A VIRTUAL INTEGRATED NETWORK EMULATOR ON XEN (viNEX)

by

MUKOSI ABRAHAM MUKWEVHO

submitted in accordance with the requirements
for the degree of

MASTER OF SCIENCE

in the subject

COMPUTER SCIENCE

at the

UNIVERSITY OF SOUTH AFRICA

SUPERVISOR: PROF. J A VAN DER POLL

CO-SUPERVISOR: MR. R M JOLLIFFE

November 2010

Abstract

Network research experiments have traditionally been conducted in emulated or simulated environments. Emulators are frequently deployed on physical networks. Network simulators provide a self-contained and simple environment that can be hosted on one host. Simulators provide a synthetic environment that is only an approximation of the real world and therefore the results might not be a true reflection of reality.

Recent progress in virtualisation technologies enable the deployment of multiple interconnected, virtual hosts on one machine. Virtual hosts run real network protocol stacks and therefore provide an emulated environment on a single host. The first objective of this dissertation is to build a network emulator (viNEX) using a virtualisation platform (XEN). The second objective is to evaluate whether viNEX can be used to conduct some network research experiments. Thirdly, some limitations of this approach are identified.

Acknowledgements

I would like to take this opportunity to thank Mr. Robert Mark Jolliffe (Bob) for his outstanding support which contributed significantly in making this research possible. Bob played a major role in identifying the foundation for this research and provided exceptional leadership and encouragement throughout. He eventually left the University to pursue his own career objectives. He continued to provide me with critical and valuable support and guidance even after leaving the University. I would like to thank him with sincere gratitude for his willingness to help me in improving my research work for the benefit of the scientific community.

After Bob left the University, Prof. John Andrew van der Poll (André) was appointed as my supervisor taking over from Robert. I would like to thank him for all the support he gave me after taking over from Bob. André offered his help and provided direction for this networks research even though it was not his primary research focus area. I would also like to thank him especially for assisting and motivating in the source of funding from the University for my conference trip to Rome (Italy) to present my research paper which appears in this dissertation as Appendix A. André also assisted significantly in providing lead information with regard to sourcing the funding of my final year through the University's post graduate funding. André was the overall eye on this research providing guidance which helped in the steering of this research work to its final outcomes.

Contents

List of Figures

List of Tables

List of Source Code Listings

List of Published Papers

Abraham Mukosi Mukwevho, John Andrew van der Poll, and Robert Mark Jolliffe. A virtual integrated network emulator on XEN (viNEX). In *Simutools '09: Proceedings of the 2nd International Conference on Simulation Tools and Techniques, pages 1 – 7.* Rome, Italy, 2009. Editors: O. Dalle, L.F. Perrone, G. Stea and G. A. Wainer, ISBN: 978-963-9799-45-5.

List of Acronyms and Pseudo Acronyms

Notation	Description	Page List
APIC	Advanced Programmable Interrupt Controller	24
ARP	Address Resolution Protocol	86
ATM	Asynchronous Transfer Mode	15, 116
CBR	Constant Bit Rate	15
F/OSS	Free and Open Source Software	40, 85
FDDI	Fiber Distributed Data Interface	116
FIFO	First In First Out	15, 18
FTP	File Transfer Protocol	15
GigE	Gigabit Ethernet	11
GPRS	General Packet Radio Service	15
GSM	Global System for Mobile Telecommunication	15
HTTP	Hypertext Transfer Protocol	15
HTTPS	Hypertext Transfer Protocol Secure	15
HVM	Hardware Virtual Machine	23, 33, 34
IP	Internet Protocol	15
IPC	InterProcess Communication	71
ISI	Information Sciences Institute	2

Notation	**Description**	**Page List**
LBL	Lawrence Berkeley National Laboratory	29, 65–68, 125, 126
MAC	Media Access Control address	15, 27, 50
MADIC	Multidimensional Applications and Gigabit Internetwork Consortium	10
MMU	Memory Management Unit	24
MTU	Maximum Transmission Unit	67, 75, 87
NAM	Network ANimator	15
NS-2	Network Simulator version 2	1, 2, 6, 8, 9, 14, 15, 17, 29, 88–90
OS	Operating System	3, 18, 25
PVM	Paravirtualised Virtual Machine	23, 33, 34, 37, 39
RED	Random Early Detection	15, 18
SSH	Secure Shell	20, 48
TCP	Transmission Control Protocol	2, 7, 15, 75, 89

Notation	**Description**	**Page List**
UCB	University of California Berkeley	29, 65–67, 69, 125, 126
UDP	User Datagram Protocol	73, 75, 76, 81, 89
UMA	Universal Memory Allocator	72, 76
vAPIC	virtual Advanced Programmable Interrupt Controller	24
viNEX	virtual integrated Network Emulator on XEN	3, 31
VMM	Virtual Machine Monitor	22, 23
vMMU	virtual Memory Management Unit	24

Glossary

Notation	Description	Page List
Bridging	In computer networks, bridging refers to the process of joining network segments by special devices called *bridges*. A *bridge* is a network device that connects network segments at layer 2 (the data link layer). The operation of the network bridge is specified by the IEEE 802.1D standard.	27
IMUNES	The Integrated Multiprotocol Network Emulator/Simulator (or IMUNES) is a network topology emulation and simulation framework which was developed using the FreeBSD operating system kernel partitioned into multiple lightweight virtual nodes. The FreeBSD partitions are interconnected using kernel-level links to form arbitrarily complex network topologies.	31, 32

Notation	**Description**	**Page List**
mbuf	An `mbuf` is a basic unit of memory management used in the BSD kernel IPC subsystem. Mbufs are used for storing network packets. Socket buffers are also stored using mbufs. A network packet may span multiple mbufs arranged into an `mbuf` chain. An `mbuf` chain is a linked list structure that provides functionality for adding or removing network headers with little overhead. The size of a single `mbuf` is 256 Bytes. Network packets normally do not fit into one `mbuf`. As a result, an external storage structure called `mbuf cluster` is provided. For more information on `mbuf` structures, see FreeBSD [27].	xv, 71
mbuf-cluster	The default size of an `mbuf cluster` is 2 KBytes. The size of an mbuf cluster depends on the architecture of the machine. For example, for i386 the size of 2 KB is normally used and 4 KB for virtualization platforms (such as Xen). Mbuf clusters are supplied as an option to provide external storage for network packets that do not fit into an `mbuf`. Without mbuf clusters, packets can still be stored into multiple linked mbufs forming a chain. For each single `mbuf cluster` allocated, one `mbuf` is also allocated and it is used to arrange `mbuf clusters` in a chain. For more information on `mbuf cluster` structures, see FreeBSD [27].	71

Notation	**Description**	**Page List**
OSPF	Open Shortest Path First (OSPF) is a routing protocol designed to propagate routing information based on the shortest path first (SPF) algorithm. OSPF consists of one or more areas which are organized into a hierarchy. OSPF areas are grouped together to form a large entity called the Autonomous System (AS). OSPF is an interior gateway protocol meaning it is designed mainly to exchange routing information within a single AS. The OSPF protocol was developed by the Interior Gateway Protocol (IGP) working group of the Internet Engineering Task Force (IETF) with the objective of overcoming some of the limitations experienced with the RIP (Malkin [50]) protocol (see RIP below). OSPF is described by RFC2328 (Moy [55]).	43, 60, 86
RIP	Routing Information Protocol is a routing protocol designed to propagate routing information using a distance-vector table. RIP uses the hop count and the distance metric. RIP is also an Interior Gateway Protocol (IGP) and it is widely used in the internet communities. One of the main drawbacks of RIP is the hop count limit of 15, any networks beyond this hop count limit are considered unreachable. Each host running RIP operates by sending the distance-vector matrix within each entry containing the addresses of known and reachable networks as well as the number of hops to reach those networks. RIP is described in RCF2453 (Malkin [50]).	60, 85

Notation	**Description**	**Page List**
TCP/IP	Transmission Control Protocol/Internet Protocol is a collection of network protocols developed at the U.S. Department of Defense (DOD) during the 1970s aimed at transmitting data between network devices.	34, 39
UML	User Mode Linux is a virtualisation technology that enables multiple virtual Linux systems to run on top of the main Linux system as applications. It is used mainly for hosting virtual servers, kernel development, experimenting with new kernels and distributions as well as for educational purposes.	6, 22
VLAN	A Virtual Local Area Network is a group of computers with a common set of requirements that communicate as if they were attached to the same domain, regardless of their physical location. A VLAN is similar to physical LAN (Local Area Network), but it allows for end stations to be grouped together even if they are not located on the same network switch. VLAN configuration is done by using software without the need to physically relocate network devices.	50, 51, 91
VPN	A Virtual Private Network is a private network that makes use of the public telecommunication infrastructure over the Internet. Privacy is maintained by the use of tunneling protocols and security functions. VPNs provide similar functionality as offered by traditional, leased lines.	7

Notation	**Description**	**Page List**
Xen	Xen [87] is a virtual machine monitor for x86-compatible architectures, namely, the IA-32 (x86, x86-64), IA-64 and PowerPC 970 architectures. The primary objective of Xen is to enable several guest operating systems to execute concurrently on the same computer hardware. It was developed at the University of Cambridge Computer Laboratory and is now developed and maintained by the Xen open source community [87]. Xen is available as free software and is licensed under the GNU General Public License (GPL2). A commercially supported version of Xen is available through Citrix and it is called XenServer Enterprise Edition which was originally offered by the XenSource community as XenEnterprise.	3

Dedicated to my parents and family for their support during my studies. A special dedication to my mother Elisa Nyaphophi Mukwevho for the outstanding support, motivations and encouragements she gave me throughout my studies.

Chapter 1

Introduction

This dissertation investigates the possibility of using a general purpose virtual platform for network emulation. Network research experiments are frequently conducted using a physical network in a simulated or an emulated environment. Examples of such environments are Pullen et al. [67], McCanne et al. [52] and White et al. [85].

Naturally, every network experiment environment has advantages and disadvantages. One of the disadvantages of conducting experiments in a real physical network is the cost of procuring and setting up the infrastructure (servers, routers and gateways). The installation and configuration of experiments in such an environment can be complex and results may be hard to repeat. An advantage of physical network environments is that they are a reasonable reflection of the real world. This implies that the experiments and protocols developed in these environments can be applied to production environment, often without major modifications.

A good example of a physical network environment for conducting experiment is Emulab (White et al. [85]). In Emulab, experiments are run on a physical network deployed on real networking hardware and servers in a laboratory of the University of Utah. The complexity of creating and setting up the network topology was made transparent from the user by making use of NS-2 (Network Simulator version 2) syntax (McCanne et al. [52]). It should be noted that Emulab only makes use of the modelling component of NS-2 and not the simulation component.

Simulated environments are based on the use of software modules to model the behaviour of real network components to create an environment for conducting

network experiments. Generally, simulated environments are simpler to configure and they are less expensive than real networks. A significant disadvantage of software simulated environments is that the environment is only an approximation of the real world and therefore the results might not be very accurate.

NS-2 (McCanne et al. [52]) is a classical example of a comprehensive software simulated environment. NS-2 is developed through collaboration of open source communities that are managed by ISI (Information Sciences Institute) of the University of Southern California. NS-2 is a discrete event simulator targeted at network research. NS-2 provides substantial support for the simulation of TCP (Transmission Control protocol), routing, and multicast protocols over wired, wireless and satellite networks. Every component of the network is simulated via some software module in NS-2. NS-2 therefore inherits all the advantages and disadvantages of software simulated environments as stated above.

Emulated environments combine the use of elements of both simulated and physical network environments for creating a network research environment. Network emulators provide a mechanism of imitating a much larger network in a small laboratory environment. This is achieved by using a relatively small set of network nodes with some special network nodes introduced between some nodes in order to simulate the behaviour of a much larger network. An example of such an environment is Emulab (White et al. [85]). Inside the special network nodes, software simulators are installed in order to simulate the characteristics of the larger networks such as bandwidth, propagation delays or packet losses. The advantage of emulated environments is that real network stacks and protocols are deployed on the network nodes as compared to simulated environments.

Emulab also serves as a good example of a network emulator where simulation is provided by the FreeBSD [27] nodes that are inserted between network nodes to simulate network link conditions. Dummynet (Rizzo [70]) is loaded on the FreeBSD nodes in order to simulate the network link conditions such as propagation delay, bandwidth limitation or packet loss.

In this research work, we propose a virtual network emulator viNEX, which is a pseudo-acronym for Virtual INtegrated Network Emulator on Xen. The idea of constructing a virtual network emulator on a single physical machine has been explored before, see Section 2.3. Our contribution, however, lies in exploring the possibility of using a general purpose platform, namely, Xen to create a virtual

network emulator that can run on a single physical machine. This objective corresponds to the first part of our hypothesis in Section 1.2.1.

Our network emulator viNEX (virtual integrated Network Emulator on XEN) developed in this dissertation uses a very similar approach as the one for Emulab. We still make use of the FreeBSD nodes (with Dummynet enabled) to model the network links and conditions. We are using NetBSD as the Operating System (OS) for running the network nodes of the OS emulator. Our network nodes therefore run a real OS with real network protocols stacks. The major difference is that our emulator makes use of virtual network nodes running on top of a single server, enabled by the Xen virtualisation technology.

1.1 Rationale

This research seeks to achieve the following objectives:

- *Develop a virtual network emulator viNEX.*

  Recent progress in computer virtualisation (Singh [76]) has offered the flexibility of hosting multiple, fully functional server instances on a single server. All server instances run real and complete network protocol stacks. viNEX was built using the XEN virtualisation platform. ViNEX provides network researchers with a low-cost single-server network emulator that interacts with real network protocol stacks.

- *Demonstrate that viNEX can be used for conducting network experiments.*

  This objective was demonstrated by conducting the Van Jacobson experiment on viNEX. The results were found to be comparable to those obtained by Van Jacobson in his original experiment.

- *Identifying possible limitations of using general purpose, virtual environments for network emulation.*

  Performance bottlenecks will always occur on a virtualisation platform, even though the underlying virtualisation platforms continue to improve. For example, there are future plans to improve network performance on Xen [88]. The current version of viNEX may not be applicable for high network performance emulation, but it might be useful for other applications, such as

protocol development and verification, as well as educational purposes. Some limitations of this approach were identified by conducting a scalability experiment in Chapter 5.

1.2 Research problem

Recent progress in virtualisation technologies enable the deployment of multiple interconnected, virtual hosts on one machine. Such virtual hosts run real network protocol stacks and therefore provide an emulated environment on a single host. The literature is somewhat quiet about the use of general purpose virtualisation platforms (such as Xen and VMware) for virtual emulators.

Our research problem is as follows: To investigate a possibility of building a virtual network emulator (viNEX) using a virtualisation platform (XEN) and to evaluate whether such an emulator can be used for conducting network research experiments. We also identify some factors affecting the scalability of this emulator and suggest possible ways of addressing them.

The above research problem leads us to the following hypothesis.

1.2.1 Hypothesis

Our hypothesis is twofold:

A general purpose virtualisation platform (such as Xen) can be used to build a network emulator (viNEX) which satisfies the following conditions:

1. The emulator can be used for conducting network research experiments on networks which fall within limited performance boundaries.
2. Such an emulator (viNEX) can be hosted on a single server.

1.3 Research design and methodology

This section briefly presents the methods used in our research. Our research uses the *literature survey*, *prototype* and *experiment* methods (Olivier [59]). Our

literature survey consists of the background of simulation and emulation presented in Chapter 2.

The primary method used in this research is *prototyping*. The prototype involves the construction of the viNEX emulator system which we use to demonstrate the concept of building an emulator, using a general purpose virtual platform (Xen). Experimentation was used as a secondary method in this work. We conducted two experiments on viNEX in Chapter 5. The first experiment (Section 5.1) evaluates the functionality of the viNEX emulator. The Van Jacobson experiment is conducted on viNEX and the results are compared to those obtained in the original experiment (Van Jacobson [81]).

Our research is also *quantitative*. The second experiment examines the scalability of viNEX (see Section 5.4). This is achieved by measuring the amount of memory required to run an experiment on viNEX.

Two *observations* were made while conducting the experiments in Chapter 5. Both observations are used to justify our twofold hypothesis above. We return to the hypothesis in Chapter 6.

1.4 Scope

The scope of this research is limited to the development, and evaluation of the viNEX network emulator. The evaluation of viNEX includes a demonstration of the ability to host network research experiments on viNEX as well as an assessment of the performance limiting factors that could affect the scalability of this emulator.

Some virtualisation performance factors affecting the scalability of viNEX have been identified (see Section 5.4). Since the emulator is designed to run on a single server, scalability of the network topology is limited to the amount of memory and the CPU capability of the server.

The development of the emulator includes a set of shell scripts that are used to perform configuration and management of the emulator. We shall perform the verification and validation of the emulator by running a set of experiments on the emulator, capture results and analyse the results graphically. The objective of such verification is to identify whether network protocols behave as expected and consistently, when deployed on top of the emulator.

We have used shell scripting as the language for *developing* a set of emulator configuration and management scripts. The set includes scripts to be used for automating topology creation as well as link configuration. During the initial stages of this research, we intended to use NS-2 OTcl notation as the modelling language as implemented in NS-2. However, due to the complexities of the technical challenges we encountered during the research, the scope was adjusted accordingly. As a result, the modelling language was changed to shell scripts. NS-2 syntax is quite common in the network research space. Since NS-2 was successfully implemented in Emulab (White et al. [85]), we propose the possibility of using NS-2 in viNEX for future developments.

Regarding the evaluation of the emulator, we made use of some existing TCP analysis and traffic generation tools. We used the `iperf` (Gates and Warshavsky [28]) tool for traffic generation and maximum bandwidth measurement. TCP network data packets were captured using `tcpdump` (Van Jacobson and McCanne [82]). Network data packet analysis was done using the `tcptrace` (Ostermann [61]) tool. The graphical analysis of the results was initially done by using `xplot` (Shepard [74]), but owing to the limited features of `xplot` we migrated to `gnuplot` [30]. Migration was achieved by converting `xplot` datasets to `gnuplot` input files using the `xpl2gpl` (Ostermann [61]) script.

1.5 Related work

The virtualisation of network emulators is currently receiving much research attention. This is mainly due to the need for companies to reduce the number of computer servers overcrowding data centres, to be alleviated through virtualisation technologies. The following paragraphs describe some of the research work in the virtualisation domain.

A large-scale virtualisation initiative is being done on Emulab. Instead of using a general purpose virtualisation tool like Xen, Emulab opted for the FreeBSD Jail mechanism. FreeBSD Jails provide a lightweight mechanism for virtualisation by implementing process isolation. The Emulab virtualisation approach is described in Hibler et al. [36].

User Mode Linux (UML) is a virtualisation technology that was used quite extensively in network emulation. This includes the following research efforts:

- The work done using UML (User-Mode Linux), reported on in Spenneberg [78], targeted mainly at evaluating VPN networks, UML was used to evaluate VPN protocols such as TCP and IPSec (IP Security).

- The UML-based emulator for MPLS networks (Balachander and Venkataram [5]).

- UML was also used in the implementation of VNUML (Virtual Network UML)as described by Fernandez et al. [26]. VNUML is targeted mainly at the evaluation of IPv6 routing protocols.

FreeBSD OS is also widely used for the virtualisation of network emulators. Some examples are:

- The FreeBSD network stack was virtualised through the cloning technique that allows for multiple network stacks on the same kernel as proposed in Zec [89]. This approach depends on the FreeBSD Jail (Poul-Henning Kamp [66]) framework for application environment isolation. Each instance of the protocol stack resembles a full network stack capable of running network routing protocols as well as networking applications.

- IMUNES is another example of a virtual emulator. It was proposed in Zec and Mikuc [90]. IMUNES also extends the FreeBSD kernel. The kernel was extended by introducing the capability of maintaining several networking stacks that are used to run different networking applications concurrently.

- ENTRAPID introduced the virtualisation of different 4.4BSD kernels. This enabled the deployment of different network protocol stacks on virtual kernels. ENTRAPID is described in Huang et al. [38].

Other examples of network emulators include the hypervisor-based testbed (Duchamp and Angelis [20]) aimed at conducting network security experiments, the virtual integrated TCP testbed (or VITT) aimed at evaluating TCP performance (Caini et al. [9]). Other research is looking at the possibility of using para-virtualisation as the basis for a federated PlanetLab architecture (Edwards and Harwood [21]). PlanetLab [63] is a testbed aimed at rapid prototyping and the testing of Internet-based experiments.

1.6 Organisation of the dissertation

The remainder of this dissertation is organised as follows;

Chapter 2 provides background material for this research through a discussion of the current state of virtualisation techniques. It also presents the literature review of this dissertation with special emphasis on the differences between network emulation and simulation.

Chapters 3 and 4 describe the implementation of viNEX. In Chapter 3 we present a high-level design and the architecture of viNEX. The significant components of the viNEX emulator are also described in this chapter. The foundation of the viNEX emulator is formed by the ability to model and emulate network links. Chapter 4 provides the technical details of link implementations in viNEX.

The evaluation of viNEX is discussed in Chapter 5 which includes experiments conducted on viNEX as well as a discussion of the results. The research work is concluded in Chapter 6 including the recommendations for future research work.

The paper (Mukwevho et al. [56]) that was published as part of this research appears as Appendix A. The paper describes the initial version of viNEX which was based on Xen HVM technology. We have since enhanced viNEX by changing to Xen PVM to improve network performance. Appendix B presents the source code of all the emulator configuration and management bash scripts as referenced in this dissertation.

Appendix C shows the TCP/IP reference model. Examples of modelling some common network topology structures in viNEX are presented in Appendix D. Appendix E describes the Van Jacobson experiment that we conducted using NS-2. Finally, Appendix F presents the remainder of the scripts used to conduct the Van Jacobson experiment on viNEX.

Chapter 2

Network simulation and emulation

This chapter presents the background and context of our research work. We begin by presenting an overview of network testbeds in Section 2.1. The network emulation and simulation concepts are discussed in Section 2.2. This section also presents an overview of NS-2, Emulab and Dummynet. In Section 2.3 we presents an overview of Xen [7] and its architecture. We also present an overview of how the concepts link together in Section 2.4. A summary of the original Van Jacobson experiment is presented in Section 2.5. Finally, a summary of this chapter appears in Section 2.6.

2.1 Overview of network testbeds

This section gives an overview of different network testbed environments. We consider the different categories of network testbeds and outline their benefits to the network research community. The work in this section is synthesised mostly from the report on network testbeds done by NSF (Kurose [48]).

Network testbeds play a significant role in the evolution of computer networks. Network protocols are frequently developed and experimented on a testbed environment first before being deployed in the real world. For example, the Internet protocols (such as TCP/IP) were first developed and experimented in the ARPANet [67] testbed before they became publicly available. Furthermore, both

DARTnet [18] and MAGIC (Multidimensional Applications and Gigabit Internetwork Consortium) (DARPA [17]) testbeds have played a primary role in the development of IP telephony and high-speed networks. A geographical and functional overview of the MAGIC-I testbed is presented in Figure 2.1.

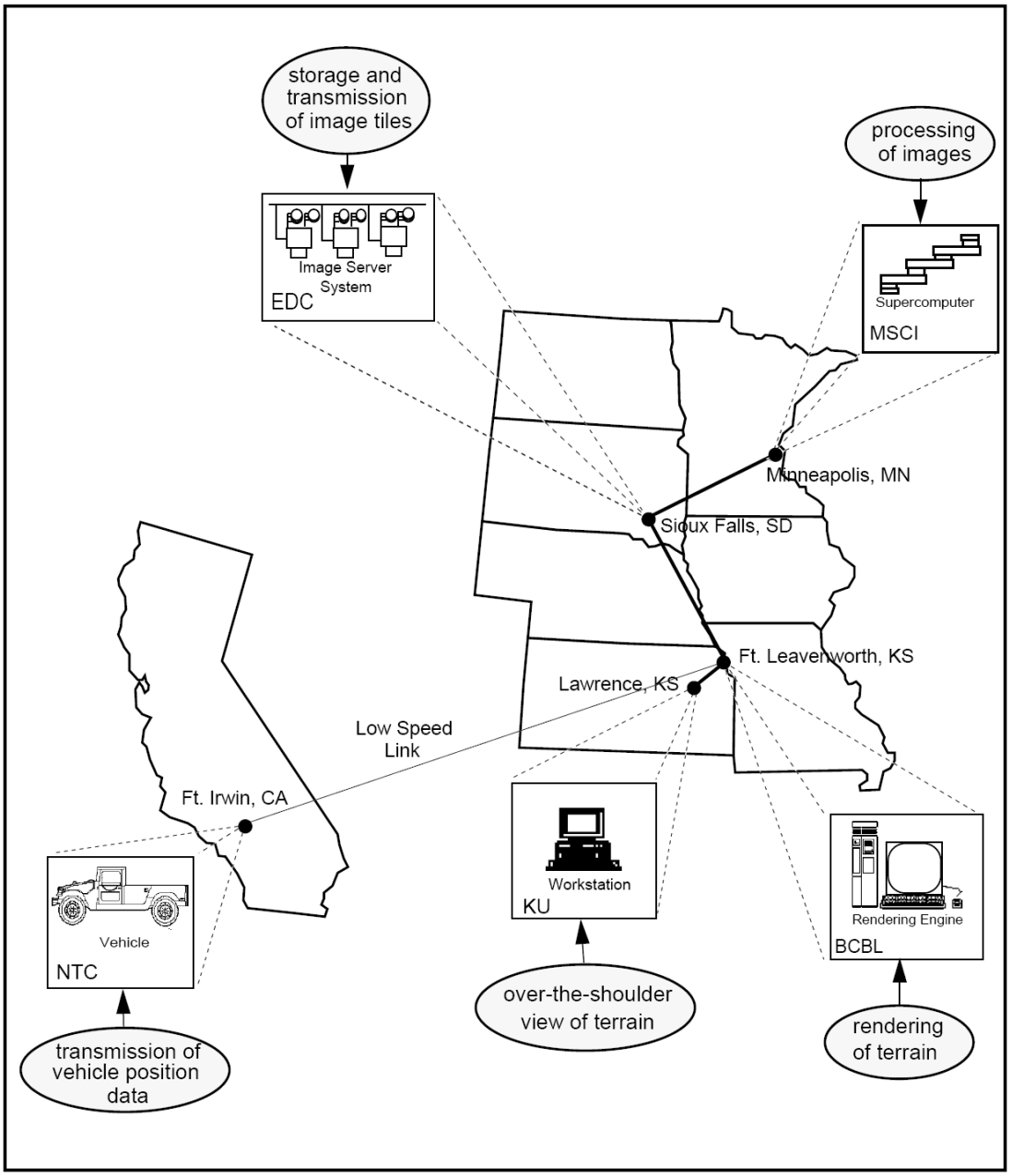


FIGURE 2.1: The topology of MAGIC Gigabit testbed, from DARPA [17].

The MAGIC testbed is formed by three major components, namely, the terrain visualisation application (TerraVision), distributed ISS (Image Server System) and

a high-speed internetwork. TerraVision creates a dynamic aerial view of the U.S. Army NTC (National Training Centre) landscape located at Fort Irwin in California. The main objective of MAGIC is to monitor the locations of vehicles at the battlefield in real time and thereby provide real time monitoring of the battlefield to a commander. Coordinates of the vehicles are transmitted to the BCBL (Battle Command Battle Laboratory) where the rendering of the terrain occurs — see Figure 2.1. Images are then transmitted over the high-speed network to EDC (Earth Data Centre) where they are stored on the ISS. TerraVision interacts with the ISS in order to retrieve the image tiles that form the rendering of the battlefield. If the requested tile is not stored on the ISS, the ISS sends the raw image data to the supercomputer at MSCI (Minnesota Supercomputer Center Inc.) where it is processed into tiles. The ISS then transmits the tiles to TerraVision for over-the-shoulder view of the terrain on a workstation located at KU (University of Kansas). A detailed description of MAGIC is given in DARPA [17].

Network testbeds may be classified into two main categories, namely, testbeds that require production quality network services and testbeds that allow some room for disruptive experimentation on the network infrastructure (Kurose [48]). The latter are also known as research testbeds. The following paragraphs briefly describe each category:

- **Testbeds that require production quality network services:** These testbeds are dedicated to specific applications and are normally characterised by high network quality features. Examples of network quality features are *availability*, *transmission speed*, *bandwidth* and *propagation delays*.

  The Internet2 [41] is an example of this class of testbeds. In the Internet2, the IP network has 99.999% uptime a year (which equates to a downtime of less than 5.5 minutes) and over 10 GigE (Gigabit Ethernet) bandwidth capacity.

- **Network research testbeds:** These testbeds are used for experimental research in computer networking. Our network emulator (viNEX) belongs to this category. Other examples include Netbed (White et al. [84]), Emulab (White et al. [85]) and PlanetLab [63].

Since viNEX is an example of a network research testbed, we discuss this category further below.

2.1.1 Network research testbeds

Network research testbeds may be distinguished into two broader classes:

- **Multi-user, experimental facilities (MXF):** These type of testbeds are designed to serve a defined set of user communities of network researchers. For example, the MOSIS (Metal Oxide Silicon Implementation Service) networking system provides university researchers with access to fast-turnaround semiconductor manufacturing services. MOSIS was developed at Xerox PARC and institutionalised by DARPA (Connway [14]).
- **Proof of concept testbeds (PCT):** These testbeds are aimed at addressing a specific research objective. PCT testbeds are normally dismantled once the research objective has been achieved. An example of a PCT is the SEQUIN project (IST [44]) to check the feasibility of the proposed IP-Premium implementation in production networks, with a strong emphasis on end-to-end quality of service (QoS) delivery across networks.

Traditionally, network testbed components (such as links, nodes, routers and switches) were based in a facility at a single geographical area. An example is the original Netbed (White et al. [84]) facility at the University of Utah. In response to the evolving network research community needs, such as high availability, dynamic configuration and control of the experiment environment, four additional types of network testbeds have emerged. These are cluster, overlays as well as federated and network research kits. A brief description and an example of each type follow:

- **Clustered testbeds:** These type of testbeds are based mainly on the concept of network emulation. They are normally formed by a cluster of network elements deployed in a laboratory facility with remote access provided through an Internet interface. An example of this type is Emulab (White et al. [85]). In Emulab, network resources are deployed in a laboratory at the University of Utah. Researchers are provided with a remote interface via the Internet to access and configure their experiments.
- **Overlay testbeds:** These testbeds are designed to be an overlay on an existing network. Some examples of overlay testbeds are:

  - The original ARPAnet which was designed as an overlay over a telephone network (Hauben [33]).
  - MBone (Savetz et al. [73]) which was designed as a multicast overlay over the public Internet.
  - Netbed (White et al. [84]) and PlanetLab [63] which both have components that communicate over the public Internet across sites.

- **Federated testbeds:** This type is formed by interconnecting a number of independent testbeds. Some benefits of federated testbeds are:
  - Through the federation, researchers are exposed to more hardware and are therefore able to run larger scale experiments than what their local hardware can support.
  - Researchers are exposed to a variety of networking technologies in addition to their locally supported technologies. For example, a remote testbed may have wireless functionality that can be accessed via the federation mechanism.
  - Federated testbeds enable efficient and optimal use of testbed resources. Researchers can make use of any idle testbed resources and thereby facilitate efficiencies.

  CONET [13] is an example of a federated testbed. It is composed of a federation of collaborative research clusters which are located at different geographical locations.

- **Network research kits:** Network research kits are composed of a set of hardware and software components that can be deployed on a local network laboratory. Some examples include:
  - Our emulator viNEX which is designed to run on a single computer host, serves as a good example of this category.
  - IMUNES (Kukec et al. [47]) being a network emulator that can be installed on a single FreeBSD host.
  - Wireless Sensor Network Kit (Crossbow [16]) which is a commercial classroom kit for teaching wireless sensor and mesh networks. It is deployed in a classroom environment through a USB device plugged into students' PCs.

2.2 Network emulation and simulation

In this section we present some *emulation* and *simulation* concepts in the context of computer networks.

Network *simulators* are usually implemented as a collection of software modules, providing a synthetic environment for conducting network experiments. Simulators are often deployed and executed on a single host. NS-2 [52] is a popular example of a network simulator in the network research domain. Simulator experiments are normally configured using text files. For example, in NS-2 one text file is used to configure the entire network experiment. As a result, experiments are repeatable and more control is provided to the user. Simulators do have some disadvantages though, since protocols are developed and tested in a synthetic, simulated environment interacting with simulated components. However, some realism is lost and a significant amount of code changes might be required to move the developed artifacts into a live network.

Network *emulators* are constructed by combining elements of live networks and simulated software with the objective of imitating a larger network. For example, a larger WAN network can be emulated by using few network nodes deployed in a laboratory environment. Some special nodes can be inserted between network links and simulator software can be loaded to simulate some WAN network conditions. The Dummynet (Rizzo [70]) module has been used extensively in emulated network research environments (such as Emulab (White et al. [85])) to emulate network conditions such as delays, random losses and bandwidth limitations. One of the key distinctions between network emulation and simulation is that network emulation runs real protocol stacks on the network nodes while simulators normally use software models.

Emulab [22] is an example of an emulated environment. In Emulab, some transparent FreeBSD delay nodes are inserted between topology links to simulate the network boundary conditions using the Dummynet module. NS-2 is an example of a limited network emulator. Limited emulation functionality was recently introduced to NS-2 whereby real network traffic can be subjected to the simulated network components. The emulation facility of NS-2 is described in Fall [24].

2.2.1 Overview of Network Simulator 2 (NS-2)

NS-2 (McCanne et al. [52]) is a discrete event-driven simulator written in C++ (ISO/IEC [43]) and OTcl (Heidemann et al. [35]) used for conducting network research. NS-2 was developed at the University of California Berkerely; it is currently at version 2 (commonly referred to as NS-2) and version 3 is currently under development.

NS-2 works at the network packet level and it provides software mechanisms to simulate wired and wireless networking protocols. The NS-2 package is comprised of object-oriented classes to implement protocols, traffic generators and router queue management techniques. NS-2 implements network protocols such as ATM (Asynchronous Transfer Mode), IP (Internet Protocol), TCP, FTP (File Transfer Protocol), GSM (Global System for Mobile Telecommunication), and GPRS (General Packet Radio Service). NS-2 also implement objects to simulated traffic generators and sinks for network protocols like FTP, CBR (Constant Bit Rate), HTTP (Hypertext Transfer Protocol), HTTPS (Hypertext Transfer Protocol Secure) and Telnet. NS-2 also implements router queue management schemes such as DropTail, RED (Random Early Detection) and FIFO (First In First Out). NS-2 also includes mechanisms for simulating LANs (Local Area Networks) and WANs (Wide Area Networks) by implementing multicasting and MAC (Media Access Control) layer protocols.

NS-2 was originally targeted at Unix-based platforms (such as Linux and BSD versions). The recent Cygwin extensions for Windows made it possible to host NS-2 on a Windows-based platform. Network experiments are modelled using the Tcl scripting language. NS-2 is quite popular in the network field, for example its Tcl scripting language syntax was adopted as the modelling language for both Emulab and Netbed environments (White et al. [85]).

The NS-2 product suite contains a tool called NAM (Network ANimator), which is a Tcl/Tk-based tool used to animate the network packet traces. NAM works from the trace files generated as output from running an experiment in NS-2. The trace file contains network topology details which include nodes, links and packet traces.

2.2.2 Emulab

Emulab (White et al. [85]) is an emulated network testbed. The primary installation of Emulab is managed by the Flux Group and it is hosted in the School of Computing at the University of Utah [22]. Emulab is a subset of Netbed (White et al. [84]) and it is aimed at offering emulation capabilities. At the highest level, Emulab consists of a physical testbed environment and special FreeBSD delay nodes to simulate different network conditions. The testbed is formed by a collection of PCs which are interconnected by network switches in a manner that allows for arbitrary network topology configurations. The testbed PCs can be used as end-nodes, traffic generators, traffic consumers, routers or link emulators.

Emulab is formed by a collection of different networking environments for conducting research. Some of the supported environments are:

- **Emulation:** Emulated experiments enable researchers to create an arbitrary network topology, allowing for a controllable, predictable, and repeatable environment. Researchers also have full `root` access to the PC nodes. Emulab furthermore adds the flexibility by allowing users to load an operating system of their choice onto the PC nodes. Details are described in White et al. [85].

- **Live-Internet Experimentation:** Emulab is integrated with the RON (Andersen et al. [2]) and PlanetLab [63] testbeds through the live Internet. Such level of integration provides researchers with an opportunity to run their experiments across the live Internet around the world.

- **802.11 Wireless:** Emulab has a wireless testbed with devices complying to the 802.11a/b/g IEEE wireless specifications (IEEE [40]). Emulab provides a web interface which enables users to have full remote access and control to mobile devices. The mobile testbed (known as DETER) is comprised of a set of mobile nodes with two wireless interfaces each, and a wired control network.

- **Software-Defined Radio:** Emulab makes use of the USRP devices from the GNU Radio project (GNU [29]). These USRP devices provide control over the physical network layer (OSI Layer-1) of the above wireless network.

- **Sensor Networks:** Emulab has a sensor-network testbed which is formed by 25 Mica2 motes. A mote is a serial port that enables maximum control and debugging capabilities (Emulab [22]).
- **Simulation:** By making use of the NS-2 emulation facility (Guruprasad et al. [32]), simulated experiments can interact with a real network. Therefore, simulators can also interact with Emulab, including some network components within Emulab.

Emulab's software may be bought and installed at a different network site. In addition to the primary site located at the University of Utah, the software has been shipped and installed at more than 24 sites located around the world — see Figure 2.2 for other locations. Emulab is a public facility and it is available to researchers and students in the academic fields free of charge.

FIGURE 2.2: Other Emulab sites around the world, from Emulab.net [23]

Experimenters can use NS-2 scripts or a graphic tool to define network topologies for their experiments. NS-2 scripts are formed by a sequence of commands that are defined using a specific notation. The notation is referred to as NS-2 *syntax*.

Using the NS-2 *syntax*, researchers can specify traffic generators and sinks as well as discrete network events. Listing 2.1 gives an NS-2 example script for Emulab

and the resulting topology is depicted in Figure 2.3. Once the topology is defined, the next step in Emulab is to load the experiment in a process called swap-in. Emulab automatically maps the *virtually-defined* topology to physical resources.

```
1 set ns [new Simulator]
2 set node1 [$ns node]
3 set router [$ns node]
4 set node2 [$ns node]
5 set linkA [$ns duplex-link $node1 $router 100Mb 0ms
     DropTail]
6 set linkB [$ns duplex-link $router $node2 1Mb 10ms DropTail]
7 $ns rtproto Static
8 $ns run
```

Listing 2.1: A sample source NS-2 script for creating the Emulab experiment topology in Figure 2.3 (White et al. [85]).

Listing 2.1 is an OTcl script which creates a three-node network on Emulab. The effect of the code in Listing 2.1 is described next.

Line 1 creates a Simulator object and assigns it to a variable `ns`. The `Simulator` object is used as a global object for configuring a network experiment. It is responsible for storing references to experiment nodes, links, event schedulers and basic protocol configuration settings (e.g packet size). Lines 2 to 4 reserve, create and boot the experiment nodes. The two network links, namely, `linkA` and `linkB` are created by lines 5 and 6 respectively. Line 5 creates a network link between `node1` and the `router` and calls it `linkA`. `LinkA` is configured as a duplex link with a `100 Mb/s` capacity, a `0 ms` delay and it uses the DropTail queuing strategy. `LinkB` is the link between the `router` and `node2` and it is defined to carry traffic using a `1 Mb/s` bandwidth, a `10 ms` delay and likewise uses the DropTail queueing strategy. Line 7 specifies that no routing protocol will be used for this experiment and therefore network addresses will be statically assigned during boot time. Finally, the execution of the script is triggered by the `run` command at line 8.

In Emulab, *network link emulation* is provided by special nodes called *link emulators* which are inserted between the end-points of a network link. Link emulators run the FreeBSD OS with Dummynet enabled for emulating network link conditions such as bandwidth control, propagation delay and packet losses (White et al. [85]). Dummynet is also used to implement various network link-queuing strategies like FIFO, RED, or DropTail.

A network experiment in Emulab is comprised of two distinct networks, namely, the *control network* and the *experiment network*. At the heart of Emulab lies the *control network* as depicted in Figure 2.3. The purpose of the control network is to provide an isolated network for controlling and managing the experiments without interfering with the network traffic of the experiment. The *experiment network* is only limited to each specific experiment. Applications running inside the topology nodes are not aware of the main control network. Topology nodes make use of the control network to gain to the shared network resources such as the NFS mounted file systems `/proj` and `/users`. Once an experiment has been uploaded into Emulab, researchers have two possible methods of accessing their experiments. These are:

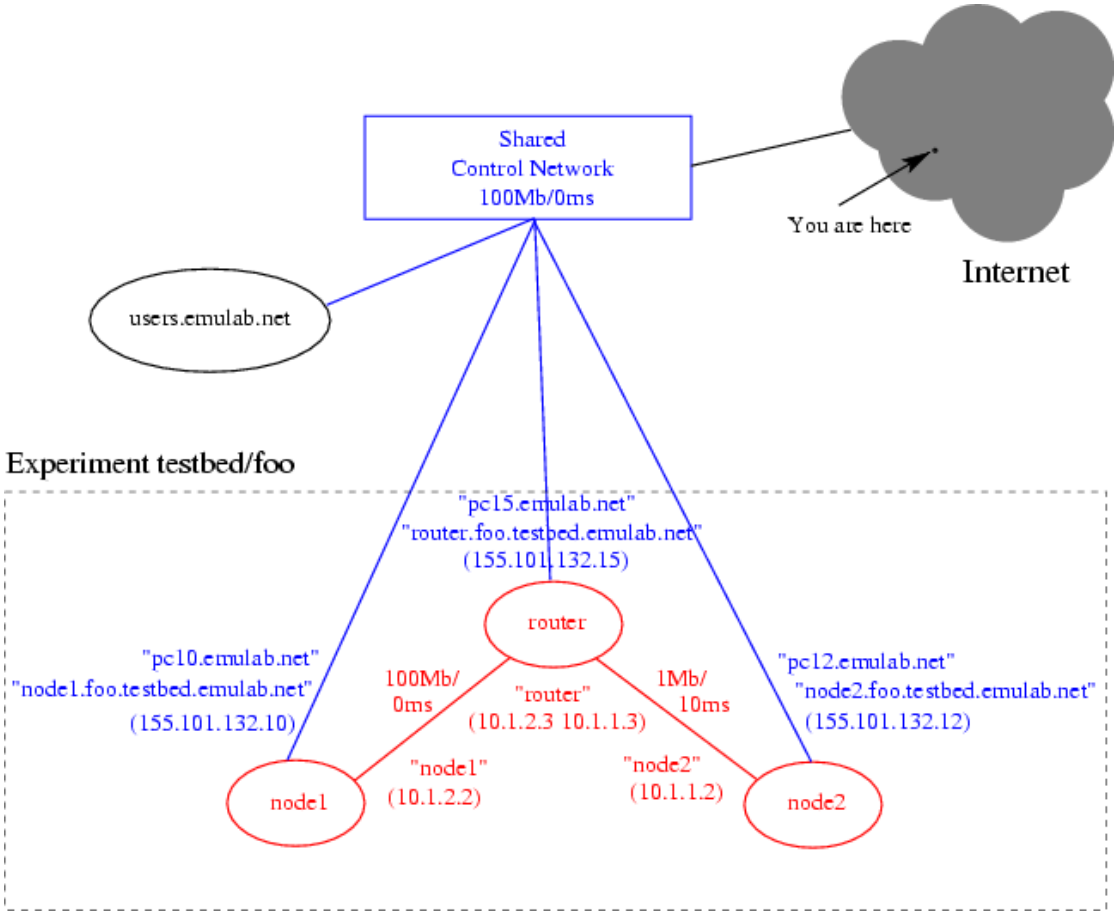


FIGURE 2.3: Emulab experiment topology, from Emulab [22]

- **Employing a web interface:** This is the primary method of access — users can access Emulab using their web browser application by navigating to the URL `http://www.emulab.net/`. They will be required to enter their

login credentials before being granted access to their experiments. The web interface is used mainly to load, configure and to execute experiments.

- **Using SSH (Secure Shell) via host `users.emulab.net`:** This mechanism is used mainly for obtaining direct access to the file systems and command console of the topology nodes. To obtain access to the topology nodes, users must first access the `users.emulab.net` node using SSH. From this node, users can then use SSH to access the file systems of topology nodes. The SSH access method also provides users with a direct access to the trace files. This method enables users to issue direct OS commands to the nodes, allowing them to execute network utilities such as `traceroute` or `tcpdump`. The `traceroute` utility may be used to check connectivity between two topology nodes. `Tcpdump` is used to capture network packets passing through a network interface.

Next we present an overview of Dummynet.

2.2.3 Dummynet

Dummynet (Rizzo and Carbone [71]) is a tool that was initially designed for testing network protocols. It is now being used primarily for network bandwidth management. Dummynet is used to simulate network properties such as queue and bandwidth limitations, delays and packet losses. The implementation of Dummynet on FreeBSD is aimed at TCP and IP protocols. Dummynet operates on the inbound or outbound packets between the TCP and the IP layers. Dummynet operation is shown in Figure 2.4.

Dummynet is implemented as a loadable kernel module on the FreeBSD [27] operating system. By design, Dummynet operates between any pair of network layers, as shown in Figure 2.4. It intercepts packets that are being output from one layer and then it applies some network property on the packets before passing the packets to the next layer. Dummynet can intercept network packets at either layer 2 or layer 3 of the OSI reference model (Zimmermann [91]). The following paragraphs discuss the operation of Dummynet.

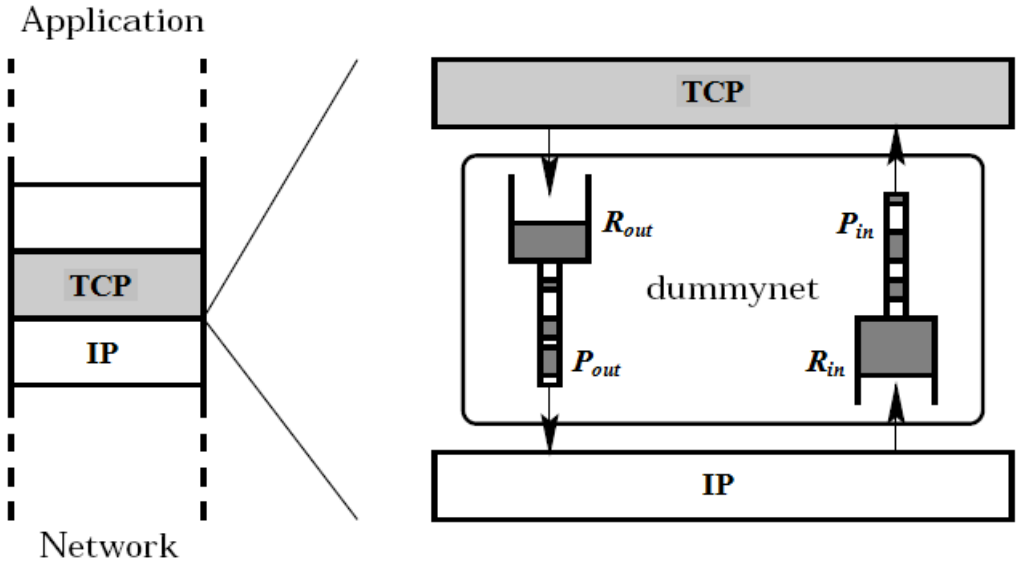


FIGURE 2.4: Dummynet operating principle, adapted from Rizzo [70]

Dummynet uses the notion of a *pipe* to regulate network traffic. A pipe can be configured with settings such as bandwidth, packet delay and loss rate. A pipe is modelled by two queues, namely, the R and the P queues as shown in Figure 2.4. A a pair of queues R/P is required in each direction of the communication. The R_{in}/P_{in} pair is used for processing inbound traffic. The R_{out}/P_{out} pair is used for processing outbound network traffic. To simplify our discussion, we shall refer to these queues simply by using the R and the P symbols. Assuming t_p to be the link delay, B the maximum link bandwidth and r the probability of packets getting lost, the operating principles of Dummynet are:

- **Bandwidth management:** The R queue is used to control bandwidth. Packets are moved from the R to the P queue at the maximum rate of B bytes/s. Bandwidth control is achieved by setting or adjusting the size of the R queue.
- **Queuing algorithms:** Modelling of network link-queueing algorithms such as FIFO, Droptail or RED is achieved by setting the queueing algorithm of the R queue.
- **Link delays:** The P queue is used to model link delays. Packets are buffered on the P queue for a maximum duration of t_p before being passed to the next protocol layer. This procedure implements the configured link delay.

- **Packet loss:** Random packet loss is implemented before passing the packets to the next protocol layer from the P queue. Packets are dropped using the configured random probability (r).

In FreeBSD, the IP Firewall (`ipfw`) is used to filter and route traffic to the Dummynet pipes. The command-line tool `ipfw` provides an interface to create firewall rules for matching packets destined for Dummynet as well as to configure pipe settings.

The following section presents the design and features of the Xen [7] virtualisation technology.

2.3 Xen Overview

Xen (Chisnall [10]) is an open source virtualisation platform based on the *hypervisor* (Singh [76]) technology. The Xen Hypervisor is also known as the VMM (Virtual Machine Monitor). Xen originated at the University of Cambridge as part of a research project led by Ian Pratt. Xen is currently developed by the Xen open source community comprising of engineers from technology vendors. A list of these technology vendors can be referenced at (Xen [87]).

During the initial phases of this research we investigated a number of virtualisation platforms as part of identifying a suitable platform for implementing viNEX. Some of the virtualisation platforms examined include Vmware, VPN and Xen. Xen was selected as the platform for implementing viNEX due to its open source nature and the active status of its research community. UML is also open source but was not chosen because of its limited community activity.

Xen supports both *para-virtualisation* and *full virtualisation*. The initial releases of Xen were targeted mainly at supporting *para-virtualisation* guests on x86 architectures, allowing for a maximum of 100 guests running concurrently (Barham et al. [6]). Support for unmodified guests (*full virtualisation*) was introduced later on by Intel after they extended the Xen VMM to use their Intel Virtualisation Technology (VT) (Dong et al. [19]). AMD [1] followed suit by creating extensions for Xen to run on their hardware virtualisation technology, known as Pacifica or AMD-V. Xen can run on a vast majority of architectures which include IA-32,

x86, x86-64, IA-64 and PowerPC 970, AMD [1] and SPARC (ported by Sun Microsystems [79]).

We now consider the Xen architecture and present an overview of its internal components.

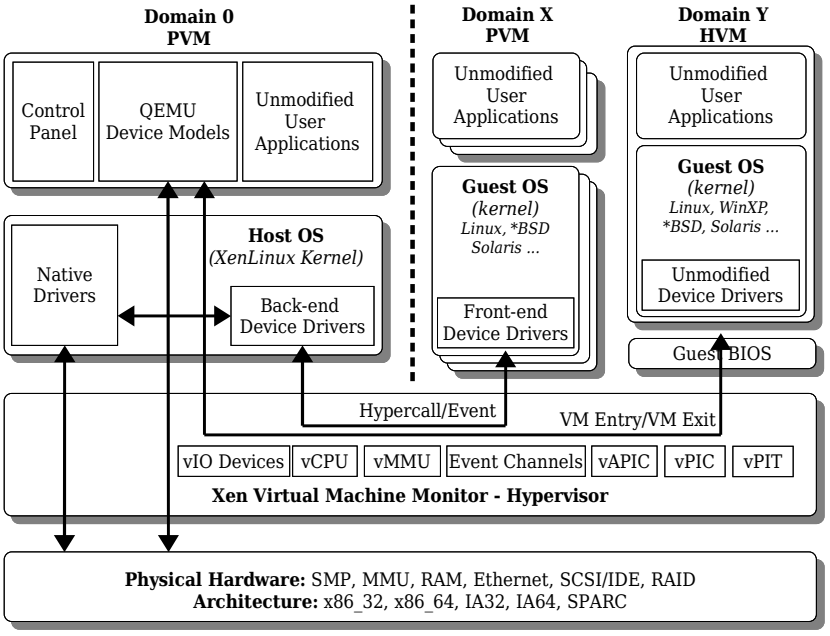


FIGURE 2.5: The Xen 3.x Architecture Including both PVM and HVM Guests, adapted from Barham et al. [6]

2.3.1 Xen Architecture

Figure 2.5 shows a high level Xen architecture and its significant components. The main components of a Xen architecture are the Xen Hypervisor (VMM), Domain-0 (`Dom0`) and Domain-U (`DomU`) which can either be a PVM (Paravirtualised Virtual Machine) or a HVM (Hardware Virtual Machine) guest.

The above components are discussed next.

2.3.1.1 Xen Hypervisor

The Xen hypervisor, also known as the VMM provides the core virtualisation functions that make guest operating system hosting possible. It runs inside the special privileged domain known as *Domain-0* or *Dom0*. The *hypervisor* provides

a virtualisation layer that controls access to the underlying physical resources. The virtualisation functions are provided by devices (vIO Devices), memory (vMMU — virtual Memory Management Unit) and CPU virtualisation (vCPU). The vAPIC (virtual Advanced Programmable Interrupt Controller) module is responsible for implementing virtual APIC (Advanced Programmable Interrupt Controller) devices. Similarly, the vPIC module provides virtualisation of the PIC devices. The vPIT module provides a virtual interface to the platform's PIT (Programmable Interrupt Timer). All these modules are depicted in Figure 2.5. A discussion of the vAPIC, vPIC and vPIT modules is beyond the scope of this dissertation and further details are presented in Chisnall [10].

An overview of the three main modules of the VMM is presented next.

- **Device Virtualisation** is provided by the *vIO Devices* module which is responsible for the underlying virtual I/O devices. It delegates all the I/O requests from the guest domains to the physical I/O devices of the system. The I/O devices include a keyboard, a mouse, network cards and a hard disk. *Event Channels* provide a mechanism for the direct communication between the *frontend device drivers* and the *backend device drivers* of paravirtualised guests.

- **Processor Virtualisation** is provided by the *vCPU* unit. The vCPU provides the abstraction of the the physical CPU to the guests. It is responsible for executing, suspending and resuming the CPU instructions from each guest. It delegates the unsafe CPU instructions of the guests and run them securely on the underlying CPU.

- **Memory Virtualisation** is provided by the *vMMU (virtual Memory Management Unit)* module which is responsible for sharing the physical RAM among the guests. It provides the abstraction of the hardware MMU (Memory Management Unit) to the guest operating systems. This is achieved by maintaining a special table called the *shadow page table*. The guests use the GPA (Guest Physical Address) to reference memory pages. The vMMU module then translates the GPA into the corresponding physical machine address (known as the MPA) to reference the physical memory page. The shadow page table is responsible for keeping the mappings between the GPAs and MPAs. Further details of the memory virtualisation process appear in Dong et al. [19].

2.3.1.2 Domain-0

The `Domain-0` virtual machine is normally referred to as `Dom0`. It provides the management and control functions of the Xen environment. The Xen hypervisor (VMM) is hosted by the kernel running inside `Dom0`. `Dom0` has the highest system privileges and as a result, it is the only domain that is allowed direct access to the underlying physical resources (Barham et al. [7]). `Dom0` is always started by default when a XenLinux kernel boots up. It runs the control panel module which is responsible for creating, destroying and managing the guest domains (DomUs).

2.3.1.3 PVM Guest Domains

PVM guest domains are paravirtualised machines (DomUs) running on top of the hypervisor. Each DomU runs a modified kernel ported to run on Xen. To port a kernel to Xen, system calls of the kernel are replaced by special calls which invoke the Xen VMM functions. PVMs use these system calls (also known as *hypercalls*) as a mechanism for communicating with the hypervisor. Access to hardware devices is provided through special frontend device drivers. Frontend device drivers are Xen-aware device drivers that make use of event channels to communicate to the backend device drivers in `Dom0`.

2.3.1.4 HVM Guest Domains

HVM guest domains run unmodified versions of the OS. There is no need to port the OS to run on the Xen platform. The HVM relies on the guest BIOS provided by the hypervisor to boot-up. The guest BIOS does not have direct access to the physical hardware, instead, it uses the QEMU device models to process hardware requests. The HVM communicates with the hypervisor through the VM Entry and VM Exit requests. HVM guests run standard native device drivers which rely on the guest BIOS emulation.

A detailed description of HVM and PVM domain types is beyond the scope of this dissertation, but details appear in Barham et al. [7]. For the rest of this dissertation the term `DomU` will be used to refer to a Xen *guest* domain.

2.3.2 Xen Networking

Dom0 is the only domain which is allowed direct access to the physical network interfaces. All guest domains have to forward their network packets via Dom0. All network packets between the guest domains and Dom0 must pass through the Hypervisor. The network interfaces inside the guest domains are controlled by the frontend device drives and they are therefore known as *frontend* interfaces. Network interfaces running inside Dom0 and linked to the guest domain's frontend interfaces are called *backend* interfaces. For each frontend ethernet interface (ethX), a corresponding backend interface vifX.Y is created in Dom0 where X is the guest domain id and Y uniquely identifies the backend interface among all the backend interfaces allocated for DomX (see Palivan [62]). The Xen hypervisor provides communication between the frontend and backend devices through the I/O channels — the low-level implementation details can be referenced in Govindan et al. [31].

Xen provides two models for network communication between guest domains and Dom0, namely, *routed* and *bridged* networking. Both models can be used with Network Address Translation (NAT) in order to enable external forwarding of network packets to, or from guest domains. The *routed* and *bridged* networking models are shown in Figure 2.6 and Figure 2.7 respectively.

The next two sections describe the routed and bridged models further. Further details on the operation of each model may be observed in Palivan [62]:

2.3.2.1 Routed Network

The *routed* network model uses Dom0's IP routing module for routing DomU network packets. A routing table entry must exist for each guest domain for packets to be forwarded to that domain. This model is illustrated in Figure 2.6. For packets to flow to, and from the guest domain, each backend interface linked to the guest domain's frontend interface must have an IP address assigned to it. Packets are routed by Dom0 using the native Linux routing modules at layer-3. Routing is performed at the IP level and no bridging is involved. This model becomes more complex as the routing tables grow in size and may become difficult to manage.

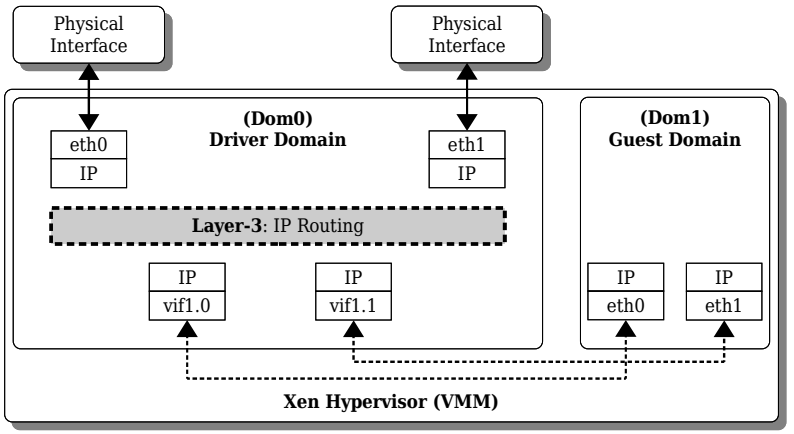


FIGURE 2.6: The Xen **Routed** Network Model, adapted from Barham et al. [7]

2.3.2.2 Bridged Network

Xen uses network bridging by default. When the hypervisor starts, the original ethernet (`eth0`) interface is shut down, renamed to `peth0` and brought up again. The real ethernet interface is referenced using `peth0` as illustrated in Figure 2.7, see Barham et al. [7]. A Linux software bridge is created for each ethernet interface. The bridge is used to link the backend interfaces to the real ethernet (`pethX`) interfaces to enable exterior packet forwarding. The bridge forwards packets to the guest domains using the MAC addresses of their backend (`vifX.Y`) interfaces. Bridges (Buytenhek [8]) are layer-2 devices and therefore perform routing using

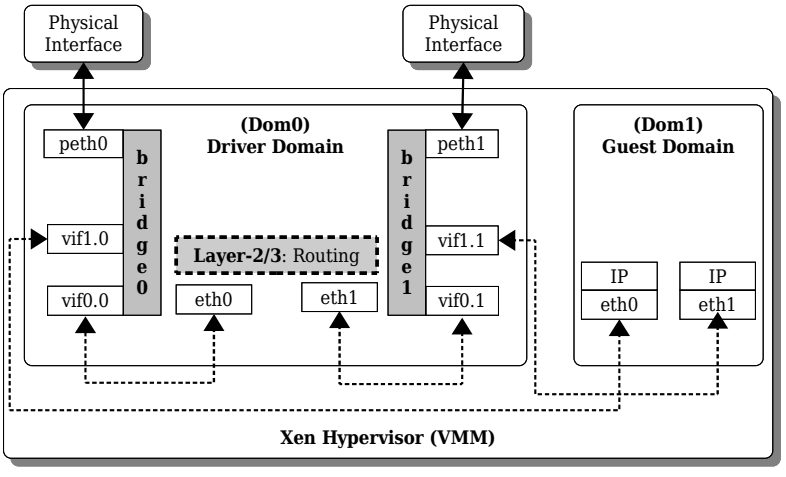


FIGURE 2.7: The Xen **Bridged** Network Model, adapted from Barham et al. [7]

MAC addresses. Network frames are forwarded based on Ethernet addresses and not IP addresses. As a result, since forwarding is done at layer-2, all protocols may pass their traffic transparently through the testbed. It is for this reason that the testbed can be viewed as a network backbone on which network traffic may be routed and the protocol-specific functions performed at the topology nodes. In this research, we have modified the default behaviour of the Xen bridged networking in order to create dynamic links between guest domains. Further details on network links are presented in Chapter 4.

2.4 Linking the concepts

In this section we present a diagrammatic view of the relationships among the concepts discussed in this chapter. These relationships are depicted in Figure 2.8. The links in Figure 2.8 represent "used by" relationships.

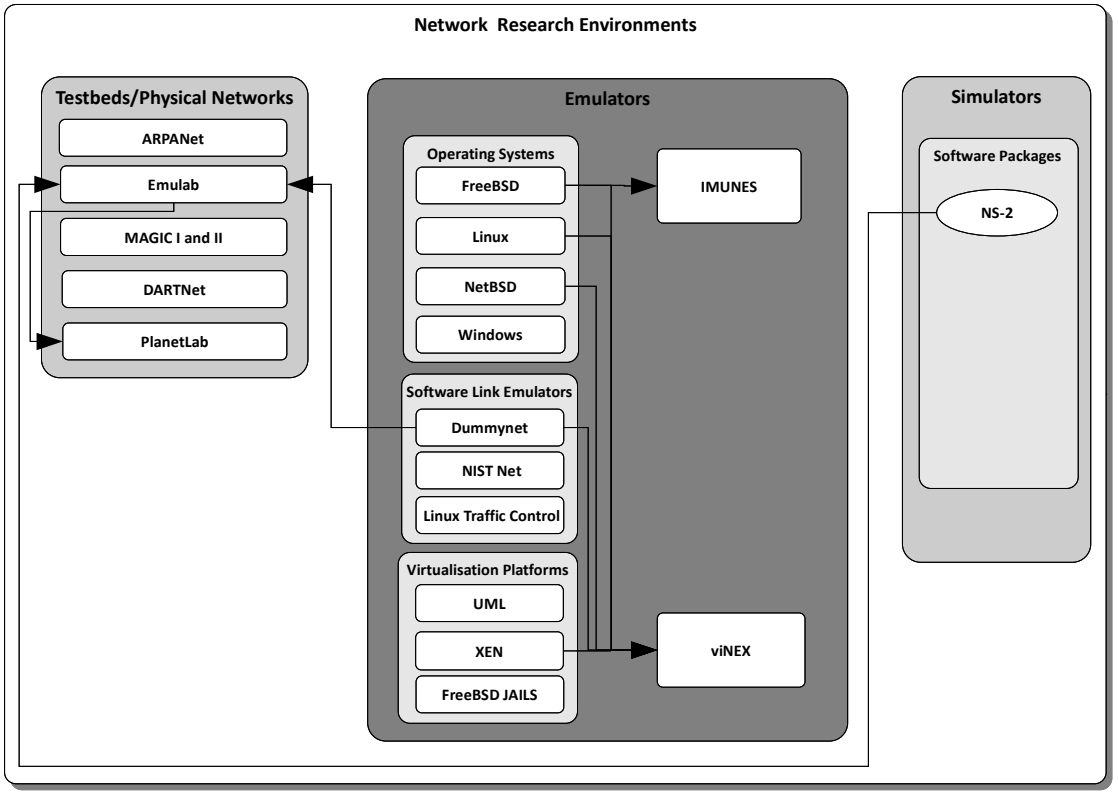


FIGURE 2.8: Linking together the concepts discussed in this chapter

We may summarise Figure 2.8 as follows. Network research environments can be divided into three categories, namely testbeds, emulators and simulators. Testbeds are normally based on physical networks. Examples of testbeds include Emulab (White et al. [85]), ARPANet (Hauben [33]), MAGIC (version I and II) (DARPA [17]), DARTNet [18] and PlanetLab [63].

Emulators are frequently designed to run on a single machine. For example, viNEX (Mukwevho et al. [56]) was constructed using the Linux, FreeBSD and NetBSD operating systems and the Dummynet package was used for link emulation. All the components of viNEX were hosted on a single server by using the Xen [7] virtualisation platform. Software simulators form a class of network research environments which is based on software models. NS-2 is an example of a software simulator.

It is also important to note the relationships amoung the above three research environments. Testbeds can make use of emulators to implement the emulation of network links. An example of this association is the use of Dummynet in Emulab for link emulation. Furthermore, testbeds can also make use of simulator components. The NS-2 OTcl scripting language was used in Emulab as the modelling language for defining network experiments.

It is also possible for simulators to communicate with real network testbeds. For example, part of the VINT/NS project is to introduce emulation to the NS-2 simulator by developing interfaces that enable the simulator to communicate with real network nodes. Further information on the VINT/NS project can be referenced in Fall [24].

In our research, we have identified the Van Jacobson [81] experiment as an example of a class of network experiments that can be conducted on viNEX. This experiment is used extensively as part of the justification of our hypothesis in Chapter 5 and Chapter 6. We therefore provide an overview of the original experiment in the following section.

2.5 Summary of the Van Jacobson Experiment

The Internet experienced a sequence of collapses due to network congestions starting from October 1986 (Van Jacobson [81]). As a result, the Internet bandwidth

dropped significantly. For example, the bandwidth between the LBL (Lawrence Berkeley National Laboratory) and UCB (University of California Berkeley) sites dropped from 32 *Kbps* to 40 *bps*. The two sites (LBL and UCS) are separated by 365.76 metres and three hops between them (Van Jacobson [81]).

Van Jacobson [81] subsequently conducted his experiment in 1988 to investigate the possible cause of the above collapses. The investigation was focused on studying the 4.6BSD TCP stack under severe network conditions. A microwave bottleneck link was introduced between the two sites and all traffic was routed past this bottleneck link. The network topology used in the Van Jacobson experiment is depicted in Figure E.1 of Appendix E.

In his paper Van Jacobson [81] describes five algorithms which were constructed as a result of the initial attempts to resolve the above Internet collapses. The five algorithms described by Van Jacobson are: round-trip-time variance estimation, exponential retransmit timer backoff, slow-start, aggressive receiver acknowledgement and dynamic window resizing on congestion.

Van Jacobson conducted his experiment twice by using a different TCP stack for each iteration. The two TCP stacks used were the original 4.3BSD TCP stack (as described in RCF793 (Postel [64]) without congestion avoidance and control) and the enhanced TCP stack (with the above congestion algorithms implemented).

The result of the Van Jacobson experiment showed a significant improvement in the utilisation of the bottleneck link between the UCB and LBL sites. The improved utilisation confirmed that the new TCP with congestion control algorithms enabled, performed much better when compared to the old 4.3BSD TCP stack without congestion control. The results of the experiment indicated that only 1% of all packets were retransmitted using the new TCP stack whereas the results of the old TCP stack showed a 50% retransmission of packets. The results of the old and the new TCP stacks are shown in Figure 8 and Figure 9 respectively in the Van Jacobson paper [81].

We conducted the above Van Jacobson experiment on viNEX as part of verifying and validating our emulator. Details of conducting the Van Jacobson experiment are presented in Chapter 5.

2.6 Summary

Network testbeds play a primary role in the evolution of computer networks. The ARPAnet (Pullen et al. [67]) testbed was used to develop and test the original Internet protocols including TCP and IP. Testbeds were normally constructed as a real network spanning a large geographic area, for example, the MAGIC-I Gigabit testbed was deployed across a major part of the Unites States of America, see Figure 2.1.

As the needs of researchers evolved, new types of testbeds were constructed. Recent advances in computing, such as the Dummynet emulator have made it possible to emulate wide area networks on a single PC. It is possible to host a cluster of network nodes in one building and use the software simulator to mimic the behaviour of wide area networks. For example, the Emulab and Netbed (White et al. [85]) testbeds consist of a cluster of PCs and network switches hosted in one building. This was primarily driven by the need to have a freely accessible research facility for network researchers across the world. The FreeBSD nodes with Dummynet loaded have made it possible for the Emulab PCs to emulate a wide area network despite of having all nodes located in one room.

This chapter also presented the two main classes of network testbeds, namely, *testbeds that require production quality network services* and *network research testbeds*. Our interest is focused on the *network research testbeds* class because these are used by network researchers for creating new knowledge in computer networks. Network research testbeds are further divided into two broader classes, namely, the *multi-user experimental facilities (MXF)* and *proof-of-concept testbeds (PCT)*. In response to the growing need of network researchers, four new classes of network research testbeds have emerged. These include the *clustered, overlay, federated and network research kits testbeds*. Our emulator OS belongs to the network research kits class, since it consists of a collection of software modules that are deployed on a single host. The viNEX emulator differs from existing emulators (such as IMUNES) on the basis that the nodes run real operating systems with full network protocol stacks as opposed to simulated stacks.

The advances in computer virtualisation have made it possible to create a network of virtual nodes on a single host computer. As a result, we have seen some emerging research initiatives aimed at exploring the possibility of hosting an entire network testbed on a single host. For example the IMUNES (Zec and Mikuc [90]) system

is a network emulator that runs on a single host. The approach used in the IMUNES is different to ours (Kukec et al. [47]). In the IMUNES, network nodes are represented by the cloned network protocol stack, whereas our approach is to use a traditional virtualisation platform such as Xen.

We also discussed aspects of the Xen hypervisor virtualisation technology. The discussion on Xen included an overview of its architecture and network models. Xen supports two types of networking, namely, routed and bridged mode. The bridged mode is the default and the most preferred owing to its simplicity.

Finally, in Section 2.4 we presented a high-level overview of the concepts discussed in this chapter and how they link to each other. The relationships among these concepts are depicted in Figure 2.8. Network research environments are normally classified into three environments, namely testbeds, emulators, and simulators. These environments are interrelated. For example, interfaces can be developed for simulators to communicate with testbeds and vice-versa. Testbeds in turn made use of emulators to provide link emulation.

The next chapter presents the design and architecture of the viNEX network emulator.

Chapter 3

viNEX Design and Architecture

Chapter 2 provided a background on network research environments. Three classes of network research environments were described, namely *testbeds*, *emulators* and *simulators*. ViNEX was identified as an example of a network emulator because network nodes in viNEX run the standard and real networking protocols without any modifications or simulations. Details appear in Section 2.4 and Figure 2.4.

In this chapter, we propose the design of the virtual network emulator viNEX, which is a pseudo-acronym for Virtual INtegrated Network Emulator on Xen. The objective of this chapter is to develop the viNEX architecture and its components.

This chapter is structured as follows. Section 3.1 provides the high-level architecture and presents the significant components of viNEX. At the highest level, viNEX consists of two major components, namely, the *control network environment* and the *experiment topology environment*. The control network environment and the experiment topology environment are described in Section 3.1.1 and Section 3.1.2 respectively. Section 3.2 gives the details of the viNEX development environment including the additional F/OSS software packages used. A summary to this chapter is presented in Section 3.3.

As described in Chapter 2, Xen offers two approaches to virtualisation, namely, PVM (Section 2.3.1.3) and HVM (Section 2.3.1.4). Our initial release of viNEX (version 1.0) used the HVM approach is documented in our paper (Mukwevho et al. [56]) which was published as part of the proceedings of the SIMUTOOLS 2009 Conference. This paper is available in A.The preliminary results of the bandwidth measurement experiment indicated a rather slow network performance, averaging

about 3 Kb/s (Kilobits per second) between the network links. This performance is slow relative to the default 10 MB/s (Megabytes per second) minimum speed of most modern network links.

The above performance was attributed mainly to the slow performance of the Xen networking drivers under HVM guests. This eventually triggered a need to improve on networking performance in the context of our experiment. Due to the above slow performance of network links using PVM, a decision was made to switch from HVM to PVM, leading to the viNEX 2.0 release. In the context of this dissertation, we shall use the term viNEX to refer to the viNEX 2.0 release. A significant improvement on networking performance was noted as a result. Using the PVM approach, a maximum bandwidth of 12 MB/s (Megabytes per second) on a single network link between two PVM guest domains was measured. The performance improvement is attributed to the efficiency of the PVM guest's network drivers which have been found to perform better than their HVM counterparts (Menon et al. [53]).

The viNEX emulator runs the TCP/IP protocol stacks for both the control network and the experiment topology network. The standard network layers of the the TCP/IP protocol suite are described in Appendix C.

Recall that in Section 2.3, we mentioned that in the context of this dissertation the term `DomU` will be used to refer to a Xen *guest* domain. The term `Dom0` will be used to refer to the main Xen domain (also known as `Domain0`) which hosts the *hypervisor*.

In this research, we reuse the core networking functionality of the bridged network model in Xen. We rely on Xen to pass all the traffic from a guest domain, via a bridge connected to the backend interface of the guest domain. We then make use of the Linux Ebtables (Russell [72]) module to intercept network traffic at layer-2 and avoid network routing at layer-3 to optimize performance. Performance is enhanced by avoiding the execution of layer-3 routing code which in turn saves the need for CPU time by the virtual machine. The Linux Ebtables module adds bridging functionality at the Ethernet layer.

ViNEX was originally designed with the following principles in mind:

1. *Performance:* We aim to achieve acceptable performance by minimising network overheads as a result of performing inter-domain routing at the OSI

layer-2. Network packets that are passed between two directly connected nodes (DomUs) are not passed to layer-3 of the Linux `Dom0` host for IP routing, but are passed directly to layer-2 using the Ebtables module.

2. *Protocol Independence:* The testbed should be *generic* and be able to carry traffic independent of the protocol type. This is enabled by the routing at layer-2 as described above.

3. *Simple Configuration:* A simple command-line interface for creating complex network topologies should be used. We create a set of shell scripts to cater for such an interface. Details are discussed in Section 3.1.1.2.

3.1 viNEX Architecture

The viNEX high-level architecture is comprised of three major components, namely, the *Xen Hypervisor (VMM)*, *Control Network Environment (Dom0)* and the *Experiment Topology Environment*. These components are depicted in Figure 3.1.

The VMM provides virtualisation for the entire viNEX environment. It is the standard VMM as described in the Xen architecture in Section 2.3 of Chapter 2.

The *Control Network Environment* is used to provide `Dom0` with a direct connection to all the experiment network nodes and it is primarily used during the initial stages of the experiment creation to initialise the `DomU` experiment nodes. IP forwarding through the control network is disabled by setting the system variable `/proc/sys/net/ipv4/ip_forward` to `0`. As a result, the `DomU` experiment nodes cannot communicate directly with each other through the control network. The control network is therefore reserved for controlling the `DomU` nodes from `Dom0`. The *experiment topology environment* provides a network through which the `DomU` nodes communicate directly with each other as defined by the topology of the experiment. Such network is used for sending all the experiment-related traffic.

Inside the Control Network Environment, we find two significant components, namely, the *configuration and management scripts*, and the *traffic shaping node* (FreeBSD Guest). From now onwards, we shall use the term *gateway* to refer to the *traffic shaping node*. In the context of computer networks, a *gateway* is a computer that controls or allows access to another network or computer (Silberschatz et al.

[75]). In the context of viNEX, we are use the notion of a *gateway* to refer to the traffic-shaping node because all traffic between the network nodes must pass through such gateway. This is to allow for traffic shaping functions to be applied on the packets through the Dummynet component. The functionality of Dummynet can be referenced in Section 2.2.3.

The *Experiment Topology Environment* consists of an experiment network topology which is formed by experiment nodes and network links. Due to the IP address range, the experiment topology network is not directly accessible from `Dom0` (refer to Figure 3.1). This implies that users connecting from `Dom0` cannot use any of the IP addresses of the experiment nodes to communicate directly from the control network.

Next we discuss the *control network* and the *experiment topology* environments of viNEX.

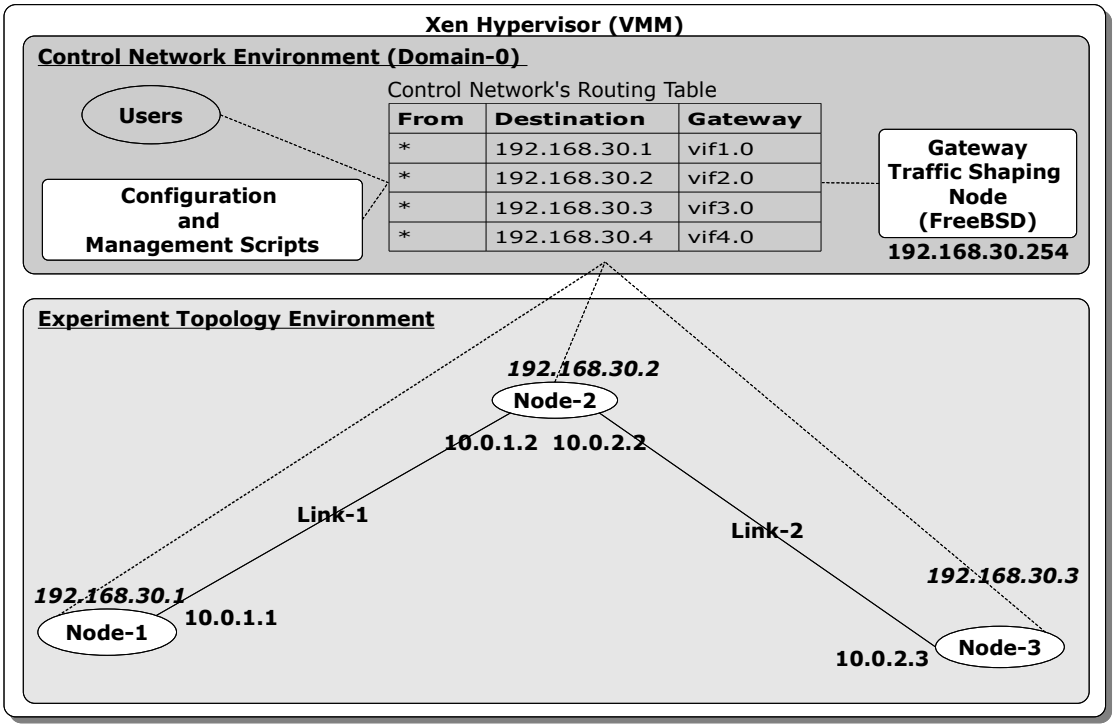


FIGURE 3.1: The high level architecture of viNEX

3.1.1 Control Network Environment

Our design approach follows that of Emulab (White et al. [85]). A *control network* is created to allow `Dom0` users direct access to the experiment *nodes*. The *control network* is critical for the initial configuration of experiment nodes. It is used by setup scripts to access the experiment nodes and configure their network settings. Configuration of network settings is achieved by executing direct commands on the experiment nodes, using the `ssh` remote connectivity tool from `Dom0`.

Connectivity of the control network is depicted by dotted-lines in the high-level architecture diagram in Figure 3.1. The layer-3 routing table in `Dom0` is used for passing traffic directly to the guest domains. Each topology node (`Node-X`) is assigned an IP address which resembles a pattern of `196.30.225.X`. It is important to note that any suitable IP address range can be used for this purpose. The first three octets `196.30.225` can be replaced by any valid values for IP addresses. Two IP addresses `196.30.225.254` and `196.30.225.252` are assigned to the Gateway node interfaces. `Dom0` is assigned the IP address `196.30.225.253`, configured to its `eth0` interface.

Next we discuss the two components of the control network, namely, the traffic shaping node as well as the configuration and management scripts.

3.1.1.1 Traffic Shaping Node

The traffic-shaping node is used to emulate network links between the topology nodes. It is also responsible for simulating network conditions such as bandwidth limitation, packet delay and random packet loss. The network conditions are provided by Dummynet running on the traffic-shaping node. All network traffic between any two topology nodes is channeled to pass through the traffic shaping node, as per Figure 3.1.

The viNEX traffic-shaping node is transparent from the experiment topology nodes, in other words, the experiment topology nodes cannot communicate directly with the traffic-shaping node. Traffic shaping is done by Dummynet (Rizzo [70]). The traffic-shaping node is a FreeBSD 7.0 PVM guest domain running on top of the Xen hypervisor. We interchangeably use the term *FreeBSD gateway node* to refer to the traffic-shaping node as well. A *single* instance of the FreeBSD

gateway node is created and every network link that belongs to the experiment topology is configured to pass through this node.

Next we describe the configuration and management component of viNEX.

3.1.1.2 Configuration and Management Shell Scripts

The configuration and management shell scripts provide an interface for controlling and managing viNEX. The interface is provided through a collection of four shell scripts, namely, `start-gateway.sh`, `start-node.sh`, `create-link.sh` and `modify-link.sh`. Appendix B gives the source code listings for these scripts. A brief description of the functionality provided by each shell script follows:

`start-gateway.sh` — This script is used for booting up the FreeBSD traffic shaping node. The source code of this script appears in Listing B.1.

`start-node.sh` — This script is a generic one which is used for starting any viNEX experiment topology node, see Listing B.2.

`create-link.sh` — This is for creating links between each pair of nodes as specified by the experiment. Link properties governing the behaviour of the link are passed through the arguments to this command as shown in Listing B.3.

`modify-link.sh` — This script is used to alter the properties of a links after it has been created. It allows for modification of the link behaviour without having to restart the entire experiment. The source code for this script is shown in Listing B.4.

3.1.2 Experiment Topology Environment

The Experiment Topology Environment forms the actual network topology used to emulate a network. It consists of the topology nodes and network links as discussed next.

3.1.2.1 Topology Nodes

Topology nodes are the network nodes that make up the experiment under investigation. They are standard Xen PVM nodes. For the purpose of this research, we have chosen NetBSD [57] as the operating system to be used for topology nodes. NetBSD [57] was chosen for its ability to run at a very low RAM footprint, thereby allowing for the creation of a network experiment with a large number of topology nodes on viNEX. As part of this research, we reduced the size of NetBSD kernel to about 4 MB compared to the standard GENERIC kernel of 10 MB in size. The NetBSD kerned was reduced by commenting out various drivers in its kernel compile configuration file. This prevents the drivers from being compiled or loaded into memory which results into a more scaled down kernel. Each guest domain has been configured to run at a maximum RAM of 32 MB to allow for scalability.

The other reason NetBSD was chosen is because of its simplicity in configuring various TCP/IP variants. NetBSD supports TCP/Reno, TCP/NewReno and TCP/Sack. An overview of these protocols is given in Fall and Floyd [25]. Different TCP/IP flavors may simply be enabled through the use of special kernel settings using the `sysctl` command. We need the support for multiple TCP/IP flavours in order to repeat the Van Jacobson experiment (Van Jacobson [81]) on viNEX for validating our work. A brief description of the Van Jacobson experiment was presented in Section 2.5 and the details of how this experiment was conducted on viNEX are presented in Chapter 5.

It should be noted that the experiment topology nodes do not access the traffic shaper directly. The traffic shaper is configured as a *transparent* node between any pair of experiment topology nodes. The main purpose of the traffic shaper node is to regulate and shape traffic as specified by the experiment definition in Section 3.1.1.1 above. Link parameters such as bandwidth, delays and random packet losses are passed through the command line tool for creating links. This was made possible through the use of Linux bridging together with Ebtables (Russell [72]).

Using Ebtables, each ethernet frame is intercepted and channelled to go via the traffic shaper node at layer-2. Since routing occurs at layer-2, no IP address is used for this purpose. A rule is inserted into the BROUTING (Bridge Routing) table of Ebtables. The rule uses the MAC addresses of the frames to be forwarded

to the traffic shaper. A sample of Ebtables rules created for the two-node topology in Figure 4.1 appears in Listing 4.3.

3.1.2.2 Network Links

Network links are used to define the communication between pairs of experiment topology nodes. All topology links are defined inside the traffic shaper node, which exists in the control network as shown in Figure 3.1. The traffic shaper node uses a combination of software bridging together with VLANs (Virtual Local Area Networks) in order to model the link between two nodes. Dummynet, together with `ipfw` (as described in Section 2.2.3) are then used to model the behavior of the traffic between the created VLANs.

The notion of network links is of fundamental importance to this research work, and in Chapter 4 we describe network links and connectivity in viNEX in further detail.

3.2 Software environment

The construction of viNEX was conducted on a stand-alone Linux server. The hardware and software configurations used to build viNEX are presented in Table 3.1. Table 3.2 presents the additional software components and utilities used as part of viNEX. ViNEX was built using free and open source software (F/OSS). All the software technologies which are listed in Table 3.1 are based on F/OSS.

Table 3.1: viNEX development environment

Operating System	CentOS 5.1 64-bit
Memory	1 GB
CPU	Intel Core 2 Duo, 3.0 GHz CPU with vT-x Support
Other Software	Xen 3.2, NetBSD 4.0, FreeBSD 7.0

We also made use of the following F/OSS software utilities to develop viNEX as well as to document this dissertation.

Table 3.2: Additional software packages used

Name	Function
ipfw	Ipfw (Antsilevich et al. [4]) is a program used to control the IP firewall and Dummynet on FreeBSD.
dummynet	The Dummynet (Rizzo [69]) package was used for link emulation in viNEX.
ebtables	Ebtables (Russell [72]) is an OSI layer-2 filter module used to route bridged packets moving between guest domains in viNEX.
LaTeX	A type setting language used to write this dissertation.
iperf	Iperf (Gates and Warshavsky [28]) is a UDP and IP traffic generator. We used iperf to measure bandwidth performance.
tcptrace	The tcptrace (Ostermann [61]) tool was used for analysing tcpdump files.
tcpdump	We used tcpdump (Van Jacobson and McCanne [82]) to capture the network packets during experiment runtime.
NS-2	NS-2 (McCanne et al. [52]) was used to repeat the Van Jacobson [81] experiment.
pdfcrop.pl	Pdfcrop (Oberdiek [58]) was used to crop the white spaces of the PDF documents to create some of the figures in this dissertation.
bridgeutils	Bridgeutils (Buytenhek [8]) is a software module which implements network bridging functions on Linux.
sysctl	Sysctl is a configuration utility which is used by most Unix-oriented operating systems (such as Linux, FreeBSD and NetBSD) to configure their kernel settings.

3.3 Summary

In this chapter, we introduced the high-level architecture of the viNEX emulator. The high-level architecture of viNEX is depicted in Figure 3.1. viNEX is comprised

of three logical components, namely, the Xen Hypervisor, the Control Network Environment and the Experiment Topology Environment.

The Control Network Environment includes the Configuration and Management Scripts as well as the Traffic Shaping Node components. The configuration and management scripts provide a command-line interface to manage and control network experiments. The function of the traffic shaping node is to emulate links and to simulate network properties such as packet delays, random losses and bandwidth limitations.

The Experiment Topology Environment is composed of experiment nodes and emulated network links which interconnect the nodes. Each network link is configured to pass traffic through the transparent traffic-shaping node.

In Chapter 4 which follows, we discuss network links further using a sample network topology.

Chapter 4

Network Links and Connectivity

The purpose of this chapter is to extend our discussion on viNEX by considering the components involved in the abstraction of a network link between a pair of network topology nodes. All network nodes are DomU virtual nodes, running NetBSD on top of Xen.

This chapter is structured as follows. Section 4.1 introduces the mechanism for creating network links by presenting a simple two-node topology on viNEX. This section also examines the low-level network components forming a network link in viNEX. The viNEX emulator can be used to create the common network topology structures as described by Tanenbaum [80]. These are the fully connected, partially connected, hierarchical, star and ring network topologies. The common topology structures and the scripts used to create them on viNEX are presented in Appendix D.

Section 4.2 provides a sequence of steps which take place during the propagation of network frames between two communicating nodes in the experiment topology. Section 4.3 describes how the routing information is propagated across all the experiment network nodes. We initially chose to use the RIP (Malkin [50]) protocol for route propagation owing to its simplicity and availability. Both RIP version 1 and 2 are supplied with NetBSD by default. We have since migrated to Open Shortest Path First (OSPF) as the default routing protocol after experiencing the 15-hop limit on RIP. It may also be possible to configure other routing protocols such as BGP or EGP on viNEX, but this would be a topic of further research. A summary of this chapter is presented in Section 4.4.

4.1 Network Links

In this section, we introduce the mechanisms implemented to model network link connectivity in the viNEX emulator environment. Our objective is to highlight the low-level network components involved in creating a network link on viNEX.

Each network link in viNEX is associated with a set of properties that governs how traffic is routed between the nodes. Properties of network links are concerned mainly with traffic shaping, which addresses issues of bandwidth, packet delays and loss rates. Such properties enable the virtual links to emulate their real world counterparts.

Figure 4.1 gives an example of a basic two-node network topology. The corresponding viNEX shell script used to create the topology is shown in Listing 4.1. As depicted in Figure 4.1, all network links in viNEX are defined to pass through the FreeBSD-based gateway, regardless of being shaped or not. This is to allow for full control of the network links by the testbed operators during the execution of an experiment.

Figure 4.1 shows a network topology with only two experiment nodes connected together by a network link with bandwidth limited to 2 Mbits, a packet delay of *10ms* and a random loss rate of *50%*. The corresponding low-level network components of the topology in Figure 4.1 are shown in Figure 4.2. We discuss each component of Figure 4.2 in the following sections.

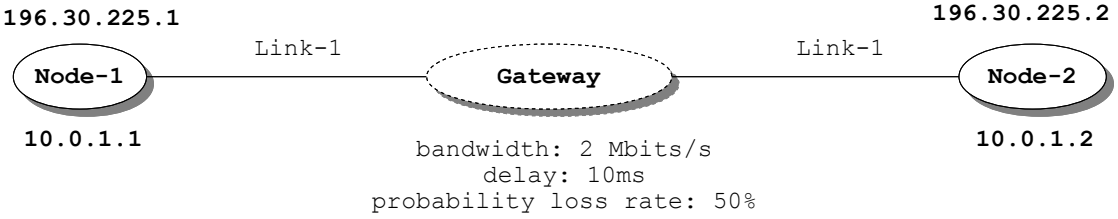


FIGURE 4.1: A **basic** two-node network topology

A viNEX network link between two domains consists of a number of network elements, namely, *frontend interfaces*, *backend interfaces*, *bridges*, *VLAN subinterfaces*, *Ebtables rules*, *Ipfw rules* and *Dummynet pipes*. Some of these components have already been discussed earlier in the introduction to Xen in Section 2.3. The

```
# Start the Gateway
/xen/freebsd/scripts/start-gateway.sh
# Start Node-1
/xen/netbsd/scripts/start-node.sh Node-1
# Start Node-2
/xen/netbsd/scripts/start-node.sh Node-2
# Create a link between Node-1 and Node-2
/xen/netbsd/scripts/create-link.sh --link-id 1 --from
   Node-1 --to Node-2 --bw 2Mbit/s --delay 10ms --plr 0.5
```

LISTING 4.1: viNEX configuration script for creating the topology in Figure 4.1

purpose of revisiting these components in this chapter is to put them into the context of the viNEX design.

In a two-node topology, *three* nodes are created, namely, the FreeBSD gateway node (called `Gateway`), `Node-1` (NetBSD), and `Node-2` (NetBSD). A sample Xen configuration for this scenario is given in Listing 4.2, which is displayed by issuing the `xm list` command after booting the nodes. Xen automatically assigns an integer-based node ID to each node. Using the node IDs, we are able to identify all the network (vif) backend interfaces allocated to each domain inside `Dom0`. In this example, the gateway node is assigned id 1, `Node-1` is assigned id 2 and `Node-3` was given id 3 (see the ID column in Listing 4.2).

```
[root@viNEX experiment]# xm list

Name      ID   Mem   VCPUs  State   Time(s)
Domain-0   0  1425     2    r-----   102.0
Gateway    1   512     1    -b----    31.7
Node-1     2    16     1    -b----    12.4
Node-2     3    16     1    -b----    13.1
```

LISTING 4.2: Xen configuration of Figure 4.1 topology

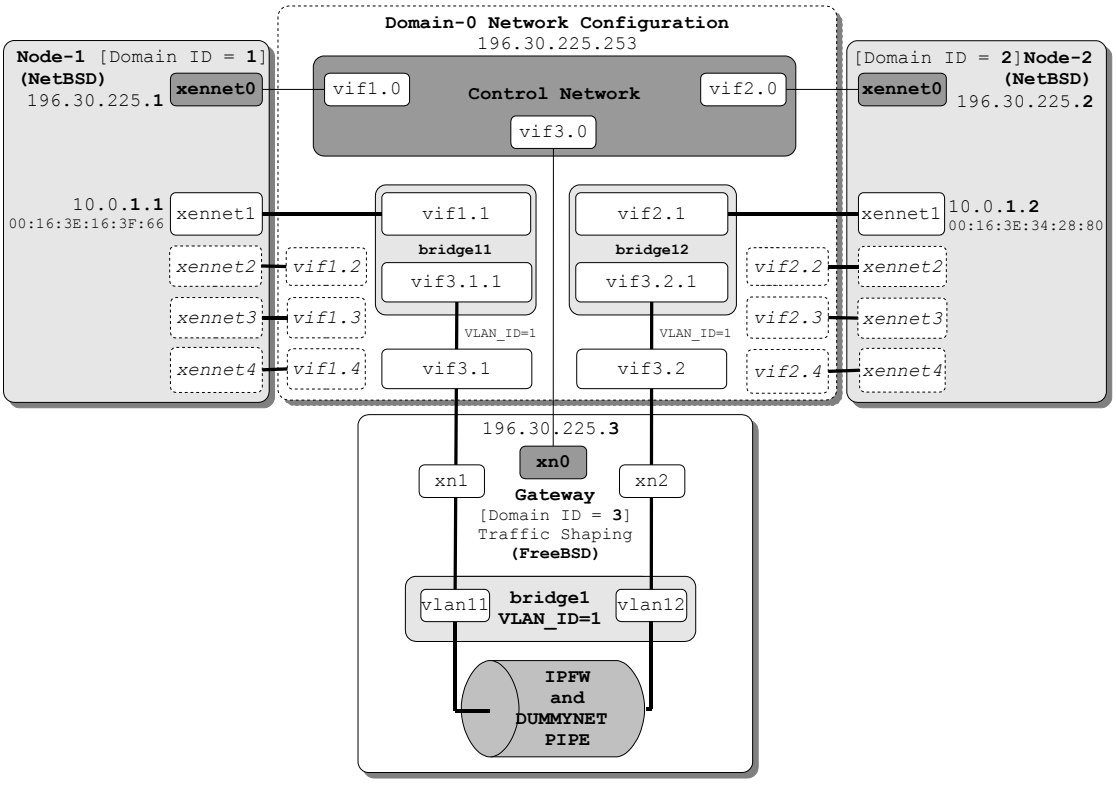


FIGURE 4.2: The viNEX components forming a network link between two topology nodes

In the following sections we discuss each of the above network elements of viNEX in further detail.

4.1.1 Frontend interfaces

Frontend interfaces are network interfaces that are located inside each domain. Their main responsibility is to provide network connectivity to the topology nodes with other nodes, for example, see the `xennet1` and `xennet2` interfaces in Figure 4.2. Since we are using PVM, all domains are running a specialised `netfront` driver built by the Xen team for PVM specific connectivity requirements. The `netfront` driver communicates directly with the `netback` driver running inside `Dom0`.

All interfaces involved are depicted in Figure 4.2. Each node is allocated *five* frontend interfaces and they are configured during the node startup process. Network interfaces are created by the Xen hypervisor during the `DomU` boot-up process,

hence we have to define a fixed set of five interfaces inside the Xen configuration scripts before the `DomU` nodes are started. Each of the frontend interfaces is used to connect to a unique network link, in other words, an experiment node can have a maximum of 5 network links.

During the development of viNEX, we defined a set of rules which are used to create and configure experiments. Seven rules were defined and we describe these below, starting with rules 1 and 2.

The following rule (Rule 1) is used for creating network interfaces belonging to each topology node in viNEX:

> **Rule 1:** A network node (`DomU`) in viNEX is always allocated a set of five frontend network interfaces called `xennet0`, `xennet1`, `xennet2`, `xennet3`, and `xennet4` (see Figure 4.2). The lowest interface, `xennet0` is always reserved to provide connectivity of a DomU to the control network.

The `xennet0` interface is automatically assigned an IP address belonging to the control network during startup (see `xennet0` in Figure 4.2). The IP address allocation rule for the control network is captured below:

> **Rule 2:** Let `Z` (a positive integer) be the domain identifier allocated to `Node-X` by Xen. The `xennet0` interface of `Node-X` is allocated an IP address to the value of `196.30.225.Z`.
>
> Looking at the sample topology in Figure 4.1, `Node-1` and `Node-2` are allocated IP addresses `196.30.225.1` and `196.30.225.2` respectively. Figure 4.2 captures these allocations.

Note that the above control IP address scheme `196.30.225.X/24` was picked to ensure that the control and experiment networks are completely isolated from each other. The `196.30.225` address prefix can be set to any valid IP address prefix as long as the chosen prefix does not intersect with the experiment network range which is set to `10.0.X.Y/24` by default.

The control network is required for enabling the link configuration scripts to do SSH login into the guest domain for network configuration. The remaining four interfaces (`xennet1`, `xennet2`, `xennet3` and `xennet4`) in Figure 4.2 are used to

create experiment topology links and they are allocated in any order as required by the topology creation process.

After the execution of the `create-link` command in line 8 of Listing 4.1, frontend interfaces are configured and brought up in both `Node-1` and `Node-2`. For this scenario interface `xennet1` is used for both `Node-1` and `Node-2`, because it happened to be available. Note that interfaces directly connected on opposite sides do not need to have matching names, since interfaces are picked by the configuration script based on their availability. The script checks for the next interface that is available. An interface is available if it has not been enabled. Every run of the topology creation process will yield the same results provided the commands of the topology creation script are executed in the same order as before.

The experiment network is using the IP address range of `10.0.X.Y/24` by default. This address is completely configurable and can be changed inside the `create-link.sh` script. The rule used for allocating IP addresses to link interfaces is defined by:

> **Rule 3:** Suppose `Node-X` was assigned an integer-based domain identifier `Z`. Each frontend interface located inside `Node-X` and directly connected to the topology network via network link `Link-Y` is then configured with IP address `10.0.Y.Z/24`. Network links in viNEX are named using a pattern `Link-Y` where variable `Y` is a positive integer which is allocated as an identifier to each link.
>
> An example of this rule is shown in Figure 4.2 where `xennet1` frontend interfaces of both `Node-1` and `Node-2` are allocated IP addresses `10.0.1.1` and `10.0.1.2` respectively. `Node-1` and `Node-2` are connected via `Link-1` as depicted in Figure 4.1.

In the event that a network researcher requires a different network configuration scheme, it can be achieved by manually using the SSH remote connectivity tool to directly connect the topology nodes. Once they connect with SSH, connectivity can be configured using the `ifconfig` and `route` network configuration utilities. Details of how to use the `ifconfig` and `route` utilities can be obtained in the NetBSD manual pages at [57].

4.1.2 Backend Interfaces

Backend interfaces are located inside Dom0 and they connect directly to their respective frontend interfaces located inside the topology domains. Considering Figure 4.2, the xennet1 frontend interfaces of Node-1 and Node-2 are connected to backend interfaces vif2.1 and vif3.1 respectively. The following rule governs the allocation of backend interfaces and their mapping onto frontend interfaces for each topology node (DomU).

> **Rule 4:** Let xennet0, xennet1, xennet2, xennet3 and xennet4 be the frontend interfaces of Node-X as defined by Rule 1 above, and let integer Z be the domain identifier of Node-X. The backend interfaces vifZ.0, vifZ.1, vifZ.2, vifZ.3, vifZ.4 and vifZ.4 are allocated by the Xen hypervisor inside Dom0 and paired with their corresponding frontend interfaces.
>
> The vif1.0, vif1.1, vif1.2, vif1.3 and vif1.4 interfaces are examples of backend interfaces allocated to Node-1 as depicted in Figure 4.2.

The following rule is used for connecting frontend and backend interfaces in viNEX:

> **Rule 5:** Suppose xennet0, xennet1, xennet2, xennet3 and xennet4 are the frontend interfaces of Node-X with domain identifier Z and vifZ.0, vifZ.1, vifZ.2, vifZ.3 and vifZ.4 be the backend interfaces allocated to Node-X. The frontend interfaces xennet0, xennet1, xennet2, xennet3 and xennet4 are paired with backend interfaces vifZ.0, vifZ.1, vifZ.2, vifZ.3 and vifZ.4 respectively.
>
> An example of this pairing is shown in Figure 4.2; the frontend interface xennet0 of Node-1 is paired with backend interface vif1.0, xennet1 is paired with vif1.1, xennet2 is paired with vif1.2, xennet3 is paired with vif1.3 and xennet4 is paired with vif1.2.

The frontend interfaces xn0 and xn1 of the gateway are connected directly to the backend interfaces vif1.0 and vif1.1 respectively, as shown in Figure 4.2.

4.1.3 Bridges

A total of three bridges are created for each link. These are the two Linux bridges inside Dom0 and one FreeBSD bridge inside the gateway node. Both Linux and

FreeBSD bridges are software implementations of the original hardware-based bridges. Further details on Linux bridge module can be obtained at [49]. Information on the FreeBSD bridge is available in FreeBSD [27].

The purpose of the two Linux bridges is to connect the traffic from the topology nodes directly to the gateway node for traffic shaping in a protocol independent manner. Packets are forwarded based on their Ethernet addresses (MAC) and not IP addresses. The backend interfaces of `Node-1` and `Node-2` are joined with the VLAN (Virtual Local Area Network) subinterfaces of the gateway node to allow for traffic routing at layer-2 of the OSI model.

The single FreeBSD bridge that is created inside the gateway node, is used to maintain traffic at layer-2. As a result, Dummynet is also configured to operate at layer-2.

The following rule is used for creating network bridges belonging to a network link in viNEX.

> **Rule 6:** Suppose `Link-X` is used to connect `Node-Y` and `Node-Z` in viNEX, and three software-based bridges are created as follows: The two Linux bridges inside `Dom0`, namely, `bridgeX0` and `bridgeX1` together with a single FreeBSD bridge inside the gateway node called `bridgeX`, where `X` is an integer, is used to identify a network link in viNEX.
>
> The first bridge (`bridgeX0`) is used to carry network traffic travelling between `Node-Y` and the Gateway. Similarly, `bridgeX1` is used to connect `Node-Z` with the `Gateway` node. The FreeBSD bridge (`bridgeX`) inside the gateway node is used to link traffic moving between `Node-Y` and `Node-X` in both directions.
>
> Figure 4.2 presents an example of this rule. Three bridges, namely, `bridge10`, `bridge11` (both inside `Dom0`) and `bridge1` (inside node `Gateway`) were created for `Link-1`.

4.1.4 VLAN Interfaces

The gateway node (see Figure 4.2) has two fixed interfaces (`vif1.0` and `vif1.1`) connecting it to `Dom0`. Since all links have to go through gateway node for traffic shaping, we had to derive a mechanism that will allow the sharing of these two

fixed interfaces for all the link connections. One such mechanism was to use VLAN tagging. For each `Link-X` in the topology, a corresponding VLAN with VLAN_ID = `X` is created to isolate the link's traffic. For the example in Figure 4.1, a VLAN with VLAN_ID = 1 is defined for `Link-1` (see Figure 4.2).

Two sets of VLAN subinterfaces are created as follows: A set of two VLAN subinterfaces (`vif3.1.1` and `vif3.2.1`) is created inside `Dom0` and another set of two VLAN interfaces (`vlan11` and `vlan12`) is created inside the gateway node. For each `Link-X`, a VLAN `X` is created inside `Dom0` on the gateway interface. The purpose of the VLAN is to keep traffic for `Link-X` separate in its own broadcast domain so that traffic-shaping rules specific to this link can be applied, once it enters the gateway node. For `Link-1`, VLAN interfaces `vif3.1.1` and `vif3.2.1` were created and enslaved to the corresponding link bridges using the `brctl` command of the `bridgeutils` package. Details about these packages appear in [49].

The purpose of the VLAN interfaces inside the gateway node (`vlan11` and `vlan12`) is, firstly to enable the filtering of traffic by `ipfw` for Dummynet at layer-2 without having to use IP addresses. Secondly, these VLAN interfaces are used to establish the direction of the traffic flow. Traffic flowing from `Node-1` towards `Node-2` (see Figure 4.2) enters the gateway through `vlan11`. Similarly, traffic flowing from `Node-2` to `Node-1` enters the gateway via `vlan12`. Dummynet modules and `ipfw` use the source and target interfaces to select the rule to be applied.

The following rule describes the creation of VLAN interfaces belonging to a viNEX network:

> **Rule 7:** Suppose we are given the gateway node with domain identifier `W`. Assume further that the software bridges have already been created as per `Rule 6` above. Let `Node-Y` and `Node-Z` be two arbitrary nodes to be connected by `Link-X` in a viNEX topology. A total of four VLAN subinterfaces are created as follows:
>
> The names of the VLAN subinterfaces are `vifW.1.X, vifW.2.X, vlanXY` and `vlanXZ`. The VLAN subinterfaces `vifW.1.X` and `vifW.2.X` are both located inside `Dom0` while `vlanXY` and `vlanXZ` are located inside the gateway node.
>
> VLAN subinterfaces `vifW.1.X` and `vifW.2.X` are enslaved to `bridgeX0` and `bridgeX1` respectively. Inside the gateway node, `vlanXY` and `vlanXZ` are enslaved to the fixed frontend interfaces `xn1` and `xn2` respectively.

4.1.5 Ebtables rules

Ebtables (Russell [72]) is a Linux packet filter that enables us to intercept bridged network traffic at layer-2 and can perform actions such as routing and MAC address translation. We make use of Ebtables to intercept any bridged traffic submitted to `Dom0` via a backend interface of a DomU node. The PREROUTING (Russell [72]) chain of the NAT table of Ebtables is used to perform MAC address translation of the destination using the `dnat` instruction. The destination MAC is changed to the MAC address of the directly-connected destination as defined by the topology of the experiment. Using the experiment in Figure 4.1, two Ebtables rules are created for each link inside the NAT PREROUTING table using the link configuration script. A fragment of the code is given in Listing 4.3.

```
1 [root@viNEX experiment]# ebtables -t nat -L
2 Bridge table: nat
3
4 Bridge chain: PREROUTING, entries: 2, policy: ACCEPT
5 -i vif2.1 -j dnat --to-dst 0:16:3e:34:28:80 --dnat-target
     ACCEPT
6 -i vif3.1 -j dnat --to-dst 0:16:3e:16:3f:66 --dnat-target
     ACCEPT
7 ...
```

Listing 4.3: Ebtables rules for the topology of Figure 4.1

The two rules created are shown in lines 5 and 6 of Listing 4.3. The rule in line 5 translates the MAC address of any frame that arrived through backend interface `vif2.1` to the MAC address of the frontend interface of `Node-2` (`00:16:3e:34:28:80`). This enables the frame to be passed to `Node-2` directly after being traffic shaped by the `Gateway`. Similarly, the rule in line 6 translates the MAC address of any frame arriving directly from `Node-2` through interface `vif3.1` to the MAC address of the frontend interface of `Node-1` (`00:16:3e:16:3f:66`). The MAC addresses are automatically configured by the NetBSD operating system during the boot-up phase. Ebtables extracts the MAC interfaces from the Ethernet frames which contain the IP network packets.

The above mechanism ensures that network traffic is routed at layer-2 without being passed to any higher OSI layer. Therefore, network traffic flow is maintained at layer-2 between any two directly-connected network nodes, implying that no IP routing is performed. IP routing is performed only inside the topology nodes, using

the routes that are discovered through the configured routing protocol (OSPF is configured by default). The VLAN and bridge devices that are located inside the FreeBSD gateway node also assist in maintaining network traffic at layer-2. The Dummynet and `ipfw` subsystems inside the gateway are also configured to operate on packets at layer-2. Network packet filtering is done by using the interface names and not the IP addresses.

The bridging mechanism described above ensures that all frames are channelled to pass through the transparent FreeBSD `Gateway` irrespective of any MAC address translation. The VLAN mechanism also ensures that the traffic remains within the link domain without interfering or broadcasting to incorrect destinations.

4.1.6 Ipfw Rules

`Ipfw` is a notational shorthand for IP firewall. It is a FreeBSD packet filtering mechanism and a traffic accounting module (FreeBSD [27]). Packet filtering is specified using a set of rules that are created by using the `ipfw` command line utility on FreeBSD. Listing 4.4 presents the `ipfw` rules for the topology in Figure 4.1. Dummynet pipes are also created using the `ipfw` command line.

```
1 Gateway# ipfw show
2 00800  27  1260 pipe 10 ip4 from any to any via vlan11
     layer2
3 00900  27  1260 pipe 11 ip4 from any to any via vlan12
     layer2
4 65535  19  4943 allow ip from any to any
```

Listing 4.4: IPFW rules for the topology of Figure 4.1

4.1.7 Dummynet pipes

Like `ipfw`, Dummynet (Rizzo [70]) is also a module used for network simulation. It simulates the presence of a real network by providing mechanisms for defining network link properties such as finite queues, bandwidth limitations, network delays and loss rates. A Dummynet pipe configuration for the topology in Figure 4.1 is shown in Listing 4.6. This Dummynet pipe was created after the execution of the command at line 8 of Listing 4.5.

```
1 # Start the Gateway
2 /xen/freebsd/scripts/start-gateway.sh
3 # Start Node-1
4 /xen/netbsd/scripts/start-node.sh Node-1
5 # Start Node-2
6 /xen/netbsd/scripts/start-node.sh Node-2
7 # Create a link between Node-1 and Node-2
8 /xen/netbsd/scripts/create-link.sh --link-id 1 --from
    Node-1 --to Node-2 --bw 2Mbit/s --delay 10ms --plr 0.5
```

Listing 4.5: viNEX configuration script for creating the topology in Figure 4.1

```
1 Gateway# ipfw pipe show
2 00010:   2.000 Mbit/s   50 ms   50 sl.plr 0.500000 1 queues
    (1 buckets) droptail
3
4       mask: proto: 0x00, flow_id: 0x00000000,  ::/0x0000
    -> ::/0x0000
5 BKT ___Prot___ _flow-id_ ___Source
    IPv6/port_____________Dest. IPv6/port_ Tot_pkt/bytes
    Pkt/Byte Drp
6   0       udp 1610612736    fe80::216:3eff:fe1f:f3ea/5353
    ff02::fb/5353      7     1963  0    0   1
7 00011:   2.000 Mbit/s   50 ms   50 sl.plr 0.500000 1 queues
    (1 buckets) droptail
8
9       mask: proto: 0x00, flow_id: 0x00000000,  ::/0x0000
    -> ::/0x0000
10 BKT ___Prot___ _flow-id_ ___Source
    IPv6/port_____________Dest. IPv6/port_ Tot_pkt/bytes
    Pkt/Byte Drp
11  0       udp 1610612736   fe80::216:3eff:fe3e:db80/5353
    ff02::fb/5353      7     1500  0    0   4
```

Listing 4.6: Dummynet pipe for Link 1 in Figure 4.1

Line 2 of Listing 4.6 shows the link properties that were passed to the link creation script (line 8 of Listing 4.5). Starting from the left, 00010 is the unique id for the pipe, 2.000 Mbit/s is the link bandwidth, 50 sl is the queue size and 0.50000 is the packet loss rate. The link has 1 queue which contains 1 bucket for storing network packets. The queue uses the tail drop queuing technique as described in (Comer [12]). The tail drop technique simply drops any arriving packet in the event of the queue being full.

All network links on viNEX are full duplex links resulting in two Dummynet pipes being created by the link configuration script. The pipes are used to carry traffic

moving in two directions across the link. Considering the output of the command `ipfw pipe show` at line 1 of Listing 4.6, we note that two Dummynet pipes, namely, `00010` and `00011` were created for the link (`Link-1`) between `Node-1` and `Node-2`. Pipe `00010` is responsible for carrying traffic from the source `Node-1` to the target node `Node-2`. Pipe `00011` is used for carrying traffic moving in the opposite direction from `Node-2` to `Node-1`.

Each Dummynet pipe is associated with its own queue. The queue is used for carrying any network packets that are in transit. The default queue size is 50 slots but this may be specified manually during link configuration, using the `qsize` parameter. The above queues were configured with the default size of 50 slots. Each queue slot is about 1500 bytes, which is large enough to fit a standard Ethernet frame.

The following section considers the details of the path followed by network packets during their transmission between the source and the target nodes in a viNEX network experiment topology.

4.2 Network traffic flow

The previous sections focused mainly on the description of network elements used for modelling network links. In this section, we provide an overview of the steps involved in propagating network traffic between two nodes in an experiment topology. We also identify and describe the role played by each network element in the propagation of network traffic within viNEX. All network links are configured to pass through the gateway node irrespective of their traffic being shaped or not. A shaped link refers to a network link with some bandwidth restriction (such as delay or packet loss rate) applied to the traffic flowing through the link.

A total of sixteen steps are performed in viNEX to transmit traffic across a network link. To discuss each step, we shall reuse the same two-node topology as depicted in Figure 4.1. For ease of reference we repeat this configuration as Figure 4.3 in this section.

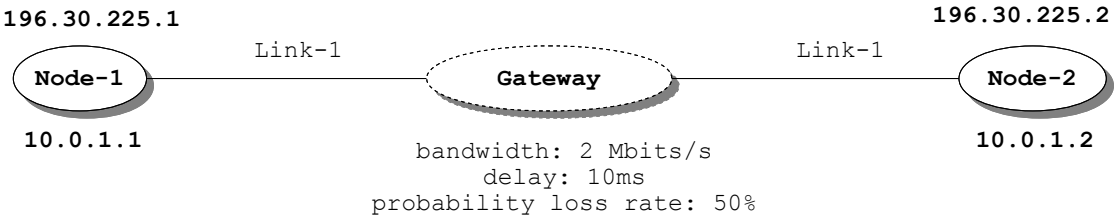


FIGURE 4.3: A two-node network topology with a shaped network link (repeated from Figure 4.1)

The steps alluded to above are labelled 1 to 16 in Figure 4.4. Traffic is generated from `Node-1` flowing across the traffic-shaping gateway node to the destination node, `Node-2`. Both nodes are using the TCP/IP protocol for communication and therefore, `Node-2` generates a TCP acknowledgment (ACK) for every segment received.

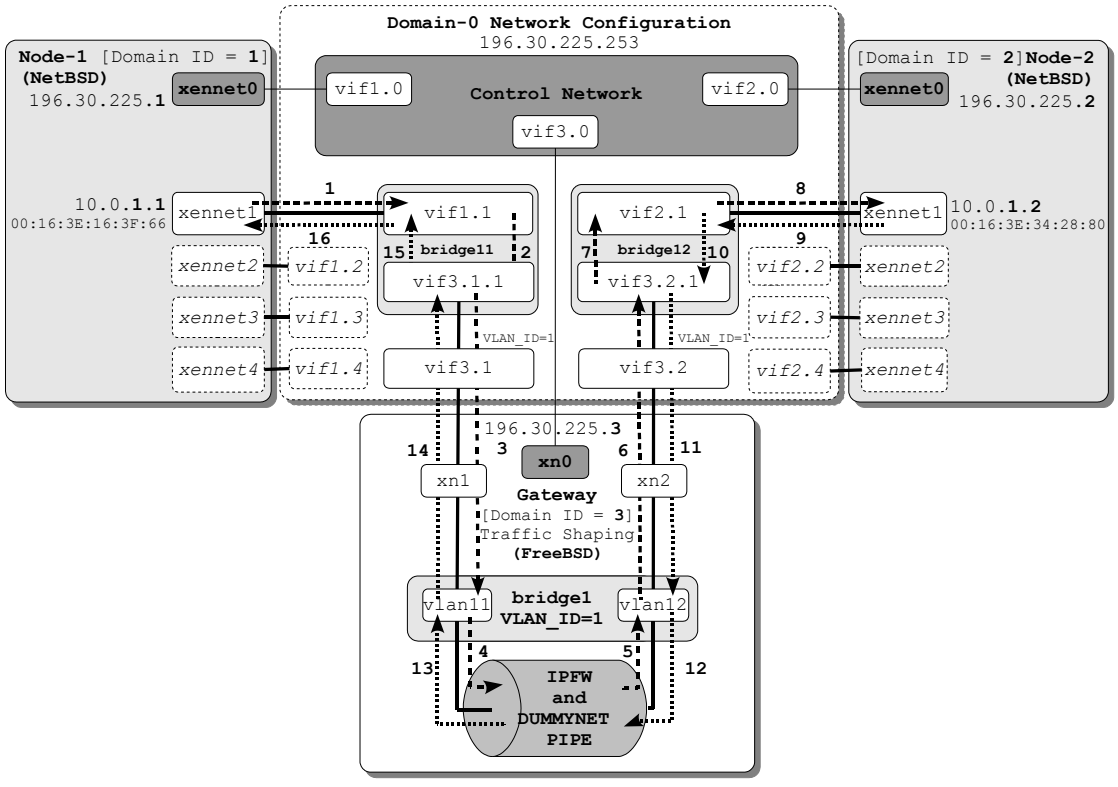


FIGURE 4.4: viNEX network traffic flow across the shaped `Link-1` between `Node-1` and `Node-2` of Figure 4.3

Next we describe each of steps 1 to 16 depicted in Figure 4.4.

1. `Node-1` initiates a transmission by injecting a sequence of TCP/IP packets that form a segment into its frontend interface, `xennet1`. Network frames are transmitted directly into `Dom0` through the backend interface `vif1.1`. Since the `vif1.1` interface is a member of `bridge11`, the Ethernet frames are passed into the bridge first instead of `Dom0`. Recall that the transmission of network packets occurs at layer-2 until they reach the next hop where IP routing is performed.

2. At `bridge11`, the Ebtables PREROUTING chain is invoked resulting in the execution of the rule at line 5 of Listing 4.3. The rule changes the MAC address of the frame to the value `00:16:3E:34:28:80`. This is the MAC address of the `xennet1` frontend interface located inside `Node-2` as shown in Figure 4.4. The bridge forwarding operation is then performed on the frame. As a result, the network frame is forwarded to the VLAN subinterface `vif3.1.1` which is linked directly to the backend interface `vif3.1`. At this stage, VLAN tagging takes place.

   The frame is tagged with VLAN 1 and passed to the backend interface `vif3.1`. The packet travels in a VLAN tagged mode until it reaches `vif3.2.1` when VLAN tagging is removed. VLANs play a significant role in viNEX, since they prevent packets belonging to different network links to interact. Once a packet is VLAN tagged, it will remain within that VLAN and will not be routed through a different link. The backend interface `vif3.1` is in turn connected to the gateway.

3. Interface `vif3.1` passes the network frame to the gateway via the frontend interface `xn1` for traffic-shaping functions to be performed.

4. Interface `xn1` passes the frame to `bridge1` through the VLAN interface `vlan11`. The IPFW rule at line 2 of Listing 4.4 is applied to the network frames at layer-2 of interface `xn1`. The rule stipulates that the frame must be passed to Dummynet `pipe 10`. Dummynet is then invoked and the network properties defined on pipe 10 are applied to the frame before it is forwarded. The Dummynet properties of `pipe 10` are shown at lines 2 to 6 of Listing 4.6. The properties of 2 Mbit/s bandwidth, 10 milliseconds delay and a loss rate of 0.5 are applied to the frame by Dummynet.

5. After the above network properties have been simulated on the frame, it is forwarded by Dummynet through the frontend interface `xn2` via the `vlan12` interface belonging to `bridge1`.

6. Interface `xn2` passes the frame to its backend interface `vif3.2`, which in turn sends the frame to the bridge through its `vif3.2.1` interface.

7. The frame is forwarded by `bridge12` to the backend interface `vif2.1`.

8. Interface `vif2.1` is connected to the frontend interface `xennet1` inside the destination, `Node-1`. The frame is passed to `Node-2` for processing. Since the destination MAC address of the frame was set to the value of the interface `xennet1`'s MAC address, interface `xennet1` performs a check on the destination MAC of the frame and determines that it matches its address. Therefore, the frame is destined for the `Node-2` host. The frame is then passed to the higher network layers where TCP/IP handles the packet and generates an acknowledgment. If the IP address of the packet did not match IP address of `Node-2`, the packet will be forwarded to the next host according to the routing table of `Node-2`.

9. The above acknowledgment packet is mapped onto an Ethernet frame and injected into the network by `Node-2` through the frontend interface `xennet1`. The frame is passed to the backend interface `vif2.1`, a member of `bridge12`.

10. Ebtables is again invoked to translate the MAC address of the returning frame to the MAC address of interface `xennet1` located inside `Node-1`. The MAC address of `xennet1` of `Node-1` is `00:16:3E:16:3F:66`, as per Figure 4.4. Since the input interface is `vif2.1`, the Ebtables rule at line 5 of Listing 4.3 is matched and executed to perform the MAC address translation.

    The translated frame is then passed through the VLAN subinterface `vif3.2.1`. At this stage, the VLAN tagging process occurs again. The frame is tagged with VLAN 1 and passed to the backend interface `vif3.2`. The packet travels in a VLAN tagged mode until it reaches `vif3.1.1` where the VLAN tagging is removed.

11. The above translated Ethernet frame is passed to the gateway for traffic shaping by `vif3.2` through the frontend interface `xn2`.

12. All frames received by the gateway node are first passed to the `ipfw` firewall. Since the frame was received via interface `xn2`, the frame is passed to `bridge1`

via the VLAN interface `vlan12`. The `ipfw` rule at line 3 of Listing 4.4 is matched. This rule makes reference to the Dummynet `pipe 11`. The bandwidth limitation of 2 Mbit/s, the delay of 10 milliseconds and a loss rate of 0.5 are applied to the frame.

13. Dummynet passes the traffic-shaped frame to the VLAN interface `vlan11` of `bridge1`.

14. Interface `vlan11` forwards the frame to the frontend interface `xn1` which in turn sends the frame to the backend interface `vif3.1` inside `Dom0`. The frame is then passed to `bridge11` by `vif3.1` through its VLAN subinterface `vif3.1.1`.

15. VLAN tagging is removed and the frame is forwarded to the backend interface `vif1.1` by `bridge11`.

16. The acknowledgment Ethernet frame finally reaches `Node-1` through `xennet1`. This step concludes the sequence of steps involved in the transmission between two topology nodes through a shaped network link.

In Section 4.1.5, we mentioned that the delay node is transparent from both the sending and receiving nodes. In other words, the sending and receiving nodes are not aware that their traffic is passing through an intermediate Gateway. `Node-1` and `Node-2` in Figure 4.3 communicate as if they were directly connected.

To demonstrate the above, we issued a `traceroute` command from `Node-1` to `Node-2` and we obtained the results shown in Listing 4.7. Listing 4.7 shows all the network hops encountered along the network route from `Node-1` to `Node-2`. Note that the IP address of the gateway is not included in the output of the `traceroute` command.

```
1 Node-1# traceroute 10.0.1.2
2 traceroute to 10.0.1.2 (10.0.1.2), 64 hops max, 40 byte
    packets
3  1  10.0.1.2 (10.0.1.2)  30.466 ms
```

LISTING 4.7: Output of `traceroute` command inside `Node-1` of Figure 4.3

Network links in viNEX provide us with the capability of simulating the behaviour of real networks. Simulation is provided by the Dummynet and `ipfw` modules of FreeBSD loaded inside the gateway node. The following section considers how network topology routes are configured and maintained throughout the experiment.

4.3 Routing and Route Propagation

Routing table maintenance and route propagation is done automatically through some of the standard Internet routing protocols. We have successfully deployed and ran standard routing protocols such as Routing Information Protocol (RIP) (Malkin [50]) and OSPF (Moy [55]) on viNEX.

RIP was enabled and configured on all experiment topology nodes during the node boot-up process. In NetBSD, RIP is implemented by the `routed` daemon process. The `routed` daemon comes with the standard NetBSD OS by default. It is enabled by using the `routed=YES` setting inside the `/etc/rc.conf` file. When a new link is configured during a topology creation step, RIP automatically broadcasts the IP addresses of the directly-connected neighbours using the UDP protocol. It is also possible to manually configure the routes by executing commands at each node.

RIP was used during the initial development phases of viNEX until we experienced some of its known limitations. The number of hops in RIP are limited to 15 as described in RFC2453 [50]. Networks beyond the 15 hops limit are considered unreachable. Clearly this creates a problem when creating a topology with more than 15 hops in viNEX.

Recall that in viNEX each host is assigned an IP address of the form `10.0.X.Y/24` where `X` is an integer identifying the link and `Y` is a unique domain identifier that was generated by Xen for the host. In viNEX, we make use of the `255.255.255.0` network mask. This implies that each link is considered to be a unique subnetwork identified by the first three octets (24 bits) of the IP address. Consequently, RIP will count each network link as a separate network. As stated above we can reach only the first 15 hops in the network. Any link beyond 15 is considered unreachable. This limitation resulted in a decision to switch to the more scalable OSPF (Moy [55]) routing protocol.

In OSPF, the number of hops between nodes is unlimited. OSPF is based on the shortest path first (SPF) algorithm (Weiss [83]). The OSPF protocol splits the networks into hierarchical areas. These areas are joined together to form one large autonomous system. In viNEX, we have created one area, namely, Area 0 which is known as the backbone area.

NetBSD OS currently does not come with a standard implementation of the OSPF protocol. Consequently, we were challenged to find a light-weight and robust daemon process implementing OSPF on NetBSD. We identified the `ospfd` daemon from GNU Zebra (Ishiguro [42]) as a possible solution. The `ospfd` daemon comes as part of a collection of routing daemons developed by GNU Zebra. GNU Zebra contains a set of daemons for implementing routing protocols such as RIP, BGP and OSPF. The GNU Zebra project is part of the GNU Projects and it is distributed as free software under the GNU General Public License.

4.4 Summary

In this chapter, we presented the low level composition of network links in viNEX. Network links are composed mainly in an aggregation of four main types of network elements. These are frontend interfaces, backend interfaces, bridges and VLAN devices. Network links were described in Section 4.1.

A viNEX network link is formed by a collection of ***thirteen*** devices as depicted in Figure 4.2:

- ***Four*** frontend interfaces, namely, the two `xennet1` interfaces inside `Node-1` and `Node-2` plus two interfaces `xn1` and `xn2` inside the gateway node.
- ***Four*** backend interfaces. These are `vif1.1`, `vif2.1`, `vif3.1` and `vif3.2`.
- ***Three*** software bridges, which are `bridge11` and `bridge12` both located inside `Dom0`, and `bridge1` located inside the gateway node.
- ***Two*** VLAN interfaces (`vlan11` and `vlan12`) which are members of `bridge1` inside the gateway node.

Both `ipfw` and Dummynet are used to simulate network link conditions inside the FreeBSD `Gateway` node. All network links are configured to pass through the transparent gateway node irrespective of being traffic shaped.

In Section 4.2, we presented the steps involved in a flow of network traffic between a pair of topology nodes. A total of sixteen steps are involved in the transmission of network frames between two directly-connected topology nodes.

Finally, we looked at how routes are propagated across a network. Two options exist for viNEX. Routes can be configured manually using standard networking commands or by configuring a network routing protocol such as RIP or OSPF. Initially, RIP was used as the default routing protocol on viNEX, but we have since migrated to the use of OSPF as the default. This was aimed at overcoming some of the limitations experienced with RIP. OSPF is only enabled on the NetBSD nodes that form the experiment topology network. Routing protocols are not installed on the `Gateway` and `Dom0` nodes, since they do not directly form part of the experiment topology.

This chapter concludes our technical description of viNEX. The next chapter focuses on the experiments performed on viNEX as well as a discussion of the results.

Chapter 5

Evaluation of viNEX

In the previous chapter we discussed the low level components which are used in constructing a viNEX network link. We also extended our discussion in Chapter 4 to observe the steps involved in the propagation of network traffic across a network link.

In this chapter, we consider a number of experiments conducted on viNEX and provide an analysis of the results for each experiment. The objective is to evaluate viNEX. The experiments were aimed at addressing three key issues, namely, verifying the functionality of, analysing packet throughput and determining the scalability of viNEX.

The rest of this chapter is structured as follows. Section 5.1 provides an assessment of the functionality of the viNEX emulator, while Section 5.2 focuses on measuring the amount of memory used to model a network link in viNEX. An enhancement to `iperf` to report one-way-delays is discussed in Section 5.3. The details of the `iperf` enhancement appear in Appendix G. The measurement of a one-way-delay time is later used in the calculation of the *bandwidth delay* product in Section 5.4. We also conduct an investigation to assess the scalability and limitations of viNEX in Section 5.4. Finally, we conclude this chapter with a summary in Section 5.5.

5.1 Verifying viNEX

The aim of this section is to conduct an assessment to determine if the operation of viNEX is comparable to that of a real network. We conduct our comparison

by repeating the Van Jacobson experiment on viNEX as described in the original paper (Van Jacobson [81]), which is a study of congestion avoidance and control. A summary of the Van Jacobson experiment is presented in Section 2.5 of this dissertation. In the context of this research, the term "Van Jacobson experiment" will be used to refer to the experiment conducted in the study of congestion control and avoidance by Van Jacobson. We shall reuse the same experiment topology as depicted in Figure 7 of the Van Jacobson paper [81].

The Van Jacobson experiment was conducted using both the original 4.3BSD TCP stack and the newer TCP stack enhanced with congestion control and avoidance algorithms. Due to the lack of a readily available 4.3BSD stack for the NetBSD operating system, we make use of the TCP/Reno variant of the TCP protocol (Fall and Floyd [25]). Our results are compared with those obtained by Van Jacobson using the new TCP with congestion control and avoidance.

In summary, the Van Jacobson experiment was aimed at deriving a set of algorithms to handle TCP/IP congestion avoidance and control. It was critical for the investigation to be done following the worldwide series of Internet congestion collapses which started in October 1986.

The Van Jacobson paper [81] describes all five algorithms which were developed after the congestion collapses were experienced. The algorithms include round-trip-time (RTT) variance estimation, exponential retransmit timer back-off, slow-start, a more aggressive receiver acknowledgment policy and dynamic window sizing during congestion.

We also performed the Van Jacobson experiment using the NS-2 simulator. This provided us with an additional mechanism to verify the viNEX results against. The NS-2 simulator provided a platform to validate the effects of changing the experiment parameters (such as bandwidth) on viNEX. The details of the NS-2 experiment and results appear in Appendix E of this dissertation.

It should be noted that our viNEX emulator is in a prototype phase, and the aim of conducting this comparisons is to assess the operational state of viNEX.

5.1.1 Experiment setup and procedure

The viNEX network topology used for this experiment is depicted in Figure 5.1. This topology resembles the topology used in the original Van Jacobson experiment [81].

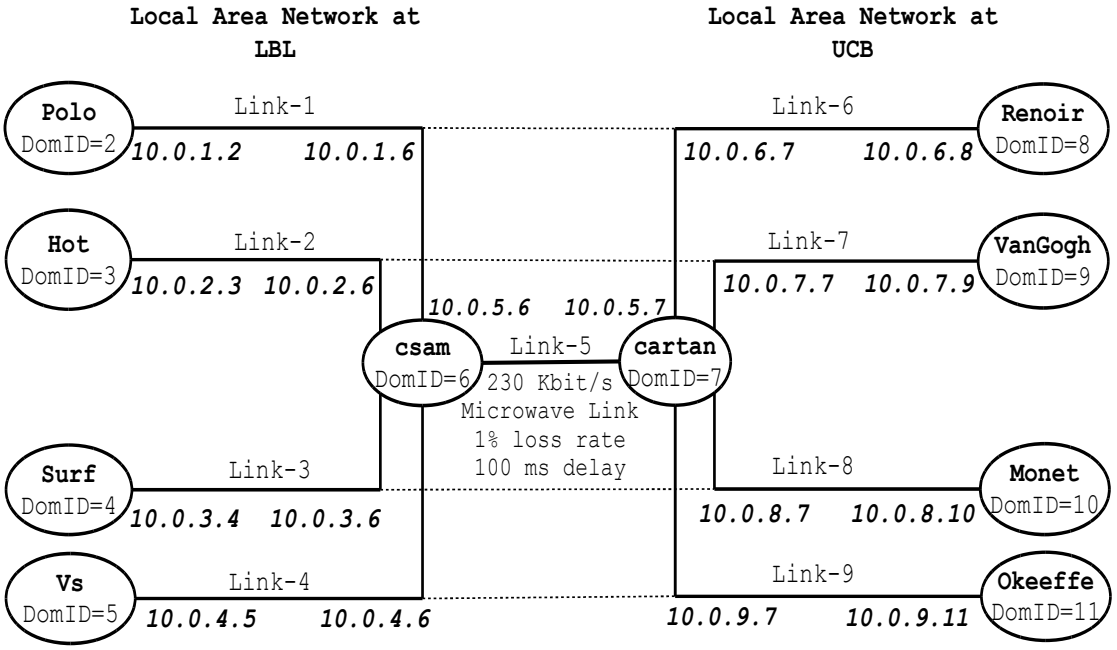


FIGURE 5.1: The Van Jacobson Experiment topology used for the viNEX repeat experiment (adopted from Van Jacobson [81])

The Van Jacobson experiment was conducted on the two LANs between the LBL (Lawrence Berkeley National Laboratory) and the UCB (University of California, Berkeley) campuses (see Appendix E). In viNEX, there is no notion of LANs, every network link is a *point-to-point* link. For example, looking at Figure 5.1, if node `Polo` needs to communicate with node `Vs` which is located on the same LAN at LBL, the communication traffic will always be routed via the `csam` node. In a LAN environment, it would have been possible for the two nodes to communicate directly without having to send their packets to an intermediate node (`csam`). This shortcoming can be addressed by creating a direct network link between `Polo` and `Vs`. A LAN topology can be modeled by creating links between all nodes to enable direct communication between them.

The experiment topology in Figure 5.1 was created using the shell script in Listing 5.1.

```
1  # Start Gateway node
2  /vinex/scripts/start-gateway.sh
3
4  # Create nodes at LBL
5  /vinex/scripts/start-node.sh Polo
6  /vinex/scripts/start-node.sh Hot
7  /vinex/scripts/start-node.sh Surf
8  /vinex/scripts/start-node.sh Vs
9  /vinex/scripts/start-node.sh csam
10
11 # Create nodes at UCB
12 /vinex/scripts/start-node.sh cartan
13 /vinex/scripts/start-node.sh Renoir
14 /vinex/scripts/start-node.sh VanGogh
15 /vinex/scripts/start-node.sh Monet
16 /vinex/scripts/start-node.sh Okeeffe
17
18 # Create node links at LBL
19 /vinex/scripts/create-link.sh --link-id 1 --from Polo   --to csam
20 /vinex/scripts/create-link.sh --link-id 2 --from Hot    --to csam
21 /vinex/scripts/create-link.sh --link-id 3 --from Surf   --to csam
22 /vinex/scripts/create-link.sh --link-id 4 --from Vs     --to csam
23
24 # Create the Microwave link
25 /vinex/scripts/create-link.sh --link-id 5 --from csam --to cartan --bw 230Kbit/s
       --plr 0.01 --delay 100
26
27 # Create node links at UCB
28 /vinex/scripts/create-link.sh --link-id 6 --from Renoir  --to cartan
29 /vinex/scripts/create-link.sh --link-id 7 --from VanGogh --to cartan
30 /vinex/scripts/create-link.sh --link-id 8 --from Monet   --to cartan
31 /vinex/scripts/create-link.sh --link-id 9 --from Okeeffe --to cartan
```

Listing 5.1: Shell script for creating the Van Jacobson Experiment topology of Figure 5.1

Next we discuss the content of Listing 5.1:

1. Line 2 of the script creates and starts a single instance of a FreeBSD gateway node used for modelling the network links.

2. The four nodes (`Polo`, `Hot`, `Surf` and `Vs`) at LBL are created and started by the instructions at lines 5 to 9.

3. The UCB nodes (`Renoir`, `VanGogh`, `Monet` and `Okeeffe`) are created by lines 12 to 16.

4. The point-to-point links at LBL are created by lines 19 to 22. The four nodes are configured to pass their network traffic to `csam` (refer to Figure 5.1).

5. Line 25 issues the command to create a microwave link between the `csam` and `cartan` gateway routers. The link is configured with a bandwidth limitation of 230Kbit/s, a packet delay of 100ms and a random packet loss of 1%. The microwave link queue size is set to the default value of 50 slots.

6. Network point-to-point links at UCB are created by lines 28 to 31. The links at UCB are all configured to pass through the `cartan` router node.

The above steps complete the creation of the network topology in Figure 5.1. The viNEX topology-creation script in Listing 5.1 is executed through the Linux command line in `Dom0`. The network nodes and links are created and remain idle, waiting for network traffic to be generated for the experiment. All NetBSD nodes were configured to use the TCP Reno stack and congestion algorithms through the `sysctl` command. Details of the `sysctl` command-line utility may be referenced in NetBSD [57].

Inside each of the sending hosts at LBL, a 1 MB file was created to be sent to the UCB hosts. The hosts were paired to form a total of four simultaneous conversations. The pairing is indicated by the dotted lines between the nodes in Figure 5.1.

The default MTU (Maximum Transmission Unit) of the network interface is 1500 bytes. The frontend interfaces of all the experiment nodes were configured to use a MTU of 552 bytes for the purpose of the experiment. The code of this script is given in Listing F.1 of Appendix F. The purpose of this configuration was to ensure that each sending host sends 2048 packets. The size of each packet was configured to 512 bytes. Note that the remaining 40 bytes (552 - 512 bytes) are used for the standard IP packet header.

To generate the network traffic, a command-line script was written to perform simultaneous FTP conversations between the experiment nodes. Algorithm 5.1 captures a pseudocode description of this process. The script is executed from the Dom0 command shell. For each of the nodes at LBL, the scripts connect with SSH; it then issues a command to send a 1 MB file (with filename "1.MB") using a standard FTP, to the paired host at UCB (see line 6 of Algorithm 5.1). One of the requirements of the Van Jacobson experiment is to start each conversation 3 seconds apart (line 7). The script achieves this by sleeping for 3 seconds before issuing the next FTP command for the next host to start sending information.

The experiment terminates once all the nodes have completed sending their 1 MB files. Next, we present the results we obtained by repeating the Van Jacobson experiment on viNEX.

Algorithm 5.1 Initiate the four simultaneous FTP conversations on viNEX

1: **begin**
2: **for all** $node_i$ in {Polo, Hot, Surf, Vs} **do**
3: $destinationNode_i \leftarrow pairedNodeOf(node_i)$
4: $sshConnectTo(destinationNode_i)$
5: $filename \leftarrow$ "$1.MB$"
6: $ftpTransmitFileInBackgroundMode(filename, destinationNode_i)$
7: $sleep(3)$
8: **end for**
9: **end**

5.1.2 Results

To analyse the behaviour of the sending hosts, we used the `tcpdump` utility at the Dom0 server to capture network packets passing through the backend interfaces of each LBL host. The results of `tcpdump` were captured into a trace file. The trace file contains the recorded IP packets which are used for further analysis. In this manner, we were able to examine the sequence numbers of the packets sent from source nodes. From the trace file that was generated by `tcpdump`, the time and sequence numbers were extracted and plotted against each other. The resulting graph is depicted in Figure 5.2.

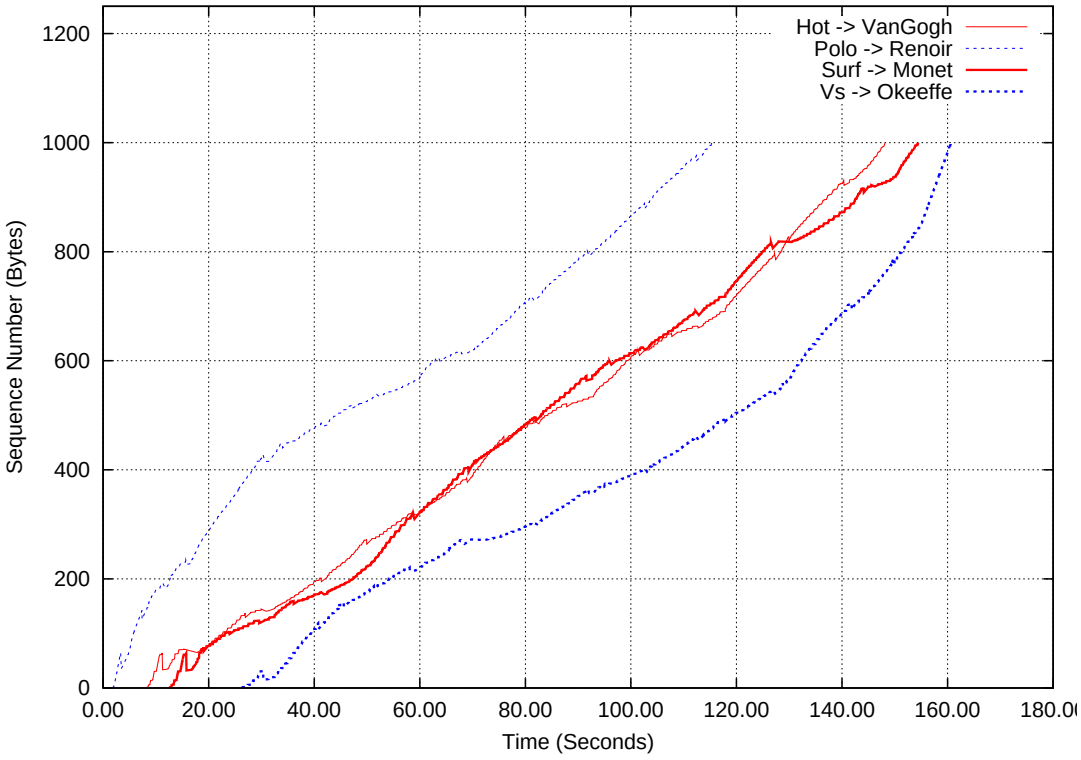


FIGURE 5.2: Analysis of multiple, simultaneous TCP/Reno senders on viNEX with a bottleneck link running at 230 Kbit/s

The graph in Figure 5.2 shows a consistent behavior of the sending TCP stacks. All four conversations show a steady incline with a few dips along the graph. All sending nodes seem to be receiving a fair amount of bandwidth with no single node appearing to monopolise the link.

A total of 7975 packets were sent for all four conversations. The resulting tcpdump showed that 164 out of 7975 packets were retransmitted, implying that approximately 2% of all packets were retransmitted. The retransmissions were due to timeouts as a results of the congested link between the `csam` and `cartan` routers. The number of dropped packets was obtained from the Dummynet pipe used for modelling the link between `csam` and `cartan` routers through the `ipfw pipe list` command. Van Jacobson recorded a retransmission rate of 1% (Van Jacobson [81]).

We measured the throughput of each conversation on viNEX. To measure throughput, we used `tcpdump` to capture the packets that were successfully received by the nodes at UCB. This was achieved by configuring `tcpdump` to listen on the

backend interfaces of the receiving nodes at UCB. The throughput was calculated as the total size of packets received per second. The resulting graph is shown in Figure 5.3.

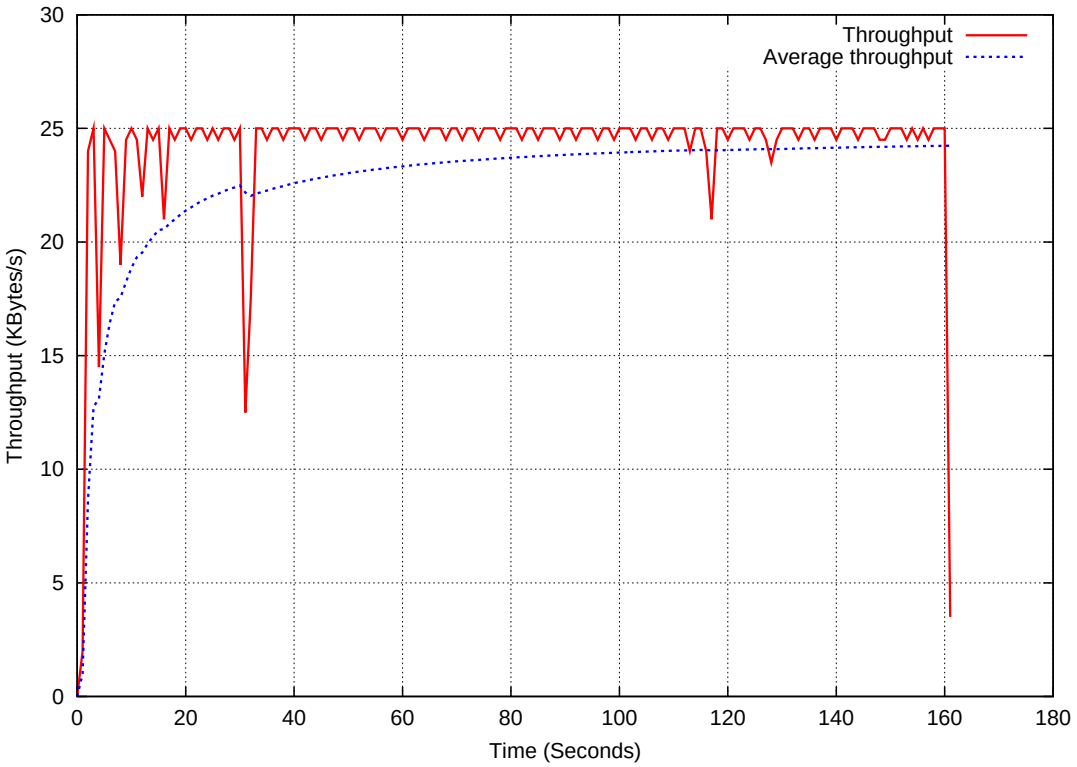


FIGURE 5.3: The throughput achieved by the four TCP/Reno conversations on viNEX as depicted in Figure 5.1

The dotted curve in Figure 5.3 shows the moving average throughput observed from the beginning of the transmission up to any give point in the graph. The graph shows the sum of all throughputs of the four simultaneous conversations. The maximum of the average throughput was 24.5 KBytes/s. The original Van Jacobson experiment recorded a throughput of 25 KBytes/s (Van Jacobson [81]).

The results obtained on viNEX above compare well with those obtained by Van Jacobson. The throughput measured on viNEX above is close to the throughput measured by Van Jacobson. This leads us to an important observation.

- ***Observation #1:*** The results obtained by conducting the Van Jacobson experiment on viNEX are comparable to the results obtained by Van Jacobson in his original experiment. It is plausible therefore that our viNEX emulator can be used for conducting network research experiments.

The following section describes the relationship between the kernel network memory and Dummynet queue size on the FreeBSD gateway node.

5.2 Measuring Dummynet kernel memory

In FreeBSD, the memory allocated to store network traffic is managed by the kernel IPC (InterProcess Communication) subsystem. The basic unit of memory management is known as the `mbuf`. An `mbuf` is 256 Bytes in size [27]. The size of an Ethernet packet is 1500 Bytes and therefore it does not fit into one `mbuf`. As a result, the `mbuf cluster` structure is used. A single `mbuf cluster` is 2 KBytes in size. For Ethernet packets, the remaining space (548 bytes) is not used.

The above `mbuf` and `mbuf cluster` sizes are used as default configurations for a FreeBSD OS running on a physical machine. In the Xen virtual environment, the memory settings have been adjusted to optimise performance. The FreeBSD OS for Xen uses 4 KBytes as the size for an `mbuf cluster` which is equal to the size of one memory page. In this configuration, an `mbuf` is still 256 bytes in size. The size of an `mbuf cluster` was made equal to the size of a memory page (4KB) to enhance the performance of network communications between domains.

Naturally, the memory capacity of viNEX is dependent on the above `mbuf cluster` size. The capacity (C) is calculated as the product of bandwidth and propagation delay. Given a bandwidth of B Kbytes/s and a propagation delay of d seconds, the capacity (C) is calculated by (5.1) below.

$$C = B \times d \tag{5.1}$$

In FreeBSD, memory is allocated in block sizes of 4 KBytes instead of the standard 2 KBytes. This adds a significant overhead to the memory measurements as we show next.

Every `mbuf cluster` (4 KB) has one `mbuf` (256 bytes) allocated with it. If the total number of packets in a Dummynet queue buffer is 2315 (say), the total

amount of memory allocated by FreeBSD to store these packets is:

$$\begin{aligned} Memory &= (2315 \times 4) + (2315 \times (256/1024)) \\ &= 9260 + 578.75 \\ &= 9838.75\ KBytes \end{aligned}$$

We return to these calculations in Section 5.4.3.

For Dummynet, the network packets for the queues are stored using the `mbuf` and `mbuf cluster` architectures. Dummynet does not make physical copies of the network packets into its queues but only keeps track of the pointers to the `mbuf` and `mbuf structures`. This improves the performance and efficiency of the Dummynet operation. The performance is improved on as a result of avoiding the memory copy and move instructions for each packet received. The `mbuf` and `mbuf clusters` are allocated once when the arriving packets are copied from their input network interfaces into the kernel memory space.

In addition to the memory used for storing network packets a special `mbuf tag` (`m_tag`) is created and a pointer to the `m_tag` is inserted into the `mbuf` for each packet processed by Dummynet. Mbuf tags are used to carry the meta-data related to a packet, e.g. for Dummynet the `mbuf` tags are used to carry the state of the packet. The state is used to check for any special treatment of Dummynet packets. Mbuf tags are allocated separately from the `mbuf` and `mbuf cluster` memory allocations. The size of an `mbuf tag` structure varies and the maximum size set to 256 bytes.

Both `mbuf` and `mbuf clusters` are allocated using the `uma(9)` kernel zone allocator function in C. The `mbuf tags` are allocated dynamic kernel memory space using the `malloc(9)` function. This distinction is important for any accounting function of memory usage. In order to account for `mbuf` and `mbuf cluster` memory usage in FreeBSD, the `vmstat -z` command is used. The amount of `mbuf tag` memory allocated can be returned by executing the `vmstat -m` command.

The memory allocated to store an `mbuf tag` is relatively small compared to the UMA (Universal Memory Allocator) memory used for storing both `mbuf` and `mbuf clusters`. Our analysis will therefore focus on the measurement of the `mbuf` and `mbuf cluster` memory usage in the UMA zone.

5.3 Measuring one-way-delay

Network experiments for TCP protocols normally make use of the results of the `ping` command to estimate network packet delays. Such delays are normally referred to as as round trip time (RTT) (Choi and Yoo [11]). To determine a one-way network delay, one could use the value of RTT/2. However, this may not be a realistic measurement since it does not take into consideration the direction in which network congestion could occur. For example, a large value of RTT does not mean congestion is equal in both directions. Congestion may be asymmetric and any calculation of a one-way-delay has to take this into account.

We use the UDP (User Datagram Protocol) for measuring the scalability and performance of network links under viNEX. The `iperf` (Gates and Warshavsky [28]) tool is used for generating UDP traffic on the client side and also acts as a receiver on the server side. `Iperf` calculates and reports on bandwidth and network jitter, but it does not report on any one-way-delays. Various analytical models have been constructed in an attempt to estimate end-to-end delays. One such model is given in (Jin-Hee Choi [45]).

For the purpose of conducting experiments in viNEX, we extended the `iperf` tool by modifying its source code to include the one-way transmission delay of packets in the report. The list of source code changes done to iperf appears in Appendix G. The end-to-end delay can also be viewed as the amount of time a packet spends in transit while moving from the source to the destination.

One of the main challenges of calculating an end-to-end delay, stems from a requirement that the clocks of both the sender and receiver are always synchronised. One possible solution to this challenge could be to make use of the Network Time Protocol (NTP) [51]. This would require NTP to be configured on each topology node. Fortunately for Xen, the clocks of the guest domains are synchronised with the `Dom0` clock by default, during startup. Synchronisation of clocks can be enabled or disabled using the `sysctl` command.

The logic of the enhancements done to `iperf` for reporting on one-way-delays is presented in Algorithm 5.2. The essence of Algorithm 5.2 is: If `iperf` is running in UDP mode, the client embeds the sending time (`sendTime`) to every datagram just before it is transmitted. Our modifications were done to the `iperf` server (receiver). At the receiver end we modified the program to determine the time a

packet was sent and to capture the arrival time of the UDP datagram. The arrival time is taken at line 4 and the time the packet was sent is extracted at line 7.

The transit time for each packet is calculated by subtracting the send time from the packet arrival time (line 8). The transit times of all successfully received datagrams are summed and divided by the number of datagrams to obtain the moving mean transit time. The calculation of this mean is performed by the `calculateMovingMeanTransitTime(...)` function at line 10. This answer is then used in the `iperf` periodic reports (line 11) as well as the final transmission report on the server side.

Algorithm 5.2 One-way-delay enhancement to the `iperf` tool for one datagram

```
 1: {Modified version of the iperf receiver}
 2: begin
 3:   if newUDPDatagramArrived() then
 4:     receiveTime ← getTimestamp()
 5:     datagram ← receiveDatagram()
 6:     if isEnabled(UDP) then
 7:       sendingTime ← getSendingTime(datagram)
 8:       transitTime ← (receiveTime − sendingTime)
 9:       meanTransitTime ←
10:         calculateMovingMeanTransitTime(transitTime)
11:       printPeriodicReport(meanTransitTime)
12:     end if
13:   end if
14: end
```

Note that we have an implicit loop in line 10 where we determine the moving average for datagrams received up to that point.

The next section presents the experiment conducted to assess the scalability of viNEX.

5.4 Scalability of viNEX

In viNEX, all links are configured to pass through the FreeBSD gateway node as described in Chapter 4. It is therefore imperative to analyse the amount of computing resources required for creating a link across the FreeBSD gateway.

The main objective of this investigation is to measure the amount of physical memory required to model a network link on the FreeBSD gateway. The memory

required is comprised mainly of the Dummynet memory used for storing network packets in the queues associated with link pipes. Although it is important to also measure the amount of CPU time required by the FreeBSD to run the network link, such study is a topic for future research.

In order to measure memory requirements, we had to generate a large number of network packets to keep the Dummynet queues constantly occupied with network traffic. Consequently, the UDP protocol was chosen for its ability to inject network packets constantly without having to wait for acknowledgments. UDP client applications (such as iperf (Gates and Warshavsky [28])) are capable of generating packets at a constant rate. For the purpose of this experiment, the `iperf` UDP client was configured to generate packets at a constant rate of `5 MBytes/s`.

TCP was not an option in our work owing to its self-clocking feature. In the event of a Dummynet link becoming congested, a TCP sender will adjust its sending rate by using appropriate congestion control mechanisms, which could result in a significant reduction of the number of network packets sent.

5.4.1 Experiment setup

To measure memory utilisation, we conducted an experiment on viNEX using a topology with three network nodes as depicted in Figure 5.4. The three nodes are connected by two links to form a chain topology. The bandwidth for each link (`Link-1` and `Link-2`) was set to `768 Kbit/s`.

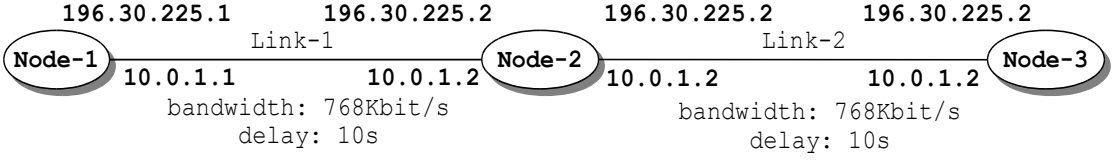


FIGURE 5.4: The chain network topology used to measure memory utilisation of the FreeBSD gateway node

Each link in Figure 5.4 was configured with a significantly high delay of 10 seconds. The purpose of using such a large delay was to allow for a high build up of memory allocations on the FreeBSD gateway, i.e. to have sufficient amounts of data for our observations. The network MTU was set to 1500 bytes which is equal to the

standard Ethernet frame size (IEEE [39]). The queue size for each link was also configured to a value of 96 KBytes to match the configured bandwidth of 768 Kbits/s.

5.4.2 Experiment procedure

To conduct our experiment, we used the `iperf` (Gates and Warshavsky [28]) traffic-generating tool to generate UDP (User Datagram Protocol) packets. UDP was chosen for its ability to send packets at a much higher rate when compared to TCP (He and Chan [34]).

We configured the `iperf` client to send datagrams at a rate of 5 MBytes/s for 120 seconds to ensure a sufficient amount of traffic being generated for the FreeBSD gateway to handle. On the server side, `iperf` was used as a measurement tool by generating reports on the performance statistics of the UDP protocol. Our main focus was on the throughput and the one-way-delay outputs. `Iperf` was configured to report on *jitter*, throughput and one-way-delay every second. The `iperf` server automatically generates a summary report at the end of the 120s transmission time.

On the FreeBSD gateway node, the analysis was focused on measuring the amount of kernel network memory allocated to manage the packets in the Dummynet queues. Network memory was measured by using the `netstat -m` command. The amount of memory reported on by `vmstat -m` equates to the number of `mbuf` clusters and mbufs allocated in the UMA zone. To view the number of mbufs and `mbuf` clusters, the `vmstat -z` command was used.

As mentioned in Section 5.2 above, for the purpose of this investigation, the amount of UMA memory allocated to the `mbuf` and `mbuf` clusters for storing network packets will be measured. The memory allocated for storing `mbuf` tags storage can be obtained by using the `vmstat -m` command.

5.4.3 Discussion of results

The results of this experiment are shown in Figure 5.5. The following five elements were measured during the experiment:

1. Kernel memory allocated to store the transit packets at any point during the transmission (Figure 5.5(a)).

2. Total size in number of packets stored in the four Dummynet queues used for modelling the two links of the experiment topology in Figure 5.4. Figure 5.5(b) depicts this information.

3. The size of Dummynet queues in byte units, shown in Figure 5.5(c).

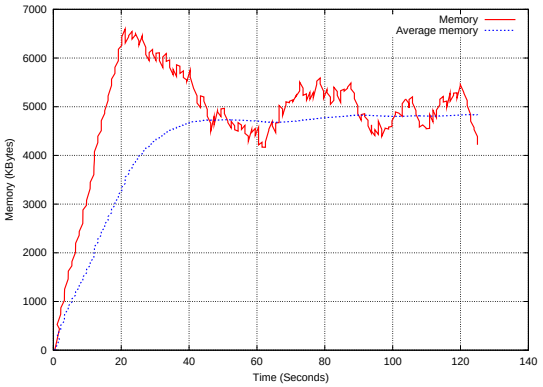
4. Throughput achieved across the connection as measured at the receiver node (Figure 5.5(d)).

5. One-way-delays of packets measured at the receiver (Figure 5.5(e)).

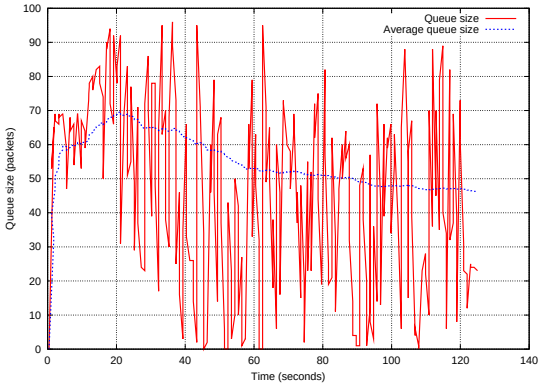
Figure 5.5(a) shows a graph of the kernel memory usage plotted over time. Recall that the experiment was executed for 120 seconds. The graph shows the total amount of kernel memory allocated to store network packets being transmitted over `Link-1` and `Link-2` of Figure 5.4.

Considering Figure 5.5(a), we observe the initial, steep incline for the first 20 seconds of the transmission. During the first 10 seconds, the UDP packets are simply building up in the queue for `Link-1`. After the first 10 seconds, the packets begin to emerge from `Link-1` and start filling up the queue for `Link-2`. The system is keeping more and more packets in transit during this stage. After 20 seconds, packets start emerging from the queue of `Link-2` and move towards their destination (`Node-3`). The gradient of the curve declines as a result of packets moving out of the queues for the remainder of the experiment. The sending UDP maintains its constant sending rate and thereby keep the queues of `Link-1` and `Link-2` at relative capacity for the 120-seconds duration of the experiment.

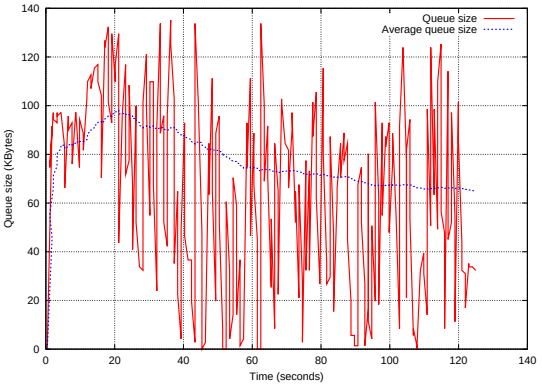
Next, we turn our attention to memory allocation. Rizzo and Carbone [71] claim that, in the worst case scenario, the amount of memory used for storing network packets belonging to the Dummynet queues and pipes, is proportional to the sum of the capacity of each pipe. Recall that the capacity of a single pipe is given by equation (5.1) above. Consequently, for a number of viNEX pipes we propose formula (5.2) to be an upper bound on the total amount of memory used for all pipes. In particular, if we have a total of n viNEX pipes and B_i and d_i are the *bandwidth* and *delay* configured for $pipe_i$ respectively, the total amount of memory (M) allocated for packet storage in viNEX is given by:



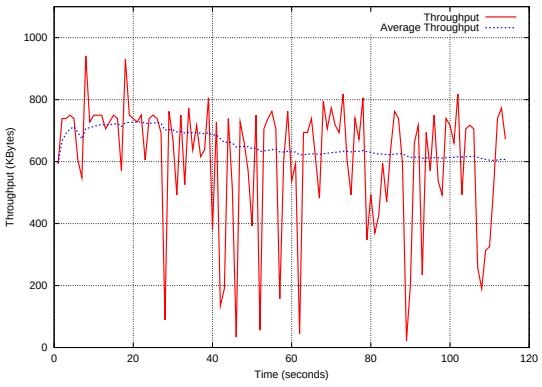
(a) Kernel memory allocated for storing transit packets



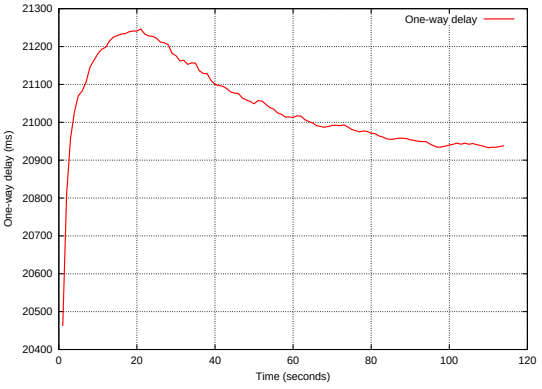
(b) Number of packets in the queue



(c) Queue size in KBytes



(d) Throughput measured at the receiver, configured bandwidth is 768 Kbit/s



(e) One-way delay measurements

FIGURE 5.5: The memory allocated, queue size (in packets), queue size (in Kbytes), throughput and one-way-delay measurements

$$M \leq \sum_{i=1}^{n} B_i \times d_i \tag{5.2}$$

To validate formula (5.2), we calculate the expected amount of memory usage for a number of points on the graph in Figure 5.5(a) and compare each of these with the corresponding value measured by during the experiment. For example, if we select a point during the first `20` seconds, say `t = 20`, we arrive at the following:

According to the graph in Figure 5.5(e) the value of the one-way-delay at `t = 20` is `21 241` milliseconds (or 21.241 seconds). For a mathematical approximation of the corresponding value, recall that in Section 5.2 we identified a single `mbuf cluster` to be of size 4 KBytes and an `mbuf` to be of size 256 bytes. Both these structures are allocated to store a single network packet. Both `Link-1` and `Link-2` are configured with a bandwidth of 768 Kbit/s = 96 KBytes/s each. Therefore, according to formula (5.2), the total amount of memory used to store all the packets at t = 20 is:

$$\begin{aligned} M &= (4 + (256/1024)) * 96 * 21.241 \\ &= (4.25 * 96) * 21.241 \\ &= 8666.3 \ KBytes \end{aligned}$$

According to Figure 5.5(a), at t = 20s the actual amount of memory used was measured to be 6261 KBytes which is less than the calculated value of 8666.3 KBytes obtained by formula (5.2).

The above procedure has been applied to further values of `t` in Figure 5.5(a) and the results are shown in Table 5.1.

TABLE 5.1: A comparison of calculated memory values to the measured values

Time `t`*(s)*	**Formula (5.2)** *(KBytes)*	**Figure 5.5(a)** *(KBytes)*
20	8666.3	6261
40	8608.8	5753
60	8573.3	4579
80	8556.1	5353
100	8543.5	4837
120	8542.7	5282

From the above analysis, we observe that the values calculated by the formula (5.2) are always larger than the values measured in Figure 5.5(a). Therefore, the results in Table 5.1 support the conclusion that formula (5.2) can be applied in calculating a reasonable determinant of an upper-bound on the total amount of memory required in a worst case scenario to emulate network links.

- ***Observation 2:*** Given an experiment topology together with bandwidth and packet delay requirements, formula (5.2) gives a reasonable determinant of an upper bound on the the total amount of memory required for the worst case scenario to host network links on viNEX.

The average throughput measured from the receiver was 735 KBit/s, which is about 4% less than the configured bandwidth of 768 Kbit/s. There could be a number of factors contributing to this degradation. One possibility could be linked to the granularity of the Xen CPU scheduler for allocating CPU time to the domains. Every node is allocated a small chunk of CPU time to execute and once its quantum expires, the domain is paused and the next domain is allocated CPU time. The study on the impact of CPU scheduling to such environments is beyond the scope of this research and is a topic for future work.

5.5 Summary

In this chapter we furnished details of two network experiments we conducted on viNEX. The first experiment was aimed at verifying the operation of viNEX. This was done in Section 5.1. The Van Jacobson experiment was conducted on viNEX and the results were compared to the original experiment as described in the paper [81]. Although we only repeated the experiment using the newer version of TCP (TCP/Reno), our observations were quite similar to what was observed by Van Jacobson [81]. This result led us to Observation #1, namely, that our viNEX emulator can be used for conducting network research experiments.

In Section 5.4, we conducted an experiment to assess any scalability constraints on viNEX. Since all the links are configured to pass through the FreeBSD gateway node, our focus was to identify any computing resource limitations that may be imposed on that node. The primary constraint was identified to be the amount of

memory required to store packets belonging to the Dummynet queues and pipes for modelling the links. This result led us to Observation #2, namely, formula (5.2) presents an upper bound on the total amount of memory required to host network links on viNEX.

For each network packet, the kernel allocates a total amount of memory to the value of 4.25KB which is comprised of a 4KB mbuf cluster and a single mbuf of size 256 Bytes. In a worst case scenario, the amount of memory required to store network packets for a link is proportional to the capacity of the link given by equation 5.1. The UDP was sending at a much higher rate compared to the link bandwidths, resulting in many packets being queued for a longer periods in the Dummynet queues. The one-way-delays were increased as a result. We enhanced the `iperf` tool to calculate these delays. In turn these are used in the calculation of link capacities as reported on in Section 5.3.

The next chapter presents the conclusions to our work and recommendations for future research in this area.

Chapter 6

Conclusions and Future Work

This chapter presents some conclusions that may be drawn from our work and considers possible future work in this area. Section 6.1.1 revisits our research hypothesis and analyses to what extent it has been validated. Our conclusions and summary of results are discussed in Section 6.1.2. The advantages and disadvantages are discussed in Section 6.2 and Section 6.3 respectively. Finally, in Section 6.4, we provide some proposals for future work.

6.1 Contribution of this work

6.1.1 Hypothesis

In Chapter 1 we proposed the following hypothesis:

A general purpose virtualisation platform (such as Xen) can be used to build a network emulator (viNEX) which satisfies the following conditions:

1. The emulator can be used for conducting network research experiments on networks which fall within limited performance boundaries.
2. Such an emulator (viNEX) can be hosted on a single server.

6.1.2 Justification of the hypothesis

In this dissertation we presented the design and implementation details of a virtual network emulator (viNEX) based on the Xen virtualisation platform. The approach used for designing viNEX was based on Emulab (White et al. [85]). Two separate networks were constructed, namely, the control network (described in Section 3.1.1) and the experiment topology network (discussed in Section 3.1.2). The control network was used for the configuration and management of the experiment nodes. No experiment traffic was allowed to flow across the control network. The experiment topology network is dedicated for network traffic related to the experiment under investigation.

Two experiments were conducted on viNEX, namely, the Van Jacobson experiment and the scalability experiment, both reported on in Chapter 5.

The purpose of conducting the Van Jacobson experiment was to verify the functionality of viNEX. Four nodes at the LBL laboratory were connected to four corresponding nodes at the UCB laboratory. A microwave link was used to connect the two networks by linking the gateway routers. In viNEX, the microwave link was emulated by a 230 Kbit/s connection between the gateway routers. In order to emulate the behaviour of a microwave link, a probability loss rate of 1% was configured on the link. We then initiated four simultaneous TCP streams from each node by sending a 1MB file, using FTP, from LBL to UCB.

Owing to a lack of availability of the original 4.3 BSD TCP stack, we conducted the Van Jacobson experiment using the TCP Reno which includes the congestion control algorithms. The entire transmission took 160 seconds to execute on viNEX. The observation made by Van Jacobson [81] showed a total transmission time of 180 seconds for the whole experiment. We also measured the throughput of each connection at the receiving nodes, using `tcpdump`.

Following on the outcome of the Van Jacobson experiment on viNEX (in Section 5.1) we made the following observation:

- ***Observation 1:*** The results obtained from the Van Jacobson experiment conducted on viNEX are comparable to the results obtained by Van Jacobson in his original experiment. We therefore conclude that the viNEX emulator can be used for conducting network research experiments.

From the above observation, we infer that viNEX is operational and therefore can be used as a network emulator for research purposes. Since viNEX was built using a traditional virtualisation platform (Xen) we conclude that the *first* part of our hypothesis, as stated in Section 6.1.1, has been proven.

The second experiment (Section 5.4) aimed to identify a limit on the total amount of kernel network memory required on the FreeBSD gateway node to host all viNEX network links. Each network link on viNEX requires some amount of memory to store the network packets that are in transit at any point during network transmissions. The FreeBSD gateway node should have sufficient memory to carry packets from all links that form the experiment topology. Insufficient memory on the gateway node may lead to a performance bottleneck. The outcome of the experiment indicated that formula (5.2) represents an upper bound on the amount of memory required to store packets in the *worst case scenario*.

During the above scalability experiment, we made the following observation:

- ***Observation 2:*** Given an experiment topology together with bandwidth and packet delay requirements, formula (5.2) gives a reasonable determinant of an upper bound on the the total amount of memory required for the worst case scenario to host network links on viNEX.

For the ease of reference, we repeat formula 5.2 below:

$$M \leq \sum_{i=1}^{n} B_i \times d_i \tag{6.1}$$

Observation 2 implies that we can determine an upper bound on the amount of memory required to run the FreeBSD gateway node. Additional memory is required to host the NetBSD nodes that are used to form the experiment topology. As mentioned in Section 3.1.2.1, each topology node is configured to use 32 MB of RAM from `Dom0`. Therefore, given the number of topology nodes required for an experiment as well as the bandwidth and delay properties for each network link, it is possible to estimate the total amount of memory required on a *single* server to host the experiment on viNEX. From *Observation 2* above, we therefore conclude that viNEX can be hosted on a single server provided it has enough memory resources to run the FreeBSD and NetBSD nodes, thereby validating the second part of our hypothesis in Section 6.1.1 above.

6.2 Advantages

In this section we discuss some of the advantages of viNEX.

6.2.1 Free and open source software (F/OSS)

ViNEX was developed using F/OSS [60]. Some of the major software components used include `Xen`, `FreeBSD`, and `NetBSD`. We have also used some open source tools such as `iperf`, `tcptrace`, `zebra ospfd` and `pdfcrop`. During the development of viNEX, we realised some important advantages of F/OSS. These are:

- A researcher has direct access to the primary software authors and experts in the field.
- The source code can be modified to meet our specific requirements. The Xen kernel was recompiled with the `vmxenabled` compiler option set to `yes` to enable the boot-up of the NetBSD HVM guest kernel. Through direct access to its source code, the `iperf` tool was enhanced to allow for the measurement of one-way-delays of UDP packets (see Section 5.3).
- One may scale to arbitrary instances without additional licensing constraints. We can configure an arbitrary number of NetBSD instances and enable all of them without further licensing restrictions.
- A developer can learn from the readily available source code. Very little is documented on the internal operations of Dummynet. Dummynet technical documentation is included as part of the `C` source files, namely, `ip_dummynet.c` and `ip_dummynet.h`.
- One may package and distribute enhancements without additional licensing costs. Part of our future work is to package viNEX onto a CD or upload it to SourceForge [77] to allow for a free distribution to network researchers. SourceForge is a location from which one may download F/OSS software.

6.2.2 Standard network protocols on viNEX

We deployed standard network protocols on viNEX without any modifications to their source code. For example, we were able to deploy the standard RIP

and OSPF routing protocols. The transmission of network traffic between any pair of viNEX nodes takes place at layer-2. The ARP (Address Resolution Protocol) is enabled to allow for IP to MAC address resolution.

6.2.3 Managing and configuring viNEX experiments

A set of scripts was developed to simplify the management and configuration of experiments on viNEX. Experiments can be created and configured by using a set of four scripts: `start-gateway.sh`, `start-node.sh`, `create-link.sh` and `modify-link.sh`. The source code of each of these scripts is listed in Appendix B.

6.2.4 viNEX as a research tool

We also propose viNEX as a tool that can be used in academia for teaching computer networks. ViNEX provides an environment in which students can explore network protocols and make observations that may facilitate their learning process. For example, a trace of network packets in a TCP connection can be captured using `tcpdump` into a text file. The text file may subsequently be used to analyse various properties of the TCP packets.

6.2.5 Availability of viNEX

ViNEX is available to users. Unlike Emulab which is available only through an Internet connection, viNEX can be deployed on a local computer and does not require an Internet connection in order to conduct experiments on it.

6.3 Disadvantages

During the development of viNEX, we identified a number of disadvantages to be addressed as future work.

6.3.1 Inefficient memory allocation

Since every network link is configured to pass through the FreeBSD gateway node, it was imperative to analyse the amount of memory required by this node. Analysis of the memory requirements was done in Section 5.4. For each network packet a total of 4352 bytes is allocated for storage. This is made up of a single 4 KByte `mbuf cluster` and a 256 byte `mbuf`. The `mbuf cluster` was adjusted to 4 KBytes in an attempt to improve on network performance in viNEX. Performance is improved as a result of the reduced number of calls to the Xen Hypervisor. ViNEX is configured to use an MTU of 1500 (the standard size of an Ethernet frame). The maximum size of an IP packet is therefore 1500 bytes. This implies that only 1500 bytes of the 4352 bytes are used and the remaining 2852 is not used. Therefore about 65% of the allocated memory is unused.

The amount of unused memory can however be lowered. This may be achieved by lowering the globally configured size of an `mbuf cluster` at the FreeBSD operating system level. The size of an `mbuf cluster` is controlled by the kernel option. The `mbuf cluster` size has to be a power of two, and is computed using $2^{\texttt{MCLSHIFT}}$. The lowest value of `MCLSHIFT` is 11 which sets the size of `mbuf cluster` to $2^{11} = 2048 = 2$ KBytes. About 548 bytes (26%) will remain unused which is a significant improvement, compared to the above 65%. `MCLSHIFT` is a kernel option which can be configured before the kernel is compiled.

Alternatively, to prevent the above inefficiency of 65% of allocated but unused memory, the network MTU can be increased to 4096 bytes so that `jumbo` network frames are used. This is however a non-standard Ethernet frame size. As a result, viNEX nodes will not be able to communicate with external Ethernet networks that use a standard MTU of 1500 bytes. It is possible that this problem may be solved as part of future Xen releases.

6.3.2 Tracing of dropped packets on viNEX

There are two locations where network packets may be dropped during the execution of an experiment, namely, at the Dummynet pipes that are hosted inside the FreeBSD gateway node and also at the NetBSD experiment nodes. It is imperative for researchers to have a view of the number of packets dropped at both locations.

Currently we do not have a tool to automatically trace the number of dropped packets in viNEX. Development of such a tool is a topic for future work. It is, however, possible to trace dropped packets on viNEX using a manual procedure as follows. The number of packets dropped on the Dummynet pipes can be obtained by using the `ipfw pipe show` command which generates a report on the number of packets dropped in each pipe queue. Network packets dropped on the NetBSD nodes can be obtained by using the `tcpdump` command inside the domain and capturing the output to a text file.

6.4 Future work

We may improve on viNEX in a number of ways, and in this section we elaborate on some of these.

6.4.1 Extending NS-2

The experiment configuration and control scripts were written using the bourne shell scripting language on the Linux platform. The viNEX scripts are presented in Appendix B. These scripts are not flexible and may be hard for a researcher to use. To configure complex experiments, a sound knowledge of the shell scripting language is required. For example, to create an FTP agent for generating experimental source traffic, users have to write their source code in a shell scripting language. The code has to make use of the SSH tool to open a connection to the source node and then execute the FTP command.

The NS-2 notation is well known among network researchers and it is used for configuring experiments on the Emulab testbed (Anderson et al. [3]). We aim to use the NS-2 Tcl-based format in future to describe the experiment network topologies in viNEX. This will replace the need to use the shell scripts. Researchers may benefit from the simpler and more flexible NS-2 notation. We identified six components of NS-2 to be enhanced for integration with viNEX.

- **Tcl scripts:**

  The basic Tcl commands of NS-2 could be extended to invoke the viNEX

shell scripts. These include basic commands such as `node` and `link` to create network topologies.

- **Agents:**

  The collection of classes used for configuring the transport layer (4) of networking in NS-2 is called Agents. In viNEX, we could extend the `C` classes of NS-2 to include a viNEX TCP Agent. The viNEX agent could then be used to configure the network on the NetBSD experiment nodes. For example, to use TCP/NewReno the viNEX agent can simply set the `sysctl` configuration for selecting congestion control to `newreno`, i.e., `net.inet.tcp.congctl.selected=newreno`.

- **Sinks:**

  Sinks are the consumers of network traffic (refer to the NS-2 manual [52]). They are responsible for receiving network traffic and to implement protocol-specific behaviour. For example, the NS-2 Agent/TCPSink class is used to simulate the TCP protocol receiving functions such as the control of network traffic by generating acknowledgments. For viNEX, this function can be used to configure the specific `sysctl` settings on the receiving side, e.g. the network receiving buffers. Examples of these are `net.inet.tcp.recvspace` for TCP and `net.inet.udp.recvspace` for UDP.

- **Application Layer:**

  Application classes are used to generate experiment network traffic in NS-2. To model the application layer, a new viNEX application class could be added to the existing class collection of NS-2. For example, to use FTP for generating source traffic, the application class for viNEX can simply output an FTP command into a script which can then be executed by the sending node.

- **Links:**

  Network links on viNEX are full duplex. In NS-2, duplex links are configured by invoking the `duplex-link` function and passing the necessary settings for configuring the link properties such as the delay, bandwidth, loss rate and

the queuing algorithm. The `duplex-link` function can be extended to output the Linux shell command which in turn invokes the viNEX link creation script (`create-link.sh`) and pass the necessary information along.

- **Discrete event model:**

  NS-2 uses a discrete event model for generating network events at a specific time. This is done by using the `at(...)` method. The `at(...)` method can be integrated with viNEX in such a way that it can generate Linux shell commands which, in turn can issue an SSH command to execute the required event on the topology node. For example, if `Node-1` is required to begin an FTP transmission at a 3 seconds apart simulation time. The `at(...)` command can issue an SSH command via the Linux operating system to `Node-1` to begin an FTP at a 3 seconds apart simulation time to perform the necessary function.

In general, the integration of viNEX with NS-2 calls for an extension to the standard NS-2 parser to output a set of shell scripts to construct and execute a viNEX network experiment. The NS-2 `run` command could be used to issue the Linux operating system command to execute a dynamically constructed viNEX script.

6.4.2 Simplifying network packet tracing

Currently, there is no generic way of tracing network packets belonging to a stream in viNEX. To trace network packets, we use the `tcpdump` utility to capture packets and write them to a text file. This a manual and tedious procedure. Currently, therefore researchers are required to manually execute the `tcpdump` utility or to write custom shell scripts for automating this function.

As part of the above extension to integrate NS-2 with viNEX, a more generic packet tracing mechanism is required. Such a mechanism could be integrated to the standard NS-2 packet tracing module. An advantage would be that the NAM (Network ANimator) component of NS-2 could be used to provide a graphical display of network packet flows.

6.4.3 Dummynet on Linux

Dummynet and IPFW have recently been ported to Linux and the source code, modules and binaries are available at the Dummynet site (Rizzo [69]). It is therefore possible to run Dummynet on a Linux host. Consequently, future releases of viNEX could use the Linux `Dom0` for traffic shaping and thereby render the FreeBSD gateway node redundant.

The VLAN devices and network bridges located inside the FreeBSD gateway node add some overhead to the transmission of packets between experiment nodes. The removal of the gateway node will therefore alleviate these overheads, leading to an improved network performance and throughput in viNEX.

Appendices

Appendix A

A Virtual Integrated Network Emulator on XEN (viNEX)

This appendix contains the peer-reviewed paper on viNEX. The paper was published in the proceedings of the 2^{nd} SIMUTOOLS Conference held in Rome (Italy) from 2 - 6 March 2009.

A Virtual Integrated Network Emulator on XEN (viNEX)

Abraham Mukosi Mukwevho [*] [†]
School of Computing
University of South Africa
P O Box 392, UNISA, 0003
mukosi@gmail.com

John Andrew van der Poll
School of Computing
University of South Africa
P O Box 392, UNISA, 0003
vdpolja@unisa.ac.za

Robert Mark Jolliffe
bobjolliffe@gmail.com

ABSTRACT

The recent progress on virtualization technologies has made it possible to deploy multiple hosts instances with operating systems running real network protocol stacks on one single server. The objective of this paper is to explore whether it is feasible to use such environments for network emulation and simulation. Some significant amount of research is taking place in this area, this includes Emulab [6] virtualization, and IMUNES [12] system. Both Emulab and IMUNES are based on FreeBSD Jails.

Very little is known about using traditional virtualization platforms (such as Xen and VMware) for virtual emulators. As part of our research, we will attempt to develop a virtual emulator (viNEX[1]) based on Xen. Having identified the limits and weaknesses of this approach, we also propose some areas where viNEX can be useful.

Categories and Subject Descriptors

D.4.8 [**Performance**]: Measurements; I.6.7 [**Simulation Support Systems**]: Environments

General Terms

Network Simulation and Emulation

Keywords

Computer networks, Simulators, Emulators

[*]Abraham Mukwevho is a M.S student and primary author of this paper at the University of South Africa.

[†]Configuration and Administration **scripts** mentioned in this paper can be accessed online at - **http://sites.google.com/site/mukosi/**

[1]from now onwards, **viNEX** will be used as a short for **V**irtual **I**ntegrated **N**etwork **E**mulator based on **X**EN

SIMUTools March 02-06 2009 Rome, Italy

1. INTRODUCTION

Network research experiments have traditionally been conducted in an *emulated* or *simulated* environment. Emulators (such as Emulab [6]) are normally based on physically deployed networks which are associated with high procurement and maintenance costs, complex configurations and infrastructure (servers, routers and gateways). On the other hand, network simulators such as NS-2 [15] provide a self-contained and simple environment that can be hosted on a single host. Simulators provide a synthetic environment which is only an approximation of the real world and therefore the results might not be a true reflection of real world. Furthermore, network protocol components developed in a simulated environment require a significant amount of code refactoring in order to migrate them into the real world. This is mainly because simulated environments do not run real network protocol stacks, instead they use software modules that mimic real world protocol stacks. It is also possible to combine both emulation and simulation in one environment, for example; in Emulab simulation can be provided by instantiating NS-2 traffic generators or sinks on one of the topology nodes. To overcome limitations associated with simulators, emulators provide an alternative approach whereby network protocols can be developed while interacting with real protocol stacks, and hence eliminating the need to migrate protocol code to the real world. Network emulators have traditionally been based on physical network deployments (a good example is the original Emulab). The recent advances on the development of virtualization technologies has now made it possible to deploy multiple hosts on one single environment and interconnect them to provide a complete network environment. These virtual hosts run real network protocol stacks and therefore provide an emulated environment that can be used for network research experiments.

Our fundamental goal is to explore the possibility of using a traditional virtualization platform like Xen to build a stand-alone network emulator hosted on one single server or PC. Part of the rationale for this research is to be able to create a freely distributable experimental environment for use, for example, by distance education students who don't have computer laboratory access. We are also aware that the software bridges will be a performance hit. What we don't yet know is how slow will be too slow. Furthermore, the viNEX environment is never going to be useful for high performance fast network emulation, but it might still be useful for other educational scenarios.

Our emulator (viNEX) was built using free and open source software. The use of open source software in networking re-

search continues a very long tradition, this includes NS-2, FreeBSD Jails, and Emulab. We selected Xen as the virtualization platform because at the time the research was initiated Xen was the most viable open source platform you could run "any OS on". Despite our focus on building mini-nodes using NetBSD, the system is not, and is not meant to be, restricted to using NetBSD nodes. In other words the ability to run any OS is part of our high level design goals. Other open source technologies used include; FreeBSD 7.0 and NetBSD 4. NetBSD is used to implement the experiment topology nodes. FreeBSD is used to provide traffic shaping and link emulation using Dummynet and IPFW which are also open source technologies.

We would also like to make a special note to our audience - please note that our emulator is work-in-progress therefore it is by no means in a complete status, it an ongoing research work and we are continually improving it. On the other hand, we are aware of some limitations of this approach, we will be addressing some of them as part of the research. All scripts and progress work on viNEX can be referenced online at [13]

The rest of this paper is structured as follows. In Section 2, we give background work in support of this research. The original contribution work (viNEX) is described in Section 3 through to Section 4. We approach the conclusion of this paper by looking at current and other related research work at Section 5. We conclude this paper and give pointers to future research work in Section 6.

2. BACKGROUND

To begin, we provide some overview on *emulation* and *simulation* to form the foundation work for viNEX. Due to the limit in scope for this paper, we could not provide a complete background on Xen networking. A significant amount of technical writing on Xen exists, interested readers can refer to [19], [5] and [24].

2.1 Network Emulation and Simulation

Network research environments can be classified into three categories, i.e. testbed, network simulation and network emulation. A network *testbed* is a physically deployed and configured environment dedicated for conducting network research experiments. It is formed by real networking elements such as end hosts, routers, links (cables), bridges and switches. Some good examples of testbeds include; Emulab [6], and PlanetLab [4]. Both these environments consist of a set of physical servers deployed in a laboratory environment interconnected by switches. Network experimenters normally access these shared testbed environments over the Internet to setup and execute their experiments, results are obtained by downloading captured log files. Some advantages of testbed environments include; experiments are executed in real time and interacting with real protocol stacks deployed at the end hosts and routers. Testbed environments are not dynamic and have a lot of drawbacks; they are difficult to setup, configuration can be tedious and time consuming, physical hardware is extremely expensive to procure, hardware logistic and storage space problems. Furthermore, although offering a real world environment, conducting experiments on testbed offers an uncontrollable, unpredictable and non-repeatable environment.

Network *simulators* are normally implemented as a collection of software modules providing a synthetic network experiment environment. Simulators achieve this by defining and modeling network behavior through the abstraction of network elements, this include a virtual simulated time, and a discrete network events system for traffic generators. Simulators are normally deployed and executed on a single host. Despite the limited emulator functionality of NS-2 [15], NS-2 is a classical and popular example of a network simulator. Network simulators offer a repeatable and controlled environment for experiments. Simulators are easy to setup and configure, and a result, they offer a lot of control to experimenters making them an ideal choice for rapid protocol prototyping and evaluation.

Network *emulation* refers to a hybrid technique that leverages on the features and benefits of both testbed and simulated techniques. Emulation combines the real network elements of the testbed approach to the synthetic or simulated elements of simulation. In most cases, simulated elements of an emulated environment include - network links and intermediate nodes.

Emulab, despite being a physical testbed environment, is a good example of an emulated environment. In Emulab, a transparent FreeBSD delay nodes are inserted between topology links in order to simulate the network boundary conditions using the Dummynet module. NS is also an example of a limited network emulator. NS has recently introduced some limited emulation functionality whereby real network traffic can be subjected to emulated network components. The emulation facility of NS is described in detail at [7].

3. IMPLEMENTATION OF VINEX

The following sections give a technical description of the emulator (viNEX) and its implementation details. Development of viNEX was conducted on a single Linux host, see Table 1 for environment hardware and software configurations;

Table 1: viNEX Development Environment

Operating System	*CentOS 5.1*
Memory	*1 GB*
CPU	*Intel Core 2 Duo, 3.0 GHz CPU with vT Support*
Other Software	*Xen 3.2, NetBSD 4, FreeBSD 7 (with Dummynet and IPFW enabled)*

3.1 Architecture and Design

Figure 1 depicts the high-level architecture of viNEX. The main components of viNEX include: *control network*, *experiment topology nodes*, *traffic shaping node*, *network links*, and testbed *configuration and management scripts*.

3.1.1 Control network

Similar to Emulab [6], a separate *control network* is created to allow users direct access to the experiment *nodes* from within Domain 0. The *control network* is used by setup scripts for access to the nodes in order to configure them for networking by executing commands using SSH. Each topology node (Node X) is assigned a Class C IP address 196.30.225.X for the control network.

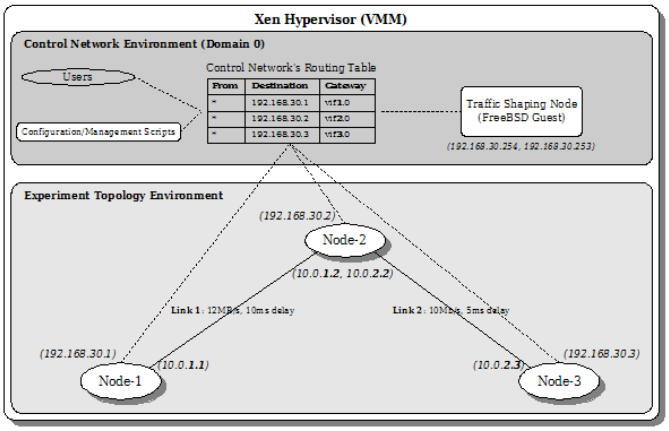


Figure 1: High-Level Testbed Architecture

3.1.2 *Experiment topology nodes*

These nodes form the topology to be used for conducting a network experiment. They are standard Xen HVM guest nodes. All experiment nodes are NetBSD 4 nodes running a minimal kernel.

3.1.3 *Traffic shaping node*

The traffic shaping node is a FreeBSD node configured as a transparent gateway between network links. In addition to link modeling, the traffic shaper is used to model network boundary condition such as: bandwidth limitation, packet delay, and random packet loss.

3.1.4 *Network Links*

Links are used to model communication between any pair of experiments topology node. Links are defined inside the traffic shaper node. The traffic shaper node uses a combination of software bridging together with VLANs in order to model the link between two nodes. Dummynet and IPFW are used for bandwidth and delay simulation.

3.1.5 *Configuration scripts*

This is provided through a collection of Linux shell scripts as follows. All these scripts can be obtained online at [13]:

`start-gateway.sh` - is used to boot the FreeBSD traffic shaping node.

`start-node.sh` - is used for starting any NetBSD experiment topology node as required

`create-link.sh` - is for for creating links between each pair of nodes as specified by the experiment.

`modify-link.sh` - is used to alter the link properties after it has been created.

3.2 Network Links

We now expand and discuss the lower level details of the network link abstraction between any pair of topology nodes. For the purpose of this discussion, please assume the two node topology depicted in Figure 2 below;

A *link* between any pair of virtual nodes is formed by the following network elements; *front-end interfaces, back-end interfaces, vlan subinterfaces, bridges, ebtables rules, ipfw rules and dummynet pipes.* A sample Xen configuration for the scenario in Figure 2 is shown in Listing 1. The important information to note here is the IDs assigned by Xen to each node.

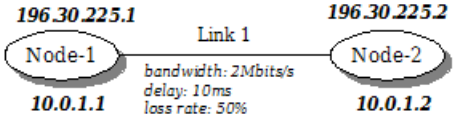


Figure 2: A basic two-node network topology

```
[root@mukosi experiment]# xm list
Name      ID   Mem   VCPUs  State   Time(s)
Domain-0   0  1425     2    r-----   102.0
Gateway    1   512     1    -b----    31.7
Node-1     2    16     1    -b----    12.4
Node-2     3    16     1    -b----    13.1
```

Listing 1: Xen configuration of Figure 2 topology

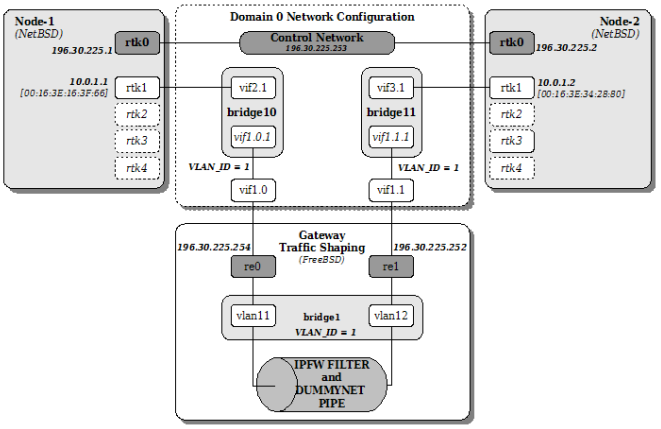


Figure 3: Components of a link connection between two nodes

The following paragraphs briefly describe each network link element as identified above:

3.2.1 *Front-end interfaces*

These are interfaces running inside each domain. Since we are using HVM, all domains are running the native unmodified network drivers. All interfaces involved are depicted in Figure 3. Each node is allocated *five* front-end interfaces, they configured during the node startup process. The lowest interface, `rtk0` is always reserved for the control network and it is automatically assigned the control network IP address during startup. The IP address allocation scheme for the control network is such that for each Node X, interface `rtk0` is allocated a Class C IP address 196.30.225.X.

3.2.2 *Back-end interfaces*

They are interfaces inside Dom0 and directly connected to front-end interfaces inside the topology domains as well as the Gateway node. Looking at Figure 3, Node-1 and Node-2's front-end interfaces are directly connected to back-end interfaces `vif2.1` and `vif3.1` respectively. Similarly,

the Gateway node's front-end interfaces `rtk0` and `rtk1` are connected to back-end interfaces `vif1.0` and `vif1.1` respectively.

3.2.3 Bridges

For each link, two Linux software bridges are created. The purpose for the bridges is to connect the traffic from the topology nodes directly to the Gateway node for traffic shaping in a protocol independent manner. Packets are forwarded based on Ethernet address and not IP address. Both the node's back-end interface and the Gateway's vlan subinterfaces are joined together to allow traffic routing at layer 2, as a result all protocols can be carried across the links.

3.2.4 VLAN Sub-Interfaces

The Gateway node only has two fixed interfaces (`vif1.0` and `vif1.1`) connecting it directly to the Dom0. Since all links have to go through the FreeBSD Gateway node for traffic shaping, we had to derive a mechanism that will allow the sharing of these two fixed back-end and front-end interfaces among all links. For each link X in the topology, a corresponding VLAN with VLAN_ID = X is created in order to isolate the link's traffic. For the example in Figure 2, a VLAN with VLAN_ID = 1 is defined for Link 1 (see Figure 3).

3.2.5 Ebtables Rules

Ebtables [20] is a Linux packet filter that enabled us to intercept bridged traffic at layer 2 and be able to BROUTE them. Packets are forwarded at layer 2 without having to be passed to layer 3 for routing. The PREROUTING chain of the Ebtables NAT table is used to perform the MAC address translation of the destination using the `dnat` instruction. The destination MAC is changed to the MAC address of the directly connected destination as defined by the topology of the experiment before the packet is passed into the traffic shaper Gateway. Using the basic experiment in Figure 2, for each link, two Ebtables rules are created inside the NAT PREROUTING table by the link configuration script - see Listing 2 for details.

```
1 [root@mukosi experiment]# ebtables -t nat -L
2 Bridge table: nat
3 Bridge chain: PREROUTING, entries: 2, policy: ACCEPT
4 -i vif2.1 -j dnat --to-dst 0:16:3e:34:28:80 --dnat-target ACCEPT
5 -i vif3.1 -j dnat --to-dst 0:16:3e:16:3f:66 --dnat-target ACCEPT
6 ...
```

Listing 2: Ebtables rules for the topology of Figure 2

The two Ebtables rules are listed in line 5 and 6 of Listing 2. The rule in line 5 simply translates the MAC address of any frame that arrived through back-end interface `vif2.1` and set it to the MAC address of the front-end interface of Node-2 (`00:16:3e:34:28:80`) so that the frame can be passed directly after being traffic shaped by the Gateway. Similarly, the rule in line 6 is used to translate the MAC address of any frame arriving directly from Node-2 through interface `vif3.1` and set it to the MAC address of the front-end interface of Node-1 (`00:16:3e:16:3f:66`).

3.2.6 IPFW Rules

Packet filtering is specified using a set of rules that are created by using the IPFW command line utility of FreeBSD. See Listing 3 for the list of IPFW rules. Dummynet pipes are also created using the `ipfw` command line.

```
1 Gateway# ipfw show
2 00800  27  1260 pipe 1 ip4 from any to any via vlan11 layer2
3 00900  27  1260 pipe 1 ip4 from any to any via vlan12 layer2
4 65535  19  4943 allow ip from any to any
```

Listing 3: IPFW rules for the topology of Figure 2

3.2.7 Dummynet Pipes

Dummynet pipes are created inside the FreeBSD traffic shaper node by IPFW. They are used for simulating the network adverse conditions such as; delay, bandwidth limitation, probability drop rate, various queueing techniques.

4. PRELIMINARY RESULTS

In this section we provide some of the preliminary results that where captured as part the verification and validation of viNEX. At this stage; it should be emphasized that viNEX is by no means complete, it is in a functional state where basic networking can be accomplished.

Two experiments were run on the six-node dumbbell topology as depicted in Figure 4 below. The main objective of these experiments is to verify if TCP protocol behaves as expected when deployed on viNEX nodes. The first experiment is used to assess the maximum possible bandwidth on viNEX running without traffic loss or delay issues; the second experiment investigates the effects of imposing a delayed and lossy link between Node-3 and Node-4.

4.1 TCP stack and analysis tools used

Since all the nodes envolved in the experiments are NetBSD 4 nodes, we are making use of the latest TCP stack implemented on NetBSD 4, i.e. Reno and NewReno TCP. NewReno TCP is enabled by default in NetBSD. Table 2 lists all the TCP settings that remained constant between Experiment 1 and Experiment 2. NetBSD's NMBCLUSTERS setting was adjusted to 16484 and the kernel was recompiled. In both experiments, we have used the same synthetic load in order make performance comparisons trivial, files of sizes 5MB, 10MB, 20MB, 50MB and 100MB were transferred. In both experiments, data is transferred from Node-1 to Node-6 via the link between Node-3 and Node-4 (see Figure 4). We used the tool `iperf` [14] for traffic generation and maximum bandwidth measurement. TCP flow data packets were captured using `tcpdump` [23] and the tool `tcptrace` [16] was used to analyze them. `xplot` [21] was initially used for graphing but eventually converted xplot datasets to `gnuplot` [9] and used it for graphing. Data conversion was done using the `xpl2gpl` script (located at [16]).

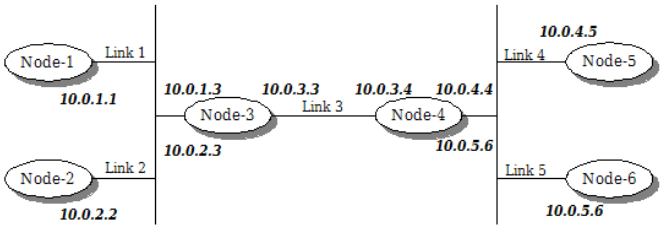


Figure 4: A six-node *dumbbell* topology used for experiments

Table 2: TCP stack settings and configurations

TCP Stack Version	*NewReno + SACK enabled*
Initial Congestion Window	*4 (4068 bytes)*
Send Buffer Maximum	*32Kb*
Receive Buffer Maximum	*64Kb)*
RFC1323 Enhancements	Enabled
NMBCLUSTERS	16384
Traffic Source and Sink	Node-1 and Node-2
Bandwidth Measurement Tool	iperf

4.2 Experiment 1: Maximum bandwidth

The following results were obtained using `iperf` to send data files of sizes 5, 10, 50 and 100MB. TCP statistics were obtained using `tcptrace` tool;

Table 3: Results without any delay and loss

Filesize	5MB	50MB	100MB
Bandwidth (Kb/s)	302.5	320.3	154.43
Data packets:	2405	39400	78483
Ack packets:	1565	25603	51196
Total packets:	3974	65007	129683
Dropped/Rexmt pkts:	0	0	0
Duration (MM:ss.mmm):	0:10.561	02:43.736	11:19.054

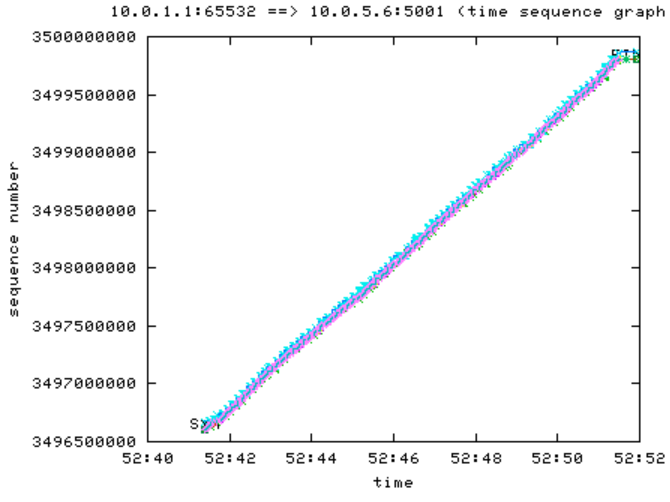


Figure 5: Time Sequence Graph Graph for traffic from Node-1 to Node-2

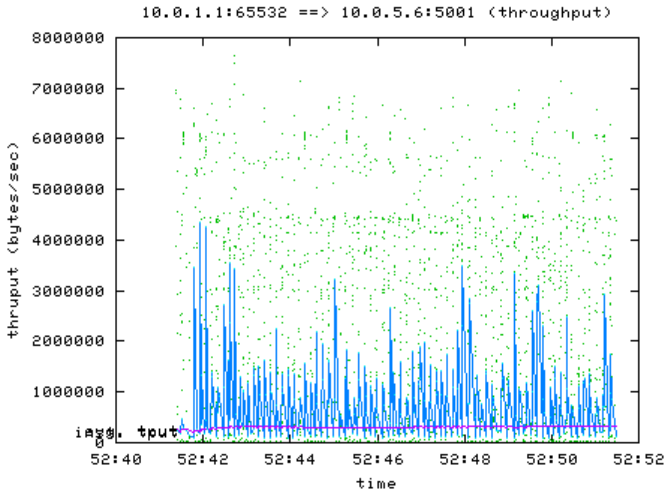


Figure 6: Throughput for the traffic from Node-1 to Node-2

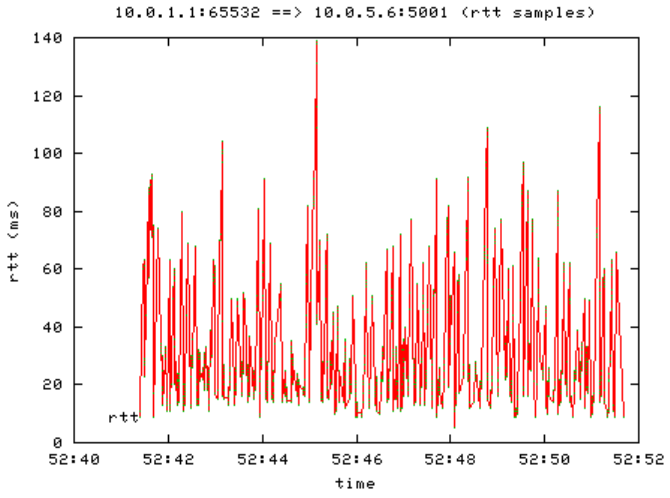


Figure 7: Round Trip Time (RTT) graph for the traffic from Node-1 to Node-2

4.3 Experiment 2: Delay and lossy links

For this experiment, the propagation delay of 20ms and random packet loss of 5% was configured on the link between Node-3 and Node-4. The same data files were transmitted and the results are shown in Table 4 below.

Table 4: Results 20ms delay and 5% loss rate

Filesize	5MB	50MB	100MB
Bandwidth (Kb/s)	72.21	70.67	67.97
Data packets:	4129	41053	82071
Ack packets:	2987	29449	58879
Total packets:	7119	70506	140954
Dropped/Rexmt pkts:	229	2085	4156
Duration (MM:ss.mmm):	01:12.724	12:22.036	25:42.748

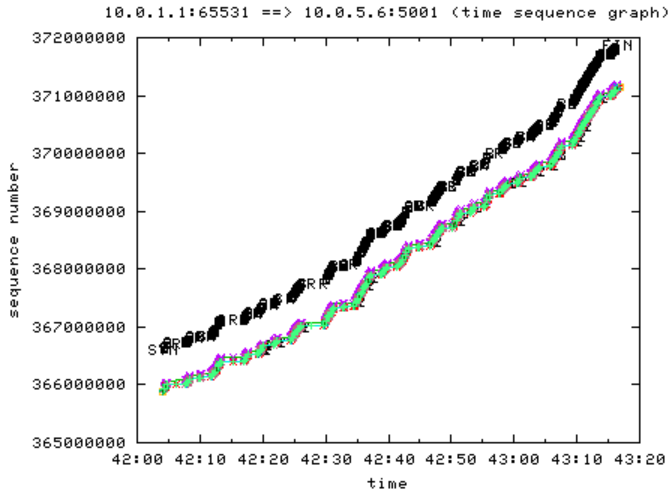


Figure 8: Time Sequence Graph Graph for delayed traffic from Node-1 to Node-2

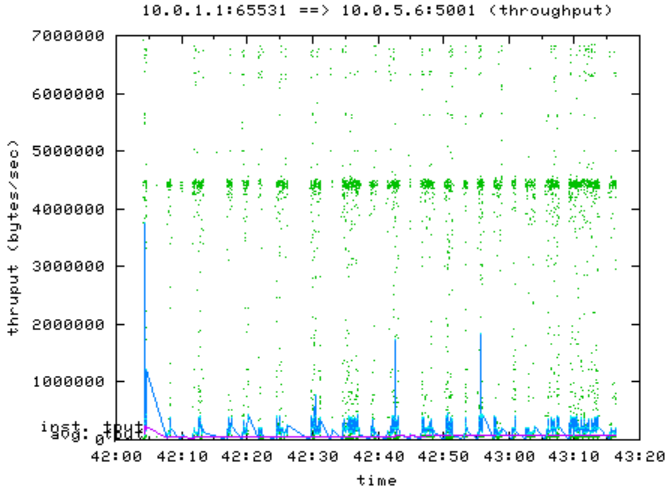


Figure 9: Throughput for delayed traffic from Node-1 to Node-2

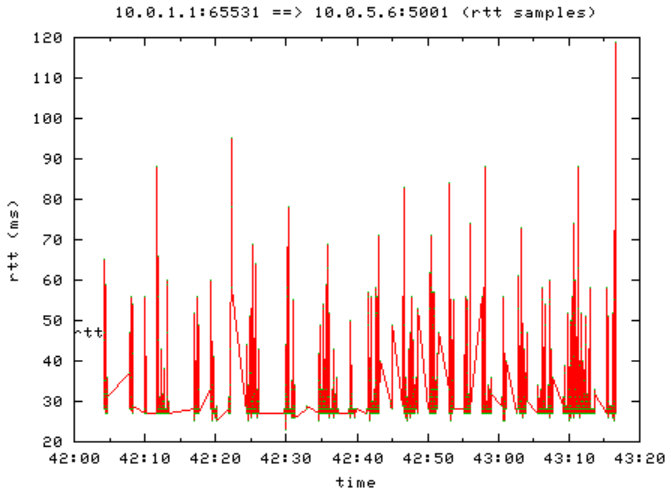


Figure 10: Round Trip Time (RTT) graph for delayed traffic from Node-1 to Node-2

5. RELATED WORK

Virtualization of network emulators is currently receiving a lot of research attention. During the time of this research, we have managed to identify a few number of research work in this space.

The first significant virtualization identified was the current large-scale virtualization initiative being done on Emulab. Instead of using a traditional virtualization tool like Xen, Emulab have chosen the approach of using FreeBSD Jail mechanism. FreeBSD Jail provide a light weight virtualization mechanism through process isolation. See [10] for a detailed description of Emulab's virtualization approach.

UML (User-Mode Linux) has been used quite extensively in virtualizing network emulation, this includes some key research in; 1) the work done using UML (User-Mode Linux) at [22], mainly targeted at evaluating VPN networks, UML was used to evaluate VPN protocols such as PPTP (Point-to-Point-Tunneling-Protocol) and IPSec (IP Security). 2) the UML based emulator for MPLS networks [1], 3) UML was also used in the implementation of VNUML (Virtual Network UML) [8], VNUML is mainly targeted at the evaluation of ipv6 routing protocols.

FreeBSD has also been used to virtualize network emulators; 1) the FreeBSD network stack was virtualized through the cloning technique that allows for multiple network stacks on the same kernel as proposed in [25]. This approach depends on the FreeBSD Jail [18] framework for application environment isolation. Each instance of the protocol stack resembles a full network stack capable of running network routing protocols as well as networking applications. 2) IMUNES is another example, it was proposed in [12]. IMUNES also extends the FreeBSD kernel by enabling it to maintain several networking stacks that are used to run different networking applications. 3) ENTRAPID introduced the approach of virtualizing different 4.4BSD kernels. This enbaled the deployment of different network protocol stacks on the virtualized kernels. ENTRAPID is described in [11].

Further examples of network emulators virtual attempts include; the hypervisor based testbed at [3] aimed at conducting network security experiments, the virtual integrated TCP testbed (or VITT) aimed at evaluating TCP performance at [2], another research is looking at the possibility of using paravirtualization as the basis for a federated PlanetLab architecture at [4] - PlanetLab [17] is a testbed aimed at rapid prototyping and testing of Internet based experiments.

6. CONCLUSIONS AND FUTURE WORK

viNEX is currently a work in progress system in the sense of being able to create nodes, configure links, and route traffic. The reason viNEX was built to investigate the limits to this approach. We are aware about the potential limits of this aproach and we are in the process to establish them.

To take this research work further, three key challenges have been identified. (1) Network performance; we are not impressed by the bandwidth rates obtained in Experiment 1 and 2 above ($\tilde{3}$ Mbit/s) - the slow performance is mainly attributed to the use of QEMU for device emulation. Future enhancements on the XEN HVM are in the pipeline and this limitation might be eliminated. (2) there is a need to identify the class of network experiments that are suitable to be run on viNEX. During the evaluation phase, we were

able to deploy the standard IP protocols on viNEX without any issues, e.g. the RIPv2 protocol was deployed on the experiment topology nodes without any modification.

Our emulator (viNEX) was developed using open source technologies; with the major technologies being Xen, FreeBSD, and NetBSD. As a result, we experienced a significant amount of the benefits and advantages of open source, such as; (1) the direct access to the primary software authors and experts, (2) the ability to modify the source to meet our custom requirements, Xen kernel was recompiled with setting `vmxenabled=yes` to enable booting of NetBSD guest kernel, (3) the ability to scale to arbitrary instances without artificial licensing constraints, arbitrary number of NetBSD instances can be booted without any licensing restrictions (4) the ability to learn from the availability of source code (5) the potential to bundle, package and distribute without additional licensing transaction costs.

There is also another opportunity to improve this research by enabling the configuration of the testbed to be done using the NS-2 tool. Key integration points will be identified in order assist interested reader to extend this work. NS-2 is a famous tool in the network research space and therefore is makes sense to use NS-2 as the modeling language for viNEX. Network researchers are already familiar with NS-2 and therefore it will make a seamless adoption of viNEX into their space.

7. REFERENCES

[1] R. Balachander and P. Venkataram. User-mode linux based mpls emulator. *TENCON 2004. 2004 IEEE Region 10 Conference*, B:601–604 Vol. 2, Nov. 2004.

[2] Carlo Caini, Rosario Firrincieli, Renzo Davoli, and Daniele Lacamera. Virtual integrated tcp testbed (vitt). In *TridentCom '08: Proceedings of the 4th International Conference on Testbeds and research infrastructures for the development of networks & communities*, pages 1–6, ICST, Brussels, Belgium, Belgium, 2008. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[3] Dan Duchamp and Greg De Angelis. A hypervisor based security testbed. In *DETER: Proceedings of the DETER Community Workshop on Cyber Security Experimentation and Test on DETER Community Workshop on Cyber Security Experimentation and Test 2007*, pages 3–3, Berkeley, CA, USA, 2007. USENIX Association.

[4] Chris Edwards and Aaron Harwood. Using para-virtualization as the basis for a federated planetlab architecture. In *VTDC '06: Proceedings of the 2nd International Workshop on Virtualization Technology in Distributed Computing*, page 13, Washington, DC, USA, 2006. IEEE Computer Society.

[5] N. Egi, A. Greenhalgh, M. Handley, M. Hoerdt, L. Mathy, and T. Schooley. Evaluating xen for router virtualization. *Computer Communications and Networks, 2007. ICCCN 2007. Proceedings of 16th International Conference on*, pages 1256–1261, Aug. 2007.

[6] Emulab. Emulab home. www.emulab.net.

[7] Kevin Fall. Network emulation in the vint/ns simulator. In *ISCC '99: Proceedings of the The Fourth IEEE Symposium on Computers and Communications*, page 244, Washington, DC, USA, 1999. IEEE Computer Society.

[8] D. Fernandez, T. de Miguel, and F. Galan. Study and emulation of ipv6 internet-exchange-based addressing models. *Communications Magazine, IEEE*, 42(1):105–112, Jan 2004.

[9] gnuplot. Gnuplot. http://www.gnuplot.info/.

[10] Mike Hibler, Robert Ricci, Leigh Stoller, Jonathon Duerig, Shashi Guruprasad, Tim Stack, Kirk Webb, and Jay Lepreau. Large-scale virtualization in the emulab network testbed. In *ATC'08: USENIX 2008 Annual Technical Conference on Annual Technical Conference*, pages 113–128, Berkeley, CA, USA, 2008. USENIX Association.

[11] X.W. Huang, R. Sharma, and S. Keshav. The entrapid protocol development environment. *INFOCOM '99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, 3:1107–1115 vol.3, Mar 1999.

[12] Miljenko Mikuc Marko Zec. Operating system support for integrated network emulation in imunes. In *In Proc. of the 1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS), Boston, MA, 2004.*, 2004.

[13] Mukosi Abraham Mukwevho. vinex home. http://sites.google.com/site/mukosi/.

[14] NLANR/DAST. Iperf. http://sourceforge.net/projects/iperf.

[15] NS-2. Ns-2 wiki. http://nsnam.isi.edu/nsnam/index.php/.

[16] Shawn Ostermann. tcptrace. http://www.tcptrace.org/.

[17] PlanetLab. Planetlab. http://www.planet-lab.org.

[18] Robert N. M. Watson Poul-Henning Kamp. Jails: Confining the omnipotent root. In *2nd SANE Conference*, May 2000.

[19] Ian Pratt, Keir Fraser, Steven Hand, Christian Limpach, Andrew Warfield, Dan Magenheimer, Jun Nakajima, and Asit Mallick. Xen 3.0 and the art of virtualization. In *Proceedings of Linux Symposium 2005*, July 2005.

[20] Paul 'Rusty' Russell. Ebtables firewalling. http://ebtables.sourceforge.net/.

[21] Timothy Jason Shepard. xplot. www.xplot.org.

[22] Ralf Spenneberg. Emulating networks using user-mode linux. http://www.samag.com/documents/s=8997/sam0401a/0401a.ht

[23] Craig Leres Van Jacobson and Steven McCanne. tcpdump/libpcap. http://www.tcpdump.org/.

[24] Xen. Xen home. http://www.xen.org/.

[25] Marko Zec. Implementing a clonable network stack in the freebsd kernel. In *In Proceedings of the USENIX 2003 Annual Technical Conference*, pages 137–150, 2003.

Appendix B

viNEX Configuration and Management Scripts

This appendix provides the source code listing of all bash scripts we wrote for the configuration and management utilities of viNEX.

Listing B.1: Listing of bash script — `start-gateway.sh`

```
#!/bin/sh
_guest_name="Gateway"
xm create /xen/freebsd/scripts/gateway.hvm
_domid=`xm list ${_guest_name} | grep ${_guest_name} | awk '{print $2}'`
_vif0="vif"${_domid}".0"
_vif1="vif"${_domid}".1"
_ip0=196.30.225.254
_ip1=192.30.225.252

# Add route for all traffic bound to $ip0 and $ip1
echo "ip route add $_ip0 dev $_vif0"
ip route add $_ip0  dev $_vif0
echo "ip route add $_ip1 dev $_vif1"
ip route add $_ip1  dev $_vif1
ip route add default via $_ip0 dev $_vif0 table xengw
```

Listing B.1: Listing of bash script — `start-gateway.sh`

Listing B.2: Listing of bash script — `start-node.sh`

```
#!/bin/sh

# Script to create and confugre a new NetBSD guest
dir=$(dirname "$0")

. "$dir/common.sh"


# To print usage

usage() {

        echo "Usage: ‘basename $0‘ [guest_id]"
        echo "Options: Required arguments"
        echo "         [node_name] : An integer > 0 used to identify the new
    node, new node will be named \'Guest_[guest_id]\', e.g. Guest-1"
        exit 1
}


# Check arguments
if [ "$#" -ne "1" ]
 then
        usage
        exit 1
fi

_guest_name=$1
_node_id=$(validate_node_name $_guest_name)

exit_status=‘echo $?‘
if [ $exit_status -ne 0 ]
then
        echo $_node_id
        echo "Exit: $exit_status"
        exit $exit_status
fi

echo "Creating a new NetBSD Xen guest named - " $_guest_name

# Create new guest node root file system device
_xen_root="/xen/netbsd"
_guest_home=$_xen_root"/images"
_guest_root_image=$_guest_home"/"$_guest_name"-root.img"
_guest_usr_common=$_guest_home"/"$_guest_name"-usr.img"
_guest_scripts_home=$_xen_root"/scripts"
_guest_xen_script=$_guest_scripts_home"/"$_guest_name".hvm"
_template_root_filesystem=$_guest_home"/template/netbsd-4.0-guest-root.img"
_guest_mount_point=$_guest_home"/mnt"

ln -sf $_guest_home"/template/netbsd-4.0-usr-common-bigger.img"
    $_guest_usr_common


```

```
53 # Create Xen guest script for the new node
54         _kernel="/usr/lib/xen/boot/hvmloader"
55         _device_model="\"/usr/lib64/xen/bin/qemu-dm\""
56         _memory="32"
57         _builder="hvm"
58         _disk="[
    'file:"$_guest_root_image",ioemu:hda,w','file:"$_guest_usr_common",ioemu:hdb,r
    ]"
59         _boot="c"
60         _guest_vif="[ 'type=ioemu,model=rtl8139', 'type=ioemu,model=rtl8139',
    'type=ioemu,model=rtl8139', 'type=ioemu,model=rtl8139',
    'type=ioemu,model=rtl8139' ]"
61         _sdl="0"
62         _vnc="1"
63         _vncclient="1"
64         _vncdisplay="1"
65         _vncconsole="1"
66         _vcpus="1"
67         _serial="\"pty\""
68         _on_reboot="'restart'"
69         _on_crash="'restart'"
70
71         echo "kernel = \""$_kernel"\""                          >
    $_guest_xen_script
72         echo "name = \""$_guest_name"\""               >> $_guest_xen_script
73         echo "builder = \""$_builder"\""                        >>
    $_guest_xen_script
74         echo "memory = " $_memory                      >> $_guest_xen_script
75         echo "disk = " $_disk                                       >>
    $_guest_xen_script
76         echo "boot = \""$_boot"\""                              >>
    $_guest_xen_script
77         echo "vif = "$_guest_vif                                    >>
    $_guest_xen_script
78         echo "device_model = "$_device_model    >> $_guest_xen_script
79         echo "sdl = "$_sdl                                          >>
    $_guest_xen_script
80         echo "vnc = "$_vnc                                          >>
    $_guest_xen_script
81         echo "vncclient = "$_vncclient                 >> $_guest_xen_script
82         echo "vncdisplay = "$_vncdisplay                        >>
    $_guest_xen_script
83         echo "vncconsole = "$_vncconsole               >> $_guest_xen_script
84
85         echo "apic = \"1\""                                         >>
    $_guest_xen_script
86         echo "vcpus =\""$_vcpus"\""                             >>
    $_guest_xen_script
87         # echo "cpus =\""0-1"\""                                    >>
    $_guest_xen_script
88         echo "serial = "$_serial                                    >>
    $_guest_xen_script
89         echo "on_reboot = "$_on_reboot                 >> $_guest_xen_script
90         echo "on_crash = "$_on_crash                   >> $_guest_xen_script
91
```

```
92 # Create file device for the root filesystem of the guest node
93         cp  $_template_root_filesystem $_guest_root_image
94
95 # Mount the guest root file system device
96         mount -o loop,offset=32256 $_guest_root_image $_guest_mount_point
97
98
99 # Configure networking by editing the /etc/rc.conf of the guest
100
101        # Create the guest's /etc/rc.conf
102        _guest_etc_rc_conf=$_guest_mount_point"/etc/rc.conf"
103
104        _ip=196.30.225.$_node_id
105
106        echo "#       \$NetBSD: rc.conf,v 1.96 2000/10/14 17:01:29 wiz Exp $"
                                    >    $_guest_etc_rc_conf
107        echo "#"

               >> $_guest_etc_rc_conf
108        echo "# see rc.conf(5) for more information."
                                                        >> $_guest_etc_rc_conf
109        echo "#"

               >> $_guest_etc_rc_conf
110        echo "# Use program=YES to enable program, NO to disable it.
       program_flags are"         >> $_guest_etc_rc_conf
111        echo "# passed to the program on the command line."
                                          >> $_guest_etc_rc_conf
112        echo "#"

               >> $_guest_etc_rc_conf
113        echo "# Load the defaults in from /etc/defaults/rc.conf (if it's
       readable)."                          >> $_guest_etc_rc_conf
114        echo "# These can be overridden below."
                                                        >> $_guest_etc_rc_conf
115        echo "#"

               >> $_guest_etc_rc_conf
116        echo "if [ -r /etc/defaults/rc.conf ]; then"
                                                                  >>
       $_guest_etc_rc_conf
117        echo "        . /etc/defaults/rc.conf"
                                                                       >>
       $_guest_etc_rc_conf
118        echo "fi"

               >> $_guest_etc_rc_conf
119        echo "# If this is not set to YES, the system will drop into single-user
       mode."                 >> $_guest_etc_rc_conf
120        echo "#"

               >> $_guest_etc_rc_conf
121        echo "rtsold=NO"
                                                                            >>
       $_guest_etc_rc_conf
```

```
122         echo "ipv6_enable=NO"
                                                                    >>
    $_guest_etc_rc_conf
123         echo "ipv6=NO"
                                                                        >>
    $_guest_etc_rc_conf
124         echo "auto_ifconfig=YES"
                                                                    >>
    $_guest_etc_rc_conf
125         echo "rc_configured=YES"
                                                                    >>
    $_guest_etc_rc_conf
126         echo "hostname="$_guest_name
                                                            >>
    $_guest_etc_rc_conf
127         echo "domainname=mukosi.com"
                                                    >> $_guest_etc_rc_conf
128         echo "routed=YES"
                                                                        >>
    $_guest_etc_rc_conf
129         echo "routed_flags=\" -t -T /rip_trace.log \""
                                                    >> $_guest_etc_rc_conf
130         echo "ftpd=YES"
                                                                    >>
    $_guest_etc_rc_conf
131         echo "sshd=YES"
                                                                    >>
    $_guest_etc_rc_conf
132         echo "dhclient=NO"
                                                                        >>
    $_guest_etc_rc_conf
133         echo "savecore=NO"
                                                                >>
    $_guest_etc_rc_conf
134         echo "virecover=NO"
                                                                >>
    $_guest_etc_rc_conf
135         echo "motd=NO"
                                                            >>
    $_guest_etc_rc_conf
136         echo "powerd=NO"
                                                                >>
    $_guest_etc_rc_conf
137         echo "postfix=NO"
                                                                        >>
    $_guest_etc_rc_conf
138         echo "flushroutes=YES"
                                                                    >>
    $_guest_etc_rc_conf
139         echo "# Add local overrides below"
                                                            >>
    $_guest_etc_rc_conf
140         echo "#"

            >> $_guest_etc_rc_conf
```

```
141         echo "net_interfaces=\"rtk1 rtk2 rtk3 rtk4 \""
                                                    >> $_guest_etc_rc_conf
142         echo "ifconfig_rtk0=\"$_ip   netmask  255.255.255.0 metric 2 \""
                                  >> $_guest_etc_rc_conf
143         echo "ifconfig_rtk1=\"  mtu 1400 \""
                                                          >>
    $_guest_etc_rc_conf
144         echo "ifconfig_rtk2=\"  mtu 1400 \""
                                                          >>
    $_guest_etc_rc_conf
145         echo "ifconfig_rtk3=\"  mtu 1400 \""
                                                          >>
    $_guest_etc_rc_conf
146         echo "ifconfig_rtk4=\"  mtu 1400 \""
                                                          >>
    $_guest_etc_rc_conf
147         # Unmount the guest file system
148         umount -d $_guest_root_image
149         umount  $_guest_mount_point
150
151         # Boot the guest node
152         echo "Starting guest node - [$_guest_name]"
153         export _CUR_DOMU_IP=$_ip
154
155         xm create $_guest_xen_script
156
157         # Get the XEN DomU ID
158         _domid=‘xm list ${_guest_name} | grep ${_guest_name} | awk ’{print $2}’‘
159         _vif="vif"${_domid}".0"
160         # Add route for all traffic bound to this DomU
161         # Enable proxy ARP for the control network
162         ifconfig $_vif down
163         # echo 1 > /proc/sys/net/ipv4/conf/$vif/proxy_arp
164         ifconfig $_vif up
165         echo "ip route add $_ip dev $_vif"
166         ip route add $_ip dev $_vif
167
168         exit 0
```

Listing B.2: Listing of bash script — `start-node.sh`

Listing B.3: Listing of bash script — `create-link.sh`

```
#!/bin/sh

# Scrtipt to create a link between two nodes

dir=$(dirname "$0")
. "$dir/common.sh"

_help()  {
        echo "Usage:  $0  --link-id identifier --from node --to node [options]"
        echo "Required arguments:"
        echo "          --link-id identifier    : integer specifying a global unique identity of the link."
        echo "          --from node             : string to set the link' source node."
        echo "          --to node               : string to set the link' target node."
        echo "Optional arguments:"
        echo "          --delay NNms            : sets the propagation delay of the link to the value of NN where NN is in "
        echo "                                    milliseconds, default is 0 for no delay."
        echo "          --bw NNunit             : sets NN as the link bandwidth, where unit can be any of bit/s Kbit/s Mbit/s "
        echo "                                    Byte/s KByte/s MByte/s, default is 0 for no bandwiith limitation."
        echo "          --plr X                 : set the propability for random packet loss where X is a floating point "
        echo "                                    number between 0 and 1 which causes packets to be dropped at random, "
        echo "                                    default is 0 for no loss."
}

check_required_argument() {
        if [ -z "$2" ]
        then
                echo "Missing required parameter - [$1]."
                my_error=1
        fi
}

check_optional_default() {
        if [ -z "$1" ]
        then
                echo $2
        else
                echo $1
        fi
}

TEMP=`getopt -o  - --long link-id:,from:,to:,delay:,bw:,plr: --name "$0" -- "$@"`

if [ $? != 0 ] ; then echo "Terminating..." >&2 ; exit 1 ; fi

# Note the quotes around `$TEMP': they are essential!
```

```
eval set -- "$TEMP"

while true;  do
        case "$1" in
                --link-id)      link_id=$2; shift 2;;
                --from)                 source_domain_name=$2 ; shift 2 ;;
                --to)           target_domain_name=$2 ; shift 2;;
                --delay)        link_delay=$2 ; shift 2;;
                --bw)           link_bw=$2 ; shift 2;;
                --plr)          link_plr=$2 ; shift 2;;
                --)                     shift ; break ;;
                *)                      _help ; exit 1 ;;
        esac
done

check_required_argument "--link-id" $link_id
check_required_argument "--from" $source_domain_name
check_required_argument "--to" $target_domain_name

if [ -n "$my_error" ] # Check if an error occured while trying to parse the
    arguments
then
        _help
        exit 1
fi


link_delay=$(check_optional_default "$link_delay"  "0ms")
link_bw=$(check_optional_default "$link_bw" "0Kbit/s")
link_plr=$(check_optional_default "$link_plr" "0")

# Get the Gateway interface names:
_gateway_node="Gateway"
_gateway_id=`xm list ${_gateway_node} | grep ${_gateway_node} | awk '{print $2}'`
source_vlan_parent_if="vif"${_gateway_id}".0"
source_vlan_parent_if_mac=`ifconfig -a | grep ${source_vlan_parent_if} | awk
    '{print $5}'`

target_vlan_parent_if="vif"${_gateway_id}".1"
target_vlan_parent_if_mac=`ifconfig -a | grep ${target_vlan_parent_if} | awk
    '{print $5}'`

source_vlan_bridge="bridge"$link_id"0"
target_vlan_bridge="bridge"$link_id"1"

source_node_id=$(validate_node_name "$source_domain_name")
source_dom_id=`xm list ${source_domain_name} | grep ${source_domain_name} | awk
    '{print $2}'`
target_node_id=$(validate_node_name "$target_domain_name")
target_dom_id=`xm list ${target_domain_name} | grep ${target_domain_name} | awk
    '{print $2}'`

source_vlan_id=$link_id$source_node_id
target_vlan_id=$link_id$target_node_id

```

```
source_gw_vlan_device="vlan"$source_vlan_id
target_gw_vlan_device="vlan"$target_vlan_id
source_gw_vlan_parent_device="re0"
target_gw_vlan_parent_device="re1"

source_cont_ip="196.30.225."$source_node_id
source_link_ip="10.0."$link_id"."$source_node_id

target_cont_ip="196.30.225."$target_node_id
target_link_ip="10.0."$link_id"."$target_node_id

gateway_ip="196.30.225.254"
gateway_link_bridge="bridge"$link_id

ssh_cmd="ssh -q root@$gateway_ip "

# Check if all the nodes involved on the link setup are accessible, up and
    running.
check_node_wait ${_gateway_node}  ${gateway_ip}
check_node_wait ${source_domain_name}  ${source_cont_ip}
check_node_wait ${target_domain_name}    ${target_cont_ip}

# Get Source MAC
ssh_login=" root@$source_cont_ip "
next_if_id=`ssh -q $ssh_login "/sbin/ifconfig -a | grep 8843 | wc -l" |  sed
    's/^ *\(.*\) *$/\1/'`
source_vif="vif"${source_dom_id}"."${next_if_id}
next_if_name="rtk"${next_if_id}
source_mac=`ssh -q $ssh_login "/sbin/ifconfig ${next_if_name} | grep address " |
    awk '{ print $2 }'`

# Get target MAC
ssh_login=" root@$target_cont_ip "
next_if_id=`ssh -q $ssh_login "/sbin/ifconfig -a | grep 8843 | wc -l" |  sed
    's/^ *\(.*\) *$/\1/'`
target_vif="vif"${target_dom_id}"."${next_if_id}
next_if_name="rtk"${next_if_id}
target_mac=`ssh -q $ssh_login "/sbin/ifconfig ${next_if_name} | grep address " |
    awk '{ print $2 }'`


function configure_link_interface() {

        dom_name=$1
        cont_ip=$2
        link_ip=$3
        vlan_ip=$4
        node_id=$5
        vlan_bridge=$6
        vlan_parent_if=$7
        gw_vlan_device=$8
        gw_vlan_parent_device=$9
        vlan_if=$vlan_parent_if"."$vlan_id

        # Run commands to configure links on the source DomU node
```

```
146         ssh_login=" root@$cont_ip "
147 
148         echo "Configure link interface on domain - $cont_ip"
149         # - Get the next available interface on thssh_cmde source domain
150         next_if_id=‘ssh -q $ssh_login "/sbin/ifconfig -a | grep 8843 | wc -l" |
    sed ’s/^ *\(.*\) *$/\1/’‘
151         next_if_name="rtk"${next_if_id}
152 
153         # Configure and bring the DomU interface up - media 100baseTX mediaopt
    full-duplex
154         echo "next_if_name = " $next_if_name
155         ssh -q $ssh_login "/sbin/ifconfig $next_if_name inet $link_ip  netmask
    255.255.255.0 metric 1 up "
156 
157         #1. Create VLAN for this link
158         # - TODO: Check if VLAN exist first
159         echo "vconfig add $vlan_parent_if $vlan_id "
160         vconfig add $vlan_parent_if $vlan_id
161         echo "ifconfig $vlan_if mtu 1496 up"
162         ifconfig $vlan_if mtu 1496 up
163 
164         # Set MAC address of the VLAN interface
165         mac_address=‘python -c ’import random; r=random.randint; print
    "00:16:3E:%02X:%02X:%02X" % (r(0, 0x7f), r(0, 0xff), r(0, 0xff))’‘
166         echo "ip link set dev $vlan_if addresss $mac_address"
167         ifconfig $vlan_if down
168         ip link set dev $vlan_if address $mac_address
169         ip link set $vlan_if up
170 
171         # Create the bridge
172         brctl addbr $vlan_bridge
173         ip link set    $vlan_bridge up
174 
175         ifconfig $vif mtu 1400 up
176         echo "brctl addif $vlan_bridge $vlan_if "
177         brctl addif $vlan_bridge  $vlan_if
178         echo "brctl addif $vlan_bridge $vif"
179         brctl addif $vlan_bridge  $vif
180         # brctl addif $vlan_bridge  $d_vlan_if
181 
182         # Add ebtables rule to force all DomU generated traffic to be routed to
    the FreeBSD gateway first
183         echo "ebtables -t nat -A PREROUTING -i $vif -j dnat --to-destination
    $ebtables_target_mac --dnat-target ACCEPT"
184         ebtables -t nat -A PREROUTING -i $vif -j dnat --to-destination
    $ebtables_target_mac --dnat-target ACCEPT
185 
186         # Configure VLAN on the FreeBSD traffic shaper ...
187         echo ${ssh_cmd} "/sbin/ifconfig $gw_vlan_device create vlan $vlan_id
    vlandev $gw_vlan_parent_device "
188         ${ssh_cmd} "/sbin/ifconfig $gw_vlan_device create vlan $vlan_id vlandev
    $gw_vlan_parent_device "
189         echo  ${ssh_cmd} "/sbin/ifconfig $gw_vlan_device link $mac_address  up"
190         ${ssh_cmd} "/sbin/ifconfig $gw_vlan_device link $mac_address   up"
191 
```

```
192         # Add the VLAN interface to the bridge
193         echo ${ssh_cmd} "/sbin/ifconfig $gateway_link_bridge  addm
        $gw_vlan_device "
194         ${ssh_cmd} "/sbin/ifconfig $gateway_link_bridge  addm  $gw_vlan_device
        up "
195
196 }
197
198 # Create gateway linking bridge ...
199 echo "${ssh_cmd} \"/sbin/ifconfig $gateway_link_bridge create \""
200 ${ssh_cmd}  "/sbin/ifconfig $gateway_link_bridge create  "
201
202 src_vlan_parent_if_mac=$source_vlan_parent_if_mac
203 tgt_vlan_parent_if_mac=$target_vlan_parent_if_mac
204 ebtables_target_mac=$target_mac
205 remote_host_ip="$target_link_ip"
206 vlan_id=$link_id
207 vif=$source_vif
208 tgt_gw_vlan_device="$target_gw_vlan_device"
209 configure_link_interface "$source_domain_name" "$source_cont_ip"
        "$source_link_ip" "$source_vlan_ip" "$source_node_id" "$source_vlan_bridge"
        "$source_vlan_parent_if" "$source_gw_vlan_device"
        "$source_gw_vlan_parent_device" "$source_vlan_parent_if_mac"
210 src_vlan_parent_if_mac=$target_vlan_parent_if_mac
211 tgt_vlan_parent_if_mac=$source_vlan_parent_if_mac
212 ebtables_target_mac=$source_mac
213 remote_host_ip="$source_link_ip"
214 vif=$target_vif
215 tgt_gw_vlan_device="$source_gw_vlan_device"
216 configure_link_interface "$target_domain_name" "$target_cont_ip"
        "$target_link_ip" "$target_vlan_ip" "$target_node_id" "$target_vlan_bridge"
        "$target_vlan_parent_if" "$target_gw_vlan_device"
        "$target_gw_vlan_parent_device" "$target_vlan_parent_if_mac"
217
218 # Add Gateway IPFW rules
219 ${ssh_cmd}  "ipfw add pipe ${link_id} ip4 from any to any via
        ${source_gw_vlan_device} layer2"
220 ${ssh_cmd}  "ipfw add pipe ${link_id} ip4 from any to any via
        ${target_gw_vlan_device} layer2"
221 ${ssh_cmd}  "ipfw pipe ${link_id} config bw ${link_bw} delay ${link_delay} plr
        ${link_plr}"
```

Listing B.3: Listing of bash script — `create-link.sh`

Listing B.4: Listing of bash script — `modify-link.sh`

```
#!/bin/sh

# Scrtipt to create a link between two nodes

dir=$(dirname "$0")
. "$dir/common.sh"

_help()  {
        echo "Usage:  $0  --link-id identifier [options]"
        echo "Required arguments:"
        echo "          --link-id identifier    : integer specifying a global unique identity of the link."
        echo "Optional arguments:"
        echo "          --delay NNms            : sets the propagation delay of the link to the value of NN where NN is in "
        echo "                                    milliseconds, default is 0 for no delay."
        echo "          --bw NNunit             : sets NN as the link bandwidth, where unit can be any of bit/s Kbit/s Mbit/s "
        echo "                                    Byte/s KByte/s MByte/s, default is 0 for no bandwiith limitation."
        echo "          --plr X                 : set the propability for random packet loss where X is a floating point "
        echo "                                    number between 0 and 1 which causes packets to be dropped at random, "
        echo "                                    default is 0 for no loss."
}

check_required_argument() {
        if [ -z "$2" ]
        then
                echo "Missing required parameter - [$1]."
                my_error=1
        fi
}

check_optional_default() {
        if [ -z "$1" ]
        then
                echo $2
        else
                echo $1
        fi
}

TEMP=`getopt -o  - --long link-id:,delay:,bw:,plr: --name "$0" -- "$@"`

if [ $? != 0 ] ; then echo "Terminating..." >&2 ; exit 1 ; fi

# Note the quotes around `$TEMP': they are essential!
eval set -- "$TEMP"

while true;  do
        case "$1" in
```

```
48                --link-id)      link_id=$2; shift 2;;
49                --delay)        link_delay=$2 ; shift 2;;
50                --bw)           link_bw=$2 ; shift 2;;
51                --plr)          link_plr=$2 ; shift 2;;
52                --)                     shift ; break ;;
53                *)                      _help ; exit 1 ;;
54        esac
55 done
56
57 check_required_argument "--link-id" $link_id
58
59 if [ -n "$my_error" ] # Check if an error occured while trying to parse the
      arguments
60 then
61         _help
62         exit 1
63 fi
64
65
66 link_delay=$(check_optional_default "$link_delay"  "0ms")
67 link_bw=$(check_optional_default "$link_bw" "0Kbit/s")
68 link_plr=$(check_optional_default "$link_plr" "0")
69
70 # Get the Gateway interface names:
71 gateway_ip="196.30.225.254"
72 ssh_cmd="ssh -q root@$gateway_ip "
73
74 # Configure link attributes on the Gateway
75 ${ssh_cmd}  "ipfw pipe ${link_id} config bw ${link_bw} delay ${link_delay} plr
      ${link_plr}"
```

Listing B.4: Listing of bash script — `modify-link.sh`

LISTING B.5: Listing of bash script — `common.sh`

```
#!/bin/sh

# This script contains a list of common utility functions that are reused by other viNEX scripts

function validate_node_name() {

        node_name=$1
        _node_name=`echo $node_name | grep -o -P "^Node-[0-9]+$"`
        node_id=`echo $node_name | grep -o -P "[0-9]+"`
        if [ "$_node_name" == "" ] || [ "$node_id" -lt 1 ]
        then
                echo "ERROR: Invalid node name specified ($node_name), node name must be in the format - NodeX where X is an integer number > 0, e.g. \"Node-1\""
                exit 65
        fi
        echo "$node_id"
}

# To check if a node is reachable using SSH, otherwise we wait until the node is up/reachable ...
function check_node_wait () {
        node_name=$1
        node_ip=$2
        echo "Checking if node (${node_name}, ${node_ip}) is reachable ..."
        cmd="ssh root@${node_ip} exit"
        error_code="1"
        while [ $error_code != 0 ]
        do
                ${cmd}
                error_code=`echo $?`
                if [ $error_code != 0 ]
                then
                        echo "Node (${node_name}, ${node_ip}) is not reachable, sleeping for 5 seconds before trying again ...."
                        sleep 5
                else
                        echo "Node (${node_name}, ${node_ip}) is reachable, done ..."
                fi
        done
}
```

LISTING B.5: Listing of bash script — `common.sh`

Appendix C

The TCP/IP Reference Model

The TCP/IP model was created in 1969 by DARPA (United States Defense Advanced Research Projects Agency) (Kozierok [46]). The resulting network was known as the Arpanet, with the first matured version of the TCP released in 1973 and formally documented in RFC 675. The TCP/IP model consists of only 4 layers and they do not map directly onto the OSI model. Some of the TCP/IP layers implement functionality of more than one of the OSI layers. Figure C.1 presents the TCP/IP reference model with its four layers.

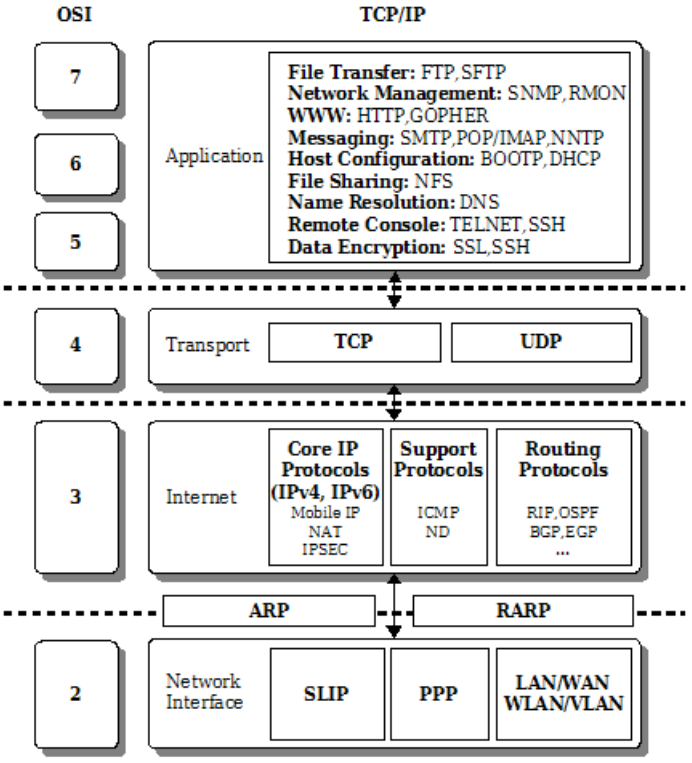


FIGURE C.1: The 4-Layered TCP/IP Protocol Stack (Kozierok [46])

The layers of the TCP/IP model are Network Interface, Internet, Transport and Application. The TCP/IP model is seen as a protocol suite rather than a reference model (Meyer and Zobrist [54]). Its layers were defined after the initial protocols have been created. Each of the four layers is used to describe one or more layers of the OSI model from layer 2 upwards. The TCP model does not describe layer 1 (the physical layer) of the OSI model, therefore it is not concerned with the underlying details of how data is transmitted over a cable.

We now briefly describe each of the four layers of the TCP/IP model as well as their mapping onto the OSI model.

Network Interface Layer: The TCP/IP architecture does not define the functionality required at the physical level for transmitting data. The Network Interface layer is concerned with defining the interfaces that make TCP/IP run on the underlying physical networks. TCP/IP can run on top of a number of different networks (this is also known as *interworking*). The Ethernet, ATM and FDDI (Fiber Distributed Data Interface) are examples of networks which can carry TCP/IP traffic. The interworking is described in a number of standards (RFCs) which

define the encapsulation of IP datagrams over Ethernet frames. Some of the well known RFCs in this space are:

- **RFC 793** of 1984 - Transmission Control Protocol (Postel [64]).
- **RFC 894** of 1984 - Standard for the Transmission of IP Datagrams over Ethernet Networks (Hornig [37]).
- **RFC 948** of 1985 - The two methods for transmitting IP datagrams over IEEE 802.3 networks (Winston [86]).
- **RFC 1010** of 1987 - Assigned numbers, now superseded by RFC 3232 of 2002 (Reynolds and Postel [68]).
- **RFC 1042** of 1988 - Standard for the Transmission of IP Datagrams over Ethernet 802.3 Networks (Postel and Reynolds [65]).
- **RFC 2464** of 1998 - Describes the transmission of IPv6 over Ethernet Networks (Crawford [15]).

Internet Layer: This layer contains the protocols that define the internetwork. It hides the details of the underlying network by creating a virtual network. The core of the IP protocol is implemented at this layer, together with its supporting protocols (such as ICMP) and the routing protocols (e.g. BGP, EGP, RIP, ARP, RARP etc.) as depicted in Figure C.1.

Transport Layer: The protocols in this layer are responsible for the delivery of data between the communicating protocols. The services provided in this layer are equivalent to those described in layer 4 of the OSI model. These protocols rely on the functionality of the IP protocol. Key services provided at this layer include congestion control, reliable data delivery, flow control and the ability to recover or deal with duplicated or missing data. The commonly used protocols at this layer are TCP and UDP, depending on the requirements for the applications running on top of this layer.

Application Layer: This layer contains application protocols that further encapsulate the underlying network from the user. These protocols are responsible for presenting the network functionality to the user in a usable format. Examples include the HTTP, FTP and SMTP protocols. These protocols are normally made available to the application developers in the form of an API.

Appendix D

Modelling standard network topology structures in viNEX

The ViNEX network configuration and control scripts may be used to model most of the standard network structures. Figure D.1 through to Figure D.4 show these network structures as identified in Chapter 15 of (Silberschatz et al. [75]). The corresponding viNEX configuration scripts for modelling these structures on viNEX are shown in Listing D.1 through to Listing D.4 below.

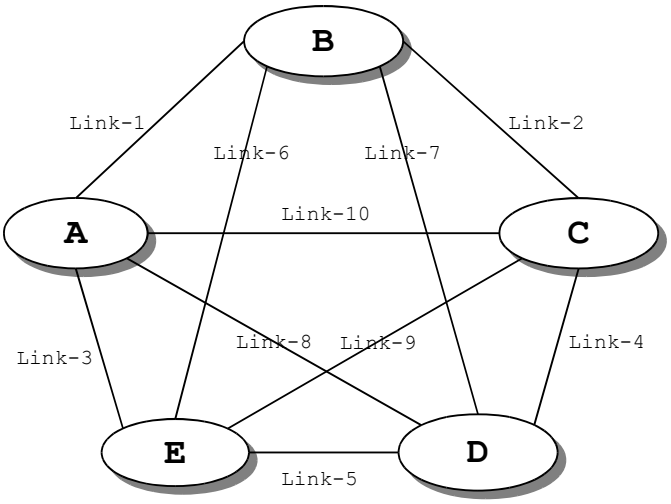


FIGURE D.1: Fully connected network structure

```
1  # Start Gateway node
2  /vinex/scripts/start-gateway.sh
3
4  # Create network nodes
5  /vinex/scripts/start-node.sh A
6  /vinex/scripts/start-node.sh B
7  /vinex/scripts/start-node.sh C
8  /vinex/scripts/start-node.sh D
9  /vinex/scripts/start-node.sh E
10
11 # Create network links without bandwidth limitation properties
12 /vinex/scripts/create-link.sh --link-id 1  --from A     --to B
13 /vinex/scripts/create-link.sh --link-id 2  --from B     --to C
14 /vinex/scripts/create-link.sh --link-id 3  --from A     --to E
15 /vinex/scripts/create-link.sh --link-id 4  --from C     --to D
16 /vinex/scripts/create-link.sh --link-id 5  --from E     --to D
17 /vinex/scripts/create-link.sh --link-id 6  --from B     --to E
18 /vinex/scripts/create-link.sh --link-id 7  --from B     --to D
19 /vinex/scripts/create-link.sh --link-id 8  --from A     --to D
20 /vinex/scripts/create-link.sh --link-id 9  --from C     --to E
21 /vinex/scripts/create-link.sh --link-id 10 --from A     --to C
```

LISTING D.1: viNEX script to create the fully-connected network in Figure D.1

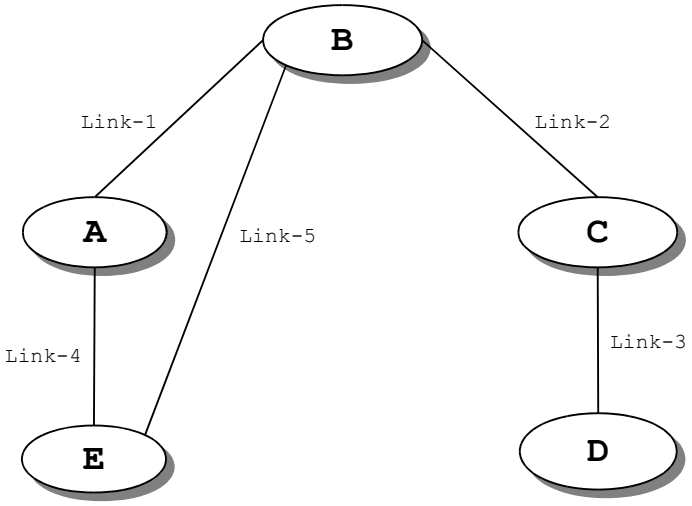


FIGURE D.2: Partially connected network structure

```
1  # Start Gateway node
2  /vinex/scripts/start-gateway.sh
3
4  # Create network nodes
5  /vinex/scripts/start-node.sh A
6  /vinex/scripts/start-node.sh B
7  /vinex/scripts/start-node.sh C
8  /vinex/scripts/start-node.sh D
9  /vinex/scripts/start-node.sh E
10
11 # Create network links without bandwidth limitation properties
12 /vinex/scripts/create-link.sh --link-id 1 --from A      --to B
13 /vinex/scripts/create-link.sh --link-id 2 --from B      --to C
14 /vinex/scripts/create-link.sh --link-id 3 --from C      --to D
15 /vinex/scripts/create-link.sh --link-id 4 --from A      --to E
16 /vinex/scripts/create-link.sh --link-id 5 --from B      --to E
```

LISTING D.2: viNEX script to create the partially-connected network in Figure D.2

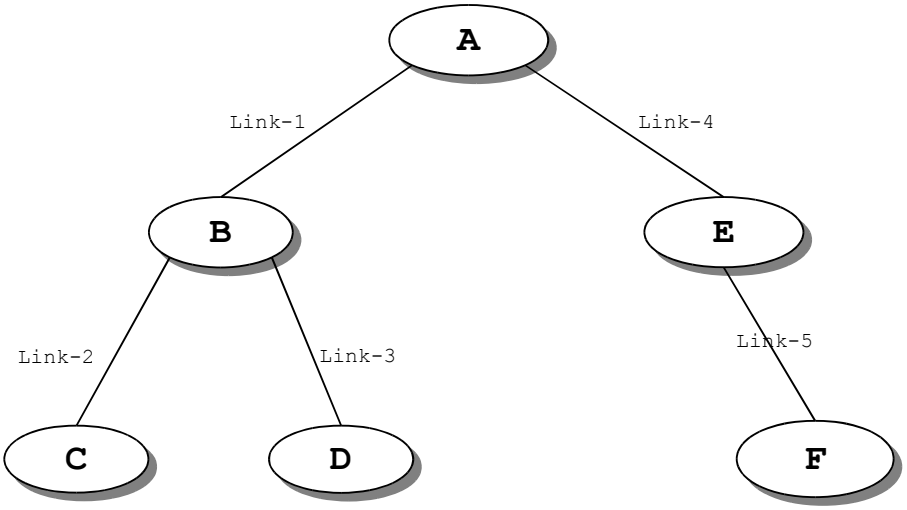


FIGURE D.3: A tree-structured network

```
# Start Gateway node
/vinex/scripts/start-gateway.sh

# Create network nodes
/vinex/scripts/start-node.sh A
/vinex/scripts/start-node.sh B
/vinex/scripts/start-node.sh C
/vinex/scripts/start-node.sh D
/vinex/scripts/start-node.sh E
/vinex/scripts/start-node.sh F

# Create network links without bandwidth limitation properties
/vinex/scripts/create-link.sh --link-id 1 --from A      --to B
/vinex/scripts/create-link.sh --link-id 2 --from B      --to C
/vinex/scripts/create-link.sh --link-id 3 --from B      --to D
/vinex/scripts/create-link.sh --link-id 4 --from A      --to E
/vinex/scripts/create-link.sh --link-id 5 --from E      --to F
```

LISTING D.3: viNEX script to create the tree-structured network in Figure D.3

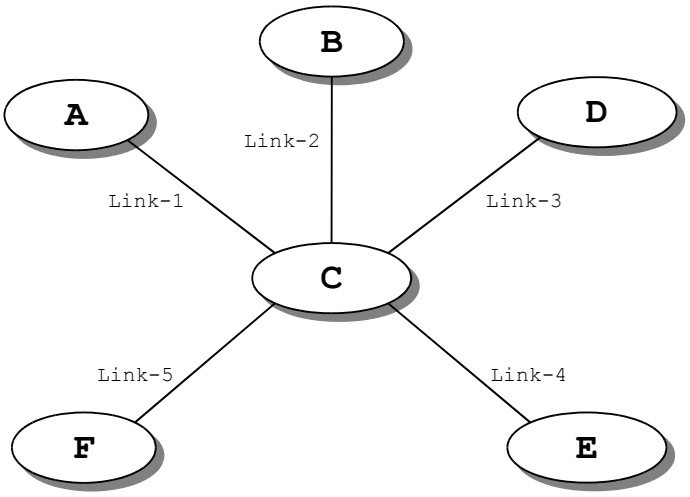


FIGURE D.4: Star network

```
# Start Gateway node
/vinex/scripts/start-gateway.sh

# Create network nodes
/vinex/scripts/start-node.sh A
/vinex/scripts/start-node.sh B
/vinex/scripts/start-node.sh C
/vinex/scripts/start-node.sh D
/vinex/scripts/start-node.sh E
/vinex/scripts/start-node.sh F

# Create network links without bandwidth limitation properties
/vinex/scripts/create-link.sh --link-id 1 --from C      --to A
/vinex/scripts/create-link.sh --link-id 2 --from C      --to B
/vinex/scripts/create-link.sh --link-id 3 --from C      --to D
/vinex/scripts/create-link.sh --link-id 4 --from C      --to E
/vinex/scripts/create-link.sh --link-id 5 --from C      --to F
```

LISTING D.4: viNEX script to create the star-structured network in Figure D.4

Appendix E

Repeating the Van Jacobson Experiment using NS-2

This appendix describes a repeat of the Van Jacobson experiment using the NS-2 simulator. During the development of viNEX, we did not have access to a real physical network to benchmark the functionality of viNEX against. Consequently, the NS-2 simulator was chosen to be used for comparing the results obtained in viNEX against those originally obtained by Van Jacobson [81].

E.1 Experiment configuration

The network topology for this experiment was configured to represent the original topology as described in Van Jacobson [81]. This topology is shown in Figure E.1.

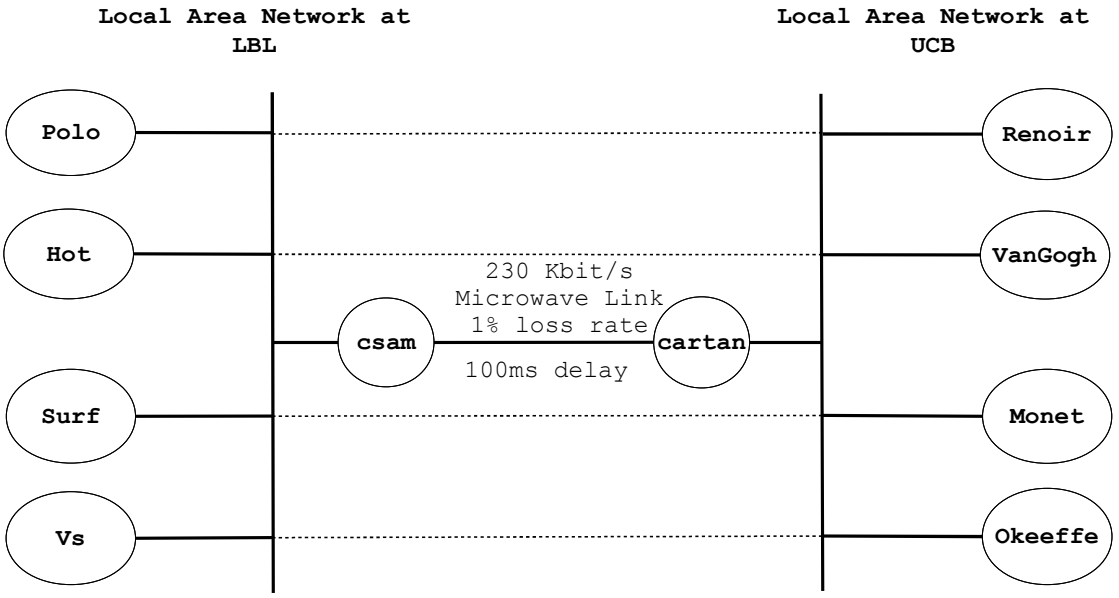


FIGURE E.1: The Van Jacobson Experiment topology used for NS-2 simulations (redrawn from Van Jacobson [81])

The topology in Figure E.1 consists two networks, namely the LBL (Lawrence Berkeley National Laboratory) and the UCB (University of California Berkeley) local area networks separated by 365.76 metres and connected by a microwave link. Each network contains a total of four hosts and one router which is used as a gateway between the two networks.

At LBL, the four hosts are named `Polo`, `Hot`, `Surf` and `Vs` and the gateway router is called `csam`. The UCB nodes are `Renoir`, `VanGogh`, `Monet` and `Okeeffe` and the gateway router is `cartan`. The microwave link between the two environments has a bandwidth capacity of 230Kbit/s, a delay of 100ms and a loss rate of 1%. Each machine at LBL was paired with a machine at UCB to form a total of four network conversation pairs. Network traffic is generated by the machines at LBL over the microwave link to the receiving machines at UCB. The NS-2 script for constructing the simulated topology of Figure E.1 appears in Listing E.1.

The experiment is started by executing command "`ns vanjacobson.tcl`" from the shell. The FTP traffic senders are started at three seconds intervals and each FTP stream is allowed to send up to 1 000 000 bytes (approximately 1MByte) over the link. All packet traces are captured into one file called `out.tr`. The experiment is run for up to 200 seconds using Ns-2 simulated time.

The operation of the NS-2 script in Listing E.1 is:

1. Network nodes are instantiated and created by line 9 through to line 22 of the script using the `set node-name [$ns node]` command.

2. The two local area networks (LANs) at LBL and UCB are created by the call to the `make-lan` function at lines 26 and 27 respectively.

3. In NS-2, the network protocol on a sending node is defined by an Agent class. The Agent definition is also used to specify the version of the protocol used. The TCP/Reno protocol is implemented by the Agent/TCP/Reno class. Lines 36 to 39 of the script show the definition of the TCP/Reno agents. The agents are then attached to their sending nodes by lines 41 to 44 through the `attach-agent` command.

4. NS-2 models the operation of a TCP traffic receiver using the TCPSink agent. TCPSink is responsible for implementing the protocol-specific functionality for receiving traffic such as packet reordering and acknowledgment generation. The TCP sinks are created by lines 46 to 49 of the script. The agents are then attached to their respective receivers at lines 51 to 54 of the script using the `attach-agent` command.

5. The actual network traffic is generated by making use of the Application/FTP classes. The Application/FTP class implements the standard FTP application layer protocol as defined in lines 56 to 59 of the script. The applications are thereafter attached to their respective agents at lines 61 through to 64.

6. The actual connection (pairing) between the sending and receiving nodes is achieved by connecting the Agents and Sinks. This is done in lines 66 to 69.

7. The bottleneck link is defined in line 71. The link bandwidth is configured to 230Kbit/s, the delay is set to 100ms and the queuing model is set to DropTail. Lines 72 and 73 define a limit on the size of the DropTail queue of the bottleneck link to 50 packets. Lines 75 to 85 are used to create the error model for simulating the packet loss rate along the microwave bottleneck. The loss rate is set to 1%.

8. Packet tracing is enabled through lines 29 and 30 of the script by using the `trace-all` command. The packet traces are all collected into a single trace file called `out.tr`.

9. Finally, the FTP transfers of 1 MB (which is roughly 1 000 000 bytes) are started at three seconds apart. This is achieved by the `$ns at` command shown in lines 87 to 93. The experiment is ended by the `$ns at 200.0 "finish"` command at line 92.

```
1
2  set ns [new Simulator]
3
4  LanRouter set debug_ false
5
6  # Initialize variables
7  set packetSize 472
8
9  # Create the nodes at LBL
10 set Polo [$ns node]
11 set Hot  [$ns node]
12 set Surf [$ns node]
13 set Vs   [$ns node]
14 set csam [$ns node]
15 set nodelistLBL "$Polo $Hot $Surf $Vs $csam"
16
17 # Create the nodes at LBL
18 set cartan    [$ns node]
19 set Renoir    [$ns node]
20 set VanGogh   [$ns node]
21 set Monet     [$ns node]
22 set Okeeffe   [$ns node]
23
24 set nodelistUCB "$cartan $Renoir $VanGogh $Monet $Okeeffe"
25
26 $ns make-lan $nodelistLBL 10Mb 1ms LL Queue/DropTail Mac/802_3
27 $ns make-lan $nodelistUCB 10Mb 1ms LL Queue/DropTail Mac/802_3
28
29 set f [open out.tr w]
30 $ns trace-all $f
31 set nf [open out.nam w]
32 $ns namtrace-all $nf
33
34 Agent/TCP set packetSize_ $packetSize
35
36 set tcp0 [new Agent/TCP/Reno]
37 set tcp1 [new Agent/TCP/Reno]
38 set tcp2 [new Agent/TCP/Reno]
39 set tcp3 [new Agent/TCP/Reno]
40
41 $ns attach-agent $Polo $tcp0
42 $ns attach-agent $Hot  $tcp1
43 $ns attach-agent $Surf $tcp2
44 $ns attach-agent $Vs   $tcp3
45
46 set sink0 [new Agent/TCPSink]
47 set sink1 [new Agent/TCPSink]
48 set sink2 [new Agent/TCPSink]
```

```
set sink3 [new Agent/TCPSink]

$ns attach-agent $Renoir $sink0
$ns attach-agent $VanGogh $sink1
$ns attach-agent $Monet $sink2
$ns attach-agent $Okeeffe $sink3

set ftp0 [new Application/FTP]
set ftp1 [new Application/FTP]
set ftp2 [new Application/FTP]
set ftp3 [new Application/FTP]

$ftp0 attach-agent $tcp0
$ftp1 attach-agent $tcp1
$ftp2 attach-agent $tcp2
$ftp3 attach-agent $tcp3

$ns connect $tcp0 $sink0
$ns connect $tcp1 $sink1
$ns connect $tcp2 $sink2
$ns connect $tcp3 $sink3

$ns duplex-link $csam $cartan 230Kb 100ms DropTail
set linkQueue [[$ns link $csam $cartan] queue]
$linkQueue set limit_ 50

# Introduce link loss rate ...
set errorModel [new ErrorModel]
$ns link-lossmodel $errorModel $csam $cartan
$errorModel set rate_ 0.01
$errorModel set enable_ 1
set rv [new RandomVariable/Uniform]
set rng [new RNG]
$rng seed 2001
$rv use-rng $rng
$errorModel ranvar $rv
$errorModel unit pkt

# $ns queue-limit $csam $cartan 50 - 1048576
$ns at 0.0 "$ftp0 send 1000000"
$ns at 3.0 "$ftp1 send 1000000"
$ns at 6.0 "$ftp2 send 1000000"
$ns at 9.0 "$ftp3 send 1000000"
$ns at 200.0 "finish"
proc finish {} {
        global ns f nf
        $ns flush-trace
        close $f
        close $nf
        exit 0
}
$ns run
```

Listing E.1: NS-2 script for creating the Van Jacobson topology in Figure E.1

The following section provides an analysis of the results obtained from the above NS-2 experiment.

E.2 Results

The analysis is based on the trace data recorded by NS-2 into the `out.tr` file. On the sending side, a total of 8566 packets were transmitted and 81 packets were found to be retransmissions. In other words, approximately 1% of all packets were retransmitted. The behaviour of the four TCP Reno senders is plotted in Figure E.2. The entire transmission of packets from the four nodes lasted approximately 155 seconds. All conversations received a fair amount of the bandwidth resulting in very little noticeable dropping in the sending rate of the senders (see Figure E.2).

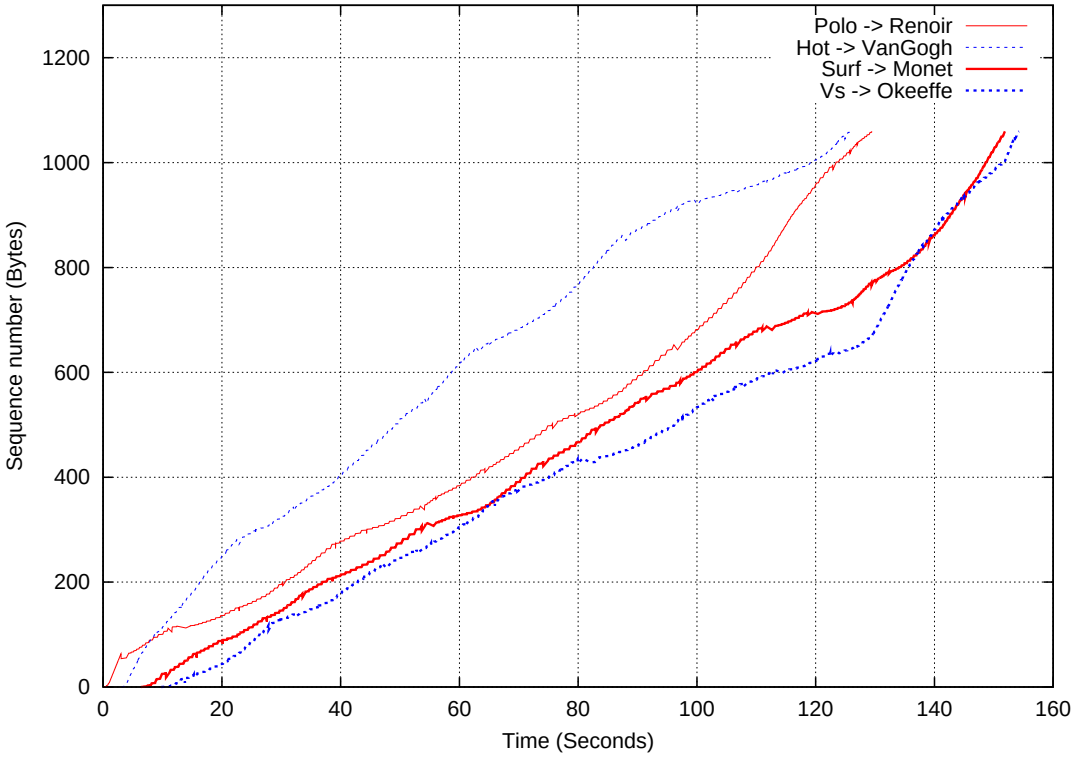


FIGURE E.2: Analysis of multiple, simultaneous TCP/Reno senders using NS-2 with a bottleneck link running at 230Kbit/s

The total throughput which comprises the sum of all throughputs achieved by the four conversations peaked at 27.5 Kbit/s and is displayed in Figure E.3. The

original Van Jacobson experiment measured a total throughput of 25 Kbit/s (two conversations transmitted at 8 Kbit/s each and the other two conversations each reached 4.5 Kbit/s).

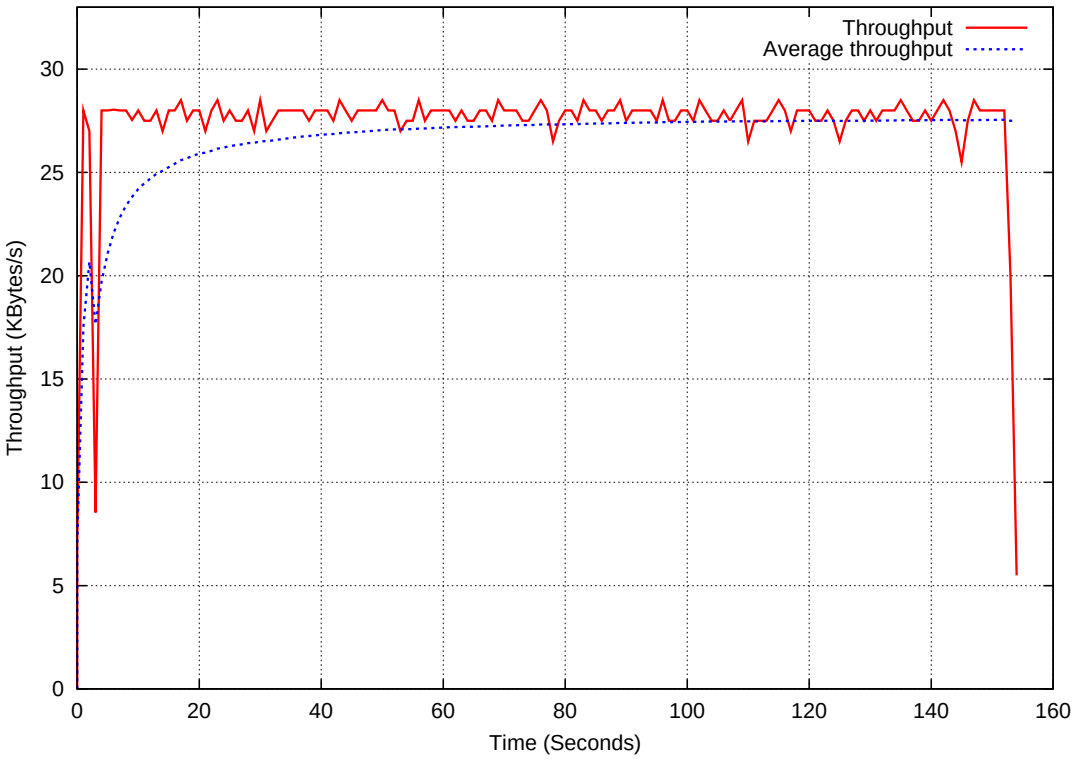


FIGURE E.3: The throughput received by the four TCP/Reno conversations as depicted in Figure E.1

Appendix F

viNEX Scripts Used For Conducting The Van Jacobson Experiment

This appendix provides the source code of the shell scripts we wrote for conducting the Van Jacobson Experiment on viNEX.

```
./ftp-send.sh Polo Renoir &
sleep 3
./ftp-send.sh Hot  VanGogh &
sleep 3
./ftp-send.sh Surf Monet &
sleep 3
./ftp-send.sh Vs   Okeeffe &
sleep 3
```

LISTING F.1: Listing of the bash script (`send.sh`) used to initiate the simultaneous conversations on viNEX 3 seconds apart

```
# Generate and send traffic ...

source_node_name=$1
target_node_name=$2

source_node_top_ip=`cat /vinex/config/topology-net-ip.cfg | grep
    $source_node_name | awk '{print $2}'`
source_node_cnt_ip=`cat /vinex/config/control-net-ip.cfg | grep
    $source_node_name | awk '{print $2}'`
target_node_top_ip=`cat /vinex/config/topology-net-ip.cfg | grep
    $target_node_name | awk '{print $2}'`

echo "Sending data using FTP from [$source_node_name - $source_node_top_ip] to
    [$target_node_name - $target_node_top_ip]"

# - Send 1MB file from Node-1 to Node-2
tracefile_bin="./results/$source_node_name-$target_node_name"".dump"
tracefile_txt="./results/$source_node_name-$target_node_name"".txt"
tracefile_bin1="./results/$source_node_name-$target_node_name"".tput.dump"
tracefile_txt1="./results/$source_node_name-$target_node_name"".tput.txt"
if [ -e $tracefile_bin ]
then
        rm -f $tracefile_bin
        rm -f $tracefile_bin1
fi

nohup tcpdump host $source_node_top_ip and $target_node_top_ip -i vif`xm domid
    $source_node_name`.1 -w $tracefile_bin  > /dev/null &
tcpdump_pid0=$!

nohup tcpdump host $source_node_top_ip and $target_node_top_ip -i vif`xm domid
    $target_node_name`.1 -w $tracefile_bin1 > /dev/null &
tcpdump_pid1=$!

ssh -q root@$source_node_cnt_ip  "/vinex/scripts/ftp-send.sh $target_node_top_ip"
kill -2 $tcpdump_pid0
kill $tcpdump_pid1

# Convert the trace file to ASCII
tcpdump host $source_node_top_ip and $target_node_top_ip -i vif`xm domid
    $source_node_name`.1 -r $tracefile_bin  > $tracefile_txt
tcpdump host $source_node_top_ip and $target_node_top_ip -i vif`xm domid
    $target_node_name`.1 -r $tracefile_bin1  > $tracefile_txt1
echo "FTP file transfer from [$source_node_name - $source_node_top_ip] to
    [$target_node_name - $target_node_top_ip] is COMPLETE ..."
exit 0
```

LISTING F.2: Bash script (`ftp-send.sh`) which is invoked by the send.sh script in Listing F.1 to perform the actual FTP transfer of the 1MB file

Appendix G

List of changes done to `iperf`

This appendix lists the source code changes done to `iperf` version `2.0.2` for NetBSD to enable the measurement and reporting of the one-way-delay of network traffic. The measured one-way-delay is used to calculate the bandwidth-delay product. These changes were generated using the `diff` command on NetBSD. Changes can be applied by using the `patch` command.

```
252c252
< Datagrams\n";
---
> Datagrams\tTransit-Time\n";
255c255
< "[%3d] %4.1f-%4.1f sec  %ss  %ss/sec  %5.3f ms %4d/%5d (%.2g%%)\n";
---
> "[%3d] %4.1f-%4.1f sec  %ss  %ss/sec  %5.3f ms %4d/%5d (%.2g%%)\t%.3f\n";
```

Listing G.1: Listing of the changes done to `Locale.c` source file.

```
244a245,248
>
>           data->info.countTransit = 0;
>           data->info.totalTransit = 0;
>
699c703,708
<
---
>           if( transit < 0)
>               data->info.totalTransit -= transit;
>           else
>               data->info.totalTransit += transit;
>           data->info.countTransit++;
>
```

Listing G.2: Listing of the changes done to `Reporter.c` source file.

```
252c252
< Datagrams\n";
---
> Datagrams\tTransit-Time\n";
255c255
< "[%3d] %4.1f-%4.1f sec  %ss  %ss/sec  %5.3f ms %4d/%5d (%.2g%%)\n";
---
> "[%3d] %4.1f-%4.1f sec  %ss  %ss/sec  %5.3f ms %4d/%5d (%.2g%%)\t%.3f\n";
```

Listing G.3: Listing of the changes done to `ReportDefault.c` source file.

Bibliography

[1] AMD. Amd virtualization. www.amd.com/virtualization. *Last visited on: 3rd November, 2010.*

[2] David Andersen, Nick Feamster, and James Moss. The ron/iris testbed. http://www.datapository.net/tb/. *Last visited on: 3rd November, 2010.*

[3] D.S. Anderson, M. Hibler, L. Stoller, T. Stack, and J. Lepreau. Automatic online validation of network configuration in the emulab network testbed. In *Autonomic Computing, 2006. ICAC '06. IEEE International Conference on*, pages 134–142, June 2006.

[4] Ugen J. S. Antsilevich, Poul-Henning Kamp, Alex Nash, Archie Cobbs, and Luigi Rizzo. Ipfw. http://www.freebsd.org/cgi/man.cgi?query=ipfw&sektion=8/. *Last visited on: 3rd November, 2010.*

[5] R. Balachander and P. Venkataram. User-mode linux based mpls emulator. *TENCON 2004. 2004 IEEE Region 10 Conference*, B:601–604 Vol. 2, Nov. 2004. doi: 10.1109/TENCON.2004.1414667.

[6] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM. ISBN 1-58113-757-5. doi: http://doi.acm.org/10.1145/945445.945462.

[7] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM. ISBN 1-58113-757-5. doi: http://doi.acm.org/10.1145/945445.945462.

[8] Lennert Buytenhek. Linux bridging (net:bridge). http://www.linuxfoundation.org/en/Net:Bridge. *Last visited on: 3rd November, 2010.*

[9] Carlo Caini, Rosario Firrincieli, Renzo Davoli, and Daniele Lacamera. Virtual integrated tcp testbed (vitt). In *TridentCom '08: Proceedings of the 4th International Conference on Testbeds and research infrastructures for the development of networks & communities*, pages 1–6, ICST, Brussels, Belgium, Belgium, 2008. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering). ISBN 978-963-9799-24-0.

[10] David Chisnall. *The definitive guide to the xen hypervisor.* Prentice Hall Press, Upper Saddle River, NJ, USA, 2007. ISBN 9780132349710.

[11] Jin-Hee Choi and Chuck Yoo. Analytic end-to-end estimation for the one-way delay and its variation. In *Consumer Communications and Networking Conference, 2005. CCNC. 2005 Second IEEE*, pages 527–532, Jan. 2005. doi: 10.1109/CCNC.2005.1405228.

[12] D. Comer. *Internetworking with TCP/IP: Principles, protocols, and architecture.* Internetworking with TCP/IP. Prentice Hall, 2000. ISBN 9780130183804.

[13] CONET. Conet (cooperating objects). http://www.cooperating-objects.eu. *Last visited on: 3rd November, 2010.*

[14] Connway. Funding a revolution: Government support for computing research (1999). http://www.nap.edu/readingroom/books/far, 1999. *Last visited on: 3rd November, 2010.*

[15] M. Crawford. RFC 2464: Transmission of IPv6 packets over Ethernet networks, December 1998. URL `ftp://ftp.internic.net/rfc/rfc1972.txt, ftp://ftp.internic.net/rfc/rfc2464.txt,ftp://ftp.math.utah.edu/pub/rfc/rfc1972.txt,ftp://ftp.math.utah.edu/pub/rfc/rfc2464.txt`. Obsoletes RFC1972 Status: PROPOSED STANDARD. *Last visited on: 3rd November, 2010.*

[16] Crossbow. Wireless sensor network classroom kit. http://www.xbow.com/Products/. *Last visited on: 3rd November, 2010.*

[17] DARPA. Magic gigabit testbed and magic-ii. http://www.magic.net/. *Last visited on: 3rd November, 2010.*

[18] DARTnet. Dartnet. http://www.isi.edu/ touch/dli95/dartnet-dli.html. *Last visited on: 3rd November, 2010.*

[19] Yaozu Dong, Shaofan Li, Asit Mallick, Jun Nakajim, Kun Tian, Xuefei Xu, Fred Yang, and Wilfred Yu. Extending Xen with Intel virtualization technology. *Intel Technology Journal*, 10(3):193–203, August 2006. ISSN 1535-766X. doi: http://dx.doi.org/10.1535/itj.1003. URL `http://developer.intel.com/technology/itj/2006/v10i3/3-xen/1-abstract.htm`.

[20] Dan Duchamp and Greg De Angelis. A hypervisor based security testbed. In *DETER: Proceedings of the DETER Community Workshop on Cyber Security Experimentation and Test on DETER Community Workshop on Cyber Security Experimentation and Test 2007*, pages 3–3, Berkeley, CA, USA, 2007. USENIX Association.

[21] Chris Edwards and Aaron Harwood. Using para-virtualization as the basis for a federated planetlab architecture. In *VTDC '06: Proceedings of the 2nd International Workshop on Virtualization Technology in Distributed Computing*, page 13, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2873-1. doi: http://0-dx.doi.org.oasis.unisa.ac.za:80/10.1109/VTDC.2006.16.

[22] Emulab. Emulab home. www.emulab.net. *Last visited on: 3rd November, 2010.*

[23] Emulab.net. Emulab sites around the world. http://www.emulab.net/doc/docwrapper.php3?docname=topo.html, 05 2009. *Last visited on: 3rd November, 2010.*

[24] Kevin Fall. Network emulation in the vint/ns simulator. In *ISCC '99: Proceedings of the The Fourth IEEE Symposium on Computers and Communications*, page 244, Washington, DC, USA, 1999. IEEE Computer Society.

[25] Kevin Fall and Sally Floyd. Simulation-based comparisons of tahoe, reno and sack tcp. *SIGCOMM Comput. Commun. Rev.*, 26(3):5–21, 1996. ISSN 0146-4833. doi: http://0-doi.acm.org.oasis.unisa.ac.za/10.1145/235160.235162.

[26] D. Fernandez, T. de Miguel, and F. Galan. Study and emulation of ipv6 internet-exchange-based addressing models. *Communications Magazine,*

IEEE, 42(1):105–112, Jan 2004. ISSN 0163-6804. doi: 10.1109/MCOM.2004.1262169.

[27] FreeBSD. Freebsd.org. www.freebsd.org. *Last visited on: 3rd November, 2010.*

[28] Mark Gates and Alex Warshavsky. Iperf. http://sourceforge.net/projects/iperf/. *Last visited on: 3rd November, 2010.*

[29] GNU. Gnu radio. http://gnuradio.org/redmine/wiki/gnuradio, 2010.

[30] gnuplot. Gnuplot. http://www.gnuplot.info/. *Last visited on: 3rd November, 2010.*

[31] Sriram Govindan, Arjun R. Nath, Amitayu Das, Bhuvan Urgaonkar, and Anand Sivasubramaniam. Xen and co.: communication-aware cpu scheduling for consolidated xen-based hosting platforms. In *VEE '07: Proceedings of the 3rd international conference on Virtual execution environments*, pages 126–136, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-630-1. doi: http://doi.acm.org/10.1145/1254810.1254828.

[32] Shashi Guruprasad, Robert Ricci, and Jay Lepreau. Integrated network experimentation using simulation and emulation. IEEE Computer Society, 2005. ISBN 0-7695-2219-X. doi: http://dx.doi.org/10.1109/TRIDNT.2005.21.

[33] Michael Hauben. History of arpanet. http://www.dei.isep.ipp.pt/ acc/docs/arpa.html/. *Last visited on: 3rd November, 2010.*

[34] Jingyi He and S.-H.G. Chan. Tcp and udp performance for internet over optical packet-switched networks. In *Communications, 2003. ICC '03. IEEE International Conference on*, volume 2, pages 1350–1354 vol.2, May 2003. doi: 10.1109/ICC.2003.1204606.

[35] John Heidemann, Lawrence A. Rowe, Lloyd Lim, Open Mash Consortium, and Pad Mah. Otcl and tclcl home. http://otcl-tclcl.sourceforge.net/. *Last visited on: 3rd November, 2010.*

[36] Mike Hibler, Robert Ricci, Leigh Stoller, Jonathon Duerig, Shashi Guruprasad, Tim Stack, Kirk Webb, and Jay Lepreau. Large-scale virtualization in the emulab network testbed. In *ATC'08: USENIX 2008 Annual Technical Conference on Annual Technical Conference*, pages 113–128, Berkeley, CA, USA, 2008. USENIX Association.

[37] C. Hornig. RFC 894: Standard for the transmission of IP datagrams over Ethernet networks, April 1984. URL `ftp://ftp.internic.net/rfc/rfc894.txt,ftp://ftp.math.utah.edu/pub/rfc/rfc894.txt`. Status: STANDARD. *Last visited on: 3rd November, 2010.*

[38] X.W. Huang, R. Sharma, and S. Keshav. The entrapid protocol development environment. *INFOCOM '99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, 3:1107–1115 vol.3, Mar 1999. doi: 10.1109/INFCOM.1999.751666.

[39] IEEE. Ieee 802 standard. http://ieee802.org/index.html, . *Last visited on: 3rd November, 2010.*

[40] IEEE. Ieee 802.11(tm) wireless local area networks. http://www.ieee802.org/11/, . *Last visited on: 3rd November, 2010.*

[41] Internet2. Internet2. http://www.internet2.edu. *Last visited on: 3rd November, 2010.*

[42] Kunihiro Ishiguro. Gnu zebra. http://www.zebra.org/what.html. *Last visited on: 3rd November, 2010.*

[43] ISO/IEC. The c++ standards committee. http://www.open-std.org/jtc1/sc22/wg21/. *Last visited on: 3rd November, 2010.*

[44] IST. Sequin project- deliverable d5.1 - proof of concept testbed. http://archive.dante.net/upload/pdf/SEQ-02-005.pdf, 1999. *Last visited on: 3rd November, 2010.*

[45] Chuck Yoo Jin-Hee Choi. One-way delay estimation and its application. *Elsevier Computer Communications, Vol. 28, Issue 7*, May 2005.

[46] Charles M Kozierok. The tcp/ip guide. www.tcpipguide.com/, 2005. *Last visited on: 3rd November, 2010.*

[47] Ana Kukec, Miljenko Mikuc, and Marko Zec. Imunes - integrated network topology emulator / simulator. http://old.tel.fer.hr/imunes/. *Last visited on: 3rd November, 2010.*

[48] Ed Kurose, J. Report of nsf workshop on network research testbeds. Technical report, National Science Foundation (NSF), October 2002.

[49] LinuxFoundation.org. Net:bridge. http://www.linuxfoundation.org/en/Net:Bridge. *Last visited on: 3rd November, 2010.*

[50] G. Malkin. RFC 2453: RIP version 2, November 1998. URL `ftp://ftp.internic.net/rfc/rfc1388.txt,ftp://ftp.internic.net/rfc/rfc1723.txt,ftp://ftp.internic.net/rfc/rfc2453.txt,ftp://ftp.internic.net/rfc/std56.txt,ftp://ftp.math.utah.edu/pub/rfc/rfc1388.txt,ftp://ftp.math.utah.edu/pub/rfc/rfc1723.txt,ftp://ftp.math.utah.edu/pub/rfc/rfc2453.txt,ftp://ftp.math.utah.edu/pub/rfc/std56.txt`. See also STD0056. Obsoletes RFC1388, RFC1723. Status: STANDARD. *Last visited on: 3rd November, 2010.*

[51] Viraj Bais Clayton Kirkwood Mark Andrews, Bernd Altmeier. Ntp: The network time protocol. http://www.ntp.org/, 10 2010. *Last visited on: 3rd November, 2010.*

[52] S. McCanne, S. Floyd, and K. Fall. ns2 (network simulator 2). http://www-nrg.ee.lbl.gov/ns/. URL `http://www-nrg.ee.lbl.gov/ns`.

[53] Aravind Menon, Jose Renato Santos, Yoshio Turner, G. (John) Janakiraman, and Willy Zwaenepoel. Diagnosing performance overheads in the xen virtual machine environment. In *VEE '05: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, pages 13–23, New York, NY, USA, 2005. ACM. ISBN 1-59593-047-7. doi: http://doi.acm.org/10.1145/1064979.1064984.

[54] D. Meyer and G. Zobrist. Tcp/ip versus osi. *Potentials, IEEE*, 9(1):16–19, Feb 1990. ISSN 0278-6648. doi: 10.1109/45.46812.

[55] J. Moy. RFC 2328: OSPF version 2, April 1998. URL `ftp://ftp.internic.net/rfc/rfc2178.txt,ftp://ftp.internic.net/rfc/rfc2328.txt,ftp://ftp.internic.net/rfc/std54.txt,ftp://ftp.math.utah.edu/pub/rfc/rfc2178.txt,ftp://ftp.math.utah.edu/pub/rfc/rfc2328.txt,ftp://ftp.math.utah.edu/pub/rfc/std54.txt`. See also STD0054 Obsoletes RFC2178. Status: STANDARD. *Last visited on: 3rd November, 2010.*

[56] Abraham Mukosi Mukwevho, John Andrew van der Poll, and Robert Mark Jolliffe. A virtual integrated network emulator on XEN (viNEX). pages 1–7,

March 2009. doi: 10.4108/ICST.SIMUTOOLS2009.5745. Editors: O. Dalle, L.-F. Perrone, G. Stea and G. A. Wainer, ISBN: 978-963-9799-45-5.

[57] NetBSD. Netbsd.org. www.netbsd.org. *Last visited on: 3rd November, 2010.*

[58] Heiko Oberdiek. pdfcrop.pl. http://www.ctan.org/tex-archive/support/pdfcrop/. *Last visited on: 3rd November, 2010.*

[59] Matrin S Olivier. *Information Technology Research.* Van Schaik Publishers, 2nd edition, 2004.

[60] Opensource.org. Open source initiative. http://www.opensource.org, 2010. *Last visited on: 3rd November, 2010.*

[61] Shawn Ostermann. tcptrace. http://www.tcptrace.org/. *Last visited on: 3rd November, 2010.*

[62] Oriana Palivan. Xen networking. http://wiki.xensource.com/xenwiki/XenNetworking/, 02 2009. *Last visited on: 3rd November, 2010.*

[63] PlanetLab. Planetlab. http://www.planet-lab.org. *Last visited on: 3rd November, 2010.*

[64] J. Postel. RFC 793: Transmission control protocol, September 1981. URL `ftp://ftp.internic.net/rfc/rfc793.txt,ftp://ftp.math.utah.edu/pub/rfc/rfc793.txt`. See also STD0007. Status: STANDARD. *Last visited on: 3rd November, 2010.*

[65] J. Postel and J. K. Reynolds. RFC 1042: Standard for the transmission of IP datagrams over IEEE 802 networks, February 1988. URL `ftp://ftp.internic.net/rfc/rfc1042.txt,ftp://ftp.internic.net/rfc/rfc948.txt,ftp://ftp.math.utah.edu/pub/rfc/rfc1042.txt,ftp://ftp.math.utah.edu/pub/rfc/rfc948.txt`. Obsoletes RFC0948 [86]. Status: STANDARD. *Last visited on: 3rd November, 2010.*

[66] Robert N. M. Watson Poul-Henning Kamp. Jails: Confining the omnipotent root. In *2nd SANE Conference*, May 2000.

[67] J.M. Pullen, D. Cohen, and D. Wood. Emerging technologies - national/defense information infrastructure and the defense information systems network. volume 3, pages 1038–1043 vol.3, Oct 1993. doi: 10.1109/MILCOM.1993.408664.

[68] J. K. Reynolds and J. Postel. RFC 1010: Assigned numbers, May 1987. URL `ftp://ftp.internic.net/rfc/rfc1010.txt,ftp://ftp.internic.net/rfc/rfc1060.txt,ftp://ftp.internic.net/rfc/rfc990.txt,ftp://ftp.math.utah.edu/pub/rfc/rfc1010.txt,ftp://ftp.math.utah.edu/pub/rfc/rfc1060.txt,ftp://ftp.math.utah.edu/pub/rfc/rfc990.txt`. Obsoleted by RFC1060. Obsoletes RFC0990. Status: HISTORIC. *Last visited on: 3rd November, 2010.*

[69] Luigi Rizzo. Dummynet home page. http://info.iet.unipi.it/ luigi/dummynet. *Last visited on: 3rd November, 2010.*

[70] Luigi Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *ACM SIGCOMM*, 2002.

[71] Luigi Rizzo and Marta Carbone. Dummynet: Revisited. Technical report, Universita di Pisa, 2009.

[72] Paul 'Rusty' Russell. Ebtables firewalling. http://ebtables.sourceforge.net/. *Last visited on: 3rd November, 2010.*

[73] Kevin Savetz, Neil Randall, and Yves Lepage. Mbone:multicasting tomorrow's internet. http://www.savetz.com/mbone/, 1998. *Last visited on: 3rd November, 2010.*

[74] Timothy Jason Shepard. xplot. www.xplot.org. *Last visited on: 3rd November, 2010.*

[75] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts*. John-Wiley and Sons, 8th edition, 2010.

[76] Amit Singh. An introduction to virtualization. http://www.kernelthread.com/publications/virtualization/. *Last visited on: 3rd November, 2010.*

[77] sourceforge. sourceforge. http://sourceforge.net/. *Last visited on: 3rd November, 2010.*

[78] Ralf Spenneberg. Emulating networks using user-mode linux. http://www.samag.com/documents/s=8997/sam0401a/0401a.htm. *Last visited on: 3rd November, 2010.*

[79] Inc. Sun Microsystems. Virtualbox. http://www.virtualbox.org/. *Last visited on: 3rd November, 2010.*

[80] Andrew Tanenbaum. *Computer Networks.* Prentice Hall Professional Technical Reference, 5th edition, 2011. ISBN 0132126958.

[81] Van Jacobson. Congestion avoidance and control. In *SIGCOMM '88: Symposium proceedings on Communications architectures and protocols*, pages 314–329, New York, NY, USA, 1988. ACM. ISBN 0-89791-279-9. doi: http://doi.acm.org/10.1145/52324.52356.

[82] Craig Leres Van Jacobson and Steven McCanne. tcpdump/libpcap. http://www.tcpdump.org/. *Last visited on: 3rd November, 2010.*

[83] Mark Allen Weiss. *Data Structures and Algorithm Analysis in C++.* The Benjamin/Cummings Publishing Company, Inc., 1994.

[84] Brian White, Shashi Guruprasad, Mac Newbold, Jay Lepreau, Leigh Stoller, Robert Ricci, Chad Barb, Mike Hibler, and Abhijeet Joglekar. Netbed: an integrated experimental environment. *SIGCOMM Comput. Commun. Rev.*, 32(3):27–27, 2002. ISSN 0146-4833. doi: http://0-doi.acm.org.oasis.unisa.ac.za:80/10.1145/571697.571716.

[85] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. pages 255–270, Boston, MA, December 2002.

[86] I. Winston. RFC 948: Two methods for the transmission of IP datagrams over IEEE 802.3 networks, June 1985. URL `ftp://ftp.internic.net/rfc/rfc1042.txt,ftp://ftp.internic.net/rfc/rfc948.txt,ftp://ftp.internic.net/rfc/std43.txt,ftp://ftp.math.utah.edu/pub/rfc/rfc1042.txt,ftp://ftp.math.utah.edu/pub/rfc/rfc948.txt,ftp://ftp.math.utah.edu/pub/rfc/std43.txt`. Obsoleted by RFC1042, STD0043. Status: UNKNOWN. *Last visited on: 3rd November, 2010.*

[87] Xen. Xen home. http://www.xen.org/. *Last visited on: 3rd November, 2010.*

[88] Xen.org. Xen roadmap proposal. http://wiki.xensource.com/xenwiki/XenRoadMap, 05 2009. *Last visited on: 3rd November, 2010.*

[89] Marko Zec. Implementing a clonable network stack in the freebsd kernel. In *In Proceedings of the USENIX 2003 Annual Technical Conference*, pages 137–150, 2003.

[90] Marko Zec and Miljenko Mikuc. Operating system support for integrated network emulation in imunes. In *In Proc. of the 1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS), Boston, MA, 2004.*, 2004.

[91] H. Zimmermann. Osi reference model–the iso model of architecture for open systems interconnection. *Communications, IEEE Transactions on [legacy, pre - 1988]*, 28(4):425–432, Apr 1980. ISSN 0096-2244.

Index

X