

國立交通大學

資訊工程系 碩士論文

移植 NCTUns 網路模擬器到 Linux 平台並提供
Emulation 的功能

Porting the NCTUns Network Simulator to Linux and
Supporting Emulation


研究生：廖國強

指導教授：王協源 教授

中華民國九十三年六月

中文摘要

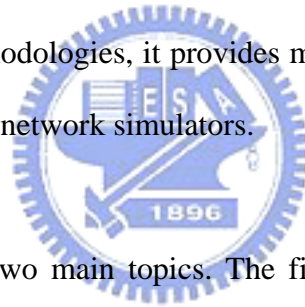
對於 IP 網路的研究者而言，一個網路模擬器是一個非常有用的工具，可以幫助他們學習或發展新的 IP 網路技術。有了一個網路模擬器，研究者可以省去很多建構真實的網路環境時所要花的時間與金錢。在模擬的環境中，所有的網路狀況以及設定都是可重複呈現的，因此研究者可很容易的得到重複的實驗結果。NCTUns 網路模擬器已經發展了很多年了。可以說它已經變成一個高精確度以及具有擴張性的一個網路模擬器。由於它的幾個新穎的模擬方法，它提供了很多獨特的優點，而這些優點是其他傳統的模擬器所無法擁有的。



在本篇論文中，可分成兩大主題。一為移植 NCTUns 網路模擬器到 Linux 平台。過去，NCTUns 只能在 FreeBSD 平台上執行。但由於最近 Linux 系統越來越普及，所以我們也希望能將模擬器移植到 Linux 平台並讓更多人使用我們的模擬器。在本篇論文中的第一部分，我們將會討論移植 NCTUns 網路模擬器的相關細節。另一主題是為 NCTUns 提供 emulation 的功能。Emulation 是一種能夠讓網路模擬器和真實的網路設備溝通的功能。這種功能可以大大的幫助 IP 技術發展者發展及測試他們的產品。在本篇論文的第二部分，我們將會討論 emulation 在 NCTUns 下的設計與實作。

Abstract

For IP network researchers, a network simulator is a very useful tool to help them study or develop new IP network technologies. With a network simulator, researchers can save much time and money required to build a real network environment. In a simulated network environment, all network conditions and configurations are repeatable, therefore researchers can easily repeat their experimental result. The NCTUns network simulator has been developed for many years. It already becomes a high-fidelity and extensible network simulator. Due to its several novel simulation methodologies, it provides many unique advantages that can not be achieved by traditional network simulators.



In this paper, we have two main topics. The first one is porting the NCTUns network simulator to Linux. In the past, the NCTUns can only run on the FreeBSD platform. However recently, the Linux system becomes more and more popular. In part I of this paper, we will discuss and describe in details how we port the NCTUns from FreeBSD to Linux. The other topic is supporting emulation in the NCTUns. Emulation is a kind of ability that can allow a network simulator to interact with real network devices. This function can greatly help IP technology developers to develop or test their products. In part II of this paper, we will discuss the detail of the design and implementation of the NCTUns network emulator.

致謝

感謝恩師王協源教授在這研究所兩年的悉心教導，讓我在專業領域方面獲益良多。研究所兩年，將會是我在資訊工程領域影響我最深的一個階段。有了這兩年扎實的專業訓練，我相信這是我往後在這領域發展最重要的基礎訓練。

感謝林華君教授、黃寶儀教授、以及吳曉光教授在論文口試期間所給予的建議與指正，讓這篇論文更為完善。



感謝網路與系統實驗室(Network and System Lab.)的所有成員，由於有你們的陪伴以及互相勉勵，讓我兩年的研究生生活更加豐富與多采多姿。

最後，感謝家人與朋友的全力支持，讓我研究所兩年能全力以赴的致力於課業與研究之上，順利的完成研究所學業。

Table of Contents

Part I: Porting the NCTUns Network Simulator to Linux	1
1. Development History	1
2. Introduction	3
3. High Level Architecture	4
3.1 Simulation Methodology	5
3.2 Job Dispatcher and Coordinator	7
3.3 Simulation Engine Design	9
3.4 Kernel Modifications	14
3.5 Discrete Event Simulation	16
4. Porting to Linux	16
4.1 User-Level Components	16
4.1.1 GUI	17
4.1.2 Job Dispatcher, Coordinator, Daemons, and Real-life Programs	17
4.2 Simulation Engine	19
4.2.1 Independent Components	19
4.2.2 System Calls	19
4.2.3 Memory Mapping	20
4.2.4 Process Scheduling	24
4.3 Kernel Modifications	27
4.3.1 IP Address Translation and Source-Destination-pair IP Scheme	27
4.3.2 Tunnel Interface	33
4.3.3 System Calls	39
4.3.4 Port Number Mapping and Translation	43
4.3.5 Support Different Time Scales	46
4.3.6 Processing Kernel Timers and Kernel Events	48
4.3.6.1 Maintain Virtual-time Timers	48
4.3.6.2 Kernel Timeout Event Triggering	51
4.3.7 Based on the Virtual Time	52
4.3.8 NCTUns Divert Socket	58
5. Evaluation	63
5.1 Simulation and Experiment Result Comparison	63
5.2 Simulation Speed	67
5.3 Fixed CBR UDP Stream on Multiple Hop Networks Case	68
5.4 MANET in the NCTUns network simulator	71
6. Future Work	74
7. Concluding Remarks for Part I	75

Part II: Supporting Emulation	76
1. Introduction.....	76
2. Design Goals.....	78
3. Features	79
3.1 Based On IP Protocol.....	79
3.2 Interact With Real Hosts	80
3.3 Interact With Real Routers.....	80
3.4 Can Establish TCP/UDP Connections between the Emulator and Real Hosts	81
3.5 An External Host Can Be an Ad-hoc/Infra-structure Mode Mobile Node in Emulator.....	82
3.6 Can Use All Features and Capabilities of NCTUns Network Simulator	82
3.6.1 Support for Various Networks	83
3.6.2 Support for Various Networking Devices	83
3.6.3 Support for Various Network Protocols	83
3.6.4 Application Compatibility and Extensibility	84
3.6.5 User Friendliness	84
3.6.6 Open System Architecture	84
4. Related Work.....	85
5. Design and Implementation	86
5.1 User-Level Daemon	87
5.2 Capture Packets from the Kernel	88
5.2.1 Using Divert Socket in FreeBSD.....	88
5.2.2 Using Netfilter and Added System Calls in Linux.....	89
5.3 Design of Emulation Daemon for External Host.....	94
5.3.1 Adding Routing Entries.....	95
5.3.2 Translate IP Address.....	96
5.3.3 Translate Port Number	97
5.3.4 Setting Packet Filter Rules.....	98
5.4 Design of Emulation Daemon for External Router.....	99
5.4.1 Translate IP Address.....	101
5.4.2 Adding Routing Entries.....	103
5.4.2.1 Adding Routing Entries on the Simulation Machine	103
5.4.2.2 Adding Routing Entries on the External Router	106
5.4.3 Unnecessary to Translate the Port Number.....	107
5.4.4 Setting Packet Filter Rules.....	107
5.4.5 “200.X.Y.Z” Format Discussion	108

5.5 Simulation Speed Should Synchronize With the Real Time.....	111
6. More Consideration	112
6.1 Re-compute Header Checksum.....	112
6.2 More than One External Host	112
6.3 More than One External Router	115
7. Evaluation	117
7.1 Accuracy	117
7.2 Throughput.....	120
8. Concluding Remarks for Part II.....	122
Reference	123



List of Figures

Figure I-3.1.1: The kernel re-entering simulation methodology.....	5
Figure I-3.1.2: The simulation network topology.....	7
Figure I-3.1.3: The packet trace of a packet that will traverse the simulation network from host 1 to host 2.....	7
Figure I-3.2: The distributed architecture of the NCTUns 1.0 network simulator.....	8
Figure I-3.3.1: The architecture of the NCTUns 1.0 network simulator.....	10
Figure I-3.3.2: The module based platform.....	10
Figure I-4.3.1: IP address translation.....	28
Figure I-4.3.2: Tcpdump module.....	36
Figure I-5.1.1: The testing network topology.....	63
Figure I-5.1.2: The total throughput comparison between the experiment case and the simulation case.....	65
Figure I-5.1.3: The experiment result of two contending TCP connections.....	66
Figure I-5.1.4: The simulation result of two contending TCP connections.....	66
Figure I-5.2.1: The simulation performance under various constant-bit-rate UDP traffic loads. (A higher ratio means a better performance).....	67
Figure I-5.3.1: The different number of forwarding nodes in each simulation case....	69
Figure I-5.3.2: The performance under different forwarding nodes (switch or router).....	69
Figure I-5.3.3: The memory usage under different forwarding nodes (switch or router).....	70
Figure I-5.4.1: The network topology that is organized by the ad-hoc mode mobile nodes.....	72
Figure I-5.4.2: The performance under different network sizes (different dimensions).....	73
Figure I-5.4.3: The memory usage under different network sizes (different dimensions).....	74
Figure II-3.4(a): A TCP connection can be set up between the simulated node and the external host.....	81
Figure II-3.4(b): A TCP connection can be set up between two external hosts.....	82
Figure II-5.1(a): The emulation daemon receives packets from the real-world network (Ethernet network), and then directs them into the simulation network.....	87
Figure II-5.1(b): The emulation daemon receives packets from the simulation network, and then injects them into the real-world network (Ethernet network).....	88
Figure II-5.2.2: The original architecture of the IP packet filtering mechanism in	

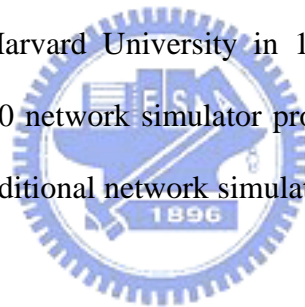
Linux and our proposed approach.....	90
Figure II-5.4 (a): the trace of TCP SYN packet with external router.....	99
Figure II-5.4 (b): the trace of TCP SYN-ACK packet with external router.....	100
Figure II-5.4.1: Translate the IP address pair from the S.S.D.D format to the 200.X.Y.Z format.....	102
Figure II-5.4.2: A complex topology for an external router.....	105
Figure II-6.2: More than one external host.....	113
Figure II-6.3: More than one external router.....	115
Figure II-7.1.1: The emulation case for testing the accuracy of RTT.....	118
Figure II-7.1.2: The variance of RTT.....	118
Figure II-7.1.3: The difference between the average and the expected RTT.....	118
Figure II-7.2.1: The emulation case for testing two contended greedy TCP connections.....	121
Figure II-7.2.2: The contending behavior between a real TCP connection and a simulated TCP connection.....	121
Figure II-7.2.3: The contending behavior between two simulated TCP connections	121



Part I: Porting the NCTUns Network Simulator to Linux

1. Development History

The NCTUns 1.0 network simulator is a high-fidelity and extensible network simulator capable of simulating various devices and protocols used in both wired and wireless IP networks. Its core technology is based on the simulation methodology invented by S.Y. Wang at Harvard University in 1999 [6, 7]. Due to this novel methodology, the NCTUns 1.0 network simulator provides many unique advantages that cannot be achieved by traditional network simulator such as OPNET [8] and ns-2 [3].



The predecessor of the NCTUns 1.0 network simulator is the Harvard network simulator [9], which was authored by S.Y. Wang in 1999. As feedbacks about using the Harvard network simulator come back, it is found that the Harvard network simulator has several limitations and drawbacks that need to be overcome and solved, and some features and functions need to be implemented and added to it. For these reasons, after joining National Chiao Tung University (NCTU), Taiwan in February 2000, Wang designed a new simulation methodology for the NCTUns 1.0 network simulator.

The NCTUns 1.0 network simulator removes many limitations and drawbacks

with the Harvard network simulator. It uses a distributed architecture to support remote simulations and concurrent simulations. It also uses an open-system architecture to enable protocol modules to be easily added to the simulator. In addition, it has a fully-integrated GUI environment for editing a network topology and specifying network traffic, plotting performance curves, configuring the protocol stack used inside a network node, and playing back animations of logged packet transfers.

Furthermore, Wang proposes an approach to apply discrete event simulation to the NCTUns 1.0 network simulator to speed up its simulation speed [2]. The Harvard network simulator used a time-stepped method to implement its simulation engine. As such, its simulation speed is very slow. To overcome this problem, the NCTUns 1.0 applied the event-driven (i.e., the discrete event simulation methodology [10]) approach to its simulation engine. As such, its simulation speed is much faster than the Harvard network simulator.

The NCTUns 1.0 network simulator has been released to the networking community on 11/01/2002. Its Web site is set up at <http://NSL.csie.nctu.edu.tw/nctuns.html>. As of 4/5/2004, according to the download user database, more than 1,610 people/organizations from more than 60 countries have registered with the NCTUns 1.0's web site and downloaded it.

We want more people or organizations to use the NCTUns 1.0 network simulator. Because the Harvard network simulator and the current version of NCTUns 1.0 network simulator can only run on the FreeBSD platform, promoting their uses has some difficulties. So we decide to port the NCTUns network simulator from FreeBSD

4.x to Linux 2.4.x. Of course, all advantages of the FreeBSD version of NCTUns 1.0 network simulator will be reserved and even improved during the porting to the Linux platform. All novel or unique design of the FreeBSD version of NCTUns 1.0 network simulator will also be implemented into the Linux version, including kernel re-entering simulation methodology, discrete event simulation methodology, open system architecture, etc. In addition, several new network types are implemented into the Linux version including traditional optical network, optical burst network, GSM/GPRS cellular network, etc. Due to our continuous improvement, the Linux version of NCTUns 1.0 network simulator is ready and is more powerful than the FreeBSD version. We plan to release the Linux version of NCTUns 1.0 network simulator soon. At that time, we expect that more and more people or organizations will use our network simulator.



2. Introduction

The FreeBSD version of the NCTUns network simulator has already become a very stable product. From releasing it at 11/01/2002 until now, many users have provided various suggestions and bug reports, and we continue to improve it according to these feedbacks. For the latest version, we are confident that the FreeBSD version of the NCTUns network simulator is a mature and stable product.

However, we observe a situation: many users expect us to port the NCTUns network simulator from FreeBSD to Linux platform. Many users are familiar with the Linux system. However, not many of them have experience with using a FreeBSD system. As such, they have to learn how to use the FreeBSD system before using the

NCTUns 1.0 network simulator. This becomes a big barrier to spread our network simulator. This problem becomes more critical if more and more people use the Linux as their usual working platform. As such, in order to continually spread our network simulator, it is better for us to port the NCTUns 1.0 network simulator to the Linux platform.

Porting the NCTUns 1.0 network simulator to Linux is not a simple task. All components must be carefully considered. This is especially true for the kernel modification part and the simulation engine. This is because other components (e.g. daemons, tools, job dispatcher, coordinator and GUI) are mostly independent of operating systems. The most difficult task of porting to Linux is to hack the Linux kernel. As we know, FreeBSD and Linux have many differences between them although both of them belong to the UNIX-like system. For example, process scheduling, BSD socket implementation, TCP/IP protocol stack implementation and soft interrupt mechanism are completely different. In addition, applying the kernel re-entering and discrete event simulation methodology to Linux is also a big challenge to us. Our goal is to reserve and even to improve all the features of the FreeBSD version of the NCTUns network simulator in the Linux version.

In part I of this paper, we will clearly explain how we port the NCTUns 1.0 network simulator to Linux. All components in the NCTUns network simulator will be discussed in the following chapters.

3. High Level Architecture

The NCTUns network simulator uses a distributed architecture to support remote

simulations and concurrent simulations. It also can use an open-system architecture to enable protocol modules to be easily added to the simulator. In the following we describe some important features and components of the NCTUns network simulator.

3.1 Simulation Methodology

The NCTUns network simulator is based on a new simulation methodology -- the kernel re-entering simulation methodology. It uses an existing real-world FreeBSD/Linux protocol stack to provide high-fidelity TCP/IP network simulation results. Figure I-3.1.1 depicts this concept.

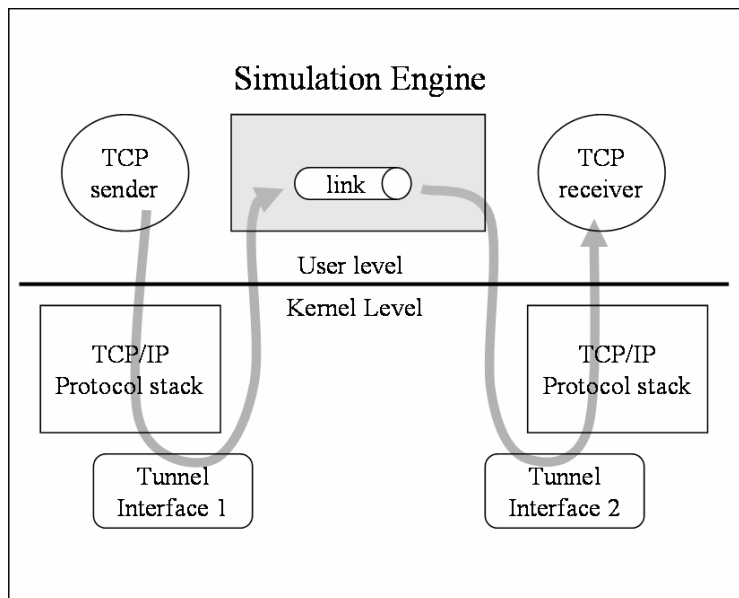
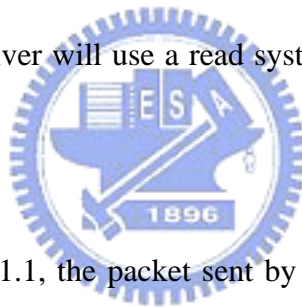


Figure I-3.1.1:
The kernel re-entering simulation methodology

In figure I-3.1.1, the TCP/IP protocol stack used in the simulation is the existing real-life stack in the kernel. Although there are two TCP/IP protocol stack depicted, actually they are the same one – the protocol stack inside the FreeBSD/Linux kernel. The tunnel interface is a pseudo network interface that does not have a real physical network attached to it. From the kernel's point of view, the tunnel interface is no different from any real Ethernet network interface.

In figure I-3.1.1, the TCP sender sends a packet into the kernel, and the packet goes through the kernel's TCP/IP protocol stack just as an Ethernet packet would do. Because we configure the tunnel interface 1 as the packet's output device, the packet will be inserted to tunnel interface 1's output queue. The simulation engine will immediately detect such an event and issue a read system call to get this packet through tunnel interface 1's special file (Every tunnel interface has a corresponding device special file in the /dev directory.). After experiencing the simulation of transmission delay and link's propagation delay, the simulation engine will issue a write system call to put the packet into tunnel interface 2's input queue. The kernel will then raise a soft interrupt and put the packet into the TCP/IP protocol stack. Then, the packet will be put into the receive queue of the socket that the TCP receiver creates. Finally, the TCP receiver will use a read system call to get packet out of the kernel.



In the case of figure I-3.1.1, the packet sent by the TCP sender passes through the kernel two times. This is the property of the kernel re-entering simulation methodology. By re-entering the kernel multiple times, we can create an illusion that a packet passes through several different hosts (i.e., the packet thinks that it passes through several different TCP/IP protocol stack). Actually, the packet is always in the same machine and passes through the same TCP/IP protocol stack. The following figures (figure I-3.1.2 and figure I-3.1.3) further illustrate this concept.

Figure I-3.1.2 shows an example simulation network topology and figure I-3.1.3 illustrates how the kernel re-entering simulation methodology works in figure I-3.1.2. In the example topology, host 1, host 2 and the router are layer-3 devices while the switch is a layer-2 device (Here we use the OSI 7-layer standard). We directly use

those protocols that are higher than the layer-3 protocol (i.e., the network or IP layer) in the kernel. As such, if any device is a layer-3 device, we only need to simulate its layer-1 and layer-2 protocol in the simulation engine and its other protocols can be simulated by directly using those protocols already in the kernel. Therefore, in figure I-3.1.2, when a packet is passed through the two hosts or the router, it will be passed into the kernel. As we can see in figure I-3.1.3, if a packet wants to traverse the simulation network from host 1 to host 2, it needs to be put into the kernel three times.

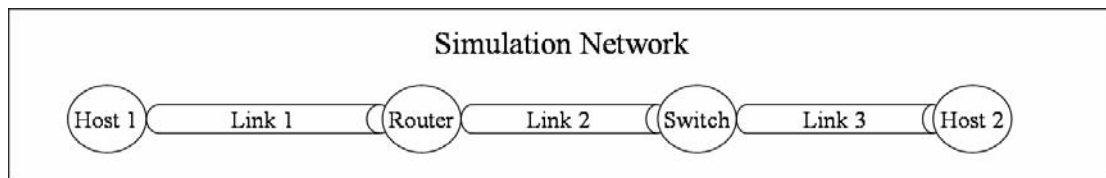


Figure I-3.1.2: The simulation network topology

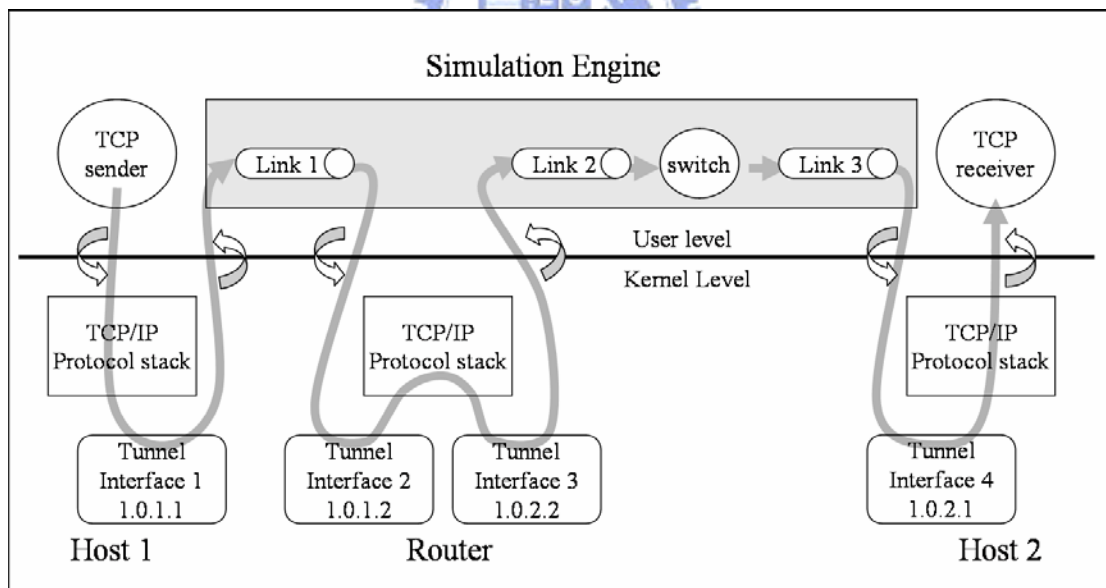


Figure I-3.1.3: The packet trace of a packet that will traverse the simulation network from host 1 to host 2.

3.2 Job Dispatcher and Coordinator

The NCTUns network simulator uses a distributed architecture to support remote

simulations and concurrent simulations. The job dispatcher is used to do this task. It should be executed and remain alive all the time to manage multiple simulation machines. On every simulation machine, the coordinator needs to be executed and remain alive to let the job dispatcher know whether currently this machine is busy running a simulation case or not. Figure I-3.2 depicts the distributed architecture of the NCTUns 1.0 network simulator.

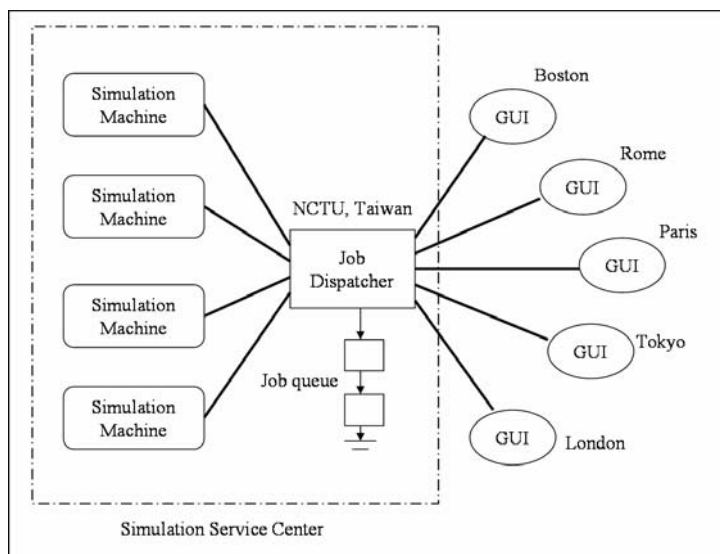


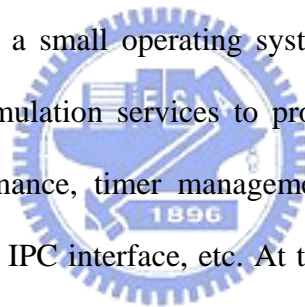
Figure I-3.2:
The distributed
architecture of the
NCTUns 1.0 network
simulator

For example, the job dispatcher in the simulation service center can accept simulation jobs from all of the world. When a user submits a simulation job to the job dispatcher, the dispatcher selects an available simulation machine to service the job. If there is no available simulation machine, the job will be put into the job queue of the job dispatcher. Every simulation machine always has a running coordinator to communicate with the GUI program and the job dispatcher. The coordinator will notify the job dispatcher whether the simulation machine managed by itself is available or not. When the coordinator receives a simulation job from the job dispatcher, it forks (executes) a simulation engine process to simulate the specified network and protocols. When the simulation engine process is running, the coordinator will communicate with the job dispatcher and the GUI program. For

example, periodically the simulation engine process will send the current virtual time of the simulation network to the coordinator. Then the coordinator will relay the information to the GUI program. This enables the GUI user to know the progress of the simulation. During a simulation, the user can also on-line set or get a protocol module's value (e.g. to query or set a switch's switch table). Message exchanges happening between the simulation engine process and the GUI program are all done via the coordinator.

3.3 Simulation Engine Design

The simulation engine is a user-level program and has complex functions. We can say that it functions like a small operating system. Through a defined API, it provides useful and basic simulation services to protocol modules. These services contain virtual clock maintenance, timer management, event scheduling, variable registration, script interpreter, IPC interface, etc. At the same time, it manages all of the tools and daemons that are used in a simulation case and decides when to start these programs, when to finish them, and when to run them. Figure I-3.3.1 shows an architecture diagram readers of the NCTUns 1.0 network simulator.



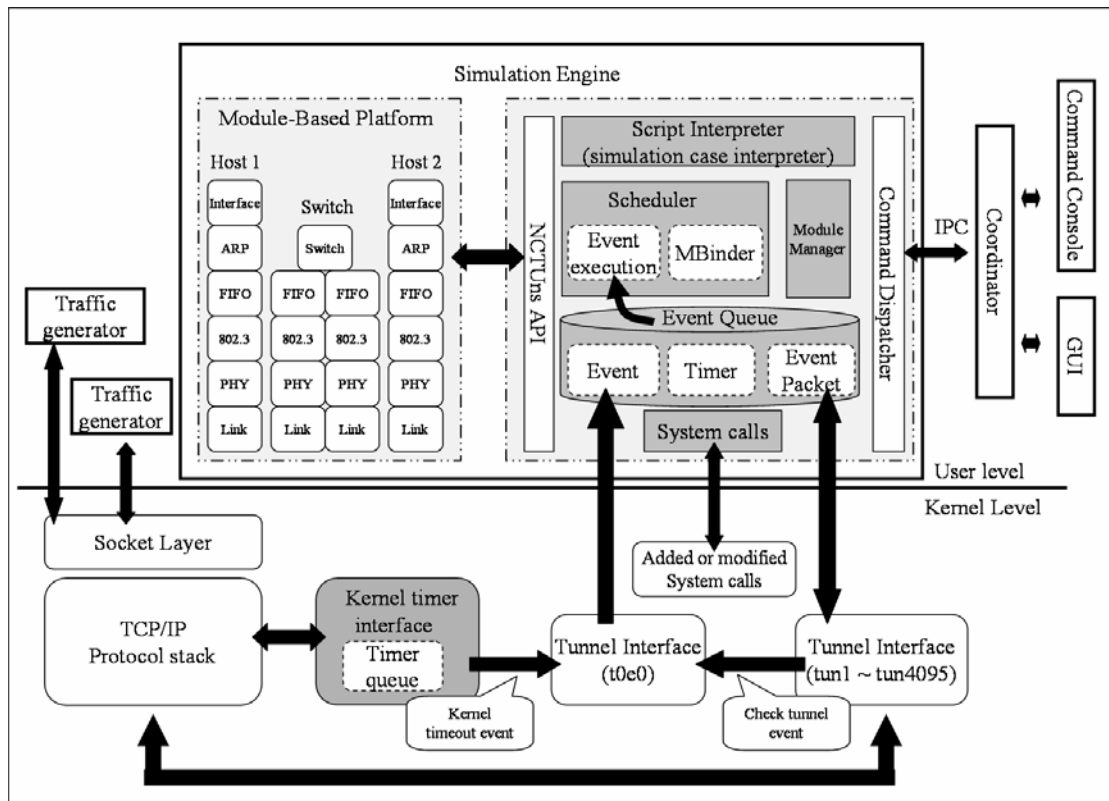


Figure I-3.3.1: The architecture of the NCTUns 1.0 network simulator

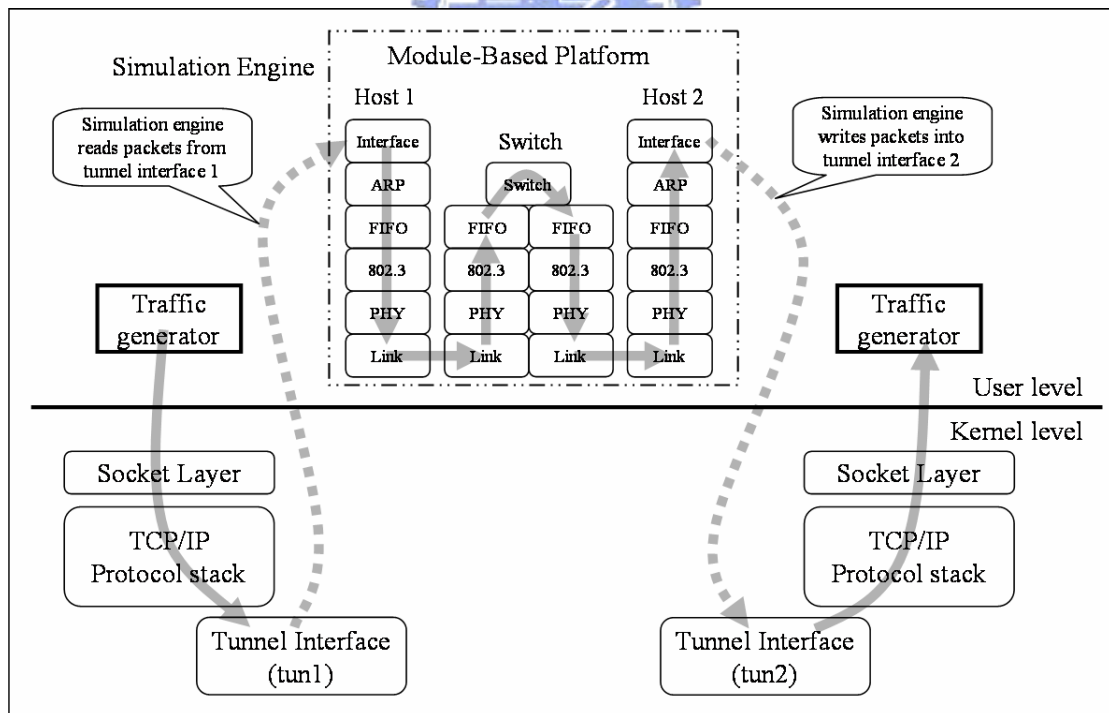


Figure I-3.3.2: The module based platform

In figure I-3.3.1, we can see the whole architecture of the NCTUns 1.0 network simulator. In the section, we first describe the organization of the simulation engine.

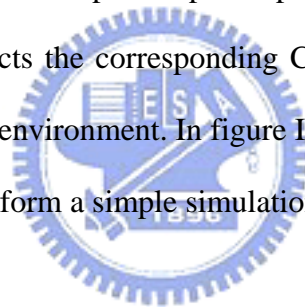
In section 3.4, we will explain how the kernel supports the simulation engine. We can simply divide the simulation engine into several components:

I. Script Interpreter

The script interpreter reads a script file of a simulation case to construct the simulation network environment, the network conditions, protocol module settings, and the network traffic.

II. Module Manager

The module manager manages all protocol modules that users registered and used in a simulation. When the script interpreter parses the script file, the module manager dynamically constructs the corresponding C++ objects and organizes them to build a simulation network environment. In figure I-3.3.1, those modules in the box of the module-based platform form a simple simulation network.



III. Command Dispatcher

The command dispatcher is used to communicate with other external components such as the coordinator, command console (It is a modified tcsh.), and the GUI program.

IV. NCTUns APIs

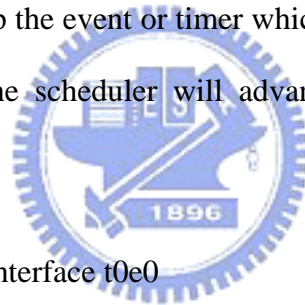
We call these APIs provided by the simulation engine NCTUns APIs. All of the protocol modules can ask for the simulation engine's services via the NCTUns APIs such as registering modules, processing events, setting timers, creating/freeing packets, etc.

V. Event Queue

The event queue has three kinds of data structure type inside it: event, timer, and event packet. The event is used to encapsulate messages that are exchanged between protocol modules. Every event has a timestamp inside it used to decide when to process it. The timer can be used to set what to do at a specified time stamp. If an event encapsulates a network packet (e.g. an IP packet), we call the event an event packet. In the event queue, all events or timers are sorted with their timestamp.

VI. Scheduler

The main job of the scheduler is to execute the event or timer in the event queue. The scheduler always picks up the event or timer which has the smallest timestamp to execute. In the meantime, the scheduler will advance the simulation time to the timestamp of the event.



VII. System calls and tunnel interface `t0e0`

We use two approaches to enable the simulation engine to communicate with the FreeBSD/Linux kernel. The first approach is through system calls. The simulation engine can use system calls that are added or modified by us to register/get information into/from the kernel. This approach suits the situation that the simulation engine actively wants to get or set some kernel parameters. The second approach is through using a tunnel interface. If the kernel wants to actively inform the simulation engine with some information, we use the tunnel interface `t0e0`. The kernel can fill a packet with some information and inserts it into the tunnel interface `t0e0`. Then the simulation engine can issue a `read()` system call to get the packet and further get the information inside it. This mechanism is mostly used by the kernel timeout event and

the tunnel check event (we will describe this in section 3.5 of part I).

VIII. IPC (Inter-Process Communication)

The IPC in the simulation engine is used to communicate with the coordinator. When a GUI user wants to send a command to the simulation engine such as pause a simulation, stop a simulation, send command to a protocol module etc., the GUI should send the command to the coordinator and then the coordinator relays the command to the simulation engine via IPC. After the simulation engine processes the command, it will send the result back to the coordinator and then the coordinator will relay the result to GUI.

Due to the module-based platform, we can dynamically construct or change the network protocol of a device. For example, figure I-3.3.2 shows a simple simulation case: the switch is a two-port layer-2 device and is connected with the host 1 and host 2 (host 1, 2 are both layer-3 devices). We can easily change the protocol module settings of host 1. We can just replace the FIFO module with a RED (Random Early Drop) module in the script file and then the module manager will dynamically construct corresponding protocol module settings according to the script file. We can also build a network device via the module-based platform. The switch device in figure I-3.3.2 is a layer-2 device and all components of the device are represented by modules. In addition to protocol modules, the layer-3 devices (such as host 1, host 2) will need kernel supports because we directly use the layer-3 protocols in the kernel.

Figure I-3.3.2 also shows the flow path that a packet will take when it is exchanged between the two traffic generators via the module-based platform. We already explain how a packet will pass through the kernel in section 3.1. When the

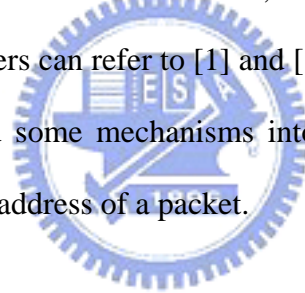
packet is read by the simulation engine from tunnel interface 1 (tun1), the packet will follow the trace of figure I-3.3.2 and then the simulation engine will insert it into tunnel interface 2. Finally, the kernel will send it to the traffic generator.

3.4 Kernel Modifications

In order to enable the kernel re-entering simulation methodology to properly work, we should modify the kernel source according to our requirements. These requirements include:

I. Allow S.S.D.S IP Scheme to Work

In order to route packets in the same kernel, we propose a special IP scheme: S.S.D.D IP format. Readers can refer to [1] and [13] to get the detailed definition. As such, we should add some mechanisms into the TCP/IP protocol stack to correctly translate the IP address of a packet.



II. Modify the Tunnel Interface Device Driver

Modifying the tunnel driver is necessary to enable the S.S.D.D IP scheme to work because we want the packet to have the normal IP scheme when it is read by the simulation engine. Also, we want the packet to have the S.S.D.D format when it is inserted into a tunnel interface.

III. Perform Port Number Mapping

We use an example to illustrate this requirement. If there are two Web servers running on the same simulation network, both of them may want to use the default port number 80 as their listening port number. However, because they are running on the same system (using the same TCP/IP protocol stack), only one of

them will successfully bind port number 80. To solve this problem, we should do port mapping in the kernel to enable the program running on different simulated nodes to use the same port number.

IV. Add or Modify System Calls

We need to add or modify some system calls to provide services that the simulation engine will require. For example, when a traffic generator is forked by the simulation engine, the simulation engine immediately needs to tell the kernel that the traffic generator belongs to which node. This operation is needed because we have to use this information to translate the IP address of those packets sent by the traffic generator.

V. Let Kernel Timers to be based on the Virtual Time

Because all of the events or timers in the simulation engine's event queue are based on the virtual time, all of kernel timers used for the NCTUns 1.0 should also use the virtual time. For example, when the TCP control block (i.e., a TCP socket handler) transmits out a packet, it may set a retransmission timer. If the TCP control block is used for a simulation network, the timer should use the virtual time. In addition, all of the system calls that are involved with real time may need to use virtual time. These system calls include `select()`, `alarm()`, `sleep()`, `gettimeofday()`, etc.

VI. Kernel Events

There are two kinds of kernel event. The first is the kernel timeout event. When the kernel wants to schedule a timer and the timer is based on the virtual time, we should use the tunnel interface `t0e0` to tell the simulation engine when to trigger this event. The second is the tunnel check event. When any packet is

queued into a tunnel interface's output queue, the kernel will insert a tunnel check event into the tunnel interface to let the simulation engine know which tunnel interface has packets to send.

3.5 Discrete Event Simulation

We apply the discrete event simulation methodology to the NCTUns network simulator to speed up the simulation speed [2]. The challenge is to combine the kernel re-entering simulation methodology with the discrete event simulation methodology. The objects simulated in the NCTUns 1.0 are not contained in a single program; rather, they are contained in multiple independent programs running concurrently on a UNIX machine such as traffic generators, the simulation engine, the UNIX kernel, etc. Therefore, we need the kernel to provide some information or services to communicate with the simulation engine. As such, the simulation engine can manage all events and timers at user level. In other words, our goal is to manage and trigger all of the events in the simulation engine regardless of whether the events are kernel events or not.

4. Porting to Linux

This chapter is the most important part of part I in this paper. In this chapter, we will discuss all components of the NCTUns 1.0 network simulator one by one and focus on their differences between the FreeBSD and the Linux versions.

4.1 User-Level Components

GUI, job dispatcher, coordinator, the simulation engine, daemons and real-life programs are all user-level programs. These programs have a common property -- they are all platform independent programs. On most UNIX-like systems, they can be easily re-compiled and executed without a large number of modifications. Naturally, they can be easily ported from FreeBSD to Linux.

4.1.1 GUI

We use the Qt library [11] to develop our GUI program. Qt is a complete C++ application development framework, which includes a class library and tools for multi-platform development and internationalization. It provides many graphical Qt tools and the numerous Qt APIs such as drag and drop, 3D OpenGL graphics and network programming, etc. It has a powerful property -- it is a cross-platform C++ application development framework. As such, we just need to maintain a single source-tree of the GUI program. When we need to port it to another platform, the only thing that we need to do is to recompile it.

4.1.2 Job Dispatcher, Coordinator, Daemons, and Real-life Programs

The job dispatcher and the coordinator are standard C++ programs. To communicate, they use TCP socket, UDP socket and UNIX domain socket. They will use these sockets either to communicate with each other or to send data to GUI and the simulation engine. As we know, all of these sockets are provided by most UNIX-like systems including FreeBSD 4.x and Linux 2.4.x. So we just need to recompile these programs on Linux 2.4.x system and then they will correctly work for the NCTUns

1.0.

The daemons that are provided by the NCTUns 1.0 contain Mobile-IP daemons, emulation daemons and routing daemons (OSPF and RIP routing daemon). Routing daemons are independent of UNIX-like system. For the Mobile-IP daemons and emulation daemons, they need some kernel support. They want the kernel to provide a mechanism to enable them to capture packets from the kernel to the user space. These daemons use this mechanism to capture packets and put them back into the kernel after modifying some field of packets. In FreeBSD 4.x, they can use BPF (Berkeley Packet Filter) and divert socket to complete this task. In Linux 2.4.x, the kernel does not support the divert socket. However, Linux provides a similar mechanism to do this work. They are the shared library “libipq” and the netfilter, iptables and ip_queue [5] kernel modules. Unfortunately, this approach can not completely satisfy these daemons’ requirement. To overcome this problem, we bring FreeBSD’s divert socket into Linux. In section 5.2.2 of part II, we will clearly explain why the original mechanism of Linux can not work and then describe our approach.

Examples of real-life programs include stcp/rtcp, stg/rtg, ttcp, Apache Web server, Web browser, FTP daemon, FTP client, telnet daemon, telnet client, etc (Stcp/rtcp, stg/rtg, and ttcp are general purpose traffic generators provided by the NCTUns 1.0). Due to our novel simulation methodology, all real-life application programs can run on our simulation network without any special modification. They only need to be recompiled on the Linux system and then they can directly run on the Linux version of the NCTUns 1.0 network simulator.

4.2 Simulation Engine

In section 3.3 of part I, we clearly describe the architecture of the simulation engine. Some components of the simulation engine are independent of the operating platform while some components are dependent. In this section, we will discuss these components one by one and compare their differences.

4.2.1 Independent Components

Script interpreter, module manager, command dispatcher, scheduler, NCTUns APIs, IPC and protocol modules belong to independent components. Their functions are independent of the used operating system. Therefore, we do not need to modify any of them.



4.2.2 System Calls

In figure I-3.3.1, we can see that the simulation engine has to ask the kernel for some services via added system calls. We will describe all of the system calls that are added or modified by us in section 4.3.4 of part I.

Because these system calls numbers registered in FreeBSD and Linux are completely different, we should decide which one will be used at compile time.

Following is an example code segment:

```
typedef struct portinfo {
    int portnumber;
};

#ifdef LINUX
    syscall(261, 0x06, nid, 0, &portnumber);
#else
    syscall(290, 0x06, nid, 0, &portnumber);
#endif /* LINUX */
```

We define the “LINUX” macro to enable the preprocessor to recognize which code

segment is used under the Linux version and which is used under the FreeBSD version. At compile time, we can control the macro “LINUX” whether it is enabled or not, and then the compiler (g++) can know that the simulation engine will be compiled to a Linux or FreeBSD version. As such, we just need to maintain a single source tree of the simulation engine. This approach also reduces the effort of maintenance.

4.2.3 Memory Mapping

The simulation engine uses the memory mapping technique to map some memory location that is allocated by the simulation engine process to a memory location in the kernel. There are two memory locations that should be mapped into the kernel. The first one is the virtual clock which we can call it the virtual time in the rest of this thesis.



The virtual time is maintained by the simulation engine. However, the simulation engine is a user-level program. Therefore, the virtual time is kept at the user level. However, the kernel also has to reference the virtual time at the kernel level. This is required for many reasons. First, the timers of TCP connections used in the simulation network need to be triggered based on the virtual time rather than the real time (recall that in the NCTUns 1.0, the in-kernel TCP/IP protocol stack is directly used to “simulate” TCP connections). Second, for those application programs launched to generate traffic in the simulation network, the time-related system calls issued by them must be serviced based on the virtual time rather than the real time. Third, the in-kernel packet logging mechanism needs to use time stamps based on the virtual time to log packets transferred in a simulation network. To achieve these goals,

actually, the simulation engine can periodically use a user-defined system call to inform the kernel of the current virtual time. However, the cost of this approach would be too high if we want the virtual time maintained in the kernel to be as precise as that maintained in the simulation engine. With the memory mapping technique, at any time the virtual time in the kernel now is as precise as that maintained in the simulation engine without any system call overhead.

The second places are the current and the maximum queue length of the FIFO (First-In-First-Out) module. It is used to enable the tunnel interfaces (tun1~tun4095) to timely drop a packet when the FIFO queue (in the FIFO module) is full. This operation is very important because when a TCP packet is dropped due to a queue full, the TCP socket handler would start the source quench algorithm to reduce its TCP congestion window size to avoid further packet dropping. Without mapping these values into the kernel, the tunnel interface does not know whether the FIFO queue is full or not. If the tunnel interface does not timely drop the packet, a lot of packets will be successively dropped in the simulation engine during the TCP slow-start phase. This will severely affect a TCP connection's throughput.

In FreeBSD, a user-level program can use “*kvm*” (Kernel Virtual Memory) interface to find the address offset of a variable in kernel memory space and this interface is included in a library: *libkvm*. Using the mechanism, the simulation engine can open the special device `/dev/kmem` and use the `mmap()` system call to do memory mapping task. The following code segment shows how to map the virtual time variable (*currentTime_*) into the kernel space (*NCTUNS_nodeVC*, the virtual time maintained in the kernel).

```

#include <kvm.h>
#include <nlist.h>

u_int64_t *currentTime_;

int tun_mmap()
{
    kvm_t *kd;
    int fd;
    off_t tick_offset;
    struct nlist nl[] = {
        {"NCTUNS_nodeVC"},
        {NULL},
    }

    kd = kvm_open(NULL, NULL, NULL, O_RDONLY, errstr);
    if (kvm_nlist(kd, nl) < 0) {
        printf("nctuns: %s\n", errstr);
    }
    tick_offset = nl[0].n_value;

    if((fd=open("/dev/kmem", O_RDONLY)) < 0) {
        printf("nctuns: open /dev/kmem error!\n");
        exit(-1);
    }
    currentTime_ = (u_int64_t *)mmap(0, sizeof(u_int64_t)*(ifcnt_+1),
        PROT_READ|PROT_WRITE, MAP_PRIVATE, fd, (off_t)tick_offset);
    ...
}

```



In Linux, it does not provide a library like libkvm in FreeBSD to enable the user-level program to get the virtual address of a global variable in the kernel space. Fortunately, this operation can be done via reading the file “*System.map*” that will be produced when a kernel image is built. It records global variables and functions with their corresponding virtual addresses. The following shows a part of the *System.map* file.

```

c03ad058 b dummy_task
c03ad080 B procBaseOnVirtualTime
c03cd07c B mtable
c03cd080 B NCTUNS_nodeVC
c03d5080 B pm_active
c03d5080 B pm_active
c03d50a4 B num_physpages

```

Third column lists the variable or function name and the first column is their

corresponding virtual addresses. For example, the NCTUNS_nodeVC global variable's virtual address is c03cd080 (it is hexadecimal). Therefore, the memory mapping mechanism in Linux is now translated to the following code segment:

```

u_int64_t *currentTime_;

#define OFFSET 0xc0000000
#define PTR_POS(Val) ((unsigned long)(Val))
int tun_mmap()
{
    int fd, offset;
    off_t tick_addr;
    u_int64_t *tick;

    if((fd=open("/dev/kmem", O_RDONLY)) < 0) {
        printf("nctuns: open /dev/kmem error!\n");
        exit(-1);
    }

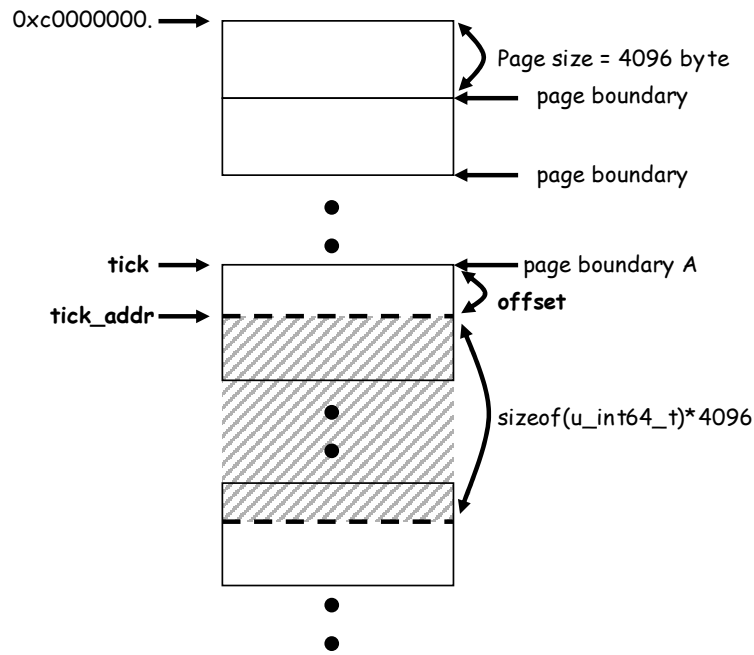
    tick_addr = search_kvm("NCTUNS_nodeVC");
    offset = PTR_POS(tick_addr-OFFSET) -
        (PTR_POS(tick_addr-OFFSET) & ~4095);
    tick = (u_int64_t *)mmap(0, PTR_POS(offset) + sizeof(u_int64_t)*4096,
        PROT_READ|PROT_WRITE, MAP_SHARED, fd,
        (PTR_POS(tick_addr-OFFSET) & ~4095));
    currentTime_ = (u_int64_t*)(PTR_POS(tick)+PTR_POS(offset));
    ...
}

```

To search a variable name from System.map and return its virtual address



For the memory mapping mechanism in Linux, the kernel only allows the address of a page boundary to be mapped to the user space. However, for any global variable in the kernel, its address may not be just a page boundary. We use the following figure to show the detail about the above example code.



The starting address of the kernel space is 0xc0000000 and the default page size is 4096 bytes. First, we will use `search_kvm()` to find out the address of the specified variable in the kernel. In the above case, `tick_addr` will store the address of the specified variable, `NCTUNS_nodeVC`. Second, we will calculate `offset`, which is the difference between the page boundary A and `tick_addr`. After the simulation engine issues the `mmap()` system call, the kernel will return the address of the page boundary A and the value will store in `tick`. Finally, the simulation engine simply adds `tick` and `offset` to get the desired address, `tick_addr`.

4.2.4 Process Scheduling

Correctly scheduling the simulation engine process and all forked traffic generator processes is very important for the event-driven approach to function correctly. This is because the default UNIX process scheduler uses a dynamic priority mechanism and thus cannot guarantee a desired scheduling order to happen. In reference [2], it clearly explains which scheduling order is correct to us and how to use the default UNIX process scheduling design to meet our requirement. Of course,

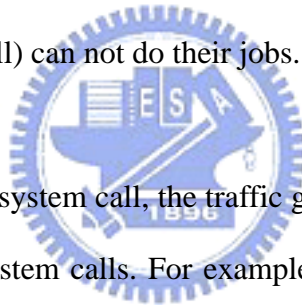
all discussion in [2] can only suit the FreeBSD system. In Linux, the process scheduling mechanism is very different from that in FreeBSD. In FreeBSD, a user-level process has two priorities: one for its execution in the user mode and the other is for its sleeping in the kernel mode. The special design for the kernel mode priority of a process is the key point which lets our network simulator work correctly [2]. However, Linux does not have a similar design.

In Linux, a user-level process also has two priorities: static priority and dynamic priority [12]. The static priority is assigned by the users for real-time processes and never changed by the process scheduler in the kernel. The dynamic priority is used for normal processes and is essentially the sum of the base time quantum (which is therefore also called the base priority of the process) and of the number of ticks of CPU time left to the process before its quantum expires in the current epoch. The static priority of a real-time process is always higher than the dynamic priority of a normal one. The scheduler will run normal processes only when there is no real-time process in the runnable state.

Let's consider the following situation. A UDP traffic generator A sends a packet and calls the sleep() system call and gets blocked in the kernel. At the same time, another UDP traffic generator B also wants to send packets. If the process scheduler switches the CPU control from A to the simulation engine instead of B, the simulation engine may immediately advance the virtual time after processing A's packets and events. This will cause that B has no chance to send packets before the virtual time is advanced. Therefore, the correct execution order should be A -> B -> the simulation engine. To overcome this problem, the simulation engine sets a traffic generator as a real-time process when forking it. At the same time, the simulation engine is still a

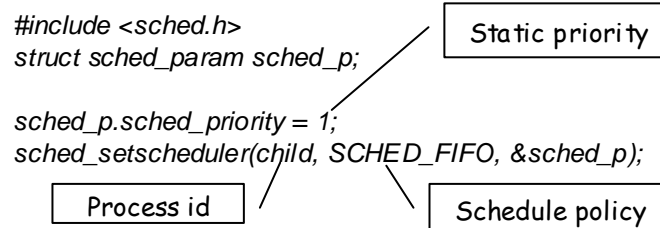
normal process. In other words, the traffic generator's priority is always higher than the simulation engine. Therefore, only when A and B both get blocked in the kernel mode, the simulation engine will be able to get CPU control. With this arrangement, we can get the desired execution order.

However, this approach has a drawback -- the traffic generator must not have infinite loop statements. Because the traffic generator has a very high priority (static priority) and the process scheduler in the kernel will never change the priority, it can continually execute until get blocked in the kernel. If the traffic generator does not get blocked any time, it will continually hold the CPU control and other processes will not have a chance to execute. If this happens, the whole system will be hung because other processes (e.g. login shell) can not do their jobs.



In addition to the `sleep()` system call, the traffic generator may get blocked in the kernel when issuing other system calls. For example, when a TCP traffic generator calls the `read()` system call, it may get blocked in the kernel due to an empty socket receive buffer. Other system calls such as `select()`, `write()`, `connect()` etc. also may cause the process to get blocked. The traffic generator programmer should carefully check that there is no busy waiting or infinite loop in his/her program.

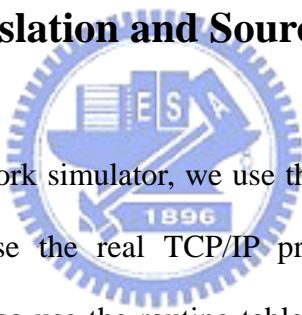
In Linux, users can use `sched_setscheduler()` system call to set a process as a real-time process. The following is an example code segment:



4.3 Kernel Modifications

In this chapter, we will clearly describe which parts of the Linux kernel need to be modified so that it can be used to the NCTUns 1.0 network simulator.

4.3.1 IP Address Translation and Source-Destination-pair IP Scheme



In the NCTUns 1.0 network simulator, we use the kernel re-entering simulation methodology. We directly use the real TCP/IP protocol stack provided by the FreeBSD/Linux kernel. We also use the routing table that is maintained in kernel to automatically forward IP packets through a simulated layer-3 router [1]. Due to these reasons, we propose a special IP scheme “S.S.D.D” (source-destination-pair) to complete this job. Suppose that the sending node has a tunnel interface that is assigned an IP address 1.0.A.B and the receiving node has a tunnel interface with an IP address 1.0.C.D, then the used source-destination-pair address will be A.B.C.D. When the sending node wants to send a packet to the receiving node, the packet’s destination IP address needs to be translated from 1.0.A.B to A.B.C.D. At the same time, the source IP address will be translated to A.B.A.B based on the sending node’s viewpoint. When the packet reaches the receiving node, the destination IP address will be modified to C.D.C.D and the source IP address will be modified to C.D.A.B

based on the receiving node's viewpoint.

Figure I-4.3.1 shows the change process about the destination and source IP address of the packet that is traversed from host 1 to host 2 in figure I-3.1.3. In this section, we will describe when to modify the destination IP address of packet A to S.S.D.D format and when to translate the source IP address of packet H to the natural IP format (1.0.X.Y). In next section (4.3.2), we will describe how to translate these IP addresses of packet B, C, D, E, F, and G in tunnel interface 1 ~ 4.

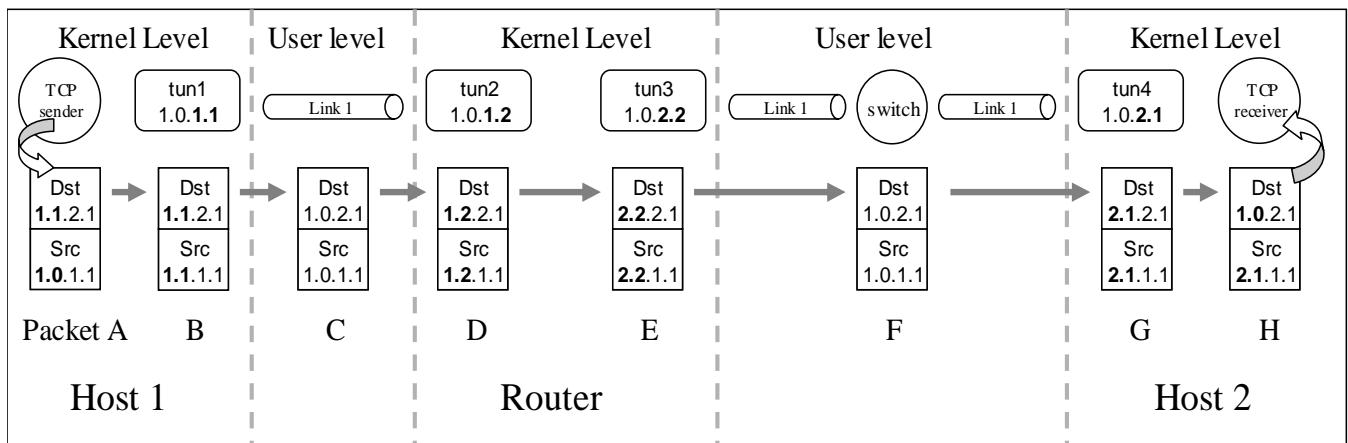


Figure I-4.3.1: IP address translation

When a process was forked by the simulation engine, the simulation engine will use system call 261 to store the simulated node id into the process handler. This information allows the kernel to know whether a process is used by a simulation or not. The structure *task_struct* is modified as below:

```

struct task_struct {
    ...
    pid_t          pid;
    ...
    struct signal_struct *signal;
    ...
//NCTUNS
    /* record the process belongs to which node */
    unsigned int   p_node;
//NCTUNS
}

```

If a process creates an INET socket such as TCP, UDP, RAW socket, we should also register the node id into the corresponding INET socket structure, which is structure *sock*. The *pmap* field of the structure *sock* is used to record the information about the mapping between the virtual and real port number of a TCP or UDP INET socket. In section 4.3.3 of part I, we will see the definition of the structure *pmap*.



```

struct sock {
    __u32          daddr;
    __u32          rcv_addr;
    ...
//NCTUNS
    u_int32_t      nodeID; /* record the process belongs to which node */
    unsigned short sk_vport; /* virtual port number */
    struct pmap     *pmap; /* point to port mapping information */
//NCTUNS
}

```

For the datagram INET socket such as UDP and RAW socket, we store the node id into the INET socket structure (*sock*) when a process calls the *socket()* system call:

```

Asmlinkage long sys_socket (int family, int type, int protocol)
{
    int          retval;
    struct socket *sock;

    retval = sock_create(family, type, protocol, &sock);
    ...
//NCTUNS
    /* If current process belongs to a simulation,
       we should store node id into sk. */
    if (current->p_node > 0) {
        sock->sk->nodeID = current->p_node;
    } else {
        sock->sk->nodeID = 0;
        sock->sk->sk_vport = 0;
    }
//NCTUNS
    ...
}

```

For the stream INET socket such as TCP, we have to store the node id when it is initialized at `tcp_v4_init_sock()`. Because `tcp_v4_init_sock()` will initialize several TCP timers such as retransmit timer, delay-ack timer etc., the kernel will determine whether these timers should use virtual time or real time based on the value of `sk->nodeID`. If we do this operation at `sys_socket()`, initializing these timers will be too early because the node id is not known yet. It will cause these timers to be based on the real time instead of the virtual time.

```

static int tcp_v4_init_sock (struct sock *sk)
{
    ...
//NCTUNS
    if (current->p_node > 0) {
        sk->nodeID = current->p_node;
        tp->nodeID = current->p_node;
    } else {
        sk->nodeID = 0;
        tp->nodeID = 0;
    }
//NCTUNS
    ...
}

```

With the node id information, we can correctly translate the IP address in the kernel. In figure I-4.3.1, when the TCP sender sends out a packet at the connection setup

phase, we will translate the destination IP address at *inet_stream_connect()*, which is shown below. In *inet_stream_connect()*, we should explain the usage of *mt_randnidtoip()*. If a simulated node is a multi-interface device (e.g., a router), it may have several tunnel interfaces and each tunnel interface has an assigned IP address. When the node sends out a packet, the kernel should choose an IP address as the node's source IP address. Then the kernel will use this IP address to translate the IP address of the packet to the S.S.D.D format.

```

int inet_stream_connect (struct sock *sk, struct sockaddr *uaddr,
                        int addr_len, int flags)
{
    ...
//NCTUNS
    /* modify the dst IP from 1.0.X.X to S.S.D.D format */
    if (sk->nodeID > 0) {
        struct sockaddr_in *sin;
        u_long          srca;
        u_char          *ptr, *ptr1;
        sin = (struct sockaddr_in *)uaddr;
        ptr = (u_char *)&(sin->sin_addr.s_addr);
        if (ptr[0]==1 && ptr[1]==0) {
            srca = mt_randnidtoip(sk->nodeID);
            if (srca != 0) {
                ptr1 = (u_char *)&srca;
                ptr[0] = ptr1[2];
                ptr[1] = ptr1[3];
            } else {
                printk("inet_stream_connect(): mt_randnidtoip() error!\n");
            }
        }
    }
//NCTUNS
    ...
}

```

During the data transfer phase, we will translate the destination IP address at *inet_sendmsg()*:


```

int inet_sendmsg (struct socket *sock, struct msghdr *msg,
                  int size, struct scm_cookie *scm)
{
    struct sock *sk = sock->sk;
//NCTUNS
/* modify the dst IP from 1.0.X.X to S.S.D.D format */
{
    struct sockaddr_in *addr=(struct sockaddr_in *)msg->msg_name;
    struct sockaddr_in *dsa;
    u_long          srcip;
    u_char          *dsta, *srca;
    if (addr && sk->nodelD>0 ) {
        dsa = addr;
        dsta = (u_char *)&(dsa->sin_addr.s_addr);
        if (dsta[0]==1 && dsta[1]==0) {
            srcip = mt_randnidtoip(sk->nodelD);
            if (srcip != 0) {
                srca = (u_char *)&srcip;
                dsta[0] = srca[2];
                dsta[1] = srca[3];
            } else
                printk("inet_sendmsg(): mt_randnidtoip() error!\n");
        }
    }
}
//NCTUNS
...
}

```



For datagram socket, it also uses `inet_sendmsg()` to send packets out. As such, a UDP or RAW packet's destination IP address will also be modified at `inet_sendmsg()` during the data transfer phase.

So far, we describe where we modify packet A's (figure I-4.3.1) destination IP address from 1.0.2.1 to 1.1.2.1. In the following, we will describe where we translate packet H's source IP address from 2.1.1.1 back to 1.0.1.1. We will recovery the source IP address at `inet_rcvmsg()`:

```

int inet_recvmmsg (struct socket *sock, struct msghdr *msg,
                  int size, struct scm_cookie *scm)
{
    struct sock *sk = sock->sk;
    ...
//NCTUNS
    /* To recovery src IP from S.S.D.D to 1.0.X.X */
    struct sockaddr_in *addr = (struct sockaddr_in *)msg->msg_name;

    /* Don't recovery RAW socket's IP address.
       We will recovery it at raw_recvmmsg(). */
    if (addr && sk->nodelD>0 &&(strncmp(sk->prot->name, "RAW", 3)!=0) {
        u_char *p;
        p = (u_char *)&(addr->sin_addr.s_addr);
        p[0] = 1; p[1] = 0;
    }
//NCTUNS
    ...
}

```

We do not recovery RAW socket's packets here. We do this task at *raw_recvmmsg()*. This is because we design a mechanism that can turn a RAW socket into a divert socket. We will further discuss this in section 4.3.8 of part I.



4.3.2 Tunnel Interface

There are two jobs that should be done in a modified tunnel driver.

I. Enable S.S.D.D IP Scheme to Work Correctly

Figure I-4.3.1 shows the IP address change history of a packet that traverses the simulation network from host 1 to host 2. When the kernel inserts a packet into a tunnel interface's output queue, *tun_net_xmit()* will be called to do this job. In figure I-4.3.1, the kernel calls *tun_net_xmit()* to insert packet A into tun1. Then *tun_net_xmit()* will then modify packet A to packet B before queuing it. The following is the pseudo code of *tun_net_xmit()*:

- * Tunnel interface Y has assigned IP address 1.0.A.B.
- * Packet X has destination/source IP address pair A.B.C.D/1.0.A.B.

```

tun_net_xmit (packet X, tun Y)
{
  1. Modify A.B.C.D/1.0.A.B to A.B.C.D/A.B.A.B
  2. If A.B = C.D
      modify A.B.C.D/A.B.A.B to 1.0.C.D/A.B.A.B
      put the packet back to network layer
  3. else
      modify A.B.C.D/A.B.A.B to 1.0.C.D/1.0.A.B
      queue the packet into output queue
      insert a tunnel check event into t0e0
}

```

If A.B is equal to C.D in step 2, it means packet X already arrives in the destination node. Therefore, we should modify the destination IP address from A.B.C.D to 1.0.C.D to avoid that the packet would be further forwarded at the network layer (When we add routing entries into the system routing table, we use S.S.D.D format. If we do not recovery the destination IP address to 1.0.X.X format, it will be subjected to unnecessary forwarding.). We also can know that this kind of packet (A.B = C.D at step 2) is a loopback packet (The source node is the same with the destination node.).

If packet X is not a loopback packet, we should recovery its destination/source IP address to 1.0.X.X. This is because that we hope that the packet has the normal IP address format when it is read by the simulation engine. As such, the protocol modules don't need to know anything about the S.S.D.D IP scheme. Therefore, packet C and F in figure 4.3.1 have the normal IP address format. Finally, we should create a tunnel check event to inform the simulation engine that the tunnel interface Y has packets to send. If we do not use this method, the simulation engine should periodically poll all tunnel interfaces to check whether there is any packet to send or not. Of course, the performance of the polling approach will be much slower than our

new design.

When the simulation engine wants to put a packet into a tunnel interface (e.g. put packet C to tunnel interface 2 in figure 4.3.1), it will issue a write system call and then *tun_get_user()* will be called to copy the packet from the user-space to the kernel-space and to queue the packet to the network layer. The following is the pseudo code of *tun_get_user()*:

```
* Tunnel interface Y has assigned IP address 1.0.A.B.  
* Packet X has destination/source IP address pair 1.0.C.D/1.0.E.F.  
  
tun_get_user (packet X, tun Y)  
{  
    1. Modify 1.0.C.D/1.0.E.F to A.B.C.D/A.B.E.F  
    2. If A.B = C.D  
        modify A.B.C.D/A.B.E.F to 1.0.C.D/A.B.E.F  
    3. put the packet to the network layer  
}
```

If A.B is equal to C.D at step 2, it means that the packet has arrived at its destination node. Of course, the packet's destination IP address will need to be returned to the 1.0.X.X format. In the other case, we let the packet's IP address reserve the S.S.D.D format to be further forwarded. In figure I-4.3.1, readers can compare packet D and packet G to clearly understand the difference between them. When packet D is put to the network layer, we can add a routing entry to specify tun3 as the gateway of a packet whose destination IP address is 1.2.2.X. Then packet D will be queued into tun3 via *tun_net_xmit(D, tun3)*. This shows that if we can correctly add these routing entries, packets can be automatically forwarded by simulated routers.

II. Support Tcpdump Module

When a packet passes through a tcpdump module, the tcpdump module will clone a copy and write the copy into the tunnel interface 0 (tun0). This is because we hope the kernel to log the packet when it passes through the tcpdump module rather than when it passes through the tunnel interface (tun1~tun4095).

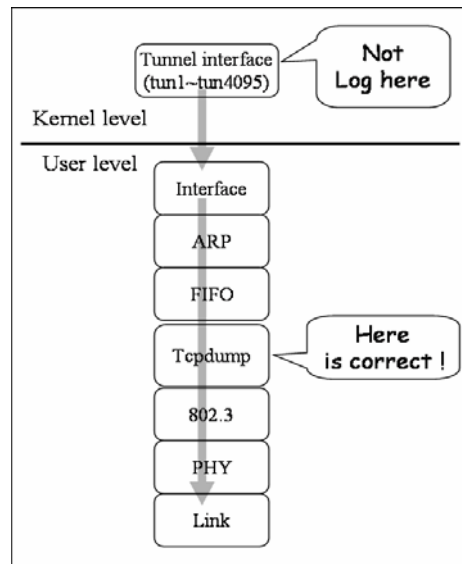


Figure I-4.3.2: Tcpdump module

There are several reasons why we should do this operation. One is to get the correct timestamp of outgoing or incoming packets. On a real-life machine, the timestamp given to an outgoing packet represents the time when the packet is transmitted to a link rather than the time when the packet is enqueued into the output queue. However, in the current design, if there is no modification, a packet will receive a timestamp that represents the time when it leaves the tunnel interface rather than when it is transmitted to a link. Another reason is that we can not log the ARP request and reply packet. Figure I-4.3.2 clearly shows this problem.

The following will describe how we achieve the above requirement. First, we should not log packets when they pass through tunnel interfaces tun1~tun4095. The following is the modified code segment:

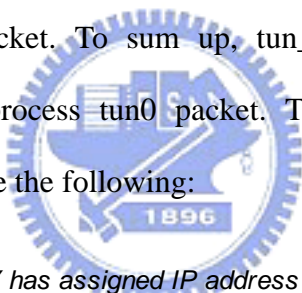
```

static int packet_rcv (struct sk_buff *skb, struct net_device *dev,
                      struct packet_type *pt)
{
    ...
    struct packet_opt *po;
    u8 *skb_head = skb->data;
    int skb_len = skb->len;
#ifdef CONFIG_FILTER
    unsigned snaplen;
#endif

//NCTUNS
    if ( (strcmp(dev->name, "tun", 3)==0) &&
         skb->pkt_type!=PACKET_TUN0)
        return 0;
//NCTUNS
    ...
}

```

Second, when the tcpdump module writes a packet into tun0, tun_get_user() will be called to process the packet. To sum up, tun_get_user() has two missions: translation IP address and process tun0 packet. Therefore, the pseudo code of tun_get_user() should look like the following:



```

* Tunnel interface Y has assigned IP address 1.0.A.B.
* (If Y = tun0, Y does not has a assigned IP.)
* Packet X has destination/source IP address pair 1.0.C.D/1.0.E.F.

tun_get_user (packet X, tun Y)
{
    1. if Y = tun0
        find X belongs to tunZ
        call nctuns_packet_rcv(X, Z)
    2. else
        Modify 1.0.C.D/1.0.E.F to A.B.C.D/A.B.E.F
        If A.B = C.D
            modify A.B.C.D/A.B.E.F to 1.0.C.D/A.B.E.F
        put the packet to network layer
}

```

If the caller is tun0, it means that we have to direct the packet to the in-kernel packet logging mechanism module. The *nctuns_packet_rcv()* function is implemented to do this job. When a network monitor program such as tcpdump is running to monitor a tunnel interface, the program will create a *SOCK_PACKET* socket to

receive the log packet. `Nctuns_packet_rcv()` is used to queue the packet to the corresponding `SOCK_PACKET` socket.

No matter which tunnel interface (`tun1 ~ tun4095`) the packet passes through, it will be cloned and be inserted to `tun0` while it passes through a `tcpdump` module. As such, `tun0` will receive packets from different tunnel interfaces. Therefore, `tun_get_user()` should know where a packet come from before calling `nctuns_packet_rcv()`. To help `tun_get_user()`, the `tcpdump` module will add a tag that records where the packet comes from to the packet when it writes the packet into `tun0`.

The following shows the `nctuns_packet_rcv()`:

```
void nctuns_packet_rcv(struct sk_buff *skb, struct net_device *dev)
{
    struct packet_type *ptype;

    if (strcmp(dev->name, "tun", 3) == 0
        && skb->pkt_type==PACKET_TUN0
        && dev->flag&IFF_UP )
        getvirtualltime(&skb->stamp);
    } else {
        kfree_skb(skb);
        return;
    }

    br_read_lock(BR_NETPROTO_LOCK);
    for (ptype = ptype_all ; ptype!=NULL ; ptype = ptype->next) {
        if ((ptype->dev == dev || !ptype->dev) &&
            ((struct sock *)ptype->data != skb->sk) ) {
            skb->mac.raw = skb->data;
            if (skb->nh.raw < skb->data || skb->nh.raw > skb->tail)
                skb->nh.raw = skb->data;
            skb->h.raw = skb->nh.raw;
            skb->pkt_type = PACKET_TUN0;
            ptype->func(skb, skb->dev, ptype);
        }
    }
    br_read_unlock(BR_NETPROTO_LOCK);
}
```

The packet's timestamp should use the virtual time.

To find whether a `SOCK_PACKET` socket is listening on the specified network device. If yes, the kernel will call its handler function, `ptype->func`, to deal with this packet.

4.3.3 System Calls

The system calls added by us are shown below:

I. System call number 259

System call 259:

```
asmlinkage int sys_NCTUNS_divert(action, fd, hook, de, addr_len)
```

Parameters:

```
int          action;  
int          fd;  
u_long      hook;  
struct divert_entry *de;  
int          addr_len;
```

The system call is used to turn a RAW socket into a divert socket and download the IP packet filter rule into the kernel. In FreeBSD, the kernel supports the divert socket type but Linux does not. However, some daemons such as Mobile-IP daemons and emulation daemons need to use the divert socket. In section 5.2.2 of part II, we will clearly explain why these daemons need to use divert sockets. In section 4.3.8 of part I, we will describe how we use this system call and other added functions to turn a RAW socket into a divert socket.

II. System call number 260

System call 260:

```
asmlinkage int sys_NCTUNS_clearStateAndReinitialize(nctuns_pid)
```

Parameters:

```
int          nctuns_pid;
```

This is a very important operation before starting a new simulation. Every time we start a new simulation, we must be sure that all states left by the previous simulation should be cleared. Otherwise, the previous states or information may affect the accuracy of the new simulation. For example, we should clear the timers left in the callout wheel due to the previous simulation. And we also should clear the IP

packet filter rules which are set by the previous simulation. This system call also re-initializes some variables used by the NCTUns 1.0. For example, we should set the current virtual time to zero and register the new process id of the simulation engine.

III. System call number 261

System call 261:

asm linkage int sys_NCTUNS_misc (action, value1, value2, value3)

Parameters:

int action;
unsigned long value1;
unsigned long value2;
unsigned long value3;

This system call is used to set or get various information maintained in the kernel about our simulator. The “action” argument determines the service types, which are shown below:

NSC_TEST:

This is used to display some kernel information about the simulator. For example, we can dump all TCP socket information which is used in a simulated node that is inside a simulation network.

NSC_GETNIDINFO:

This is used to get the IP address configuration of each tunnel interface that belongs to the same simulated node.

NSC_REGPID:

This is used to enable the process that is forked by the simulation engine to know that it belongs to which simulated node.

NSC_NIDTOTID:

For example, a simulated node has three tunnel interfaces: tun1, tun2 and tun3. Then, this action will copy the tunnel id 1, 2, 3 to the buffer specified

by the calling process.

NSC_NIDNUM:

This is used to return how many interfaces that a specified simulated node has.

NSC_R2VPORT:

This is used to translate a real port number to a virtual port number.

NSC_V2RPORT:

This is used to translate a virtual port number to a real port number.

NSC_TICKTONANO:

This is used to download the used time scale value into the kernel. Because the NCTUns 1.0 can support different time scale (1 nanosecond/tick, 10 nanosecond/tick, or 100 nanosecond/tick), we should let the kernel know which time scale will be used in a new simulation. In section 4.3.5 of part I, we will clearly describe how the kernel deal with the time scale.

NSC_SET_TUN:

This is used to set the state of the specified tunnel interface to up or down.

NSC_GET_TUN:

This is used to get the state of the specified tunnel interface -- up or down.

IV. System call number 263

System call 263:
asmlinkage int sys_NCTUNS_callout_chk (void)
Parameters:
none

This system call is used to trigger the kernel timeout events. When the event scheduler (in the simulation engine) processes a kernel timeout event, it issues this system call to inform the kernel that a kernel timer used in a simulation should be

triggered right now.

V. System call number 264

System call 264:

```
asmlinkage int sys_NCTUNS_mapTable (action, nid, tid, mac, s_port)
```

Parameters:

```
int          action;
unsigned long nid;
unsigned long tid;
char        *mac;
unsigned short s_port;
```

We maintain a data structure in the kernel to record all of the information about a simulation network and the usage of each tunnel interfaces. The data structure, which we name it *mtable*, is shown as below:

```
struct if_info {
    SLIST_ENTRY(if_info)  nextif;
    u_long                tunid;
    u_long                tunip;
    u_char                mac[6];
    struct in_ifaddr      *ifa;
}

struct pmap {
    SLIST_ENTRY(pmap)    nextmap;
    struct sock          *sk;
    struct proto         *prot;
    u_short              rport;
    u_short              vport;
    int                  unbind;
}

struct node_info {
    SLIST_HEAD(,if_info) ifinfo;
    SLIST_ENTRY(node_info) nextnode;
    SLIST_HEAD(,pmap) pmap_head;
    u_long            nodeID;
    u_short           s_port;
    u_long            numif;
    u_short           last_port;
}
```

The *if_info* is a data structure used to record the information about a tunnel interface (e.g. IP and MAC address of an interface). The *pmap* is used to record the

information about the mapping between the virtual and real port number of a TCP or UDP INET socket. The *node_info*, which is the *mtable*, is used to record the information about a simulated node. Reference [13] clearly describes the detail architecture of these structures. System call 264 helps the simulation engine to maintain these nodes' information in the kernel. The following are the operations provided by system call 264:

MT_FLUSH:

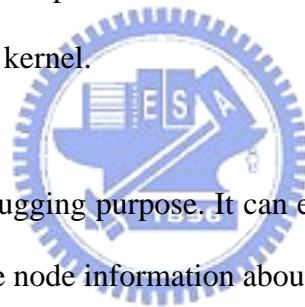
Before a new simulation starts, the simulation engine should use this operation to flush all node information of the previous simulation.

MT_ADD:

This operation can help the simulation engine to download new node information into the kernel.

MT_DISPLAY:

This is used for debugging purpose. It can enable the programmer (or user) to check whether the node information about a simulation is correct or not.



4.3.4 Port Number Mapping and Translation

In section 3.4 of part I, we already explain why we have to perform port number mapping in the kernel. Here, we will clearly describe where to do this job.

I. Record the mapping of the virtual and real port number

First, we define the virtual port and real port number. The port number used by user-level program is called as a virtual port. For example, when a program issues the *bind()* system call to bind a INET socket into per-protocol socket list, it should specify a port number. We call this port number a virtual port number. In the kernel,

there are several kernel functions to service the `bind()` system call. When these kernel functions want to use the virtual port number to bind a socket, we will instead find an unused real port number to bind the socket and record this port mapping. The port number which is really used by the kernel is the real port number according to our definition. For example, a Web server A may be running on simulated node 1 and listening on the default port number 80. Another Web server B may be running on the simulated node 2 and also listening on port 80. Both port numbers (80) are virtual port numbers. Actually, the kernel may use port number 5000 to bind the socket that is created by A, 5001 for the socket created by B. In this case, 5000 and 5001 are called real port numbers. We should record the real/virtual port mapping pair (80, 5000) in node 1 and (80, 5001) in node 2 for later uses.

The kernel functions about TCP and UDP sockets are `udp_v4_get_port()`, `tcp_v4_get_port()`, `tcp_v4_hash_connect()`. Their jobs are to bind an UDP or TCP socket into individual socket list. Here, we only take `udp_v4_get_port()` as an example to show how we do a port number mapping:

```

static int udp_v4_get_port (struct sock *sk, unsigned short snum)
{
    ...
//NCTUNS
    /* 1. If virtual port (snum) is not zero, record it and set it to zero.
       Then use original procedure to get a real port.
       2. Otherwise, if virtual port (snum) is zero, choose one by ourselves.
       And set snum to zero to let original procedure to get a real port.*/
    if (sk->nodelD) {
        int nid= sk->nodelD;
        if (snum) {
            if (!mt_lookupVport(nid, snum)) {
                sk->sk_vport = snum;
                snum = 0;
            } else {
                goto fail;
            }
        } else {
            sk->sk_vport = mt_getunusevport(nid);
        }
    }
//NCTUNS
    if (snum == 0) {
        ...
    } else {
        ...
    }
    ...
}

```

To check whether the virtual port (snum) has been used in nid or not.

Original procedure



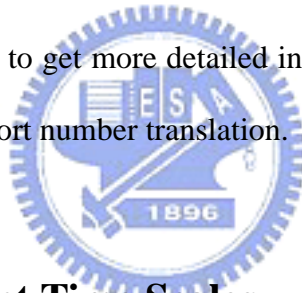
II. Perform port number translation

Let's consider a situation: a Web server A is listening on the virtual port number 80 and actually it uses a real port number 5000 in the kernel. A Web browser B wants to send a request to A and specifies the foreign port number 80 because B does not know anything about the real port number. Therefore, the destination port number of the request packet will be 80. Then, when the request packet arrives at the destination node and the kernel attempts to look for the socket that will accept the request, it will fail. This is because, actually, A is listening on the port number 5000 rather than 80. As such, before the kernel attempts to look for the listening socket, it should translate the destination port number to the corresponding real port number. For TCP and UDP protocol, their main receiving handle functions are *tcp_v4_rcv()* and *udp_rcv()*. Here,

we use `udp_rcv()` as an example:

```
int udp_rcv (struct sk_buff *skb)
{
    ...
//NCTUNS
    nodeIDd = mt_ptonid1(daddr, skb->dev);
    if (nodeIDd > 0) {
        /* If the packet belongs to virtual connection,
        we should follow the following rules:
        src: R -> V (real port -> virtual port)
        dst: V -> R (virtual port -> real port) */
        rport = mt_VtoRport(nodeIDd, ntohs(uh->dest));
        if (rport > 0)
            uh->dest = htons(rport);
        vport = mt_RtoVport(ntohs(uh->source));
        if (vport > 0)
            uh->source = htons(vport);
    }
//NCTUNS
    ...
}
```

Readers can refer to [13] to get more detailed information about the mechanism of IP address translation and port number translation.



4.3.5 Support Different Time Scales

The simulation engine supports several different time scales (1 nanosecond/tick, 10 nanosecond/tick and 100 nanosecond/tick). 1 nanosecond per tick (1 nanosecond/tick) means that 1 virtual clock corresponds to 1 nanosecond in virtual time. 10 nanosecond/tick means that 1 virtual clock corresponds to 10 nanosecond in virtual time. Of course, the kernel should know which time scale is used in a new simulation. Several related definitions and global variables are shown below:

```

// The virtual clock of nodes. In current version, we only use node 0's
// clock as the whole simulation system's clock.
u_int64_t NCTUNS_nodeVC[MAX_NUM_NODE];

// For 64bit division
long NCTUNS_rem;

/* Microscale, default 10.
   Because default time scale is 100, the default microscale would be 1000/100.
   With microscale, the unit of NCTUNS_nodeVC[0]/microscale would be microsecond. */
unsigned long microscale = 10;

#define NCTUNS_ticks_to_us \
    div_long_long_rem(NCTUNS_nodeVC[0], microscale, &NCTUNS_rem)

#define NCTUNS_ticks_to_ms \
    div_long_long_rem(NCTUNS_nodeVC[0], (1000*microscale), &NCTUNS_rem)

#define NCTUNS_ticks_to_sec \
    div_long_long_rem(NCTUNS_nodeVC[0], (1000000*microscale), &NCTUNS_rem)

#define NCTUNS_ticks          NCTUNS_ticks_to_ms
#define NCTUNS_tcp_time_stamp NCTUNS_ticks_to_ms
#define NCTUNS_xtime_tvsec    NCTUNS_ticks_to_sec

```

The `NCTUNS_nodeVC[]` is used to store the virtual time. The simulation engine also maps these variables to the kernel via the memory mapping technique. A Kernel function `div_long_long_rem()` is used to do 64-bit division. In Linux 2.4.22, original kernel function `do_div()` that is defined as a macro in `div64.h` can not correctly do 64-bit division. As such, we add the new macro `div_long_long_rem()` to fix this bug (In fact, the Linux 2.6.x already fixes the known bug.). the code of `div_long_long_rem()` is shown below:

```

/* (long)X = ((long long)divs) / (long)div
   (long)rem = ((long long)divs) % (long)div */
#define div_long_long_rem(a, b, c) div_ll_X_I_rem(a, b, c)

extern inline long
div_ll_X_I_rem(long long divs, long div, long *rem)
{
    long dum2;
    __asm__("divl %2":"=a"(dum2), "=d"(*rem) : "r"(div), "A"(divs));
    return dum2;
}

```

The variable, `microscale`, is used to turn the virtual time to a value based

micro-seconds. When the simulation engine uses system call 261 with flag NSC_TICKTONANO to bring the time scale value (*time_scale*) into the kernel, the *microscale* will be calculated by $1000/time_scale$. Then the unit of $NCTUNS_nodeVC[0]/microscale$ will be micro-second. Therefore, *NCTUNS_ticks_to_us* will return how many micro-seconds the current virtual time means. For the same reason, *NCTUNS_ticks_to_ms* or *NCTUNS_ticks_to_sec* will return how many milli-seconds or seconds the current virtual time means. According to above definitions, we also can know that *NCTUNS_ticks* and *NCTUNS_tcp_time_stamp* are based on milli-second and *NCTUNS_xtime_tvsec* is based on second.

4.3.6 Processing Kernel Timers and Kernel Events

In a simulation, the kernel may generate two kinds of kernel event – the tunnel check event and the kernel timeout event. In section 4.3.2, we already explain why we need to generate a tunnel check event and where we generate this kind of events. In this section, we will clearly explain how a kernel timeout event helps us to trigger a kernel software timer which is based on the virtual time (We call this kind of timer a *virtual-time timer* and the original timer is called a *real-time timer*).

4.3.6.1 Maintain Virtual-time Timers

In the original Linux kernel, all software timers (real-time timers) are maintained in a global structure *tvecs[]*. In order to manage real-time timers and virtual-time timers individually, we create another global structure, *callwheel*, to store virtual-time timers. When the kernel wants to schedule a timer, we should first know whether the

timer is a virtual-time timer or not. If not, the timer should be inserted to `tvecs[]` and triggered by the original kernel triggering mechanism. If yes, the timer should be inserted to the callwheel and triggered by the simulation engine. As such, the original kernel function `internal_add_timer()`, which is used to add a timer structure into `tvecs[]`, should be modified like the following code segment:

```
static inline void internal_add_timer(struct timer_list *timer)
{
    unsigned long expires = timer->expires;
    unsigned long idx = expires - timer_jiffies;
    struct list_head *vec;

    //NCTUNS
    if (check_if_nctuns_timer(timer)) {
        nctuns_callout_helper(timer, 0);
        return;
    }
    //NCTUNS
    ...
    //original procedure...
    ...
}
```

To insert the virtual-time timer into the callwheel



The added kernel function `check_if_nctuns_timer()` is used to check whether a timer is based on the virtual time. If yes, it will return a value that is larger than zero.

Its implementation is shown as below:

```

extern void tcp_write_timer(unsigned long);
extern void tcp_delack_timer(unsigned long);
extern void tcp_keepalive_timer(unsigned long);
extern void it_real_fn(unsigned long);
extern void process_timeout(unsigned long);

int check_if_nctuns_timer(struct timer_list *timer)
{
    void (*fn)(unsigned long);

    fn = timer->function;
    if (fn==tcp_write_timer || fn==tcp_delack_timer || fn==tcp_keepalive_timer) {
        struct sock *sk = (struct sock *)timer->data;
        if (sk->nodeID)
            return sk->nodeID;
    }
    if (fn==it_real_fn || fn==process_timeout) {
        struct task_struct *p = (struct task_struct *)timer->data;
        if (p->p_node)
            return p->p_node;
    }
}

struct timer_list
{
    struct list_head list;
    unsigned long expires;
    unsigned long data;
    void (*function)(unsigned long);
}

```



The *function* field of the timer list structure contains the address of the function to be executed when the timer expires. In `check_if_nctuns_timer()`, we only care about five kinds of timer structure whose *function* field points to the following functions: `tcp_write_timer()`, `tcp_delack_timer()`, `tcp_keepalive_timer()`, `it_real_fn()`, and `process_timeout()`. This is because only these five kinds of kernel timer will be used by the NCTUns 1.0 network simulator. At the same time, if the `sk->nodeID` or `p->p_node` in `check_if_nctuns_timer()` is larger than zero, it means that the timer belongs to a simulated node and should be inserted into the callwheel.

The `nctuns_callout_helper()` is used to insert the virtual-time timer into the callwheel and insert a kernel timeout event into the tunnel interface `t0e0`. The

following shows its pseudo code:

```
#define callwheelsize 8192
#define callwheelmask (callwheelsize-1)

timerbucket callwheel[callwheelsize];

void nctuns_callout_helper(struct timer_list *timer, long us)
{
    1. if t0e0 output queue is full
       fatal error!
    2. else
       2.1 timeout = timer->expires*1000 + us;
       2.2 insert timer to callwheel[timeout & callwheelmask]

       2.3 insert a kernel timeout event into t0e0
           with expire time = timeout
    3. wake up the simulation engine
}
```

Because we set the frequency of timer interrupt to 1000 HZ (once every 1 milli-second), the *jiffies*, a global variable which stores the number of happened timer interrupts, will be based on milli-second. As such, the unit of expires field of a timer structure will be milli-second. In other words, if we want to schedule a software timer, the shortest time interval will be 1 milli-second. But 1 milli-second is too large for us, we want a shorter time interval such as 1 micro-second. Therefore, in `nctuns_callout_helper()`, there is an argument *us* to bring in the micro-second information.

After inserting the kernel timeout event into the tunnel interface `t0e0`, we have to wake up the simulation engine to receive the event. Then, the simulation engine will read the event from `t0e0` and insert the event to its event queue.

4.3.6.2 Kernel Timeout Event Triggering

When the event scheduler (in the simulation engine) processes a kernel timeout

event, it will use a user-defined system call (system call 263) to trigger the corresponding kernel timer. Inside the kernel, we implement a kernel function `nctuns_callback()` to do this job:

```
void nctuns_callback(void)
{
    1. timeout = NCTUNS_ticks_to_us
    2. timer = callwheel[timeout & callwheelmask]
    3. while timer
        if timer->expires == NCTUNS_ticks
            callback timer->function
        timer = timer->next
}
```

In `nctuns_callout_helper()`, the unit of the hash key (*timeout*) is micro-second. Therefore, in `nctuns_callback()`, the hash key (*timeout*) should also be based on micro-second. As in the previous discussion, `NCTUNS_ticks_to_us` will turn the current virtual time into micro-second. However, because the `expires` field of a timer structure is based on milli-second, we need to compare the `timer->expires` with `NCTUNS_ticks` rather than `NCTUNS_ticks_to_us` when in step 3 of `nctuns_callback()`.

4.3.7 Based on the Virtual Time

In this section, we will describe which part of the kernel should be modified to reference the virtual time instead of the real time. There are two main parts that we should modify when they are used in a simulation. They are TCP timers and time-related system calls.

I. TCP Timers

As we know, a TCP socket will use several timers to manage its connection such as re-transmission timer, delay-ack timer, keep alive timer etc. Of course, if the TCP

connection is used for a simulation, its TCP timers should be triggered based on the virtual time rather than the real time. Therefore, when any statement refers to *tcp_time_stamp*, *jiffies*, and *xtime.tv_sec*, they should be respectively changed to refer to *NCTUNS_tcp_time_stamp*, *NCTUNS_ticks*, and *NCTUNS_xtime_tvsec*. Following shows some example code segments:

```
void tcp_keepalive_timer (unsigned long data)
{
    ...
    //NCTUNS
    //elapsed = tcp_time_stamp - tp->rcv_stamp;
    elapsed = (sk->nodelD ? NCTUNS_tcp_time_stamp : tcp_time_stamp) - tp->rcv_stamp;
    //NCTUNS
    ...
}
```

```
void tcp_reset_keepalive_timer (struct sock *sk, unsigned long len)
{
    //NCTUNS
    //if (!mod_timer(&sk->timer, jiffies+len))
    if (!mod_timer(&sk->timer, (sk->nodelD ? NCTUNS_ticks : jiffies) + len))
    //NCTUNS
    sock_hold(sk);
}
```

```
int tcp_v4_connect (struct sock *sk, struct sockaddr *uaddr, int addr_len)
{
    ...
    //NCTUNS
    //if (peer && peer->tcp_ts_stamp + TCP_PASW_MSL >= xtime.tv_sec)
    if (peer && peer->tcp_ts_stamp + TCP_PAWS_MSL >=
        (tp->nodelD ? NCTUNS_xtime_tvsec : xtime.tv_sec) )
    //NCTUNS
    {
        ...
    }
}
```

II. Time-related System Calls and Kernel Functions

gettimeofday()

In the kernel, it is implemented by *do_gettimeofday()*:

```

void getvirtualtime(struct timeval *tv)
{
    long rem = 0;
    tv->tv_sec = NCTUNS_xtime_sec;
    tv->tv_usec = div_long_long_rem (NCTUNS_rem, microscale, &rem);
}

void do_gettimeofday(struct timeval *tv)
{
    unsigned long flags;
    unsigned long usec, sec;

    read_lock_irqsave(&xtime_lock, flags);
//NCTUNS
    if (current->p_node) {
        getvirtualtime(tv);
        read_unlock_irqrestore(&xtime_lock, flags);
    }
//NCTUNS
    ...
}

```

alarm(), ualarm(), select(), set_itimer(), usleep()

These APIs provided by the standard C library are implemented by the system call *set_itimer()*. In the kernel, the kernel function *do_setitimer()* is used to service this system call. Therefore, if the calling process belongs to a simulation, we should change *do_setitimer()* to refer to the virtual time:

```

int do_setitimer(int which, struct itimerval *value, struct itimerval *ovalue)
{
    register unsigned long i, j;
    int k;
//NCTUNS
    struct task_struct *nctuns_proc;
//NCTUNS

    i = tvtojiffies(&value->it_interval);
    j = tvtojiffies(&value->it_value);
    if (ovalue && (k=do_getitimer(which, ovalue) < 0))
        return k;
    switch (which) {
    case ITIMER_REAL:
//NCTUNS
        if (nctuns>0 && nctuns_proc=find_task_by_pid(nctuns) && current->p_node) {
            nctuns_del_timer_sync(&current->real_timer);
            current->it_real_value = j;
            current->it_real_incr = i;

            if (!j) break;
            if (j > (unsigned long)LONG_MAX)
                j = LONG_MAX;
            i = j + NCTUNS_ticks;
            current->real_timer.expires = i;
            nctuns_callout_helper(&current->real_timer, (value->it_value.usec)%1000);
            break;
        }
//NCTUNS
        ...
        break;
    case ITIMER_VIRTUAL:
        ...
    }
}

```

To be based on the
virtual time

In `do_setitimer()`, `value->it_value` indicates how much time away from now that the kernel needs to trigger the timer. `Value->it_interval` indicates the period of the timer (the interval time that is used to periodically to trigger this timer). Then we directly use `nctuns_callout_helper()` to insert the real timer structure of the current process into the *callwheel* rather than using the `internal_add_timer()`. The reason is that we want to bring the micro-second information into `nctuns_callout_helper()`. The handle function of the real timer structure of a process is `it_real_fn()`. It processes periodical interval timers. Therefore it also needs some modifications:


```

void it_real_fn(unsigned long __data)
{
    struct task_struct *p = (struct task_struct *)__data;
    unsigned long interval;
//NCTUNS
    struct task_struct *nctuns_proc;
//NCTUNS
    send_sig(SIGALARM, p, 1);
    interval = p->it_real_incr;

    if (interval) {
        if (interval > (unsigned long) LONG_MAX)
            interval = LONG_MAX;
//NCTUNS
        if (nctuns>0 && nctuns_proc=find_task_by_pid(nctuns) && p->pnode) {
            p->real_timer.expires = NCTUNS_ticks + interval;
            nctuns_callout_helper(&p->real_timer, 0);
        } else {
            p->real_timer.expires = jiffies + interval;
            add_timer(&p->real_timer);
        }
    }
//NCTUNS
}
}

```

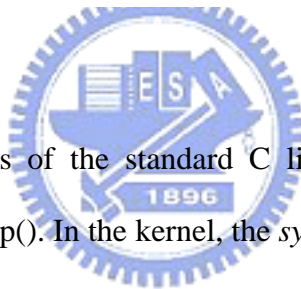
The real timer of this process is a periodical timer.

The process which the kernel will send a signal to belongs to a simulation.

Re-schedule this timer

sleep(), nanosleep()

These two functions of the standard C library are implemented by the system call nanosleep(). In the kernel, the `sys_nanosleep()` is used to service this system call.



```

asmlinkage long sys_nanosleep(struct timespec *rqtp, struct timespec *rmtp)
{
    ...
//NCTUNS
    /* Because we can control the virtual time precisely */
    if (nctuns>0 && find_task_by_pid(nctuns) && current->p_node)
        goto skip_udelay;
//NCTUNS
    if (t.tv_sec==0 && t.tv_nsec <= 2000000L &&
        current->policy != SCHED_NORMAL) {
        udelay((t.tv_nsec + 999)/1000);
        return 0;
    }
//NCTUNS
skip_udelay:
    if (nctuns>0 && find_task_by_pid(nctuns) && current->p_node) {
        expire = 1000*t.tv_sec + t.tv_nsec/1000000;
        current->stat = TASK_INTERRUPTIBLE;
        expire = schedule_timeout_us(expire, t.tv_nsec/1000);
    } else {
        expire = timespec_to_jiffies(&t) + (t.tv_sec || t.tv_nsec);
        current->stat = TASK_INTERRUPTIBLE;
        expire = schedule_timeout(expire);
    }
//NCTUNS
    ...
}

```



Schedule_timeout() is a kernel function used to make the current process sleep until a specified value have elapsed. *Schedule_timeout_us()* is a kernel function added by us. Actually, *schedule_timeout_us()* is modified from *schedule_timeout()*. Their difference is that the *schedule_timeout_us()* can use a argument to bring in the micro-second information.

```

signed long schedule_timeout_us (signed long timeout, long us)
{
    struct timer_list timer;
    ...
    expire = timeout + NCTUNS_ticks;

    init_timer(&timer);
    timer.expires = expire;
    timer.data = (unsigned long)current;
    timer.function = process_timeout;

    nctuns_callout_helper(&timer, (us%1000));
    schedule();
    nctuns_del_timer_sync(&timer);

    timeout = timer.expires - NCTUNS_ticks;
out:
    return timeout < 0 ? 0 : timeout;
}

```

4.3.8 NCTUns Divert Socket

In Linux 2.4.x, the kernel does not support the divert socket. However, some daemons such as emulation daemons and Mobile-IP daemons need a similar mechanism like the divert socket. In part II of this paper, we will describe why these daemons must use this mechanism. In this section, we will describe how we bring FreeBSD's divert socket into Linux.

First, if a user-level program wants to use our divert socket (Here, we call it NCTUns divert socket), it should include a header file, *nctuns_divert.h* :

```

#ifndef __nctuns_redirect_h__
#define __nctuns_redirect_h__

/* IP Hooks */
#define NF_IP_PRE_ROUTING 0 // After promisc drops, checksum checks.
#define NF_IP_LOCAL_IN 1 // If the packet is destined for this box.
#define NF_IP_FORWARD 2 // If the packet is destined for another interface.
#define NF_IP_LOCAL_OUT 3 // Packets coming from a local process
#define NF_IP_POST_ROUTING 4 // Packets about to hit the wire.
#define NF_IP_NUMHOOKS 5

/* Divert Operations */
#define DIVERT_INFO 0x01
#define DIVERT_FLUSH 0x02
#define DIVERT_ADDHEAD 0x03
#define DIVERT_ADDTAIL 0x04
#define DIVERT_DELETE 0x05
#define DIVERT_MOVE_TAIL 0x06

struct divert_entry {
    int proto; // protocol
    u_long srcip; // source ip
    u_long smask; // netmask for source ip
    u_long dstip; // destination ip
    u_long dmask; // netmask for destination ip
    u_long sport; // source port number
    u_long dport; // destination port number
}

#ifdef __KERNEL__

#define MAX_DIVERT_ADDR 128

struct divert_rule {
    struct list_head nextdr;
    int fd;

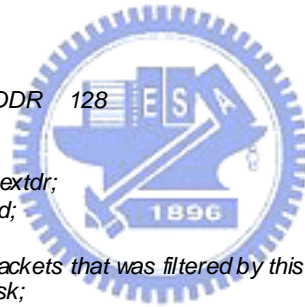
    /* The socket where packets that was filtered by this rule will be sent to */
    struct sock *sk;

    u_long hook;

    int proto;
    u_long srcip;
    u_long smask;
    u_long dstip;
    u_long dmask;
    u_long sport;
    u_long dport;
}

#endif /* __KERNEL__ */
#endif /* __nctuns_redirect_h__ */

```



The *divert_entry* structure is used to store the filter rule. Then the user-level program can use a user-defined system call (system call 259) to register a filter rule that is stored in a *divert_entry* structure into a specified hook number list. At system call 259, there is an important argument that must be set: a RAW socket file descriptor. This is used to specify which socket to receive filtered packets. As such, when a

packet is filtered by a filter rule, we can know which socket we will send the packet to. Finally, the user-level program can issue the *recvfrom()* system call to receive the filtered packets from the RAW socket that is created by itself. The following shows an example:

```
#include "nctuns_divert.h"

int main()
{
    struct divert_entry de;
    int rawfd;
    struct sockaddr_in addr;
    char buf[8192];

    rawfd = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
    if (setsockopt(rawfd, IPPROTO_IP, IP_HDRINCL, (char *)&sm, sizeof(sm)) < 0)
        perror("setsockopt");

    de.proto = IPPROTO_IP;
    de.srcip = inet_addr("140.113.1.1");
    de.smask = inet_addr("255.255.255.0");
    de.dstip = inet_addr("140.113.17.5");
    de.dmask = inet_addr("255.255.255.0");
    de.sport = 0; // 0 means don't care
    de.dport = 80;

    syscall(259, DIVERT_ADDHEAD, rawfd, NF_IP_LOCAL_OUT, (char *)&de, sizeof(de));
    while(1) {
        recvfrom(rawfd, buf, sizeof(buf), 0, (struct sockaddr *)&addr, sizeof(addr));
        ...
    }
}
```

In the kernel, we implement several kernel functions to service this work: *nctuns_reg_divert_rule()*, *divert_hook()*, and *nctuns_pkt_match()*. *Nctuns_reg_divert_rule()* is the main function used to maintain all filter rules such as adding a rule, moving a rule, deleting a rule, flushing all rules, displaying filter rules etc. It maintains a rule table that has five rule lists; every list represents a hook number.

Nctuns_pkt_match() is used to check whether a packet is matched by a filter rule or not. *Divert_hook()* will be registered to the *netfilter* [5] module as a call-back function. It will be called for every packet that traverses the respective hook within

the network stack. When `divert_hook()` is called with a packet, it will call `nctuns_pkt_match()` to check whether we should capture this packet. If yes, `divert_hook()` will put it into the corresponding RAW socket's receive queue.

Furthermore, when the user-level program uses `recvfrom()` to receive a filtered packet, we want to provide the assigned IP address of the interface where the packet came from. According to the above example code, if the program receives a packet from `recvfrom()`, the structure `addr` will be filled with the IP address of the packet's incoming interface. If the packet is an outgoing packet, `addr` will be filled with the packet's source IP address. This enables the program to give packets different treatment according to the incoming interface, especially for multi-interface devices. As such, some original RAW socket implementation needs a little change:



```

int raw_rcvmsg(struct sock *sk, struct msghdr *msg, int len,
              int noblock, int flags, int *addr_len)
{
    ...
    struct sockaddr_in *sin = (struct sockaddr_in *)msg->msg_name;
    ...
    if (sin) {
        sin->sin_family = AF_INET;
        sin->sin_addr.s_addr = skb->nh.iph->saddr;
//NCTUNS
        /* Return IP address that is bound in the input device */
        if (sk->nodeID && (skb->pkt_type==PACKET_DIVERT)) {
            struct in_device *in_dev;
            struct in_ifaddr *ifa;
            if (skb->dev)
                in_dev = in_dev_get(skb->dev);
            else
                goto next;
            if (in_dev==NULL || (ifa=in_dev->ifa_list)==NULL) {
                printk("raw_rcvmsg(): Can't find interface IP!\n");
                in_dev_put(in_dev);
                goto next;
            }
            ifa = in_dev->ifa_list;
            if (ifa)
                sin->sin_addr.s_addr = ifa->ifa_local;
            in_dev_put(in_dev);
            skb->dev = NULL;
        } else if (sk->nodeID > 0) {
            u_char *p1;
            p1 = (u_char *)&(sin->sin_addr.s_addr);
            p1[0] = 1;
            p1[1] = 0;
        }
        next:
//NCTUNS
        ...
    }
    ...
}

```

The skb is queued to the RAW socket by divert_hook().

The RAW socket is not a divert socket.

If the RAW socket is not turned into a divert socket, we should recover the IP address from the S.S.D.D format to the normal 1.0.X.X format. For example, the “ping” program uses a RAW socket to send and receive ICMP packets. When it receives a ICMP reply packet via recvfrom(), we should recover the IP address; otherwise the ping program will get an IP address with S.S.D.D format.

5. Evaluation

In this section, we will use some simulation cases to verify the accuracy of the Linux version of the NCTUns 1.0 network simulator. We also will run several simulation cases to test the scalability of our simulator. For these cases, we will discuss their results respectively.

5.1 Simulation and Experiment Result Comparison

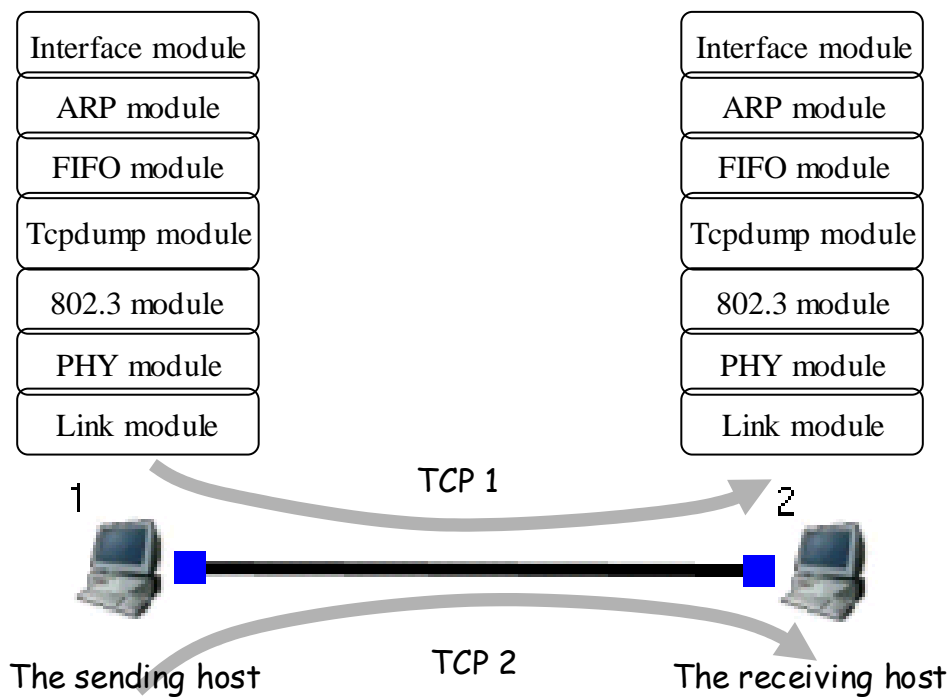


Figure I-5.1.1: The testing network topology

I. Experiment Case Setup

Figure I-5.1.1 depicts a simple network topology. Both the experiment case and the simulation case follow figure I-5.1.1 to set up their network topologies. As the figure shows, there are two hosts connected to each other by a link. The bandwidth of the link is 100Mbps (Fast Ethernet). We use two IBM A30 notebook computers to

help us complete this experiment. An IBM A30 is equipped with a 1.6GHz Pentium processor, 256MB RAM, and a Fast Ethernet network interface (100Mbps). Both of those two A30s are installed with the Fedora Core Release 1 package [16] whose Linux kernel version is 2.4.22.

By default, the Linux kernel will create a per-interface FIFO queue to hold outgoing packets for each network interface, and the default maximum queue length is 100 packets. We do not change this value. During an experiment, we will set up 2 greedy TCP connections from the sending host to the receiving host. The two TCP connections will contend for the sending host's output FIFO queue. In other words, the sending host's output queue will be the bottleneck.

II. Simulation Case Setup



In the simulation case, the organization of each node is shown in the upper side of figure I-5.1.1. In the FIFO (First-In-First-Out) module, we set the maximum queue length to 110 packets rather than 100. This is because there is an unknown packet buffer in the hardware of the Fast Ethernet NIC that will be used by the experiment case. The buffer size will influence the simulation result. In order to solve this problem, we purposely increase the maximum queue length of the FIFO module. The simulation machine is also an IBM A30 notebook. The Linux kernel version modified to support the NCTUns 1.0 network simulator is 2.4.22, which is the same with the experiment case.

III. Result Comparison

Figure I-5.1.2 shows the total throughput comparison of the two cases. From figure I-5.1.2, we can obviously discover that the total throughput of the experiment

case is larger than the throughput of the simulation case. The average difference is about 376 Kbytes/sec. The difference is caused by our MAC 802.3 module and the MAC 802.3 implementation of the A30's Ethernet NIC. In our MAC 802.3 module, we add a little random delay between two successive packets that are pushed onto the link. The average ratio is about 5% of the transmission delay of the outgoing packet. As we know, the random delay time between two successive packets will influence the total throughput significantly. If there is a smaller average delay ratio, the total throughput will be larger. Therefore, we think that the difference in figure I-5.1.2 is due to the difference between the two implementation of the MAC 802.3 protocol.

Figure I-5.1.3 shows the experimental result of two contending TCP connections. Figure I-5.1.4 shows the corresponding simulation result. By comparing these two figures, we can see that their competitive behaviors are almost the same. This result verifies the accuracy of the NCTUns network simulator.

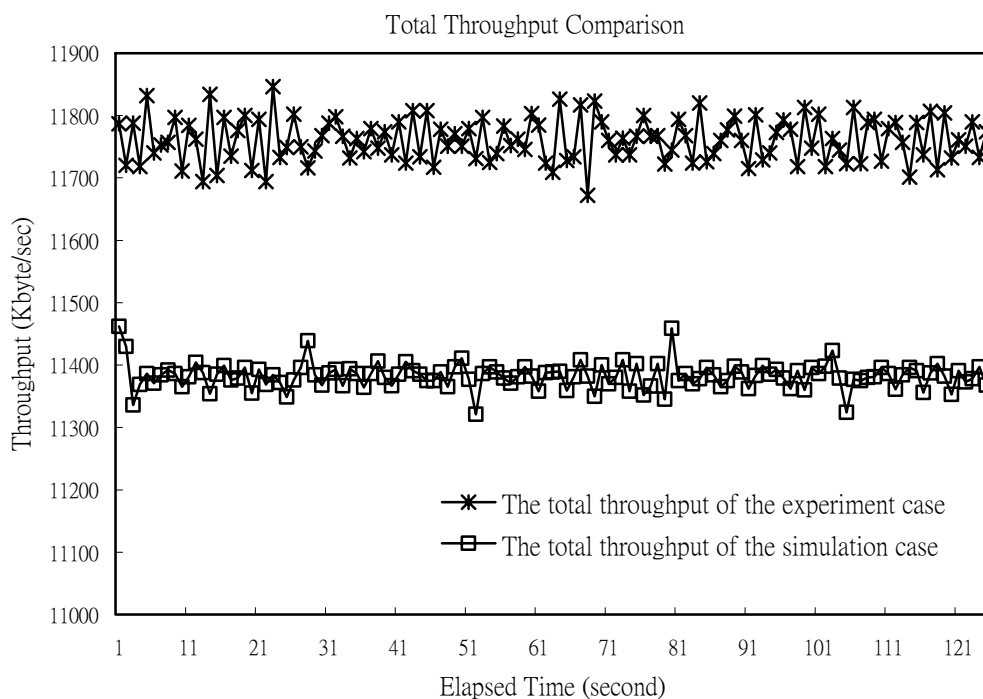


Figure I-5.1.2: The total throughput comparison between the experiment case and the simulation case

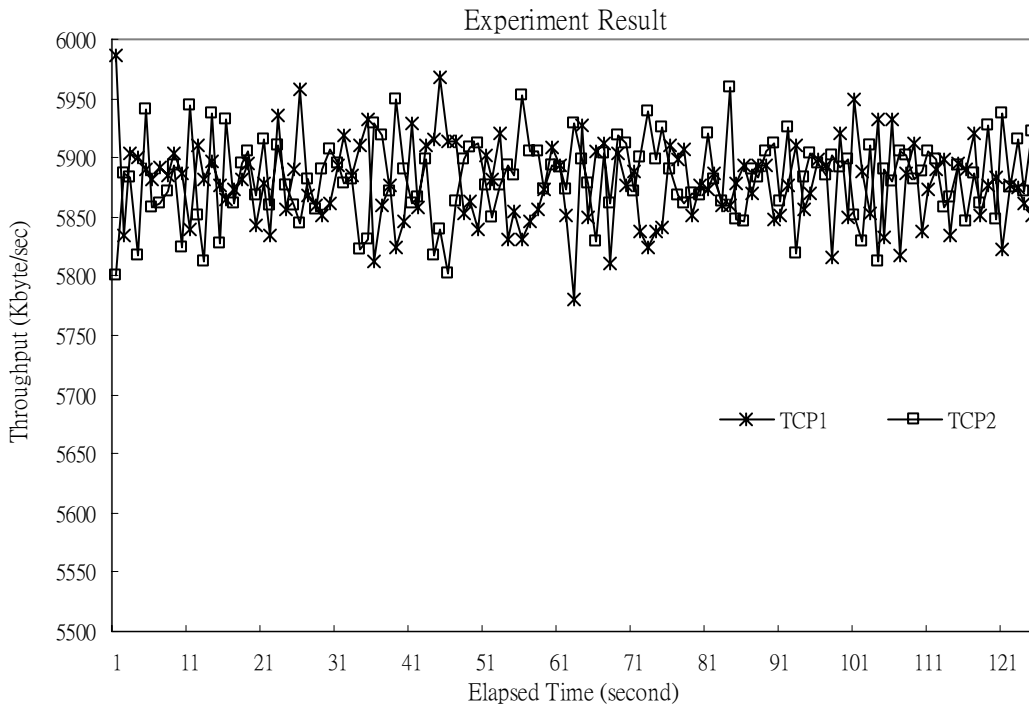


Figure I-5.1.3: The experiment result of two contending TCP connections

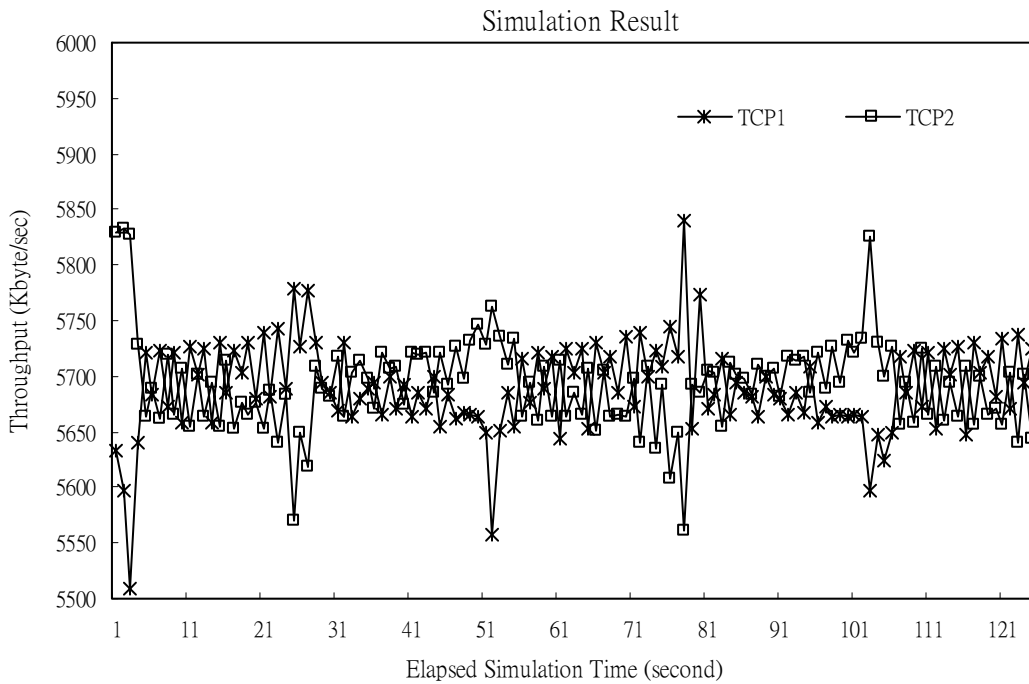


Figure I-5.1.4: The simulation result of two contending TCP connections

5.2 Simulation Speed

I. Simulation Setup

In the simulation case, two hosts are connected with a link whose bandwidth is 100 Mbits/sec. The link delay is set to 1 milli-second. The traffic is a one-way constant-bit-rate (CBR) UDP packet stream. Each UDP packet size is set to 1400 bytes. The total simulation time is set to 500 seconds and the used machine is an IBM A30 notebook.

We vary the packet interval time between two successive packet transmissions. The tested intervals are 1, 0.5, 0.1, 0.05, 0.01, 0.005, 0.001, 0.0005, and greedy transmission. The greedy transmission means that the sending host will try its best to push out its UDP packets.



II. Result

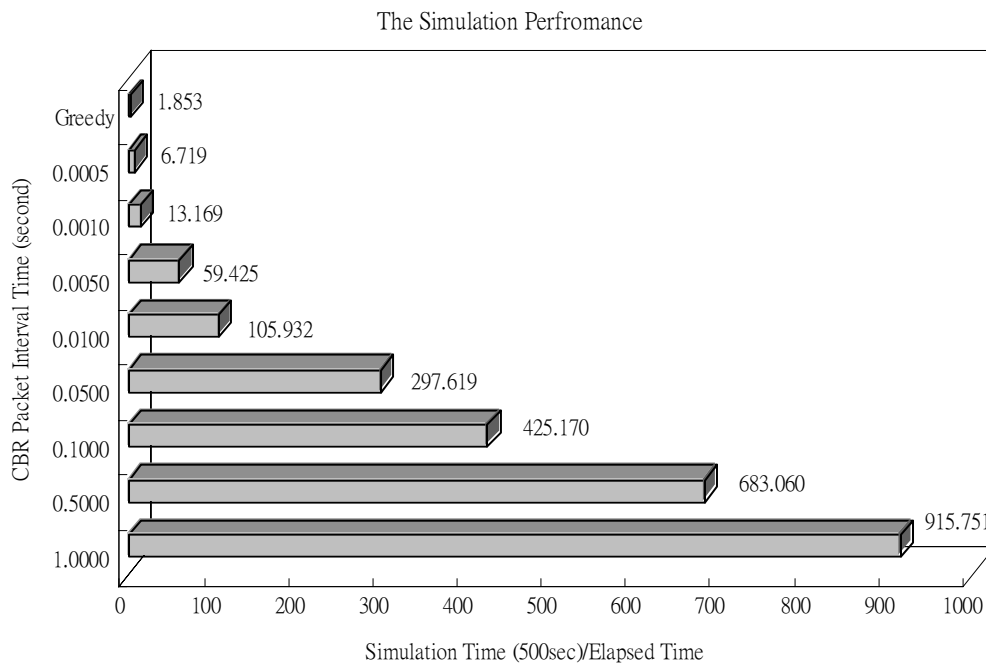
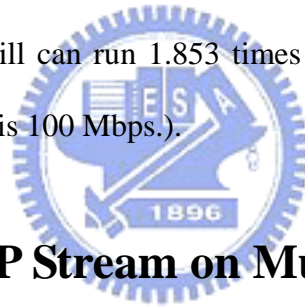


Figure I-5.2.1: The simulation performance under various constant-bit-rate UDP traffic loads. (A higher ratio means a better performance)

The performance metric is the ratio of the simulated seconds to the elapsed real seconds for running the simulation. A higher ratio means a higher performance. Figure I-5.2.1 shows the simulation performance plot. We can see that the simulation can be very quickly finished at a lower traffic load. When the interval time is 1 second, the ratio is 915.751. In other words, our simulator just spends 0.546 (500/915.751) seconds in the real time to finish the simulation. This result verifies that the discrete event simulation methodology can significantly speed up the simulation speed very much. Even at the very high traffic load such as 0.0005 seconds and the greedy transmission condition, the ratio is still greater than 1. In other words, the simulation speed is still faster than the real-world time. For the greedy transmission, the simulator still can run 1.853 times faster than the real-world time (Note that the link bandwidth is 100 Mbps.).



5.3 Fixed CBR UDP Stream on Multiple Hop Networks Case

I. Simulation Setup

We also use an IBM A30 notebook computer to run these simulation cases. In the suite, we want to see the performance about a CBR UDP packet stream to be passing through a different number of forwarding nodes. We will test two kinds of forwarding nodes -- switch and router. Figure I-5.3.1 shows the network configuration of this simulation case.

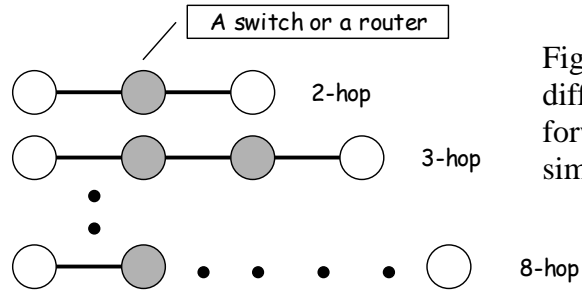


Figure I-5.3.1: The different number of forwarding nodes in each simulation case

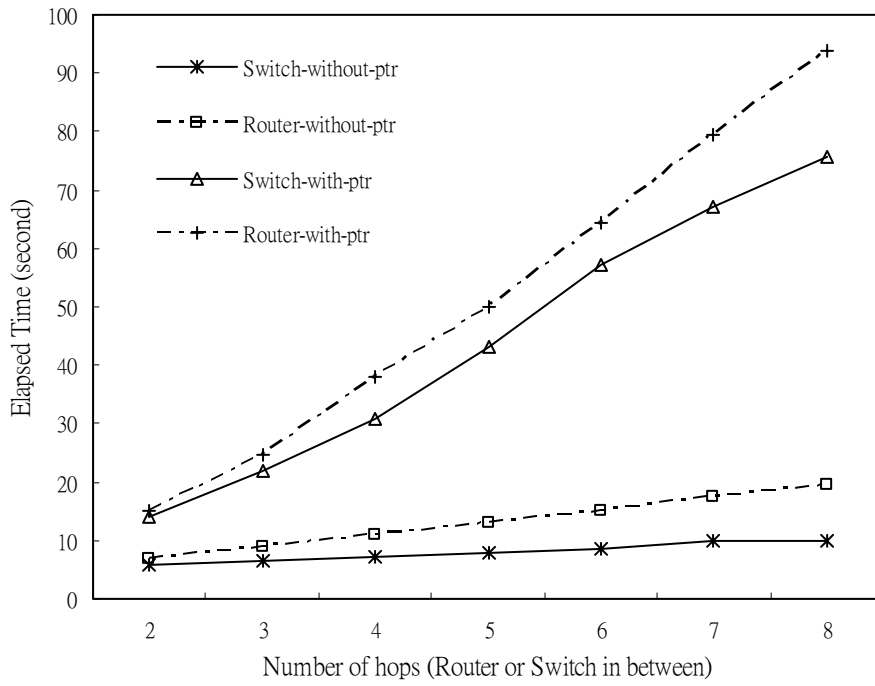


Figure I-5.3.2: The performance under different forwarding nodes (switch or router)

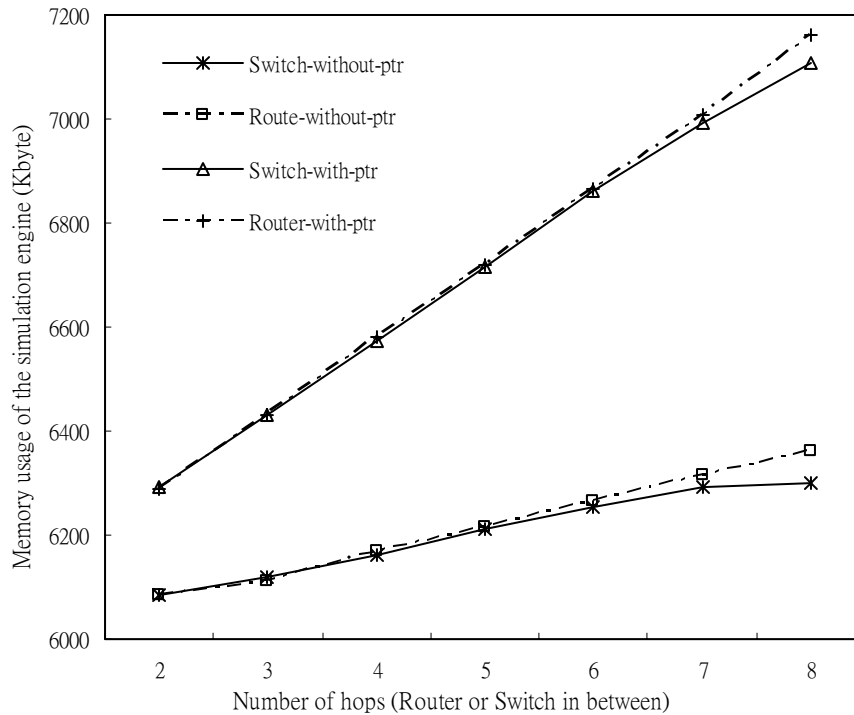


Figure I-5.3.3: The memory usage under different forwarding nodes (switch or router)



The bandwidth and delay of all links are set to 10 Mbps and 10 milli-seconds respectively. The interval time of the CBR UDP packet stream is set to 0.002 seconds and the packet length of each UDP packet is set to 1400 bytes. The total simulation time is set to 100 seconds. In these simulation cases, we care about two performance metrics -- the elapsed real-world time and the memory usage of the simulation engine. In addition, we will observe how the logging mechanism of packet trace influences the performance.

II. Report

Figure I-5.3.2 shows the performance plot about the elapsed real-world time. “Switch-without-ptr” means that the forwarding nodes are switches, and the simulator does not turn on the logging mechanism of packet trace. “Switch-with-ptr” means that

the simulator turns on the logging mechanism of packet trace. By comparing the switch-without-ptr case and the router-without-ptr case, we can see that the performance of the router-without-ptr case always takes more time than the switch-without-ptr case. This result is reasonable. When simulating a packet to be forwarded by a router, the simulator should write the packet into kernel and then read it from the kernel. However, when simulating a packet to be forwarded by a switch, all forwarding operation is simulated in the simulation engine. The overhead of issuing the read and write system calls is very high. Therefore, the router-without-ptr case and the router-with-ptr case will spend more time than the switch ones, respectively.

In figure I-5.3.2, we also can discover that the router-with-ptr case takes much more time than the router-without-ptr case. The switch-with-ptr case and the switch-without-ptr case also have the same result. Obviously, this result is caused by the logging mechanism of packet trace. Therefore, we can know that the simulator spends a lot of time logging the trace of each packet. This influences the performance very much. In figure I-5.3.3, we can also see that the simulator will consume much more memory when turning on the logging mechanism. This is because the simulator should maintain a lot of structures to record the trace of every packet. As such, if we want to further speed up the simulation speed in the future, the logging mechanism will be the most important part.

5.4 MANET in the NCTUns network simulator

I Simulation Setup

In this test suite, the network topology is a two dimensional array in which each

element is a ad-hoc mode mobile node. Figure I-5.4.1 shows this topology. We will vary the dimensions of the array to see how the simulator's speed and memory usage will change when there are more packets exchanged in the simulation network.

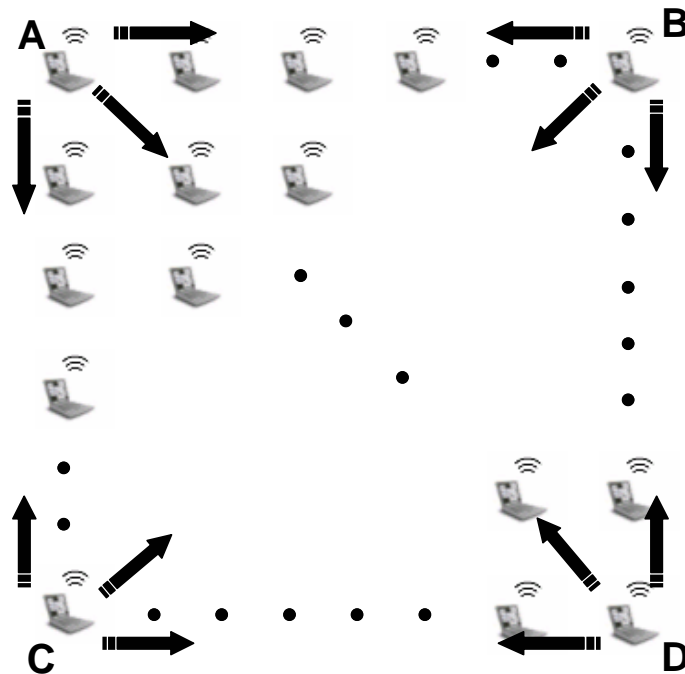


Figure I-5.4.1: The network topology that is organized by the ad-hoc mode mobile nodes.

The tested dimensions are 2 by 2, 4 by 4, 6 by 6, 8 by 8, and 10 by 10. In other words, the total numbers of mobile nodes are 4, 16, 36, 64, and 100, respectively. For all mobile nodes, their bandwidths are set to 11 Mbps and the distance between two mobile nodes is 150 meters. For the traffic settings, four corner mobile nodes will use three CBR UDP packet stream to send packets to other 3 mobile nodes. For example, in figure I-5.4.1, the destination of three CBR UDP packet stream that are generated by the node A will be the node B, C, and D. As such, for different dimensional cases, there are 12 CBR UDP packet streams carried on to the simulation network. The packet interval time is 0.01 seconds and each UDP packet size is 1400 bytes. The total simulation time is set to 100 seconds. The used machine is an IBM A30 notebook computer.

II. Report

In figure I-5.4.2, we can see the performance of the MANET (Mobile Adhoc NETWORK) in the NCTUns network simulator. When the simulator simulates a MANET under a high traffic load, the performance is much worse than the wired simulation cases. As the network size grows, the performance decreases more quickly. This is because the simulation network concurrently has much more packets in it although the rate of injecting packets into the simulation network is the same. By comparing figure I-5.4.2 and figure I-5.3.2, we can discover that the logging mechanism has a smaller effect on the MANET case. This is because the simulator spends a lot of time processing the 802.11 protocol so that the portion of the logging processing is not as high as in figure I-5.3.2. The ad-hoc mode mobile network is also an important case that needs to be improved to increase the performance of the NCTUns network simulator in the future.

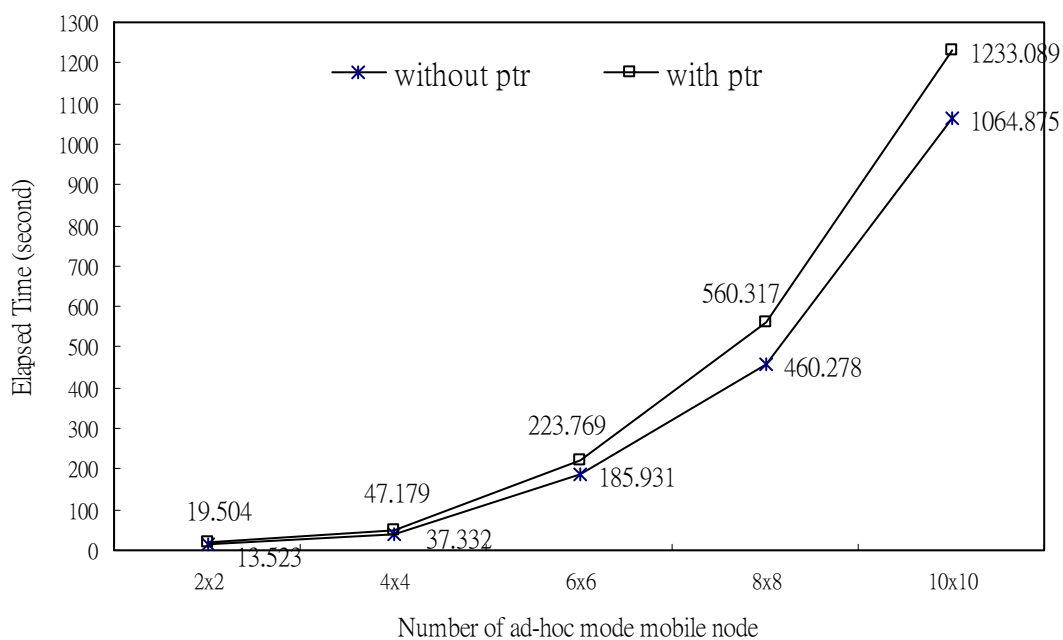


Figure I-5.4.2: The performance under different network sizes (different dimensions)

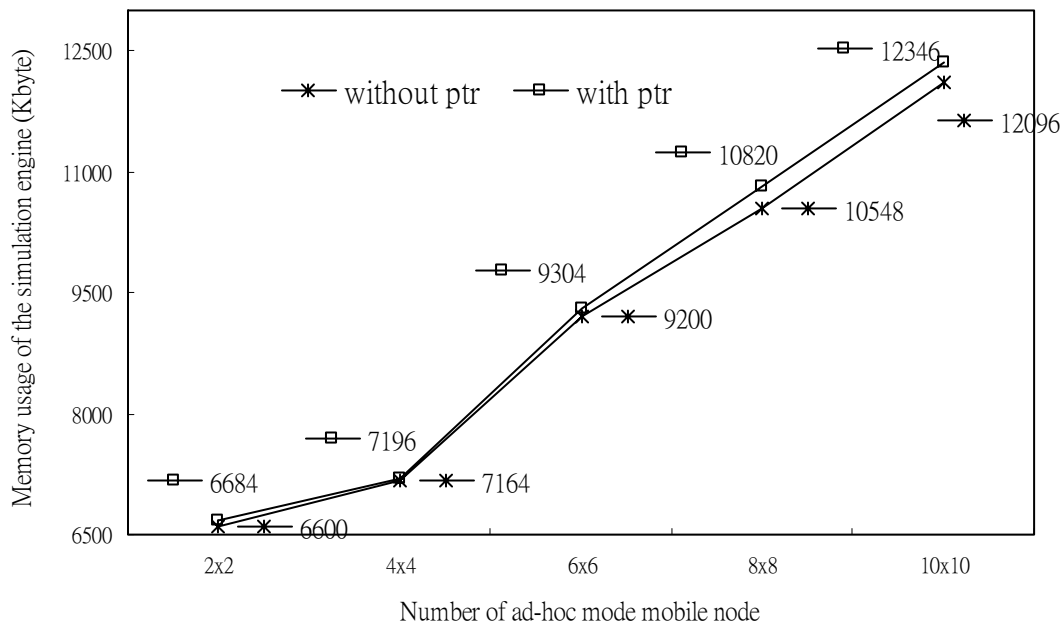


Figure I-5.4.3: The memory usage under different network sizes (different dimensions)

6. Future Work

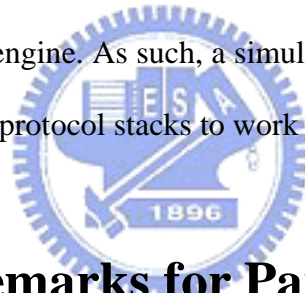


After many years of development, the NCTUns 1.0 network simulator has become a high-fidelity and extensible network simulator. However we are never satisfied with its current status. We will continually improve it and develop more functions for it. Here, we list some possible future work. One is to support larger network topologies. In the current version, it can not create a simulation network that has more than 254 network subnets or more than 4095 tunnel interfaces. This is the limit of the S.S.D.D IP scheme and size of the UNIX device number.

Another one is the performance of the simulation. Although applying the discrete time event simulation methodology to the NCTUns 1.0 can already speed up simulations, we think there are still several approaches that can help us speed up the simulation speed. These include implementing the simulation engine as a kernel

module, parallel simulation, etc.

Yet another is to overcome the drawback that all layer-3 devices must share the same TCP/IP protocol settings in the kernel. Due to the kernel re-entering simulation methodology, all layer-3 devices use the same TCP/IP protocol stack and will refer to the same settings. If a node wants to use different settings from those used by other nodes, the NCTUns 1.0 can not support this case. For example, a simulated node 1 wants to set its TCP keep-alive timer as 7000 but the simulated node 2 want to use a value of 7500. In the current design, it is impossible to achieve this goal. An approach to overcome this problem is to modify the kernel to support per-node TCP/IP protocol settings. Another approach is to port the FreeBSD's or Linux's TCP/IP protocol stack as modules of the simulation engine. As such, a simulation network may concurrently have several different TCP/IP protocol stacks to work together.



7. Concluding Remarks for Part I

In part I, we clearly describe the architecture of the NCTUns 1.0 network simulator. We also present the detail about how we port the NCTUns network simulator to the Linux platform. Based on the kernel re-entering simulation methodology and the discrete event simulation methodology, the NCTUns 1.0 provides many useful functions and good performance. After this tool is released to the network community, many people or organizations have visited its Web site and download this tool. Many users also provide various suggestions. Among these suggestions, porting NCTUns from FreeBSD to Linux is the most desired one. Therefore, we decide to port the NCTUns to the Linux platform. In the future, we will continue to improve it and develop new functions and protocol modules for it.

Part II: Supporting Emulation

1. Introduction

Emulation is a kind of ability that can allow a network simulator to interact with real network devices. An emulator allows real-world devices to interact with a simulation network and forces real-world packets to experience user-specified network conditions. In this application, we can call a network simulator a network emulator. The network emulator is a very useful tool for testing the functions and performance of a real-world machine because we can see how it will perform under various network conditions.



Today, customers have high expectations from their vendors. They demand products to be reliable, complete and fully tested. However, in modern market competition, time-to-market is a very important factor to achieve more customers. Therefore, developers would like to shorten their development cycle and limit their testing time in order to release their products earlier than other companies. Therefore, a tool having extensive testing capabilities would help developers very much. The NCTUns emulator is qualified to do the job. It is a useful and powerful tool for IP technology developers and vendors to develop and test their products. Basically, an emulator can create a configurable and repeatable network environment. In the environment, developers or testing engineers can configure and monitor the whole testing simulation network. We can configure various network conditions and

parameters in an emulator including packet dropping, delay, queuing, reordering, jitter, limited bandwidth, fragment, routing, etc. By using this kind of tool, developers can easily and quickly build a simulation network that can interact with real-world device or any IP product. Therefore, they can thoroughly test their products in limited time and reduce the time-to-market delay.

Network emulation with NCTUns is based on NCTUns network simulator [1, 2]. A real-world network can interact with a simulation network that is built by NCTUns network simulator. The simulation network and real-world network can become a mixed simulation network. In the mixed simulation network, real-world traffic and simulated traffic can communicate with each other. For example, a TCP connection can be set up between a real-world host and a host that is simulated by the NCTUns network simulator. Also, a TCP connection can be set up between two real-world hosts with their packets traversing a simulated network.

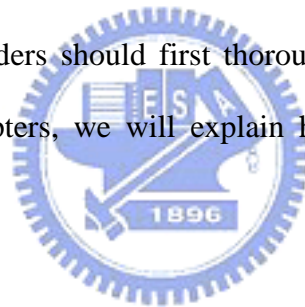


In our design, we can allow a simulation network to exchange packets with external hosts or external routers. In an emulation, we call a real-world host (or device that has one IP interface) an external host. In the meantime, we call a real-world router (or device that has multiple IP interfaces) an external router. Packets can be sent from an external host and then be directed into a simulation network. Furthermore, packets generated in a simulation network can also be directed to an external router, experience the router's packet scheduling and processing, and then return back to the simulation network.

We implement a user-level daemon to achieve these purposes. The user-level daemon is used to correctly guide packets between real-world devices and the

simulation network. Due to the kernel re-entering simulation methodology, the NCTUns 1.0 applies several un-natural schemes in the kernel such as the S.S.D.D IP format, port number translation, etc. Of course, these schemes are never used by normal real-world hosts or devices. As such, if we want real-world packets to traverse between the real-world network and the simulation network, we should clearly know the detail about the internal design of the NCTUns 1.0. With this knowledge, we can design and implement the emulation daemon in an easy way.

In part I of this paper, we already clearly describe and explain the design and implementation of the NCTUns 1.0. In part II, we will extensively refer to these concepts that are introduced in part I. Before readers read this part of the paper, we strongly recommend that readers should first thoroughly understand the content of part I. In the following chapters, we will explain how the NCTUns 1.0 supports emulation in details.



2. Design Goals

First, we want to use a simple way to support emulation under NCTUns network simulator. We hope that we don't need to modify the original architecture too much to easily achieve our goals. At the same time, we also want our emulator to have numerous functionalities that most related work have and even have more functionalities than they have. Second, we hope that the emulation can be easily implemented on all UNIX-like platforms (especially FreeBSD and Linux) because now the NCTUns network simulator can run on both FreeBSD 4.x and Linux 2.4.x platforms. In addition, we also want to find a good solution that can easily suit most UNIX-like systems. To sum up, we decide to use a user-level daemon to do this job.

This is because a user-level daemon has all advantages described above.

By using a user-level daemon, all features and functions that the NCTUns network simulator has can be used by the emulator. We can use all modules provided by the NCTUns including WAN (wild-area-network), DiffServ modules, optical network modules, packet scheduling modules, mobile node routing modules, 802.3 related modules, 802.11 related modules, etc. Most importantly, we want to provide a function that almost all existing work can not do this function is to set up a TCP/UDP connection between a simulated node and a real-world host. This is a very useful feature to test and evaluate any IP product with emulation. We can see how the product would perform under various network conditions without getting, knowing, or modifying its internal protocol stack. In the meantime, developers can test and verify whether their product's protocol behavior is correct or not. Most related work only provide an environment that let packets experience various simulated network conditions and then direct packets back to the real-world network. Both source and destination host must be in the real-world network and they just exchange their packets via the simulation network. In this situation, it is hard to verify their product's correctness unless that using a fully tested device to communicate with their products. However, if they use the NCTUns emulator, they can easily test all functionalities and correctness of a product. This capability saves their time and money.

3. Features

In this section, we will introduce and describe our emulator's features.

3.1 Based On IP Protocol

Our emulator obeys IP protocol. Any application and network protocol that are based on IP protocol can run on our emulation environment. For example, TCP, UDP, or ICMP protocol can easily work well. Therefore, any applications that are based on these protocols can be used in an emulation. For example, FTP, HTTP, SMTP, SSH, TELNET, RTP/RTCP, VoIP (Voice over IP), and IP tunneling can directly work in the emulation environment.

3.2 Interact With Real Hosts

We can physically connect the external host (or real-world device) and the network emulator together via a real-world network. Normally, we can use 100 Mbps Fast Ethernet network to connect them together. After performing some appropriate settings, packets generated by the real-world device can be directed into the simulation network built by NCTUns 1.0. Packets generated by simulated hosts can also be injected into a real-world network. That is, packets can be exchanged between real-world hosts and simulated hosts. Furthermore, packets can be exchanged between two real-world hosts. In this scenario, packets from real-world hosts will be directed to the simulation machine. After they traverse the simulation network, they will be injected to the real-world network and then arrive at another real-world host.

3.3 Interact With Real Routers

Our emulator can also co-operate with real-world routers. Packets generated by simulated hosts can be directed to the real router and then the real router sends reply packets back to the simulation machine. The emulator will then send these packets to their destination nodes in the simulation. This is a useful function for testing a real router or a multi-interface device.

3.4 Can Establish TCP/UDP Connections between the Emulator and Real Hosts

The feature that connections can be set up between real-world hosts and simulated hosts is a remarkable facility compared with other existing related work. The most related work acts like a router allowing real-world traffic to be passed through the simulation network. They can only generate different network conditions and impairments in the emulator but can not be used as an end-point to generate TCP/UDP traffic. Ns-2 [3] claims that it can also support this kind of function (ns-2 calls it “Protocol Mode”). However, until now, this function is still incomplete. Just like other related work, ns-2 only implements the “Opaque Mode”, in which the emulator acts like a router (We will further discuss ns-2 in section 4 of part I).

Figure II-3.4(a) illustrates how the NCTUns emulator can be used as a TCP/UDP end-point. In addition, the TCP/UDP connection can also be set up between two real-world hosts with their packets traversing a simulated network. Figure II-3.4(b) depicts this concept. Most related work can only support the function that figure II-3.4(b) shows.

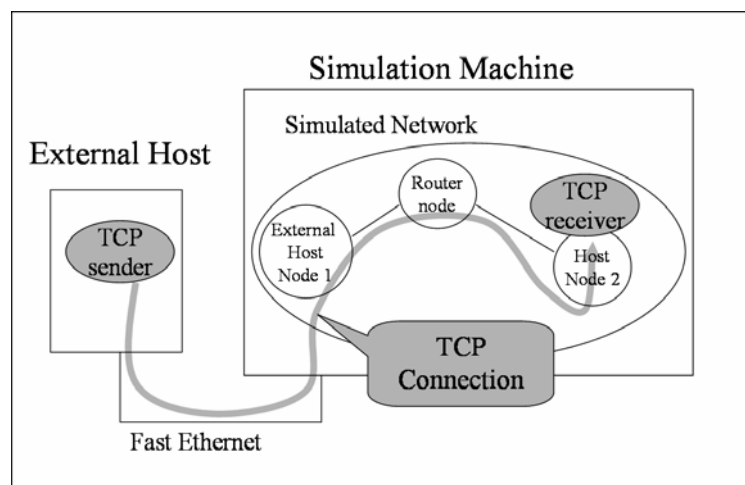


Figure II-3.4(a):
A TCP connection can be set up between the simulated node and the external host.

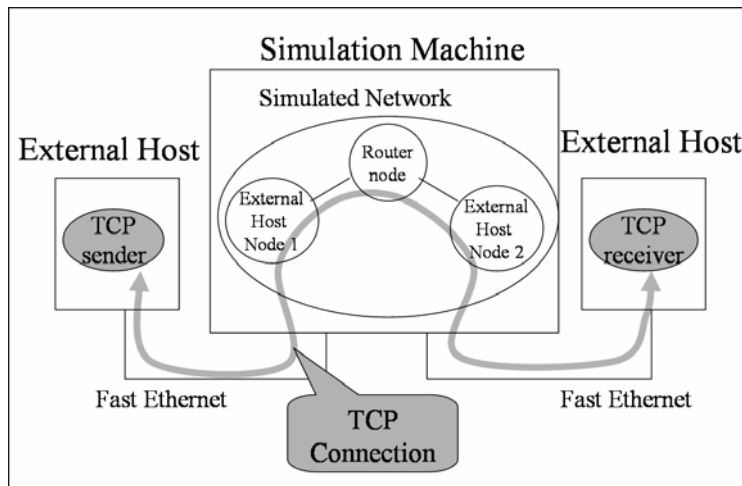


Figure II-3.4(b):
A TCP connection can be set up between two external hosts.

3.5 An External Host Can Be an Ad-hoc/Infra-structure Mode Mobile Node in Emulator

An external host in the real-world can be an external ad-hoc/infra-structure mode mobile node in the simulation network. The real-world device does not need to be a mobile device (e.g., a notebook equipped with an IEEE 802.11 interface). Actually, it can be a fixed host which uses a normal Ethernet link to connect to the simulation machine. Because the external mobile node's IEEE 802.11 MAC protocol is simulated in the virtual network, the IEEE 802.11 MAC protocol is not used between the external host and the simulation machine to exchange their packets. In addition, the mobility is simulated by the emulator rather than by the user physically moving the real-world device around the simulation machine. As such, packets generated by real-world host can experience wireless network conditions such as mobility via an emulation.

3.6 Can Use All Features and Capabilities of NCTUns Network Simulator

The NCTUns network simulator is a high fidelity and extensible network

simulator. It has many unique advantages that cannot be achieved by traditional network simulators [1]. When the simulator is turned into an emulator, all of the capabilities that NCTUns has can still be used by an emulation. Having these advantages, the NCTUns network emulator is very powerful and can provide many capabilities that other emulators can not support. In the following, we will list some capabilities and features of the NCTUns network simulator.

3.6.1 Support for Various Networks

It can simulate wired networks with fixed nodes and point-to-point links. It can also simulate wireless networks with mobile nodes and IEEE 802.11 (b) wireless network interfaces. For IEEE 802.11 (b), both the ad-hoc and infrastructure modes are supported.



3.6.2 Support for Various Networking Devices

It can simulate various networking devices such as Ethernet hubs, switches, routers, hosts, IEEE 802.11 wireless access points and interfaces, etc. A more realistic 802.11 (b) wireless physical module that considers the used modulation scheme, the received power level, the noise power level, and the derived BER is provided

3.6.3 Support for Various Network Protocols

Because of the module-based platform, users can easily develop and add new protocols on the NCTUns simulator. Now, it can simulate numerous protocols such as IEEE 802.3 CSMA/CD MAC, IEEE 802.11 (b) CSMA/CA MAC, the learning bridge protocol used by switches, the spanning tree protocol used by switches, IP, Mobile-IP, RTP/RTCP, RIP, OSPF, UDP, TCP, HTTP, FTP, Telnet, etc. More protocols and

devices are for other types of networks such as GSM/GPRS cellular networks and optical networks have been developed.

3.6.4 Application Compatibility and Extensibility

All real-life existing or to-be-developed UNIX application programs can be run on a simulation network to generate realistic network traffic. Users do not need to modify these programs. These programs can be easily run on a simulated network as long as these UNIX programs can be correctly run on the real-world network. In addition, all real-life existing UNIX network configuration tools (e.g. route, ifconfig, netstat, tcpdump) can be run on a simulated network to configure or monitor a simulated network. Users can easily use these tools provided by a UNIX system.

3.6.5 User Friendliness

NCTUns 1.0 provides an integrated and professional GUI environment in which users can easily conduct network simulations. All settings and configurations can be easily set up through GUI. This includes drawing network topologies, configuring the protocol modules used inside a node, specifying the initial locations and moving paths of mobile nodes, plotting network performance graphs, playing back the animation of a logged packet transfer trace, etc.

3.6.6 Open System Architecture

By using a set of module APIs that are provided by the simulation engine, a protocol module developer can easily implement his or her own protocol and integrate it into the simulation engine. For example, user can easily develop and test his or her routing protocol used by ad-hoc mode mobile node in our simulator.

4. Related Work

In this section, we will discuss some related work.

The NIST Net [14] network emulator is a general-purpose tool for emulating performance in IP networks. It mainly operates at the IP level. It can emulate the critical end-to-end performance characteristics imposed by various wide area network situations (e.g. congestion loss) or by various underlying subnetwork technologies (e.g., asymmetric bandwidth situations of xDSL and cable modems.). It is implemented as a Linux kernel module, and user can install the module into a Linux system running on a PC-based machine. This approach is very similar to ours. Both the NIST Net and the NCTUns emulator use the in-kernel IP layer to do an emulation. However, the NCTUns emulator can do more things that the NIST Net can do. This is because the NIST Net only allows a single Linux PC to be set up as a router to emulate a wide variety of network. It can not become an end-point of a TCP or UDP connection. Furthermore, it can not allow users to develop his/her protocol modules. For example, if a user develops a new packet scheduling algorithm, the NIST Net can not provide a mechanism to allow the user to test their algorithms in an emulation. Due to the module-based platform provided by the NCTUns 1.0, however, users can easily implement their protocol modules and test these modules in an emulation.

There is another kind of network emulators. Unlike the NIST Net and the NCTUns emulator, they do not develop a general package to be installed on a FreeBSD or Linux system running on a general-purpose PC. Rather, they develop a special device or hardware to achieve the emulation purpose. The PacketStorm IP network emulator [4] and the Hammer PacketSphere [15] belong to this kind of network emulator. Their advantages may be the performance because they can be

equipped with varied network interfaces, high-speed processors, larger memory, and more efficient operating system, etc. A general PC-based machine on the other hand may not have these powerful equipments. This is because a general operating system such as FreeBSD or Linux is not specially designed for the emulation purpose. Despite this disadvantage, our emulator and other similar related work provide a low-cost way to implement a network emulator. Normal users, researchers or students do not need to spend too much money to do emulation experiments with our network emulator.

A related work that is very similar to ours is the ns-2 [3]. Originally Ns-2 is a network simulator just like the NCTUns 1.0 network simulator. It also has the ability to turn itself into a network emulator. Ns-2 is a traditional network simulator and is implemented as a user-level program. Therefore, when it is turned into a network emulator, it needs a mechanism to capture packets from kernel just like that our emulation daemons do. Ns-2 with emulation has two modes: opaque mode and protocol mode. In the opaque mode, the simulator (ns-2) acts like a router allowing real-world traffic to be passed through. In this mode, its function is similar to NIST Net. In the protocol mode, ns-2 can be used as an end-point to generate TCP or UDP traffic. In other words, ns-2 can also provide the function shown by figure II-3.4(a). So far to the best of the author's knowledge, only the NCTUns emulator and ns-2 can provide this kind of ability.

5. Design and Implementation

In this chapter, we will describe and discuss how NCTUns network simulator can be turned into a network emulator. We will introduce the concepts about our emulation design. After reading this chapter, readers will get a big map of the

NCTUns emulator.

5.1 User-Level Daemon

We implement a user-level daemon to direct packets to the correct direction. This is the main job of the emulation daemon. The emulation daemon will capture packets from the kernel, and then either put packets into the simulation network or inject packets to the real-world network. Figure II-5.1(a) and II-5.1(b) depict what the emulation daemon does. In the following sections, we will use the two figures to explain our design in details.

Using user-level daemons has two advantages. First, it is easier to implement than using a kernel module. Second, it is easy to port them to UNIX-like systems. Because NCTUns network simulator can be run on FreeBSD 4.x and Linux 2.4.x, we also want to support emulation on both systems. Therefore, a kernel module would not suit our needs.

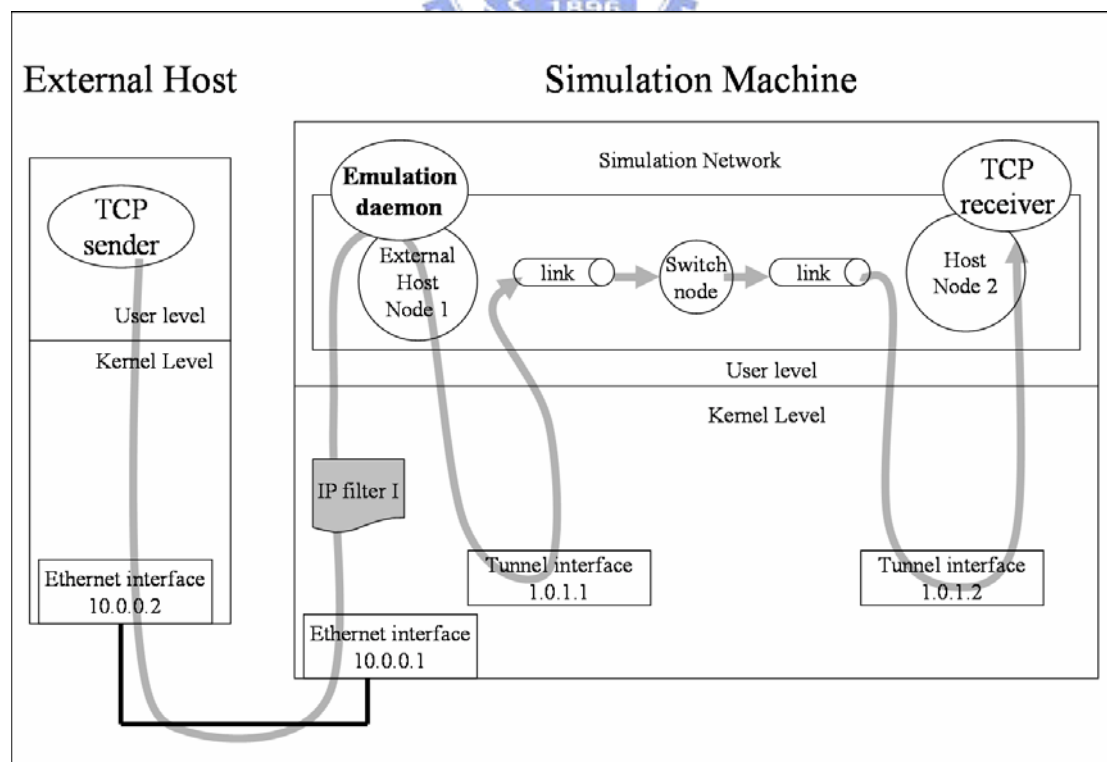


Figure II-5.1(a):
The emulation daemon receives packets from the real-world network (Ethernet network), and then directs them into the simulation network.

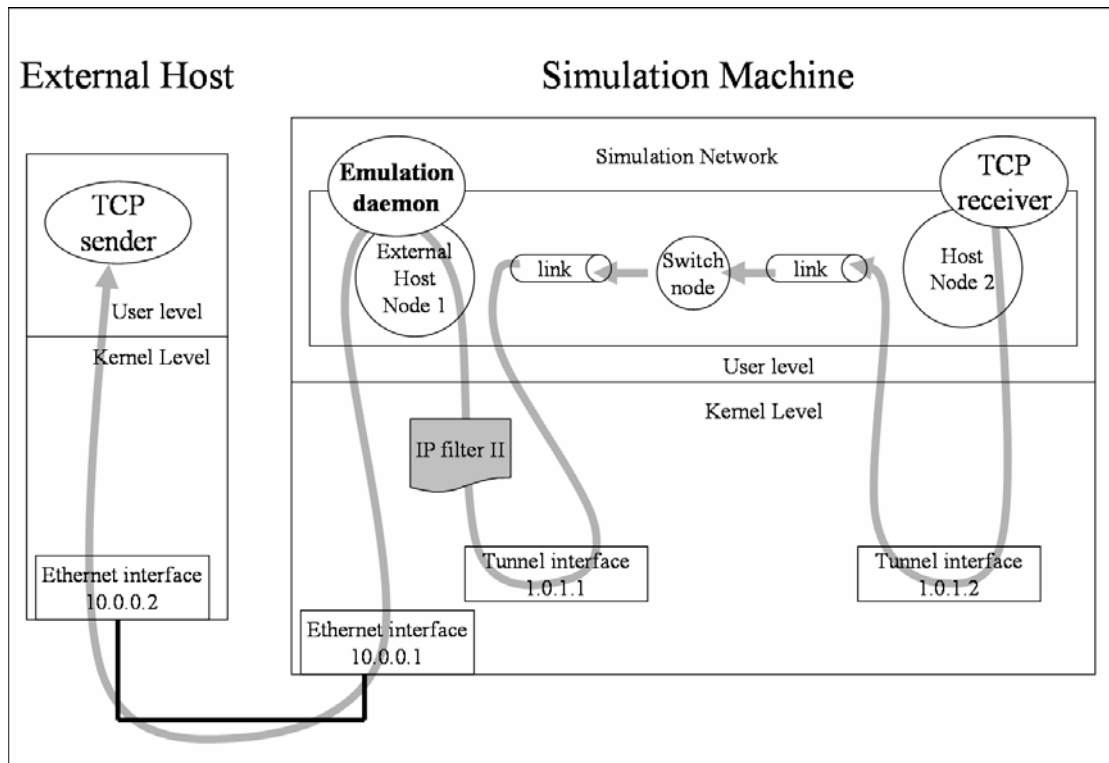


Figure II-5.1(b):
The emulation daemon receives packets from the simulation network, and then injects them into the real-world network (Ethernet network).

5.2 Capture Packets from the Kernel

The emulation daemon should have an approach or a mechanism to capture packets from the kernel. The emulation daemon also should be able to set packet filter rules according to its requirement. As such, the kernel should provide appropriate services to help the emulation daemon filter packets. In this section, we will describe how emulation daemon can capture packets from the kernel under FreeBSD and Linux systems.

5.2.1 Using Divert Socket in FreeBSD

In FreeBSD, there is a special socket type that Linux does not support. This is the divert socket. At the user-level, a program can create a divert socket with a port number (e.g. 2000). When a user configures an IP filter rule (normally using Berkeley

Packet Filter, called BPF), she can indicate which port number a captured packet should be to divert to if that packet is matched by this rule. The following command is an example of setting a filter rule with a divert port number:

```
# ipfw divert 2000 ip from 192.168.1.2 to any in
```

“ipfw” is the user-level program which is used to manage filter rules that are maintained in the kernel. The rule means that if there is any packet that uses IP protocol, its source ip address is 192.168.1.2, and it is a incoming packet (not outgoing), the kernel should divert this packet to the divert socket with port number 2000 (“to any” means that we do not care the destination IP address). Therefore, as long as we set appropriate rules, the emulation daemon can capture packets that it wants via `recvfrom()` system call. If the emulation daemon wants to put the packet back to the kernel, it can simply use `sendto()` system call via the divert socket. In figure II-5.1 (a) and (b), the “IP filter” and the arrow of sending matched packets to the emulation daemon are the result of using divert socket and BPF. This situation is feasible under FreeBSD.

5.2.2 Using Netfilter and Added System Calls in Linux

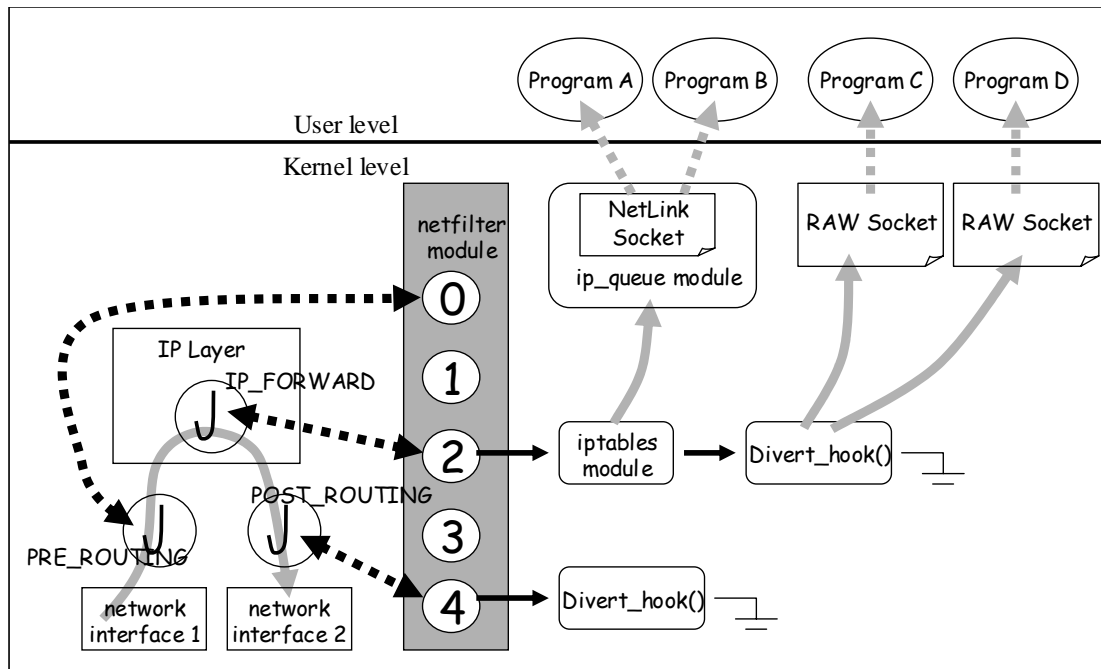


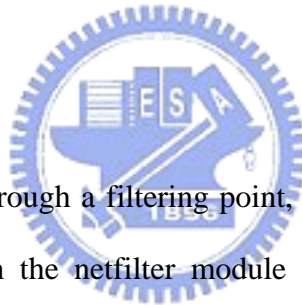
Figure II-5.2.2: The original architecture of the IP packet filtering mechanism in Linux and our proposed approach

I. The Original Filtering Mechanism

In Linux, the kernel does not support the divert socket type. In the Linux kernel, it supports a mechanism to filter packets which is *netfilter* [5]. Netfilter is a set of hooks inside the Linux kernel that allows kernel modules or functions to register callback functions with the network stack. A registered callback function is then called for every packet that traverses the respective hook within the network stack. Based on the netfilter, *iptables* is a generic table structure for the definition of filter rulesets.

Figure II-5.2.2 shows the architecture of these modules. In the netfilter module, it maintains a global structure which has five lists and each list represents a hook number. Every hook number corresponds to a packet filtering point inside the network stack. For example, in figure II-5.2.2, a packet incoming from the network interface 1 will be forwarded at the IP layer, and the IP layer will choose the network

interface 2 as the packet's output device. In the forwarding path, the packet will pass through three filtering points (note that each hook number represents a filtering point) -- IP_PRE_ROUTING, IP_FORWARD, and IP_POST_ROUTING. When the packet passes through the IP_FORWARD filtering point, it means that the IP layer is being used to forward this packet. For the same definition, IP_PRE_ROUTING or IP_POST_ROUTING means that the kernel is filtering this packet before or after routing it. Totally, there are five filtering points in the IP network stack. In figure II-5.2.2, we only show three filtering points. The other two are IP_LOCAL_IN and IP_LOCAL_OUT and their corresponding hook numbers are 1 and 3, respectively. They respectively represent the filtering point where the kernel sends out a packet generated by the local host or where the kernel receives a packet whose destination is the same as the local host.



When a packet passes through a filtering point, the kernel will send this packet to the netfilter module. Then the netfilter module will relay this packet to those functions that are registered in the corresponding filtering point (hook number). For example, in figure II-5.2.2, when packet A passes through the filtering point IP_FORWARD, the kernel will send packet A to the netfilter module. The netfilter module will first relay packet A to the iptables module in the list of hook number 2. If the iptables module does not capture packet A, packet A will be sent back to the netfilter module, then the netfilter module will continually relay packet A to the next registered function, divert_hook(). If divert_hook() does not capture packet A, packet A will be sent back to the netfilter module again. At this time, the netfilter module will discover that there is no registered function in the list of hook number 2. Finally, the netfilter module will send packet A back to the network stack (the IP layer) and the packet will continually go through the original path.

In Linux 2.4.x, if a user wants to send packets to the user space, she should use the kernel module “*ip_queue*” and *libipq* library. The *ip_queue* module uses a *Netlink* socket for kernel/user space communication. After netfilter/iptables match packets, the *ip_queue* module can queue these packets into a *Netlink* socket. Then, the user-level program can use the APIs of *libipq* to get and process these packets from the *Netlink* socket. In other words, *libipq* provides some APIs to handle the packets queued in the *Netlink* socket. In figure II-5.2.2, we can clearly see these related operations. The following is an example showing how to specify a rule via iptables:

```
# iptables -A OUTPUT -p icmp -j QUEUE
```

This rule means that any locally generated ICMP packets (e.g. ping output) should be sent to the *ip_queue* module, which will then attempt to deliver the packets to a user space application. If no user space application is waiting, the packets will be dropped. Using *ip_queue*, a user space program can get and modify these queued packets in the userspace. Then the user space program can specify what to do with these packets (such as ACCEPT or DROP) before reinjecting them back to the kernel. However, this mechanism is not suitable for our design. This is because this mechanism has a big difference with FreeBSD’s divert socket.

II. Using Netfilter and Added System Calls in Linux

The difference is in the number of packet queues. In FreeBSD, every divert socket has one packet receive queue that can store the filtered packets. Each user space program can create a divert socket with its own socket receive queue. Because every rule can specify its own port number (see the example in section 5.2.1), every program can easily get the packets that it wants. Therefore, when more than one

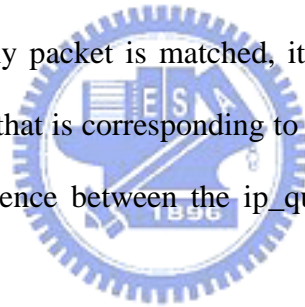
program creates divert sockets with different port numbers, packets captured by the kernel can be sent to the different divert sockets according to their port numbers. However, in Linux, there is only one packet queue available in the whole system. In other words, the ip_queue module only creates a Netlink socket to queue captured packets. In this design, every program needs to read matched packets from the same global queue and decides whether they are what it wants. Figure II-5.2.2 clearly illustrates this concept. As such, program A may receive packets that program B wants. However, program A may not want these packets. Program A may thus drop these packets or put them back to the kernel. If this situation happens, program B may not get the packets that it wants from the global queue.

In the NCTUns emulator, we use an emulation daemon to manage each external host (i.e., an emulation daemon for an external router). Multiple daemons will want to capture their packets from the kernel. For example, mobile-IP daemons or NAT daemons also needs this mechanism. Therefore, multiple programs may capture packets from the kernel at the same time. Obviously, iptables and ip_queue can not satisfy our requirement.

Therefore, we decide to design a mechanism similar to FreeBSD's divert socket by ourselves. Basically, there are two approaches. One is to directly build the divert socket into Linux kernel. This solution mainly has three tasks. First, we need to create a new socket type in the kernel. Second, we need to modify C library to support divert socket type. Third, we need to modify iptables to support queuing packets to different queues based on different divert port numbers. Although this solution can achieve our requirement, we do not take it. The reason is simple -- the effort of using this solution is larger than the other.

The other solution is to turn a RAW socket into a divert socket via added system calls and by hacking the kernel. First, the user space program needs to create a RAW socket. Then it downloads the filter rule and his RAW socket file descriptor into the kernel via a user-defined system call. Of course, we have to maintain the rule table and filtering mechanism by ourselves in the kernel. Every filter rule would be associated with the RAW socket file descriptor. If any packet is matched by this rule, we can simply queue the packet into the corresponding RAW socket receive queue.

In the kernel, we implement a simple kernel function *divert_hook()* to be registered to the netfilter module. In this function, it can filter packets according the downloaded filter rules. If any packet is matched, it will queue the packet into the RAW socket's receive queue that is corresponding to the filter rule. In figure II-5.2.2, we can clearly see the difference between the *ip_queue* module and our proposed approach.



Therefore, in figure II-5.1 (a) and (b), the “IP filter” and the arrow of sending matched packets to the emulation daemon are the result of using netfilter module, *divert_hook()* function, and the added system calls. Of course, this platform is under Linux. In section 4.3.8 of part I, we will present the implementation of related kernel functions and system calls.

5.3 Design of Emulation Daemon for External Host

Besides capturing packets from the kernel, the emulation daemon has another important task. This is to direct packets to the correct direction. This section will

describe how to divert packets from real-world hosts to the simulation machine and how to divert packets from the simulation machine to real-world hosts.

5.3.1 Adding Routing Entries

In figure II-5.1(a), the external host (the real-world host) is equipped with a Fast Ethernet interface and this interface is configured with an IP address 10.0.0.2. The simulation machine's Fast Ethernet interface is configured with an IP address 10.0.0.1. In simulation network, the external host node 1 represents the external host and the external host node 1 has a tunnel interface with an assigned IP address 1.0.1.1 (in our simulator, the IP address format is always 1.0.X.X. [1]). In figure II-5.1(a), if a TCP sender that is running on the external host wants to communicate with the TCP receiver that is running on host node 2, it should use 1.0.1.2 as the destination IP address. On the other hand, if a TCP sender running on host node 2, it should use 1.0.1.1 as the destination IP address to establish a TCP connection with the TCP receiver that is run on the external host (Note: we can not and should not use 10.0.0.2 as the destination IP address. We will explain this in section 5.3.2). As such, when the external host wants to send packets to the simulation machine, we can simply add a routing entry at the external host:

```
# route add 1.0/16 10.0.0.1  
(Assuming that the external host is a FreeBSD system)
```

The above command indicates that all outgoing packets whose destination IP address is 1.0.X.X should be first sent to the gateway whose IP address is 10.0.0.1. Because the simulation machine is configured with the IP address 10.0.1.1. As such, if any packet matches this rule, it will be sent to the simulation machine.

5.3.2 Translate IP Address

Until now, readers may be confused with the two IP address: 1.0.1.1 and 10.0.0.2. This is also the reason why the emulation daemon has to translate IP addresses. In figure II-5.1(a), the TCP sender establishes a TCP connection with the TCP receiver that binds 1.0.1.2 as its source IP address. Therefore, the TCP sender would think that the TCP connection is built between 10.0.0.2 and 1.0.1.2. However, when the TCP sender sends its packets to 1.0.1.2, these packets will be captured and delivered to the emulation daemon and their source IP address may be modified from 10.0.0.2 to 1.0.1.1. This is because the external host node 1 represents the external host in the simulation network and node 1's IP address is 1.0.1.1. All packets generated by the external host should be treated as that they were sent out from 1.0.1.1. If we do not do this, the simulation network will not know which node 10.0.0.2 belongs to. With this modification, the TCP receiver will think that the TCP connection is built between 1.0.1.1 and 1.0.1.2. The TCP receiver will return a TCP ack packet with destination IP address 1.0.1.1 (Of course, the source IP address is 1.0.1.2).

When the TCP ACK packet comes back to the external host node 1 (figure II-5.1(b)), the packets will be captured and delivered to the emulation daemon by the IP filter II. Then, the daemon will modify the destination IP address 1.0.1.1 to 10.0.0.2. In the mean time, its source IP address is still 1.0.1.2. Therefore, the ack packet can easily reach the external host and the TCP sender will not know that its packets' IP addresses have been modified.

5.3.3 Translate Port Number

Why does the emulation daemon need to translate the port number that is inside the UDP/TCP header? This is due to the design of NCTUns network simulator. In section 4.3.4 of part I, we have explained why to do port number mapping and translation in the kernel. In figure II-5.1(a), we assume that the TCP receiver's virtual port number is 8000 and the real port number is 5000. We also assume that the TCP sender's port number is 4000 (The TCP sender does not have virtual port number because it is run on a real-world host.). When the TCP sender sends out a TCP data packet A, the packet's destination port number will be 8000, and its source port number is 4000. Then, when the packet A arrives at host node 2, the destination port number will need to be modified to 5000 (Source port number will still be 4000 because the kernel can not find any virtual port corresponding to port number 4000 in external host node 1). Continuing with figure II-5.1(b), the kernel will just exchange the destination/source port number pair of packet A to be as the TCP ack packet B's destination/source port number pair. Therefore, the ack packet B's destination/source port number pair will be 4000/5000. If we do not modify packet B's port number, the TCP sender may think the TCP receiver is bound at port number 5000. However, the TCP sender originally expects to establish a TCP connection with the foreign port number 8000, not 5000. Therefore, this connection can not be set up. For this reason, when the packet B is captured by the emulation daemon, we should modify its source port number from 5000 to 8000.

Here, a problem happens. How the emulation daemon knows how to modify the packet B's port number? It is simple. When the packet A is captured by the emulation daemon, the emulation daemon records the packet's destination/source port number

pair and destination/source IP address pair. If any packet B is captured by the emulation daemon, we just compare B's destination/source IP address and destination port number with the emulation daemon's record table. If a record is matched, we fill the record's **destination port number** into B's source port number.

5.3.4 Setting Packet Filter Rules

In figure II-5.1 (a), the IP filter rule used by IP filter I should be set like the following:

```
# ipfw add divert 2000 ip from 10.0.0.2 to any in
```

This is very obviously. All packets from 10.0.0.2 should be captured to the emulation daemon that creates a divert socket with divert port number 2000.

In figure II-5.1 (b), the IP filter rule used by IP filter II should be set like the following:

```
# ipfw add divert 2000 ip from any to 1.0.1.1 in
```

This means that all packets whose destination IP address is 1.0.1.1 should be captured to the emulation daemon. Here, readers may find a problem: when host node 2 sends out the packet B (we describe it in section 5.3.3), its destination IP address will be 1.0.1.1 (In section 5.3.2, we already explain why the destination IP address is 1.0.1.1 instead of 10.0.0.2). It is possible that packet B is captured at host node 2 instead of the external host node 1 and as such the packet B will not experience the simulation of the two virtual links and a switch node. However, this situation will not happen. This is because we use the S.S.D.D IP scheme in the kernel, packet B's destination IP address is 1.2.1.1 rather than 1.0.1.1. Only when packet B reaches the tunnel interface 1.0.1.1 (when the packet arrives at the destination node), the destination IP address

will be modified back to 1.0.1.1.

5.4 Design of Emulation Daemon for External Router

The emulation daemon design for external routers is very similar to the one for external hosts. Figure II-5.4 (a) and II-5.4 (b) illustrate what the emulation daemon does.

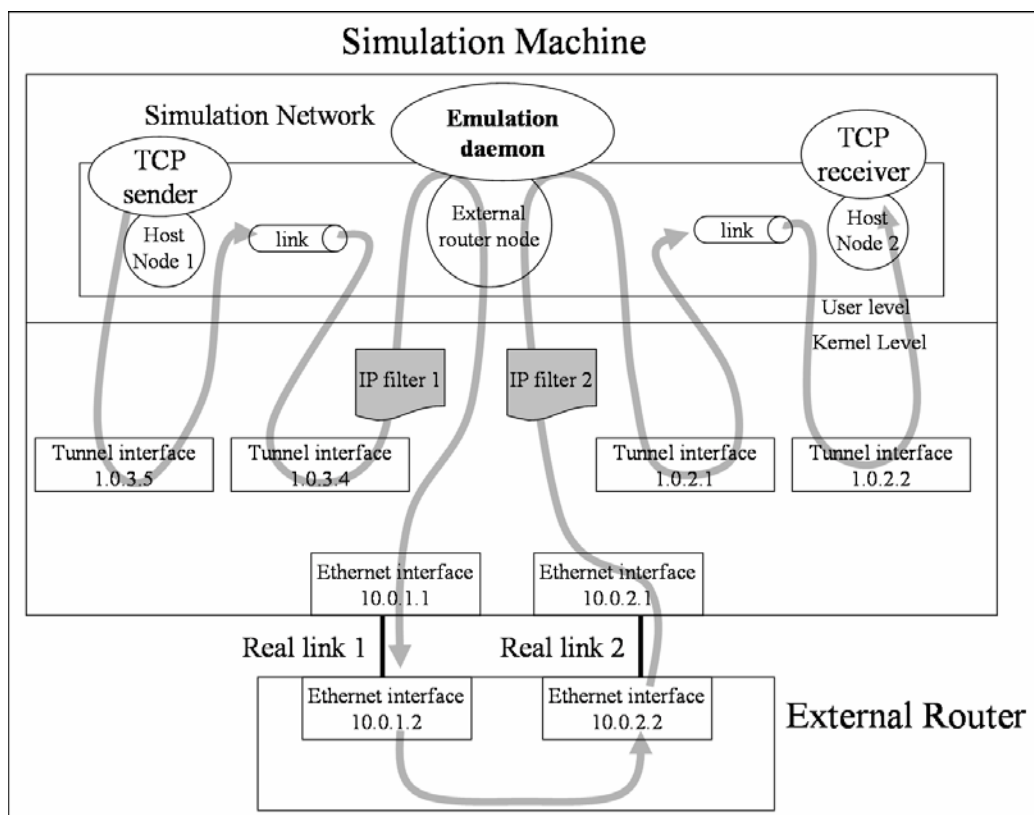


Figure II-5.4 (a): the trace of TCP SYN packet with external router

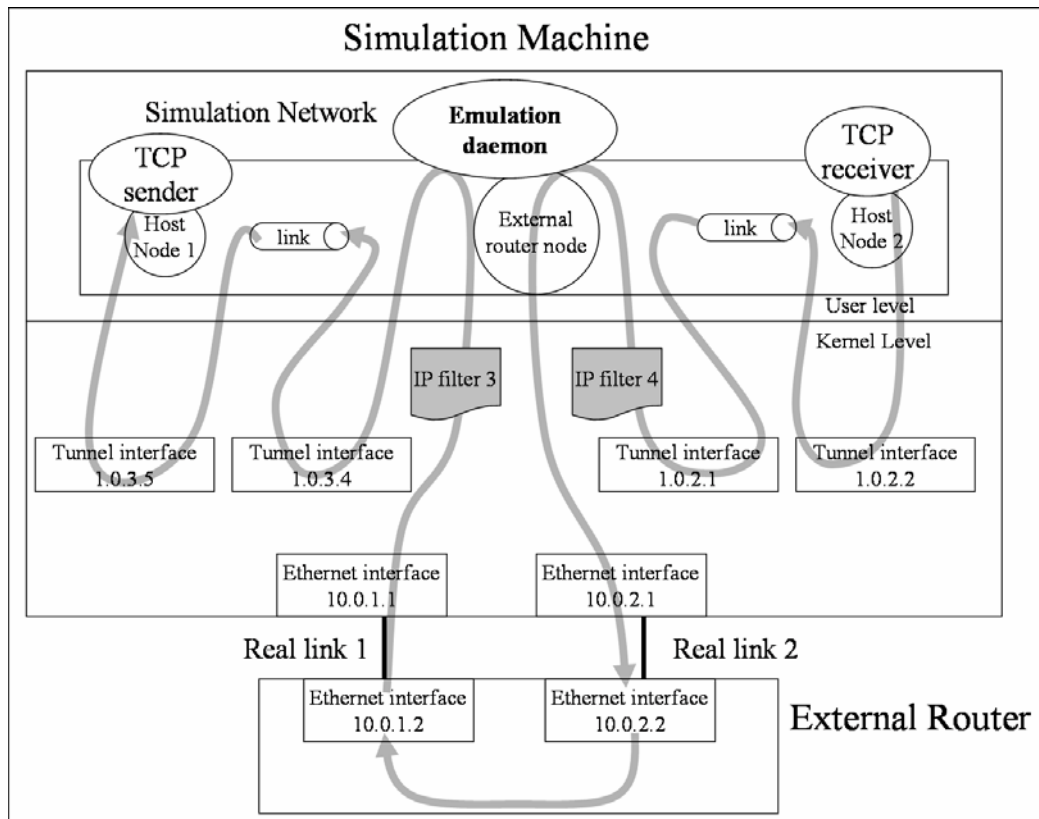


Figure II-5.4 (b): the trace of TCP SYN-ACK packet with external router

In figure II-5.4 (a) and (b), the external router node represents the external router (the real-world router) in the simulation network. The simulation machine has two Ethernet interface whose assigned IP address are 10.0.1.1 and 10.0.2.1. The external router has two interfaces 10.0.1.2 and 10.0.2.2. At the same time, the external router node has two tunnel interfaces whose assigned IP addresses are 1.0.3.4 and 1.0.2.1 in the simulation network. We assume that the TCP sender is on subnet 3 and the TCP receiver is on subnet 2 according to the IP address of the host that the program is run on. The IP filter 1, 2, 3 and 4 represent different filter rules.

In figure II-5.4 (a) and (b), the TCP sender wants to establish a TCP connection with the TCP receiver. Figure II-5.4 (a) shows how the TCP SYN packet reaches host node 2 and figure II-5.4 (b) shows how host node 2 sends a TCP SYN-ACK packet to node 1. From section 5.4.1 to 5.4.4, we will describe our approach. In section 5.4.5 of

part I, we will explain why we use a special IP scheme (200.X.Y.Z format).

5.4.1 Translate IP Address

We apply a special IP scheme to those packets which are sent to the external router. We will use figure II-5.4 (a) and (b) to illustrate the detail. In figure II-5.4 (a), the TCP sender wants to establish a TCP connection with the TCP receiver. The TCP sender will send out a SYN packet with its destination/source IP address pair being 1.0.2.2/1.0.3.5. In kernel, we will translate the IP address pair into the S.S.D.D format [1]. Therefore, 1.0.2.2/1.0.3.5 will be translated to 3.5.2.2/3.5.3.5. When the SYN packet reaches the tunnel interface with the IP address 1.0.3.4, the IP address pair will be modified to 3.4.2.2/3.4.3.5. Then, the SYN packet will be captured to the emulation daemon via IP filter 1. At this moment, the emulation daemon will modify the IP address pair from 3.4.2.2/3.4.3.5 to 200.3.2.2/200.3.2.5. In our scheme, all packets directed to the external router will have the IP address pair 200.X.Y.A/200.X.Y.B. Here, 200 is a special number chosen for setting IP filter rule and routing entries (we will discuss this in section 5.4.5). X.Y means the source/destination subnet number pair. In figure II-5.4(a), the SYN packet starts off from the TCP sender on subnet 3 and wants to communicate with the TCP receiver on subnet 2. So X.Y will be set to 3.2. In practice, emulation daemon can simply get X.Y value according to 3.4.2.2/3.4.3.5. This is because, from the destination/source IP pair 3.4.2.2/3.4.3.5, we can easily know that the packet's original source IP address is 1.0.3.5 and its destination IP address is 1.0.2.2. From the two addresses' third field, we can know that the packet's source subnet is 3 and destination subnet is 2. Finally, A and B are the original fourth value in 3.4.2.2/3.4.3.5. In this example, A is 2 and B is 5. Figure II-5.4.1 shows the concepts.

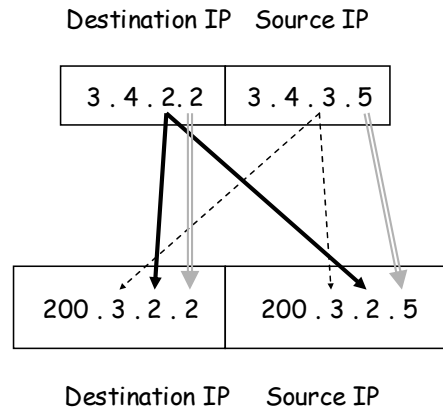
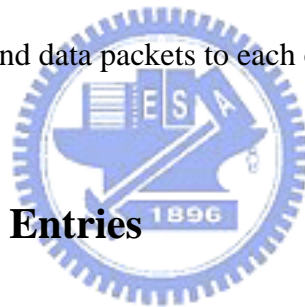


Figure II-5.4.1:
Translate the IP address pair from the S.S.D.D format to the 200.X.Y.Z format

Then, the SYN packet will be directed to the external router according to appropriate routing entry settings. After experiencing the router's packet scheduling and processing, it will return to the simulation machine (In section 5.4.2, we will describe how the external router correctly directs these packets back to the simulation machine). The SYN packet will be captured by IP filter 2. In the emulation daemon, the SYN packet's IP address pair will be modified from 200.3.2.2/200.3.2.5 to 2.1.2.2/2.1.3.5. According to 200.3.2.2, we can know that the destination subnet is 2 (200.X.Y.A, Y means the destination subnet) and the destination IP address's forth field is 200.3.2.2's forth field. Therefore, we can know that the destination IP address is 1.0.2.2. According to 200.3.2.5, we can know that the source subnet is 3 (200.X.Y.B, X is source subnet) and the source IP address's forth byte is 200.3.2.5's forth byte. Therefore, we can realize that the source IP address is 1.0.3.5. Then, in order to put the packet into simulation network, the packet's IP format should use the S.S.D.D format. Therefore, we translate 1.0.2.2/1.0.3.5 to 2.1.2.2/2.1.3.5 so that the packet will be injected into the simulation network via tunnel interface with IP address 1.0.2.1. Then, the SYN packet can correctly reach the TCP receiver.

Continuing with figure II-5.4 (b), the TCP receiver will reply a SYN-ACK

packet with destination/source IP address pair 1.0.3.5/1.0.2.2. Actually, it will be translated to 2.2.3.5/2.2.2.2 in the kernel. The following steps are the same as those in figure II-5.4(a). When the SYN-ACK packet reaches the tunnel interface with IP address 1.0.2.1, the IP address pair will be modified to 2.1.3.5/2.1.2.2. After capturing the packet via IP filter 4, the emulation daemon will modify the IP address pair to 200.2.3.5/200.2.3.2. Then, the SYN-ACK packet will traverse to the external router and return to the simulation machine. Then, IP filter 3 will filter the packet out and send it to the emulation daemon. The daemon will translate IP address pair 200.2.3.5/200.2.3.2 to 3.4.3.5/3.4.2.2. Then, the SYN-ACK packet can be directed back to the simulation network and will reach the TCP sender. Following the steps of figure II-5.4 (a) and (b), the TCP sender and receiver can complete the TCP three-way handshaking and send data packets to each other.



5.4.2 Adding Routing Entries

To automatically divert packets from the simulation machine to the external router, we should add some routing entries on the simulation machine. On the external router, we should also add some routing entries to correctly forward packets back to the simulation machine.

5.4.2.1 Adding Routing Entries on the Simulation Machine

In figure II-5.4 (a), the emulation daemon would modify the SYN packet's destination/source IP address to 200.3.2.2/200.3.2.5 before injecting the packet into the real-world network. Therefore, we can simply use the following command to add the corresponding routing entry:


```
# route add 200.3.2/24 10.0.1.2
```

This command means that any packet whose destination IP address's first three fields are equal to 200.3.2 should be sent to the gateway with IP address 10.0.1.2. In this case, we know that the SYN packet will reach the external router via the real link 1. Therefore, we can add the above routing entry in advance. According to our discussion in section 5.4.1 of part II, we also know that packets will be routed by this rule only if they leave from subnet 3 and their destination subnet is 2.

In figure II-5.4 (b), the emulation daemon will modify the SYN-ACK packet's destination/source IP address pair to 200.2.3.5/200.2.3.2 before injecting the packet into the real-world network. In section 5.4.1, we know that the packet should be sent to the external router via the real link 2. As such, in advance, we can use the following command to add a routing entry on the simulation machine:

```
# route add 200.2.3/24 10.0.1.1
```

It means that any packet whose destination IP address's first three fields are equal to 200.2.3 should be sent to the gateway with IP address 10.0.1.1. We also can know that all of these packets (destination IP address is equal to 200.2.3.X) have the same behavior: leaving from subnet 2 and their destination is subnet 3. To sum up, if a user wants the external router to work correctly in figure II-5.4 (a) and (b), two routing entries listed above should be set simultaneously. Let's to consider a more complex case depicted in figure II-5.4.2.

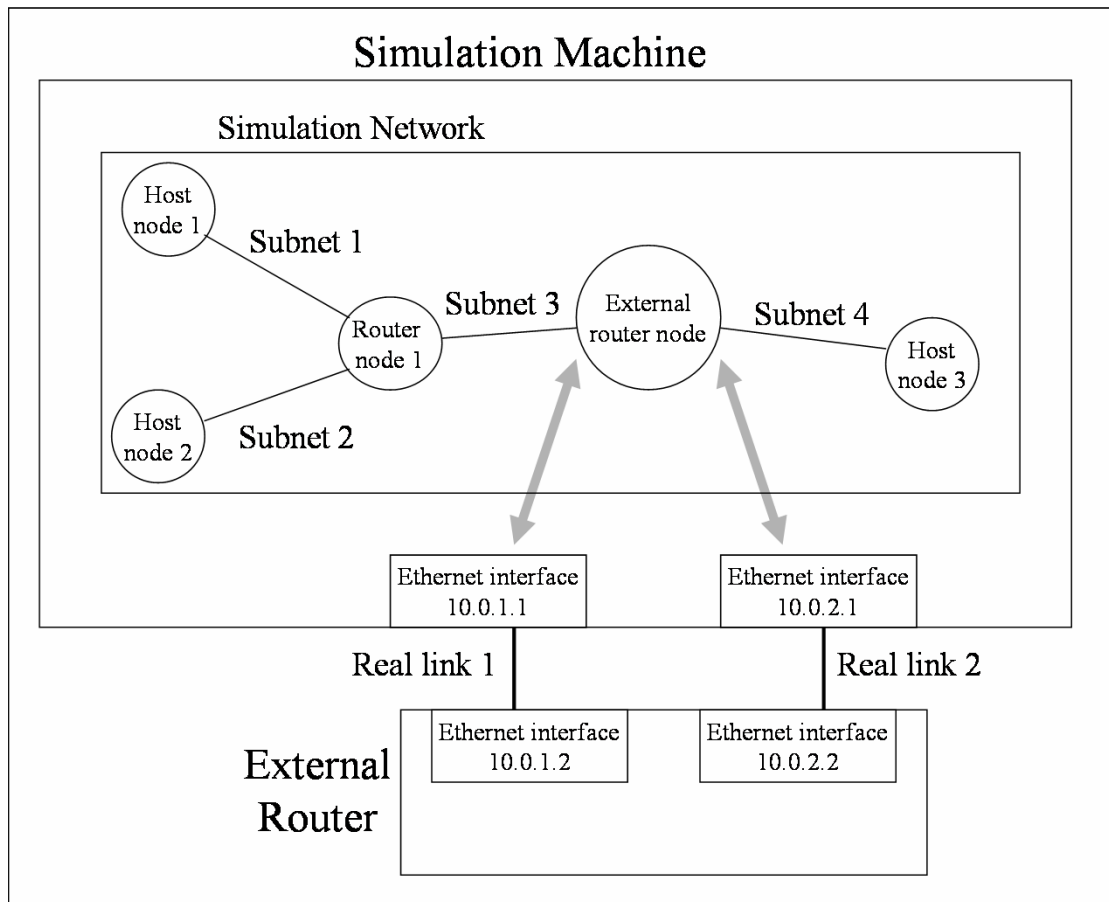


Figure II-5.4.2: A complex topology for an external router

In figure II-5.4.2, there are four subnets in the network topology. For all packets sent out from subnet 1, 2 and 3, they want to reach subnet 4. As such, we should set these packets' gateway to be 10.0.1.2. When these packets traverse to the external router node and are captured by the emulation daemon, their IP addresses will be modified to the corresponding 200.X.Y.Z format and we should tell the kernel how to route these packets. The following are the required routing entries:

```
# route add 200.1.4/24 10.0.1.2.....(1)
# route add 200.2.4/24 10.0.1.2.....(2)
# route add 200.3.4/24 10.0.1.2.....(3)
```

We have discussed the meaning of 200.X.Y format. It means that the packet is sent out from subnet X and destination subnet is Y. If a packet is sent out from subnet 4

and wants to reach subnet 1, 2, or 3, we should set its gateway to be 10.0.2.2. The following are the required routing entries:

```
# route add 200.4.1/24 10.0.2.2.....(4)
# route add 200.4.2/24 10.0.2.2.....(5)
# route add 200.4.3/24 10.0.2.2.....(6)
```

5.4.2.2 Adding Routing Entries on the External Router

On the external router, some routing entries need to be added to its routing table so that packets originated from the simulation network can be redirected back to the simulation network. We use figure II-5.4.2 to show how to add routing entries on the external router.

In figure II-5.4.2, the simulation machine adds routing entry (1), (2), and (3) to divert packets originated from subnet 1, 2, or 3 to the external router. When these packets reach the external router, the external router should send them back to the simulation machine via the Ethernet interface 10.0.2.2. And the simulation machine's Ethernet interface that is connected to the external router's Ethernet interface with IP address 10.0.2.2 is assigned with the IP address 10.0.2.1. Therefore, the external router should set these packets' gateway 10.0.2.1. The following are the required routing entries that should be added on the external router:

```
# route add 200.1.4/24 10.0.2.1
# route add 200.2.4/24 10.0.2.1
# route add 200.3.4/24 10.0.2.1
```

For the same reason, the external router should add the following routing entries for those packets that are diverted to the external router according to routing entry (4), (5), or (6):

```
# route add 200.4.1/24 10.0.1.1
```

```
# route add 200.4.2/24 10.0.1.1
# route add 200.4.3/24 10.0.1.1
```

5.4.3 Unnecessary to Translate the Port Number

In the external host case, the emulation daemon needs to record the packet's destination/source port number pair (in section 5.3.3). However, for the external router case, the emulation daemon does not need to do this. This is because the external router only processes the IP packet's routing, it never touches the transport layer information inside the packet. As everyone knows, port number belongs to the transport layer. In the current version, we do not support running a traffic generator on the external router.

5.4.4 Setting Packet Filter Rules

In figure II-5.4(a), IP filter 1 should be set like the following:

```
# ipfw add divert 2000 ip from 3.4.0.0/16 to 3.4.0.0/16 in
```

According to the S.S.D.D IP scheme, the destination/source IP address pair will be modified to 3.4.x.x/3.4.x.x format after packets arrive at the tunnel interface 1.0.3.4. Therefore, we should use the above rule to capture packets which has reached the tunnel interface 1.0.3.4.

In figure II-5.4 (a), IP filter 2 should be set like the following:

```
# ipfw add divert 2000 ip from 200.0.0.0/8 to 200.0.0.0/8 in
```

According to section 5.4.1 of part II, we know that all packets from the external router have the 200.X.Y.Z IP format. Therefore, we can simply use the above rule to

capture packets that are returned from the external router to the emulation daemon. In figure II-5.4 (b), the above rule also suits IP filter 3.

In figure II-5.4(b), IP filter 4 should be set as:

```
# ipfw add divert 2000 ip from 2.1.0.0/16 to 2.1.0.0/16 in
```

5.4.5 “200.X.Y.Z” Format Discussion

In this section, we will discuss why we need to modify IP address to the 200.X.Y.Z format. In figure II-5.4(a), the TCP SYN packet filtered by IP filter 1 has the destination/source IP address pair of 3.4.2.2/3.4.3.5. First, we should set a filter rule like the following command to capture this packet:

```
# ipfw add divert 2000 ip from 3.4.0.0/16 to 3.4.0.0/16 in.....(a)
```

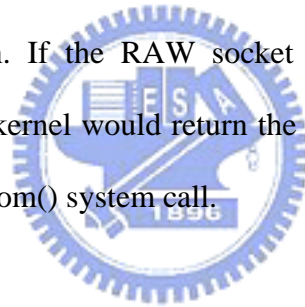
If any packet originated from the simulation network reaches the tunnel interface 1.0.3.4, their destination/source IP address pair will be modified to 3.4.x.y/3.4.a.b format. We can easily use the above filter rule to capture the SYN packet from kernel and the packet will be sent to the external router by the emulation daemon. Therefore, we can just add a routing entry using the following command:

```
# route add 3.4/16 10.0.1.2.....(7)
```

In fact, the SYN packet would be correctly directed to the external router according to above routing entry. However, a problem will happen when the SYN packet returns back to the simulation machine. When the SYN packet is back to the simulation machine, its destination IP address would match the filter rule (a), then the packet will be captured again and again by the emulation daemon.

Here, readers may see a problem. Both the SYN packet from the simulation

network and the SYN packet from the external router will be matched by rule (a). The packet from the simulation network should be sent to the external router and the packet from the external router should be directed to the simulation network. How does the emulation daemon recognize the two different kinds of packets? In other words, how does the emulation daemon know that the packet is coming from the simulation network or from the external router? The solution is simple. In FreeBSD, when a program receives packets from a divert socket, the kernel would provide the program with the packet's incoming interface's IP address via `recvfrom()` system call. Therefore, the incoming interface's IP address of the packet from the simulation network is 1.0.3.4 however the other is 10.0.2.1. Therefore, the emulation daemon can recognize the two packets. In Linux, we also use the same method. We hack the RAW socket implementation. If the RAW socket is turned into a divert socket (section 5.2.2 of part II), the kernel would return the incoming interface's IP address when the program calls `recvfrom()` system call.



Then, the emulation daemon will put the SYN packet back to the simulation network (now, the SYN packet's IP address pair is 3.4.2.2/3.4.3.5), and the packet will pass through the tunnel interface with an assigned IP address 1.0.2.1. In the original design of the NCTUns 1.0, we add a routing entry to route the packet to the tunnel interface 1.0.2.1 in advance:

```
# route add 3.4.2/24 1.0.2.1.....(8)
```

Because the NCTUns network simulator directly uses the system's routing table to forward packets, all routing entries used by NCTUns network simulator are put into the system's routing table. Therefore, the above routing entry (7) and (8) will be put into the same routing table.

Then a problem happens here. We assume that the routing entry (7) is found before the routing entry (8) when the kernel looks up the routing table (We call the SYN packet captured from the simulation network packet A and call the SYN packet sent back from the external router packet B.). According to the previous description, packet A and packet B have the same IP address pair so that packet A and packet B match the entry (7) and the entry (8) at the same time. The packet A would be routed to the external router by the routing entry (7). When it is sent back to the simulation machine, the SYN packet (packet B) hopes to be injected into the simulation network by the routing entry (8). However, because the entry (7) is listed prior to the entry (8) and the packet B also matches the entry (7), the packet B will be routed by the entry (7) instead of the entry (8). Therefore, the SYN packet will be sent to the external router again and again. In this situation, the SYN packet will cause a loop between the simulation machine and the external router (In fact, the kernel will discover this situation and eventually drop the packet.).

Even if the routing entry (8) is located prior to the routing entry (7), the SYN packet still does not traverse along our expected path. Because the packet A will match the routing entry (8) first, the packet would be directly injected back to the simulation network instead of the external router. As such, the SYN packet will reach host node 2 without passing through the external router.

Therefore, we should find a way to solve the routing entry conflict. We propose to modify IP address to 200.X.Y.Z format. When the SYN packet is captured from the simulation network (packet A), we modify its IP address pair to the 200.X.Y.Z format. In the case of figure II-5.4 (a), we will modify the SYN packet's

destination/source IP address pair to 200.3.2.2/200.3.2.5 (section 5.4.1). In order to route the modified packet to the external router, we should add a routing entry like the following:

```
# route add 200.3.2/24 10.0.1.2.....(9)
```

The routing entry (9) will not conflict with the entry (8). At the same time, this 200.X.Y.Z format reserves the all the information needed to recover the original IP address.

5.5 Simulation Speed Should Synchronize With the Real Time

The NCTUns network simulator uses the discrete event simulation methodology to speed up its simulation speed [2]. In a simulation, the simulator uses the virtual clock (virtual time) rather than the real clock. It always advances the simulation time to the smallest time stamp of the event in the even heap. Therefore, when a simulation is running (or we can say an emulation is running), the time advance steps will not be uniform. If more events need to be processed per virtual unit time, the virtual time clock will advance more slowly. Because the external host/router uses the real-world clock rather than the virtual clock, the simulation time must be adjusted to be as fast as the speed of the real-world clock. Otherwise, the time speed would not be synchronized with each other (the simulation network and the real-world host). The emulation results therefore will be wrong.

Our approach is to synchronize the virtual clock with the real-world clock periodically. We adjust the virtual clock to the real-world clock every 1 ms (milli-second) in virtual time. Therefore, the emulation function's latency accuracy is

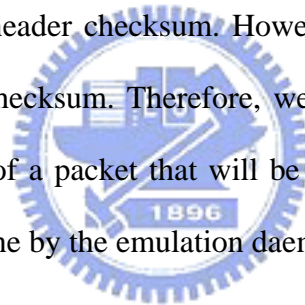
about 1 milli-second.

6. More Consideration

In this chapter, we will discuss more details about the emulation daemon design.

6.1 Re-compute Header Checksum

In NCTUns network simulator, we ignore the check of the IP and TCP/UDP header checksum. This is because we will modify the IP address and the TCP/UDP port number of a packet in a simulation [1]. If we do not ignore them, these packets will be dropped due to bad header checksum. However, the external host normally does not ignore the header checksum. Therefore, we should re-compute the IP and TCP/UDP header checksum of a packet that will be sent to the external host or the external router. This job is done by the emulation daemon.



6.2 More than One External Host

If there is more than one external host physically connected to the simulation machine, we should take care about the order of the filter rules. We will use figure II-6.2 to show what problems will happen.

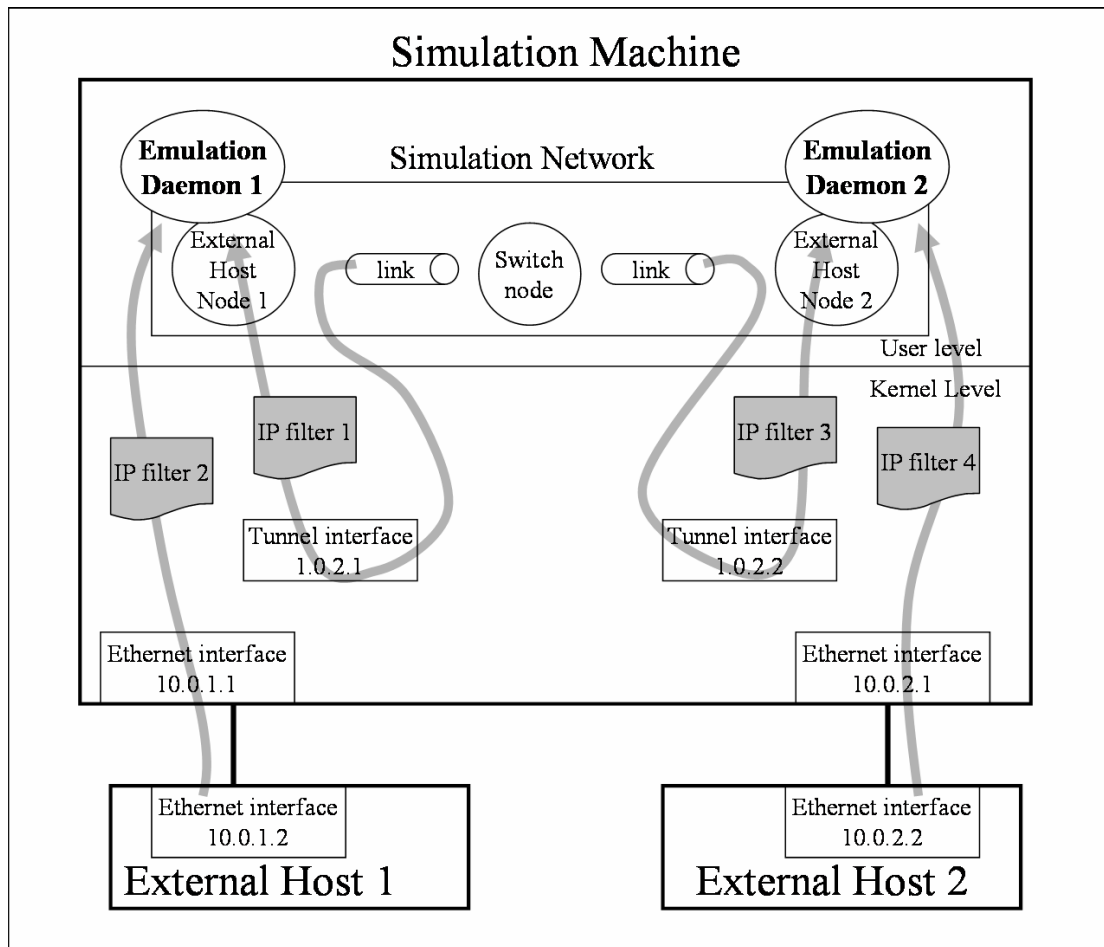


Figure II-6.2: More than one external host

In figure 6.2, external host node 1 represents the external host 1 while external host node 2 represents external host 2. The external host node 1 is configured with a tunnel interface whose assigned IP address is 1.0.2.1 and the external host node 2 has tunnel interface 1.0.2.2 (the two external host nodes are at the same subnet in this emulation case). The two external hosts are respectively connected to the simulation machine on Ethernet subnet 10.0.1/24 and 10.0.2/24. We assume that the emulation daemon 1 binds to the divert port number 2000 while the emulation daemon 2 binds to 3000.

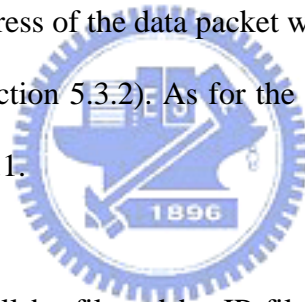
As we discuss how to set the packet filter rule in section 5.3.4 of part II, the emulation daemon 1 will set the following filter rules to capture packets that it wants:

```
# ipfw add divert 2000 ip from any to 1.0.2.1 in.....(IP filter 1)
# ipfw add divert 2000 ip from 10.0.1.2 to any in.....(IP filter 2)
```

At the same time, the emulation daemon 2 would set:

```
# ipfw add divert 3000 ip from any to 1.0.2.2 in.....(IP filter 3)
# ipfw add divert 3000 ip from 10.0.2.2 to any in.....(IP filter 4)
```

In FreeBSD, the filter rules are connected in a rule chain (In Linux, there are five chains, but the similar problem would still happen.) and the packet will be checked whether it is matched by any rule or not from the head of the chain. We assume that the order of above four rules is IP filter 1 -> IP filter 2 -> IP filter 3 -> IP filter 4. We consider a scenario: external host 1 sends a TCP data packet to external host 2. When external host 2 receives the data packet, it would send a TCP ACK packet to external host 1. The destination IP address of the data packet will be 1.0.2.2 instead of 10.0.2.2 (we already discuss this in section 5.3.2). As for the same reason, the ACK packet's destination IP address is 1.0.2.1.



First, the data packet will be filtered by IP filter 2. The data packet (now, its destination/source IP address pair is 1.0.2.2/10.0.1.2) does not match the IP filter 1 and match the IP filter 2. After experiencing the simulation of a switch node and two virtual links, it will be filtered by the IP filter 3. At this time, its IP address pair is 1.0.2.2/2.2.2.1. Then the data packet will be sent to the external host 2. Then, external host 2 will send out the TCP ACK packet with IP address pair 1.0.2.1/10.0.2.2. After the ACK packet is received by the Ethernet interface 10.0.2.1, we wish the ACK packet to be filtered by the IP filter 4 and be diverted to the emulation daemon 2. However, the ACK packet also matches the IP filter 1. According to our assumption, the IP filter 1 is listed before the IP filter 4. Therefore, the ACK packet will be filtered out by the IP filter 1 and diverted to the emulation daemon 1 instead of the

emulation daemon 2.

According to above discussion, we know that we should put the filter rule used to capture packet sent from the external host prior to other filter rules. In other words, we should filter packets according its source IP address as early as possible. Therefore, the correct order of the filter rules in this case should be: IP filter 2 -> IP filter 4 -> IP filter 1 -> IP filter 3. Setting the filtering rules' order is tricky. However, the GUI program can automatically export the correct settings in the correct order for a user.

6.3 More than One External Router

One problem will happen when supporting more than one external router. We use figure II-6.3 to show this problem.

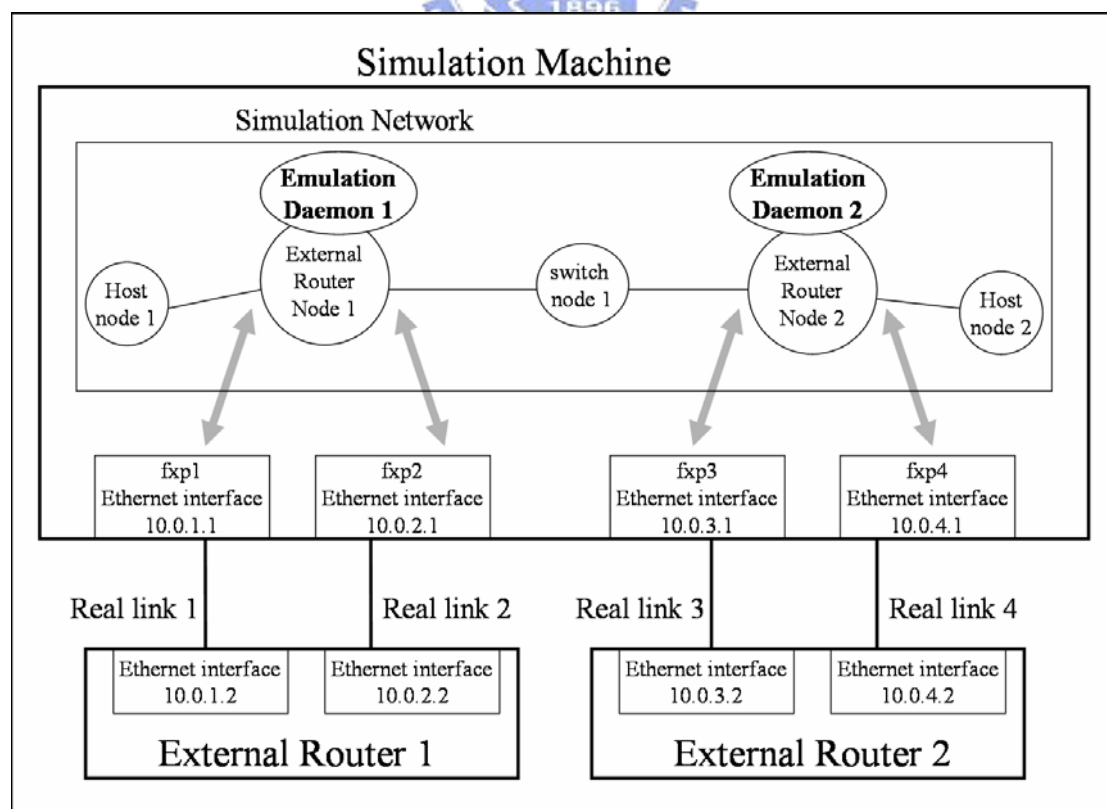


Figure II-6.3: More than one external router

In figure II-6.3, the simulation machine is equipped with four Ethernet interfaces and their names are fxp1, fxp2, fxp3 and fxp4. The emulation daemon 1 binds to divert socket port number 2000, and the emulation daemon 2 binds to 3000.

In section 5.4.4, we have discussed how to set the filter rules for one external router. There are two different kinds filter rules to set. The first one is to capture packets from the simulation network. Because the S.S.D.D IP scheme is used to set this kind of rules, the rule will be unique. In other words, the problem in section 6.2 will not happen. The other kind of rule is to capture packets that are sent by the external router. As we know, the IP address of the packets from external router has the 200.X.Y.Z format. Therefore, we should set the following rules to divert packets to the two emulation daemons:



```
# ipfw add divert 2000 ip from 200.0.0.0/8 to 200.0.0.0/8 in.....(i)
# ipfw add divert 3000 ip from 200.0.0.0/8 to 200.0.0.0/8 in.....(ii)
```

But packets from the external router 1 or from the external router 2 all match the rule (i) and (ii) at the same time. As such, all packets will be diverted to the same emulation daemon. Therefore, we should set filter rules like the following:

```
# ipfw add divert 2000 ip from 200.0.0.0/8 to 200.0.0.0/8 in recv fxp1....(iii)
# ipfw add divert 2000 ip from 200.0.0.0/8 to 200.0.0.0/8 in recv fxp2....(iv)
# ipfw add divert 3000 ip from 200.0.0.0/8 to 200.0.0.0/8 in recv fxp3....(v)
# ipfw add divert 3000 ip from 200.0.0.0/8 to 200.0.0.0/8 in recv fxp4....(vi)
```

These rules add a extra condition. That is, packets should be filtered according to their input devices. Here, “in recv fxp1” means to capture the packets whose incoming interface is the interface fxp1. With this arrangement, all packets whose incoming interface is fxp1 or fxp2 will be diverted to the emulation daemon 1. All packets whose incoming interface is fxp3 or fxp4 will be diverted to the emulation

daemon 2.

7. Evaluation

In this section, we will run some emulation cases to test the accuracy of the NCTUns emulator. All emulation cases are run on the Linux version of the NCTUns network simulator.

7.1 Accuracy

I. Emulation Setup

Figure II-7.1.1 shows the network topology for the first emulation case. Both the used emulation machine and the real external host are IBM A30 notebook computers. The two machines are physically connected with a 100 Mbps Fast Ethernet network. In figure II-7.1.1, the external host node represents the real external host in the emulator. Other hosts and routers are all simulated by the NCTUns network simulator. In the test suite, we vary the link delay and the synchronization interval time to show the time precision of our emulator. Each link delay will be set to 0.5, 1, 5, 10, and 50 milli-seconds such that the expected RTT (Round Trip Time) between the external host node and the node 1 will be 10 (0.5×20), 20 (1×20), 100 (5×20), 200 (10×20), and 1000 (50×20) milli-seconds, respectively. The bandwidths of all links are set to 10 Mbps. At the real external host, we will run a “ping” program to measure the RTT between node 1 and the real external host. The time interval of two successive ICMP request packets is set to 0.5 seconds and the packet length is set to 64 bytes. The synchronization interval time is set to 0.1, 1, and 10 milli-seconds in virtual time. We will compare the average and the variance of measured RTTs to show the effect of

the synchronization interval time.

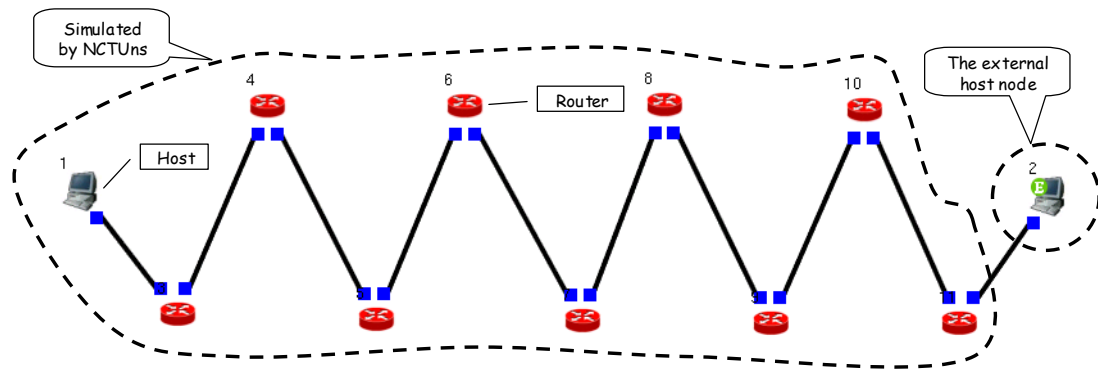


Figure II-7.1.1: The emulation case for testing the accuracy of RTT

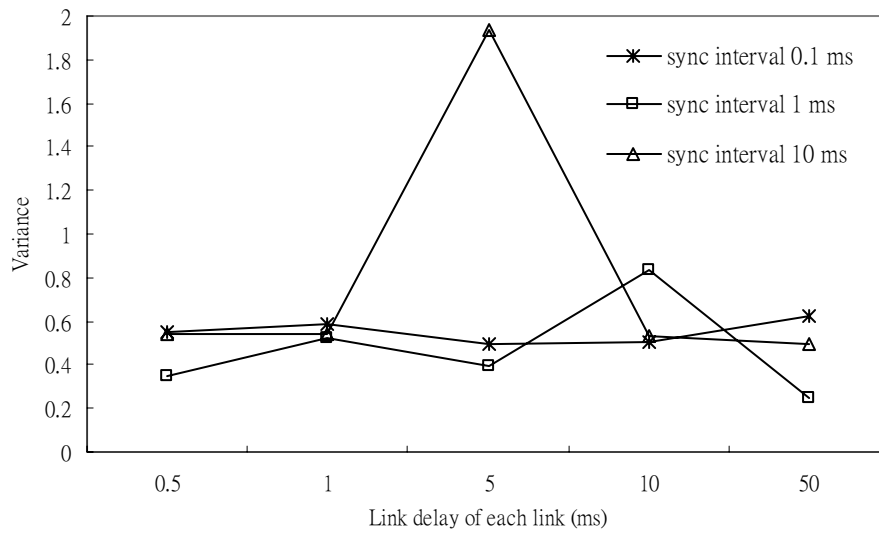


Figure II-7.1.2: The variance of RTT

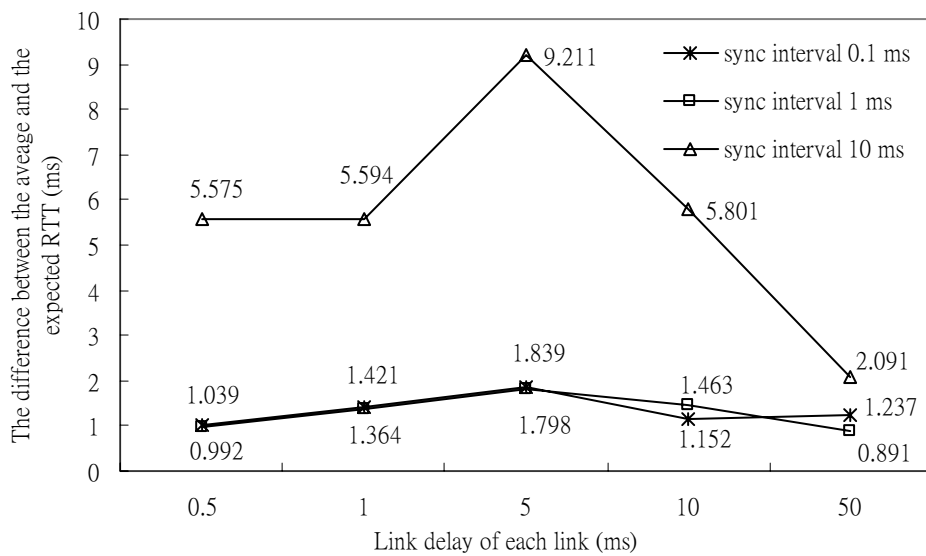


Figure II-7.1.3: The difference between the average and the expected RTT

II Report

Figure II-7.1.2 shows the variance of the measured RTTs. We can see that the two cases, whose synchronization interval times are 0.1 ms and 1 ms, have about the same performance. However, the case whose synchronization interval time is 10 ms has a worse performance. Figure II-7.1.3 shows the difference between the average of measured RTTs and the expected RTT. The expected RTT will be 20 times the time of each link delay because an ICMP request packet will pass through 20 links in a RTT. In figure II-7.1.3, we can see that the case of using a synchronization interval time of 10 ms has a large difference from the expected RTT. As such, we can say that the synchronization interval time 10 ms is too large to let our emulator have a high time precision. For the other two cases, the difference is about 1.4 milli-seconds. We think that this overhead is made by some delays. The first is the delay of the Fast Ethernet network. This is because packets need to be diverted from the real external host to the simulation machine via the real-world network. The simulation machine also needs to divert packets to the real external host via a real-world network. Another one is the delay of the emulation daemon's processing. The emulation daemons should capture packets from kernel, and then should inject the packets into either the simulation network or the real-world network. These operations is very costly because the operation of memory copy needs to be done between the kernel level and the user level. Another one is the transmission time of each packet. For this emulation case, the transmission delay of each packet will be about 0.051 (64byte/10Mbps) milli-seconds. When passing through 20 links, the total time will be 1.02 (0.051*20) milli-seconds. Therefore, we can know that the transmission delay will be the maximum portion of the difference between the average of measured RTTs and the expected RTT. To sum up, the difference shown in figure II-7.1.3 is reasonable. Therefore, we can say that the accuracy of our emulator is quite accurate

for this emulation case.

7.2 Throughput

In this section, we will use an emulation case to show the behavior of two contending greedy TCP connections in our emulator.

I. Emulation Setup

Figure II-7.2.1 shows the test network topology. All used machines are the same with the previous emulation case. The link delays and bandwidths of all links are set to 1ms and 10 Mbps. For the traffic settings, we will establish two greedy TCP connections. The first one is between node 1 and node 4. We call this connection a simulated TCP connection because both of two nodes are simulated by the NCTUns. The other is between node 1 and the real external host. We call this connection a real TCP connection because the real external host is a real-world host. The two connections will contend for the output queue of the switch. In other words, the output queue of the switch will be the bottleneck. Then we will change the real TCP connection with a simulated connection and re-run this case. By comparing these two cases, we can verify the behavior of two contending TCP connections.

II. Report

Figure II-7.2.2 shows the contending behavior between a real external host and a simulated host. Figure II-7.2.3 shows the contending behavior of two simulated TCP connections. We can clearly see that both of two figures show almost the same behavior. This result verifies the accuracy of our emulator.

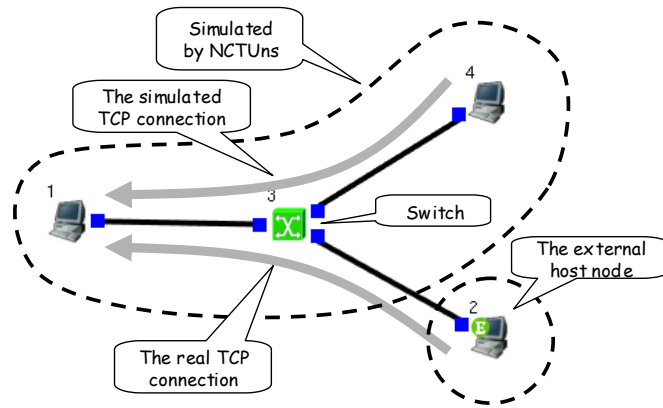


Figure II-7.2.1: The emulation case for testing two contended greedy TCP connections

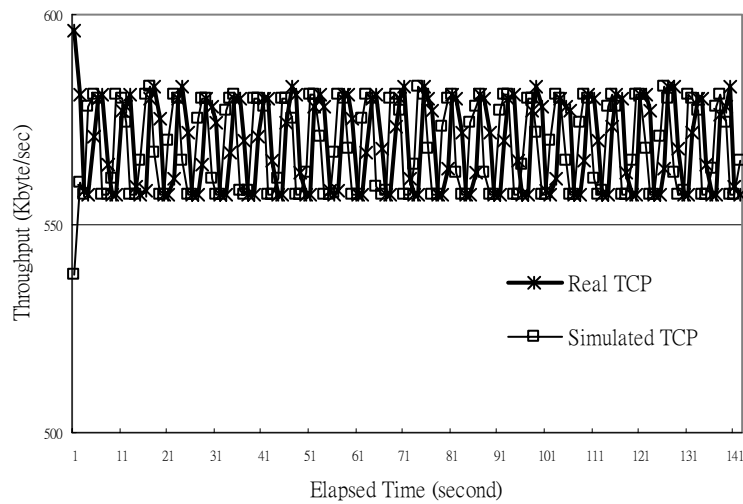


Figure II-7.2.2: The contending behavior between a real TCP connection and a simulated TCP connection

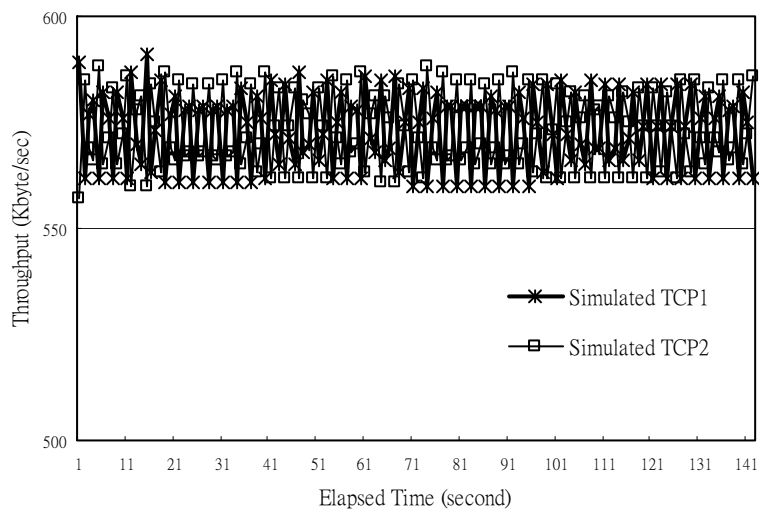
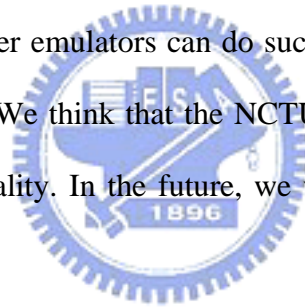


Figure II-7.2.3: The contending behavior between two simulated TCP connections

8. Concluding Remarks for Part II

In part II, we present the internal design and implementation of the NCTUns network emulator. The emulator is based on the NCTUns 1.0 network simulator. Therefore, it reserves the features of the NCTUns 1.0 simulator such as supporting various networks, supporting various network devices, supporting various network protocols, supporting open system architecture, having a user-friendly GUI environment, etc. In addition, the NCTUns emulator provides several unique features that other related work do not have. For example, it can establish TCP/UDP connections between simulated nodes and real hosts. It can also let an external host become an ad-hoc/infra-structure mode mobile node in an emulation. Our emulator can also do the work that other emulators can do such as interacting with real hosts, interacting with real routers. We think that the NCTUns emulator is among the best regarding emulation functionality. In the future, we will continue to improve it and develop new functions for it.



Reference

- [1] S.Y. Wang, C.L. Chou, C.H. Huang, C.C. Hwang, Z.M. Yang, C.C. Chiou, and C.C. Lin , “The Design and Implementation of the NCTUns 1.0 Network Simulator”, Computer Networks, Vol. 42, Issue 2, June 2003, pp. 175-197.
- [2] S.Y. Wang, C.L. Chou, C.C. Hwang, A.J. Su, C.C. Lin, K.C. Liao, H.Y. Chen, and M.C. Yu, “Applying Discrete Event Simulation to the NCTUns 1.0 Network Simulator”.
- [3] S. McCanne, S. Floyd, ns-LBNL Network Simulator,
<http://www.isi.edu/nsnam/ns/>
- [4] PacketStorm Communications, Inc., <http://www.PacketStorm.com>
- [5] The netfilter/iptables project, <http://www.netfilter.org/>
- [6] S.Y. Wang and H.T. Kung, “A Simple Methodology for Constructing Extensible and High-Fidelity TCP/IP Network Simulators”, IEEE INFOCOM’99, March 21-25, 1999, New York, USA.
- [7] S.Y. Wang and H.T. Kung, “A New Methodology for Easily Constructing Extensible and High-Fidelity TCP/IP Network Simulators”, accepted and to appear in “Computer Networks” Journal.
- [8] OPNET Inc., <http://www.opnet.com>
- [9] Harvard TCP/IP network simulator 1.0, available at
<http://www.eecs.harvard.edu/networking/simulator.html>
- [10] Richard M. Fujimoto, “Parallel and Distributed Simulation Systems,” John Wiley & Sons, Inc, 2000.
- [11] TrollTech Inc., <http://www.trolltech.com/products/qt/index.html>.
- [12] Daniel P. Bovet and Marco Cesati, “Understanding the Linux Kernel, 2nd Edition”, O’Reilly, 2002.
- [13] Chih-Hua Hwang, “The Design and Implementation of the NCTUns 1.0 Network Simulation Engine”, Master thesis, National Chiao Tung University, Hsinchu, Taiwan, 2002.
- [14] Nist net, available at <http://snad.ncsl.nist.gov/itg/nistnet>.
- [15] Empirix Inc., <http://www.empirix.com/>

[16] Fedora project, <http://fedora.redhat.com/>

