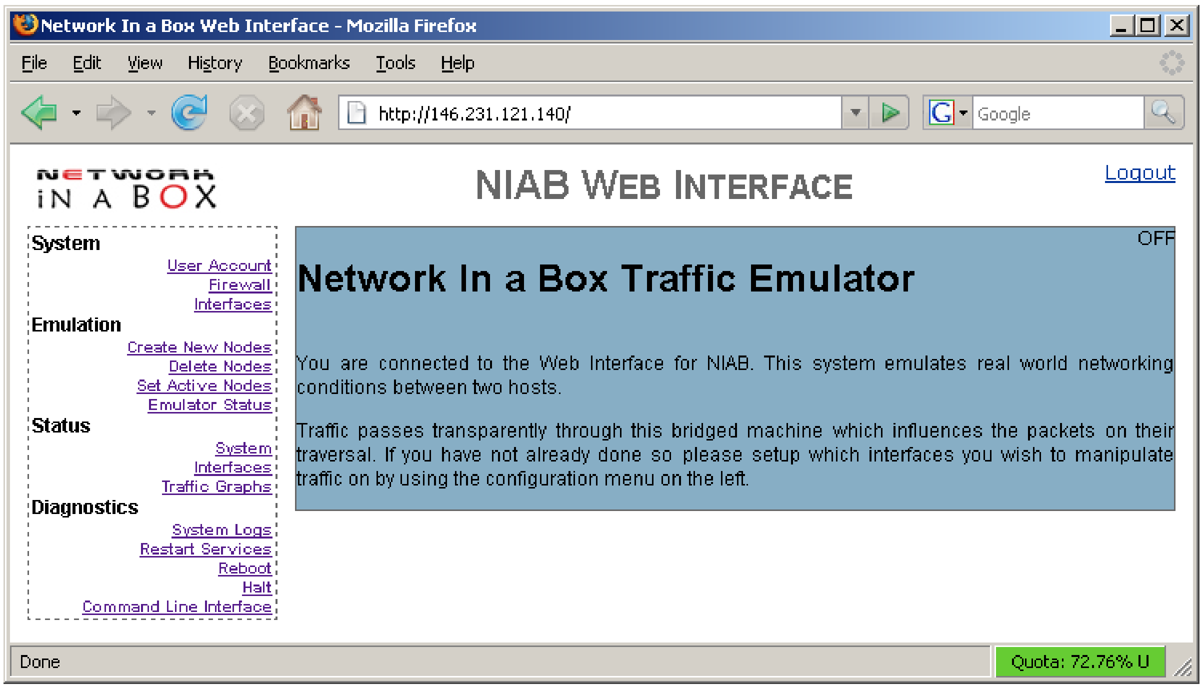
3.2.1.7 Firewall Settings

From the "Firewall" section the user can view, add and delete firewall rules which are unrelated to the NIAB system (rules from set 0). This screen is displayed in Figure 3.7. It can be noted that there are two rules in the ruleset, one which denies all udp traffic, and the default 65535 rule which allows unmatched traffic to pass.

In Figure 3.8 we demonstrate the ability to add rules to the default rule set. Here we add a rule to deny TCP traffic from the NIAB system to the default gateway in the current context. This has no relevance to the system and is purely an example.

3.2.1.8 Network Interfaces

In order to create a bridge the user selects two interfaces within the system which the bridge will make use of during operations. This is done via the "Interfaces" section dis-

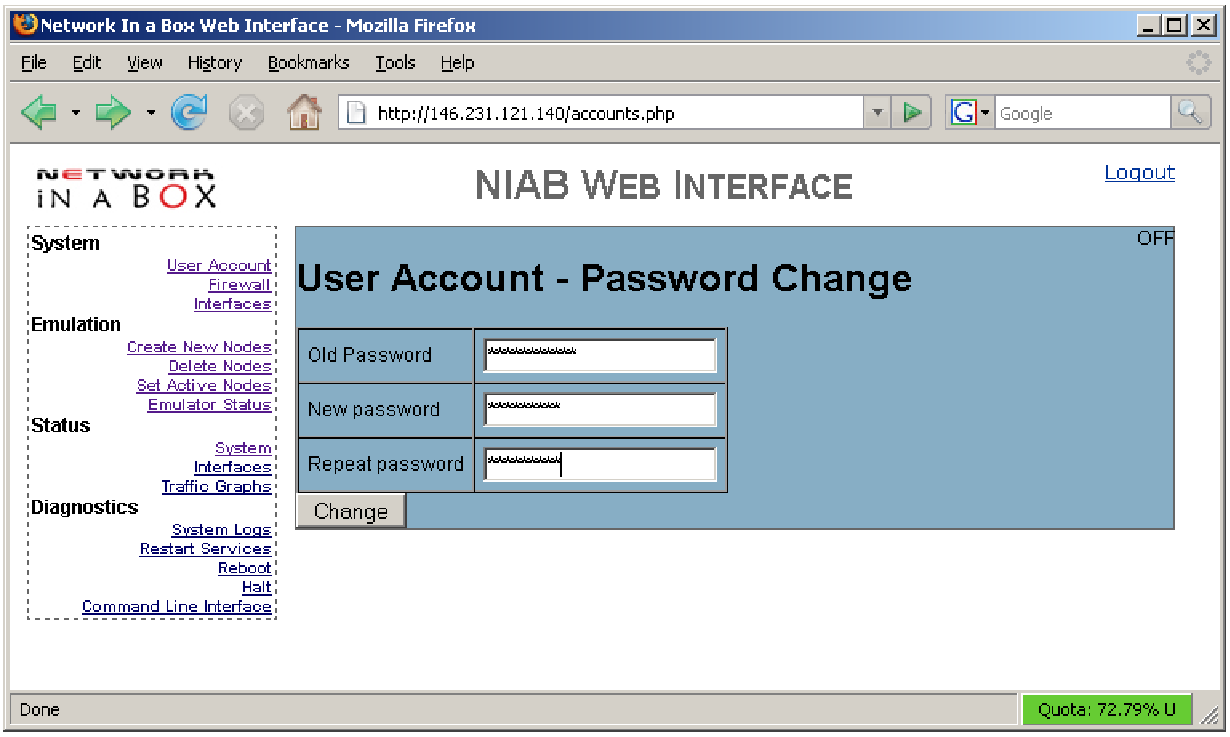


Network In a Box Traffic Emulator

You are connected to the Web Interface for NIAB. This system emulates real world networking conditions between two hosts.

Traffic passes transparently through this bridged machine which influences the packets on their traversal. If you have not already done so please setup which interfaces you wish to manipulate traffic on by using the configuration menu on the left.

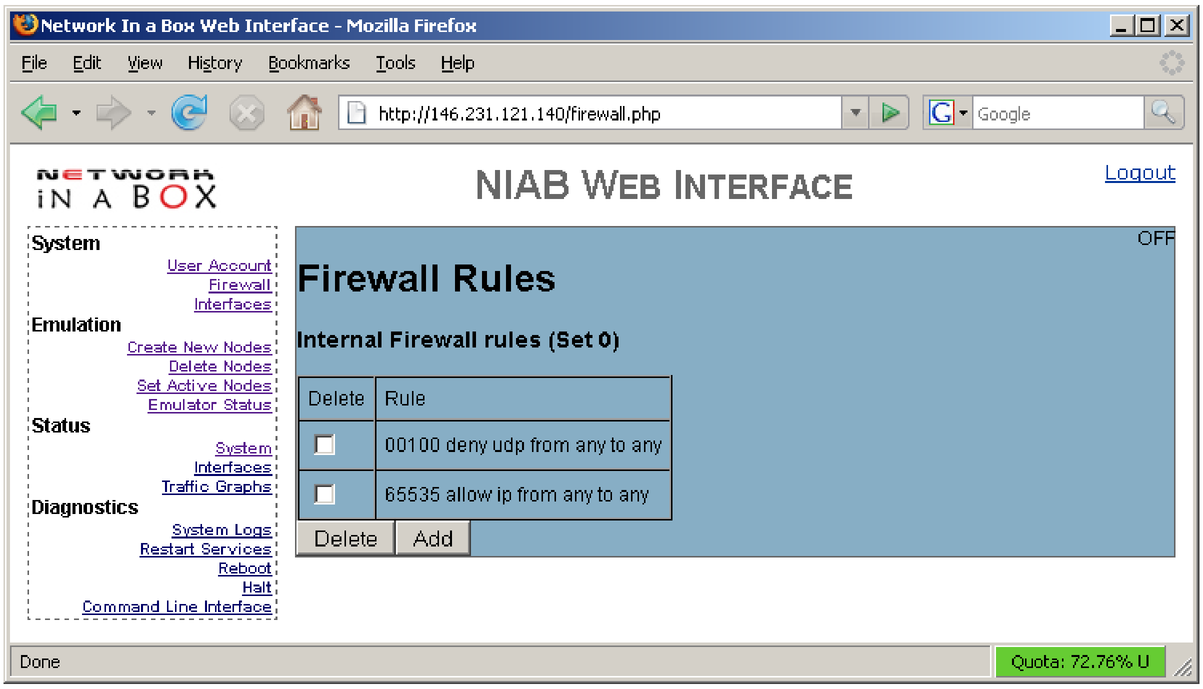
Figure 3.5: NIAB Main Screen



User Account - Password Change

Old Password	***********
New password	**********
Repeat password	**********

Figure 3.6: Changing the login password to NIAB

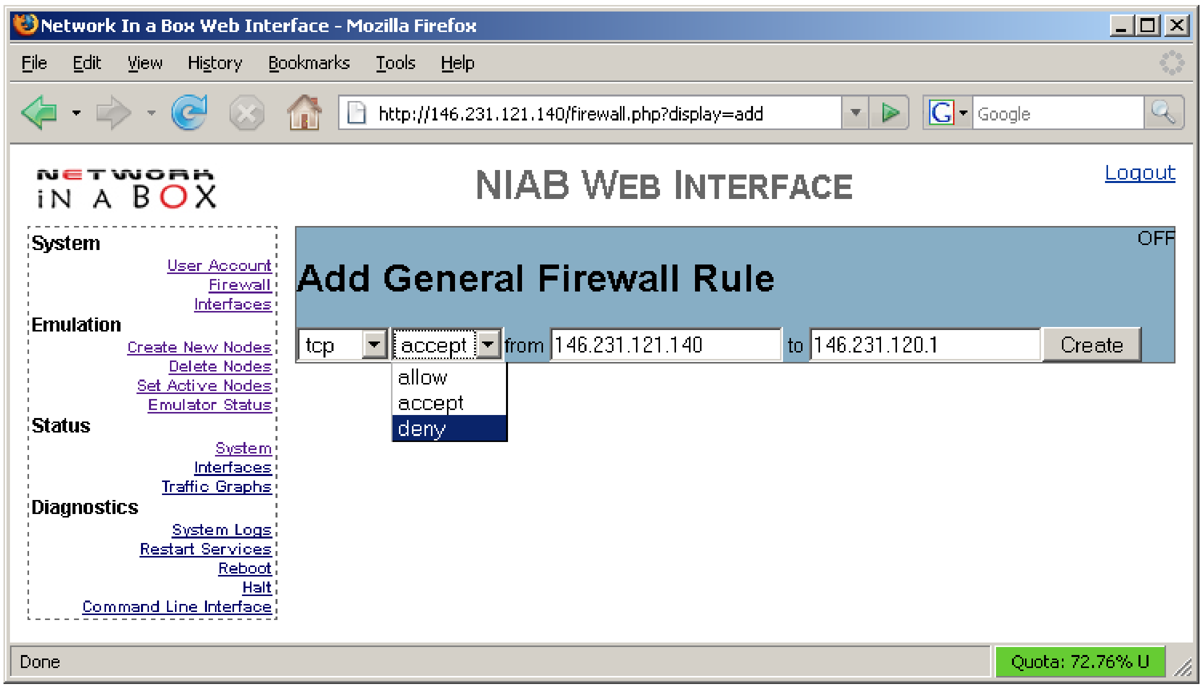


Firewall Rules

Internal Firewall rules (Set 0)

Delete	Rule
☐	00100 deny udp from any to any
☐	65535 allow ip from any to any

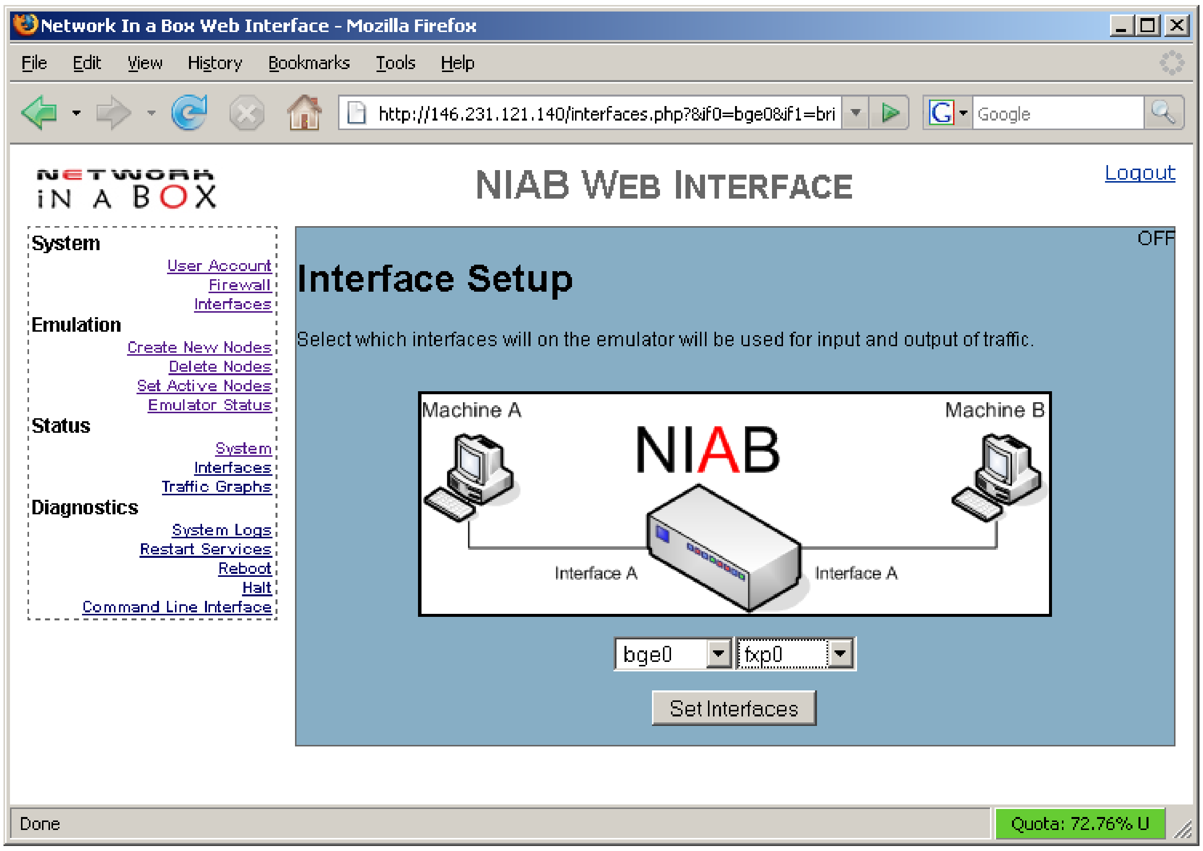
Figure 3.7: Viewing firewall rules



Add General Firewall Rule

tcp | accept | from 146.231.121.140 | to 146.231.120.1

Figure 3.8: Adding a firewall rule



Interface Setup

Select which interfaces will on the emulator will be used for input and output of traffic.

bge0 fxp0

Figure 3.9: Setting interfaces through which traffic will pass

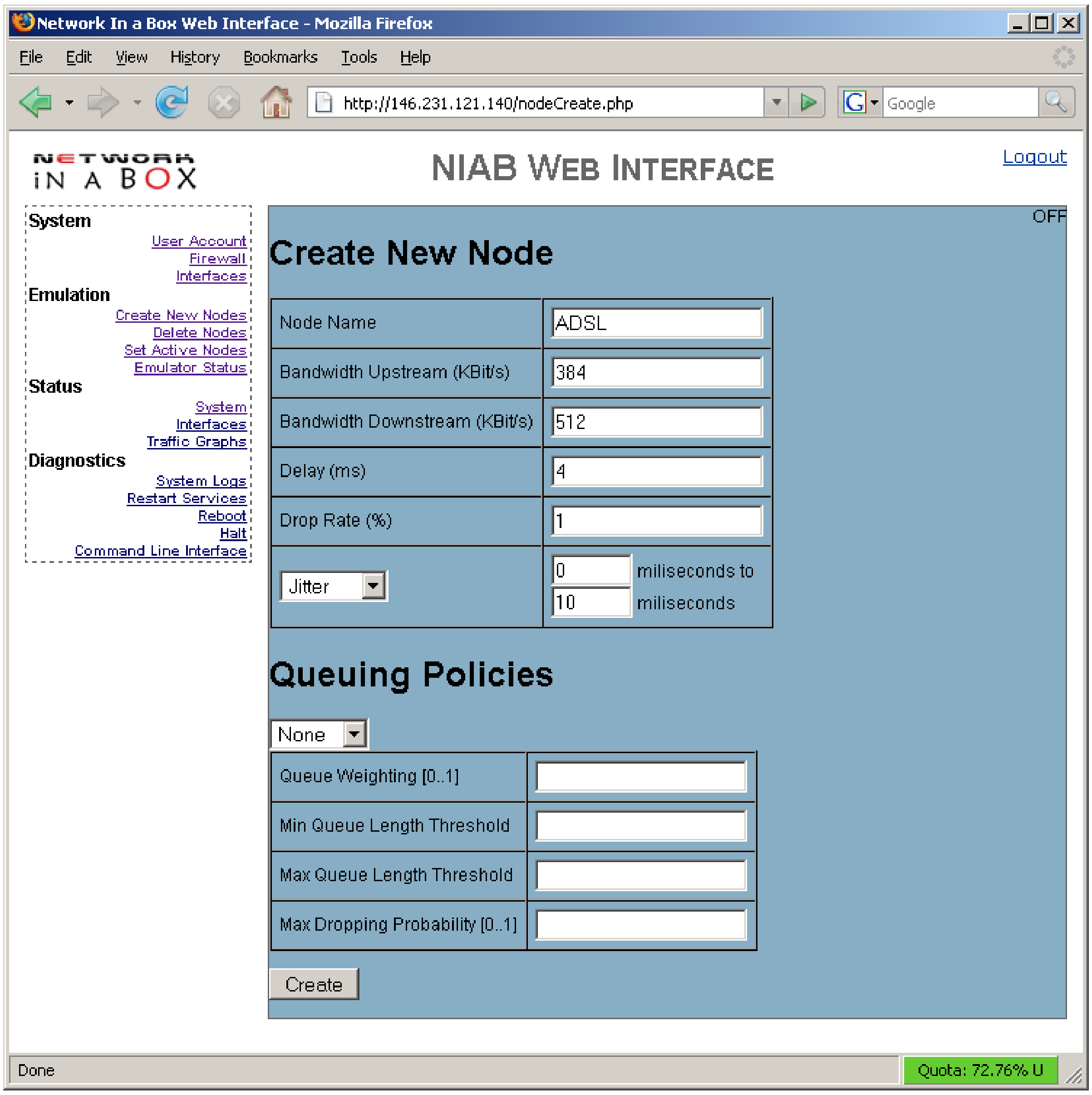
played in Figure 3.9. Here we select two interfaces (bge0 and fxp0) and set that traffic passing through them is manipulated.

3.2.1.9 Creating, Deleting and Setting Nodes

As mentioned in section 3.2.1.4, individual link properties are represented as nodes in the NIAB system. Each node has the properties of name, upstream and downstream bandwidth, packet delay, packet drop rate, jitter, and queuing policy options. In Figure 3.10 we create a node with the name "ADSL" with an upstream bandwidth of 384Kbit/s; a downstream bandwidth of 512Kbit/s; a delay of 4ms; a drop rate of 1% and jitter ranging from 0ms to 10ms. This results in the nodes.xml file being appended with a new link.

Nodes may be deleted by selecting the "Delete Nodes" option which gives a page as per Figure 3.11. In this example we select the "Satellite" and "Dial Up" nodes to be deleted. This will result in their XML information being removed from the nodes.xml file.

To select which nodes we wish traffic to pass through we select the "Set Active Nodes" option which brings up the node selection screen as displayed in Figure 3.12. All available



Create New Node

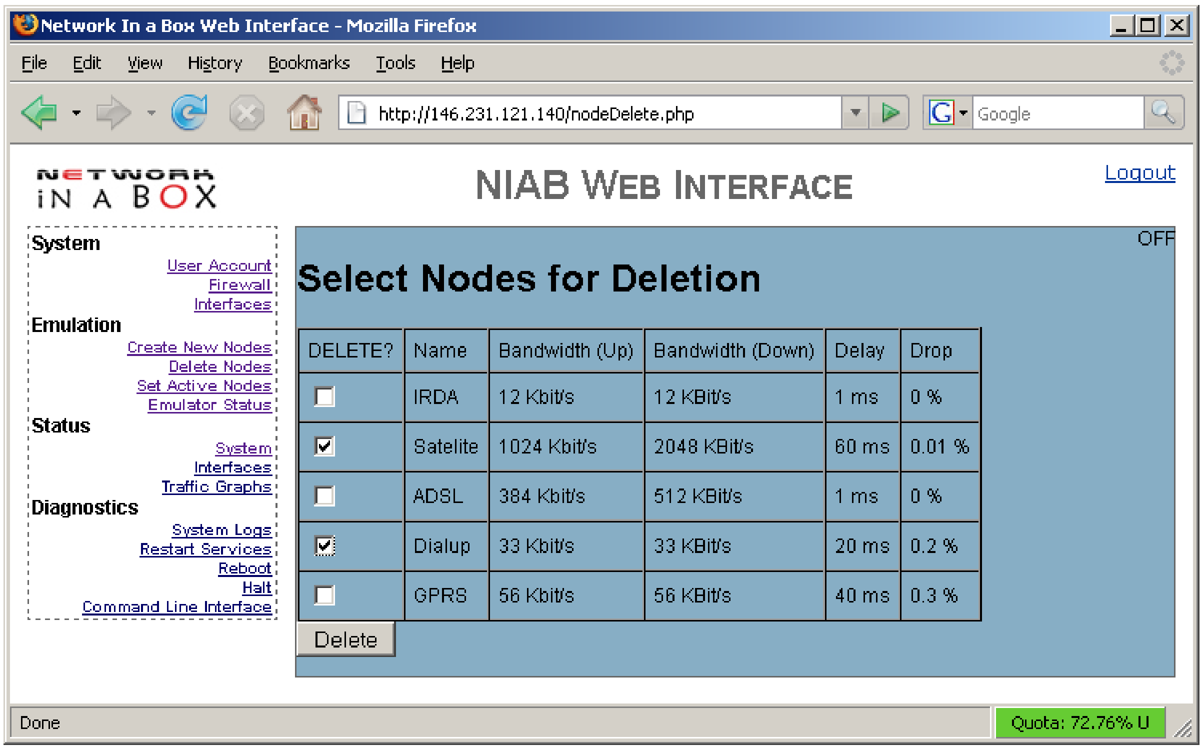
Node Name	ADSL
Bandwidth Upstream (KBit/s)	384
Bandwidth Downstream (KBit/s)	512
Delay (ms)	4
Drop Rate (%)	1
Jitter	0 miliseconds to 10 miliseconds

Queuing Policies

None

Queue Weighting [0..1]	
Min Queue Length Threshold	
Max Queue Length Threshold	
Max Dropping Probability [0..1]	

Figure 3.10: Creating a typical ADSL node



Select Nodes for Deletion

DELETE?	Name	Bandwidth (Up)	Bandwidth (Down)	Delay	Drop
☐	IRDA	12 Kbit/s	12 KBit/s	1 ms	0 %
☑	Satelite	1024 Kbit/s	2048 KBit/s	60 ms	0.01 %
☐	ADSL	384 Kbit/s	512 KBit/s	1 ms	0 %
☑	Dialup	33 Kbit/s	33 KBit/s	20 ms	0.2 %
☐	GPRS	56 Kbit/s	56 KBit/s	40 ms	0.3 %

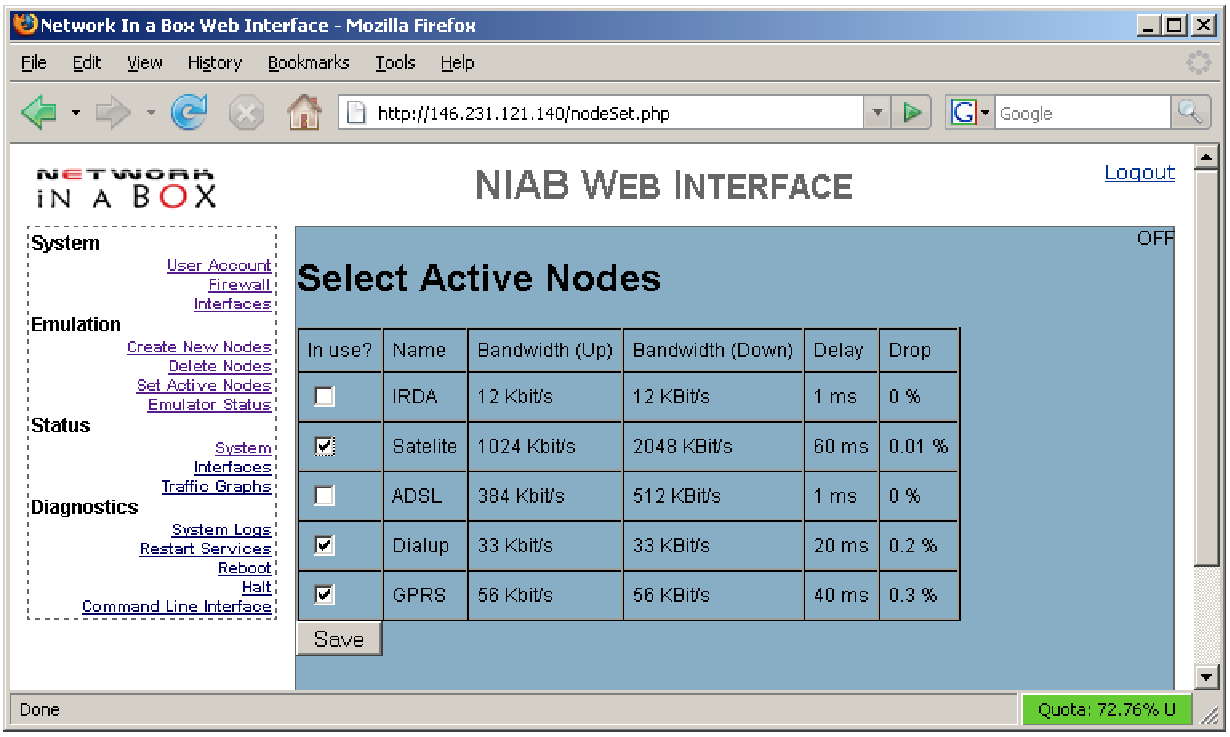
Figure 3.11: Deleting a NIAB node

nodes are listed, with a tick box next to each name. If a tick box is selected, traffic will pass through the link. In this example we select the "Satellite", "Dialup" and "GPRS" connections for traffic to pass through. This will result in the NIAB system checking to see if the emulator is currently "on". If it is, it will create rules and pipes as discussed in section 3.2.1.10.

3.2.1.10 Setting Emulator State

By default the network emulator is off. In the previous Figures (3.5 to 3.12) this state can be seen in the top right hand corner with the word "off". This state is saved in a text settings file. To toggle the state of the emulator we select the "Emulator Status" section, displayed in Figure 3.13. Turning the emulator on will result in the NIAB system parsing the nodes.xml file and identifying which nodes are set to "inUse". For each node it finds with this property, it will create a firewall rule to match its traffic. It will then create pipes to manipulate traffic with the nodes properties. Turning the emulator off results in the relevant rules and pipes being destroyed.

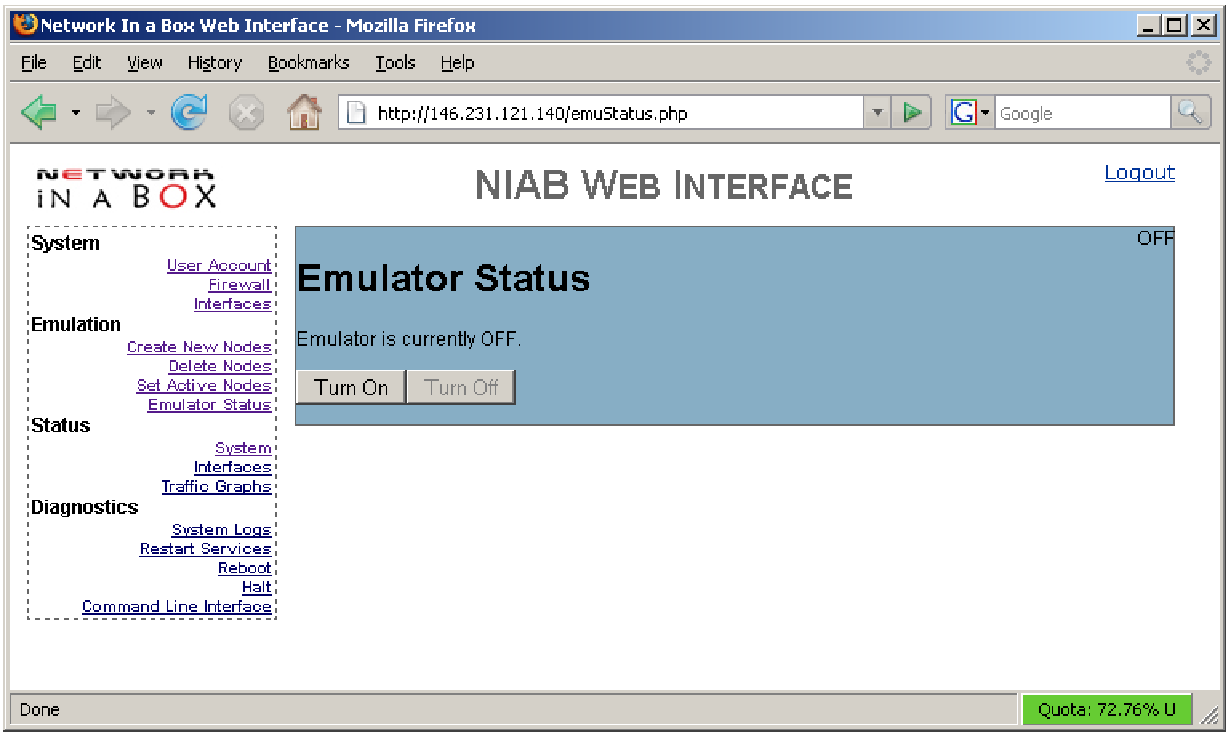
The remaining sections of the system are for general system information and use. The "System" section lists information about the NIAB host. "Interfaces" lists connection information about each network interface card, such as MAC addresses (i.e. information



Select Active Nodes

In use?	Name	Bandwidth (Up)	Bandwidth (Down)	Delay	Drop
☐	IRDA	12 Kbit/s	12 KBit/s	1 ms	0 %
☑	Satelite	1024 Kbit/s	2048 KBit/s	60 ms	0.01 %
☐	ADSL	384 Kbit/s	512 KBit/s	1 ms	0 %
☑	Dialup	33 Kbit/s	33 KBit/s	20 ms	0.2 %
☑	GPRS	56 Kbit/s	56 KBit/s	40 ms	0.3 %

Figure 3.12: Selected active nodes



Emulator Status

Emulator is currently OFF.

Figure 3.13: Setting the emulator status

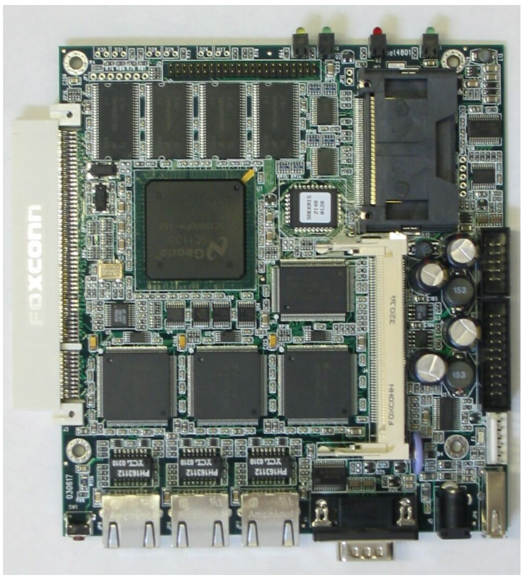


Figure 3.14: The Soekris net4801 embedded machine [11]

from the “ifconfig” command). “Traffic Graphs” is a listing of sample graphs as presented in the results section (Chapter 4) for the user’s general information.

In the diagnostics section “System Logs” displays the tail end of output from “/var/log/messages” which is where FreeBSD logs all system activity to. “Restart Services” will restart the Apache webserver reload kernel modules. “Reboot” simply calls the reboot command which reboots the FreeBSD machine, and “halt” simply turns the computer off. The “Command Line Interface” has not yet been completed to a safe and stable environment. It is foreseen that this option will give the user a simple interactive shell through which to manipulate the FreeBSD host system.

3.2.2 Phase 2 - Embedded Machine

3.2.2.1 Soekris net4801

The Soekris net4801 [11] is a compact, low-power, low-cost, advanced communication computer based on a 266 MHz class processor. It has three 10/100 Mbit Ethernet ports, 256 Mbyte SDRAM main memory and uses a CompactFlash module for program and data storage. It can be expanded using a MiniPCI type III board and a low-power standard PCI board. It has been optimized for use as a Firewall, VPN Router and Internet Gateway, but has the flexibility to take on a whole range of different functions as a communication appliance. The board is designed for long life and low power. It is depicted in Figure 3.14.

This embedded machine was seen as an optimum machine for alternative deployment due to its small size, low power consumption and unobtrusive form factor allowing maximum portability. In the previous section we discussed the development and deployment of our custom emulator on both a Virtual Machine, and a computer system and we now look to deployment on this embedded machine. A full listing of commands and scripts used to build the system are included in Appendix A.1. An overview of the emulator running on the Soekris machine is discussed in section 3.2.2.2. This process and the scripts are accredited to David Courtney whose detailed tutorial [18] enabled the building of the stripped down FreeBSD system.

3.2.2.2 Building a Mini FreeBSD System and Kernel for the Soekris

As mentioned in section 3.2.1.1 FreeBSD has been chosen as a deployment platform for our emulator. A 266Mhz processor and 256MB RAM on the Soekris machine is more than enough[1] to run FreeBSD (version 6.2 in our case), the Apache web server and our emulator. The custom operating system was created inside a jail. The FreeBSD jail mechanism [24] is an implementation of operating system-level virtualization that allows us to partition a FreeBSD-based computer system into several independent mini-systems called jails. A jail is normally used to separate services which run in a potentially dangerous environment, such as webservers, mail servers and DNS servers but in our context we shall use it to build a stripped down version of FreeBSD with limited libraries and binaries, and no documentation or other superfluous material. It is possible to strip down a FreeBSD installation to under 20MB.

Once inside our NIAB jail the directory structure was manually created and then the boot loader was rebuilt. Binary executables were then rebuilt followed by the compilation and installation of the associated libraries. Two scripts were then used to copy the binary files over to our emulator system. Certain files were noted as being necessary, and are listed in Appendix A.2.

The boot files were then configured, followed by the creation of a custom NIAB kernel. Most of the superfluous options from the GENERIC kernel were removed. The full kernel listing is available in Appendix A.3, but it is worth noting here that FreeBSD has kernel support for the Soekris Geode CPU (see Listing 6).

Libraries were then copied across to our new system followed by the population of the new /etc from the host systems /etc. Most of the files were unchanged, those that required

[1]The FreeBSD Manual [24] notes that the operating system will run on a 486 CPU with 24MB of RAM

Listing 6 Kernel support for the Soekris Geode Processor

```
options CPU_GEODE
options CPU_SOEKRIS
```

change are noted in the Appendix A.2.

In addition, it must be noted that the Soekris machine runs its CompactFlash in read only mode. This is because the card has a limited number of read erase cycles and it would quickly be destroyed if mounted read/write. Certain configuration files need to be edited on the fly. For example, using DHCP client (dhcpclient) needs to write lease information to /etc/resolv.conf. In our emulator system several run time configuration files are required to be edited too. The solution with such files is to symlink them to a file in /tmp (in RAM). For example, symlinking /etc/resolv.conf to /tmp/resolv.conf .

Extra modules and packages were added to the new system as discussed in section 3.2.1.1, namely as the Apache webserver, PHP5 (and associated modules), Perl (and associated CPAN modules) and several utilities.

The final step was to build the image to be written to the CompactFlash card. Building of the image is quite an involved process and is presented in detail in Appendix A.1. The final binary image could then be written to the CompactFlash card with the dd command. Such a card may then be inserted into the Soekris for booting. At the time of writing, we were successful in creating our stripped down custom image but unfortunately we were unable to successfully boot the Soekris machine.

3.2.3 Security Considerations

3.2.3.1 Code

One of Perl's most useful features is the idea of tainting. If you enable taint mode, Perl will mark every piece of data that comes from an insecure source, such as insecure input, with a taint flag. If you want to use a piece of tainted data in a potentially dangerous way, you must untaint the data by verifying it. This is especially useful when CGI is being used, where input is passed from web forms where potentially malicious code could be inserted. At the time of writing not all the code passes the taint test, but the theory of tainting is included here for completeness.

3.2.3.2 System

A basic login facility is present in the NIAB system. A user may only access and manipulate the system if they possess the correct credentials. This was implemented using Apache's .htaccess control.

3.2.4 Summary

This ends the chapter on the design, implementation and deployment of our custom network emulator. We discussed the design principles involved in the creation and deployment of our system as well as the programming methodologies. Next, the two stages of implementation and deployment were examined; Virtual Machine/computer followed by the Soekris net4801. At the end of "Phase 1" the code developed for the system was discussed. It involved the combination of PHP, Perl and their connection via CGI.

The creation of the image for the Soekris was quite complicated and involved and the reader is encouraged to examine the appendices. The next chapter (Chapter 4) discusses the tests conducted on the network emulator, together with the corresponding results.

Chapter 4

Testing and Results

Once development of NIAB was complete significant testing was required to ensure traffic was being manipulated in the desired manner. Several tools were used to create graphs demonstrating various characteristics of traffic flow. The process was as follows; traffic passing through the NIAB machine was captured and dumped to a hard disk with the tcpdump [5] utility. Tcpdump allows the interception and saving of TCP/IP packets being transmitted or received over a network interface. From this raw data, statistical information was generated with the tcptrace [39] utility. Tcptrace can produce several different types of output containing information on each connection seen, such as elapsed time; bytes and segments sent and received; retransmissions; round trip times; window advertisements; throughput; and more. This statistical information can be graphed using tools such as xplot [43] (or JPlot [8] or GNUplot [10]) and 'R' (The R Project for Statistical Computing) [23]. Several scripts were developed in Perl to accomplish automatic and accurate testing/timing and are presented in Appendix C.

The test procedure is outlined in Figure 4.1, and the pseudo code for an individual test is presented in Algorithm 4. The script presented in algorithm 4 executes and creates nodes within the emulator with configurations for bandwidth, delay, packet drop, jitter and multipath, depending on which test is being run. Test scripts exist for testing specific properties individually. It then collects raw tcpdump data as traffic flows through the emulator system, which is then passed onto tcptrace which generates statistical information from the dump file. This statistical information is passed through a custom NIAB script which retrieves information relevant to the scenario we are working on. This is finally passed onto the R graphing tool which produces the desired graphs.

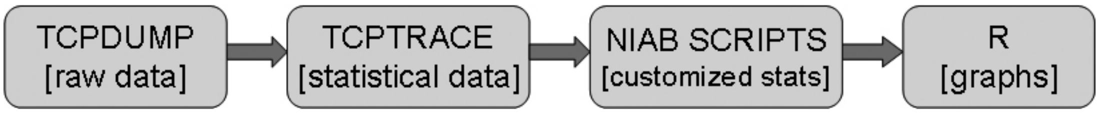


Figure 4.1: Process of testing NIAB system

Algorithm 4 Pseudo code for script to test emulator

```
create pipe
fork(tcpdump)
foreach @criteria
  emulate criteria
  sleep delay
terminate(tcpdump)
tcptrace -T -zyx -A10 -o1 file
sanatise(file)
R(file) <- graph
```

4.1 Kernel Frequency Variations

As mentioned in section 3.2.1.2 the default FreeBSD kernel is set to 100Hz implying a granularity of 10ms, and the emulator performs its task once per timer tick. NIAB's kernel was set to have a frequency of 10000Hz for an accurate simulation of high data rates. The affect of this change is noted in Figures 4.2 and 4.3. It can be seen that in Figure 4.3 with the 10000Hz the bandwidth throttling settles faster than in Figure 4.2 with 100Hz. This resulted in the NIAB system being more accurate for manipulating traffic with high data rates.

4.2 Results - Single Links

Several areas were tested and examined for simple configurations emulating one link. One pipe is used to emulate a single link in all cases except in section 4.2.3 where jitter is discussed, and section 4.2.5 where multipath effects are discussed. The results of the NIAB system for a single link are discussed in this section. The section covers each of the network variables tested for, beginning with throughput, and then round trip time, jitter, packet retransmission, multipath effects and lastly queuing policies.

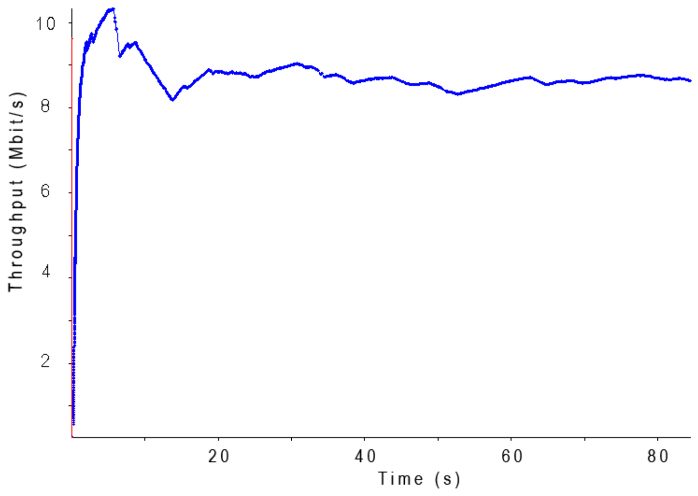


Figure 4.2: Kernel set to 100Hz

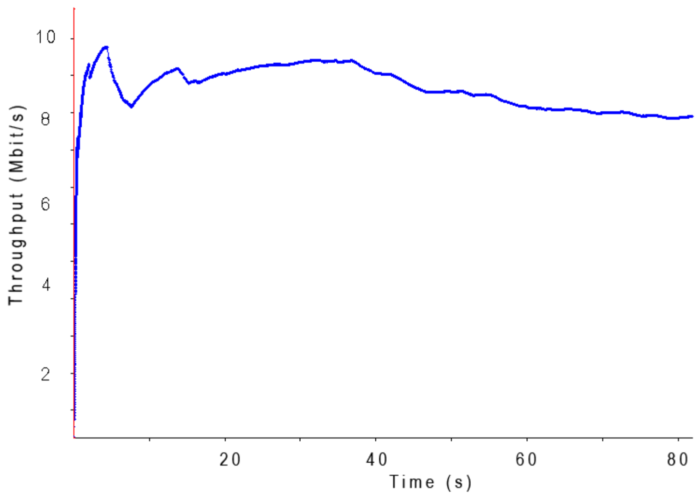


Figure 4.3: Kernel set to 10000Hz

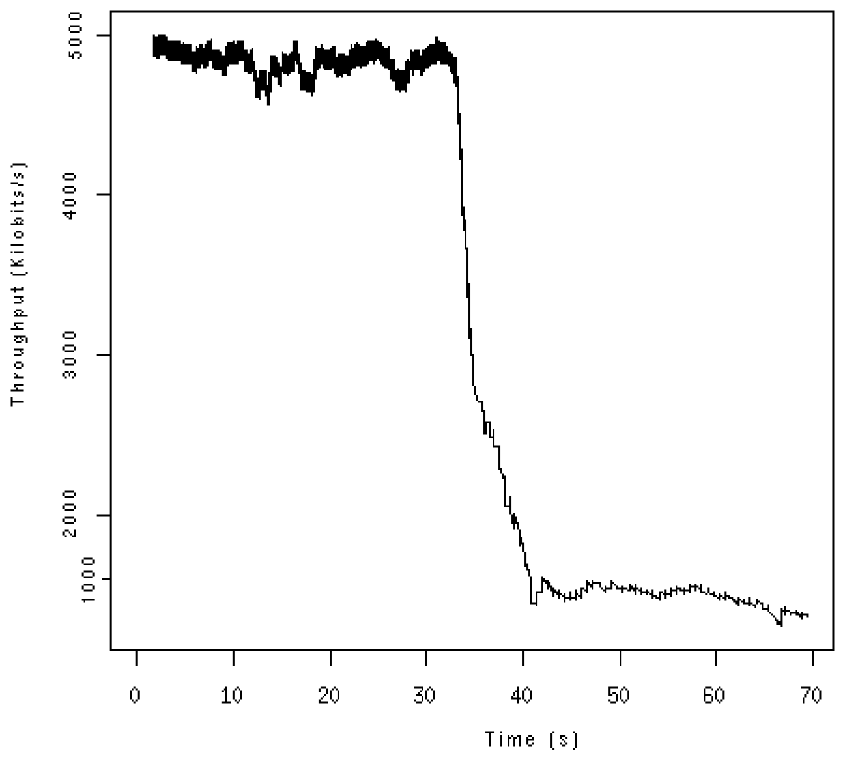


Figure 4.4: Variation of bandwidth from 5000KBit/s to 1000Kbit/s

4.2.1 Throughput Over Time

Throughput can be defined as the amount of traffic to pass through a system over a certain period of time. In Figure 4.4 we demonstrate initially setting the emulator at 5000Kbit/s and then dropping it to 1000Kbit/s. It can be noted that the drop off rate is rapid and settles around 1000Kbit/s.

As a demonstration of finer granularity in Figure 4.5 we initially set the bandwidth of the emulator to 512Kit/s (a typical DSL downstream speed) and then drop it to 384Kbit/s (a typical DSL upstream). Although not as obvious as with Figure 4.4 it can still be noted the rapid response time and settling of the curve.

4.2.2 Round Trip Time

Round Trip Time (RTT) between two points is defined as the total elapsed time for traffic to travel to the end network point and back again. In regards to TCP communication

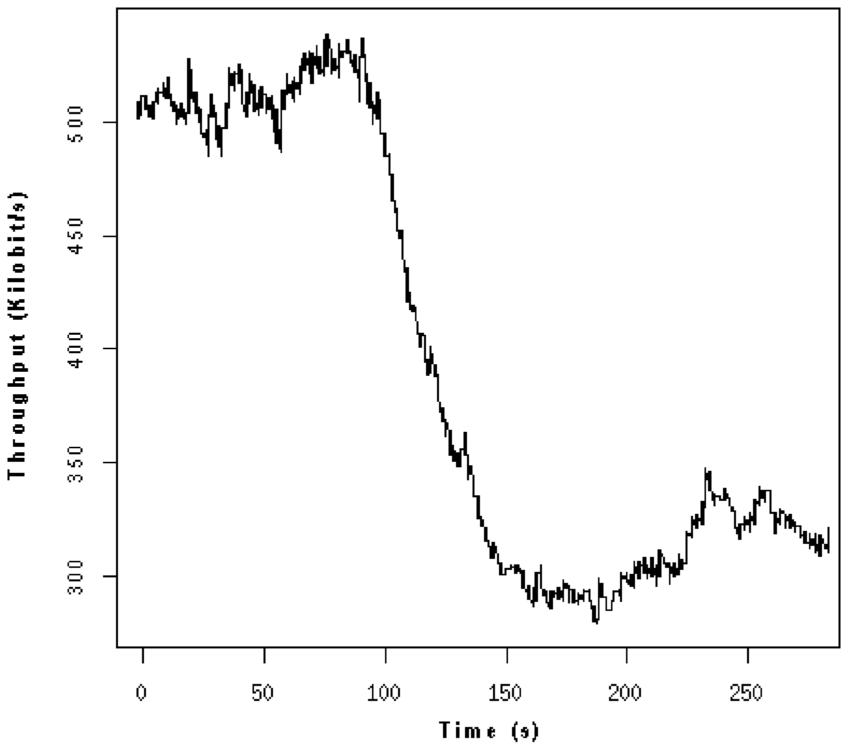


Figure 4.5: Variation of bandwidth from 512Kbit/s to 384Kbit/s

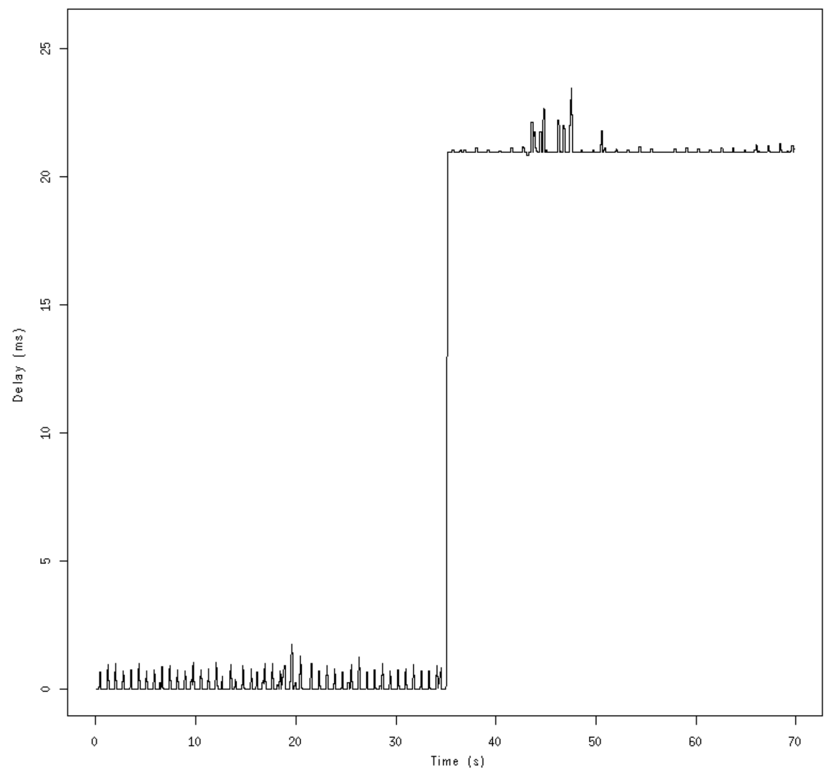


Figure 4.6: No initial delay followed by a 20ms delay

the RTT time is calculated from the 3-way handshake by measuring the time between segment transmission and ACK receipt [2]. In Figure 4.6 traffic initially has no delay, and then a delay of 20ms is incurred. From examining the graph in Figure 4.6 we can see the steep rise in the delay, at time = 35s. The Y-axis represents RTT in milli-seconds and the X-axis represents time. The line represent RTT samples calculated from non-retransmitted segments. Since we are only concerned with a single pipe at this stage, the RTT is the delay on the incoming connection. The outbound delay is negligible and can be disregarded.

In Figure 4.7 we present traffic incurring a 25ms delay, followed by a 10ms delay and finally no delay. Figure 4.7 demonstrates the emulator's ability to delay packets in a constant manner, as well as its ability to rapidly committ changes.

4.2.3 Variable Round Trip Time (Jitter)

Jitter is the measure of the variability over time of the latency across a network. It is similar to section 4.2.2 concerning RTT delay with the exception that the delay is varied

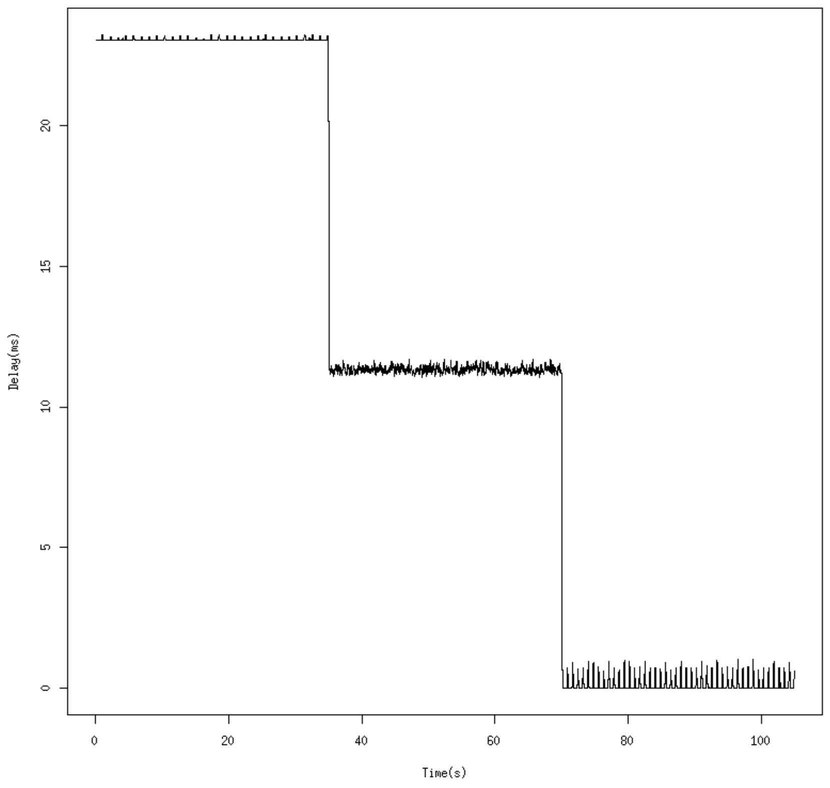


Figure 4.7: 25ms delay, followed by 10ms delay, followed by no delay

Pipe	Probability of match (%)	Delay (ms)
1	10	25
2	40	20
3	50	15

Table 4.1: NIAB Jitter with three pipes

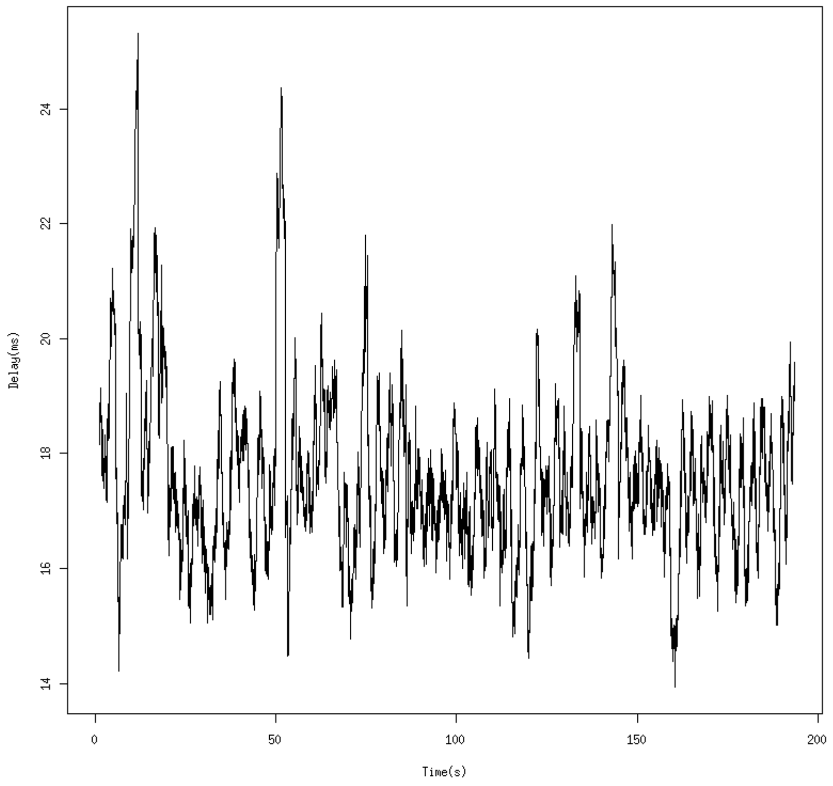


Figure 4.8: NIAB Jitter as per Table 4.1

at a relatively high and unpredictable frequency. This is a very common condition on slow network connections. A simple method for creating jitter is to create many ipfw rules, each with a certain probability of matching such that the sum of the probabilities is 100%. A separate pipe is associated with each rule bearing individual delay times. This is a simple, but effective way to create jitter. The other option is to dynamically create jitter. This would mean varying a single pipe from an external program, which would complicate matters [25]. Another possible option for creating jitter is to edit the dummynet.c source code. Whilst this option was examined, it was seen as excessive.

In Figure 4.8 we present the results of having three pipes setup with the characteristics displayed in Table 4.1. It can be seen that the delay fluctuates between the set values.

Pipe	Probability of match (%)	Delay (ms)
1	90	0
2	10	30

Table 4.2: NIAB Jitter with two pipes

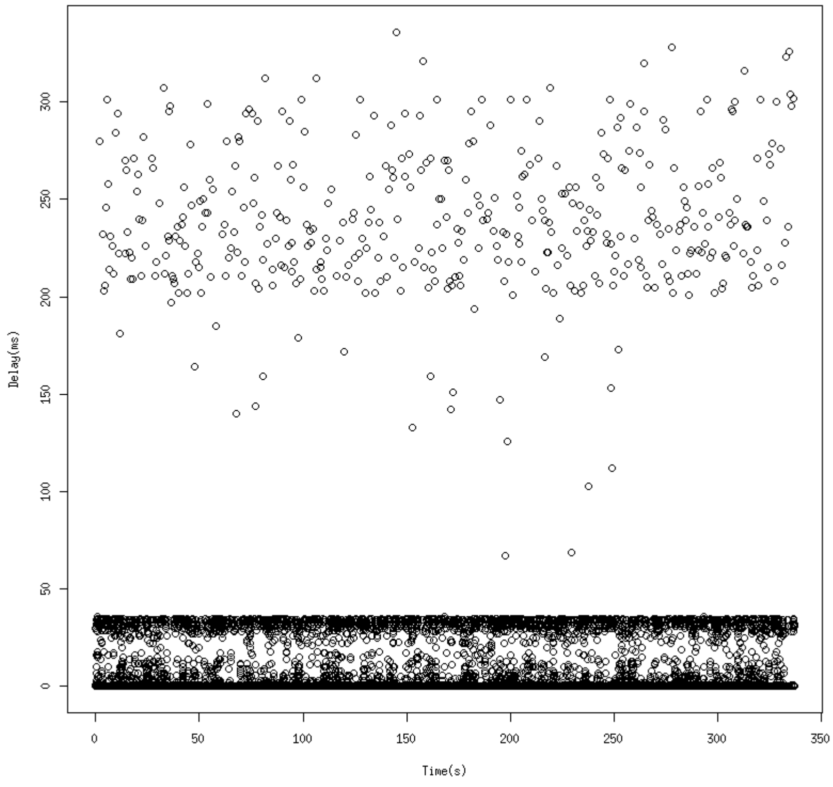


Figure 4.9: NIAB Jitter as per Table 4.2

As a further example we demonstrate a setup in which the jitter is set as per Table 4.2 and present results in Figure 4.9. In this situation it is possible to see that with a probability of 90% of traffic having no delay the majority of the points were around the 0ms delay, with a much less dense population scattered around the 30ms mark. In fact, examining the raw data of the experiment presented in Figure 4.9 we noted that a total of 38305 readings were taken in which 4014 had a delay greater than 0ms, which is 10.48%. In Figure 4.9 it can be noted that there is a spread of outliers between the 150ms and 300ms mark. These outliers number only 270 in total, and constitute less than 1% of the total readings. Such phenomena are not uncommon in situations such as this and may be accounted for by one of many factors including operating system granularity (on the fault of the emulator) or tcpdump/tcptrace's measurements (on the fault of the measuring tools).

Listing 7 Packet loss set to 20%

```
$ ping 146.231.123.106 -c 50
PING 146.231.123.106 (146.231.123.106) 56(84) bytes of data.
64 bytes from 146.231.123.106: icmp_seq=1 ttl=127 time=0.439 ms 64
   bytes
from 146.231.123.106
[....]

--- 146.231.123.106 ping statistics --- 50 packets transmitted, 38
received, 24% packet loss, time 49003ms rtt
min/avg/max/mdev = 0.247/0.373/0.709/0.086 ms
```

4.2.4 Packet Retransmits from Packet Loss Rate (PLR)

It is possible to set the emulator to randomly drop packets. This is a probabilistic function in which we specify the percentage of packet loss required. In Listing 7 the NIAB system has been set to drop 20% of all packets, and as can be seen is dropping approximately 24% of the 50 packets that were transmitted.

4.2.5 Multipath effects

With the NIAB system it is possible to emulate traffic having the option of taking several routes between two end points, such as presented in Figure 4.10.

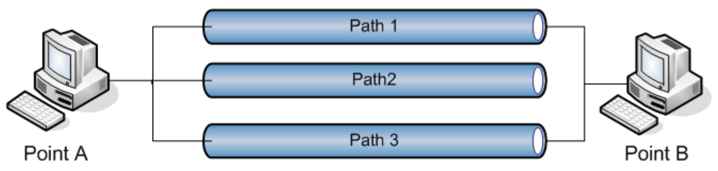


Figure 4.10: Traffic has the option of traversing different routes between node A and B

This is still represented as a single link as only *one* of the possible pipes is traversed by packets at any given point in time. This is accomplished by using several rules and pipes and a probabilistic match. It is a similar technique to that discussed in section 4.2.3 concerning jitter, except that now the pipes have more characteristics than just delay and can represent individual links (multiple links are to be discussed in section 4.3). Table 4.3 shows the results of a simple setup in which three possible routes exist between the two end points. One route has a delay of 20ms, one route drops 5% of packets and the third

Route	Property	Percentage of Hits
1	20ms delay	33.47%
2	5% drop rate	31.45%
3	Not modified	35.08%

Table 4.3: Multipath effects with NIAB

Listing 8 Random Early Detection with ipfw and dummynet

```
#ipfw pipe 1 config bw 384Kbit/s delay 2ms plr 0.01 red 0.5/4/18/0.8
```

route has no conditions applied to it. In this particular situation each route holds equal probability. The number of "hits" on each pipe are displayed in Table 4.3.

The outcome of this setup is demonstrated in Figures 4.11 and 4.12 where we note the throughput and delay times between the two "end points" fluctuating a significant amount, as traffic traverses each of the links mentioned in Table 4.3 approximately one third of the time. The throughput is a useful measurement for the control case of route 3. Delay and drop rate affect possible throughput by holding packets in the queue, and by requiring retransmissions of dropped packets. It can be seen that the throughput peaks at the control value of 1600000 bytes per second (12 megabit/s) and drops to just under 800000 bytes per second (6 megabit/s) from the effect of delays and drop rate.

At the time of writing the multipath ability had not been fully integrated into the web based user interface. These results are a demonstration of the back end capabilities.

4.2.6 Queuing Policies

As discussed in section 3.2.1.1 there are different queuing options available in the NIAB system. The default method is the Drop Tail, but the NIAB system supports the specification of the RED and GRED links.

There are four criteria which affect the queue management policy; weighting of the queue, minimum queue length threshold, maximum queue length threshold and the maximum queue probability. This is demonstrated in Listing 8 which illustrates an ADSL outbound connection with a queue weighting (w_q) of 0.5, maximum queue probability (max_p) of 0.8, and minimum and maximum thresholds of 4 and 18 slots respectively.

Using active queue management such as RED or GRED emulates the effect real world routers have on traffic flows, resulting in reduced congestion.

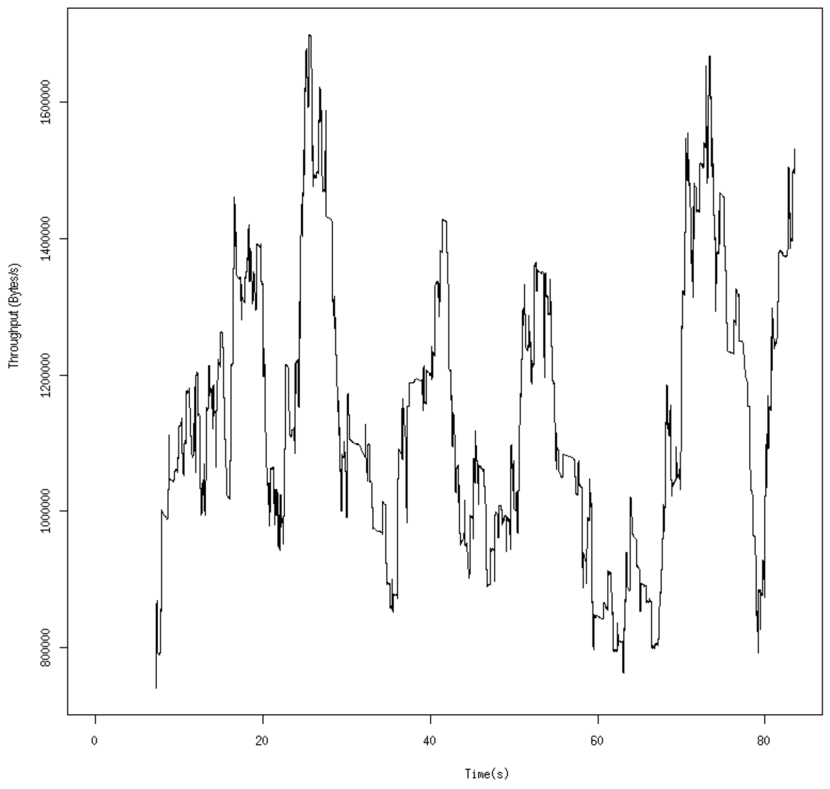


Figure 4.11: Throughput between two points with three possible paths between them

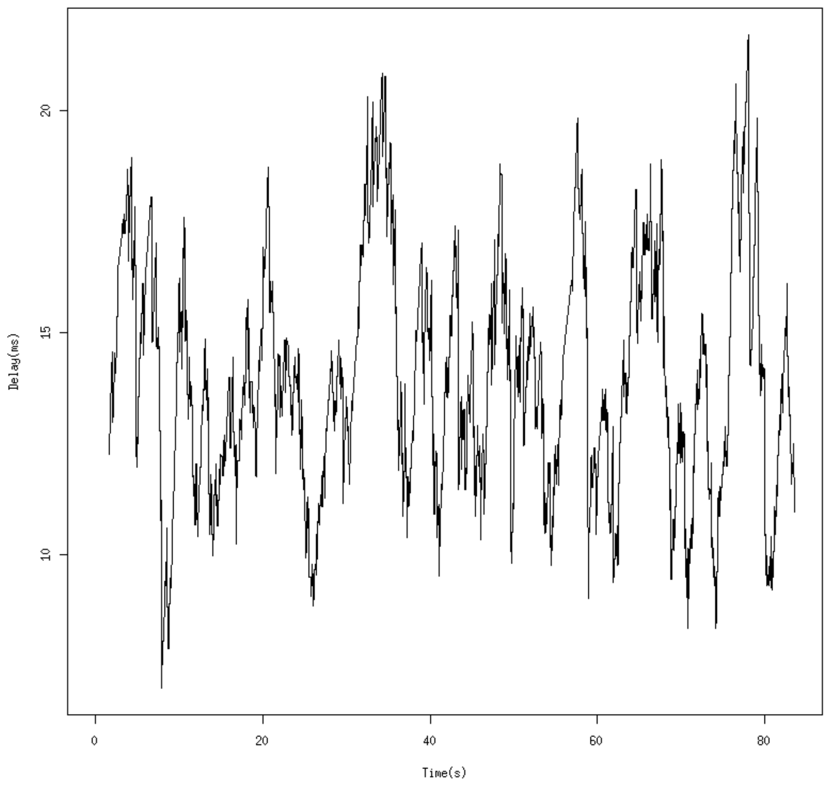


Figure 4.12: RTT between two points with three possible paths between them

Name	Upstream BW(KBit/s)	Downstream BW(KBit/s)	RTT Delay (ms)	Loss(%)
ADSL	384	512	5	1
Dialup	56	56	20	3
Satellite	1024	2048	200	0

Table 4.4: Typical Connection Properties

4.3 Results - Multiple Links

We present a scenario in which we chain several queues with multiple properties together to create the effect of traffic traversing a number of different network nodes. A set of typical connections and their associated properties are presented in Table 4.4.

In Figures 4.13 and 4.14 we demonstrate inserting an ADSL, satellite and dialup connection between the two “end points” of the network. Figure 4.13 represents upstream throughput. The downstream results were of the same nature because the links are throttled at the dialup connection’s 56Kbit/s throughput in both directions. Figure 4.14 demonstrates the RTT delay through the links. It is noted that the the delay points are centered around 220ms, with less than 1% of outliers having a delay greater than 300ms. It can therefore be noted that traffic incurrs the lowest common factors of the bandwidth settings and the summation of the delays.

4.4 Summary

From the statistical information and graphs presented we conclude that the emulator sufficiently represents real world network phenomena to a satisfactory degree as is required for general testing purposes of network dependent applications. These phenomena include bandwidth limitations, standard packet delay (RTT) and jitter (variable delay), random packet drop, multipath effects and various active queue management schemes.

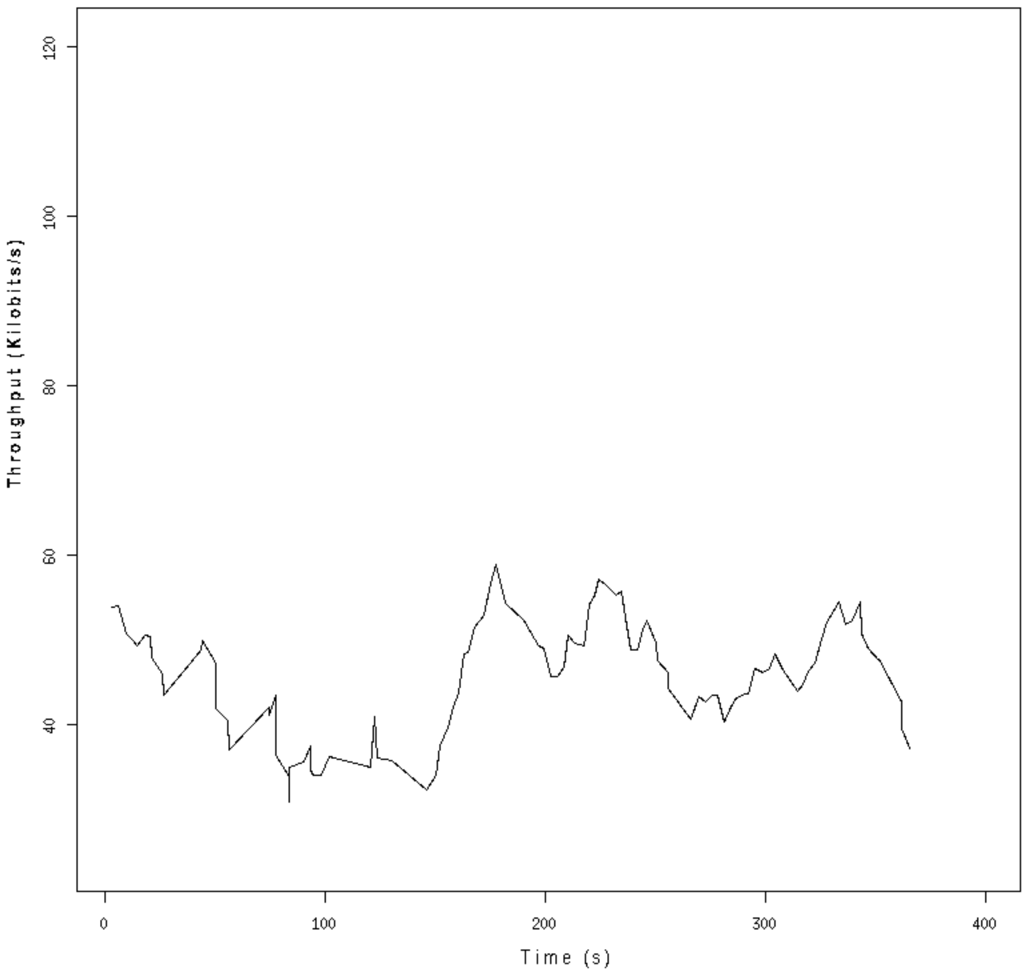


Figure 4.13: Upstream Throughput: A –> ADSL –> Dialup –> B

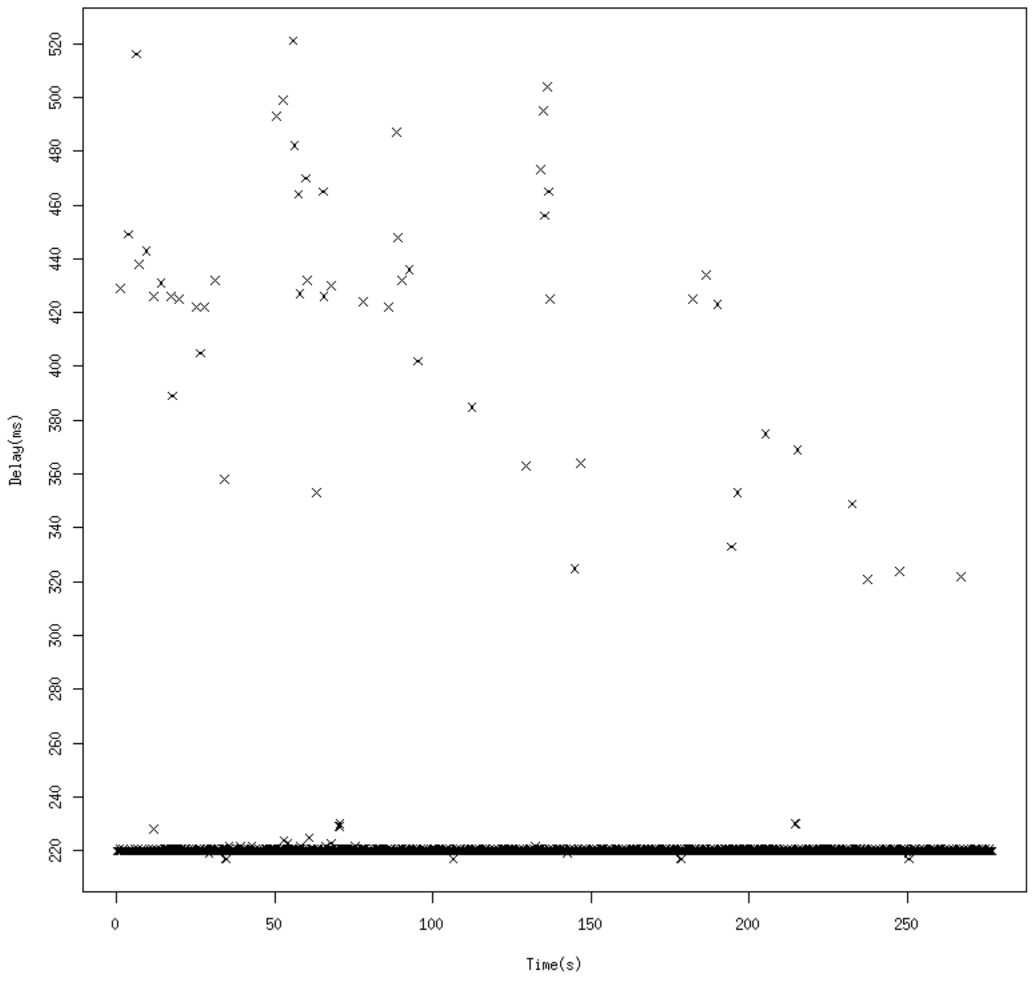


Figure 4.14: Delay: A –> ADSL –> Dialup –> B

Chapter 5

Conclusion

At the outset of this paper we decided to create a network emulator which could be deployed in several scenarios. Through a process of combining existing tools, creating front and backend scripts and deploying a custom stripped down version of FreeBSD we have achieved the original goal of this paper. We have a Virtual Machine image of the emulator and it has been tested on a standard computer. An image has been created suitable for deployment on an embedded machine such as the Soekris machine. We were however, not successful in deploying the image to the embedded machine in a stable, bootable manner.

5.1 Summary

In chapter 1 we introduced the idea of building a network emulator to be deployed in an academic environment and examined some background information relevant to this area. In chapter 2 we discussed work related to the field of networking and emulation. General networking terms and real world networks were discussed followed by an investigation into how network conditions affect packets. We then presented a literature review which initially discussed network emulation and how emulators fit into the OSI stack. Emulators were discussed in some detail and techniques to emulate network conditions were considered. These conditions included packet delay, bandwidth limitation, random packet loss, jitter and multipath effects. From here the literature review directed its attention towards simulation. Network simulators are typically discrete event-based systems which create an artificial representation of time. The simulation is based on a mathematical model and does not manipulate or use real network traffic. We can define simulation as a synthetic environment for running representations of code, whilst emulation relates to live testing in a real environment for running real code. After this the hybrid model of emulator/simulator was discussed which links a discrete simulator to an emulator allowing real traffic to be passed to the simulator from the emulator, and then passed back to the emulator after performing simulation calculations on the traffic. Hardware emulation and testbeds were then discussed and a comparison of the techniques was presented. The findings from this discussion were that the most viable, modern, small to medium scale option for testing network applications is the hybrid model incorporating emulation with aspects of simulation, although it was noted that a combination of techniques during the development of an application or protocol is desirable as each method has advantages and disadvantages, and is appropriate at a different time in the development cycle. Chapter 2 concluded that for the purposes of building an emulator in the academic domain, a hybrid model of emulation and simulation was the most viable and desirable solution. Under the constraints of cost and administration, as well as the low anticipated traffic volume and no need for absolutely precise manipulations this option was concluded to be more than acceptable.

In chapter 3 we examined the design and implementation of the custom emulator and we presented the details of its construction. This chapter considered two phases of the emulators' construction; phase 1 described the initial development and testing of the emulator on both a Virtual Machine and a standard computer and phase 2 discussed its possible deployment on an embedded machine. The chapter discussed and justified the choice of operating system (FreeBSD) for the emulator. This choice was made as FreeBSD can be stripped down to a small, lightweight size which is suitable to a system which has only to pass traffic between interfaces and apply simple manipulations to them. Furthermore, the FreeBSD ipfw firewall system along with dummynet and bridge components provided a solid infrastructure for the emulator. The chapter went on to describe ipfw and dummynet and their role in the emulator system. From here we moved onto a section discussing the back end scripts (written in Perl) which were used to manipulate ipfw rulesets and dummynet pipes. The use of storing node information in XML files was discussed and presented. Phase 1 of the development of the emulator in Chapter 3 closed by presenting the web based user interface with numerous screen-shots and a discussion on the technology used to build them; primarily PHP and CGI. Phase 2 of the building of the emulator was discussed next. Here we looked at the possibility of deploying the emulator on a Soekris embedded machine. The Soekris net4801 is a compact, low-power computer based on a 266 MHz class processor. It has three 10/100 Mbit Ethernet ports, 256 Mbyte SDRAM main memory and uses a CompactFlash module for program and data storage. The construction of a custom, stripped down kernel was then presented. Chapter

3 concludes by noting that the emulator was successfully deployed in a virtual machine and on a computer, and that the construction of an image suitable for deployment on such embedded machines as the Soekris was successful, but at the time of writing we were unsuccessful in loading the image onto the embedded machine in a stable manner.

Tests and their corresponding results were presented in chapter 4. First the testing procedure was discussed which involved passing a large volume of traffic through the emulator (with the emulator's network conditions set) whilst running the tcpdump tool to capture raw information about the traffic. Tcptrace was used to generate statistical information from the tcpdump output such as bandwidth, delay times and packet drop information. From here the tcptrace output was passed through several custom scripts extracting information relevant for our tests. This information was plotted using the R Statistical Graphing tool. Throughput, round trip time, jitter and multipath and multiple link graphs were discussed and presented. Furthermore, a discussion on active queue management was presented. Chapter 4 concludes noting that the results observed correlate with the various conditions set within the emulator system.

5.2 Problem Statement Revisited

The problem which we originally addressed was that of developing a network emulator to manipulate traffic as it passes through the emulator, primarily as a means of testing network dependent applications. By merging existing tools and technologies we have developed the network emulator. It has the functionality to delay traffic in a constant and variable manner (jitter), throttle the bandwidth, drop packets and to emulate multipath effects. Furthermore, we aimed for this emulator to be as discrete, lightweight and portable as possible. To this end NIAB has been deployed and tested in a bridged, transparent manner allowing it to be seamlessly deployed on any Ethernet connection unbeknownst to devices on either end of the wire (other than the intentional network effects it creates, of course). The system works on a FreeBSD computer as well as a virtual machine, and has the potential to be deployed on an embedded machine from the image we constructed.

5.3 Future Work and Possible Extensions

The approach taken to emulating multiple links may be viewed as slightly crude. Passing traffic through multiple ipfw rules (and therefore through multiple dummynet pipes) by manipulating the "one pass" sysctl kernel variable does work, but the results begin to falter as more nodes are added. It is thought that this may be addressed by utilizing netgraph, the kernel networking subsystem.

At the time of writing there have been several problems deploying the stripped down image we created to the Soekris machine and this is a problem which may be addressed in future work.

Adding the ability for the emulator to modify packets in a random manner may be another further extension. Furthermore, adding low order functionality to the emulator such as the ability to reorder packets, fragment packets, duplicate packets and randomly modify packets (as occurs in real world networking when networking infrastructure incurs interference from the surrounding environment) could be a useful.

Bibliography

[1] Anue-systems : Satellite communications. Online: http://www.anuesystems.com/pdf/CaseStudy_Satellite.pdf, Accessed: 10/05/2007.

[2] Atis committee - atis telecom glossary. Online: http://www.atis.org/tg2k/, Accessed: 22/10/2007.

[3] Information sciences institute - nam: Network animator. Online: http://www.isi.edu/nsnam/nam/, Accessed: 28/05/2007.

[4] Perl version 5.8.8 documentation. Online: http://perldoc.perl.org/, Accessed: 11/10/2007.

[5] Tcpdump(1) freebsd man page. Online: http://www.freebsd.org/cgi/man.cgi?query=tcpdump, Accessed: 01/07/2007.

[6] Vmware. Online: http://www.vmware.com/, Accessed: 10/28/2007.

[7] Freebsd man pages - dummynet (4). Online: http://www.freebsd.org/cgi/man.cgi?query=dummynet, Accessed: 22/10/2007, 1998.

[8] Avinash lakhiani - jplot, a java plotting tool. Online: http://irg.cs.ohiou.edu/software/tcptrace/jPlot/, Accessed: 24/10/2007, 2002.

[9] Packetstorm - ip network emulation. Online: http://www.packetstorm.com/whitepapers.php, Accessed: 11/05/2007, 2003.

[10] gnuplot, an interactive plotting program. Online: http://www.gnuplot.info/documentation.html, Accessed: 24/10/2007, 2004.

[11] Soekris engineering - net4801 series boards and systems user manual. Online: http://www.soekris.com/Manuals/net4801_manual.pdf, Accessed: 14/10/2007, 2004.

[12] Allman, M., and Ostermann, S. One: The ohio network emulator.

[13] Ashwin Gumaste, T. A. *First Mile Access Networks and Enabling Technologies.* CISCO Press, 2004.

[14] B. Braden, D. C. e. a. Recommendations on queue management and congestion avoidance in the internet. *RFC 2309* (1998).

[15] Breslau, L., Estrin, D., Fall, K., Floyd, S., Heidemann, J., Helmy, A., Huang, P., McCanne, S., Varadhan, K., Xu, Y., and Yu, H. Advances in network simulation. *Computer 33*, 5 (2000), 59–67.

[16] Clark, D. D. Ip datagram reassembly algorithms. *RFC 815* (1982).

[17] Cnodder, S. D., Elloumi, O., and Pauwels, K. Red behavior with different packet sizes.

[18] Courtney, D. minibsd 6.x guide. Online: `http://www.ultradesic.com/index.php?section=125`, Accessed: 29/10/2007, 2007.

[19] ENDRES, S. M. Simulation and emulation of the space networking environment. Master's thesis, Department of Electrical Engineering and Computer Science, CASE WESTERN RESERVE UNIVERSITY, January 2005.

[20] Fall, K. Network emulation in the VINT/NS simulator. *Proceedings of the fourth IEEE Symposium on Computers and Communications* (1999).

[21] Floyd, S., and Jacobson, V. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking 1*, 4 (1993), 397–413.

[22] Gaynor, M. Proactive packet dropping methods for tcp gateways. *http://people.bu.edu/mgaynor/papers/final.ps* (1996).

[23] Gentleman, R., and Ihaka, R. The r project. Online: `http://www.r-project.org/`, Accessed: 14/10/2007.

[24] Greg Lehey, M. K. M. *The Complete FreeBSD.* O'Reilly & Associates, Inc, 2003.

[25] Grenville Armitage, L. S. Some thoughts on emulating jitter for user experience trials. *SIGCOMM ACM Workshops* (2004).

[26] Herrscher, D., and Rothermel, K. A dynamic network scenario emulation tool. 262–267.

[27] Institute, I. S. Internet protocol, darpa internet program, protocol specification. *RFC 791* (1981).

[28] Kayssi, A., and El-Haj-Mahmoud, A. Emunet: A real-time ip network emulator.

[29] Kayssi, A., and El-Haj-Mahmoud, A. Emunet: a real-time network emulator. *ACM symposium on Applied computing* (2004), 357–362.

[30] Ke, Q., Maltz, D., and Johnson, D. B. Emulation of multi-hop wireless ad hoc networks.

[31] Leffingwell, D. Mastering the iteration: An agile white paper.

[32] Lepreau, J. The utah network emulation facility. *School of Computing Research Facility*.

[33] Longin Jan Latecki, Kishore Kulkarni, J. M. Better audio performance when video stream is monitored by tcp congestion control. *IEEE Int. Conf. on Multimedia & Expo, Baltimore* (2003).

[34] Mark Carson, D. S. Nist net. a linux-based network emulation tool. *ACM SIGCOMM Computer Communications Review 33* (2003), 111–126.

[35] Ni, L., and Zheng, P. Empower: A network emulator for wireline and wireless networks.

[36] Padhye, J., Firoiu, V., Towsley, D., and Krusoe, J. Modeling TCP throughput: A simple model and its empirical validation. 303–314.

[37] Paxson, V., and Floyd, S. Why we don't know how to simulate the internet. 1037–1044.

[38] Pentikousis, K. Active queue management. *ACM Connector Columns* (July 2001).

[39] Ramadas, M. Tcptrace manual. Online: `http://www.tcptrace.org/manual/index.html`, Accessed: 24/10/2007, 2003.

[40] Ramaswamy Ramaswamy, Ning Weng, T. W. Considering processing cost in network simulations. *ACM SIGCOMM Workshops* (2003).

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

[42] Scott Guelich, S. G., and Birzneiks, G. *CGI Programming.* O'Reilly, 2000.

[43] Shepherd, T. J. Tcp packet trace analysis. Master's thesis, Massachusetts Institute of Technology, 1991.

[44] Tanenbaum, A. S. *Computer Networks*. Prentice Hall, 2002.

[45] Teare, D. Designing cisco networks. *Cisco Press* (1999).

[46] Terzis, A., Nikoloudakis, K., Wang, L., and Zhang, L. Irlsim: A general purpose packet level network simulator.

[47] Wikipedia. Random early detection. Online: `http://en.wikipedia.org/wiki/Random_early_detection`, Accessed: 11/03/2007.

Appendix A

Construction of the Emulator

A.1 Building the Embedded Machine

The majority of work here is attributed to David Courtney and his guide [18] to building stripped down FreeBSD systems.

The building of an operating system inside a jail is discussed here, which takes place on an existing FreeBSD installation. Firstly a new folder is created for the jail (mkdir /usr/jail) after which 'sysinstall' is executed allowing us to create a fresh installation. We used a minimal installation with only 'lib' and 'sys' under 'src'. Relevant config files (such as resolv.conf and the current systems kernel) are copied into the jail. We enter the jail as per Listing 9.

Now that we have the jail we can begin the actual work of creating the stripped down custom FreeBSD system. The following sections highlight how this was achieved.

A.1.1 Creating Directory Structures

A minimal directory structure as per Listing 10 was manually created (mkdir) within the chrooted system. Permissions of the proc directory were set to 555, and the permissions of the root directory to 700. Also, a symlink was created from /var/tmp to /tmp.

Listing 9 Entering the jail

```
#mount -t devfs devfs /usr/jail/dev
#chroot /usr/jail /bin/csh
NIAB#
```

Listing 10 NIAB Directory Structure

```
/usr/niab
  -bin
  -boot
     .defaults
  -dev
  -etc
     .defaults
     .mtree
  -lib
  -libexec
  -mnt
  -proc
  -root
  -sbin
  -usr
     .bin
     .lib
       -aout
     .libexec
     .local
     .sbin
     .share
       -misc
  -var
```

Listing 11 Fixing Soekris Bug

```
cd /sys/boot
make clean
make
make install
```

A.1.2 Rebuilding the Boot Loader

There is a bug on the Soekris BIOS [11] which results in the terminal text being displayed incorrectly. This is overcome by editing '/sys/boot/i386/libi386/Makefile' and removing 'CFLAGS+= -DTERM_EMU'. The boot loader was compiled and installed with the options specified in Listing 11.

A.1.3 Building Dynamic Executables

We now rebuild binary executables for the emulator. From 'sysinstall' we choose 'Configure' and then 'Distributions' and from within the 'src' options we select: contrib, libexec, release, bin, sbin, ubin and usbin. We now compile and install the libraries as per Listing 12[1].

A.1.4 Copying the Binaries Over

Two scripts were used to copy the binary files into the emulator system. Certain files from /boot, /libexec, /bin, /sbin, /usr/sbin, /usr/libexec and /usr/share which were deemed to be required for the stripped down system. The scripts can be viewed in appendix A.2, and the files chosen are as follows.

A.1.5 Configuring Boot Files

We remove the BSD boot menu and remove boot delay by editing '/usr/niab/boot/loader.rc'. Several lines are commented out, and the autoboot option is added (a backslash denotes a comment in this context) as per Listing 13.

[1]Note: there is no need to run 'make install' after compiling a library

Listing 12 Building Dynamic Executables

```
#First libasm
cd /usr/src/lib/libsm
make clean
make

cd /usr/src/bin
make clean
make
make install

cd /usr/src/contrib/ipfilter/tools
make clean
make

cd /usr/src/sbin/ipf
make clean
make

mkdir -p /usr/share/doc/atm
cd /usr/src/sbin
make clean
make
make install

cd /usr/src/lib/bind
make clean
make

cd /usr/src/lib/libtelnet
make clean
make

cd /usr/src/lib/libsmutil
make clean
make

cd /usr/src/lib/libsmdb
make clean
make

cd /usr/src/usr.bin
make clean
make
mkdir -p /usr/share/info
make install
```

Listing 13 '/usr/niab/boot/loader.rc'

```
    \ Load in the boot menu
    \ include /boot/beastie.4th
    \ Start the boot menu
    \ beastie-start
    autoboot 0
```

Listing 14 Soekris Specific Kernel Options

```
options         CPU_GEODE
options         CPU_SOEKRIS
```

A.1.6 Kernel Compilation

In the previous sections we focused on the binaries and file structure for the emulator. We now want to build a custom stripped down kernel (which includes the kernel options discussed in section 3.2.1.1). A copy of the GENERIC kernel is made and edited. Most of the superfluous options are removed. The full kernel config file is listed in appendix A.3, but it is worthy to note here the custom options for the Soekris net4801. The Soekris runs a Geode processor of which there is support in the FreeBSD system as per Listing 13.

Furthermore, support for USB devices is added to allow an external USB hard drive to be connected for collecting tcpdump data for analysis. The new kernel is now compiled and installed into the emulator system with the following commands.

A.1.7 Populating /etc

We now need to create all configuration files for the emulator system. For the most part, these can be copied as is from the already populated /etc directory inside the FreeBSD

Listing 15 Compiling NIAB Kernel

```
#config NIAB
#cd ../compile/NIAB
#make clean && make cleandepend && make depend && make
#gzip -9 kernel
#mkdir -p /usr/niab/boot/kernel
#cp kernel.gz /usr/niab/boot/kernel
```

Listing 16 Editing '/etc/fstab' in the NIAB system

```
/dev/ad0a      /        ufs      ro        1  1
proc           /proc    procfs   rw        0  0
md             /var     mfs      rw,-s8m   2  0
md             /tmp     mfs      rw,-s8m   2  0
```

Listing 17 NIAB's '/etc/ttys' modified for serial support and no VGA

```
console    none                              unknown   off   secure
ttyv0      "/usr/libexec/getty Pc"           cons25    off   secure

#Virtual terminals
ttyv1      "/usr/libexec/getty Pc"           cons25    off   secure
[..]
ttyv7      "/usr/libexec/getty Pc"           cons25    off   secure
ttyv8      "/usr/X11R6/bin/xdm -nodaemon"    xterm     off   secure
ttyd0      "/usr/libexec/getty std.9600"     vt100     on    secure
```

host system. '/etc/fstab' needs to be modified for the system. The CompactFlash memory card is mounted as 'ro', indicated read only. This is crucial as Flash memory supports only a limited number of erase/write cycles before a particular sector can no longer be written. Memory specifications generally allow 10,000 to 1,000,000 write cycles. Typically the controller in a CompactFlash attempts to prevent premature wear out of a sector by mapping the writes to various other sectors in the card - a process referred to as wear levelling. We do not use any swap space, and rely only on the Soekris' RAM. Adding dumpdev="NO" to the rc.conf will remove warnings about the lack of a swap partition. The custom fstab file for the Soekris is displayed in Listing 16 (note the CompactFlash is mounted read only).

Custom 'host.conf' and 'rc.conf' files need to be created too. Very little needs to be changed from these files as mentioned in the previous sections.

Since the Soekris has a serial console we will need to enable it. It is a headless system (no DSUB output) and this is a useful way to connect to the system. 'etc/ttys' is edited to allow this as per Listing 17.

We now move onto the building of the final image.

Listing 18 Checking number of sectors on the CompactFlash Card

```
#bsdlabel –Awn da0s1 auto | grep sectors/unit
#sectors/unit: 500704
```

Listing 19 Creating an empty image

```
#dd if=/dev/zero of=/usr/niab−disk.bin bs=512 count=500704
#mdconfig −a −t vnode −u 0 −f /usr/niab−disk.bin
#bsdlabel –Bw md0 auto
#bsdlabel −e md0
```

A.1.8 Building the Binary Image

The final step in the process is to take the stripped down FreeBSD system and assemble it into a single, binary file which can be written to the CompactFlash memory. An automated script was used to go through the many steps involved, this can be viewed in appendix A.2 but we will discuss the manual intricacies now.

One option would be to use the CompactFlash card like a hard disk. We could use bsdlabel and newfs on on the card and output the tar archive directly on it. However, it is faster and easier to do the task in a disk image and dd the image onto the flash card. This also prolongs the life of the card by only writing each sector once.

vnconfig is used to create a virtual disk that we use bsdlablel on. We first need to work out the number of sectors (512 byte units) on the FlashCard. This is achieved by plugging the card into a memory reader/write on the FreeBSD machine and typing the commands as per Listing 18.

The output tells us that the device has 500704 sectors. We now create a disk image of this same size which is initially populated with zeros, and use this disk image file as a vn device so we can bsdlabel it to create the partition and file system (See Listing 19).

'bsdlabel' will load a text editor, from where we can add the information as per Listing 20 about the device. The line starting with 'a:' is the root partition, which will span over the whole slice.

The next step is to create a file system. The 'newfs' command constructs a UFS1/UFS2 filesystem, as per Listing 21.

The virtual disk is now mounted on /mnt. To copy the NIAB FreeBSD files onto it we type command "(cd /usr/niab ; tar cPf - .) | (cd /mnt ; tar xf -);".We now have a 256MB custom, stripped down FreeBSD system in a format which will allow us to write it to our

Listing 20 'bsdlabel' options for the NIAB system

```
# /dev/md0:
8 partitions:
#        size   offset    fstype   [fsize bsize bps/cpg]
a:     500704        0    4.2BSD        0     0     0
c:     500704        0    unused        0     0
```

Listing 21 Creating a new file system

```
#newfs -b 8192 -f 1024 -U /dev/md0a
#mount /dev/da0s1 /mnt
```

CompactFlash media.

A.1.9 Writing the Image to the Soekris Flash Card

There are two options available for writing the image to the CompactFlash card. We can netboot (with PXE) the Soekris with the CompactFlash card installed and copy the files over the network, or we could simply write the image to the card with the dd command.

The simpler solution is to write the image to the CompactFlash from a card writer/reader with the command "dd if=/usr/niab-disk.bin of=/dev/da0s1 bs=8k".

We can now boot our Soekris system.

A.2 Scripts used in Building Minimal FreeBSD System

Several scripts were used in the building of the stripped down FreeBSD system. They were written by David Courtney of Ultradesic and are aimed specifically at deploying FreeBSD to embedded machines. The scripts are included on the CD.

A.3 NIAB Kernel Listing

```
# Custom NIAB Kernel
# $FreeBSD: src/sys/i386/conf/NIAB,v 1.0.0

machine i386
cpu I586_CPU
ident NIAB

# Options Specific to the Soekris NET48XX
options CPU_GEODE
options CPU_SOEKRIS

options SCHED_4BSD
options INET
options INET6
options FFS
options SOFTUPDATES
options UFS_ACL
options UFS_DIRHASH
options MD_ROOT
options NFSCLIENT
options NFSSERVER
options NFS_ROOT
options MSDOSFS
options CD9660
options PROCFS
options PSEUDOFS
options GEOM_GPT
options COMPAT_43
options COMPAT_FREEBSD4
options SCSI_DELAY=15000
options KTRACE
options SYSVSHM
options SYSVMSG
options SYSVSEM
options _KPOSIX_PRIORITY_SCHEDULING
options KBD_INSTALL_CDEV
options AHC_REG_PRETTY_PRINT
options AHD_REG_PRETTY_PRINT
options ADAPTIVE_GIANT
deviceapic

# Bus support.
deviceisa
deviceeisa
devicepci

# ATA and ATAPI devices
deviceata
deviceatadisk
options ATA_STATIC_ID

devicenpx
devicepmtimer

# Serial (COM) ports
devicesio

devicemiibus
devicesis

# Wireless NIC cards
devicewlan
devicean
deviceawi
devicewi
#devicewl

# Pseudo devices.
deviceloop
devicemem
deviceio
devicerandom
deviceether
devicepty
devicemd
devicegif
```

```
devicefaith
devicebpf

# USB support
deviceuhci
deviceohci
deviceehci
deviceusb
deviceugen
deviceumass

#Greater granularity
optionshz=10000

#SCSI bits for USB support
devicescbus
deviceda

#Firewall
options IPFIREWALL
options IPFIREWALL_VERBOSE
options IPFIREWALL_DEFAULT_TO_ACCEPT
options IPFIREWALL_VERBOSE_LIMIT=1000

device if_bridge
options DUMMYNET
options BRIDGE
options IPSTEALTH
```

Appendix B

Emulator Code

Here we list useful code snippets of web front end code from various sections. Each snippet begins with the filename it has been extracted from, as well as comments describing what the snippet is used for. The full source is available on the CD.

B.1 PHP

Each page uses a PHP include direction to include common PHP code, such as the menu and banner.

```
<?php
  include("common.php");
?>
```

Several pages process GET and POST methods. These are accessed in PHP via the $_GET[<key>] and $_POST[<key>]

```
//Example of parsing GET request from interfaces.php
<?php
        include("common.php");
        $numIf = $_GET["num"];
        $ifA = $_GET["if0"];
?>
```

Several pages send POST or GET requests to the server.

```
//Example of accepting input from the user from various
//input boxes and POSTing the result to the server from
//newNode.php
<?php
```

```
        <form action="/cgi-bin/newNode" method="POST">
        //various input boxes
        <input type="submit" name="create" value="Create">
        </form>
?>
```

XML parsing was done via the “simplexml” module to retrieve and display node information from nodes.xml

```
// Example of parsing nodes.xml from nodeSet.php
// Here we see the use of the "simplexml" PHP
// module

<?php
$nodesFile = "./confiles/nodes.xml";
$xml = simplexml_load_file($nodesFile);

//XML parsing code
$doc = new DOMDocument();
$doc->load( $nodesFile );

$nodes = $doc->getElementsByTagName("node");
$i=0;
$nodeNames;
$arr_connections = array();

foreach( $nodes as $node ){
 $connName = $node->getAttribute('name');
 if($node->getAttribute('inUse') == "Y")
 //Q Mgt info:
 $node->getElementsByTagName("qMgt");
 foreach($qMgtBits as $qMgtBit){
  $qMgt = $qMgtBit->getAttribute('type');
  $qMgt_w_q = $qMgtBit->getElementsByTagName("w_q")->item(0)->nodeValue;
  $qMgt_min_th = $qMgtBit->getElementsByTagName("min_th")->item(0)->
      nodeValue;
  $qMgt_max_th = $qMgtBit->getElementsByTagName("max_th")->item(0)->
      nodeValue;
  $qMgt_max_p = $qMgtBit->getElementsByTagName("max_p")->item(0)->nodeValue
      ;

  $arr_qAtts = array();
  if($qMgt == "RED" || $qMgt=="GRED"){
   array_push($arr_qAtts, $connName,$qMgt, $qMgt_w_q,$qMgt_min_th,
       $qMgt_max_th, $qMgt_max_p);
```

```
    array_push($arr_connections, $arr_qAtts);
   }
  }

 $bwUp = $node->getElementsByTagName( "bwUp" )->item(0)->nodeValue;
 $bwDown = $node->getElementsByTagName( "bwDown" )->item(0)->nodeValue;
 $delay = $node->getElementsByTagName( "delay" )->item(0)->nodeValue;
 $drop = $node->getElementsByTagName("drop")->item(0)->nodeValue;
 $qMgt = $node->getElementsByTagName("qMgt")->item(0)->nodeValue;
?>
```

B.2 Perl (CGI)

Perl scripts requiring access to the nodes.xml file parse it via the “XML::Simple” and “XML::SAX” CPAN modules.

```
#Example of Perl processing XML
#Taken from setActiveNodes.xml
#!/usr/bin/perl
use XML::Simple;
use XML::SAX;

my $file = 'nodes.xml';
my $xsl = XML::Simple->new();

my $doc = $xsl->XMLin($file);
foreach my $key (keys (%{$doc->{node}})){
                print $doc->{node}->{$key}->{'name'} . $key;
                print $doc->{node}->{$key}->{'bwUp'} . "\n";
                print $doc->{node}->{$key}->{'bwDown'} . "\n";
                #etc
}
```

Identifying information concerning the network cards was done via the Ifconfig wrapper.

```
#!/usr/local/bin/perl
#Example of using ifconfig wrapper to probe NIC information
#Taken from getInterfaces perl script

use Net::Ifconfig::Wrapper;

@ifaces=getIfaces(); #get network interfaces

```

```
#Probe and return local interfaces
sub getIfaces(){
  my @interfaces;
  my $Info = Net::Ifconfig::Wrapper::Ifconfig('list', '', '', '')
   or die $@;

   foreach (sort(keys(%{$Info}))){
     push(@interfaces, $_);
    }
   return @interfaces;
}
```

Several Perl scripts required passing GET requests to PHP scripts. A request is built and then passed to the web page. The full Perl code is listed on the CD.

Appendix C

Scripts for Testing the Emulator

This appendix contains all the test scripts written to produce the results reported on in Chapter 4. The throughput and RTT scripts are used to vary these two factors over time, producing output suitable for graphing. The general script is for more general cases and only captures data and produces output suitable for graphing.

C.1 Throughput

```
#!/usr/bin/perl
#Glenn Wilkinson
#Script for testing NIAB Bandwidth
#Creates pipes and varies their bandwidth between intervals. Output
    suitable for R
#./test_bw.pl <fname> <delay> [<bw>]

if($#ARGV+1 < 2){
        print "\nUsage: ./test_bw.pl <dumpFname> <delay> [<bw>]\n";
        exit;
}

my $delay=$ARGV[1];
my $fname=$ARGV[0];


#We start the initial pipe, otherwise our graph may have a huge outlier
    left
#`ipfw -f flush`;
$sex=`ipfw -q delete 100`;
```

```
`ipfw -q 100 add pipe 1 ip from any to any`;
`ipfw pipe 1 config bw $ARGV[2]Kbit/s`;

print "Pipe created\n";


sleep 2; #Let it settle

my $pid=fork();
if( not defined $pid){
        print "Resource fail";
}
elsif($pid==0){
        `tcpdump -q -i bridge0 -w $fname`;
        print "Saving dump to file " . $fname . "\n";
}
else{ #parent

        for($i = 2; $i<$#ARGV+1; $i++){
                `ipfw pipe 1 config bw $ARGV[$i]Kbit/s`;
                print "Limiting BW to " . $ARGV[$i] . "\n";
                sleep $delay;

        }


`killall tcpdump`;
waitpid($pid,0);
}#end parent

`ipfw -q delete 100`;



#Next Step
###########################################
#TCPTRACE
`tcptrace -T -y -zxy -A10 -o1 testDump`; #Will dump to b2a.xpl

#Remove red bits
my $in = open(FILE, "b2a_tput.xpl") or die("Unable to open file in tcptrace
     section");
my $out = open(FILE2,">" . $fname . ".txt") or die ("Unable to open output
    file in tcptrace section");
```

```
my $tmp;
my $str;

while($str = <FILE>){

        chomp($str);
        if($str eq "blue"){
                $tmp=<FILE>;$tmp=<FILE>; #Read next to lines and discard

        }
        elsif($str eq "red"){
                $tmp=<FILE>;
                $tmp=~m/^dot\s(\S*)\s(\S*)/;
                print FILE2 $1 . ";". $2 . "\n";
                $tmp=<FILE>;
        }
}

close FILE;
close FILE2;
`rm testDump`;
`delete a2b_tput.xpl`;
`delete b2a_tput.xpl`;
print "Your data is in " . $fname . ".txt\n";
```

C.2 RTT

```
#!/usr/bin/perl
#Glenn Wilkinson
#Script for testing NIAB RTT
#Creates pipes and varies their delay between intervals. Output suitable
    for R
#./test_bw.pl <fname> <delay> [<bw>]

if($#ARGV+1 < 2){
        print "\nUsage: ./test_RTT.pl <dumpFname> <delayInterval> [<delay
            >]\n";
        exit;
}

my $delay=$ARGV[1];
my $fname=$ARGV[0];


```

```
#We start the initial pipe, otherwise our graph may have a huge outlier
    left
#`ipfw -f flush`;
$sex=`ipfw -q delete 100`;
`ipfw -q 100 add pipe 1 ip from any to any`;
`ipfw pipe 1 config delay $ARGV[2]ms`;

print "Pipe created\n";


sleep 2; #Let it settle

my $pid=fork();
if( not defined $pid){
        print "Resource fail";
}
elsif($pid==0){
        `tcpdump -q -i bridge0 -w $fname`;
        print "Saving dump to file " . $fname . "\n";
}
else{ #parent

        for($i = 2; $i<$#ARGV+1; $i++){
                `ipfw pipe 1 config delay $ARGV[$i]ms`;
                print "Setting delay of " . $ARGV[$i] . "ms\n";
                sleep $delay;

        }


`killall tcpdump`;
waitpid($pid,0);
}#end parent

`ipfw -q delete 100`;



#Next Step
##########################################
#TCPTRACE
`tcptrace -R -zxy -A10 -o1 testDump`; #Will dump to a2b.xpl

#Remove red bits
```

```
my $in = open(FILE, "a2b_rtt.xpl") or die("Unable to open file in tcptrace section");
my $out = open(FILE2,">" . $fname . ".txt") or die ("Unable to open output file in tcptrace section");
my $tmp;
my $str;

while($str = <FILE>){

        chomp($str);
        if($str eq "blue"){
                $tmp=<FILE>;$tmp=<FILE>; #Read next to lines and discard

        }
        elsif($str eq "red"){
                $tmp=<FILE>;
                $tmp=~m/^dot\s(\S*)\s(\S*)/;
                print FILE2 $1 . ";". $2 . "\n";
                $tmp=<FILE>;
        }
}

close FILE;
close FILE2;
`rm testDump`;
`rm a2b_rtt.xpl`;
`rm b2a_rtt.xpl`;

print "\nYour data is in " . $fname . ".txt\n";
```

C.3 General

```
#!/usr/bin/perl
#Use this script to generate output suitable for plotting with R. Removes average
# plots from tcptrace. Use in the case of manually manipulating pipes.
#Start a tcpdump session first, e.g:
#tcpdump -i bridge0 -w testDump && ./genGraphs


#TCPTRACE
`tcptrace -G -y -zxy -A10 -o1 testDump`; #Will dump to x2x.xpl

#print "\n\n" . $fname . "\n\n";
```

```

#Remove red bits
my $in = open(FILE, "b2a_tput.xpl") or die("Unable to open file in tcptrace section");
my $out = open(FILE2,">testDump.txt") or die ("Unable to open output file in tcptrace section");
my $tmp;
my $str;

while($str = <FILE>){

        chomp($str);
        if($str eq "blue"){
                $tmp=<FILE>;$tmp=<FILE>; #Read next to lines and discard

        }
        elsif($str eq "red"){
                $tmp=<FILE>;                    #read "dot" line
                $tmp=~m/^dot\s(\S*)\s(\S*)/;    #and remove word dot, put in ;
                print FILE2 $1 . ";". $2 . "\n";
                $tmp=<FILE>;                    #read and discard "line"
        }
}

close FILE;
close FILE2;
```