Available at www.ComputerScienceWeb.com
POWERED BY SCIENCE DIRECT®

ELSEVIER

Computer Networks 42 (2003) 175–197

COMPUTER NETWORKS

www.elsevier.com/locate/comnet

The design and implementation of the NCTUns 1.0 network simulator

S.Y. Wang *, C.L. Chou, C.H. Huang, C.C. Hwang, Z.M. Yang, C.C. Chiou, C.C. Lin

Department of Computer Science and Information Engineering, National Chiao Tung University, 1001 Ta Hsueh Road, 30050 Hsinchu, Taiwan

Received 2 May 2002; received in revised form 20 November 2002; accepted 20 January 2003

Responsible Editor: I. Nikolaidis

Abstract

This paper presents the design and implementation of the NCTUns 1.0 network simulator, which is a high-fidelity and extensible network simulator capable of simulating both wired and wireless IP networks. By using an enhanced simulation methodology, a new simulation engine architecture, and a distributed and open-system architecture, the NCTUns 1.0 network simulator is much more powerful than its predecessor—the Harvard network simulator, which was released to the public in 1999. The NCTUns 1.0 network simulator consists of many components. In this paper, we will present the design and implementation of these components and their interactions in detail.

Keywords: Network simulator; Simulation methodology

1. Introduction

Network simulators implemented in software are valuable tools for researchers to develop, test, and diagnose network protocols. Simulation is economical because it can carry out experiments without the actual hardware. It is flexible because it can, for example, simulate a link with any bandwidth and propagation delay or a router with any queue size and queue management policy. Simulation results are easier to analyze than experimental results because important information at critical points can be easily logged to help researchers diagnose network protocols.

Network simulators, however, have their limitations. A complete network simulator needs to simulate networking devices (e.g., hosts and routers) and application programs that generate network traffic. It also needs to provide network utility programs to configure, monitor, and gather statistics about a simulated network. Therefore, developing a complete network simulator is a large effort. Due to limited development resources, traditional network simulators usually have the following drawbacks:

- Simulation results are not as convincing as those produced by real hardware and software

* Corresponding author.
E-mail address: shieyuan@csie.nctu.edu.tw (S.Y. Wang).

1389-1286/03/$ - see front matter
doi:10.1016/S1389-1286(03)00181-6

equipment. In order to constrain their complexity and development cost, most existing network simulators can only simulate real-life network protocol implementations with limited detail, and this can lead to incorrect results. For example, OPNET's modeler product [1] uses a simplified finite state machine model to model complex TCP protocol processing. As another example, in ns-2 [2] package, it is documented that "there is no dynamic receiver's advertised window for TCP."

- These simulators are not extensible in the sense that they lack the standard UNIX POSIX application programming interface (API). As such, existing or to-be-developed real-life application programs cannot run normally to generate traffic for a simulated network. Instead, they must be rewritten to use the internal API provided by the simulator (if there is any) and be compiled with the simulator to form a single big and complex program. For example, since the ns-2 network simulator itself is a user-level program, there is no way to let another user-level application program "run" on top of it. As such, a real-life application program cannot run normally to generate traffic for a network simulated by ns-2.

To overcome these problems, Wang proposed a simulation methodology in [3,4] and used it to implement the Harvard network simulator. The Harvard network simulator has two desirable properties as follows. First, it uses the real-life UNIX TCP/IP protocol stack, real-life network application programs, and real-life network utility programs. As such, it can generate more accurate simulation results than a traditional TCP/IP network simulator that abstracts a lot away from a real-life TCP/IP implementation. Second, it lets the system default UNIX POSIX API (i.e., the standard UNIX system call interface) be provided on every node in a simulated network. Any real-life UNIX application program, either existing or to-be-developed, thus can run normally on any node in a simulated network to generate traffic. One important advantage of this property is that since an application program that is developed for simulation study is a real UNIX program, the program's simulation implementation can be its real implementation on a UNIX machine. As such, when the simulation study is finished, we can quickly implement the real system by reusing its simulation implementation.

Although the methodology proposed in [3,4] can provide the above two advantages, it has several limitations and drawbacks. To remove these problems, we enhanced the methodology, designed a new simulation engine architecture, and used these improvements to develop a new network simulator called "the NCTUns 1.0 network simulator." In the rest of the paper, we will present these enhancements as well as the features, components, design, and implementation of the NCTUns 1.0 network simulator. (For the sake of brevity, we will just call it the "NCTUns 1.0" in the rest of the paper.)

2. Related work

The predecessor of the NCTUns 1.0 is the Harvard network simulator [5], which was authored by Wang in 1999. Since its release in July 1999, as of January 1, 2002, the Harvard network simulator has been downloaded by more than 2000 universities, research institutions, industrial research laboratories, and ISPs.

As feedback about using the Harvard network simulator gradually comes back, it becomes clear that the Harvard network simulator has several limitations and drawbacks that need to be overcome and solved. Also, it is clear that some useful features and functions need to be implemented and added to it. For these reasons, Wang decided to develop the NCTUns 1.0.

In the literature, some approaches also use a real-life TCP/IP protocol stack to generate results [6–10]. However, unlike our approach, these approaches are used for emulation purposes, rather than for simulation purposes. Among these approaches, Dummynet [10] most resembles our simulator. Both Dummynet and our simulator use tunnel interfaces to use the real-life TCP/IP protocol stack on the simulation machine. However, there are some fundamental differences. Dummynet uses the real time, rather than the simulated

network's virtual time. Thus the simulated link bandwidth is a function of the simulation speed and the total load on the simulation machine. As the number of simulated links increases, the highest link bandwidth that can be simulated decreases. Moreover, in Dummynet, routing tables are associated with incoming links rather than with nodes. As such, the simulator does not know how to route packets generated by a router, as they do not come from any link.

OPNET, REAL [11], ns-2, and SSFnet [12] represent the traditional network simulation approach. In this approach, the thread-supporting event scheduler, application programs that generate network traffic, utility programs that configure, monitor, or gather statistics about a simulated network, the TCP/IP protocol implementation on hosts, the IP protocol implementation on routers, and links are all compiled together to form a single user-level program. Due to the enormous complexity, such a simulator tends to be difficult to develop and verify. In addition, a simulator constructed using this approach cannot provide UNIX POSIX API for real-life application programs to run normally to generate network traffic. Although some simulators may provide their own internal API, real-life application programs still need to be rewritten so that they can use the internal API, be compiled with the simulator successfully, and be concurrently executed with the simulator during simulation.

ENTRAPID [9] uses another approach. It uses the virtual machine concept [13] to provide multiple virtual kernels on a physical machine. Each virtual kernel is a process and simulates a node in a simulated network. The system calls issued by an application program are redirected to a virtual kernel. As such, UNIX POSIX API can be provided by ENTRAPID and real-life application programs can be run in separate address space normally. However, because the complex kernel needs to be ported to and implemented at the user-level, many involved subsystems (e.g., the file, disk I/O, process scheduling, inter-process communication (IPC), virtual memory subsystems) need to be modified extensively. As such, the porting effort is very large and the correctness of the ported system may need to be extensively verified.

3. High level architecture

The NCTUns 1.0 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. Functionally, it can be divided into eight separate components described below:

- The first component is the fully-integrated GUI environment by which a user can edit a network topology, configure the protocol modules used inside a network node, specify mobile nodes' moving paths, plot performance curves, play back animations of logged packet transfers, etc. From a network topology, the GUI program can generate a simulation job description file suite. Since the GUI program uses Internet TCP/IP sockets to communicate with other components, it can submit a job to a remote simulation machine for execution. When the simulation is finished, the simulation results and generated log files are transferred back to the GUI program. The user then can either examine logged data, plot performance curves, or play back packet transfer animations, etc.
  While a simulation is running at the remote simulation machine, the user can query or set an object's value at any time. For example, the user may query or set the routing table of a router or the switch table of a switch at any time. If the user does not want to do any query or set operation during a simulation, the user can choose to disconnect the currently running simulation so that he (she) can use the GUI program to handle other simulation cases. The user can later reconnect to a disconnected simulation at any time, whether it is still running or has finished. A user thus can submit many simulation jobs in a short period of time. This can increase simulation throughput if there are many simulation machines available to service these jobs concurrently.
- The second component is the simulation engine. A simulation engine is a user-level program. It functions like a small operating system. Through a defined API, it provides useful and basic simulation services to protocol modules

(to be described soon). Such services include virtual clock maintenance, timer management, event scheduling, variable registrations, etc. The simulation engine needs to be compiled with various protocol modules to form a single user-level program, which we call the "simulation server". When executed to service a job, the simulation server takes a simulation job description file suite as its input, runs the simulation, and generates data and packet transfer log files as its output. When a simulation server is running, because it needs to use a lot of kernel resources, no other simulation server can be running at the same time.

- The third component is various protocol modules. A protocol module is like a layer of a protocol stack. It performs a specific protocol or function. For example, the ARP protocol or a FIFO queue is implemented as a protocol module. A protocol module is composed of a set of functions. It needs to be compiled with the simulation engine to form a simulation server. Inside the simulation server, multiple protocol modules can be linked into a chain to form a protocol stack.
- The fourth component is the simulation job dispatcher, which is a user-level program. It should be executed and remain alive all the time to manage multiple simulation machines. We use it to support concurrent simulations on multiple simulation machines. The job dispatcher can operate between a large number of GUI users and a large number of simulation machines. When a user submits a simulation job to the job dispatcher, the dispatcher will select an available simulation machine to service this job. If there is no available machine at this time, the submitted job can be queued in the dispatcher as a background job. Background jobs are managed by the dispatcher. Various scheduling policies can be used to schedule their service order.
- The fifth component is the coordinator. which is a user-level program. On every machine where a simulation server program resides, a coordinator program needs to be executed and remain alive. Its task is to let the job dispatcher know whether this machine is currently busy running a simulation or not. When executed, it immediately registers itself with the dispatcher to join the dispatcher's simulation machine farm. Later on, when its status (idle or busy) changes, it will notify the dispatcher of its new status. This enables the dispatcher to choose an available machine from its machine farm to service a job.

  When the coordinator receives a job from the dispatcher, it forks (executes) a simulation server to simulate the specified network and protocols. At certain times during a simulation, the coordinator may also fork (start) or kill (end) some real-life application programs, which are specified in the job to generate traffic for the simulated network. Because the coordinator has the process IDs of these forked traffic generators, the coordinator passes these process IDs into the kernel to register these traffic generators with the kernel. From now on, all time-related system calls issued by these registered traffic generators will be performed based on the virtual time of the simulated network, rather than the real time.

  When the simulation server is running, the coordinator communicates with the job dispatcher and the GUI program on behalf of the simulation server. For example, periodically the simulation server sends the current virtual time of the simulated network to the coordinator. The coordinator then forwards this 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 an object's value (e.g., to query or set a switch's switch table). Message exchanges happening between the simulation server and the GUI program are all done via the coordinator.
- The sixth component is the modifications that need to be made to the kernel of the simulation machine so that a simulation server can correctly run on it. For example, during a simulation, the timers of TCP connections used in the simulated network need to be triggered by the virtual time rather than by the real time.
- The seventh component is various protocol daemons (programs) running at the user-level. Like the routing daemon "routed" or "gated" running on UNIX machines that exchange routing messages and set up system routing tables,

when the NCTUns 1.0 is running to simulate a network, some protocol daemons can run at the user-level to perform specific jobs. For example, the real-life routed (using the RIP routing protocol) or gated (using the OSPF routing protocol) daemons can run with the NCTUns 1.0 to set up the routing tables used by the routers in a simulated network.

- The last component is all real-life application programs running at the user-level. As stated previously, any real-life user-level application program can run on a simulated network to either generate network traffic, configure network, or monitor network traffic, etc. For example, the tcpdump program can run on a simulated network to capture packets flowing over a link and the traceroute program can run on a simulated network to find out the routing path traversed by a packet.

Fig. 1 depicts the distributed architecture of the NCTUns 1.0. It shows that, due to the nature of the distributed architecture, simulation machines can be very far away from the machines where the GUI programs run. For example, the simulation service center may be at NCTU in Taiwan while the GUI users come from many different places of the world.

When the components of the NCTUns 1.0 are run on multiple machines to carry out simulation jobs, we say that the NCTUns 1.0 is operating in the "multiple machine" mode. This mode can support remote simulations and concurrent simulations. These components can also run on the same machine to carry out simulation jobs. This mode is called the "single-machine" mode and is more suitable for a user who has only one machine. Due to the nature of the IPC design, the NCTUns 1.0 can be used for either mode without changing its program code. Only the mode parameter in its configuration file needs to be changed.

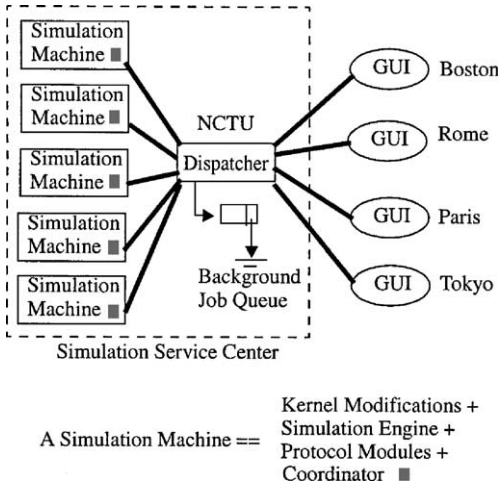

Fig. 1. The distributed architecture of the NCTUns 1.0.

4. Design and implementation

4.1. Fully-integrated GUI environment

The NCTUns 1.0 has a fully-integrated GUI environment by which a user can easily perform simulation studies. The GUI program is composed of four main components. In the following, we will present each of them.

The first component is the topology editor, which is shown in Fig. 2. The topology editor provides a convenient and intuitive way to graphically construct a network topology, specify various parameters of network devices and protocols, and specify the application programs that will be run during simulation to generate traffic.

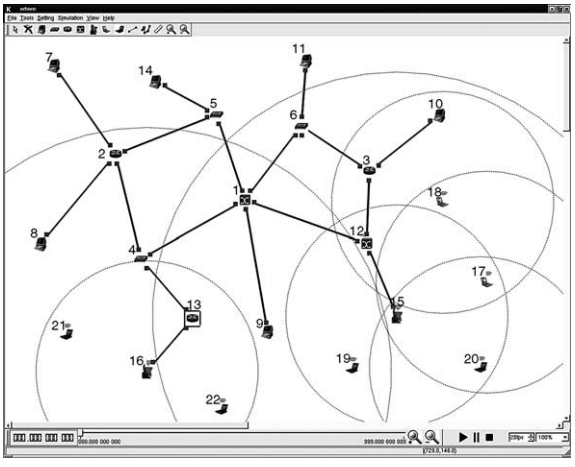

Fig. 2. The topology editor of the NCTUns 1.0 network simulator.

A constructed network can be either a fixed wired network or a mobile wireless network.

The second component is the performance monitor, which is shown in Fig. 3. The performance monitor can easily and graphically display the plots of some monitored performance metrics such as a link's utilization or a TCP connection's achieved throughput.

The third component is the packet animation player, which is shown in Fig. 4. By using the packet animation player, a logged packet transfer trace can be graphically replayed at any speed. Both wired and wireless networks are supported. The network at the top of Fig. 4 is a fixed wired network. When the packet animation player starts, packets are represented as line segments with arrows flowing smoothly on the links. The network at the bottom is a mobile ad hoc network. When the player starts, a wireless transmission is represented by two circles centered at the transmitting node. These two circles represent the transmission and interference ranges of the wireless network interface. Their display time is proportional to the packet transmission time of this wireless transfer. The packet animation player is a very useful tool because it can help a researcher to visually debug the behaviors of a protocol. It is also very useful for educational purposes.

The last component is the node editor, which is shown in Fig. 5. A node in the NCTUns 1.0 represents a network device such as a switch or an IEEE 802.11(b) wireless LAN access point. The node editor provides a convenient environment to flexibly configure the protocol modules used inside a network node. By using this tool, a user can use the mouse to graphically add, delete, or replace a protocol module with his (her) own module. As

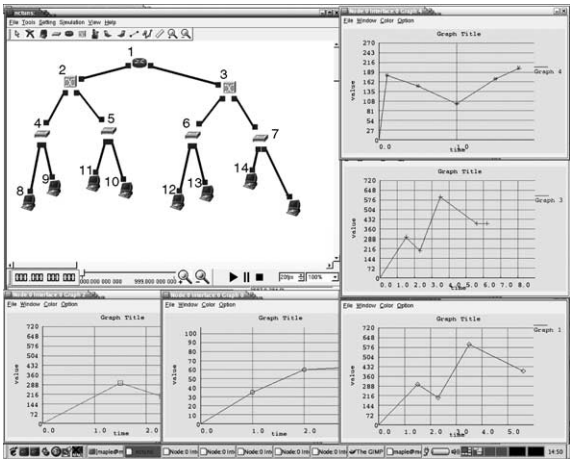

Fig. 3. The performance monitor of the NCTUns 1.0 network simulator.

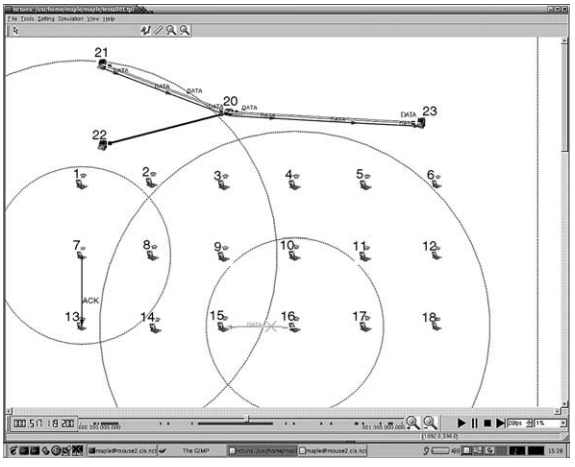

Fig. 4. The animation player of the NCTUns 1.0 network simulator.

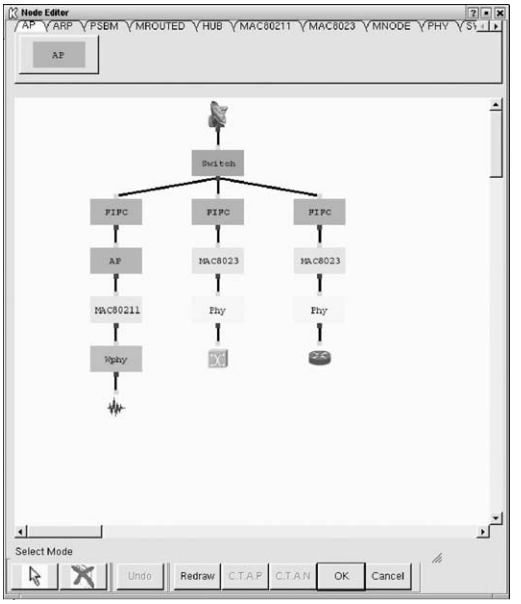

Fig. 5. The node editor of the NCTUns 1.0 network simulator.

such, the node editor enables a user to easily test the functionality and performance of a new designed protocol. Fig. 5 shows the internal protocol stacks used by a router, which in this case has three network interface ports. In Fig. 5, each square box represents a protocol module. We see that each network interface port is configured with a chain of protocol modules (i.e., a protocol stack). The protocol modules supported by the NCTUns 1.0 are classified into different categories (e.g., MAC, PHY, Packet Scheduling, etc.). They are displayed at the top of the node editor.

4.2. The enhanced simulation methodology

The NCTUns 1.0 uses an enhanced simulation methodology, which enables it to be much more powerful and useful than the Harvard network simulator. The enhancements come from the desires to support multiple subnets in a simulated network, simulate various network devices operating at different layers, simulate various protocols, simulate various types of networks, support both broadcast and unicast transfer modes for application programs, let users use the familiar real-life IP address and port number scheme to specify the network parameters of application programs, etc. In summary, the goal of the enhanced simulation methodology is to allow users to simulate any desired network and operate it in exactly the same way as they operate a physical real network.

In the following, we present the design and implementation of the enhanced simulation methodology.

4.2.1. Tunnel network interface

Tunnel network interfaces is the key facility in the used simulation methodology. A tunnel network interface, available on most UNIX machines, is a pseudo network interface that does not have a real physical network attached to it. The functions of a tunnel network interface, from the kernel's point of view, are no different from those of an Ethernet network interface. A network application program can send out its packets to its destination host through a tunnel network interface or receive packets from a tunnel network interface, just as if these packets were sent to or received from a normal Ethernet interface.

Each tunnel interface has a corresponding device special file in the/dev directory. If an application program opens a tunnel interface's special file and writes a packet into it, the packet will enter the kernel. To the kernel, the packet appears to come from a real network and just be received. From now on, the packet will go through the kernel's TCP/IP protocol stack as an Ethernet packet would do. On the other hand, if the application program reads a packet from a tunnel interface's special file, the first packet in the tunnel interface's output queue in the kernel will be dequeued and copied to the application program. To the kernel, the packet appears to have been transmitted onto a link and this pseudo transmission is no different from an Ethernet packet transmission.

4.2.2. Simulating single-hop networks

Using tunnel network interfaces, we can easily simulate the single-hop TCP/IP network depicted in Fig. 6(a), where a TCP sender application program running on host 1 is sending its TCP packets

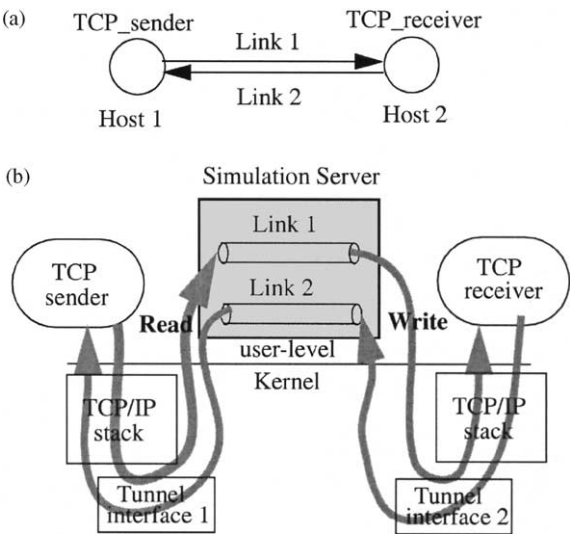

Fig. 6. (a) A TCP/IP network to be simulated. (b) By using tunnel interfaces, only the two links need to be simulated. The complicated TCP/IP protocol stack need be simulated. Instead, the real-life TCP/IP protocol stack is directly used in the simulation.

to a TCP receiver application program running on host 2. We set up the virtual simulated network by performing the following two steps. First, we configure the kernel routing table of the simulation machine so that tunnel network interface 1 is chosen as the outgoing interface for the TCP packets sent from host 1 to host 2, and tunnel network interface 2 is chosen for the TCP packets sent from host 2 to host 1. Second, for the two links to be simulated, we run a simulation server to simulate them. For the link from host i to host j ($i = 1$ or 2, $j = 3 - i$), the simulation server opens tunnel network interface i's and j's special file in/dev and then executes an endless loop until the simulated time elapses. In each step of this loop, it simulates a packet's transmission on the link from host i to host j by reading a packet from the special file of tunnel interface i, waiting the link's propagation delay time plus the packet's transmission time on the link, and then writing this packet to the special file of tunnel interface j.

After performing the above two steps, the virtual simulated network has been constructed. Fig. 6(b) depicts this simulation scheme. Since the trick of replacing a real link with a simulated link happens outside the kernel, the kernels on both hosts do not know that their packets actually are exchanged on a virtual simulated network. The TCP sender and receiver programs, which run on top of the kernels, of course do not know the fact either. As a result, all existing real-life network application programs can run on the simulated network, all existing real-life network utility programs can work on the simulated network, and the TCP/IP network protocol stack used in the simulation is the real-life working implementation, not just an abstract or a ported version of it.

Note that in this simulation methodology, the kernel of the simulation machine is shared by all nodes (hosts and routers) in a virtual simulated network. Therefore, although in Fig. 6(b) there are two TCP/IP protocol stacks depicted, actually they are the same one—the protocol stack of the single simulation machine.

4.2.3. Simulating multi-hop networks

The above simulation methodology can only simulate a network composed of two hosts that are directly connected by a full-duplex link. To simulate a multi-hop network composed of layer-1 hubs, layer-2 switches, and layer-3 routers, to allow multiple subnets to exist in a simulated network, and to let packets be routed automatically through routers as they are forwarded toward their destination nodes, we need to enhance the basic simulation methodology. In the following, we use Fig. 7 to illustrate the enhanced simulation methodology.

Suppose that we want to simulate the network depicted in Fig. 7(a), which has two subnets. The first subnet is subnet 8 (its network address is 1.0.8.X) while the second subnet is subnet 9 (its network address is 1.0.9.X). A layer-3 router (i.e., router 1) connects both of these two subnets together and forward packets between them. In subnet 9, a layer-2 switch (i.e., switch 1) connects to both router 1 and host 2 and switches packets between them. In the following, we define the schemes used in the NCTUns 1.0.

- *Interface IP address scheme:* In a simulated network, multiple subnets can exist. For each layer-3 or above network node (e.g., a host or a router), if it has multiple network interfaces, each one is simulated by a tunnel network interface. A tunnel network interface has an IP address assigned to it, just like a normal network interface does. Suppose that a tunnel interface connects to subnet A and its host number on this subnet is B, its IP address is configured as 1.0.A.B in this scheme. (In the rest of the paper, we assume that IPv4 addresses are used to construct a simulated network.) Arbitrarily chosen, 1.0.X.X represents the network address of the whole simulated network. Using the common netmask of 255.255.255.0, a simulated network can have up to 255 subnets, each having up to 255 hosts or routers residing on it. This interface IP address scheme is the same as the standard IP address scheme used in real-life networks.
  If a tunnel interface is used in a simulation, its IP address needs to be configured. We can use the UNIX ifconfig program to do this task. For example, to configure tun1, we can use the "ifconfig tun1 1.0.8.1 netmask 255.255.255.0" command. Other tunnel interfaces used in Fig. 7(a) are configured in a similar way.

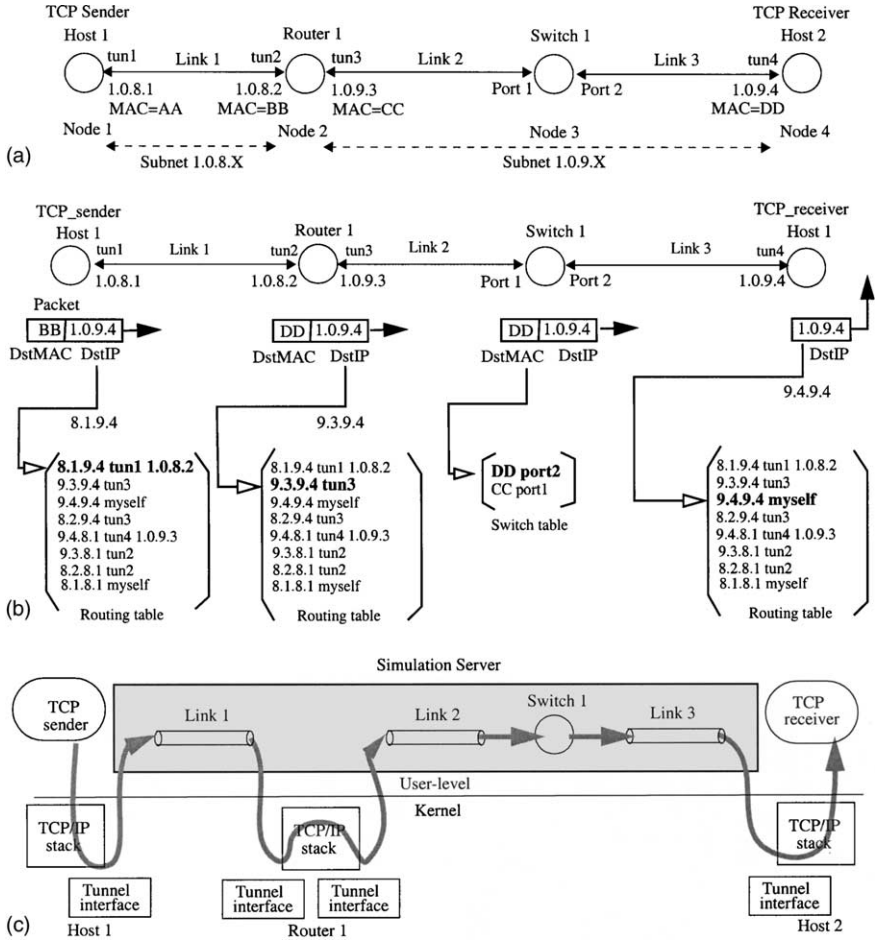

Fig. 7. (a) An example network to be simulated. (b) The automatic routing scheme is used to automatically forward packets across layer-3 routers. (c) The simulation server participates in the simulation to simulate links and switches.

In the NCTUns 1.0, a layer-1 network node (e.g., a hub) or a layer-2 network node (e.g., a switch) does not have any IP address assigned to its interface ports. This is correct as in real-life networks an IP address is used for addressing a layer-3 network interface. Note that although the familiar network mask of 255.255.255.0 is used as the network mask for a simulated network, it can be set to any valid value as well. In short, the IP address scheme used in this methodology is the same as that used in real-life networks.

Fig. 7(a) shows that tun1 is used by host 1 to connect to subnet 8, tun2 used by router 1 to connect to subnet 8, tun3 used by router 1 to connect to subnet 9, tun4 used by host 2 to connect to subnet 9, and switch 1 does not have any IP address assigned to its interface ports. We see that each tunnel interface is configured with an IP address and a MAC address. These MAC addresses can be arbitrarily chosen as long as they are different on a subnet.

- *Source-destination-pair IP address scheme:* After assigning an IP address to each tunnel interface used in a simulated network, now an application program running on a node can send packets to an application program running on a different node. Assuming that the sending node has a tunnel interface whose assigned IP address is 1.0.A.B and the receiving node has a tunnel interface

whose assigned IP address is 1.0.C.D, in this methodology the sending application program should use A.B.C.D as the destination IP address when sending packets to the receiving node.

We call such addresses the "source-destination-pair" addresses. These addresses are not used by any interface in a simulated network. Instead, they are used by sending application programs to indicate their intended destination nodes. Using the source-destination-pair address scheme enables packets to be automatically forwarded through layer-3 routers in a simulated network. The details about the automatic routing scheme will be explained later. Although using source-destination-pair addresses to specify the address parameters of application programs is unnatural to simulator users, by using the fully-integrated GUI environment, a user need not know the concept and need not use the source-destination-pair address scheme at all. In the GUI program, the user can still use 1.0.C.D as the destination address when specifying the address parameters of application programs. On the simulation machine, the coordinator will automatically translate the destination address to A.B.C.D before launching these application programs.

- *Automatic routing scheme:* To let the simulation machine's kernel automatically route a packet through many layer-3 routers in a simulated network, we can properly configure the routing entries of the simulation machine's system routing table. The automatic routing design has two main advantages. First, we can use the real-life IP protocol stack of the simulation machine's kernel to forward packets in a simulated network. Simulation results thus can be more accurate. Second, we can reuse the system default routing scheme to add, delete, or change routing entries and look up the routing table. As such, we need not waste time and effort to reimplement the same scheme in the simulator. Note that although there may be many routers in a simulated network, they all share and use the same system routing table.

  For example, in Fig. 7(b), several routing entries are added to the system routing table of the simulation machine. When host 1 wants to send packets to host 2, it uses the 8.1.9.4 source-destination-pair address to look up the routing table. The found entry is [8.1.9.4 tun1 1.0.8.2]. This entry indicates that the packet needs to be sent through tun1 and the used gateway IP address should be 1.0.8.2. The ARP module at the sending node then finds the MAC address used by 1.0.8.2 (by using the ARP request/reply protocol) and puts it (i.e., BB) in the MAC header of the packet as the destination MAC address. The MAC module at the sending node then sends out the completed MAC frame, which will then reach the interface whose assigned IP address is 1.0.8.2.

  Note that the source-destination-pair address 8.1.9.4 is used only for looking up the routing table. After the corresponding routing entry is found, the source-destination-pair address 8.1.9.4 is no longer used. The destination IP address carried in the IP header of the packet is always 1.0.9.4. It remains the same from the source node to the destination node, no matter how many routers the packet needs to traverse. When the MAC frame arrives at router 1, its MAC header is stripped off by the MAC module at router 1. At the IP layer of the simulation machine's kernel protocol stack, the 1.0.9.4 address carried in the IP header is taken out and translated to 9.3.9.4 source-destination-pair address for looking up the routing table. The reason why **9.3**.9.4 is used is because 1.0.**9.3** is one of router 1's IP addresses. Actually, because 1.0.**8.2** is also one of router 1's IP addresses, the **8.2**.9.4 source-destination-pair address can also be used. In Fig. 7(b), we see that both [9.3.9.4 tun3] and [8.2.9.4 tun3] routing entries exist in the system routing table. As such, whether 1.0.9.4 is translated to 9.3.9.4 or 8.2.9.4, the found routing entry will indicate that the destination node (i.e., host 2) is already on the same subnet as router 1 (because there is no gateway IP address associated with this entry) and the packet should be sent out via tun3 directly to 1.0.9.4. The ARP module at router 1 then finds the MAC address used by 1.0.9.4 (i.e., DD) and puts it into the MAC header of the packet. The completed MAC frame is then sent out through tun3.

When the MAC frame arrives at switch 1, its destination MAC address is taken out by switch 1 for looking up the switch table. Because the found switch entry is [DD port2], which indicates that this MAC frame should be forwarded out via port 2, this MAC frame is forwarded out without modification via port 2 of switch 1. Note that the switch is simulated by the simulation server (which is compiled and linked with the switch protocol module). Unlike a layer-3 router, which is simulated by letting packets reenter the kernel IP protocol stack, a layer-2 switch or a layer-1 hub is simulated internally inside the simulation server.

When the MAC frame arrives at host 2, its MAC header is stripped off. The destination IP address 1.0.9.4 is taken out and translated to the source-destination-pair address 9.4.9.4 before the kernel looks up the routing table. Because the first two numbers 9.4 is the same as the second two numbers 9.4 in the source-destination-pair address 9.4.9.4, the kernel knows that this packet has reached its final destination node and therefore there is no need to look up the routing table. The kernel then delivers the packet to the TCP/UDP layer for further processing.

4.3. Simulation engine

The NCTUns 1.0 is a network simulator, not a network emulator. As such, it can simulate networks with a very large number of links and nodes. Links with very high bandwidth can also be simulated. As a simulator, when simulating a network, the simulation engine needs to maintain a virtual clock for the simulated network. Simulation events are triggered and executed based on the virtual clock, rather than the real clock.

The virtual clock in the simulation engine is maintained by a counter. The time unit represented by one tick of the counter can be set to any value (e.g., one nanosecond) to simulate high speed links. The current virtual time thus is the current value of the counter times the time unit used. The simulation engine uses the discrete-event simulation method to advance its virtual clock. During simulation, the counter is continuously advanced to the timestamp of the event to be processed next.

The simulation engine needs to pass the current virtual time down into the kernel. This is required for many purposes. First, the timers of TCP connections used in the simulated network need be triggered by the virtual time rather than by the real time. Second, for those application programs launched to generate traffic in the simulated network, the system calls issued by them must be performed based on the virtual time rather than the real time. For example, if we launch a ping program in a simulated network to send out a ping request every 1 s, the sleep (1) system call issued by the ping program must be triggered by the virtual time, not the real time. Third, the in-kernel packet logging mechanism (i.e., the Berkeley-packet-filter (BPF) scheme used by tcpdump) needs to use timestamps based on the virtual time, rather than the real time, to log packets transferred in a simulated network.

The simulation engine needs to pass the current virtual time to the kernel in a low-cost and fine-grain way. The simulation engine can pass the current virtual time into the kernel by periodically making a system call. (For example, the simulation engine can make the system call once every 1 ms in virtual time.) However, the cost of this approach will be too high when we want the virtual time maintained in the kernel to be as precise as that maintained in the simulation engine. For example, the in-kernel packet logging mechanism needs a microsecond-resolution clock to generate timestamps. To solve this problem, the simulation engine uses a memory-mapping technique. The simulation engine maps the memory location that stores the current virtual time in the simulation engine to a memory location in the kernel. As such, at any time the virtual time in the kernel is as precise as that maintained in the simulation engine without any system call overhead.

4.4. Protocol modules

Protocol modules are compiled and linked with the simulation engine to simulate layer-2 and below devices, protocols, and transmission medium. Although the automatic routing scheme enables the simulation machine's kernel to use its layer-3 and above TCP/IP protocol stack to forward

packets, layer-2 and below devices, protocols, and transmission medium are not simulated when this scheme is used. As such, the simulation server (i.e., the simulation engine plus protocol modules) needs to simulate transmission medium, all layer-2 and below protocols, and devices. For example, Fig. 7(c) shows that, to simulate the network depicted in Fig. 7(a), the simulation server needs to simulate link 1, link 2, link 3, and switch 1. (It does not need to simulate host 1, host 2, and router 1 because they are "simulated" by using the automatic routing scheme.)

Layer-2 and below devices, protocols and transmission medium are simulated as protocol modules. Several protocol modules may be chained together to form a protocol stack. A layer-3 interface (i.e., a tunnel interface) uses such a protocol stack to simulate its layer-2 and below processing. For example, a layer-3 interface normally has the following protocol modules. First, an ARP module is required to find the MAC address used by an IP address (i.e., the destination IP address of an outbound packet). Second, a packet scheduling and buffer management (PSBM) module is required for storing and scheduling outbound packets. (The simplest one is a FIFO queue.) Third, a Medium Access Control (e.g., 802.3 or 802.11) module is required for controlling when to send a packet onto the link. Lastly, a physical layer (PHY) module is required to simulate the characteristics of the transmission medium (e.g., delay, bandwidth, Bit-Error-Rate, etc.). These modules are chained together. When a layer-3 interface sends out a packet onto a link, the packet will be passed down module-by-module to the PHY module. In the other direction, when the PHY module of a layer-3 interface receives a packet, the packet will be passed up module-by-module to the layer-3 interface if a lower-layer module does not discard it (e.g., to simulate bit errors).

Although by default each layer-3 interface (i.e., tunnel interface) has an output queue (FIFO) associated with it inside the kernel, the NCTUns 1.0 does not use it. Instead, whenever the kernel enqueues a packet into a tunnel interface's output queue, a notification event is immediately passed to the simulation server, which enables the simulation server to immediately dequeues the packet and reads it out from the kernel. This operation takes no time in virtual time because the simulator's virtual clock is stopped during this period.

The simulation server then passes the packet to the ARP module associated with this tunnel interface, which in turn passes the packet down to the PSBM module below it. At the PSBM module, any sophisticated PSBM scheme can be used. This design enables a host or a router to use various sophisticated PSBM scheme for its ports. For example, a router's first port can use a PSBM module that implements the Round-Robin scheme while its second port can use a PSBM module that implements the FIFO scheme. Another advantage of this design is that a PSBM module developed for layer-2 switches can be readily used for layer-3 routers. No extra time and effort are needed.

As an example, Fig. 8(b) shows how the simulation server simulates the network depicted in Fig. 8(a). Suppose that the TCP sender sends a packet to the TCP receiver. On host 1, the packet will pass through the TCP/IP protocol stack and be enqueued into the output queue of tun1. The simulation server will immediately dequeue it and read it out from the kernel. The simulation server then delivers it to the protocol stack created for tun1. The packet then passes the ARP module, the PSBM module, the 802.3 module, and finally reaches the PHY module of this protocol stack.

Before being delivered to the other end of the link, the packet needs to wait a certain amount of time to simulate the delay of link 1 and its packet transmission time on link 1. While waiting, it is stored as a timeout event in the simulation engine's event heap. When the packet's timer expires, the simulation server then delivers the packet to the protocol stack created for tun2 by moving the packet to the PHY module of the second protocol stack. The packet then is passed up and reaches the 802.3 module. At the 802.3 module, the packet's destination MAC address is checked against tun2's MAC address to see whether this packet should be accepted or discarded. If the packet should be accepted, it is passed to the PSBM module. The PSBM module simply passes the packet to the ARP module because its PSBM functions are for outbound packets, not for inbound packets. When

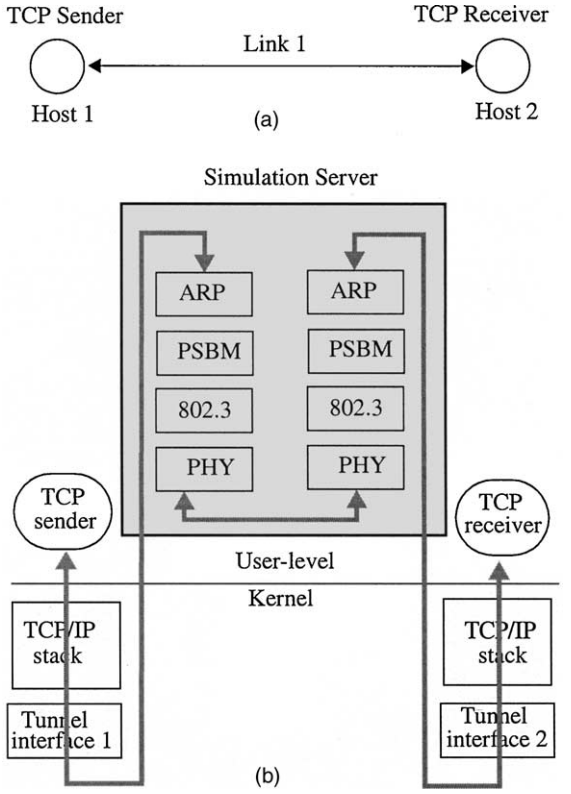

Fig. 8. (a) A network to be simulated. (b) Using the simulation server to simulate the network depicted in (a).

the ARP module receives the packet, because ARP protocol is for outbound packets only, it simply writes the packet into the kernel. The packet then passes through the TCP/IP protocol stack and finally reaches the TCP receiver.

To enable the user-level simulation server to quickly detect that the kernel has enqueued a packet into a tunnel interface's output queue, a memory-mapping technique similar to that used for passing the current virtual time down into the kernel is used. In the kernel, a bit-map is used to record the empty or non-empty status of every tunnel interface's output queue. The memory location that stores this bit-map in the kernel is mapped to a memory location in the simulation server. By using this technique, the simulation server can immediately detect that a packet has been enqueued into a tunnel interface's output queue without any system call overhead.

4.5. Kernel modifications

Some parts of the simulation machine's kernel need to be modified. In the following, we present some important kernel modifications.

4.5.1. IP address translation

The use of source-destination-pair IP address scheme enables the kernel to automatically forward a packet toward its destination node. However, when a simulator user specifies an application program's destination IP address parameter, he (she) should be able to use the normal IP address scheme for this task. For example, in Fig. 7(a), the destination IP address parameter given to the TCP sender should be 1.0.9.4, rather than 8.1.9.4. The internally used and unnatural source-destination-pair IP address scheme should be hidden from the user. The user need not know how we use the source-destination-pair address to automatically route packets.

Internally, the kernel needs to perform the address translation on each node that is on the path from the source node to the destination node, and use the translated IP address to look up the routing table. However, to translate the address, the kernel first needs to know the identity of the current node. That is, when a packet is forwarded to and enters node i, the kernel should know that the identity of the current node is i. After obtaining this information, the kernel can look up the interface table associated with node i and pick up an interface IP address to perform the translation. (The kernel keeps an interface table for each node, which records the IP addresses used by this node.) As an example, in Fig. 7(a), when a packet is forwarded to node 2, the kernel can pick up an IP address (say 1.0.9.3) from node 2's interface table and translate 1.0.9.4 to 9.3.9.4 before looking up the routing table. (Note: the kernel could pick up 1.0.8.2 and translate 1.0.9.4 to 8.2.9.4 as well. The reason has been explained in Section 4.2.3.)

To pass the current node identity to the kernel when a packet arrives at a node, the simulation server, after simulating the packet's transmission on a link, can put the identity of the destination node of the link (e.g., i) into the packet's header before writing it into the kernel.

Although the above method seems to work successfully for all nodes on the path, actually it cannot work successfully for the source node. For a non-source node, before a packet enters it, the packet must be transmitted on a link. As such, the simulation server knows the identity of the destination node of this link. However, for the source node, since the packet does not come from any link, this information is unavailable and thus cannot be provided by the simulation server.

We solve this problem by explicitly telling the kernel the current node identity when an application program is launched. Since in the NCTUns 1.0 every application program is launched by the coordinator, the coordinator is designed to issue a system call to the kernel before launching an application program. The system call passes the identity of the node on which the application program is intended to run into the kernel. The kernel then stores this information in one of its variables. Very soon when the application program is launched, the kernel will store this information in the control block of this launched process. Form now on, every packet generated by this application program can carry this information in its header when it is sent down from the socket layer to the IP layer. This solves the address translation problem on the source node.

4.5.2. Port number translation

An inherent problem with the proposed simulation methodology is that application programs cannot bind to the same port in a simulated network, even though they are running on different nodes in the simulated network. The reason is that, since these application programs are running on a single-machine (i.e., the simulation machine), they cannot choose the same port to bind. In real-life networks, however, this is possible and should be allowed. For example, in a network, there may be a Web server binding to port 80 on every host and a RIP routing daemon binding to port 520 on every router.

From an application program's viewpoint, it does not matter which port to use as long as it can use the port to communicate with its partners. As such, when multiple application programs running on different nodes want to bind to the same port in a simulated network, a network simulator user can solve this problem by letting them choose different port numbers to bind. Although this solution works and does not affect the simulation result, it makes a simulated network unnatural to the simulator user, which should be avoided. A better solution would be that these application programs are still allowed to bind to the same port when they are launched; however, the kernel internally translates the port number used by them to different port numbers to avoid port number collisions.

To achieve this goal, the kernel maintains a bitmap to record which port numbers have been used and which have not been used. During a simulation, suppose that an application program (say A) running on node i wants to bind to port number j, the kernel will find an unused port number (say k) and instead let application program A bind to port number k. The kernel then creates an association (nodeID $= i$, real_port_num $= j$, remapped_port_num $= k$) and inserts it into a hash table.

With this arrangement, if an application program (say B) wants to send packets to application program A, application program B can use the port number originally used by application program A (i.e., j) as the destination port number. Application program B need not know the port number translation details. The simulated network looks like a real network to it.

The port number translation process occurs at the destination node(s), not at the source node. When application program B sends a packet to application program A, before the packet reaches the destination node, the destination port number carried in the packet remains j, not k. Only after the packet reaches the destination node is its destination port number translated to k. Finding k is achieved by searching the hash table using the key pair (i, j), where j is readily available from the packet header. As for the value of i (the current node identity), the kernel can obtain this information by using the method described in Section 4.5.1.

Translating the port number at the destination node(s), not at the source node, has two advantages. The first advantage is that it supports broadcast transfers on a subnet. If the translation

is performed at the source node, only unicast transfers can be supported. Broadcasting a packet on a subnet to multiple application programs that bind to the same port but run on different machines (e.g., the routing daemons case) will be impossible. At present, we have not investigated how to support multicast transfers. The second advantage is that we can use the tcpdump program to correctly filter and capture packets in a simulated network. The tcpdump program can use port numbers to filter and capture packets. When a user wants to capture the packets sent from application program A to B, naturally he (she) will set the filtering destination port number to j. If we translate the port number at the source node, the destination port number carried in the packet will be k when it is traversing the network. This will make the tcpdump program unable to capture this packet.

4.5.3. Process scheduling

We modified the default UNIX process scheduler so that the processes of the simulation server and all launched traffic generators can be scheduled in a controlled way. The default UNIX process scheduler uses a priority-based dynamic scheme to schedule processes. As such, the order in which the simulation server and traffic generator processes are scheduled cannot be precisely controlled. Also, the CPU cycles allocated to each of these processes cannot be guaranteed. This may result in a potential problem. For example, after getting the control of CPU, the simulation server may use the CPU too long before releasing it to traffic generators. Because the simulation server is responsible for advancing the virtual clock while it is executing, if it monopolizes the CPU too long, no network traffic can be generated during this long period of time, which should not occur. To avoid this potential problem, we modified the default UNIX process scheduler so that the simulation server and all traffic generator processes are explicitly scheduled according to the timestamp order of their events.

4.6. System functions

In addition to simulating network devices and protocols, to be a useful software, the NCTUns 1.0 provides many useful system functions. In the following, we present two of them.

4.6.1. Per-node command console shell

For each node in a simulated network, we provide a command console. A GUI user can easily invoke a node's command console by right-clicking the node's icon in the topology editor. Immediately a terminal window (like the X terminal window) will appear and automatically log into the (possibly remote) simulation machine. On the simulation machine, a shell program is then executed to process the real-life UNIX commands that may be typed in by the GUI user.

The command console is a very useful feature. During a simulation, in a node's command console, a user can launch application programs or execute UNIX commands at run time, just like he (she) is operating in a real-life network node's command console. For example, a user can run the "netstat" command to get the packet transfer statistics of an interface. The user can run the "traceroute" command to see the routing path between any pair of nodes in the simulated network. This is useful for quickly checking the routing paths generated by routing daemons. The user can also run the "tcpdump" command to monitor the packets flowing on an interface. Actually, any real-life command can be executed in the command console. The user can immediately get the output of these commands without waiting until the simulation is finished.

To make a command console totally natural to the user, we modified the system default shell program so that the user will not see anything inconsistent. The modification handles interface name conversion and filtering. On a real-life UNIX machine, a user may execute the "ifconfig" command to check the settings of an (or all) interface(s). The output is useful as it includes the name assigned to the interfaces. (For example, the first Intel EtherExpress Ethernet interface is assigned the name fxp0, the second assigned the name fxp1, etc.) Knowing an interface's name is important as some utility programs need this information. For example, if we run the tcpdump program to monitor the packets flowing on an interface, we need to know the interface's name

and give it as a parameter to the tcpdump program.

If the default shell program is not modified, when the user uses the "ifconfig -a" command to see all interfaces used by this node, he (she) will see all the tunnel interfaces used by the simulation machine and will not know which tunnel interfaces are internally used for the interfaces of this node. As such, the shell program needs to perform two tasks. The first task is to filter out unrelated output and the second task is to convert interface names between tunXXX and fxpXXX, where XXX represents a number.

For example, suppose that 256 tunnel interfaces (tun0, tun1, ..., tun255) are used by the simulation machine to simulate a network, and among them, tun1, tun8 and tun9 are internally used to simulate the three interfaces used by a node in the simulated network. Suppose that in the topology editor, these three interfaces are given the names fxp0, fxp1, and fxp2, respectively. Now in the node's command console, if the user executes the ifconfig -a command, what he (she) should see is the settings about fxp0, fxp1, and fxp2, rather than the settings about tun0, tun1, ..., and tun255. The shell program needs to internally convert tun1 to fxp0, tun8 to fxp1, and tun9 to fxp2 before displaying the command's output. It also needs to filter out the settings of all other tunnel interfaces before displaying the output. To the user, the names of the interfaces used by this node are fxp0, fxp1, and fxp2. He (she) should be able to use any of these interface names (fxpXXX) as a parameter for any real-life command or program.

To achieve this goal, the interface name conversion and filtering operations must be performed for both the input and output of the shell program. For example, after the user finds that the node has three interfaces named fxp0, fxp1, and fxp2, he (she) may decide to execute the tcpdump program to monitor the packets flowing on fxp2. (The exact command is "tcpdump -i fxp2.") Before launching the tcpdump command, the shell program needs to intercept this command string and convert fxp2 back to tun9 so that the internally-launched tcpdump command will be "tcpdump -i tun9" rather than "tcpdump -i fxp2."

To intercept both the input and output of the shell program, we fork a process and insert it between the shell process and the system terminal device driver. This process acts as a relaying process. All input to and output from the shell process must be relayed by this process. As such, it has a chance to perform its tasks. This process actually performs more tasks than those described here. This is because a command string may contain the shell I/O redirection (i.e., >) and pipe (i.e., |) operators. The interface name conversion and filtering operations must still be handled properly in such cases.

The command console shell needs to perform two other tasks, which are also performed by the coordinator. First, before launching an application program, the shell needs to pass the current node identity into the kernel. (The reason is explained in Section 4.5.1.) Second, after launching an application program, the shell needs to register the forked process with the kernel. (The reason is explained in Section 4.5.)

4.6.2. Tcpdump packet filtering and capturing tool

The tcpdump program is a packet filtering and capturing tool. It is a user-level program that can pass filtering rules to the kernel and display captured packets. Packet filtering operations are actually performed by the BPF module in the kernel. When a packet is sent or received at an interface, the device driver of the interface passes the packet to the BPF module for evaluation. If the BPF module decides to accept this packet, it will associate a timestamp with the packet. The module gives each captured incoming packet a timestamp to record when it is received by the interface. The module also gives each captured outgoing packet a timestamp to record when it is transmitted onto a link.

The tcpdump program operates on an interface. Since from the kernel's viewpoint, a tunnel interface is no different from a real interface, the tcpdump program should be able to work correctly to capture packets flowing on a node's interface in a simulated network. In the current design, however, some modifications are needed to let the tcpdump program generate correct output.

In Section 4.4, we show that in the current design, the packets that should be sent through a

tunnel interface are no longer queued in the output queue of the tunnel interface. Instead, they are immediately dequeued by the simulation server as soon as they are enqueued into the tunnel interface's output queue. The only place where they may be queued (and delayed) is in the PSBM module associated with this tunnel interface, which is in the simulation server. This causes a problem as now the timestamps given by the tunnel interface's device driver to these packets are incorrect. 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 (or enters the PSBM module, they are the same.), rather than when it is transmitted to a link.

To solve this problem, we disabled the part of the tunnel interface device driver that is responsible for passing each outgoing and incoming packet to the BPF module. We also developed a tcpdump module and insert it between the MAC and PHY modules. This tcpdump module cannot hold any packet. When receiving a packet from the MAC module (if it is an outbound packet) or from the PHY module (if it is an inbound packet), the tcpdump module makes a copy of the packet, gives it a special tag, and associates it with the current timestamp. The tcpdump module then writes the copy into the kernel through the tunnel interface that the user-level tcpdump program is currently operating on. The tunnel interface's device driver, when seeing this special tag, passes the packet to the BPF module for evaluation. If the BPF module decides to accept this packet, it then passes this packet to the user-level tcpdump program.

As an example, suppose that during a simulation a user wants to monitor the traffic of a node's interface and this interface is internally simulated by tun4. In the node's command console, the user will execute the user-level tcpdump program and the command console shell will internally translate this command string to "tcpdump -i tun4." From now on, a user-level tcpdump program is running and monitoring packets on tun4. At the same time, the shell also asks the simulation server to insert a tcpdump module between the MAC and PHY modules that are associated with tun4. From now on, when receiving a packet (either incoming or going), the inserted tcpdump module will make a copy of this packet, give it a special tag, and associate it with the current timestamp. The module will then write it into the special file of tun4 (i.e., /dev/tun4). After entering tun4, due to the special tag, the copy of the packet will be passed to the BPF module for evaluation. If accepted, the copy will be received by tcpdump program operating on tun4 at the user-level.

The above design has two advantages. First, we can fully exploit the power of the tcpdump program. Any feature of the real-life tcpdump program can be used. Second, we reuse the user-level tcpdump program and in-kernel BPF module code to the maximum extent. They need not be modified at all. We only need to disable a very small part of the tunnel device driver code and write a very simple tcpdump module.

5. Scalability issues

Because in our scheme a single UNIX machine is used to simulate a whole network (including nodes' protocol stacks, traffic generators, etc.), the scalability of the simulator is a concern. In the following, we discuss several scalability issues.

5.1. Number of nodes

Because our scheme simulates multiple routers and hosts by letting packets re-enter the simulation machine's kernel, there is no limitation on the maximum number of routers and hosts that can be simulated in a network. For hubs and switches, since they are simulated in the simulation server, there is no limitation on them either.

5.2. Number of interfaces

In our scheme, because each layer-3 interface uses a tunnel interface, the maximum number of layer-3 interfaces that can be simulated is limited by the maximum number of tunnel interfaces that a BSD UNIX system can support, which currently

is 256. (This limitation is caused by UNIX using an 8-bit integer as a device's identity.) Since this problem can be easily solved, (for example, we can clone tunnel interfaces, give the cloned interfaces different names, and use them in the same way as we use the original tunnel interface.), there is no limitation on the maximum number of layer-3 interfaces that can be simulated in a network. For layer-1 and layer-2 interfaces (used in hubs and switches, respectively), since they are simulated in the simulation server, there is no limitation on them either.

5.3. Number of routing entries

In our scheme, the kernel routing table needs to store many source-destination-pair IP addresses so that packets can be automatically forwarded across routers by the kernel. Since the kernel routing table is used only by routers, if a simulated network has only one subnet (and thus has no router), the kernel routing table need not be used and can be empty.

The NCTUns 1.0 supports the "subnet" concept. Therefore, the more efficient subnet-routing scheme can be used instead of the less-efficient host-routing scheme. For example, for the network depicted in Fig. 7(a), instead of storing the two routing entries [8.1.9.3 tun1 1.0.8.2] and [8.1.9.4 tun1 1.0.8.2] in the kernel routing table, we can store only one routing entry [8.1.9 tun1 1.0.8.2] in the table. Because the subnet-routing scheme can be used, suppose that in a simulated network there are S different subnets and on average there are H hosts residing on a subnet, the number of source-destination-pair routing entries that need to be stored in the kernel routing table would be about $S * H * S$. As an example, suppose that S is 30 and H is 20, the number of required routing entries would be about 18,000.

We have tested several network configurations that need to store over 60,000 routing entries in the kernel routing table. We found that because the BSD UNIX systems use the radix tree [14] to efficiently store and look up routing entries, using a large number of routing entries in a simulation is feasible and does not slow down simulation speed much.

5.4. Number of application programs

Since application programs running on a UNIX simulation machine are all real independent programs, the simulation machine's physical memory requirement would be proportional to the number of application programs running on top of it. Although, at first glance, this requirement may seem severe and may greatly limit the maximum number of application programs that can simultaneously run on a UNIX machine, we found that the virtual memory mechanism provided on a UNIX machine together with the "working set" property of a running program greatly alleviate the problem. The reason is that, when an application program is running, only a small portion of its code related to network processing will need to be present in the physical memory. In addition, because UNIX machines support the uses of shared libraries and shared virtual memory pages, the required memory space for running the same application program multiple times can be greatly reduced.

6. Simulator performance

Here we report the simulation speed of the NCTUns 1.0 under several network and traffic configurations. The used machine for performance testing is an IBM A31 notebook computer equipped with a 1.6 GHz Pentium processor and 128 MB RAM.

6.1. Variable CBR UDP on a single-hop network case

In this test suite, the network topology is a single-hop network in which a sending host and a receiving host are connected together by a link. The bandwidth of the link is set to 10 Mbps and the delay is set to 10 ms, in both directions. The traffic generated is a one way constant-bit-rate (CBR) UDP packet stream. Each UDP packet size is set to 576 bytes.

We varied the packet inter-arrival time of the CBR packet stream to see how the simulator's speed will change when it needs to process more events in each simulated second. The packet inter-

arrival time is the time interval between two successive packet transmissions. The tested intervals are 0.001, 0.005, 0.025, 0.125, 0.625, and 3.125 s, respectively.

The performance metric reported is the ratio of the simulated seconds to the elapsed seconds for running the simulation. In all of our tests, the simulated seconds is set to 999 s. A simulation case with a higher ratio means that it can be finished more quickly than a case with a lower ratio. A simulation case with a ratio of 1 means that it needs the same amount of time in real time to finish simulating the amount of time that it wants to simulate.

Fig. 9 shows the ratio vs. CBR packet inter-arrival time performance plot. We see that due to the discrete-event simulation engine design, a simulation case can be finished very quickly if it does not have many events to process (i.e., traffic) per simulated second. We also see that the ratio (2.5) of the high-load case (0.001 s) is still greater than 1. This means that the simulator can still run 2.5 times faster than the real world under this load on the testing machine.

6.2. Multiple greedy TCP connections on a single-hop network case

Since during a simulation the applications that are run to generate traffic are real-world programs, we are interested to see how the simulator's speed will change if more applications need to be run at the same time. To perform this test, we used the network configuration depicted in Fig. 10.

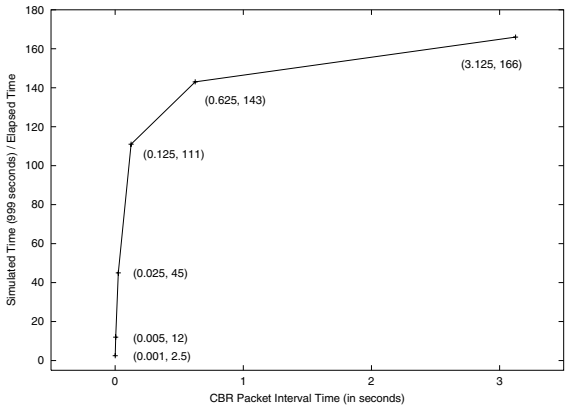

Fig. 9. The simulation performance under various CBR UDP traffic load. (A higher ratio means a better performance.)

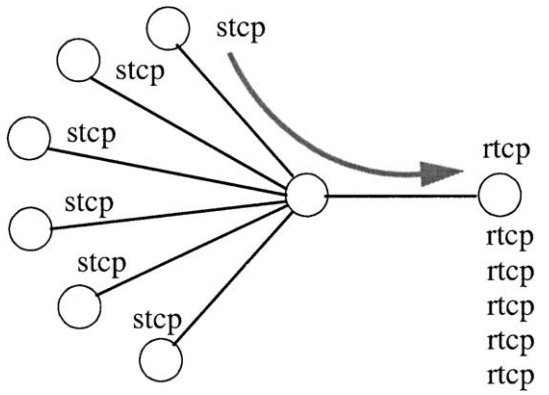

Fig. 10. The multi-source-node network topology used to test whether the simulation performance will degrade when more applications need to be run to generate traffic.

In this configuration, there are six source nodes, one destination node, and a bottleneck router node. The bandwidth and delay of all links are set to 10 Mbps and 10 ms, respectively. The maximum packet queue length of the FIFO queue in the bottleneck router is the default 50 packets. Between a pair of a source and the destination node, we can set up a greedy TCP connection by running the stcp program on the source node and the rtcp program on the destination node. The length of TCP data packets is 1500 bytes, which is Ethernet's MTU.

In this test, we varied the number of greedy TCP connections that are set up to compete for the bottleneck link's bandwidth. The numbers tested are 1, 2, 3, 4, 5, and 6, respectively. In all of these cases, the bottleneck link's bandwidth is always 100% utilized.

Fig. 11 shows that the simulator's speed does not degrade as more applications (stcp and rtcp) are run to generate traffic. This phenomenon can be explained because the number of events that need to be processed per simulated second remains about the same. No matter how many greedy TCP connections (their stcp and rtcp programs) are launched to send and receive their data, the aggregate amount of data that can be pumped into the bottleneck link or received from the bottleneck

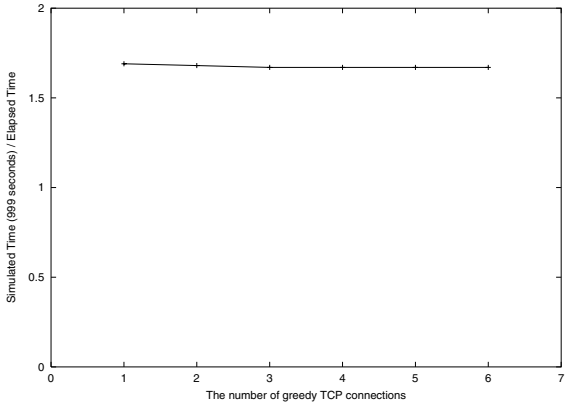

Fig. 11. The simulation performance remains about the same for different numbers of application programs running as traffic generators.

link per simulated second is always fixed to the 10 Mbps rate. As such, the stcp and rtcp programs of the six greedy TCP connections will take turns to run, and their aggregate context-switching rate is about the same as in the single greedy TCP connection case.

6.3. Fixed CBR UDP on multi-hop networks case

For a discrete-event simulation engine, the more events it needs to process in each simulated second, the slower its simulation speed will be. In Section 6.1, we shows that on a single-hop network if we decrease the CBR packet stream's packet inter-arrival time, the simulator will become slower. Here we show that, given a fixed CBR packet inter-arrival time, if packets need to go through more hops, the simulator will also become slower.

The network configurations used are multi-hop chain networks shown in Fig. 12. The node on the left hand side is the source host while the node on the right hand side is the destination host. The bandwidth and delay of all links are set to 10 Mbps and 10 ms, respectively. The packet inter-arrival time of the CBR UDP packet stream is set to 0.025 s and the packet length of each UDP packet is set to 576 bytes.

We performed two suites of performance tests. In the first suite, all of the intermediate forwarding

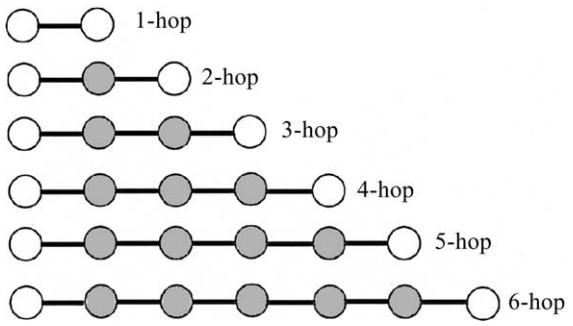

Fig. 12. The multi-hop networks used to test the simulation performance. The source host is on the left while the destination host is on the right. Intermediate forwarding nodes may be routers or switches.

nodes are routers. In the second suite, all of them are switches. We made these two cases to observe how much more costly a router is in forwarding a packet than a switch.

Fig. 13 shows the performances of these two suites. We see that as the number of hops increases, the simulation performance decreases. This phenomenon is reasonable as in such a case, the number of events that need to processed in each simulated second increases.

We also see that a router is more costly than a switch in forwarding a packet. This phenomenon can be explained as follows. When simulating a router's forwarding a packet, the simulation en-

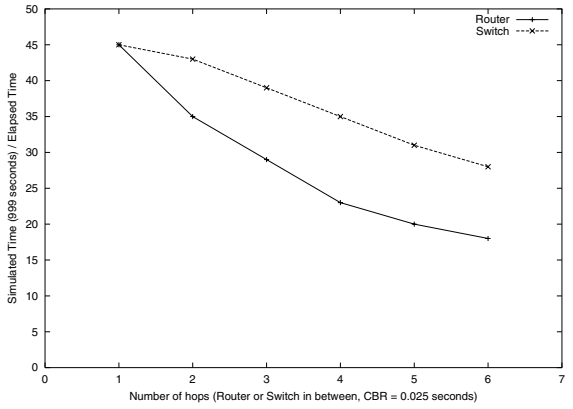

Fig. 13. The simulation performance decreases as the number of hops increases. The cost of forwarding a packet by a switch is less than that of forwarding a packet by a router.

gine needs to read a packet out of the kernel and then write it into the kernel. However, to simulate a switch's forwarding a packet, the forwarding operation can be simulated totally inside the simulation engine without issuing any read or write system call to the kernel. Since issuing system calls are costly, switches can forward packets more efficiently than routers in a simulation.

6.4. Discussions

For the following reasons, the NCTUns 1.0's speed may be slower than that of a traditional network simulator such as ns-2. First, real-life protocol stacks are executed rather than their abstractions. Second, real-life application programs are executed to generate traffic. Third, real data payload are carried in each transmitted packet and thus they need to be copied. Fourth, since packets need to be read out of the kernel and then written into the kernel, a lot of system calls need to be made.

The reported results show that the performance of the NCTUns 1.0 in its current form is still satisfactory. When the network topology is not large and the traffic load is not high, its simulation speed is faster than the real world. Currently, we are working to further improve its performance by first identifying its performance bottleneck and then using more efficient data structures and algorithms for its execution.

7. Simulation result validations

The results generated by a simulator need to be carefully validated and shown to be correct before it can be trusted. In the following, we explain how we validated simulation results.

For UDP traffic cases, our simulation results show that each packet is transmitted (and received) exactly at the specified times. For example, if a CBR UDP packet stream has a packet inter-arrival time of 0.001 s, our results show that the first packet is transmitted at 0 s, the second packet is transmitted at 0.001 s, and the third packet is transmitted at 0.002 s, etc. Actually, we found that transmitting a packet stream with any packet inter-arrival time distribution is accurately simulated. This is because the simulation engine can just advance its virtual clock to the timestamps of these transmitting (and receiving) events.

For TCP traffic cases, our simulation results also show that each packet is transmitted or received under correct TCP error and congestion control. We dumped all TCP timer timeout events (e.g., delay-ack and retransmission timers) and used the tcpdump packet trace to perform correlation checks across TCP timer events and packet transfers. The checks confirm that the TCP protocol is correctly simulated during a simulation. Actually, this is a natural result as the NCTUns 1.0 uses the in-kernel real-life TCP/IP protocol stack to generate simulation results.

To confirm that the simulator can correctly simulate a link's bandwidth and delay, we also have performed extensive validation tests covering various network configurations. All of these results can be explained and shown to be correct. Due to space limitation, we do not present these cases here.

8. Discussions and limitations

Since only a single UNIX machine (with its own protocol stack) is used to simulate multiple nodes, the NCTUns 1.0 has a limitation that it allows only one version of TCP/IP protocol stack in a simulated network. Studying interactions between different TCP versions (e.g., TCP tahoe and TCP reno) or between different TCP implementations (e.g., FreeBSD and Linux) thus cannot be done by using our simulator as is. One way to overcome this limitation is to use a distributed simulation approach. In such a distributed approach, a UNIX machine with a particular protocol stack can be used to simulate nodes using the same stack, while other UNIX machines with different stacks may be used to simulate nodes using different stacks.

Currently, the FreeBSD platform provides a user-level "sysctl -a or -w" command to view or change various parameters used by the in-kernel TCP/IP protocol stack. For example, a user can use this command to change the size of a socket send and receiver buffer, the TCP delay ACK time,

whether to perform TCP delay ACK mechanism, whether to use TCP newreno version, etc. Although right now this command can change a large number (93) of protocol parameters or options, a user may still need to change the kernel source code and recompile the kernel if his (her) changes cannot be performed by this command. Changing the kernel source code, however, may not be comfortable to all users.

When installing the NCTUns 1.0, the user needs to have the root privilege to be able to recompile the kernel. This may be a problem for some users who do not have their own computers. Since recompiling the kernel may not be comfortable to all users, the NCTUns 1.0 package provides an installation script to perform this task automatically without human intervention.

Since the NCTUns 1.0 uses real-life protocol and application implementations to "simulate" a network and its traffic, its generated results may vary from one platform to another platform (e.g., from FreeBSD to Linux) or from one release to another release (e.g., from FreeBSD 2.8 to FreeBSD 4.6), although they are all correct. Thus, when reporting or comparing simulation results generated by the NCTUns 1.0, a user should report the used platform and release version as well.

9. Ongoing work

Aimed to be a production-level and high-quality software, the NCTUns 1.0 still has many places to improve in its future versions. For example, its simulation speed needs to be improved and many useful system functions (such as "print") need to be added. We are working on these to-do items.

10. Conclusions

In this paper, we present the internal design and implementation of the NCTUns 1.0 network simulator. Based on an enhanced simulation methodology and a new simulation engine architecture, the NCTUns 1.0 network simulator provides much better functionality and performance than its predecessor—the Harvard network simulator. Its distributed and open-system architecture design supports remote simulations and concurrent simulations, and allows new protocol modules to be easily added to its simulation engine. With its fully-integrated GUI environment, non-real-life internal designs and implementations are totally hidden from the user. A user, when using the GUI environment to operate a simulated network, will feel like he (she) is operating a real network.

Due to its unique advantages, the NCTUns 1.0 network simulator was selected as a research demonstration at ACM MobiCom'02 international conference, held in Atlanta, USA, from 09/23/2002 to 09/28/2002. The NCTUns 1.0 network simulator has been released to the networking community on 11/01/2002 and its web site is set up at http://NSL.csie.nctu.edu.tw/nctuns.html.

Acknowledgements

The authors are very grateful to the three anonymous reviewers for their detailed and valuable comments, which make this paper much better than its original version. This research was supported in part by MOE program for promoting Academic Excellence of Universities under the grant number 89-E-FA04-1-4 and 91-E-FA06-4-4, the Lee and MTI Center for Networking Research, NCTU, and the Institute of Applied Science and Engineering Research, Academia Sinica, Taiwan.

References

[1] OPNET Inc. Available from <http://www.opnet.com>.
[2] S. McCanne, S. Floyd, ns-LBNL Network Simulator. Available from <http://www-nrg.ee.lbl.gov/ns/>.
[3] S.Y. Wang, H.T. Kung, A simple methodology for constructing extensible and high-fidelity TCP/IP network simulators, IEEE INFOCOM'99, New York, USA, March 21–25, 1999.
[4] S.Y. Wang, H.T. Kung, A new methodology for easily constructing extensible and high-fidelity TCP/IP network simulators, Computer Networks 40 (2) (2002) 257–278.
[5] Harvard TCP/IP network simulator 1.0. Available from <http://www.eecs.harvard.edu/networking/simulator.html>.
[6] K. Fall, Network emulation in the Vint/NS simulator, ISCC99, July 1999.
[7] Nist net. Available from <http://snad.ncsl.nist.gov/itg/nistnet>.

[8] J.S. Ahn, P. Danzig, Z. Liu, L. Yan, Evaluation of TCP Vegas: emulation and experiment, ACM SIGCOMM'95.
[9] X.W. Huang, R. Sharma, S. Keshav, The ENTRAPID protocol development environment, IEEE INFOCOM'99, New York, USA, March 21–25, 1999.
[10] L. Rizzo, Dummynet: a simple approach to the evaluation of network protocols, Computer Communication Review 27 (1) (1997) 31–41.
[11] S. Keshav, REAL: a network simulator, Technical Report 88/472, Department of Computer Science, UC Berkeley, 1988.
[12] SSF network module (SSFnet). Available from <http://www.ssfnet.org>.
[13] A. Meyer, L.H. Seawright, A virtual machine time-sharing system, IBM Systems Journal 9 (3) (1970) 199–218.
[14] G.R. Wright, W.R. Stevens, TCP/IP Illustrated, vol. 2, Addison-Wesley, Reading, MA, 1995.

S.Y. Wang is an Assistant Professor of the Department of Computer Science and Information Engineering at National Chiao Tung University, Taiwan. He received his Ph.D. degree in computer science from Harvard University in 1999. His research interests include computer networks, network simulations, and operating systems. He is the author of the Harvard network simulator and the NCTUns 1.0 network simulator. Due to its unique advantages, the NCTUns 1.0 network simulator was selected as a research demonstration by ACM MobiCom'02 held on 09/23/2002. As of 01/01/2003, it has been downloaded by more than 500 organizations world-wide since its release on 11/01/2002. More information about these simulators is available at http://NSL.csie.nctu.edu.tw/nctuns.html.

C.L. Chou currently is a Ph.D. student at the Department of Computer Science and Information Engineering, National Chiao Tung University (NCTU), Taiwan. He received his master degree in computer science from NCTU in 2002 and has participated in the NCTUns 1.0 network simulator project for three years.

C.H. Huang received his master degree in computer science from NCTU in 2002 and currently is working for a company. He participated in the NCTUns 1.0 network simulator project from 2000 to 2002.

C.C. Hwang currently is a second-year master student at the Department of Computer Science and Information Engineering, National Chiao Tung University (NCTU), Taiwan. He has participated in the NCTUns 1.0 network simulator project for two years.

Z.M. Yang received his master degree in computer science from NCTU in 2002 and currently is working for a company. He participated in the NCTUns 1.0 network simulator project from 2000 to 2002.

C.C. Chiou received his master degree in computer science from NCTU in 2001 and currently is working for a company. He participated in the NCTUns 1.0 network simulator project from 1999 to 2001.

C.C. Lin is a first-year master student at the Department of Computer Science and Information Engineering, National Chiao Tung University (NCTU), Taiwan. He has participated in the NCTUns 1.0 network simulator project for one year.