ScaleNet: A Platform for Scalable Network Emulation

A Thesis Submitted
in Partial Fulfillment of the Requirements
for the Degree of
Master of Technology

by

Sridhar Kumar Kotturu

to the

Department of Computer Science & Engineering
Indian Institute of Technology, Kanpur

June, 2005

Certificate

This is to certify that the work contained in the thesis entitled "*ScaleNet: A Platform for Scalable Network Emulation*", by *Sridhar Kumar Kotturu*, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

[illegible]

June, 2005

(Dr.Bhaskaran Raman)
Department of Computer Science & Engineering,
Indian Institute of Technology,
Kanpur.

Abstract

The need for large-scale protocol development environments for protocol testing and verification has been increasing with the rapid growth of the Internet and the evolution of network protocols. Due to approximations in network simulation, accuracy of the reproduction of protocol execution may be less than desired levels. For real networks, it is very hard to reconfigure and its behaviour is not easily reproducible. Whereas with emulation, it is easy to reconfigure and its behaviour is easily reproducible.

Dummynet, netbed and remote unix lab environment(RULE) are some of the emulation platforms. Dummynet can not emulate complex network topologies. It is not implemented as a loadable kernel module. Its implementation only exists between TCP and IP layers, so it can not emulate other protocols(eg UDP). Netbed requires several resources. It uses FreeBSD jail functionality for creation of several virtual hosts. This jail functionality does not exist in other operating systems, so in that case it has to do one-to-one mapping of the virtual resources to physical resources. RULE also creates several virtual hosts on a physical machine, but routing is not possible between these virtual hosts and it is not scalable.

In this work, we have created ScaleNet, an emulation platform that can emulate large networks using limited physical resources. This emulation platform can be used to test any network protocol. Several virtual hosts are created on few physical systems. NIST Net is used for applying effects such as bandwidth limitation, delay, packet duplication, packet drops etc. The approach is scalable and cost effective. We have shown that our emulation platform scales upto 50 virtual hosts per physical host, while the scalability of Netbed is restricted to about 10-20 virtual hosts per machine.

Acknowledgements

Iam grateful to my thesis supervisor Dr.Bhaskaran Raman for his guidance through out this thesis work. I learnt how to do research from him. Inspite of his busy schedule he was always approachable and gives valuable suggestions.

I also wish to thank all the faculty members of the department of computer science and engineering for their excellent teaching. I also wish to thank all the technical staff of the department.

I also like to thank Pratik Mehta for his suggestions and comments on this report. I also like to thank Anantha Kiran and Srinivasa Rao Myla for their feedback on the work presented here. I also wish to thank all my classmates for their support during my stay at IITK.

I would like to thank my parents and brothers for taking me to this stage in life.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Background and Motivation

Emulation is a combination of *simulation* and *field testing.* Simulation is a process of execution of representations of code in synthetic environment while *field testing* is executing real code in real environment. Emulation involves running real protocols and operating systems and it applies synthetic delays and faults.

Large networks can be created by using simulation, but they may not exactly model the real environment. Simulators can not accurately model factors such as processing overheads, scheduling of processes, disk speeds and CPU load. Real networks can be used for protocol testing etc., but it is hard to reconfigure and its behaviour is not easily reproducible. Emulated network doesn't require real deployed network. It only needs a software model. It is easy to vary emulated network configuration and its behavior can be easily reproduced at will. So emulation is the better alternative.

With the rapid growth of the Internet, network technologies and network protocols there is a need for protocol development environments. These testbeds can be used for network protocol testing, understanding some peculiar behaviour, finding the coding bugs, performance analysis of applications etc.

There are several emulation platforms like dummynet[1], nistnet[2], FreeBSD jail hosts[3], netbed[4]. Dummynet can not emulate complex network topologies. It can not apply effects such as packet duplication, delay variation. So we can not exactly get the real network behaviour. NIST Net can not create several virtual hosts on a physical machine. FreeBSD jail hosts can create several virtual hosts on a physical machine. But routing is not possible between these virtual hosts and it is not scalable. Netbed is very costly. It requires several hardware resources. For creating several virtual hosts on a physical machine, it uses FreeBSD jail host functionality. This *jail* functionality exists only for FreeBSD, so on other operating systems it has to do one-to-one mapping of virtual hosts to physical hosts. It uses dummynet for emulation of the links, so we can not get the exact network behaviour.

Our goal is to emulate large networks using few physical resources. Using these emulated large networks, any kind of network protocol can be studied. Since large networks are emulated, it is very useful for studying internet scale protocols. Arbitrary network topologies can be created. It is cost effective and scalable.

We can do *performance analysis* of applications using this emulation platform. This emulated platform can be used for *debugging* purposes. We may not find all the bugs in real implementations in simulations but these can be found in emulation.

1.2 Challenges in Scalable Network Emulation

In the emulation platform, we seek to create several virtual hosts on a limited number of physical machines. For this different IP aliases have to be assigned to the Ethernet card and each IP alias is treated as a different virtual host. The support of routing tables for each virtual host is a challenge. Each application program has to be assigned to some virtual host. While doing this application program code should not be changed. For example an application issues a command to add a route to some destination. In normal system there is only one system routing table and the route is added in that routing table. But in this emulation platform there are several virtual hosts on a physical machine. So there should be some mechanism to specify the particular virtual host whose routing table has to be updated. One possibility is to include the virtual host IP address, whose routing table has to be updated, in the system call itself. But this requires changing the application program.

Another issue is that routing between different IP aliases is not possible which poses a major problem. Suppose three IP aliases say 1, 2 and 3 are assigned to Ethernet interface. Now we want to route packets from IP alias 1 to 3 through alias 2. For this, add an entry in the routing table corresponding to virtual host 1.

```
#route add 3 gw 2 dev eth0
```

The above command implies that packets destined for 3 are sent to gateway 2. But routing is not done, since the IP addresses are local aliases and the routing table is not consulted to send packets destined for local aliases. This is shown in the Figure 1.1. The packet should be sent from 1 to 2 and then from 2 to 3, whereas it is directly delivered to alias 3 in the above example.

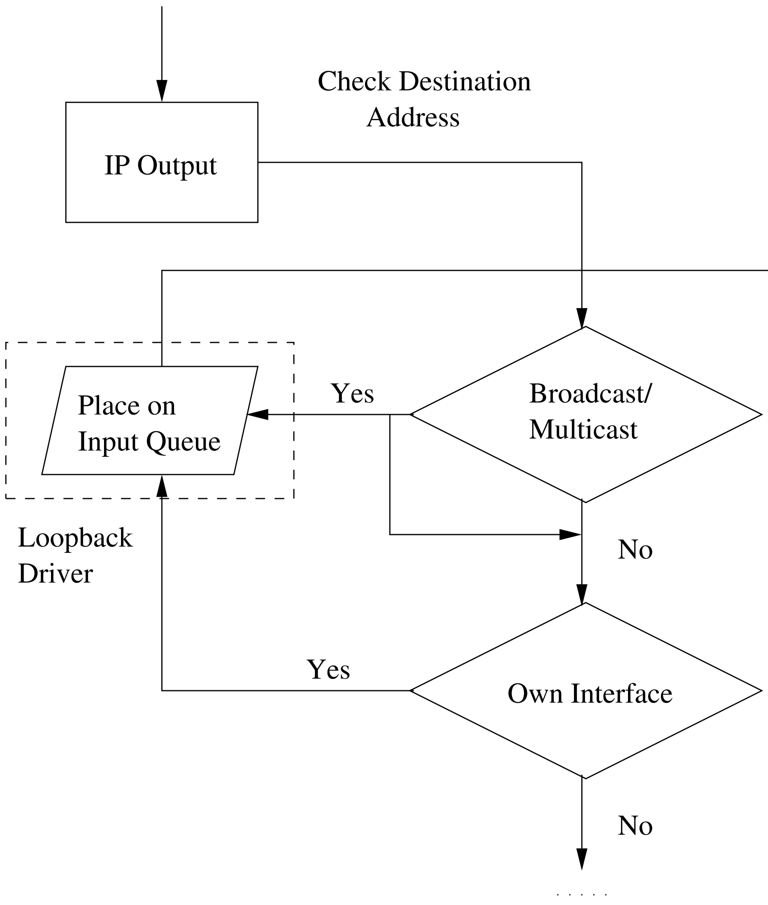


Figure 1.1: Loopback of packets between Ethernet Device Aliases

1.3 Contributions of our work

With this emulation tool large networks with any arbitrary network topology can be created using few systems. We use NIST Net[2] for applying the effects such as bandwidth limitation, delay, packet drops, packet duplication etc on the packets in the emulation platform. The unmodified application code is run on this emulation platform. As large scale networks are emulated, it is very useful for studying Internet scale protocols such as BGP.

The required topology is generated on which the protocol testing has to be done. Several virtual hosts have to be created on each physical system. After that, nodes in the topology have to be assigned to the virtual hosts on the physical systems. Routing tables have to be assigned to each virtual host according to the topology of the network. NIST Net module has to be loaded and initialized with the required bandwidth limitations, delays etc on each physical host. Any network application can be run on this emulation platform.

NIST Net module removes the IP protocol handler and registers its own handler for handling incoming packets. After applying all the effects it hands over the packet to the IP protocol handler for further processing.

In this emulation platform packets are captured at netfilter hooks in the Linux kernel network stack and nexthop IP address is found from the routing table corresponding to the current virtual host. Nexthop IP address may lie on the same physical machine or on another physical machine. Extra IP header is added to the packet with nexthop IP address as the destination IP address and the current virtual host IP address as the source IP address. If the nexthop is on another physical machine then the destination MAC address is changed to the MAC address of the nexthop and the packet is transmitted to the nexthop.

We have created 50 virtual hosts on a single physical machine whereas Netbed created 20 virtual hosts on a single physical machine.

1.4 Organization of the Report

This thesis report is organized as follows. Chapter 2 describes the related work. Chapter 3 describes the design and implementation of the emulation platform. Chapter 4 gives the experimental results. Chapter 5 contains conclusions and future work.

Appendix A describes the aspects of Linux kernel programming relevant to ScaleNet. Appendix B describes source code organization.

Chapter 2

Related Work

In this section we discuss other emulation platforms and their drawbacks. In Section 2.1, we discuss dummynet. In Section 2.2, we discuss NIST Net. In Section 2.3, we discuss FreeBSD jail hosts. In Section 2.4, we discuss netbed. In Section 2.5, we discuss User Mode Linux. In Section 2.6, we discuss `Alpine` and a Summary of all the emulation platforms is given in Section 2.7.

2.1 Dummynet

Dummynet[1] is one of the most well-known network emulators. It is a simple, flexible and accurate network emulator that was built with modifications to the communication protocol stack. It emulates the effects of finite queues, bandwidth limitations and delays. It runs in an operational system, thus utilizing the real traffic generators and real protocol implementations. This tool allows carrying out experiments with network protocols by running a set of unmodified real world applications.

Dummynet inserts *routers* with bounded queue size and a queueing policy, communication links with given bandwidth limitation and delay, packet reordering and packet loss in the flow of data. The *routers* and *communication links* can be modeled by using two queues: router queue and link queue. They are between the protocol layer under observation and the next lower protocol layer as shown in Figure 2.1.

There are two sets of queues one in each direction of the communication. Router queue is characterized by number of packets and a queueing policy. Link queue is characterized by bandwidth limitation and delay. When a layer communicates with the other layer, packets are inserted in the router queue. Packets are inserted until the queue reaches its maximum size. Any queueing policy can be used. Packet reordering can be done in this queue. Packets are moved from router queue to link queue according to the bandwidth limitation. Packets are stored in the link queue for certain amount of time which corresponds to the delay that needs to be applied on the packets, after which packets are removed from this link queue. Packet drops can occur at this point of time. Thus this emulates the bandwidth limitation and delay. Packets moving to the link queue and removing the packets from link queue can be done at periodic intervals which is submultiple of communication delay.

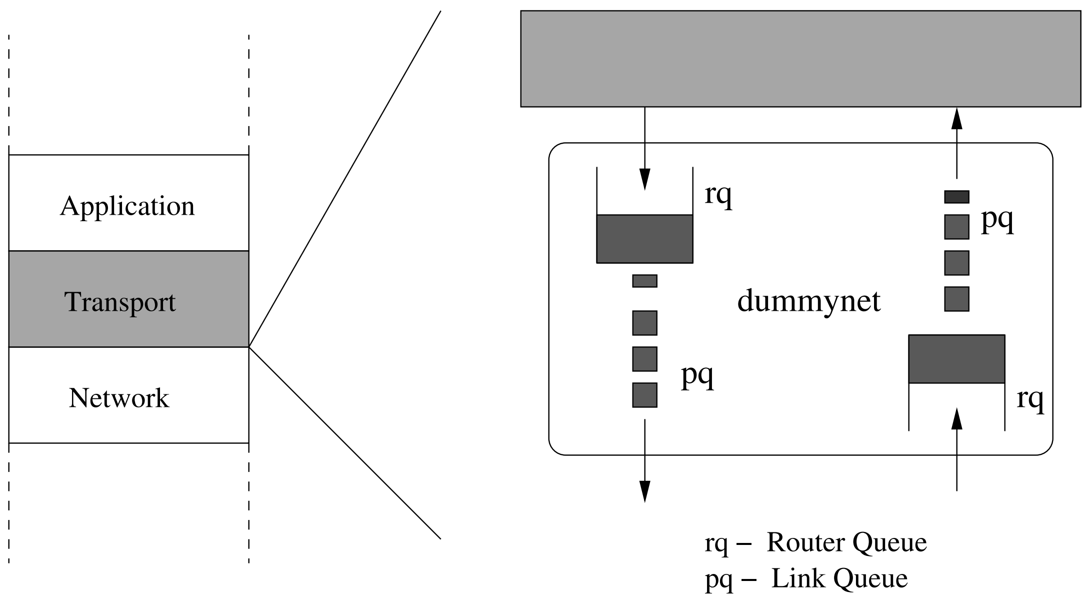


Figure 2.1: The Architecture of Dummynet.(from [1])

There are certain limitations to the Dummynet.

1. The granularity of the operating system timer causes several approximations.
2. The periodic timer may run late and may miss some of the clock ticks. Another limitation is that the events in the Dummynet may occur synchronously with

the system timer. This won't happen in real networks and may hide or amplify certain phenomena.

3. Dummynet was not implemented as a Loadable Kernel Module and is not flexible to make any modifications. So each time we make some modifications, we have to recompile the kernel.
4. It can not emulate complex network topologies.
5. It can not apply effects such as packet duplication, delay variation etc. So we can not exactly get the real network behaviour.
6. It can not apply the effects for selected data flows.
7. Dummynet implementation exists only between TCP and IP. So it can not apply effects to other protocol packets such as UDP.
8. Dummynet exists only for FreeBSD, which is not so popular.

2.2 NIST Net

NIST Net[2] emulates the behaviour of any network at a particular router. It applies that network behaviour on the packets passing through it. It is shown in the Figure 2.2.

NIST Net has a table of emulation entries. Each entry consists of three parts. First part is the packet matching criteria and second part contains the effects to be applied to the packets which satisfy the selection criteria and third part contains a set of statistics for that emulation entry. We can load thousands of emulation entries. These entries can be added or removed from the emulator dynamically.

Packets are matched according to the source and destination IP address, type of service and source, destination ports. NIST Net applies effects such as fixed and variable packet delay, bandwidth limitations, random and congestion dependent packet loss, packet duplication, packet reordering. Congestion dependent loss is emulated using Derivative Random Drop(DRD). It has two parameters: DRDmin

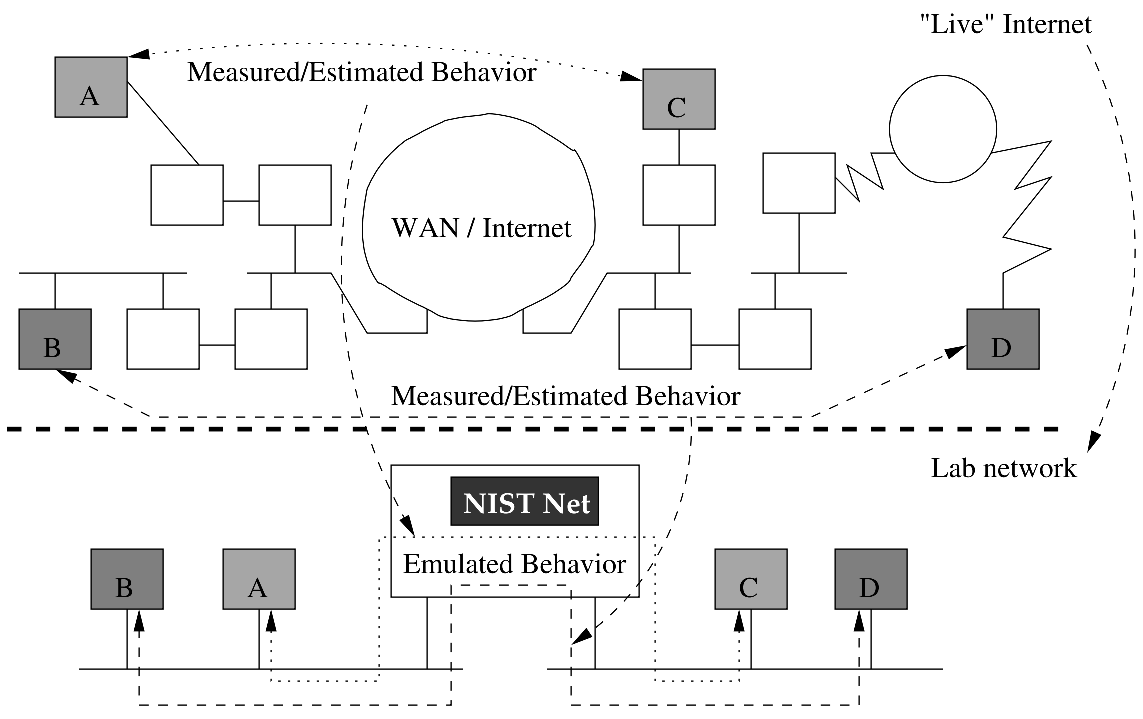


Figure 2.2: NIST Net as a "network in a box". (from [2])

and DRDmax. If the queue length is less than DRDmin, no packet is dropped and 95% packets are dropped if the queue length is greater than DRDmax. NIST Net provides several statistics for each emulation entry. It provides the number of packet drops, duplications, average bandwidth, queue length and number of bytes sent.

NIST Net consists of two main parts: a loadable kernel module and user interfaces. Since NIST Net is a loadable module, it can be loaded or unloaded at any time without interrupting any active connections. The NIST Net module has two hooks in the Linux kernel. Packet interception code replaces IP Packet handler with its own handler and intercepts all the IP packets and checks all the emulator entries. If this packet matches any emulator entry, the effects for this emulation entry are applied on the packet. After that, it calls the IP handler to handle the packet. The fast timer uses system clock as a timer source for scheduling delayed packets. NIST Net provides both graphical user interface and command line interface. Command line interface is useful for writing shell scripts.

It is not possible to create several virtual hosts using NIST Net. NIST Net won't do any routing.

We use NIST Net for applying effects such as bandwidth limitation, delay, packet drops, packet duplications etc. in ScaleNet. NIST Net is designed to be scalable in terms of the number of emulation entries, and also in terms of the amount of bandwidth it can support. We leverage these aspects of scaling directly from NIST Net.

2.3 FreeBSD Jail Hosts

Several FreeBSD[5] virtual hosts can be created on a single machine using the FreeBSD's *jail* functionality. Remote Unix Lab Environment(RULE)[3] creates several virtual hosts by using the FreeBSD *jail* functionality. The main purpose of the RULE is to minimize the infrastructure cost. At Swinburne University of Technology, they have placed several FreeBSD hosts in a rack and these can be accessed by students from remote terminals or 802.11-equipped laptops.

Jail Hosts have distinct IP address, user accounts. These are shown in Figure 2.3. The three jail hosts A, B and C appear as three different hosts from outside.

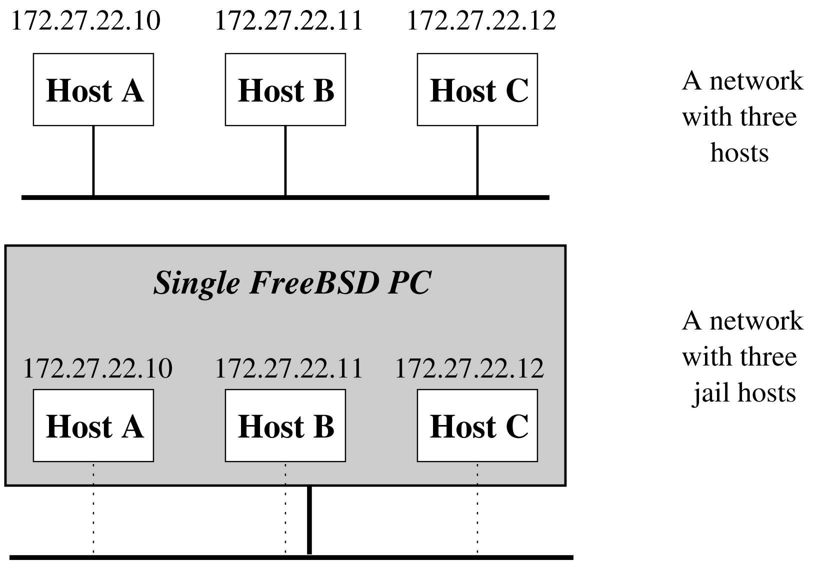


Figure 2.3: Jail hosts appear as independent IP hosts on the network. (from [3])

Root access to the jail host is given to the student. Students can manage user or group accounts within the jail host. They can compile and run network applications. Each student can run his own web, email or ftp servers.

Processes belonging to a particular jail host inherit that jail host's restricted context. Filesystem accesses by processes belonging to a jail host are remapped relative to the jail host's root directory. For example let the jail host A root directory be /jailA. If a process in jail host A refers /usr/java then it is remapped to /jailA/usr/java. Network communications are also remapped. For example if process in jail host A wants to bind to a TCP socket with wild card IP Address '*' and to port 8000, then it is remapped to <Jail host A IP>:8000.

Virtual jail hosts are managed by Jail Host Toolkit(JHT). It is used by primary host administrator. JHT is a set of scripts for building, booting and killing a jail host.

The main limitation of FreeBSD jail hosts is that they are not scalable. Each jail host's file system is placed on a separate FreeBSD disk partition. There can be upto eight FreeBSD partitions in a FreeBSD disk slice. There are four FreeBSD slices. So there is a theoretical upper limit of 24 jail hosts per primary host. A jail host cannot get access to raw socket.

2.4 Netbed

Emulab is an emulation platform and Netbed[4] is an extension of Emulab. Netbed integrates simulation, emulation and live network experimentation. Simulated resources are integrated with real traffic by using *nse*. Emulab is used for emulation. Netbed is a time and space shared platform. The user specifies the network topology either graphically or by ns-scripts. The nodes in the user specified topology are mapped to local nodes, distributed nodes or simulated nodes. The links are mapped to local links, wide area links or emulated using dummynet. Currently, the *emulab* portion of *netbed* contains 168 PCs of varying configurations and each has four interfaces. Experiment creation times in Emulab are three minutes for a single node topology, and six and half minutes for an 80-node topology(from [6]).

Netbed automatically maps virtual resources onto available physical resources by using *simulated annealing*[6]. Simulated Annealing is a randomized heuristic search technique. It uses a *cost function*, for determining the cost of a particular configuration, and a *generation function*, for generating a new configuration from the old configuration. This new configuration is evaluated by the *cost function*. If this new configuration is better than the previous one then it is accepted. Otherwise it is accepted with certain probability. This is done to avoid local minima.

Original Emulab mapped virtual nodes and links one-to-one onto dedicated PCs and Ethernet links. Later they used FreeBSD Jail host functionality to create multiple virtual hosts on a physical system.

The main limitation of Emulab is it requires several hardware resources. So it is very costly. Several virtual nodes are created by using FreeBSD jail host functionality. Only FreeBSD supports virtual jail hosts. So in other cases, it forces one-to-one mapping between virtual nodes and physical machines. Netbed scales upto 20 virtual hosts per physical machine whereas ScaleNet scales upto 50 virtual hosts. As mentioned earlier, FreeBSD jail hosts are not scalable.

2.5 User Mode Linux

User Mode Linux(UML) [7] is a Linux kernel that can be run as a normal user process on Linux machine. Since Linux kernel can be used as a normal user level process, kernel development and debugging is very easy. We can use normal debuggers such as *gdb* for debugging the kernel. We can use various other Linux distributions on a single disk partition. It can be used as a secure sandbox since the processes running in the UML have no access to the physical machine. We can create arbitrary network topologies using UMLs.

The main disadvantages with User Mode Linux are:

- User Mode Linux runs applications inside itself at 20% slowdown compared to the host system. (from [7])
- Lot of extra overhead in creating virtual host, since entire kernel image is used

for creating virtual host. So it is not scalable.

2.6 Alpine

Application-Level Protocol Infrastructure for Network Experimentation(`Alpine`)[8] moves an unmodified FreeBSD network stack into a userlevel library. Since the protocol stack is moved to userlevel, debugging of network protocols becomes easy. `Alpine` sends outgoing packets from userlevel network stack using raw socket, thus avoiding the traversing of the kernel network stack by the packet. It receives packets by using `libpcap`. It prevents the kernel from processing of packets destined for applications using `Alpine` by filtering at the firewall. The central port server maintains the firewall up-to-date with the ports that `Alpine` applications are using. `Alpine` provides all the socket related system calls. Applications are linked with `Alpine` library by setting the LD_PRELOAD environment variable to the `Alpine` library. By this applications use `Alpine's` networking stack instead of kernel network stack.

Disadvantages of `Alpine` are given below.

- `Alpine` does not support network emulation.
- If the network is too busy or machine is slow, this won't work well. The kernel allocates limited buffer for queueing the received packets from the network. If this buffer is full, kernel drops the packets.
- Extra overhead to copy each and every packet to userlevel and processing in the userlevel. SIGALRM handler is used to poll for packets once every 10ms (from [8]).
- Extra overhead in maintaining up-to-date information at the firewall.
- It exists only for FreeBSD.

2.7 Summary

The comparison of all emulation platforms is shown in the Table 2.7.

	Performance	Many VMs per PM	Hardware Resources	Scalable	OS
Dummynet	High	No	Low	-	FreeBSD
NIST Net	High	No	Low	-	Linux
FreeBSD jail hosts	Low	Yes	High	No	FreeBSD
Netbed	High	Yes	High	Partly	FreeBSD
User Mode Linux	Low	Yes	High	No	Linux
Alpine	Low	No	Low	No	FreeBSD
ScaleNet	High	Yes	Low	Yes	Linux

Table 2.1: Comparison of the emulation platforms.

Chapter 3

Design and Implementation of ScaleNet

We use NIST Net and Linux to build ScaleNet. NIST Net works only on Linux. Linux provides a clean way of accessing the data packets in the network stack using modules. Linux is so popular and very good documentation is available.

ScaleNet is built using loadable kernel modules. Modules can be loaded and unloaded dynamically. There is no need to rebuild and reboot the kernel each time we make some modifications to the modules.

We use NIST Net for applying effects such as bandwidth limitation, delay, packet drops, packet duplications etc. in ScaleNet. NIST Net is designed to be scalable in terms of the number of emulation entries, and also in terms of the amount of bandwidth it can support.

We create several virtual host IP addresses using IP aliases. A packet may traverse via multiple IP aliases, and may use the loopback interface, as well as the ethernet interface. Loopback packet MTU is 16436 bytes and that of ethernet packets is 1500 bytes. Extra IP header is added to the packet for routing and NIST Net purposes. To prevent the fragmentation of the packet, MTUs of both loopback and ethernet packets are changed to 1480 bytes.

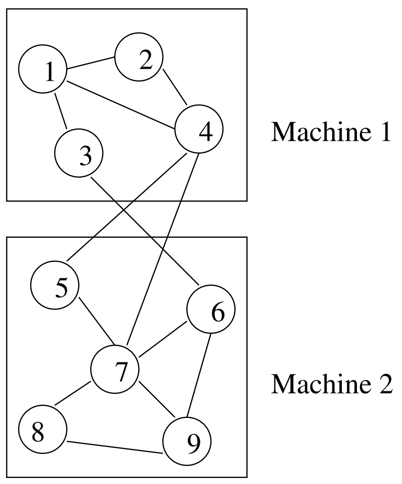


Figure 3.1: An example network topology

An example topology is shown in the Figure 3.1. There are nine virtual hosts distributed on two physical machines. This topology can be emulated using ScaleNet. Each virtual host have an IP address, routing table and some associated applications.

Overview of ScaleNet design is given in Section 3.1. We discuss the processing of outgoing and incoming packets in Section 3.2 and Section 3.3 respectively. Network topology and routing is described in Section 3.4. Virtual hosts are described in Section 3.5.

3.1 Overview of ScaleNet Design

We are building emulation platform as Linux loadable kernel modules. (Linux kernel programming aspects which are relevant to ScaleNet are described in Appendix A.) Linux kernel version 2.4.20-8 is used because NIST Net doesn't support 2.6.* kernel version. NIST Net supports Linux kernel versions 2.2.* and 2.4.*.

ScaleNet architecture is shown in the Figure 3.2. In the emulation platform there is one module for handling incoming packets, and another module for handling outgoing packets. Kernel module `IP-IP-in` handles incoming packets and `IP-IP-out` handles outgoing packets. `RoutingTables` module handles the routing tables and IP addresses of virtual hosts, nexthop IP address and corresponding MAC address and all the IP addresses of a router node. `RoutingTables` module gets this data from a userlevel program, allocates sufficient memory and exports all data to rest of the kernel. `chardev` module provides *ioctl* interface to access and modify routing tables.

syscall_hack module redirects the relevant system calls. For example if an application belongs to a virtual host issues *route add* command, it is directed to the routing table corresponding to the virtual host. *pid_ip* module associates an application with a virtual host. It also provides *ioctl* interface to access and change this association dynamically. `dst_entry_export` module exports `dst_entry` objects, which have routing information for the destination, to other parts of the kernel.

We use NIST Net for applying various effects on the packets passing in the emulation platform. NIST Net removes the IP protocol handler and inserts its own module to capture the packets from the networking code. After processing the packet completely, NIST Net hands over the packet to the IP protocol handler.

Consider an example for sending a packet from virtual host 1 to 3 via virtual host 2. This is shown in the Figure 3.3. An application sends the packet with source IP address 1 and destination IP address 3. An entry is added in the NIST Net specifying the characteristics of the link between 1 and 2. Similarly another entry corresponding to the link between virtual hosts 2 and 3 is added. The packet is captured by `IP-IP-out` module at the netfilter hook NF_IP_LOCAL_OUT. It identifies that these IP addresses belongs to the emulation platform. It consults the routing table of the virtual host 1 for finding the nexthop address for sending packet to virtual host 3. At virtual host 1, nexthop address for 3 is 2.

Extra IP header is added for routing and NIST Net purposes. NIST Net captures packets based on IP header details. In the current packet the destination IP address is 3 and source IP address is 1. If we want to apply the effects for link between virtual host 1 and 2, NIST Net won't capture this packet since the addresses are not matched. So we add extra IP header at the beginning of the packet with nexthop IP address (virtual host 2) as the destination IP address and current virtual host

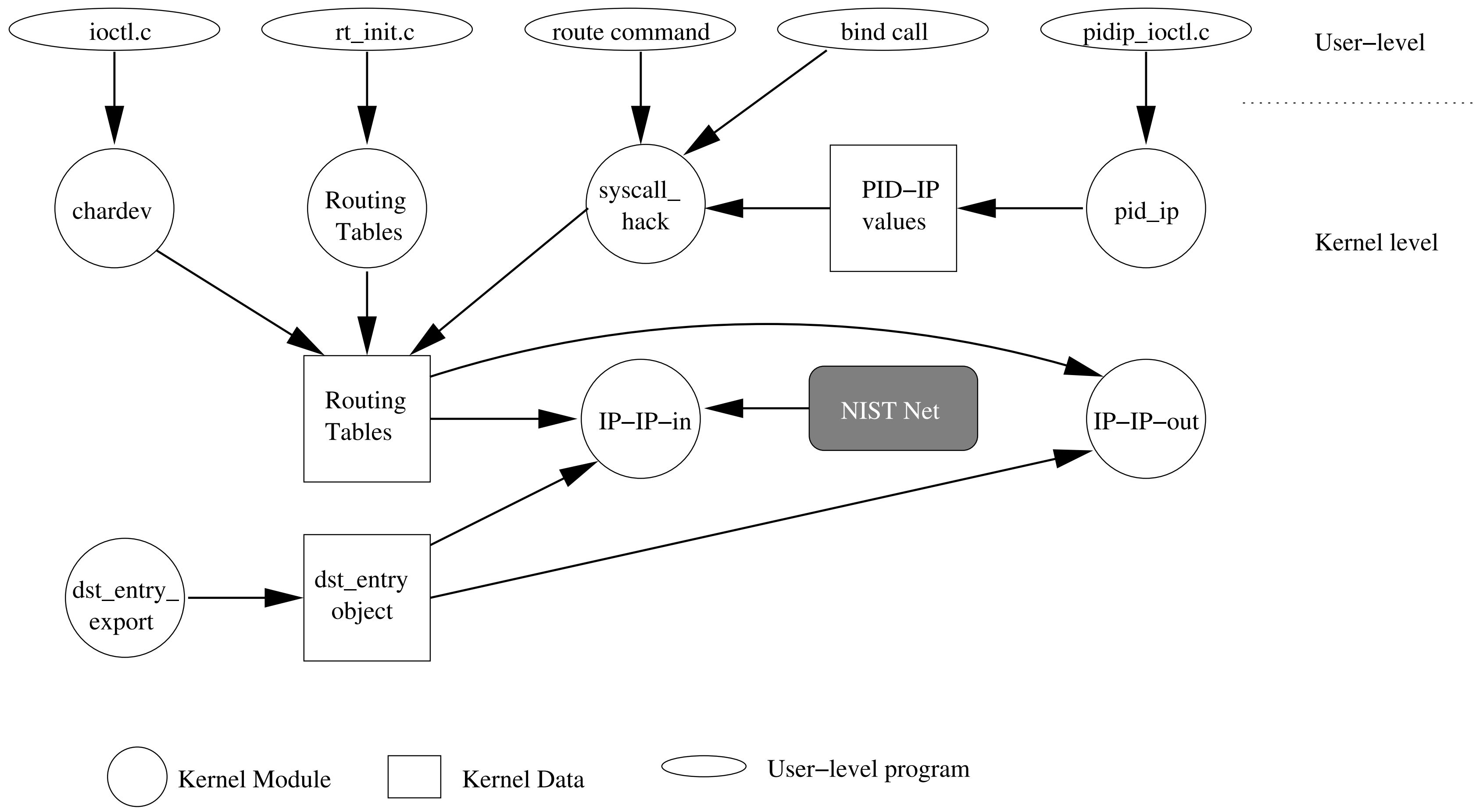


Figure 3.2: ScaleNet Architecture

address (virtual host 1) as the source IP address.

After adding the extra IP header `IP-IP-out` module leaves the packet for further processing. Since these are local alias addresses, the loopback driver places the packet in the input queue.

The packet is captured by the NIST Net module. It applies all the effects corresponding to the link between 1 and 2. After applying the required effects it hands over the packet to the IP protocol handler. The packet is captured by IP-IP-in module at the netfilter hook NF_IP_PRE_ROUTING. It checks whether the packet reached final destination. Since the packet is not reached the final destination, it finds the nexthop address using routing table corresponding to the current virtual host(2). It changes the destination IP address to the nexthop and source IP address to the current virtual host address in the extra IP header and fills the remaining fields appropriately. It once again sends the packet using *dev_ queue_ xmit* function. Since the addresses are local alias addresses, loopback driver places the packet in the incoming queue.

Once again the packet is captured by the NIST Net module and it applies all the effects corresponding to the link between 2 and 3 and hands over the packet to IP protocol handler. The packet is captured by IP-IP-in module and it checks whether the packet is reached final destination. Since the packet reaches the final destination, it removes the extra IP header and returns the packet for further processing. Next application receives the packet at virtual host 3.

NIST Net marks all the packets which are passing through the NIST Net module to avoid infinite loops. There is a temporary control buffer array in the `sk_buff` structure(described in appendix A.5). NIST Net marks some location in the control buffer when it sees a packet. When NIST Net captures a packet, first it checks whether the packet is marked or not. If the packet is marked then it won't apply any effects on that packet, because it has already applied effects on that packet.

Suppose we have three virtual hosts on a physical machine, say 1, 2 and 3. We are sending packets from virtual host 1 to 3 via 2. The link between 1 and 2 has certain characteristics and the link between 2 and 3 has some other characteristics. These are emulated by NIST Net. As NIST Net sees the packet between virtual

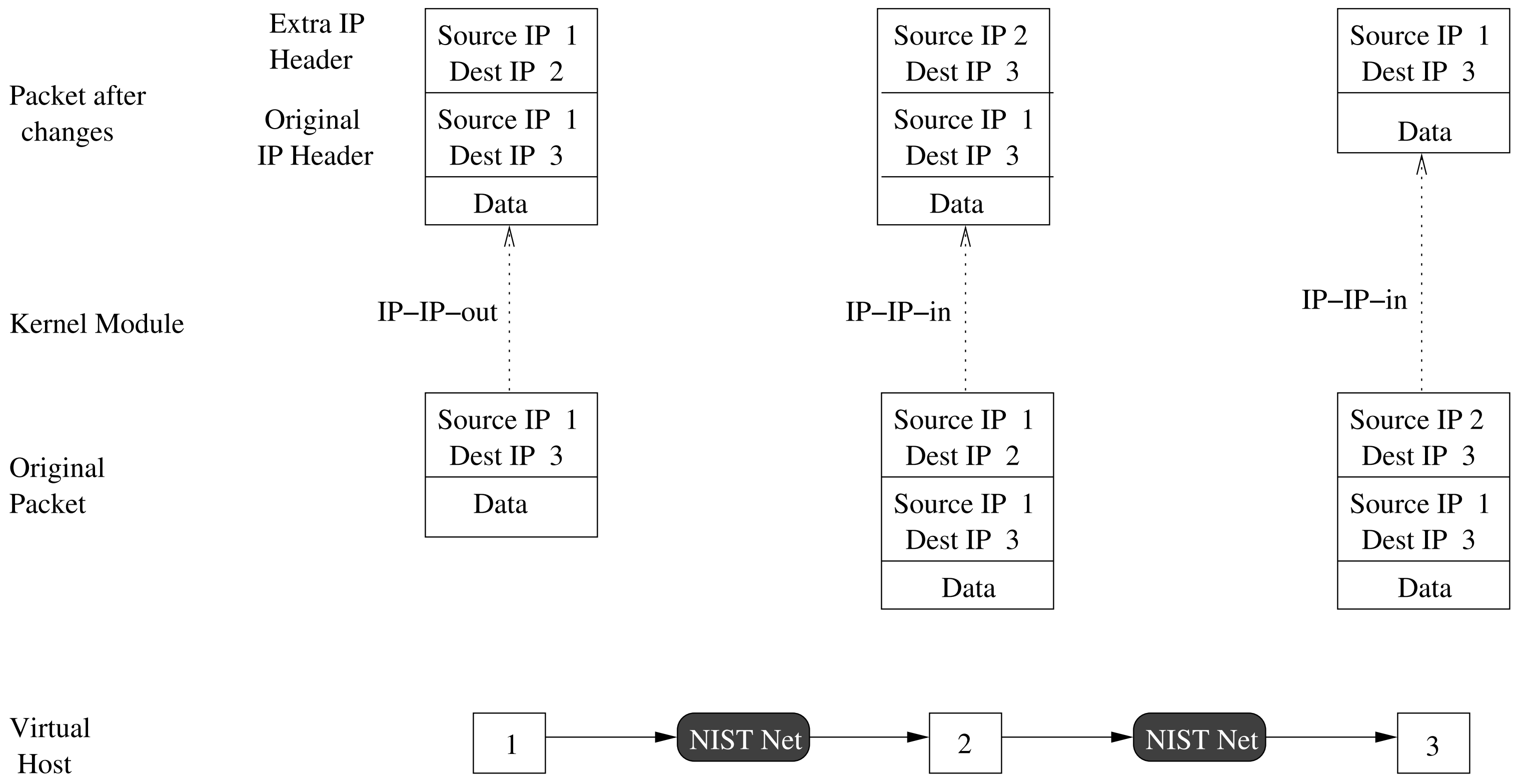


Figure 3.3: An Example for sending a packet from virtual host 1 to 3

hosts 1 and 2, it marks the packet and applies all effects corresponding to that link. Next packet is passed from 2 to 3. NIST Net sees this packet and observes that the packet is marked. So it won't apply any effects on this packet. To overcome this we are removing the NIST Net mark in control buffer of the `sk_buff` in our emulation platform modules.

Suppose we are sending a packet from virtual host 1 to 3 via virtual host 2. Virtual hosts 1 and 2 are on the same machine. Virtual host 3 is on another machine. Since virtual hosts 1 and 2 are on the same machine loopback interface is used in sending the packet from 1 to 2. The MTU of the loopback interface is 16436 bytes whereas Ethernet interface MTU is 1500 bytes. While sending the packet from 1 to 3, first it reaches virtual host 2. Since it reaches virtual host 2 using loopback interface, the packet size is atmost 16436 bytes. After that the packet has to traverse from virtual host 2 to 3. Since the MTU of ethernet packet is 1500 bytes, the packet needs to be fragmented. So to avoid fragmentation, we made loopback packet MTU size to 1500 bytes.

We added extra IP header to the packet. So the packet size is increased by 20 bytes (size of the IP header). For example if the size of the original IP packet is 1500 bytes, after adding the extra IP header its size becomes 1520 bytes. But the ethernet interface can handle only upto 1500 bytes. So the packet needs to be fragmented. To avoid fragmentation, we made the MTUs of both loopback and ethernet interfaces to 1480 bytes. So we can add extra IP header to the packet without fragmentation.

In this section we described the overall design of ScaleNet. In the next section we describe the processing of outgoing packets.

3.2 Processing of Outgoing Packets

Processing of outgoing packets is shown in the Figure 3.4. Outgoing packets are captured at netfilter hook NF_IP_LOCAL_OUT. (Netfilter hooks are described in Appendix A.2.) Packets are captured only if source IP address of the packet belongs to any of the virtual hosts in the emulation platform on our machine. We are not considering the destination IP address because we don't know the IP addresses of virtual nodes on other machines. If the packet doesn't match the selection criteria, NF_ACCEPT is returned. It puts the packet again in the network stack.

`rind_find()` function is used for finding whether the source IP address of the captured packet is any of the virtual hosts on our machine. It searches the list of all IP addresses of virtual hosts on our machine. If it finds the source IP address of the packet, it returns the index of that IP address in the list. Otherwise it returns -2.

Nexthop IP address is found by using the function `route_find()`. This function takes two arguments: the index of the routing table corresponding to the source IP address of the packet which we found earlier and the destination IP address in the host byte order. If the nexthop address is available, this function returns the nexthop address in the network byte order. Otherwise it returns -2 and the packet is dropped by returning NF_DROP.

`CheckNextHop()` function is used to check whether nexthop is on the same machine or on other machine. It takes two arguments: A pointer to skbuffer and nexthop IP address in host-byte order. If the nexthop is on another machine then find the nexthop MAC address and fill the MAC Header details of the packet.

After finding the nexthop IP address, extra IP header is added to the packet i.e. IP-over-IP. Instead of filling all the fields in the extra IP header, fields in the original IP header are copied to the extra IP header and required fields are changed in the extra IP header. The size of the extra IP header is added to the *total length* field in the extra IP header. The destination and source IP addresses are filled with the nexthop IP address and the current virtual host IP address respectively. Since some of the packet fields are changed we need to recalculate the checksum of the IP header. First we need to put zero in the checksum field of the IP header. All the words in the IP header are added and one's complement of the result gives the checksum of the header.

While packet is traversing the network stack, it is placed in the `sk_buff` structure. Refer Appendix A.5 for more details on `sk_buff` structure. Extra IP header is added at the beginning of `sk_buff` structure. If sufficient space is available at the beginning then it is added directly. If space is not available at the beginning

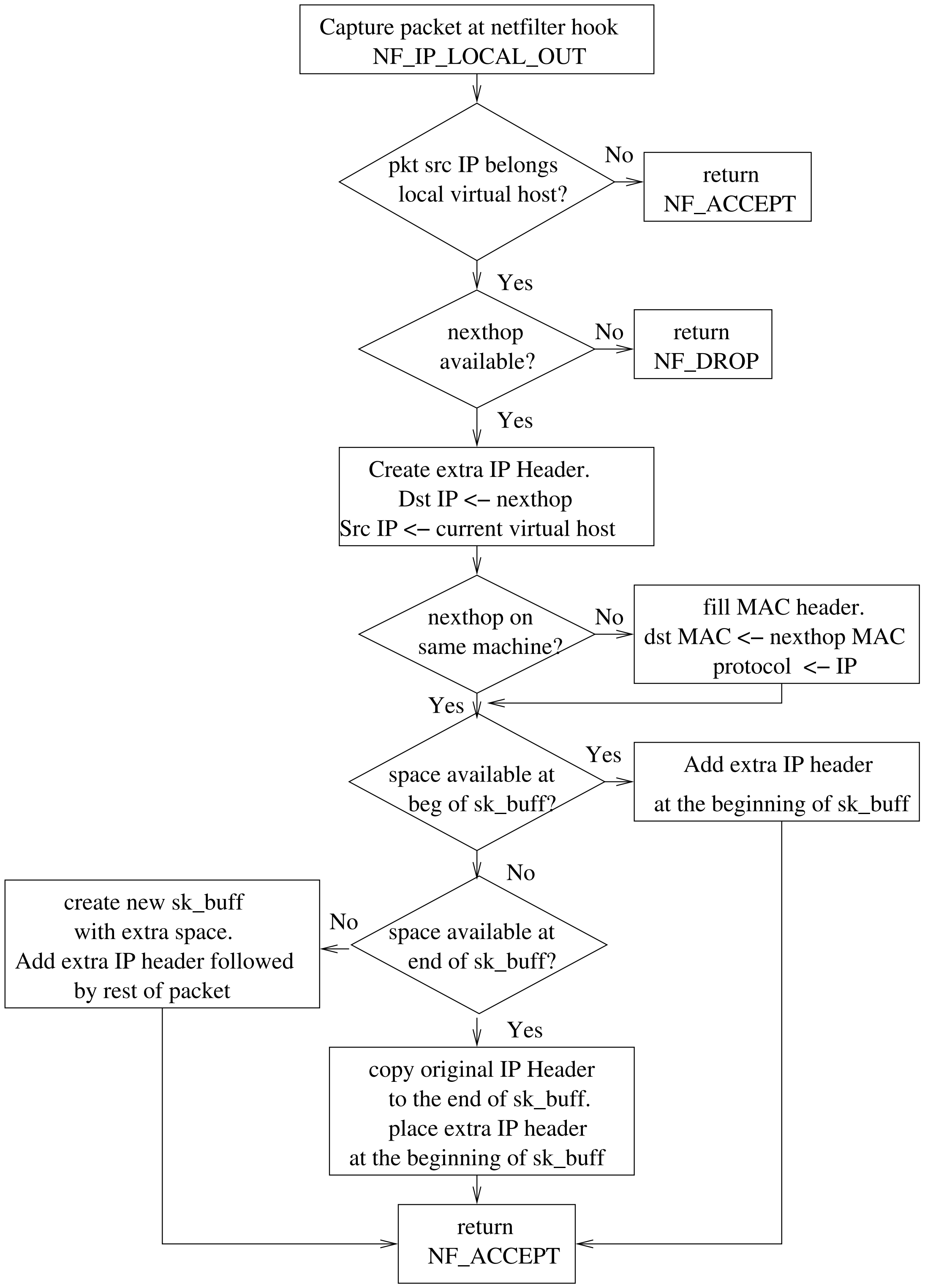

Capture packet at netfilter hook NF_IP_LOCAL_OUT
pkt src IP belongs local virtual host?
No
return NF_ACCEPT
Yes
nexthop available?
No
return NF_DROP
Yes
Create extra IP Header. Dst IP <– nexthop Src IP <– current virtual host
nexthop on same machine?
No
fill MAC header. dst MAC <– nexthop MAC protocol <– IP
Yes
space available at beg of sk_buff?
Yes
Add extra IP header at the beginning of sk_buff
No
space available at end of sk_buff?
No
create new sk_buff with extra space. Add extra IP header followed by rest of packet
Yes
copy original IP Header to the end of sk_buff. place extra IP header at the beginning of sk_buff
return NF_ACCEPT

Figure 3.4: Processing of Outgoing packets

then check at the end of the `sk_buff`. If sufficient space is available at the end of `sk_buff` then the original IP header is copied to the end of `sk_buff` and the extra IP header is added at the beginning. Extra IP header should be added only at the beginning. This is required for NIST Net. If the extra IP header is not added at the beginning then NIST Net won't capture this packet. Finally if there is no space both at the beginning and at the end of the `sk_buff` then new `sk_buff` is allocated with the required size and entire packet is placed inside the new `sk_buff`. New `sk_buff` is allocated by using `dev_alloc_skb(len)` where `len` is the required buffer size in bytes.

If space is available at the beginning of `sk_buff` then the required `sk_buff` pointers are changed by using `skb_push()` function which is described in Appendix A.5. If space is available at the end of `sk_buff` then the required `sk_buff` pointers are changed by using `skb_put()` function. At the time of sending the packet to the final destination, we need to remove the extra IP header and the original IP header should be placed in the right place. First we have to find whether the original IP header is at the beginning or at the end of the `sk_buff`. *protocol* field in the IP header is used for identifying the original IP header position. If the original IP header is at the beginning of `sk_buff` then the *protocol* field contains IPPROTO_IPIP otherwise some new value IPPROTO_EPF.

Finally packet is returned for further processing by returning NF_ACCEPT.

In this section we described the processing of outgoing packets. In the next section we describe the processing of incoming packets.

3.3 Processing of Incoming Packets

Processing of incoming packets is shown in the Figure 3.5. Incoming packets are captured at netfilter hook NF_IP_PRE_ROUTING. If the destination IP address of the outer IP header is one of the IP addresses of the emulation platform on this machine then only packet is processed. Otherwise return NF_ACCEPT.

NIST Net marking is removed from the control buffer `cb[]` (refer AppendixA.5)

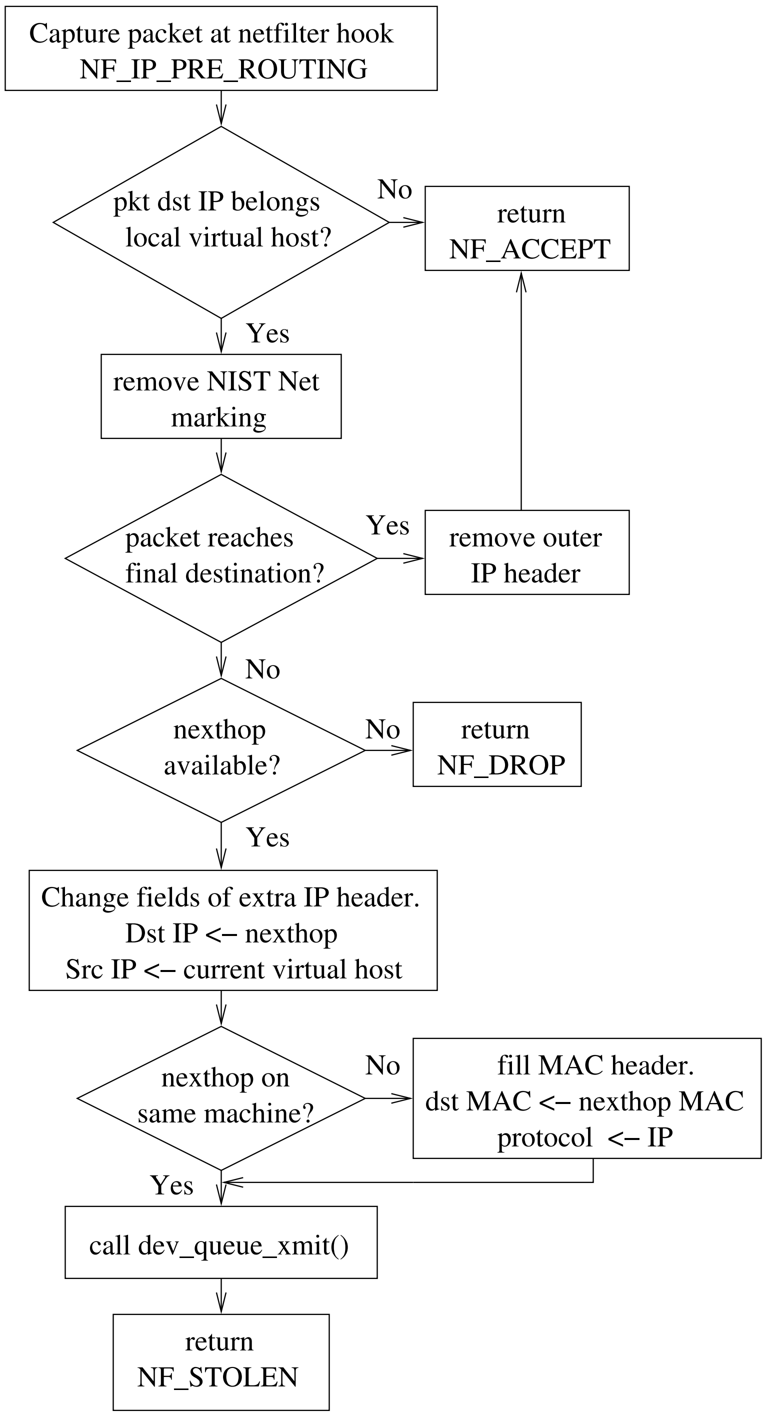

Capture packet at netfilter hook NF_IP_PRE_ROUTING
pkt dst IP belongs local virtual host?
No
return NF_ACCEPT
Yes
remove NIST Net marking
packet reaches final destination?
Yes
remove outer IP header
No
nexthop available?
No
return NF_DROP
Yes
Change fields of extra IP header. Dst IP <– nexthop Src IP <– current virtual host
nexthop on same machine?
No
fill MAC header. dst MAC <– nexthop MAC protocol <– IP
Yes
call dev_queue_xmit()
return NF_STOLEN

Figure 3.5: Processing of Incoming packets

of the `sk_buff`. So next time the packet goes to NIST Net, it applies all the specified effects on that packet.

We have to find the position of inner IP header. I.e. we have to find whether it is at the beginning of `sk_buff` or at the end of `sk_buff`. If the *protocol* field in the outer IP header is `IPPROTO_IPIP` then inner IP header is at the beginning of `sk_buff`. Otherwise it is present at the end of `sk_buff`.

If destination IP address of outer IP header is equal to the destination IP address of the inner IP header then the packet reaches final destination. If the packet reaches final destination then outer IP header of the packet is removed and the packet is sent to upper protocol layers for further processing. While removing the outer IP header, if the inner IP header is at the beginning then just adjust the `sb->data` pointer. If the inner IP header is at the end then adjust the `sb->tail` and copy the inner IP header to the beginning of `sk_buff`. Refer Appendix A.5 for further details on these pointers. Return NF_ACCEPT for further processing of the packet.

If the packet not reaches the final destination then we have to find the nexthop IP address. `route_find()` function finds the nexthop IP address using the routing table of the current virtual host. Instead of creating new IP header, we are changing some fields in the outer IP header. As earlier destination IP address of the outer IP header is changed to the nexthop IP address and source IP address is changed to the IP address of the current virtual host. `Checksum` field is made zero and new checksum of the IP header is calculated by using `checksum()` function.

`CheckNextHop()` function is used to find whether nexthop is on the same machine or on other machine. If the nexthop is on another machine then find the nexthop MAC address and fill the MAC header details of the packet.

Packet type is changed to outgoing packet and is sent by calling `dev_queue_xmit()` function. If this function is not called then the packet is directly delivered to the local interface. Next `NF_STOLEN` is returned to the netfilter hook. It tells netfilter that the hook function will take processing of the packet from here on and netfilter should drop its processing completely.

In this section we described the processing of incoming packets. In the next section we describe the creation of the network topology and accessing and modification of the routing tables.

3.4 Network Topology and Routing

During initialization of the emulation platform all the routing tables, IP addresses of virtual hosts, nexthop IP addresses which lie on other machine are read from userspace to the kernel. These are written to the `/proc/rtable` file by a user-level program. Call-back function `proc_write_rtable()` is called when anything is written to the `/proc/rtable` file.

Each routing table entry consists of three fields: Mask field, IP address of a Host or Network and the nexthop IP address. In searching the routing table, the destination IP address is ANDed with the mask and the result is compared with the second field. If they match then the third field is the nexthop IP address for the given destination. All the entries of the routing table are searched until any matching entry is found. There are NUM_NODES routing tables each containing at most NUM_ENTRIES entries. A chunk of memory is allocated for each routing table.

`proc_write_rtable()` function first reads the number of virtual nodes (NUM_NODES), maximum number of routing table entries (NUM_ENTRIES) in each routing table, number of IP addresses (NUM_IP_ADDRS) and number of nexthop IP addresses (NUM_NHOP_MAC) which lie on other physical machines. Both the number of nodes and number IP addresses are read because a router can have several IP addresses. Then it allocates (#virtual nodes * Max Entries per routing table * 3 * sizeof(unsigned int)) bytes of memory for routing tables. A number is associated with IP address and all IP addresses of a router have same number. Memory is allocated for IP addresses and their corresponding index. Similarly memory is allocated for nexthop IP addresses and their corresponding MAC addresses.

`proc_write_rtable()` reads the nexthop IP address and corresponding MAC address. Next it reads the IP addresses and their corresponding indices. Finally it reads routing tables one by one. After reading these it exports these to other parts of the kernel.

Routing tables can be accessed or updated using *ioctl* interface. The `ioctl's` are described in Appendix A.4. The `ioctl` call has three arguments: The file descriptor for the file `/proc/char_dev`, the `ioctl` number which specifies the type of the command and the *argument* to the `ioctl` command.

Three `ioctl` calls are supported. First one is for adding a routing table entry to the specified routing table. For this, the `ioctl` number is IOCTL_ADD_RT_ENTRY. These variables are defined in `chardev.h` file which should be included by both kernel module and the userlevel program. The argument for this call is a pointer to a character. It contains the routing table number and the routing table entry. It is terminated by a NULL character. Second `ioctl` call is for removing a routing table entry from the specified routing table. The `ioctl` number for this call is IOCTL_REM_RT_ENTRY. The argument to this `ioctl` command is also a pointer to a character. The data contains the routing table number, routing table entry to be deleted and is terminated by a NULL character. Third `ioctl` call is for showing routing table entries. The `ioctl` number is IOCTL_SHOW_RT_ENTRY. It takes an integer argument which specifies the routing table to be displayed.

In this section we described the creation of the network topology. In the next section we describe how to associate an application with a virtual host and redirection of system calls.

3.5 Virtual Hosts

Several IP aliases are created on each machine using `ifconfig` command. For example an IP alias 10.0.0.1 to `eth0` Ethernet interface is created as follows.

```
#ifconfig eth0:1 10.0.0.1
```

3.5.1 Association between Applications and Virtual Hosts

We want to associate an IP address with the application program. If an application program issues system calls like `route add`, this acts on the system routing table. But we want to change this behaviour such that this system call acts on the routing table corresponding to a virtual host.

We associate application programs with an IP address of a virtual host as follows. A wrapper program forks and executes all the application programs belong to a virtual host. This wrapper program acts just like a shell. It takes the application to be executed and its arguments and calls *execve* to execute that application. After this application finishes its execution, it waits for executing another application. All the applications executed in this wrapper program correspond to the virtual host that is associated with this wrapper program.

The `PID` of the wrapper program is associated with the virtual host IP address. *pid_ ip* module handles this association. Each process has a *task_ structure*, which maintains all the information about that process. It also has pointer to the parent process of this process. So we can traverse the current process parent, grand-parent etc. If this process is executed in a wrapper program, while traversing the ancestor processes, wrapper program is reached. Otherwise *init* process is reached. If a wrapper program is reached, this process corresponds to the virtual host which is associated with the wrapper program.

The `pid_ip` module also provides an `ioctl` interface for accessing and modifying the *pid-ip* association. Three `ioctl` commands are supported. IOCTL_ADD_PIDIP command is used for adding pid-ip entry. It takes a pointer which points to the pid-ip entry that is to be added. IOCTL_REM_PIDIP command is used for removing pid-ip entry. It takes a pointer to the pid-ip entry that is to be removed. IOCTL_SHOW_PIDIP command is used for showing all pid-ip values.

3.5.2 System Call Redirection

sys_ call_ table[1] is an array of pointers to the system call functions. For hacking the system calls, the pointer to the system call function which we want to hack is replaced with our own function handler. This function will do whatever it wants and if required calls the original system call function.

[1]This symbol is exported by the kernel. If it is not exported then export it in the file /usr/src/linux/kernel/ksyms.c. After exporting that, rebuild and reboot the kernel.

bind and *route* system calls are hacked. In Linux, all the socket related system calls are multiplexed using *socketcall* system call. *socketcall* system call handling function pointer is replaced with our own function handler. In this function, if this call is not *bind* system call then original system call handler function is called. Otherwise whether this process belongs to any virtual host in the emulation platform is determined. If so then it is binded to that virtual host IP address. Otherwise original system call handler function is called.

ioctl system call is hacked for *route* command. In this, if this *ioctl* call is not *route* command or this process does not belongs to any of the virtual host in the emulation platform then original system call handler function is called. Otherwise the *route* command is directed to act on the routing table belongs to the virtual host that is associated with the current process. In this way application programs can directly manipulate the routing tables corresponding to the virtual hosts in the emulation platform.

Chapter 4

Experimental Results

In this chapter we describe various tests performed and the problems encountered.

Since NIST Net is used for applying effects such as delay, bandwidth limitation, packet drops, packet duplication etc on the packets passing through the machine on which NIST Net is running, we have to find out how NIST Net performs in applying these effects.

NIST Net has some problems in applying the bandwidth limitation in the case of client and server running on the same machine for TCP packets. There is also another problem in case of sending UDP packets continuously.

NIST Net packet duplication and packet drop features are verified using the *ping* command. Expected results are coming. NIST Net bandwidth test results are given in the Section 4.1.

After that we have to find out how the emulation platform scales up. We created 20, 50 virtual hosts per physical machine and performed several tests in both cases using TCP and UDP packets. Bandwidth and delay test results for the emulation platform in the case of 20 virtual hosts per physical machine are given in the Section 4.2 and the results for 50 virtual hosts per physical machine are given in the Section 4.3.

There are several tuning parameters. By setting these values appropriately, we can increase the performance of the system. These are described in the Section 4.4.

4.1 NIST Net Bandwidth Tests

Netperf[9], bw_tcp[10] and ScaleNetTest programs are used for doing bandwidth limitation tests. ScaleNetTest programs are set of socket programs used for doing bandwidth limitation tests. In all the cases similar results are coming. In this section we describe the results obtained using ScaleNetTest programs only. We can bind to the particular virtual host using ScaleNetTest programs. Statistics are also printed at user specified intervals using these programs.

Tests are performed on both TCP and UDP packets. Both TCP and UDP are used because of the different characteristics of the protocols. In TCP case, sender and receiver agree on the maximum segment size, receive window size etc. TCP adjusts its behaviour according to the available bandwidth, congestion in the route between source and destination and RTT. Where as in the UDP case, there is no flow control between the sender and receiver. It won't consider the congestion, available bandwidth etc.

4.1.1 TCP Packets

When the server and client programs are running on different machines and NIST Net is running on the server machine, we are getting expected results.

In this experiment, client and server are running on the same machine. These results are shown in the Table 4.1. Tests are performed by varying the number of packets that are sent from client to server. 50 packets, 100 packets and continuous packet streams, each packet of size 1000 bytes, are sent from client to server and the throughput is calculated in each case. To know how the NIST Net behaves for smaller number of packets, tests are performed for 50 and 100 packets.

In the case of sending fixed number of packets from client to server, server program notes the time before receiving the packets. After receiving all the packets, it notes the time and calculates the throughput. In the case of sending packets continuously, server sets an alarm clock for delivery of a SIGALRM signal at the specified number of seconds. In the signal handler function, it calculates the throughput for

Bandwidth applied using NIST Net (Bytes/sec)	Throughput obtained using 50 packets (Bytes/sec)	Throughput obtained using 100 packets (Bytes/sec)	Throughput:sending packets continuously (Bytes/sec)
1000	1468	1001.29	-
2000	2935	2001	-
4000	5868	4000	3276
5000	7335	5001	-
6000	8802	6001	-
8000	11903	8000.1	(6553,9830)
10000	14931	10002	9830
20000	29000	20136	19661
40000	(50000, 60000)	40536	49152
100000	(139000, 174000)	105312	-
131072	249036.8	146800.64	133693.44
262144	(655360, 724828.16)	(314572.8, 340787.2)	271319.04
524288	(5767168, 6946816)	(655360, 1966080)	737935.36
1048576	(5636096, 7864320)	(5242880, 6553600)	4949278.72

Table 4.1: Bandwidth tests using TCP packets and both client, server are running on the same machine.

that duration and sets the alarm clock again. Like this, it prints the throughput at fixed intervals.

In some cases throughput is varying highly. In those cases we put a dash for that value. For lower values of applied bandwidth, we are getting some reasonable values. But for higher values, we are getting higher bandwidth than the applied bandwidth using NIST Net. When packets are sent continuously, for 8 Mbps applied bandwidth throughput is 37.76 Mbps. It is not clear why this is happening, but when the MTU of loopback packets is changed expected results are coming.

Default value of the MTU of loopback packet is 16436 bytes and that of Ethernet packet is 1500 bytes. We changed the MTU of loopback packets to 1480 bytes using the command

```
#ifconfig lo mtu 1480
```

After changing the MTU, we are getting expected results. In this experiment

Bandwidth applied using NIST Net (Bytes/sec)	Throughput obtained using 100 packets (Bytes/sec)	Throughput: Sending packets continuously (Bytes/sec)
1000	844.2	925
2000	1915.86	1856
4000	3819.93	3855
8000	7672.51	7568
16000	15337.93	15200
32000	30575.38	30272
50000	47225.35	47553
64000	59969.73	60900
100000	93067.76	95170
131072	127139.84	130809.856
262144	243793.92	261619.712
524288	≈342097.92	522977.28
1048576	≈343408.64	1045954.56
2097152	≈933232.64	2093219.84
4194304	-	4170711.04
8388608	≈1588592.64	4187750.4
13107200	-	-
No Bandwidth limit	-	4194304
Emulator Off	-	≈4369940.48

Table 4.2: Bandwidth tests using TCP packets. Both client and server are running on the same machine(PIII). The MTU of loopback packets is changed to 1480 Bytes.

both server and client are running on the same machine and NIST Net is running on the server machine. Bandwidth tests are done in two cases: sending 100 packets and continuously sending packets. Each packet contains 1000 Bytes. The programs used for these tests are same as those in the previous case. The configuration of machine used is Pentium III 997 MHz processor, 256 MB RAM and 20 GB HDD. These results are shown in the Table 4.2.

In the case of sending 100 packets from client to server, each packet of size 1000 bytes, for higher bandwidths the throughput is saturated because of TCP slow start. In the case of sending packets continuously, the throughput is saturated around 32 Mbps.

Bandwidth applied using NIST Net (Bytes/sec)	Throughput obtained using 100 Packets (Bytes/sec)	Throughput: Sending packets continuously (Bytes/sec)
1000	866	925
2000	1915.81	1856
4000	3818.88	3855
8000	7637.42	7568
16000	15275.6	15200
32000	30551.96	30416
50000	47799.21	47553
64000	60784.44	60900
100000	96023.85	95170
131072	131334.144	130809.856
262144	261488.64	261619.712
524288	507248.64	522977.28
1048576	983040	1045954.56
2097152	1867776	2093219.84
4194304	3377725.44	4182507.52
8388608	4909957.12	8365015.04
13107200	$\approx$4978114.56	9749135.36
No Bandwidth limit	-	10040115.2
Emulator Off	-	10261626.88

Table 4.3: Bandwidth tests using TCP packets. Both client and server are running on the same machine(PIV). The MTU of loopback packets is changed to 1480 Bytes.

Same test is conducted on a machine with configuration Pentium IV 3 GHz processor, 2 GB RAM and 80 GB HDD. Tests are conducted on both low and high configuration machines to know the NIST Net behaviour on low and high configuration machines. The programs used for this test are same as in the previous case. The results for the second case are shown in the Table 4.3.

As in the previous case, in the case of sending 100 packets, the throughput is saturated around 37 Mbps. Values are varying very much in the case of no bandwidth limitation applied and emulator is off. So we put a dash in those cases. In the case of sending packets continuously results are coming as expected.

Bandwidth applied using NIST Net (Bytes/Sec)	Throughput (Bytes/sec)
1000	972
2000	1945
4000	3891
8000	7786
16000	15562
20000	19461

Table 4.4: Bandwidth tests using UDP packets. Both client and server are running on the same machine. We are sending 50 packets from client to server each of size 1000 bytes.

4.1.2 UDP Packets

For UDP, there is no flow control and congestion control. The client program sends all packets at once. In several cases packets are dropped due to unavailability of buffer space at the receiver.

We have done tests with UDP packets in two cases. In first case both client and server are on the same machine and in second case client and server are running on different machines. Tests are done in two cases because in the first case only loopback interface is used where as in the second case both loopback and Ethernet interface are used.

▮ *Client and server are on the same machine*

Throughput is calculated by sending 50 packets, each of size 1000 bytes, from client to server. The client program sends the specified number of packets to the server. The server program notes the time at the beginning and end of the data transfers and calculates the throughput. It also prints the throughput after receiving the specified number of bytes.

The results are shown in the Table 4.4. Expected results are coming.

In the case of sending 17300 packets, each packet of size 1000 bytes, from client

Bandwidth applied using NIST Net (Bytes/sec)	Throughput (Bytes/sec) (For every 100 pkts received)
10000	22109
10000	9824
10000	9825
10000	9825
:	:
:	:

Table 4.5: Bandwidth tests using UDP packets. Both client and server are running on the same machine. We are sending 17400 packets. Throughput is calculated for every 100 packets.

to server expected results are coming. The programs used for this test are same as in the previous case. By using NIST Net, 20000 bytes/sec bandwidth limitation is applied. We are getting throughput 19650 Bytes/sec.

Next 17400 packets are sent from client to server, each packet of size 1000 Bytes. Expected results are not coming in this case. These results are shown in the Table 4.5. Here the throughput is calculated for every 100 packets received.

NIST Net is queueing atmost 17344 packets, each packet of size 1000 bytes, at any point of time. It is allocating fixed amount of memory. After the allocated memory is over, it is not queueing the packets. If it receives more than 17344 packets then it is sending these excess packets without applying any effects. So in the first 100 packets received throughput is 22109 bytes/sec where as applied bandwidth is 10000 bytes/sec.

Next 18300 packets are sent from client to server, each packet of size 1000 bytes. In this case also NIST Net is passing 756 excess packets without applying any effects. The results are shown in the Table 4.6. Throughput is calculated for every 100 packets received.

Next packets are sent continuously. 20000 bytes/sec bandwidth limitation is applied using NIST Net. We are getting throughput around $(55959709, 68027210)$ bytes/sec. In this case NIST Net is queueing 17344 packets. After that it is leaving

Bandwidth applied using NIST Net (Bytes/sec)	Throughput(Bytes/sec) (For every 100 pkts received)
20000	1032663
20000	48551
20000	19650
20000	19649
20000	19649
:	:
:	:

Table 4.6: Bandwidth tests using UDP packets. Both client and server are running on the same machine. We are sending 18300 packets. Throughput is calculated for every 100 packets.

the packets without applying any effects. So we are getting excess bandwidth than the applied bandwidth.

▮ *Client and server are on different machines*

In the case of client and server are running on different machines, if the number of packets that are sent from client to server are less than 17344, expected results are coming. If packets exceed this, NIST Net is not queueing excess packets.

17400 packets are sent from client to server, each packet of size 1000 bytes. The programs used for this test are same as in the previous case. The results are shown in the Table 4.7. Here the throughput is calculated for every 100 packets received.

In this case also, NIST Net is queueing atmost 17344 packets. After that it is leaving the packets without applying any effects. That's why in the first 100 packets received, we are getting 26649 bytes/sec.

Next packets are sent continuously from client to the server. By using NIST Net, 20000 bytes/sec bandwidth limitation is applied. We are getting throughput around $11,840,000$ bytes/sec.

If packets are sent at constant bit rate, which is slightly higher than the bandwidth applied, then there are no packet drops until the queue size reaches 17344

Bandwidth applied using NIST Net (Bytes/sec)	Throughput(Bytes/sec) (For every 100 pkts received)
20000	26649
20000	19650
20000	19649
20000	19649

Table 4.7: Bandwidth tests using UDP packets. Client and Server are running on the different machines. We are sending 17400 packets. Throughput is calculated for every 100 packets.

packets, each packet of size 1000 bytes/sec.

In this section we discussed the NIST Net behaviour for both TCP and UDP packets. In the next section scalability tests for the emulation platform are described.

4.2 Creating 20 Virtual Hosts per System

We created a network topology consisting of 40 nodes. We created this topology on two machines. Each machine has 20 virtual hosts. This topology is shown in the Figure 4.1. The reason for choosing this network topology is each machine in the emulation platform has several virtual hosts which are connected through loopback interface and there are limited number of physical machines which are connected through ethernet interface. The topology in the Figure 4.1 resembles these aspects. In the case of sending a packet from 10.0.1.1 to 10.0.4.10, the packet traverses the virtual hosts on the machine 1 through loopback interface and goes to machine 2 using ethernet interface and traverses the virtual hosts on the machine 2 through loopback interface and comes again to machine 1 using ethernet interface and traverses the virtual hosts on machine 1 and goes to machine 2.

Several tests are performed by varying the bandwidth and delay values as in the previous section to see how the emulation platform performs with many virtual hosts per physical machine. Both TCP and UDP protocols are used because of their different characteristics. In each test we used 40000 packets. In all the cases

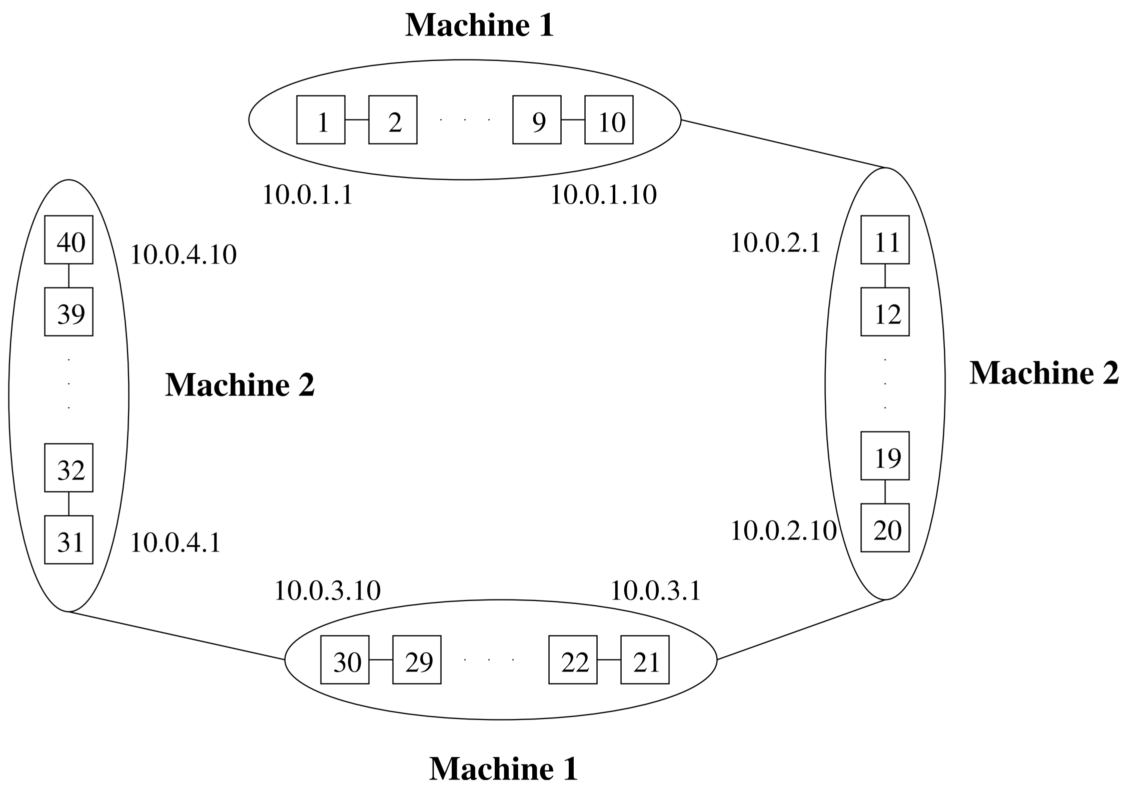


Figure 4.1: A network topology consisting of 20 nodes per system.

expected behaviour is coming.

40000 TCP packets are sent from 10.0.1.1 to 10.0.4.10. Here source IP lies on the machine 1 and destination IP lies on machine 2. Various bandwidth limitations are applied using NIST Net. We pick two delay values 5, 10ms and calculated the throughput in the two cases. We pick two delay values to know how the NIST Net performs at various delay values. In this experiment each link has 10ms delay.

In this experiment, the client TCP program sends 40000 packets to the server. The server program calculates the throughput(number of bytes received / total time) after receiving the specified number of packets.

The results are shown in the Table 4.8. From the table, the maximum possible throughput is around 154000 bytes/sec. The window size of TCP is 65535 bytes. The TCP receiver can receive 65535 bytes in RTT. In the topology shown in the

Bandwidth applied using NIST Net (Bytes/Sec)	Throughput (Bytes/sec)
4096	3892
8192	7784
16384	15572
32768	31136
65536	62222
131072	123933
262144	154125
393216	154361

Table 4.8: Sending 40000 TCP packets from 10.0.1.1 to 10.0.4.10. For each link 10ms Delay is applied.

Figure 4.2, each link between 10.0.1.1 and 10.0.4.10 has a delay of 10ms. In the backward direction, i.e. from 10.0.4.10 to 10.0.1.1, we haven't applied any delay. For example, link between 10.0.1.1 and 10.0.1.2 has 10ms delay, whereas link between 10.0.1.2 and 10.0.1.1 has no delay. There are 39 links between 10.0.1.1 and 10.0.4.10. So the total propagation delay is 390ms. For 100Mbps link, the transmit time for 65535 bytes is around 5ms. In the direction from 10.0.4.10 to 10.0.1.1, there is no propagation delay. So RTT is 395ms. So the maximum possible data transferred per unit time is (65535bytes/395ms), which is 165911 bytes/sec. Excluding the headers(each packet has 20 bytes TCP header, 20 bytes IP header and 20 bytes extra IP header) we are getting around 154000 bytes/sec (from Table 4.8). If we add all headers size(154 headers each of size 60bytes i.e. 9240bytes), it is 154000+9240 = 163240 bytes/sec. So we are getting expected results.

Next 40000 UDP packets are sent from 10.0.1.1 to 10.0.4.10. Each link has 10ms delay. If we are sending all the packets at the maximum possible rate then NIST Net queue overflows and it won't apply the effects on these excess packets. So client sends packets at constant rate to server, which is taken as an input from the user. The server program receives all the packets and calculates the throughput.

The bandwidth limitation applied, throughput, client data sending rate are shown in the Table 4.9.

Bandwidth applied using NIST Net (Bytes/sec)	Throughput (Bytes/sec)	Client send rate(Bytes/sec)
2048	1954	2500
4096	3908	4500
8192	7816	8500
16384	15634	16500
32768	31268	33000
65536	62537	66000
131072	125069	132000
262144	250146	262500
393216	375278	393500
524288	500321	524500
655360	625504	655500
786432	750347	786500
1048576	1001276	1048600
1179648	1126465	1180000

Table 4.9: Sending 40000 UDP packets from 10.0.1.1 to 10.0.4.10. For each link 10ms Delay is applied.

In the case of UDP packets, there is no flow control. So the client program sends the packets continuously. So there is no saturation in the throughput value. So, expected results are coming in this case.

Next each link delay is changed to 5 ms and 40000 TCP packets are sent from 10.0.1.1 to 10.0.4.10. The programs used for this test are same as in the previous TCP case.

The results are shown in the Table 4.10. In the case of 10ms link delay, maximum throughput is 153000 bytes/sec. Since throughput is inversely proportional to the delay, throughput is doubled to 305366 Bytes/sec as the delay is halved to 5ms. So, expected results are coming.

Next 40000 UDP packets are sent from 10.0.1.1 to 10.0.4.10. The programs used for this test are same as in the previous UDP test. The results are shown in the Table 4.11.

For 9 Mbps (1179648 bytes/sec) bandwidth limit applied, client sends at the

Bandwidth applied using NIST Net (Bytes/sec)	Throughput (Bytes/sec)
4096	3892
8192	7784
16384	15572
32768	31141.5
65536	62262.79
131072	124395
262144	248082
393216	305150.5
524288	305366.67

Table 4.10: Sending 40000 TCP packets from 10.0.1.1 to 10.0.4.10. For each link 5ms Delay is applied.

rate 1180000 bytes/sec. In this case not all the packets reached destination. Each machine has a buffer for received packets. If there is no space in the buffer, it drops the packets. In 9 Mbps case, this buffer might be full and it dropped some of the packets. So some packets not reached the application at the destination. This won't happen in TCP case.

4.3 Creating 50 Virtual Hosts per System

We created another topology, similar to the previous 40 node topology, consisting of 100 nodes distributed on two machines. Each machine has 50 nodes. The topology of the network is shown in the Figure 4.2.

Tests similar to the previous section are performed in this section. Expected results are coming.

In all the experiments in this section, same programs as in the previous section are used. As in the previous section both TCP and UDP packets are used for doing the tests. Both are used because both have different characteristics. The tests are performed by varying the bandwidth limitation and delay. We pick 5ms and 10ms delay values for the experiments. Several bandwidth limitations are applied.

Bandwidth applied using NIST Net (Bytes/sec)	Throughput (Bytes/sec)	Client send rate(Bytes/sec)
2048	1954	2500
4096	3908	4500
8192	7816	8500
16384	15633	16500
32768	31268	33000
65536	62534	66000
131072	125064	132000
262144	250140	262500
393216	375260	393500
524288	500293	524500
655360	625454	655500
786432	750276	786500
1048576	1001152	1048600
1179648	Not all pkts reached destination	1180000

Table 4.11: Sending 40000 UDP packets from 10.0.1.1 to 10.0.4.10. For each link 5ms Delay is applied.

First 40000 TCP packets are sent from 10.0.1.1 to 10.0.4.25. Each packet has 1000 bytes. Source IP lies on first machine and destination IP lies on second machine. Each link has 10 ms delay.

The results are shown in the Table 4.12. The maximum possible throughput is 62013 bytes/sec. The reason for this saturation is same as that in the previous section.

Next 40000 UDP packets, each packet of size 1000 bytes, are sent from 10.0.1.1 to 10.0.4.25. The programs used for this test are same as those in the previous case. Each link has 10ms delay. The bandwidth limitation applied, throughput and client data sending rate are shown in the Table 4.13.

Except in the 8 Mbps case, expected results are coming. For 8 Mbps, not all the packets reached destination. The receive buffer at the destination likely dropped some of the packets due to unavailability of space.

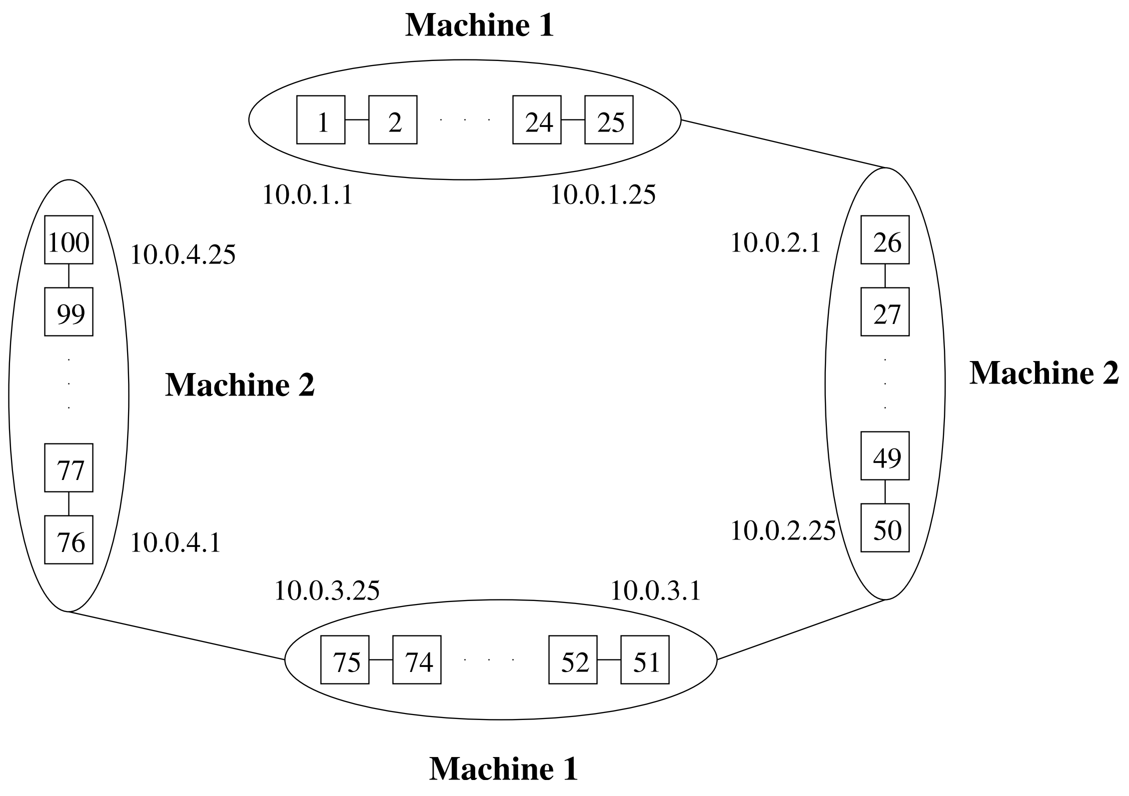


Figure 4.2: A network topology consisting of 50 nodes per system

Next 40000 TCP packets, each packet of size 1000 bytes, are sent from 10.0.1.1 to 10.0.4.25. Each link has 5 ms delay. The programs used for this test are same as those used in the previous TCP case.

The results are shown in the Table 4.14. The maximum possible throughput is around 122270 bytes/sec. Observe that in the case of 10ms delay, maximum throughput is 62000 bytes/sec. Since the delay is halved to 5 ms, the maximum throughput is increased to 122270 bytes/sec. Expected results are coming in this case.

Finally 40000 UDP packets, each packet of size 1000 bytes, are sent from 10.0.1.1 to 10.0.4.25. The programs used for this test are same as those in the previous UDP case.

The results are shown in the Table 4.15. Except for 8 Mbps, expected results are

Bandwidth applied using NIST Net (Bytes/sec)	Throughput (Bytes/sec)
4096	3892
8192	7782
16384	15561
32768	31079
65536	61024
131072	61921
262144	61995
393216	62011
524288	62013

Table 4.12: Sending 40000 TCP packets from 10.0.1.1 to 10.0.4.25. For each link 10ms Delay is applied.

coming. For 8 Mbps, not all packets reached destination. The receive buffer might be full, so some of the packets are dropped.

4.4 Some Tuning Parameters

There are several tuning parameters which can affect the performance. These are described below.

txqueuelen Size of the queue between the kernel subsystem and the driver for NIC. If this queue is full, all the packets coming from the kernel subsystem are dropped. So the application has to retransmit the packets. If this is set to some high value, these droppings won't occur.

netdev_max_backlog Maximum number of packets queued on the INPUT side, when the interface receives packets faster than kernel can process them. If this limit is reached, all the packets coming from the interface are dropped. If this is set to some high value, these droppings won't occur.

rmem_default The default setting of the socket receive buffer in bytes. If this buffer is full, all the incoming packets are dropped. This value can be increased

Bandwidth applied using NIST Net (Bytes/sec)	Throughput (Bytes/sec)	Client send rate(Bytes/sec)
4096	3908	4500
8192	7816	8500
16384	15634	16500
32768	31268	33000
65536	62537	66000
131072	125069	131500
262144	250146	262500
393216	375279	393500
524288	500325	524500
786432	750346	786500
917504	875913	918000
1048576	Not all packets reached destination	1049000

Table 4.13: Sending 40000 UDP packets from 10.0.1.1 to 10.0.4.25. For each link 10ms Delay is applied.

by user upto a maximum value of rmem_max.

rmem_max The maximum receive socket buffer size in bytes.

wmem_default The default setting of the socket send buffer size in bytes. If this buffer is full, the *send*, *write* etc commands will block until the space is available. This can be increased upto wmem_max.

wmem_max The maximum send socket buffer size in bytes.

The commands used for setting the above parameters are shown below.

- `/sbin/ifconfig eth0 txqueuelen VALUE`; The default value is 100.
- `/sbin/sysctl -w net.core.netdev_max_backlog=VALUE`; The default value is 300.
- `/sbin/sysctl -w net.core.rmem_max=VALUE`; Default value is 131071 bytes.

Bandwidth applied using NIST Net (Bytes/sec)	Throughput (Bytes/sec)
4096	3892
8192	7784
16384	15569
32768	31122
65536	62158
131072	120066
262144	122024
393216	122155
524288	122270

Table 4.14: Sending 40000 TCP packets from 10.0.1.1 to 10.0.4.25. For each link 5ms Delay is applied.

Bandwidth applied using NIST Net (Bytes/sec)	Throughput (Bytes/sec)	Client send rate(Bytes/sec)
4096	3908	4500
8192	7816	8500
16384	15633	16500
32768	31267	33000
65536	62537	66000
131072	125066	131500
262144	250138	262500
393216	375259	393500
524288	500293	524500
786432	750277	786500
917504	875775	918000
1048576	Not all packets reached destination	1049000

Table 4.15: Sending 40000 UDP packets from 10.0.1.1 to 10.0.4.25. For each link 5ms Delay is applied.

- `/sbin/sysctl -w net.core.rmem_default=VLAUE`; Default value is 65535 bytes.
- `/sbin/sysctl -w net.core.wmem_max=VALUE`; Default value is 131071 bytes.
- `/sbin/sysctl -w net.core.wmem_default=VALUE`; Default value is 65535 bytes.

While doing the tests, we used all default parameters.

Chapter 5

Conclusions and Future Work

In this work, we have created an emulation platform which emulates large-scale networks using few physical resources. Several virtual hosts are created in each physical machine and applications are associated with virtual hosts. There is no need to modify application programs code for running on the emulation platform. With this emulation platform any kind of network protocol may be tested. Performance analysis and debugging can be done using this emulation platform. In [11], they did simulation of BGP using 11806 AS nodes. In our emulation platform, this can be done by using 240 systems. Similarly OSPF protocol and peer-to-peer networks can be studied using this emulation platform.

There are several aspects of the work which can be done in the future.

- Automatic mapping of user specified topology to the physical resources has to be done and routing tables have to be generated from the topology.
- We need to provide a graphical user interface.
- We redirected `bind` and `route` system calls. All other system calls which should act on the current virtual host rather than on the physical machine needs to be identified and we have to redirect them.
- We have to implement locking of the shared data structures.

- While sending packet from one machine to other machine we are changing MAC header details. This should be avoided.
- Packet is not reaching the application. For this we have captured `dst_entry` object and used it in the `sk_buff` of the packet. After that packet is reaching the application. We have to avoid capturing the `dst_entry` object and needs to investigate why the packet is not reaching the application.
- During initialization of the emulation platform sometimes the system is crashing. We have to investigate this.
- We have done experiments with only one TCP flow. We have to do with more than one TCP flow.
- We have to run some networking protocol on this emulation platform and needs to analyze the memory and processing requirements at each virtual host. Memory is the primary bottleneck since we have to store all the data of all the virtual hosts.

Appendix A

Linux Kernel Programming

A.1 Linux Kernel Modules

Linux Kernel Modules are pieces of code that can be loaded and unloaded dynamically. Modules have several advantages. We do not need to include each and every functionality in the kernel image, so that its size can be reduced. If the modules facility is not there then we have to rebuild and reboot the kernel every time we need new functionality.

Kernel modules must implement at least two functions: *Initialization* function and *cleanup* function. Initialization function is `init_module()`, which is called when the module is inserted into the kernel, and cleanup function is `cleanup_module()`, which is called just before the module is removed from the kernel.

Consider the following simple kernel module.

```
#define MODULE
#define __KERNEL__

#include<linux/module.h>
#include<linux/kernel.h>
```

```
int init_module(void)
{
        printk("In the initialization routine.\n");

        return 0;
}

void cleanup_module(void)
{
        printk("In the cleanup routine.\n");
}
```

Observe that here we are using `printk()` instead of `printf()`. `printk()` is used to log messages or to give warnings by the kernel. If you are in the *text* mode then these will also be printed on the console. Generally these messages are logged in the file `/var/log/messages`. The symbol __KERNEL__ means that the code will be run in the kernel mode. The symbol __MODULE__ means that this is a kernel module. For compiling this module use the following command.

```
gcc -c hello.c -o hello.o -I /usr/src/linux/include
```

It produces an *object* file rather than *executable* file by using the *-c* flag. This object file will be linked dynamically with the kernel when the module is inserted into the kernel. The `-I dir` means add the directory `dir` to the list of directories to be searched for header files. Directories named by -I are searched before the standard system include directories. The Directory `/usr/src/linux/include` contains the Linux kernel header files.

Kernel modules are inserted into the kernel by using any of the commands

```
modprobe module_name
insmod   module_name
```

The main difference between these two commands is that `modprobe` uses `insmod` to load any pre-requisite modules, and then the requested module, where as with insmod command first we have to load all prerequisite modules required by this module followed by this module. The standard directory for modules is `/lib/modules/kernel-version/`.

For example module_1 is the prerequisite for module_2. If we want to load module_2 by using `insmod` command then

```
insmod /lib/modules/2.4.20-8/kernel/module_1.o
insmod /lib/modules/2.4.20-8/kernel/module_2.o
```

If we want to load module_2 by using `modprobe` then

```
modprobe -a module_2
```

Once the work is finished with the module, it can be unloaded using the command

```
rmmod module_name
```

The modules that are currently loaded into the kernel can be obtained by using the command `lsmod`.

More information about Linux Kernel Module Programming can be found in [12] [13].

A.2 Netfilter Hooks

Netfilter is a subsystem in the Linux kernel. Netfilter has various hooks in the kernel's network code. We can register any function at these netfilter hooks. The handler functions for a particular hook are called in their order of priority for all the packets passing through that netfilter hook.

Netfilter defines five hooks for IPv4. These hooks are mentioned in the Table A.1. The IPv4 traversal diagram is shown in Figure A.1.

Hook callback functions should return one of the predefined return codes after they finished processing of the packet. These codes are shown in the Table A.2.

These return codes are explained below.

NF_DROP This packet should be dropped completely and any resources allocated for it should be released.

NF_ACCEPT Send the packet for further processing in the network stack.

Hook	**Called...**
NF_IP_PRE_ROUTING	After sanity checks, before routing decisions.
NF_IP_LOCAL_IN	After routing decisions if packet is for this host.
NF_IP_FORWARD	If the packet is destined for another interface.
NF_IP_LOCAL_OUT	For pkts coming from local processes on their way out.
NF_IP_POST_ROUTING	Just before outbound packets "hit the wire".

Table A.1: Available IPv4 hooks

Return Code	Meaning
NF_DROP	Discard the packet.
NF_ACCEPT	Send the packet for further processing.
NF_STOLEN	Steal the packet.
NF_QUEUE	Queue packet for userspace.
NF_REPEAT	Call this hook function again.

Table A.2: Netfilter return codes

NF_STOLEN The hook function will take processing of this packet from here and that netfilter should drop all processing of it. The packet and its respective sk_buff structure are still valid.

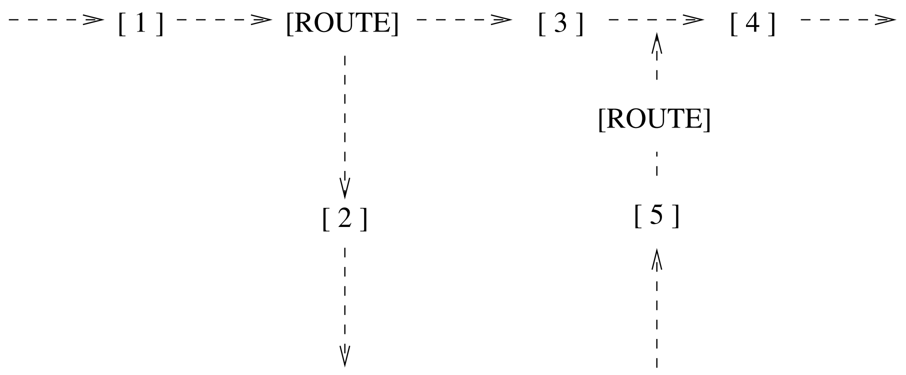
NF_QUEUE Queue the packet in user space. This is described in A.2.1.

NF_REPEAT Call the hook function again.

A netfilter hook function is registered by using `nf_register_hook()` function. `nf_register_hook()` takes the address of `nf_hook_ops` structure. The definition of `nf_hook_ops` structure is as follows:

```
struct nf_hook_ops {
        struct list_head list;

        nf_hookfn *hook;
```



Different Hooks:

1. NF_IP_PRE_ROUTING
2. NF_IP_LOCAL_IN
3. NF_IP_FORWARD
4. NF_IP_POST_ROUTING
5. NF_IP_LOCAL_OUT

Figure A.1: IPv4 Netfilter Hooks

```
        int pf;
        int hooknum;

        int priority;

};
```

list Maintains the list of netfilter hook functions.

hook Pointer to a function that will be called for the hook.

pf Specifies a protocol family. For IPv4 the protocol family is PF_INET.

hooknum The particular hook to install this function for and is one of the values listed in Table A.1.

priority The priority of this hook function.

More information on netfilter hooks can be found in [14] [15].

A.2.1 Userspace Packet Queuing

Netfilter provides a mechanism for passing packets out of the network stack to userspace. Whenever a netfilter hook function returns NF_QUEUE, the packet is queued in the userspace. The userspace application do whatever processing it needs to do with the packet and sends the packet back into the kernel with a verdict specifying what to do with the packet (such as ACCEPT or DROP).

For each supported protocol, a kernel module called a queue handler may register with netfilter to perform the mechanics of passing packets to and from userspace. The standard queue handler for IPv4 is ip_queue. It uses a netlink socket for kernel/userspace communication.

Libipq library provides an API for communicating with ip_queue.

A.3 Proc File System

The /proc file system is a virtual file system in the Linux. The files in the procfs provide information regarding the state of the machine to userland programs. *procfs* can be used to communicate with the kernel. In this we describe various functions used for creation and communication with *procfs* entries.

For dealing with *procfs*, proc_fs.h header file have to be included.

```
struct proc_dir_entry* proc_mkdir(const char* name,
struct proc_dir_entry* parent);
```

proc_mkdir() function creates a directory *name* in *parent.*

```
struct proc_dir_entry* create_proc_entry(const char*
name, mode_t mode, struct proc_dir_entry* parent);
```

create_proc_entry() function creates a *procfs* file with the name *name*, file mode *mode* in the directory *parent.* If parent is NULL, file is created in */proc/* directory.

```
void remove_proc_entry(const char* name, struct
proc_dir_entry* parent);
```

remove_proc_entry() function removes *name* in the directory *parent.*

Whenever a *procfs* file is read or written by a userlevel process, corresponding function is called. These functions are called *call back* functions. Call back functions should be initialized after the *procfs* file is created.

```
struct proc_dir_entry* entry;


entry->read_proc  = read_proc_func;
entry->write_proc = write_proc_func;
```

```
int read_func(char* page, char** start, off_t off,
int count, int* eof, void* data);
```

To communicate from the kernel to the userspace, read function is used. The *read_func()* function writes at most *count* bytes into *page* starting from *off*. It sets *eof* on reaching end of file. When this function is shared by more than one *proc* file, *data* field is used to distinguish between the files.

```
int write_func(struct file* file, const char* buffer,
unsigned long count, void* data);
```

To communicate from userspace to kernel, write function is used. The *write_func()* reads at most *count* bytes from *buffer* which lies in userspace.

More information on procfs can be found in [16].

A.4 ioctl's

Communication with the physical devices can be done in two ways. One is reading or writing to the device file. Second is communication through the *ioctl's.* When we write to device files, device drivers send this information to the devices.

Read ioctl's are used for communicating from a user process to the kernel. Write ioctl's are used for communicating from the kernel to a user process. The ioctl function has three arguments. First argument is the file descriptor of the device file. Second argument is the ioctl command number. Third argument is the *argument* to the ioctl command. We can use a cast to pass any data type.

An *ioctl* number for read ioctl is created as follows.

```
#define IOCTL_SEND_INFO _IOR(MAJOR_NUM, 0, long)
```

_IOR means it is read ioctl. MAJOR_NUM is the major device number. 0 is the number of the ioctl command. `long` is the type of the ioctl command argument.

```
#define IOCTL_RW_INFO _IOWR(MAJOR_NUM, 1, int)
```

The above `ioctl` is used for both input and output.

`register_chardev()` function is used for registering a character device driver inside the `init_module()` function of a module.

```
register_chrdev(MAJOR_NUM, DEVICE_NAME, &fops);
```

The third argument `fops` is a pointer to *file_ operations* structure which will hold the call back functions of the device we created. For talking to the device we have to create a device file. This can be created using the following command.

```
#mknod DEVICE_FILE_NAME c MAJOR_NUMBER 0
```

Whenever we read or write to this device file, corresponding call-back function is called. Unregistering of the character device driver is done as follows.

```
unregister_chrdev(MAJOR_NUM, DEVICE_NAME);
```

A.5 sk_buff structure

In the layered network architecture each protocol needs to add headers and tails to the data received from upper layers and needs to remove these at the receiving end. Linux uses SOCKET BUFFER sk_buff structure to efficiently pass the data between the layers.

Some of the important fields are explained below. These fields are also shown in the Figure A.2.

next Pointer to the next buffer in the list.

prev Pointer to the previous buffer in the list.

list List we are on.

sk Pointer to the socket we belongs to.

dev Pointer to the device we arrived on/are leaving by.

h, nh and mac Pointers to the transport, network and MAC layer headers respectively.

dst Points to destination cache that is used to send the packet to a particular destination.

cb[ ] This is the control buffer. It is free to use for every layer. We can put all the private data here.

len The length of the current protocol packet.

csum Checksum of the data portion of the buffer.

priority Packet queuing priority.

truesize Total size of the data buffer.

Each *sk_ buff* has a block of data associated with it. The *sk_ buff* has four data pointers, which are used to manipulate and manage the socket buffer's data.

head This is a pointer to the data buffer associated with *sk_ buff.* This is set at the time of allocating *sk_ buff.*

data This is a pointer to the current start of the data in the buffer. This varies depending on the current protocol layer that is handling *sk_ buff.*

tail This is a pointer to the current end of data in the buffer.

end This is a pointer to the end of data buffer. This is set at the time of allocating *sk_ buff.*

The sk_buff handling code provides standard mechanisms for adding and removing protocol headers and tails to the application data. These safely manipulate the *data, tail* and *len* fields in the sk_buff.

push() It moves the data pointer towards the start of the data buffer and increments the *len* field. It is used when adding protocol headers to the start of the data to be transmitted.

pull() It moves the data pointer away from the start of the data buffer and decrements the *len* field. It is used when removing protocol headers from the start of the data that has been received.

put() It moves the tail pointer towards the end of the data buffer and increments the *len* field. It is used when adding protocol information to the end of the data to be transmitted.

trim() It moves the tail pointer towards the start of the data buffer and decrements the *len* field. It is used when removing protocol tails from the received packet.

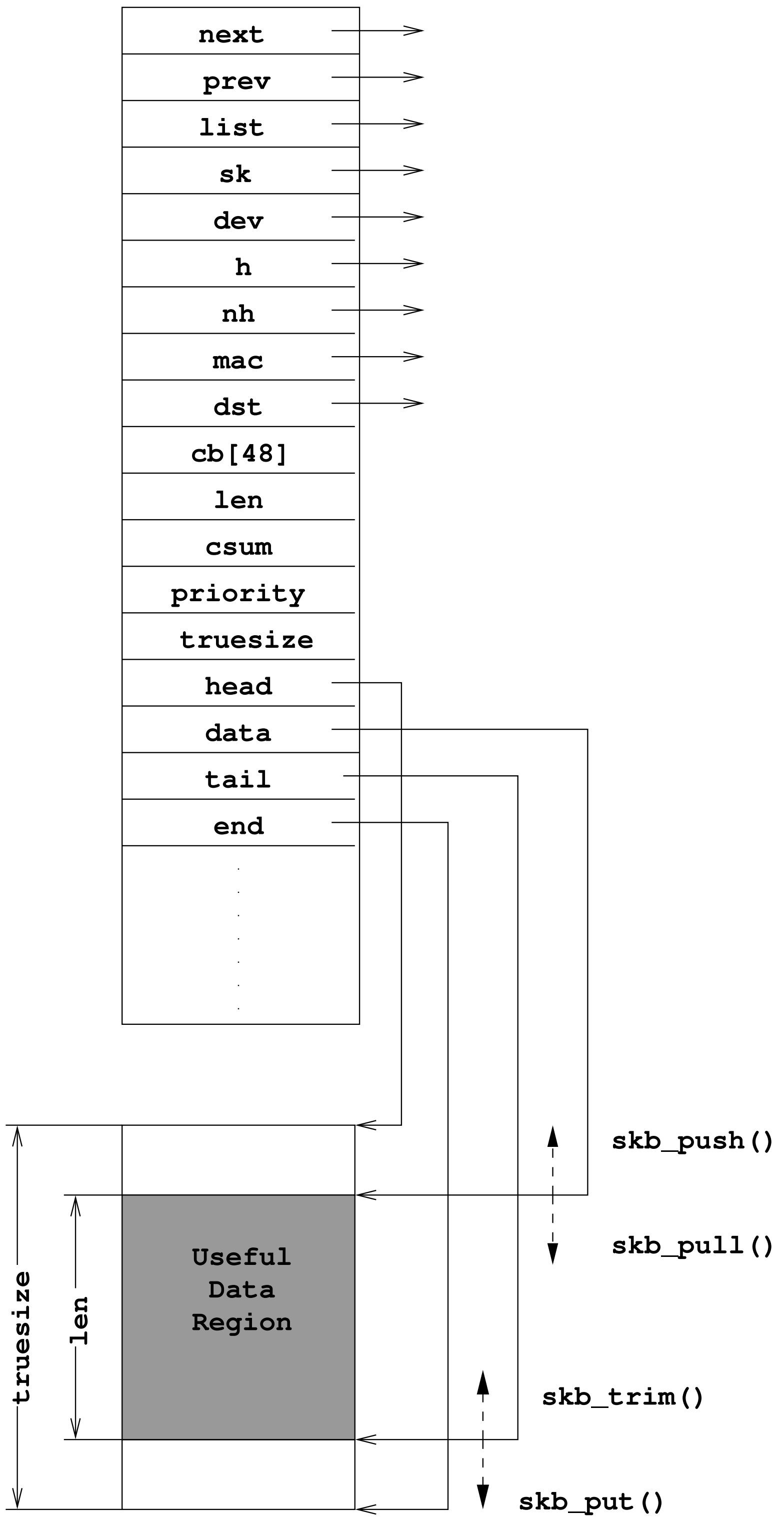


Figure A.2: sk_buff Structure

Appendix B

Source Code Organization

IP-IP-in This module handles processing of incoming packets into the network stack.

IP-IP-out This module handles processing of outgoing packets from this machine.

RoutingTables This module handles all the routing tables. It has call-back functions which are called when any user-level program writes to the *proc* file. This module exports all the routing tables to other parts of the kernel.

chardev This module provides *ioctl* interface for accessing and modifying the routing tables. It has call-back functions which are called when any user-level program calls *ioctl's*.

chardev.h Definitions for the *ioctl* calls for accessing and modifying the routing tables.

syscall_hack This module hacks the system calls bind and route. Applications are associated with a virtual host. This module finds this association and modifies the system calls to act on that virtual host. For example, a route command issued by an application that belongs to a virtual host is changed to act the command on the virtual host routing table rather than system routing table.

pid_ip This modules maintains the association between the virtual hosts and application programs. It associates PID of the wrapper program with virtual host IP address. All the applications executed in this wrapper are assigned the virtual host corresponding to the wrapper. This module also provides an *ioctl* interface to modify this association.

pid_ip.h Definitions for the *ioctl* calls for modifying the association between the wrapper and the virtual host.

my_header.h Header file which contains the definitions for global variables, flags etc of the emulation platform.

dst_entry_export This module captures dst_entry objects and exports these to other parts of the kernel.

rt_init.c This userlevel program writes all the routing tables, IP addresses etc to *proc* file */proc/rtable*, which is handled by *RoutingTables* module.

pidip_ioctl.c This userlevel program is used for modifying the PID-IP associations.

runme Shell script which compiles all the modules and inserts them into the kernel.

Several userlevel programs generates TCP and UDP traffic and calculates the bandwidth.

Bibliography

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

[2] Mark Carson and Darrin Santay. NIST Net – A Linux-based Network Emulation Tool. *ACM SIGCOMM Computer Communications Review*, 33(3):111–126, July 2003.

[3] Grenville Armitage. Maximising student exposure to networking using freebsd virtual hosts. *ACM SIGCOMM Computer Communications Review*, 33(3):137–143, July 2003.

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

[5] FreeBSD, October 2003. http://www.freebsd.org.

[6] Robert Ricci, Chris Alfeld, and Jay Lepreau. A solver for the network testbed mapping problem. *ACM SIGCOMM Computer Communications Review*, 33(2):65–81, April 2003.

[7] User mode linux. http://user-mode-linux.sourceforge.net/.

[8] Stefan Savage David Ely and David Wetherall. Alpine: A user-level infrastructure for network protocol development. In *Proc. of the Third USENIX Symposium on Internet Technologies and Systems*. http://alpine.cs.washington.edu/.

[9] http://www.netperf.org.

[10] bw_tcp. http://www.bitmover.com/lmbench/bw_tcp.8.html.

[11] Fang Hao and Pramod Koppol. An internet scale simulation setup for bgp. *ACM SIGCOMM Computer Communications Review*, 33(3):43–57, July 2003.

[12] Peter Jay Salzman and Ori Pomerantz. The linux kernel module programming guide, 2003. http://www.faqs.org/docs/kernel/.

[13] Bryan Henderson. Linux loadable kernel module howto, 2005. http://tldp.org/HOWTO/Module-HOWTO/.

[14] Owen Klan. How to use netfilter hooks, 2003. http://uqconnect.net/%7Ezzoklan/documents/netfilter.html.

[15] Rusty Russell and Harald Welte. Linux netfilter hacking howto, 2002. http://www.netfilter.org/documentation/HOWTO//netfilter-hacking-HOWTO.html.

[16] Erik (J.A.K.) Mouw. Linux kernel procfs guide, 2001. http://kernelnewbies.org/documents/kdoc/procfs-guide/lkprocfsguide.html.

[17] pragmatic / THC. (nearly) complete linux loadable kernel modules. http://reactor-core.org/linux-kernel-hacking.html.

[18] Daniel P.Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly, 2nd edition, December 2002.

[19] Alessandro Rubini and Jonathan Corbet. *Linux Device Drivers*. O'Reilly, 2nd edition, June 2001. http://www.xml.com/ldd/chapter/book/.

[20] M Dziadzka U Kunitz R Magnus C Schroter M Beck, H Bohme and D Verworner. *Linux Kernel Programming.* Addison Wesley Professonal, 3rd edition, August 2002.

[21] Cross referencing linux. http://lxr.linux.no/source/.

[22] A map of the networking code in linux kernel 2.4.20. http://datatag.web.cern.ch/datatag/papers/tr-datatag-2004-1.pdf.

[23] Jonathan Sevy. Linux network stack walkthrough.

[24] The linux documentation project. http://www.tldp.org/.

[25] Juan-Mariano de Goyeneche. Kernel links for understanding the linux kernel. http://jungla.dit.upm.es/%7Ejmseyas/linux/kernel/hackers-docs.html.

[26] Linux kernel sources. http://www.kernel.org/pub/linux/kernel/.