Available online at www.sciencedirect.com

Simulation Modelling Practice and Theory 15 (2007) 57–81

www.elsevier.com/locate/simpat

The design and implementation of the NCTUns network simulation engine

S.Y. Wang [*], C.L. Chou, C.C. Lin

Department of Computer Science, National Chiao Tung University, 1001 Ta-Hsueh Road, Hsinchu 300, Taiwan

Received 19 September 2005; received in revised form 13 May 2006; accepted 26 September 2006

Abstract

NCTUns is a network simulator running on Linux. It has several unique advantages over traditional network simulators. This paper presents the novel design and implementation of its simulation engine. This paper focuses on how to combine the kernel re-entering and discrete-event simulation methodologies to execute simulations quickly. The performance and scalability of NCTUns are also presented and discussed.
© 2006 Elsevier B.V. All rights reserved.

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. Simulation is flexible because it can study the performances of a system under various conditions. Simulation results are easier to analyze than experimental results because they are repeatable. In addition, in some cases involving high-speed movements (e.g., vehicle movements in Intelligent Transportation Systems), conducting simulations is much safer than conducting real experiments.

Developing a complete network simulator requires much time and efforts. A complete network simulator needs to simulate the hardware characteristics of networking devices (e.g., hub or switch), the protocol stacks employed in these devices (e.g., the bridge-learning protocol used in a switch), and the execution of application programs on these devices (for generating realistic network traffic). It also needs to provide network utility programs for configuring network topologies, specifying network parameters, monitoring traffic flows, gathering statistics about a simulated network, etc. Developing a high-fidelity and easy-to-use network simulator requires a large effort. Due to limited development resources, most traditional network simulators usually have the following limitations:

[*] Corresponding author. Tel.: +886 35 131 550.
E-mail address: shieyuan@csie.nctu.edu.tw (S.Y. Wang).

1569-190X/$ - see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.simpat.2006.09.013

- Simulation results are only abstracted version of the results produced by real hardware and software equipments. To constrain their complexity and development cost, most network simulators simulate real-life network protocol implementations with limited details, and this may 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 self-documented that "there is no dynamic receiver's advertised window for TCP." In addition, in a real-life network, a UDP-based application program can change its destination host (i.e., change the used destination IP address for an outgoing UDP packet) at run time on a per-packet basis. However, in ns-2, this cannot be easily done because in ns-2 the traffic sink agent (an object in C++) on the destination node must be paired with the traffic source agent on the source node at the beginning of a simulation via a scripting command. As such, a traffic generator program in ns-2 cannot dynamically change its destination node at run time like a normal real-life application program does.
- Most network simulators do not support the standard UNIX POSIX application programming interface (API) system calls. As such, existing and 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, large, 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 on a network simulated by ns-2. This means that it cannot be used to generate realistic traffic in a ns-2 simulated network. This also means that its performance under various conditions cannot be evaluated in a ns-2 simulated network.

To overcome these problems, the authors in [3,4] proposed a kernel re-entering simulation methodology and used it to develop the Harvard network simulator [5]. Later on, the authors improved the methodology and used it to develop the NCTUns network simulator [6]. (For brevity, we will just call it "NCTUns" in the rest of the paper.) Using this methodology, real-life protocol stacks can be directly used to generate more realistic simulation results and real-life application programs can be directly run on a simulated network.

The kernel re-entering simulation methodology can be applied to different simulation engine designs. The Harvard network simulator used a time-stepped method to implement its simulation engine and handle the interactions between the engine and real-life application programs. However, its simulation speed is unnecessarily low when traffic load is light. To overcome this problem, NCTUns applies the event-driven (i.e., the discrete event simulation methodology [7]) method to its simulation engine. With this novel design, NCTUns handles such interactions efficiently and achieves high simulation speeds when traffic load is light.

The contribution of this paper is the novel design of combining the discrete event simulation and the kernel re-entering simulation methodologies and its detailed implementation. With this design, simulations in NCTUns can be executed quickly and precisely despite the fact that there may be some other irrelevant activities occurring on the system. In addition, with this design, simulation results are repeatable because the interactions between the simulation engine process and all involved traffic generator processes are precisely controlled in the kernel.

The design and implementation presented in this paper contain novelty and they are not just straightforward engineering efforts. Note that although discrete event simulation is a mature field, combining it with the kernel re-entering simulation methodology is a new challenge. The challenge is due to the fact that the objects simulated in NCTUns are not contained in a single program but rather distributed in multiple independent components running concurrently on a UNIX machine. Obviously, one possible way to handle this situation is to treat this problem as a parallel simulation problem and apply parallel simulation approaches [7] to these components. However, conservative parallel simulation approaches, although simple to implement, normally result in very low simulation performance under tiny lookahead values (which is the case among these components). On the other hand, optimistic parallel simulation approaches, although potentially may achieve higher performance speedups on multiprocessor machines, are complicated and difficult to implement, and result in no performance gain on a uniprocessor machine, where most simulation users run their simulation cases. Since the original event-driven approach and existing parallel simulation approaches cannot solve this problem well, this paper proposes a novel, efficient, and simple to implement approach to solve this problem.

In the rest of the paper, we first survey related work in Section 2. In Section 3, we briefly present the kernel re-entering simulation methodology, giving readers enough background to understand how the kernel re-entering and discrete-event simulation methodologies can be combined. In Section 4, we move on to present the detailed design and implementation of the discrete-event simulation engine of NCTUns. In Section 5, we present simulation performance under various simulation conditions. In Section 6, we discuss the scalability issue. Finally, we conclude the paper in Section 7.

2. Related work

In the literature, some approaches also use a real-life TCP/IP protocol stack to generate results [8–12,14,16,17]. However, unlike the kernel re-entering methodology, these approaches are used for emulation purposes, rather than for simulation purposes.

Dummynet [11], when it was originally proposed, also used tunnel interfaces to use the real-life TCP/IP protocol stack in the simulation machine. However, since its release in 1997, Dummynet has changed substantially and now is used as a real-time traffic shaper or a bandwidth and delay manager in the FreeBSD kernel. It is no longer used as a network simulator.

ENTRAPID [12] uses another approach to use the real-life protocol stacks. 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 (i.e., the application program needs to be relinked so that the system calls issued by it are replaced with IPC library calls). As such, UNIX POSIX API can be provided by ENTRAPID and real-life application programs can be run in separate address space. This approach is heavy-weight in the sense that each virtual kernel process is a huge process. As such, when simulating a large network, the required memory space is likely to be huge and the simulation speed is likely to be low, because a large number of virtual kernel processes need to be context-switched frequently.

VMware [14] can also implement virtual machines. It can provide virtual x86-like hardware environments within which a number of different operating systems can be executed. With this capability, a number of virtual machines can be implemented on a single real machine to act as hosts or routers. They can be configured to form an emulated network and packets generated by real-life application programs run on these virtual machines can be exchanged through the emulated network. Although VMware can be used as an emulator, it is not a network simulator. As such, its results are not repeatable and a simulation case cannot be finished sooner than the simulated time in the real life. In addition, it is very heavy-weight. A virtual machine consumes much resource and runs substantially slower than the real machine – sometimes by an order of magnitude.

Emulab [15] can provide a large testbed for network experiments. Given a virtual topology, it can map the nodes in the virtual topology to some networking devices in the real life. It can integrate nodes that run emulations, nodes that run simulations, and live networks. It is a software platform that automatically allocates local and distributed networking resources to support a network experiment. It, however, is neither a network simulator nor an emulator by itself.

The virtual host approach [16] allows multiple virtual hosts to be instantiated on a real FreeBSD machine. It uses the jail (8) functionality provided by FreeBSD kernel to implement virtual hosts. Only one copy of the kernel is shared among multiple virtual hosts and each virtual host uses a different portion of the file system as its own file system. A virtual host has a shell and is assigned a single IP address when it starts up. All processes that are forked in a virtual host's shell run in the virtual host's root file system and can use the virtual host's IP address to communicate with a real-life network. Although this approach is light-weight, virtual hosts created using this approach cannot be formed as a generic simulated network or an emulated network. This is because each virtual host can have only one network interface (as its name suggests). As such, this approach cannot be used to simulate a network in which routers containing multiple interfaces exist.

The virtual router approach [17] allows multiple virtual routers to be instantiated on a real machine, each implemented by a user-level process. Virtual routers can be configured and connected by logical channels to form an emulated network. Such logical channels are implemented using UDP tunnels. Real-life application programs can exchange their packets through a network emulated by the virtual router approach. This is because these packets are intercepted in the kernel and then redirected to virtual routers. However, a network formed by virtual routers is not a simulated network. Instead, it is more like an emulated network. This is because the process scheduling order among multiple virtual router processes is not controlled and thus results are not repeatable.

OPNET modeler, REAL [18], ns-2, and SSFnet [19] represent the "traditional" network simulation approach. In this approach, the thread-supporting event scheduler, application code (not real-life application programs) that generates 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. A simulator constructed using this approach cannot easily provide UNIX POSIX API for real-life application programs to run normally to generate network traffic. Although some simulators may provide their own non-standard API to let real-life application programs interact with them (via IPC library calls), real-life application programs still need to be rewritten so that they can use the internal API, be compiled and linked with the simulator, and be concurrently executed with the simulator during simulation.

For these traditional network simulators, they can easily use the discrete-event simulation methodology to achieve high simulation speeds. This is because all objects that need to be simulated are created and contained in the single simulator program. As such, the timestamps of all simulation events generated by them can be known in advance and triggered correctly.

As for NCTUns, only one copy of the kernel (the default one) is used to simulate many virtual hosts and routers. Using the kernel re-entering simulation methodology, a virtual host or router is implemented by redirecting their packets into the same and only one kernel. Comparing NCTUns with the VMware approach, which uses a full operating system to implement a virtual router, and the Virtual Router approach, which uses a user-level process to implement a virtual router, NCTUns is very light-weight because no operating system or process is needed to implement a virtual host or router. In addition, all real-life application programs can exchange their packets across a network simulated by NCTUns without any modification, recompilation, or relinking. More importantly, its results are repeatable because the UNIX process scheduler in the kernel has been modified to precisely control the execution order of the NCTUns simulation engine process and all forked traffic generator processes. In contrast, the VMware and Virtual Router approaches cannot generate repeatable results.

NCTUns can be easily turned into an emulator. This is done by purposely slowing down its virtual clock and synchronizing it with the clock in the real life. If in a simulation case the virtual clock runs more slowly than the real clock (e.g., a network with very high traffic loads), it is impossible to "slow down" the virtual clock to synchronize it with the real clock. This means that the CPU power of the simulation machine is not high enough to turn the simulation case into an emulation case. In the emulation mode, with the support of user-level emulation daemons, a virtual host/router in an emulated network and a real host/router in a real network can exchange their packets through the emulated network. For example, a real TCP connection can be set up between a virtual host and a real host to exchange packets between them. The detailed design, implementation, and performance evaluations of NCTUns emulation are presented in [20].

Fig. 1 compares the architectures used by a traditional network simulator (e.g., OPNET, ns-2, SSFnet), a virtual kernel network simulator (e.g., ENTRAPID), and a kernel re-entering network simulator (i.e., NCTUns). This case assumes that the simulated network has six nodes and on each node there is an application program running on top of it to generate traffic. In the traditional network simulator approach, the application programs (abbreviated as App. in the figure) are fake application programs. Normally, they are very simple functions responsible for transmitting packets using some specific traffic patterns. In the virtual kernel network simulator approach, application programs are real programs. However, because their system calls need to be redirected to their corresponding virtual kernel processes, they need to be modified, recompiled, and relinked. In the kernel re-entering network simulator approach, application programs are real programs and do not need to be modified, recompiled, or relinked. In the figure, the grey area represents the place where a simulated network resides in each approach. Details about the kernel re-entering methodology will be presented in Section 3.

To be complete, Fig. 2 compares the architectures used by a VMware network emulator and a virtual router network emulator. In both approaches, application programs are real-life application programs without any

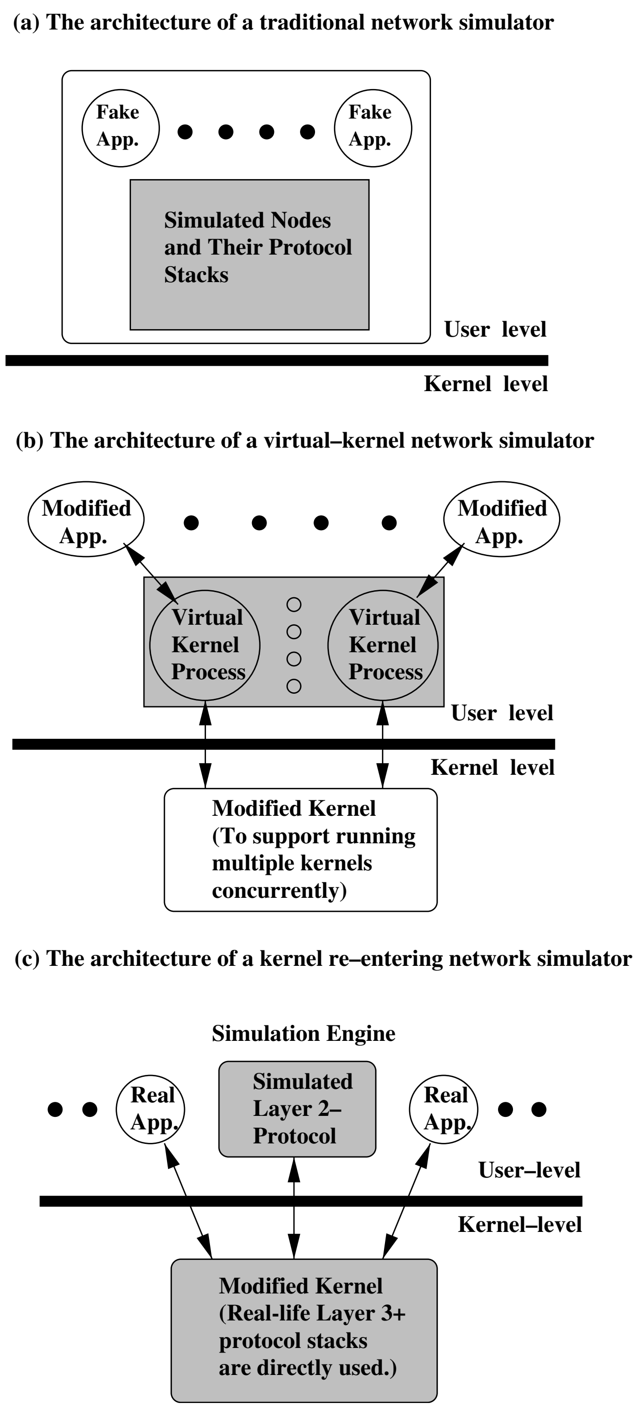
(a) The architecture of a traditional network simulator
Fake App.
Fake App.
Simulated Nodes and Their Protocol Stacks
User level
Kernel level
(b) The architecture of a virtual–kernel network simulator
Modified App.
Modified App.
Virtual Kernel Process
Virtual Kernel Process
User level
Kernel level
Modified Kernel (To support running multiple kernels concurrently)
(c) The architecture of a kernel re–entering network simulator
Simulation Engine
Real App.
Simulated Layer 2– Protocol
Real App.
User–level
Kernel–level
Modified Kernel (Real-life Layer 3+ protocol stacks are directly used.)

Fig. 1. A comparison of the architectures used by a traditional network simulator, a virtual-kernel network simulator, and a kernel re-entering network simulator. The gray area represents the place where a simulated network resides. This case assumes that the simulated network has six nodes and on each node there is an application program running on top of it to generate traffic.

modification. The VMware is very heavy-weight and is not likely to support many nodes either for simulation or emulation purposes. Although the virtual router approach is less heavy-weight than the VMware approach, it is still relatively heavy-weight and thus is unlikely to support a large number of nodes. Furthermore, because

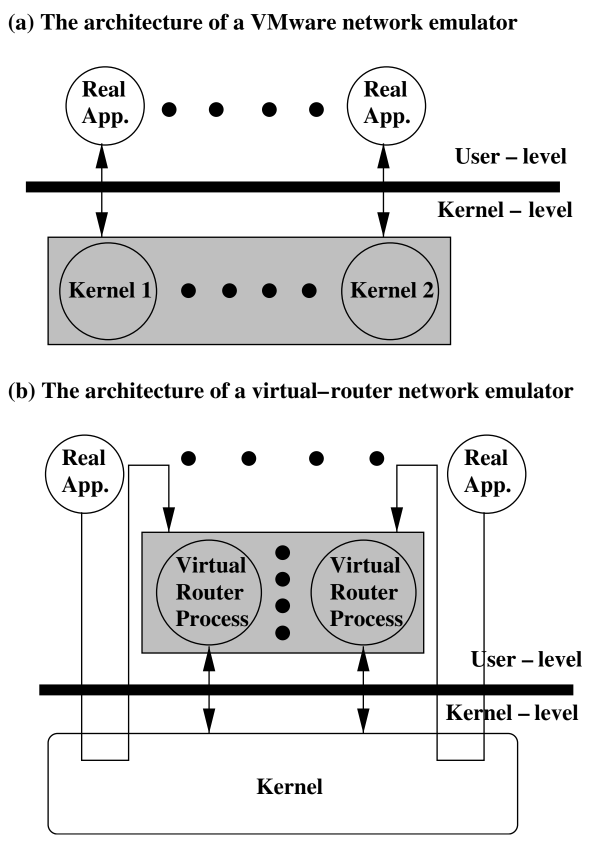

Fig. 2. A comparison of the architectures used by a VMware network emulator and a virtual-router network emulator. This case assumes that the emulated network has six nodes and on each node there is an application program running on top of it to generate traffic.

the process scheduling order among multiple virtual router processes is not precisely controlled, results generated by the virtual router approach is not repeatable, which makes it unsuitable for being a network simulator.

Going back to Fig. 1, one sees that in the kernel re-entering approach, simulation events may be generated by multiple real application programs, by the modified kernel (e.g., generating a TCP retransmit timeout event), and by the simulation engine program that simulates data-link layer protocols and physical-layer link characteristics (e.g., bandwidth, delay, and Bit-Error-Rate). Because these separate entities run independently, triggering their events in the correct order is more difficult to achieve than in a traditional network simulator.

The differences between this paper and other NCTUns-related papers are summarized below. In [3], the concept of the kernel re-entering simulation methodology was first proposed. In [4], the simple Harvard network simulator was presented as a proof-of-concept of this methodology. In [6], the improved kernel re-entering methodology and new functionalities (such as the integrated GUI environment) were presented. In [21], how NCTUns can be applied to various wireless network research areas was presented. None of these papers presents how to combine the discrete event simulation and the kernel re-entering simulation methodologies. This paper is the first one that presents this novel combination and its detailed design and implementation.

3. Kernel re-entering methodology

The initial version of the kernel re-entering simulation methodology was proposed in [3,4]. Later on, it was improved in [6]. As such, in the following, we will just present the basic concept of this methodology. After readers have had the required background, we will use a one-hop network as an example to present how to combine the discrete-event methodology with the kernel re-entering methodology.

3.1. Tunnel network interface

Tunnel network interfaces is the key facility in the kernel re-entering 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. Currently, the NCTUns installation script automatically creates 4,096 tunnel interfaces by default. Since a tunnel network interface is a software object and occupies little memory space in the kernel, this number can be further increased without any problem.

3.2. Simulating single-hop networks

Using tunnel network interfaces, one can easily simulate the single-hop TCP/IP network depicted in Fig. 3(a), where a TCP sender application program running on host 1 is sending its TCP packets to a TCP receiver application program running on host 2. One can set up the virtual simulated network by performing the following two operations. First, one configures 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, one runs a simulation engine process to simulate them. For the link from host i to host j ($i = 1$ or 2 and $j = 3 - i$), the simulation engine opens tunnel network interface i's and j's special files in /dev and then executes an endless loop until the total simulated time has elapsed. 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

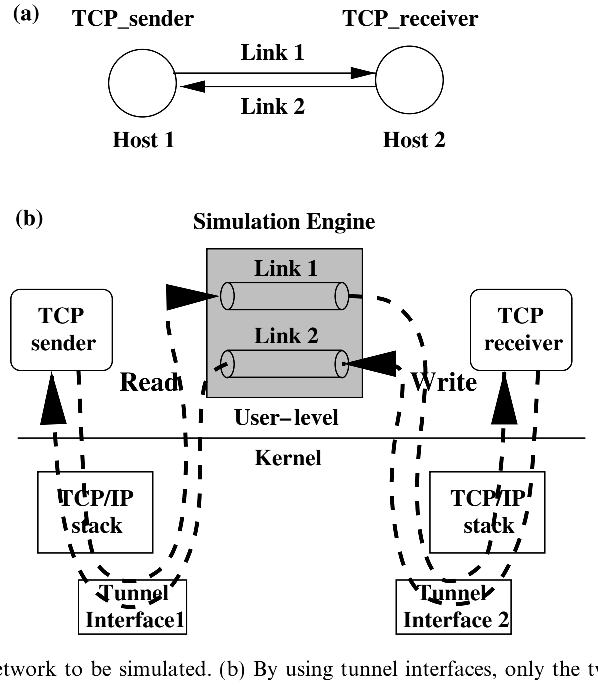

Fig. 3. (a) A single-hop TCP/IP network to be simulated. (b) By using tunnel interfaces, only the two links need to be simulated. The complicated TCP/IP protocol stack need not be simulated. Instead, the real-life TCP/IP protocol stack in the kernel is directly used in the simulation.

of tunnel interface *i*, waiting the link's propagation delay time plus the packet's transmission time on the link (in virtual time), and then writing this packet to the special file of tunnel interface *j*.

While the simulation engine is running, the virtual simulated network is constructed and alive. Fig. 3(b) depicts this simulation scheme. Since 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 the kernels on the sending and receiving hosts are the same one – the kernel of the simulation machine. This is why this simulation methodology is named "kernel re-entering methodology."

4. Discrete-event simulation engine

From the above description, one sees that it is natural and easy to use the time-stepped simulation approach to implement the simulation engine. Actually, for this reason, the Harvard network simulator uses the time-stepped approach to implement its simulation engine.

Although the time-stepped implementation is easy, its simulation speed can be very low. The reasons are explained below. On one hand, to achieve high simulation accuracy, one would prefer to use a small time step so that the simulation engine can poll the tunnel interfaces at a very fine granularity (e.g., every one nanosecond in virtual time). This is because otherwise packets entering a tunnel interface's output queue any time between two successive polls cannot be detected immediately and processed individually. Instead, they can only be detected and processed in a batch, and this may lower the accuracy of simulation results. On the other hand, to achieve high simulation speeds, one would prefer to use a large time step so that the virtual clock can advance quickly without wasting much time on many time steps in which there is no event to trigger. Apparently, there is a trade-off between simulation speed and simulation accuracy in this time-stepped implementation.

To achieve both high simulation speed and high simulation accuracy at the same time, it is desired that the event-driven approach can be applied to the kernel re-entering simulation engine of NCTUns. However, as will be seen later, it is not easy to do so.

4.1. Virtual clock advancing and sharing

When NCTUns is "simulating" a real-life network and application programs (actually it directly uses them), the virtual clock of the simulated network is kept in the user-level simulation engine process, which is responsible for advancing the virtual clock based on the event-driven approach. As in the time-stepped approach, the simulation engine process keeps a list of events sorted based on their timestamps. (A heap data structure is used to efficiently sort events.) To safely use the event-driven approach, it is very important that all events generated by the simulation engine process itself, the kernel, and all traffic generator processes must be scheduled into the simulated future (that is, the timestamp of any new event inserted into the list must be at least as large as the current time of the virtual clock), otherwise, the simulation will fail.

This problem – events are generated by multiple independent entities and they must be triggered in causal order, actually is a classical problem in the simulation society, and in the past, two parallel and distributed simulation approaches (i.e., conservative and optimistic) have been proposed to solve it. If either approach is used to solve this problem, the simulation engine process, the kernel, and every application program process must maintain their own virtual clocks and the simulation times of these virtual clocks must be synchronized by using a synchronization protocol.

Although these approaches may be able to solve this problem, they are too heavy-weight and thus may not result in good performances. In addition, they need to get and modify the source code of every used application program to correctly synchronize their simulation times. This will force NCTUns to lose one of its unique advantages that all real-life application programs can be directly used in a simulation without any modification. For these reasons, we did not adopt the parallel and distributed simulation approaches but instead proposed a novel approach for NCTUns.

The simulation engine process needs to pass the current virtual time down into the kernel. This is required for many reasons. First, the timers of TCP connections used in the simulated network need to be triggered based on the virtual time rather than the real time (recall that in NCTUns, 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 simulated network, the time-related system calls issued by them must be serviced based on the virtual time rather than the real time. For example, if one launches a ping program in a simulated network to send out ping packets once every second, the sleep (1) system call issued by the ping program must be triggered based on the virtual time, rather than the real time. Third, the in-kernel packet logging mechanism (i.e., the Berkeley packet filter 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.

In addition to being able to pass the current virtual time into the kernel, the simulation engine process must perform this operation in a low-cost and fine-grain way. In a simple way, the simulation engine process can pass the current virtual time into the kernel by periodically calling a user-defined system call or calling the user-defined system call whenever the virtual time changes. (For example, the simulation engine process can call the system call once every 1 ms in 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 process. (One such demand is that the in-kernel packet logging mechanism needs a microsecond-resolution clock to generate timestamps.) To solve this problem, we let the simulation engine process use a memory-mapping technique to map the memory location where the current virtual time is stored to a memory location in the kernel. With this technique, now at any time the virtual time in the kernel is as precise as that maintained in the simulation engine process without any system call overhead.

4.2. Event-passing channel

Because traffic generator processes and the kernel may generate events and these events should be inserted into the event list kept in the simulation engine process, we use the output queue of tunnel interface 0 (tun0) as the event-passing channel among these entities. (Note: NCTUns does not use tunnel interface 0 to simulate any link in a simulation.)

If a traffic generator process or the kernel generates an event, since the event is always generated in the kernel (see Section 4.3), the event will be immediately enqueued into the output queue of tun0. The user-level simulation engine process will immediately detect its arrival and read it out of the queue through the /dev/tun0 special file. The details about event detection is presented in Section 4.4.

4.3. Event generation

Because the simulation engine process keeps the event list, it will never miss its generated local events. As such, we focus only on the events generated by traffic generator processes and the kernel.

4.3.1. Traffic generator events

If a traffic generator process, after generating and sending out a packet, can immediately tell the simulation engine process when it will generate and send out its next packet, the simulation engine process will never miss such an event. In the following, we discuss the traffic events generated by one-way UDP, two-way request/reply UDP, and TCP traffic generators separately.

4.3.1.1. One-way UDP traffic events: Supporting one-way UDP traffic generators is easy. Normally, a one-way UDP traffic generator uses a particular traffic pattern to send out its packets. For example, the sending process of a constant-bit-rate (CBR) packet stream may send out a packet every M milliseconds, where M can be varied to change the traffic load (i.e., density) of the packet stream. Actually, any traffic pattern can be used and CBR is just one special case. For example, one can use an exponential distribution for the inter-packet-transmission times of the packet stream to generate a Possion traffic stream.

Despite the used traffic pattern, normally the sending process of a one-way UDP traffic generator can be simply implemented as follows: (written in a pseudo language)

```
(1) while (simulation is not done) {
(2) sleep (waitsometime);
(3) sendto (destination, packet);
(4) }
```

The statement in line 3 is the sendto() socket system call, which sends out a packet onto the network. The statement in line 2 is the sleep() system call, which asks the UNIX kernel to put the calling process into the sleep state for the specified period of time. (The usleep() system call can also be used, which can provide a microsecond resolution.) The length of the sleeping time (i.e., the value of the waitsometime variable) determines the inter-packet-transmission time. It can be a fixed value or drawn from a distribution.

To make the above one-way UDP traffic generator correctly work in a simulated network, NCTUns needs to perform two tasks. First, the kernel needs to service the sleep(waitsometime) system call based on the virtual time, rather than the real time. Second, the intention that after sleeping waitsometime seconds the calling process will continue to send out a packet should be represented as a timer event and immediately inserted into the event list kept in the simulation engine process.

To perform both tasks, we modified the sleep() system call so that when the sleep(waitsometime) system call is called, a KERNEL-TIMER-SETUP event recording the wake-up time is created and inserted to the event-passing channel. The simulation engine process will immediately detect the arrival of such an event, read it out, and then insert it into its event list. The event detection procedure is detailed in Section 4.4.

4.3.1.2. Two-way request/reply UDP traffic events: Supporting two-way request/reply UDP traffic generators can also be easily done. The receiving process of a two-way request/reply traffic generator can be simply implemented as follows:

```
(1) while (simulation is not done) {
(2) recvfrom (& requestPacketBuffer, & source);
(3) sendto(source, replyPacket);
(4) }
```

In line 2, the receiving process calls a recvfrom() socket system call to wait for a request packet to arrive. The recvfrom() is a blocking system call, which means that it will block (i.e., the calling process will be put into the sleep state) in the kernel if no request packet arrives. If a request packet arrives, its content will be copied to the provided packet buffer (i.e., requestPacketBuffer) and the IP address and port number of the sending process will be copied to the provided address variable (i.e., source). In line 3, the receiving process calls a sendto() socket system call to send back its reply packet, using the IP address and port number stored in the source variable. After sending back the reply packet, the receiving process will again get blocked on the recvfrom() system call, waiting for the next request packet to arrive.

The following summarizes the involved processing when the kernel receives a packet from a network interface. After a packet is received, it will be processed by the TCP/IP protocol stack and finally inserted into the socket receive buffer used by the receiving process. The sleeping receiving process (which is now blocked on the recvfrom() system call) will then be waked up by the kernel and its state be restored back to the ready state. When the receiving process is scheduled again (i.e., given the CPU), it will restart from the middle of the recvfrom() system call where it was blocked. The recvfrom() system call will copy the packet from the socket receive buffer to the packet buffer provided by the receiving process and then return from the kernel to the receiving process. When getting the CPU again, the receiving process will then call the sendto() system call to send out a reply packet. This reply packet will be inserted into the output queue of some tunnel interface used by the receiving node.

Section 3.2 illustrates that when a request packet arrives at the receiving node, the simulation engine process will write it into the kernel to deliver it to the receiving process. The receiving process may or may not send back a reply packet. However, because the event-driven approach requires that a generated event (here, it is the generated reply packet) be immediately detected and inserted into the event list, we let the simulation engine process immediately check all involved tunnel interfaces after it writes a packet into the kernel. Here, a tunnel interface is involved in a packet-write operation if it is used by the node on which the receiving process is run. If any packet is found in the output queues of these involved tunnel interfaces, the simulation engine process will immediately get it, make it an event, insert it into the event list, and start simulating sending it back on the reverse link. Since the simulation engine process does not advance its virtual clock during this period of time, this event is detected immediately after its creation in virtual time.

In the above description, we implicitly assume that after the simulation engine process writes the request packet into the kernel and before it checks all involved tunnel interfaces, the receiving process has had a chance to gain the CPU to generate a reply packet. However, this property is not guaranteed without explicitly controlling the scheduling order between the simulation engine process and the traffic generator processes. We present the details of this scheduling design in Section 4.7.

Of course, after receiving a packet, not every receiving process will generate and send back a reply packet. For example, in the pseudo code, if the sendto() system call in line 3 is removed, the receiving process will become a normal one-way receiving process. Therefore, sometimes checking involved tunnel interfaces does not necessarily find any packet to be sent. However, this does not affect the correctness of simulation results. It just affects simulation performance insignificantly.

The above design not only applies to the UDP protocol, but also applies to the ICMP protocol. For example, it can support the ping request/reply traffic generator equally well, which uses ICMP's echo request/reply commands. Since the ICMP protocol stack resides in the kernel, after an ICMP echo request packet is received by the kernel, the ICMP echo reply packet will be directly generated by the kernel, not by any user-level receiving process. Although the ICMP request/reply processing is different from the UDP request/reply processing, the same event-driven design can handle both cases correctly.

4.3.1.3. TCP traffic events: Supporting TCP traffic generators is more difficult than supporting UDP traffic generators. This is because, unlike in the UDP traffic generator case, when to send out TCP packets onto a network actually is triggered by the TCP protocol module in the kernel (due to TCP congestion control), rather than directly by a user-level sending process. In addition, the TCP protocol module uses a self-clocking mechanism (i.e., send out new data packets when receiving ACK packets) to automatically send out its packets. As such, unlike in the UDP traffic generator case, sending a TCP data packet is not preceded by a sleep() system call. The simulation engine process therefore cannot easily know the TCP module's intention to send out its next packet. In the following, we use a greedy TCP transfer to explain how the event-driven design can be used. It is assumed that the sending and receiving nodes are directly connected by a link.

4.3.1.3.1. Connection setup phase: Here we focus on the TCP connection setup phase, which can be supported by using the same design for supporting request/reply traffic generators. On the receiving node, a receiving process (e.g., a web server) will call a blocking accept() socket system call to wait for a connection request packet (i.e., the first SYN packet of the TCP 3-way handshaking connection setup procedure) to arrive. On the sending node, a sending process (e.g., a web client) will call a connect() socket system call to send out the connection request packet (i.e., the first SYN packet). The SYN packet will be enqueued into the output queue of a tunnel interface (which simulates the link in the forward direction), and the simulation engine process will immediately detect it, make it an event, and then insert it into the event list. The event then will be immediately triggered and processed, which starts simulating sending this packet onto the link. After the packet has arrived at the other end of the link, the simulation engine process will write it into the kernel, and the SYN packet will be received by the TCP module in the receiving node.

The TCP module in the receiving node will immediately generate the second connection setup packet (i.e., the SYN + ACK packet) and send it back. This packet will then be enqueued into the output queue of the tunnel interface that simulates the link in the reverse direction. As in the two-way request/reply traffic generator case, the reply packet will be immediately detected by the simulation engine process and read out of the kernel, and the simulation engine process will start simulating sending this packet onto the link in the reverse direction. When the packet arrives at the other end of the link, the simulation engine process will write it into the kernel. The TCP module in the sending node will then generate the third connection setup packet (i.e., the ACK packet). Again as in the two-way request/reply case, this packet will be detected and read by the simulation engine process immediately, and the simulation engine process will start simulating sending it onto the link in the forward direction. When the packet arrives at the receiving node, the TCP connection's 3-way handshaking connection setup procedure is finished. At this time, both the sending and receiving processes will return from their connect() and accept() system calls, respectively.

4.3.1.3.2. Data transfer phase: Here we focus on the TCP data transfer phase. Before we present the event-driven design, we first summarize the behavior of the sending and receiving processes in normal uses. After the connection is set up, the sending process will continuously call the send() socket system call to write its data into its socket send buffer. If the socket send buffer is not full, the send() system call will immediately return, allowing the sending process to write more data into the socket. Otherwise, the send() system call will get blocked and the sending process will be put into the sleep state. When the socket send buffer becomes non-full (because some data have been sent and their corresponding ACK packets have come back), the kernel will wake up the sending process, enabling it to write more data into the socket. The pseudo code of the sending process is shown below.

```
(1) connect(destination);
(2) while (simulation is not done) {
(3)   send (destination, moreNewData);
(4) }
```

On the other hand, the receiving process will continuously call the receive() socket system call to try to read more data out of its socket receive buffer. Since the receive() system call is a blocking system call, if there is no data in the socket receive buffer to be read out, the receiving process will be put into the sleep state. The kernel will wake it up when some new data packets have arrived. The pseudo code of the receiving process is shown below.

```
(1) accept(& source);
(2) while (simulation is not done) {
(3)   receive(source, & packetBuffer);
(4) }
```

In the above description, we just describe when data is written into/read out of a socket buffer and do not describe when packets are sent onto the network. This is because the TCP protocol implements error, flow, and congestion controls. It is the TCP protocol that determines when to send out a packet, not the sending process. In contrast, since the UDP protocol implements no such controls, it is the sending process that determines when to send out a packet, not the UDP protocol module. In the following, we will present how to use the event-driven approach to support asynchronous TCP packet transfers.

We first focus on the TCP module in the receiving node. When a data packet arrives at the receiving node, normally the TCP module will send back an ACK packet immediately. In addition, because TCP uses a cumulative ACK scheme, when a data packet is lost, the TCP module in the receiving node will send back a duplicate ACK packet whenever an out-of-order data packet is received. Both of these scenarios can be supported by using the design for supporting the request/reply traffic generators. If the TCP module uses the delay-ACK option, which may delay an ACK packet transmission by up to 200 ms for trying to piggyback some data, the transmission of the ACK packet will be triggered by the expiration of the TCP's fast timer. The design for supporting this scenario is presented in Section 4.3.2.

Then, we focus on the TCP module in the sending node. Although we said earlier that the times when the TCP module sends out a packet are independent of the times when the TCP sending process calls the send() system call, in some cases the TCP module's output function (i.e., tcp_output() in BSD implementation) still needs to be directly triggered by the TCP sending process calling the send() system call. This happens when the send() system call finds that the TCP connection currently has no data packets outstanding in the network and its retransmit timer is not active (e.g., the TCP connection just finishes its connection setup phase or the TCP connection currently is idle). In such a case, since no ACK packet is expected to come back to trigger out more data packets (i.e., self-clocking), the send() system call must send out the data by itself and this is done by directly calling the TCP module's output function (tcp_output()). On the other hand, if the send() system call finds that either there are outstanding data packets in the network or the TCP retransmit timer is already active, it will just insert the data into the socket send buffer and simply return. This is because the send() system call knows that its data will later be sent out by tcp_output(), which will either be triggered by a returning ACK packet or the expiration of the retransmit timer.

Having presented the TCP output procedure, we now present how the event-driven approach is used to support asynchronous TCP data transfers. First, it is clear that data packets generated and sent due to the arrival of a returning ACK packet can be supported by the same design for supporting request/reply traffic generators. Second, the data packet generated and sent due to directly calling tcp_output() by the send() system call can be immediately detected by the simulation engine process. This is because if the data packet is triggered out due to the arrival of the last packet of the 3-way handshaking connection setup procedure (like what a web server does), this first data packet can be supported by the same design for supporting request/reply traffic generators. If the data packet is not the first data packet but a packet generated when the TCP sending process becomes active again after an idle period of time, then before the sending process calls the send() system call, it must have called a sleep() system call. Apparently, this can be supported using the same design for supporting one-way UDP traffic generators presented in Section 4.3.1.1. The last reason why a data packet would be generated and sent is due to the expiration of the TCP retransmit timer. The design for supporting this case is presented in Section 4.3.2.

4.3.2. Kernel events

Here we present the event-driven design for supporting the events generated inside the kernel. Kernel events come from two main categories. The first category is TCP timer events. A TCP connection has several timers. For example, it has retransmit, persist, keepalive, delay-ACK, and 2MSL timers. Except for the 2MSL timer, which deallocates all the resources used by an already-closed TCP connection, when a TCP timer expires, a packet will be generated and sent onto the network for some purposes. The second category is the timer events generated by traffic generator processes calling time-related system calls such as sleep() and alarm().

Although these timer events are generated due to different reasons and from different control paths, starting from FreeBSD 3.0 (right now the latest FreeBSD version is 6.1), they all map to callout requests and are triggered by the same callout mechanism in the kernel. (The Linux kernel adopts a similar design.) In each invocation of the clock interrupt service routine, the real time clock maintained in the kernel is advanced by one tick (10 ms or 1 ms in real time, depending on whether the kernel HZ option is set to 100 or 1000 when the kernel was compiled). The clock interrupt service routine also checks whether the current real time is the same as the triggering time of some callout request events. If any such event is found, it will call the event's associated function (e.g., the TCP retransmit function).

To use the event-driven approach to quickly trigger these timer events in virtual time, upon creation, these timer events are immediately passed to the simulation engine process through the event-passing channel. How the simulation engine process can immediately detect these events is presented in Section 4.4.

To let the simulation machine still work properly while NCTUns is running, only those events generated by traffic generator processes and their TCP connections are triggered based on the virtual time. All other events are still triggered based on the real time. In order for the kernel to distinguish between traffic generator processes and all other application processes, when a traffic generator process is forked by the simulation engine process, it is automatically registered with the kernel. In addition, later when it creates a TCP connection, the TCP connection is also automatically registered with the kernel. With this registration information, the kernel can thus correctly trigger events based on different clocks.

4.4. Event detection

Upon creation, if not created by the simulation engine process, an event needs to be passed to the simulation engine process and inserted into the event list immediately. For the event-driven approach to work correctly, "immediately" here means that it takes no time in virtual time, that is, the virtual clock should be frozen between the time when an event is created and the time when it is inserted into the event list. In previous sections, we have presented various scenarios in which events may be generated and inserted into the event-passing channel. Now we present how the simulation engine process can immediately detect these events. That is, how does it know that an event has just been inserted into the event-passing channel?

In the NCTUns design, there is no notification mechanism for notifying the simulation engine process that an event was just created and inserted into the event-passing channel. This is because these events already serve notification purposes. One cannot recursively use another kind of notification events to notify the simulation engine process of the arrivals of these "notification" events. Instead, the simulation engine process must know when an event may just have been created and inserted into the event-passing channel by itself. Indeed, the simulation engine process has the capability to perform this task. This capability comes from the fact that an event can be created only after the simulation engine process has performed one of three different operations. Therefore, the simulation engine process can actively check the event-passing channel every time when it has performed any of such operations. In the following, we present these different operations.

4.4.1. Forking a traffic generator process

After a traffic generator process is forked by the simulation engine process, it will begin to do some work. For the one-way UDP traffic generator case in Section 4.3.1.1, it will call the sleep() system call and cause a KERNEL-TIMER-SETUP event to be generated and inserted into the event-passing channel. For the request/reply traffic generator case in Section 4.3.1.2, it will get blocked in the kernel when executing the recvfrom() system call and generate no event. For the TCP traffic generator case, the sending process will generate a PACKET-GEN event and cause it to be inserted into the event-passing channel when it calls the connect() system call. On the other hand, the receiving process will get blocked in the kernel and generate no event when executing the accept() system call.

From the above description, we see that after forking a traffic generator process, some events may be generated and inserted into the event-passing channel. The simulation engine process thus needs to check the event-passing channel after forking a traffic generator process.

4.4.2. Writing a packet into a tunnel interface's special file

In Section 4.3, in the request/reply and TCP traffic generator cases, we see that some events may be generated after the simulation engine process writes a packet into a tunnel interface's special file (e.g., /dev/tun3). For example, when an ACK packet arrives at the sending node, the TCP module in the sending node may (1) trigger out more data packets, (2) cancel the current retransmit timer, and (3) set up the retransmit timer again using a fresher timeout value. These operations will generate PACKET-GEN, KERNEL-TIMER-CANCEL, and KERNEL-TIMER-SETUP events, respectively, and cause them to be inserted into the event-passing channel. The simulation engine process thus needs to check the event-passing channel after writing a packet into a tunnel interface's special file.

4.4.3. Triggering a callout request

After the simulation engine process triggers a KERNEL-TIMER-SETUP event (presented in Section 4.5), it also needs to check the event-passing channel because some events may have been generated. For example, in the one-way UDP traffic generator case in Section 4.3.1.1, after a KERNEL-TIMER-SETUP event is triggered by the simulation engine process, the traffic generator process will (1) return from the sleep() system call, (2) call the sendto() system call, and then (3) call the sleep() system call again, which makes it blocked again in the kernel. The above step (2) and (3) will generate a PACKET-GEN and KERNEL-TIMER-SETUP event, respectively, and cause them to be inserted into the event-passing channel. Therefore, the simulation engine process also needs to check the event-passing channel after triggering a callout request event.

4.5. Event triggering

In the event-driven approach, the simulation engine process always repeats the following cycle: (1) jump to the triggering time of the event with the smallest timestamp, and (2) trigger the event by performing the function associated with the event. The events that may be stored in the event list can be classified into four categories. In the following, we present them separately.

The events in the first category are PACKET-GEN events. A PACKET-GEN event indicates that a packet has just been generated and inserted into the output queue of some tunnel interface (e.g., tun4). When the simulation engine process triggers such an event, it will read a packet out of the output queue of the specified tunnel interface and start simulating sending it onto a link. These events must be triggered immediately after they are created and detected.

The events in the second category are KERNEL-TIMER-SETUP events. A KERNEL-TIMER-SETUP event indicates that a kernel timer should be set up and expire at the specified time. When triggering such an event, the simulation engine process calls a user-defined system call, which is provided by NCTUns. The system call will find the corresponding timer event in the kernel's callout request queue, dequeue it, and call the function associated with this timer event. For example, if a KERNEL-TIMER-SETUP event was created due to a traffic generator process calling a sleep() system call, triggering this event will simply change the state of the calling process from SLEEP to READY and make the sleep() system call return from the kernel. On the other hand, if such an event was created due to setting up a TCP retransmit timer, triggering such an event will call the TCP module's retransmit function.

The events in the third category are KERNEL-TIMER-CANCEL events. A KERNEL-TIMER-SETUP event, after being detected and inserted into the event list, may need to be canceled before it is triggered. For example, a KERNEL-TIMER-SETUP event created due to the TCP module setting up a TCP retransmit timer will likely need to be canceled before its triggering. This is because (1) the TCP module will set up a TCP connection's retransmit timer whenever it sends out a data packet and detects that the retransmit timer is not active, and (2) it will cancel the retransmit timer as soon as the corresponding ACK packet comes back. Triggering a KERNEL-TIMER-CANCEL event can be done by simply finding the corresponding KERNEL-TIMER-SETUP event and remove it from the event list.

The events in the fourth category are all other events generated by the simulation engine process itself. For example, a MAC802.3 module may generate a timer event whose expiration will cause a packet to be sent after an exponential backoff period.

4.6. Requirements for traffic generator processes

For the event-driven approach to work correctly, NCTUns requires that if a traffic generator process would send out a packet in the future, the intention must be passed to the simulation engine in advance (e.g., by preceding a sendto() system call by a sleep() system call). Otherwise, the traffic generator process's sending out a packet must be an immediate response to the arrival of an packet (e.g., sending back an ACK packet in response to the arrival of a data packet). As we have presented previously, in the current UNIX network subsystem design, these requirements are automatically met.

Another requirement is that a traffic generator process cannot be a CPU-bound busy-looping process. That is, it cannot keep generating and dumping packets onto the network without sometimes calling a sleep() system call to release the CPU. The reason is clear as the simulation engine process needs to gain CPU cycles to advance its virtual clock. If a traffic generator process would keep using the CPU without releasing it, the simulation engine process will have no chance to advance the virtual clock and thus the whole simulation cannot proceed. Therefore, NCTUns requires that when the simulation engine process releases its CPU control to a traffic generator process, the traffic generator process, after executing some statements of its program, must get blocked again (either by explicitly calling sleep() or automatically getting blocked due to no packet to receive). This requirement ensures that the simulation engine process can acquire the CPU control most of the time to advance the simulation.

Fortunately, almost all normal traffic generators meet this requirement automatically. For the greedy TCP connection case, because TCP implements congestion control (for fairly sharing the available bandwidth in a network) and flow control (for not to overflow the receiving process's socket receive buffer), the TCP sending process cannot keep using the full CPU speed to continuously generate and dump TCP packets onto the network. Instead, it can only dump its congestion window size worth of data per its connection's round-trip-time (RTT) in virtual time. As such, the TCP sending process actually is put into the sleep state most of the time.

The TCP receiving process also is put into the sleep state most of the time because new data cannot continuously arrive in each CPU cycle.

For the greedy UDP traffic generator case, this could be a problem if the sending process is implemented in the following way:

```
(1) while (simulation is not done) {
(2)   sendto (destination, packet);
(3) }
```

Fortunately, most existing greedy UDP traffic generator sending processes (e.g., the ttcp or the netperf performance benchmarking program that can be easily fetched on the Internet) are implemented in the following way:

```
(1) while (simulation is not done) {
(2)   result = sendto(destination, packet);
(3)   if (result == noBuffer)
(4)     sleep(sometime);
(5) }
```

In line 2, the sending process gets the result of sending a UDP packet. In lines 3 and 4, if the result indicates that the UDP packet is dropped on the local host due to a full queue (the UNIX UDP/IP protocol stack can return this no-buffer-available information), the sending process puts itself into the sleep state for a while (the period is usually set to a value that is large but can still keep the output queue nonempty at all time). The reason for doing this is evident. If the sending process does not put itself to sleep for sometime, it will keep generating and sending packets and most of these packets will be dropped on the local host without having a chance to enter the network. For example, on a high-speed machine equipped with a 3.0 GHZ Pentium processor, if a greedy UDP sending process keeps using the full CPU speed to continuously send packets onto a network through a 10 Mbps Ethernet card, more than 90% of its packets will be dropped on the local host due to buffer overflow in the Ethernet device driver's output queue.

Since the bad implementation causes problems for the sending process's data error recovery, provides no extra benefit for the sending process, and only unnecessarily wastes CPU cycles, all greedy UDP traffic generators provided by NCTUns use the good implementation.

4.7. Explicit 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. Since the default UNIX process scheduler uses a dynamic priority mechanism and thus cannot guarantee a desired scheduling order, we have to solve this problem.

As presented in Section 4.6, when the simulation engine process wants to release the CPU to a traffic generator, we should let the intended traffic generator process really gain the CPU control and execute some of its program statements until it gets blocked again. Also, when the traffic generator process gets blocked again, we should let the simulation engine process regain the CPU. Ensuring this correct operation sequence, however, requires us to fully control UNIX process scheduling.

For example, for the one-way UDP traffic generator case presented in Section 4.3.1.1, after the simulation engine process triggers (i.e., calls the user-defined system call presented in Section 4.5) the KERNEL-TIMER-SETUP event created due to the traffic generator process calling the sleep() system call, the simulation engine process will expect that the traffic generator process would have returned from its sleep() system call, called the sendto() system call to send out a packet, and called the sleep() system call again and now blocked. This requires that the UNIX process scheduler explicitly switch the CPU control from the simulation engine process to the traffic generator process, and then from the traffic generator process back to the simulation engine process. We found that this requirement can be automatically met due to the default UNIX process scheduling design without modifying the default UNIX process scheduler.

In the default UNIX process scheduling design, 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. When a process gets blocked in the kernel, its kernel-mode priority will temporarily be assigned a very high priority, which is higher than any user-mode priority. As such, when it is waked up in the kernel, it can immediately regain the CPU, finish its system call, and return from the system call. (This design is to avoid holding too many valuable resources in the kernel, which may cause resource deadlock situations.) Due to this design, when the simulation engine process performs its operation (e.g., triggering a KERNEL-TIMER-SETUP event) and causes a sleeping traffic generator process to be waked up, the traffic generator process (which is still in the kernel mode) will automatically take the CPU control from the simulation engine process (which is in the user mode). Because in NCTUns, a traffic generator process will get blocked again after executing some program statements, the CPU control will again automatically be transferred to the simulation engine process, which allows the simulation to proceed.

5. Simulation performance

In the following, we present the performances of the event-driven version of NCTUns simulation engine under various traffic conditions. We also compare the performances of the event-driven version of NCTUns (simply named NCTUns), the time-stepped version of NCTUns (named NCTUnsTS), and ns-2. For NCTUnsTS, it polls all tunnel interfaces used in a simulation case every 1 ms in virtual time to detect packet events.

The used machine for the performance testing is an IBM T30 notebook computer equipped with a 1.6 GHZ Pentium-M processor and 256 MB RAM. The operating system used is Fedora Linux 1.0, which is a non-commercial descendent of Red-Hat 9.0 distribution with Linux kernel version 2.4.22.

5.1. Self-evaluation

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 1000 Mbps and the delay is set to 1 ms, in both directions. The traffic generated is a one-way constant-bit-rate (CBR) UDP packet stream. Each UDP packet's payload size is set to 1400 bytes.

We varied the packet interval 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 interval time is the time interval between two successive packet transmissions. The tested intervals are 0.00001, 0.00005, 0.00025, 0.00125, 0.00625, 0.03125, 0.15625, 0.78125, and 3.90625 s, respectively. (Note: The interval is enlarged by 5 times gradually in the tests.)

The performance metric reported is the ratio of the simulated seconds to the seconds required for finishing the simulation. In all of our tests, the simulated seconds is set to 3000 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 as the simulated time to finish simulating the case.

Table 1 lists the test results of the relationship between speedup ratio and CBR packet interval time. The CBR bandwidth corresponding to each CBR time interval and the elapsed time for finishing the simulation case are also listed. We see that due to the discrete-event simulation engine design, a simulation case can be finished quickly if it does not have many events (i.e., traffic) to process per simulated second. Besides, when the network traffic load is less than 232 Mbps, a simulation case can be finished sooner than real time on an IBM T30 notebook computer.

5.2. Comparison with ns-2 and NCTUnsTS

Here we compare the performances of NCTUns with those of ns-2 and NCTUnsTS. The performance metrics used are (1) required memory space, (2) required time, and (3) required disk space for storing packet trace

Table 1
The simulation performance under various CBR UDP traffic load

CBR time interval (s)	CBR bandwidth	Elapsed time (s) for simulating 3000 s	Ratio (higher is better)
0.00001	1 Gbps	15243	0.19
0.00005	232.96 Mbps	3135	0.96
0.00025	46.59 Mbps	621	4.83
0.00125	9.32 Mbps	124	24.19
0.00625	1.86 Mbps	25	120.00
0.03125	372.73 Kbps	6	500.00
0.15625	74.55 Kbps	2	1500.00
0.78125	14.91 Kbps	1.15	2608.69
3.90625	2.981 Kbps	0.81	3750.00

(A higher ratio means a better performance.)

log files. The required memory is the maximum size of the simulation engine process during a simulation run. This information can be returned by the "top" or "vmstat" command on UNIX. We first compare their performances on a fixed network. Then we compare their performances on a wireless ad hoc network.

5.2.1. Fixed network

The configuration of the tested fixed network is depicted in Fig. 4. In this configuration, there are five source nodes, one destination node, and a bottleneck router node. The bandwidth and delay of all links are set to 100 Mbps and 1 ms, respectively. The maximum packet queue length of the FIFO queue in the bottleneck router is set to 300 packets. We conducted two tests. In the first test, there is a CBR UDP flow between each pair of a source and the destination node. That is, in total five UDP flows will compete for the bandwidth of the bottleneck link. The length of each UDP packet is 1000 bytes and their packet interval time is set to 0.00005 s. As such, the load of each UDP flow is 160 Mbps. In the second test, each CBR UDP flow is replaced with a greedy TCP flow and in total five greedy TCP flows compete for the bandwidth of the bottleneck link. The length of each TCP packet is 1500 bytes, which is Ethernet's MTU. In both tests, the time to be simulated for each simulation case is 100 s.

Table 2 shows the performance comparisons between NCTUns, ns-2, and NCTUnsTS for the CBR UDP traffic case. In contrast, Table 3 shows the performance comparisons between NCTUns, ns-2, and NCTUnsTS for the greedy TCP traffic case. Here we just show the comparison results. These results are discussed later in Section 5.3.

5.2.2. Wireless ad hoc network

The configuration of the tested wireless ad hoc network is depicted in Fig. 5. In this configuration, there are in total 100 wireless ad hoc nodes deployed as a 10×10 grid. Each node is equipped with a 11 Mbps IEEE 802.11(b) wireless interface operating in the ad hoc mode. The transmission and interference ranges of such a wireless interface is set to 250 and 550 m, respectively, which are the default settings used in ns-2. The hor-

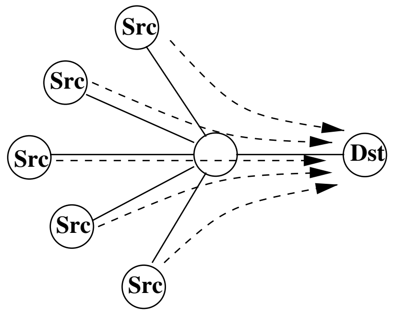

Fig. 4. The multi-source-single-destination fixed network topology used to compare the performances of NCTUns, ns-2, and NCTUnsTS.

Table 2
Performance comparisons between (A) NCTUns, (B) ns-2, and (C) NCTUnsTS

Metric	(A)	(B)	(C)	(A)/(B)	(A)/(C)
With trace file					
Memory usage (KB)	16288	13612	16288	1.196	1
Required time (s)	700	635	5875	1.102	0.119
Trace file size (MB)	600	7066	600	0.085	1
Without trace file					
Memory usage (KB)	12504	13427	12504	0.931	1
Required time (s)	372	51	5464	7.294	0.068

The tested traffic is CBR UDP traffic.

Table 3
Performance comparisons between (A) NCTUns, (B) ns-2, and (C) NCTUnsTS

Metric	(A)	(B)	(C)	(A)/(B)	(A)/(C)
With trace file					
Memory usage (KB)	13652	12684	13652	1.076	1
Required time (s)	187	251	5375	0.745	0.035
Trace file size (MB)	210	2664	210	0.079	1
Without trace file					
Memory usage (KB)	12068	12288	12068	0.982	1
Required time (s)	84	19	5125	4.421	0.016

The tested traffic is greedy TCP.

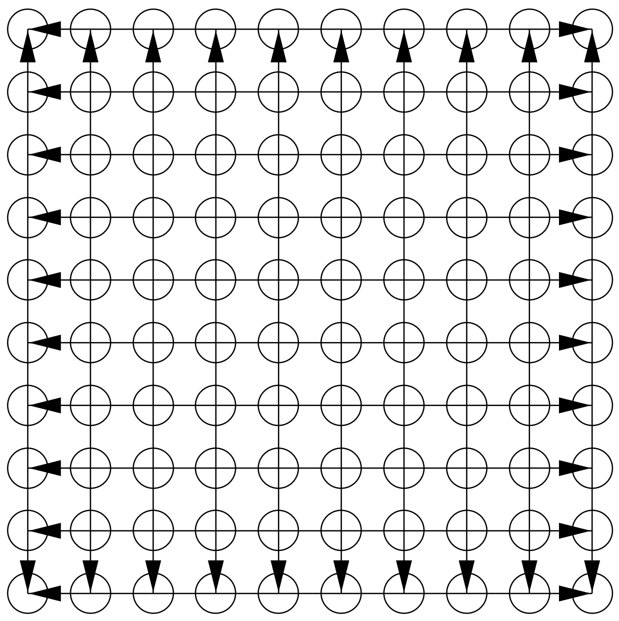

Fig. 5. The multi-source-multi-destination wireless ad hoc network topology used to compare the performances of NCTUns, ns-2, and NCTUnsTS.

izontal and vertical distance between two neighboring nodes is set to 240 m so that a node can only communicate with its direct neighbors in the horizontal and vertical directions. As such, if a node on the left side of the grid would like to send a packet to another node on the right side of the grid, the packet needs to traverse 9 hops to reach its destination node. That is, all of the nodes on the same row need to forward the packet to its right neighboring node except for the destination node.

The maximum packet queue length of the FIFO queue used by each wireless interface is the default value of 50. The traffic load is generated by 40 CBR one-way UDP flows. Among them, ten flows are in the east direction and each flow is placed on a different row. The source node and destination node of each such a flow is on the left and right of the grid, respectively. That is, each such a flow is a 9-hop flow. Similarly, in the west, north, and south directions, there are ten 9-hop flows in each of these directions.

For this wireless network performance evaluation, we conducted three tests with different levels of traffic load. In the first test, the packet interval time of each CBR flow is set to 1 s. In the second test, the generated load is increased 10 times by reducing the interval from 1 s to 0.1 s. In the third test, the generated load is increased 100 times (relative to the first test) by reducing the interval from 1 s to 0.01 s. We used these tests to observe the performance differences among the three simulators under light, medium, and heavy traffic load, respectively. In these tests, the packet size of each UDP packet is 1000 bytes. The used routing path between a pair of nodes is the shortest path between them. The routing paths between all pairs of nodes are pre-generated by the simulators before a simulation run is executed. In these tests, the time to be simulated for each simulation case is 100 s.

Tables 4–6 show the performance comparisons between NCTUns, ns-2, and NCTUnsTS in these three tests. Again, here we just show the comparison results and leave the discussions of these results in Section 5.3.

5.3. Discussions

5.3.1. Comparison with ns-2

From the fixed network results shown in Tables 2 and 3, one sees that the memory space used by NCTUns and ns-2 are very close to each other, regardless of the traffic types (CBR UDP or greedy TCP) and whether

Table 4
Performance comparisons between (A) NCTUns, (B) ns-2, and (C) NCTUnsTS

Metric	(A)	(B)	(C)	(A)/(B)	(A)/(C)
CBR packet interval time = 1 s					
With trace file					
Memory usage (KB)	14828	36228	14828	0.409	1
Required time (s)	7	14	59	0.500	0.119
Trace file size (MB)	0.8	31	0.8	0.026	1
Without trace file					
Memory usage (KB)	14680	21932	14680	0.669	1
Required time (s)	5	9	56	0.556	0.089

The generated traffic is CBR UDP with packet interval time being 1 s.

Table 5
Performance comparisons between (A) NCTUns, (B) ns-2, and (C) NCTUnsTS

Metric	(A)	(B)	(C)	(A)/(B)	(A)/(C)
CBR packet interval time = 0.1 s					
With trace file					
Memory usage (KB)	14832	45988	14832	0.323	1
Required time (s)	33	44	87	0.750	0.379
Trace file size (MB)	7	108	7	0.065	1
Without trace file					
Memory usage (KB)	14684	31684	14684	0.463	1
Required time (s)	19	29	73	0.655	0.260

The generated traffic is CBR UDP with packet interval time being 0.1 s.

Table 6
Performance comparisons between (A) NCTUns, (B) ns-2, and (C) NCTUnsTS

Metric	(A)	(B)	(C)	(A)/(B)	(A)/(C)
CBR packet interval time is 0.01 s					
With trace file					
Memory usage (KB)	26052	48628	26052	0.536	1
Required time (s)	637	90	674	7.078	0.945
Trace file size (MB)	115	359	115	0.320	1
Without trace file					
Memory usage (KB)	25272	34324	25272	0.736	1
Required time (s)	395	43	418	9.186	0.945

The generated traffic is CBR UDP with packet interval time being 0.01 s.

the trace file option is enabled/disabled. As for simulation speed, it depends on whether the trace file is generated or not. Therefore, we discuss the size of the generated trace file first. It is noticeable that the size of the trace file generated by ns-2 is more than 10 times larger than that generated by NCTUns. We looked into the format of the ns-2 trace file speculating that it may contain more information than a NCTUns trace file. However, we found that actually the amount of information contained in both trace files are almost equal. We attribute the smaller size of a NCTUns trace file to our use of a highly compact binary format.

Going back to simulation speed comparison, one sees that when the trace file is generated, the performance of NCTUns is close to or even better than that of ns-2. See the (A)/(B) ratios of 1.102 in Table 2 and 0.745 in Table 3. However, when the trace file is not generated, the performance of NCTUns is a few times slower than that of ns-2. See the (A)/(B) ratios of 7.294 in Table 2 and 4.421 in Table 3. This phenomenon can be easily explained by the disk I/O overhead occurred in generating these trace files. Another phenomenon is that NCTUns performs better in the greedy TCP case than in the CBR UDP case. See the (A)/(B) ratios of 7.294 in Table 2 and 4.421 in Table 3. This is because TCP employs congestion control while UDP does not. As such, the amount of TCP packets is limited by the bandwidth of the bottleneck link, reducing the number of packet events that need to be processed per simulated second. In contrast, since the amount of UDP packets is not limited and reduced by the bottleneck link, the UDP simulation case needs more time to finish than the TCP simulation case.

From the wireless ad hoc network results shown in Tables 4–6, one sees that the performance differences between NCTUns and ns-2 are different from those in fixed network cases. First, the memory usage of NCTUns is always smaller than that of ns-2 in all tested cases. See the (A)/(B) ratios of 0.409 and 0.669 in Table 4, 0.323 and 0.463 in Table 5, and 0.536 and 0.736 in Table 6 for this phenomenon. This indicates that NCTUns manages memory space more efficiently than ns-2 for wireless simulations. Second, when the traffic load is light or medium, the simulation speed of NCTUns is better than that of ns-2. However, when the traffic load is heavy, the simulation speed of NCTUns is worse than that of ns-2. See the (A)/(B) ratios of 0.500 and 0.556 in Table 4, 0.750 and 0.655 in Table 5, and 7.078 and 9.186 in Table 6 for this phenomenon. One reason for explaining this phenomenon is that ns-2 has a larger fixed performance overhead than NCTUns for simulating a network case but ns-2 has a less cost of simulating a packet transmission than NCTUns. As such, NCTUns outperforms ns-2 in cases with light or medium traffic load but ns-2 outperforms NCTUns in cases with heavy traffic load. The fixed network performances presented in Tables 2 and 3 also show that the cost of simulating a packet transmission is higher in NCTUns than in ns-2 (see the (A)/(B) ratios of 7.294 in Table 2 and 4.421 in Table 3).

Several reasons can explain why this is the case. First, in NCTUns, real-life protocol stacks are directly executed. In contrast, in ns-2, only their abstractions are executed. Second, in NCTUns, real-life application programs are directly executed to generate realistic traffic. However, in ns-2 simple functions embedded in ns-2 are used to generate artificial traffic. Third, in NCTUns, real data payloads are carried in each transmitted packet and thus they need to be copied. However, in ns-2 only fake data is used. Fourth, in NCTUns, 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, which is not needed in ns-2. Lastly, in NCTUns, since events need to be read out of the event-passing channel (/dev/tun0), a lot of system calls need to be made, which is not needed in ns-2. From these explanations, one sees that performing a few times slower than ns-2 is the cost that NCTUns needs to pay for having several unique advantages over ns-2.

The reported results, however, show that the performance of NCTUns in its current form is still satisfactory. In Section 5.1, one sees that when the traffic load is not too high, the virtual clock of NCTUns still runs much faster than the clock of the real world. Currently, we are working to further improve its performance and believe that its performance can still be improved by using more efficient data structures and algorithms for its event processing. A more effective design change would be to immerse the simulation engine process into the kernel by making it a kernel process. This will avoid a lot of packet copy operations and system calls, resulting in significant simulation performance improvements.

5.3.2. Comparison with NCTUnsTS

Because NCTUns and NCTUnsTS differ only in their simulation engine design, the sizes of their memory usages and the sizes of their generated trace files are the same. We thus focus on their simulation speed differences.

From the fixed network results shown in Tables 2 and 3 and the wireless ad hoc network results shown in Tables 4–6, one sees that NCTUns outperforms NCTUnsTS greatly under light and medium traffic loads and the performance of NCTUns is downgraded to that of NCTUnsTS under heavy traffic loads. See the (A)/(C) ratios of 0.119 and 0.068 in Table 2 and 0.035 and 0.016 in Table 3. Also see the (A)/(C) ratios of 0.119 and 0.089 in Table 4, 0.379 and 0.260 in Table 5, and 0.945 in Table 6 for this phenomenon.

This phenomenon clearly shows the inherent performance advantages and limitations of the event-driven approach. Under light loads (i.e., when packet events are sparse), as the virtual clock of the event-driven approach can usually advance by a large amount of time, a simulation case can be finished more quickly in the event-driven approach than in the time-stepped approach. However, as the traffic load increases (i.e., packet events become denser), the performance gain of the event-driven approach over the time-stepped approach diminishes. This is because the average time amount that the virtual clock of the event-driven approach can advance diminishes as well and eventually approaches the time unit used in the time-stepped approach, resulting in no gain.

The (A)/(C) ratios presented in all of these tables show that for a particular case, the (A)/(C) ratio when the trace file is generated is larger than the (A)/(C) ratio when the trace file is not generated. For example, see the (A)/(C) ratios of 0.119 and 0.089 in Table 4, 0.379 and 0.260 in Table 5, and 0.945 in Table 6. This phenomenon is easy to explain. Because generating trace files incurs disk I/O time, which is accounted in the required time to simulate a case, the performance speedup brought by the event-driven approach will be the best when there is no I/O time accounted in the required time. As such, when the trace file is not generated, the (A)/(C) ratio becomes smaller (which better shows the effectiveness of the event-driven approach).

6. Scalability

Because in NCTUns 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.

6.1. Number of nodes

Because NCTUns simulates multiple routers and hosts by letting their 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, because they are simulated in the simulation engine process, there is no limitation on them either.

6.2. Number of interfaces

In NCTUns, 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, (we have cloned tunnel interfaces, given the cloned interfaces different names, and used them in the same way as one uses the original tunnel interface.), there is no limitation on the maximum number of layer-3 interfaces that can be simulated in a network. Currently, the default number of layer-3 interfaces supported by NCTUns is 4,096 and this number can be increased very easily. For layer-1 and layer-2 interfaces (used in hubs and switches respectively), since they are simulated in the simulation engine process, there is no limitation on them either.

6.3. Number of routing entries

In NCTUns, the kernel routing table needs to store some 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 for simulated routers, if a simulated network has only one subnet and thus no router is needed (e.g., a mobile ad hoc or sensor network), the kernel routing table need not be used for the simulation and can be empty.

In addition, NCTUns supports the "subnet" concept. Therefore, the more efficient subnet-routing scheme can be used instead of the less-efficient host-routing scheme. This greatly reduces the number of required routing entries. 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 [22] 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.

6.4. 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 needs to be present in the physical memory and other inactive code can be swapped out to the disk. 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 is greatly reduced.

Fig. 6 shows the maximum memory space consumption vs. the number of mobile nodes relationship for a simulated mobile ad hoc network. The tested machine is a 3 GHZ Pentium PC with 2 GB main memory. The

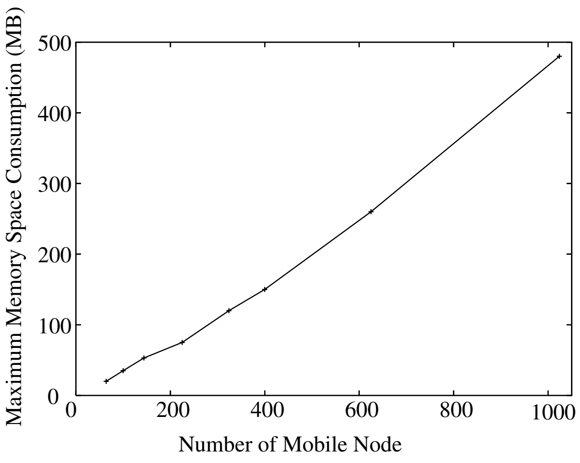

Fig. 6. The maximum memory space consumption vs. the number of mobile nodes.

tested network is a mobile ad hoc network with a varying number of nodes (8×8, 10×10, 12×12, 15×15, 18×18, 20×20, 25×25, and 32×32). Each mobile node has an IEEE 802.11 (b) wireless interface and two real-life application programs are running on top of it to send UDP packets to other nodes and to receive UDP packets from other nodes, respectively. The packet size is 1400 bytes and the network traffic load generated by each node is one packet per second. The time to be simulated is 60 s.

The reported maximum memory space consumption includes the space used by the simulation engine process and that used by all forked real-life application programs. During a simulation, we used the "top" command to constantly monitor the simulation machine's current available memory space. The maximum memory space consumption of a simulation case is the difference between the available memory space before the simulation is run and the minimum available memory space detected during the simulation.

From Fig. 6, one sees that, on average each mobile node and its two real-life application programs occupy about 472 KB of memory space and simulating 1024 such nodes requires about 479 MB of memory space. Nowadays this 479 MB memory space requirement can be met easily given the low memory prices on the current market. Note that the tested machine has 2 GB memory space and the observed maximum memory space consumptions are all less than 2 GB. This means that the reported space requirements represent the requirements when the operating system's virtual memory mechanism is not triggered to save memory space usage. Actually, less memory space may be enough for these cases with the help of the virtual memory mechanism.

6.5. Parallel and distribution simulations

In a simulation case where thousands of nodes need to be simulated and several application programs need to be run on each of them, the required CPU speed for running this huge simulation case and the required memory space for storing the program code of these application programs may inevitably increase to a level that cannot be handled by a single machine. In this situation, turning NCTUns into a parallel and distributed network simulator will be necessary to distribute the CPU and memory loads over multiple machines. Currently, a version using the conservative time synchronization approach is available. We are implementing the optimistic approach to see which version would perform better.

7. Conclusions

This paper presents the novel design and implementation of the NCTUns network simulation engine. By combining the kernel re-entering simulation methodology and the discrete-event simulation methodology, NCTUns provides unique advantages over traditional network simulators without sacrificing simulation performances. The proposed design and implementation are novel, efficient, simple to implement, and can be applied to other similar simulation environments as well.

References

[1] OPNET Inc. (<http://www.opnet.com>).
[2] S. McCanne, S. Floyd, ns-LBNL Network Simulator. (<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, March 21–25, New York, USA, 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] S.Y. Wang, Harvard TCP/IP network simulator 1.0, available at <http://www.eecs.harvard.edu/networking/simulator.html>.
[6] S.Y. Wang, C.L. Chou, C.H. Huang, C.C. Hwang, Z.M. Yang, C.C. Chiou, C.C. Lin, The design and implementation of the NCTUns 1.0 network simulator, Computer Networks 42 (2) (2003) 175–197.
[7] Richard M. Fujimoto, Parallel and Distributed Simulation Systems, John Wiley & Sons, Inc., 2000.
[8] Mark Carson, Darrin Santay, NIST Net – A Linux-based network emulation tool, ACM Computer Communication Review 33 (3) (2003).
[9] F. Kevin, Network Emulation in the Vint/NS simulator, ISCC99, July 1999.
[10] Jong Suk Ahn, P. Danzig, Zhen Liu, Limin Yan, Evaluation of TCP Vegas: Emulation and Experiment, ACM SIGCOMM'95.
[11] L. Rizzo, Dummynet: A simple approach to the evaluation of network protocols, ACM Computer Communication Review 27 (1) (1997) 31–41.

[12] X.W. Huang, R. Sharma, S. Keshav, The ENTRAPID protocol development environment, IEEE INFOCOM'99, March 21–25, New York, USA, 1999.
[13] A. Meyer, L.H. Seawright, A virtual machine time-sharing system, IBM Systems Journal 9 (3) (1970) 199–218.
[14] VMware: Enterprise-class Virtualization Software, <http://www.vmware.com>.
[15] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasadm, M. Newboldm, M. Hiber, C. Barb, A. Joglekar, An integrated experimental environment for distributed systems and networks, in: Proceedings of the Fifth Symposium onOperating Systems Design and Implementation, USENIX Association, Dec 2002, pp. 255–270.
[16] Grenville Armitage, Maximizing student exposure to networking using FreeBSD virtual hosts, ACM Computer Communication Review 33 (3) (2003).
[17] Florian Baumgartner, Torsten braun, Eveline Kurt, Attila Weyland, Virtual routers: A tool for networking research and education, ACM Computer Communication Review 33 (3) (2003).
[18] S. Keshav, REAL: A Network Simulator, Technical Report 88/472, Dept. of Computer Science, UC Berkeley, 1988.
[19] SSF network module (SSFnet), available at <http://www.ssfnet.org>.
[20] S.Y. Wang, K.C. Liao, Innovative network emulations using the NCTUns tool, in: Computer Networking and Networks, Nova Science Publishers, 2006, ISBN 1-59454-830-7.
[21] S.Y. Wang, Y.B. Lin, NCTUns network simulation and emulation for wireless resource management, Wiley Wireless Communication and Mobile Computing 5 (8) (2005) 899–916.
[22] Gary R. Wright, W. Richard StevensTCP/IP Illustrated, Vol. 2, Addison Wesley, 1995.