A New Methodology for Easily Constructing Extensible and High-Fidelity TCP/IP Network Simulators

S.Y. Wang

shieyuan@csie.nctu.edu.tw
Department of Computer Science and Info. Engr.
National Chiao Tung University
Hsinchu, Taiwan

H.T. Kung

kung@harvard.edu
Division of Engineering and Applied Sciences
Harvard University
Cambridge, MA 02138, USA

Abstract

This paper proposes a new methodology for easily constructing extensible and high-fidelity TCP/IP network simulators. The methodology uses a kernel-reentering technique to reuse the existing real-life network protocol stacks, real application programs that generate traffic, and real utility programs that configure, monitor, or gather network statistics to the maximum extent. Only an event scheduler and some modifications to the kernel are needed to "glue" these existing components to collectively simulate a network.

A simulator constructed this way has many advantages that a traditional network simulator cannot provide. First, reuse of real-life implementation in the simulator can generate more accurate results than a traditional simulator that abstracts a lot of away from the real implementation. Second, it can save much time and effort that would be needed if a high-fidelity simulator is developed from scratch. Third, because real application programs cannot distinguish a simulated network constructed by the simulator from a real one, all existing real-life and future application programs can directly run on any node in a simulated network.

1. Introduction

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

Network simulators, however, have their limitations. A complete network simulator not only needs to simulate hosts and routers, it also needs to simulate application programs that generate network traffic. It also needs network utility programs that configure, monitor, or gather statistics about a simulated network. As such, developing a complete network simulator is a large effort and traditional network simulators tend to have the following drawbacks due to limited development resources:

- Simulation results are usually not as convincing as those produced by real hardware and software equipment. In order to constrain their complexity and development cost, most existing network simulators can only simulate real-world network protocol implementations with limited detail, and this may generate wrong results in some situations.

  For example, in ns [2], there are many differences between the real-life and the simulation implementation of the TCP/IP stack. First, in ns a TCP connection must use a fixed length for all of its packets (because there is no real application programs running to exchange data). However, in real networks, this is unnecessary. Second, in ns IP fragmentation is not handled. However, in real networks, IP fragmentation may occur. Third, in ns the receiver of a TCP connection does not implement the dynamic advertised window mechanism (because there is no real application program running to use the received data). However, in real network usages, TCP receivers always use the dynamic advertised window mechanism to implement flow control between the TCP sender and receiver.

The NCTUns 1.0 network simulator is a new and much more powerful simulator than the simulator presented in this paper. Information about this new simulator is available at http://NSL.csie.nctu.edu.tw/nctuns.html.

- These simulators are not extensible in the sense that they lack the UNIX system call (POSIX) application programming interface (API). As such, existing or to-be-developed real application programs cannot be run normally in separate address space to generate traffic on nodes in a simulated network. Instead, in order to interact with the simulator, 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 program.

  This causes the following two problems. First, these network simulators are limited to the study of only network-level performance such as link utilization, packet drop rate, etc. Application-level performance of a real distributed application program (e.g., the response time of a distributed database system when running on a particular network configuration) cannot be studied. However, a system designer or network planner may need to know whether a given network topology and associated link capacities can provide reasonable application-level performance. Indeed, some commercial simulation systems have been developed to meet this application; see e.g. [3]. Second, the lack of UNIX POSIX API prohibits the use of these network simulators in areas where user-developed real programs need to run on nodes to carry out tasks cooperatively. Examples of these areas include intelligent mobile agents [4] and Mobile IP [5].

In this paper, we propose a simple simulation methodology that alleviates these drawbacks. A simulator constructed under this methodology has three desirable properties as follows. First, it reuses the real-life UNIX TCP/IP protocol stack, existing real network application programs, and existing real network utility programs. As a result, it can generate more accurate simulation results than a traditional TCP/IP network simulator that abstracts a lot away from a real-life TCP/IP implementation. Second, it provides the UNIX POSIX API (i.e., the standard UNIX system call interface) on every node in a simulated network. Any real UNIX application program, either existing or to be developed, thus can run on any node in a simulated network to generate traffic. One important advantage of this property is that since such a developed application program for simulations is a real UNIX program, the program's simulation implementation can be its real implementation on an UNIX machine. As a result, when the simulation study is finished, we can quickly implement the real system by reusing its simulation implementation. Third, the simulator can be easily constructed with minimal time and effort. By reusing existing code to the maximum extent, this methodology enables a network researcher to easily construct his/her own TCP/IP network simulator.

A simulator constructed under our methodology has been operational for several years. Its simulation results have been validated extensively against results obtained from real hardware, and shown to correctly reflect TCP/IP network behaviors. (For example, the simulation results presented in [6] were all confirmed by real experiment results and more validation results are stored in our database available at http://NSL.csie.nctu.edu.tw/NSL_DATA.) On July 1, 1999, this simulator was released for the public and was named Harvard TCP/IP network simulator 1.0 [7]. Since that time, as of August 1, 2001, more than 1,000 universities, research institutes, industrial research laboratories, and ISPs have downloaded the simulator.

2. Related Work

In the literature, some approaches also use a real-life TCP/IP stack to generate results [8, 9, 10, 11]. However, unlike our approach, these approaches are used for emulation purposes, rather than for simulation purposes. Among these approaches, Dummynet [11] most resembles our simulator. Both Dummynet and our simulator use tunnel interfaces to use the real-life TCP/IP protocol stack on the simulation machine. However, there are some fundamental differences. Dummynet uses the real time, rather than the simulated network's virtual time. Thus the simulated link bandwidth is a function of the simulation speed and the total load on the simulation machine. As the number of simulated links increases, the highest link bandwidth that can be simulated decreases. Moreover, in Dummynet, routing tables are associated with incoming links rather than nodes. Thus, the simulator will not know how to route packets generated by a router, since they do not come from any link

OPNET [3], REAL [12], ns [2], SSFnet [13], PARSEC and GloMoSim [14] represent the traditional network simulation approach. In this approach, the thread-supporting event scheduler, application programs that generate network traffic, utility programs that configure, monitor, or gather statistics about a simulated network, the TCP/IP protocol implementation on hosts, the IP protocol implementation on routers, and links are all compiled together to form a single user-level program. Due to the enormous complexity, such a simulator tends to be difficult to develop and verify. A simulator constructed using this approach cannot provide UNIX POSIX API for real application programs to run in separate address space as they normally do on a real host. Although some simulators may provide their own internal API, real application programs still need to be rewritten so that they can use the internal API, be compiled with the simulator successfully, and be concurrently executed with the simulator during simulation.

ENTRAPID [15] uses another approach. It uses the virtual machine concept [16] to provide multiple virtual kernels on a physical machine. Each virtual kernel is a process and simulates a node in a simulated network. The system calls issued by an application program are redirected to a virtual kernel. As such, UNIX POSIX API can be provided by ENTRAPID and real application programs can be run in separate address space as normal. Because the complex kernel needs to be ported to and implemented at the user level, many involved subsystems (e.g., the file, disk I/O, process scheduling, inter-process communication, virtual memory subsystems) need to be modified extensively. As a result, the porting effort is very large and the correctness of the ported system need to be extensively verified.

The contribution of this paper is that we propose a *new* methodology that can *easily* construct network *simulators* with many *advantages*.

3. Simulator Architecture

Our simulator architecture differs from traditional ones in how it integrates the various components that implement the following functions in a simulated network:

1. Links with various delays and bandwidths
2. Routers that forward IP packets
3. Hosts that use TCP/UDP/IP protocol to send and receive packets
4. Application programs that generate network traffic
5. Network utility programs that configure, monitor, or gather statistics about a network

Unlike traditional approaches such as REAL [8] and ns [2], our simulator architecture does not need to combine component 1, 2, 3, 4, and 5 together to form a single program. Instead, a simulator constructed under our methodology has separate and independent components for 1, 2, 3, 4, and 5. When these components run concurrently on an UNIX machine, collectively their executions simulate a network. Since the constructed system is intended to be used as a simulator, the time used in the system is virtual time, rather than real time. Section 5.1 will give the details.

In addition, our simulator only needs to simulate links (component 1), and the complicated components 2, 3, 4, and 5 need not be simulated. In a simulated network, these complicated components' existing real-life code and programs are directly used in the same way as they are used in a real network. In contrast, a traditional network simulator needs to simulate all of these components in a single user-level program and therefore cannot simulate these components in great detail. Some traditional simulators simulate only an abstract of the real-life TCP/IP protocol implementation. Others port the in-kernel real-life TCP/IP protocol implementation to the user level trying to increase simulation fidelity. However, because many involved kernel subsystems need to be carefully removed (e.g., the mbuf buffer system, which is extensively used in the BSD UNIX TCP/IP protocol stack), the ported version usually differs from the real-life implementation and its behavior may be different from the original behavior.

In the rest of this section, we will describe the key ideas and techniques that make our simulator architecture feasible.

3.1. Tunnel Network Interface

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

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

Using tunnel network interfaces, we can easily simulate a single-hop TCP/IP network depicted in Figure 1 (a), where a TCP sender on host 1 is sending its TCP packets to a TCP receiver on host 2. We set up the virtual simulated network by performing the following two steps. First, we configure the kernel routing table of the simulation machine so that tunnel network interface 1 is chosen as the outgoing interface for the TCP packet sent from host 1 to host 2, tunnel network interface 2 for the TCP packets sent from host 2 to host 1. Second, for each of the two links to be simulated, we run an application program (called "virtual

link" object here) to simulate it. For the link from host i to host j (i = 1 or 2, j = 3 - i), the application program would open tunnel network interface i's and j's special file in /dev and then execute an endless loop. In each step of this loop, it simulates a packet's transmission on the link by reading a packet from the special file of tunnel interface i, waiting the link's propagation delay time plus the packet's transmission time on the link, and then writing this packet to the special file of tunnel interface j.

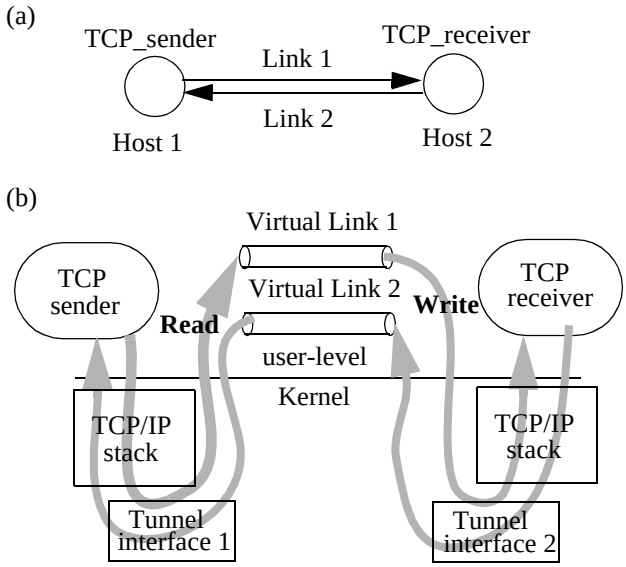

Figure 1: (a) A TCP/IP network example to be simulated. (b) By using tunnel interfaces, only the two links need to be simulated. The complicated TCP/IP protocol stack need not be simulated. Instead, the real-life working TCP/IP protocol stack is directly used in the simulation.

After performing the above two steps, the virtual simulated network has been constructed. Figure 1 (b) depicts this simulation scheme. Since our trick of replacing a real link with a simulated link happens outside the kernel, the kernels on both hosts do not know that their packets actually are exchanged on a virtual simulated network. The TCP sender and receiver programs, which run on top of the kernels, of course do not know the fact either. As a result, all existing real network application programs can run on the simulated network, all existing real 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.

By using the kernel-reentering technique presented in Section 4, the kernel of the simulation machine is shared by all nodes (hosts and routers) in a virtual simulated network. Therefore, although in Figure 1 (b) there are two TCP/IP protocol stacks depicted, actually they are the same one -- the protocol stack of the single simulation machine.

3.2. Opaque and Transparent Network Cloud Simulation Models

To extend simulated networks from single-hop networks as shown in Figure 1 (a) to multi-hop networks, we need to simulate an additional object type -- intermediate routers. A traditional way of simulating a network composed of links and routers is to simulate them in a user-level program. We call a simulated network formed this way an "opaque network cloud." It is "opaque" because the kernel can not see through the network cloud. As Figure 2 (a) illustrates, once a packet is injected into an opaque network cloud, it will be covered by the opaque network cloud when it traverses through the routers on the way to its destination host. The kernel of the simulation machine cannot see this packet because the packet will not enter and leave the kernel again until it finally reaches its destination host. OPNET Modeler [3] and ns [2] are examples of those network simulators that use the opaque network cloud simulation model.

In contrast, when a packet arrives at a router, our methodology simulates the packet forwarding operation by performing the following three steps. First, the virtual link object from which the arriving packet comes writes the packet into the kernel. Second, the kernel automatically forwards the packet toward the correct direction (i.e., puts it into the correct output port's queue). (This routing step can be done automatically because any UNIX host in real life can also function as a router.) Third, the virtual link object of the next hop pulls this packet out of the kernel (i.e., fetch it from the output port's queue) and then simulates its transmission on the next hop. We call a simulated network formed this way a "transparent network cloud." "Transparent" here means that, as shown in Figure 2 (b), after a packet is injected into the network cloud, the packet will go down (enter the kernel) and go up (leave the kernel) when passing through each router on the way to its destination host. Thus the kernel will see the packet when it traverses through the network cloud.

A network simulator that uses the transparent network cloud model has many advantages over one that uses the opaque model. First, since the real-life working protocol stack is used in routing and forwarding a packet, there is no need to spend time and effort on porting the in-kernel routing protocol stack to a user-level program to simulate routers. Second, because the unmodified real-life UNIX routing protocol stack is used, simulation results are more credible than those generated otherwise. Third, the standard UNIX system call interface (API) is exposed and supported on every node. All real application programs that can run on hosts can now run on routers as well.

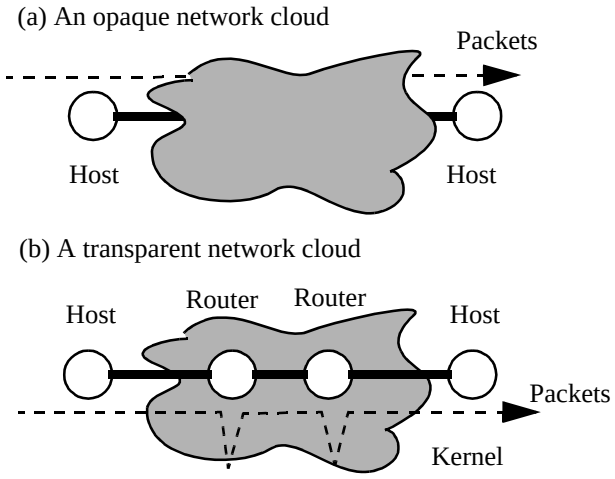

Figure 2: Opaque and transparent network clouds differ in whether or not packets traversing through routers in the simulated network are "visible" to the kernel of the simulation machine.

Figure 3 (b) and (c) illustrate the differences between two simulated networks that are based on the opaque and transparent network cloud simulation models, respectively. Both of them are constructed to simulate the same network in (a).

4. Design

This section presents our addressing, routing, and address-remapping schemes that are designed to support the transparent network cloud simulation model. As described earlier, the single simulation machine will act as both hosts and routers during a simulation. Using the network of Figure 3 (a) as an example, we illustrate the operations occurred on the simulation machine when a packet is sent across the network from node 1 to node 2. As depicted in Figure 4, the operations involve a sequence of leaving and entering the kernel operations. That is, to forward a packet along the path from the TCP sender to the TCP receiver in Figure 3 (a), the packet will leave the kernel along link 1 and then re-enter it, leave the kernel along link 3 and then re-enter it, and finally, leave the kernel along link 5 and then re-enter it.

To route packets, the simulation machine needs to maintain a routing table. Since the simulation machine simulates all the routers and hosts in a simulated network, by using the union of the routing tables in all of the nodes, one may think that the simulation machine should be able to route packets correctly. However, when unified together, these routing tables contain conflicting information. For example, in the network of Figure 3 (a), when forwarding a packet destined to host 2, host 1 will choose link 1 as its next hop, router 1 will choose link 3 as its next hop, and router 2 will choose link 5 as its next hop. If all these conflicting (destination IP address, next hop) pairs are stored in the single simulation machine's routing table, the kernel cannot choose the correct next hop for forwarding a packet.

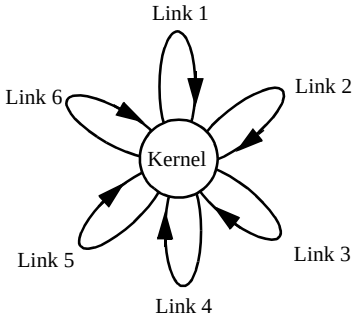

Figure 4: Routing a packet along a route in the simulated network of Figure 3 (c) is a sequence of leaving and entering the kernel operations applied to the simulation machine.

One solution to this problem is to have a separate routing table in the kernel for every node in the simulated network. Each packet re-entering the kernel then tells the kernel which node it should simulate at this time. For example, when passing through router 2, the packet will tell the kernel that now the kernel should simulate router 2. The kernel then uses this information to look up the corresponding routing table to retrieve the correct routing entry.

Although the above method can solve the route conflict problem, it is not our preferred solution for the following reasons. First, maintaining a separate routing table in the kernel for every node of a simulated network is not the standard mechanism used by an UNIX machine. In order to work around the route conflict problem, we will need to extensively modify the kernel code that is related to routing and forwarding. This violates our goal of minimizing modification to a real-life network protocol stack. Second, using a different routing mechanism means that we no longer can use many existing utilities, such as "route," to configure routes. Many new utility programs would have to be developed to adapt to the changes.

Our preferred solution is to use a special address-remapping and route-setup scheme so that using the default kernel routing table does not result in the route conflict problem. The basic idea is to remap the destination IP address of a packet to a new one before it arrives at and is forwarded by a router. We call the mapped version of an IP address on node i as this IP address's **"As-Seen-By-Node(i)"** address. Because our address-remapping scheme

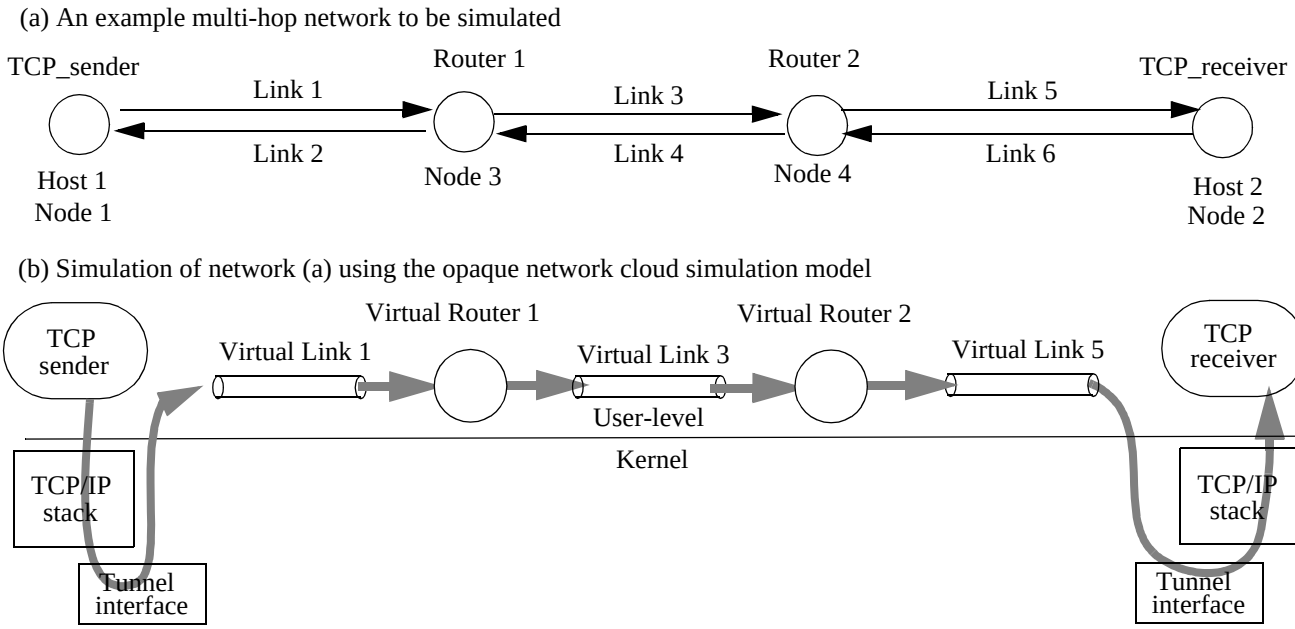

Figure 3: (a) A simple multi-hop network to be simulated; (b) A simulation using the opaque network cloud simulation model;. (c) A simulation using the transparent network cloud simulation model; and (d) The IP address remappings occurred in the simulation. Notice that for presentation simplicity, links 6, 4 and 2 are not shown in (b) and (c).

guarantees that none of these **"As-Seen-By-Node(i)"** addresses will be the same, the route conflict problem will not happen. Figure 3 (d) illustrates the address-remapping idea. In the following sections, we will explain our addressing, address-remapping, and routing schemes in detail.

4.1. Use a Private Address Scheme for a Node's (Normal) IP Address(es)

Like the usual practice over the Internet that each node has one IP address associated with each of its network interfaces, we assign one IP address to each network interface of a node. These addresses are called "normal" in the rest of the paper to distinguish them from the **"As-Seen-By-Node(i)"** addresses, which will be defined in Section 4.3. Because one-way simplex links are supported in a simulated network, some IP addresses of a node may be associated with the node's incoming interfaces while the others associated with its outgoing interfaces. In our scheme, only those IP addresses that are associated with outgoing interfaces are used to reference nodes. For example, when we want to send a packet to a node j, we will use one IP address of node j that is associated with one of node j's outgoing interface to be the packet's destination address. We will not use an IP address that is associated with an incoming interface to reference a node.

The network address of a simulated network in our system is "192.168.0.0". "192.168." was chosen because it is within the experimental address space defined in RFC 1918. Using it assures that the simulation machine can still be connected to a real network without interfering with other networks during the simulation. For simplicity, however, in the rest of this paper, we will use "1.1.0.0" as the network address of a simulated network. Every simplex link is defined as a subnet in the simulated network and has "1.1.Link_ID.0" as its sub-network address, where Link_ID is its link identity.

Consider a simplex link from node A to node B. We define the IP address of node A on this link as "1.1.Link_ID.NodeA_ID", where NodeA_ID is the identity of node A in the simulated network. Although the IP address of node B on this link will not be used in the simulation (because this link is node B's incoming link), due to the requirement that we must specify a local and a remote address when configuring a tunnel network interface, we still define node B's address as "1.1.Link_ID.254". Arbitrarily chosen, "254" will hereafter remind us that an IP address ending with "254" should not be used to reference a node. The IP addresses of all nodes on all links in a simulated network are assigned using the same scheme described here. As explained earlier, if a node has multiple links, only its IP addresses that are associated with its outgoing simplex links are used. Therefore, IP addresses in the form of "1.1.Link_ID.254" will never be used to reference a node. For example, host 1 in Figure 3 (a) will have two IP addresses. The first address, "1.1.1.1", will be used while the second, "1.1.2.254", will not be used. For the same reason, router 1 in Figure 3 (a) will have four IP addresses, but only "1.1.3.3" and "1.1.2.3" will be used to reference it.

4.2. Construct the Virtual Simulated Network

We then use these defined IP addresses to configure tunnel network interfaces on the simulation machine to construct a virtual simulated network. The way we configure a simulated network is exactly the same way a person configures a real network composed of nodes and point-to-point links. A tunnel network interface, which is just like a SLIP or PPP network interface for a point-to-point link, is used for a one-way simplex link in the simulated network. We associate tunnel network interface i with link i so that its local IP address and its remote IP address are "1.1.i.SourceNode_ID" and "1.1.i.254', respectively, where SourceNode_ID is the node identity of simplex link i's source node.

4.3. Define a Node's IP Address(es) Seen from Other Nodes

Although we have defined a node's (normal) IP address(es), these addresses are only useful in helping us construct a virtual simulated network and reference nodes. To solve the route conflict problem discussed in Section 4, we need to define and introduce the concept of the "As-Seen-by-Node(i)" IP address of a node's IP address, where i is an index variable. The "As-Seen-by-Node(i)" IP address of node j's IP address is the address that node i should use when sending a packet to node j. In our methodology, we define a node's "As-Seen-by-Node(i)" IP address, where i is the node identity of any other node in the simulated network, as follows:

```
Suppose "1.1.Link_ID.Node_ID" is one of a node's
IP addresses as defined in Section 4.1, then the
"As-Seen-by-Node(i)" IP address of this address is
"1.1.Link_ID.i".
```

As defined above, a node, in addition to its (normal) IP address(es), has a unique "As-Seen-by-Node(i)" IP address for node i in the simulated network. For example, in Figure 3(a), node 2's "As-Seen-by-Node(1)" IP address is "1.1.6.1", its "As-Seen-by-Node(3)" IP address "1.1.6.3" and its "As-Seen-by-Node(4)" IP address "1.1.6.4". In our methodology, if node i wants to send a packet to node j, it uses node j's "As-Seen-by-Node(i)" IP address as its packet's destination IP address so that its packet can be

correctly routed in the simulated network. The next session explains how this scheme works.

4.4. Use "As-Seen-by-Node(i)" IP Addresses to Route Packets

Using "As-Seen-by-Node(i)" IP addresses enables us to correctly route packets without the route conflict problem discussed in Section 4. This is achieved because even if there are multiple sending nodes that all want to send packets to the same destination node j, these packets' destination IP addresses will be different from each other. In fact, these sending nodes will all use node j's "As-Seen-by-Node(i)" IP addresses, and i will be different for different sending nodes. Since we no longer can find two routing entries in the simulation machine's routing table that have the same destination IP address but different next hops to forward a packet, the route conflict problem is solved.

To set up routes for every pair of nodes, the routes for all possible "As-Seen-by-Node(i)" IP addresses in a simulated network should be set up. This can be easily accomplished by using the existing user-level "route" utility provided on an UNIX machine.

If node i has only one link (i.e., it is a host), setting its routes to all other nodes' "As-Seen-by-Node(i)" IP addresses is simple. We only need to set its outgoing link (thus the link's associated tunnel network interface) as the next hop for these IP addresses. If node i has multiple links connecting it to the rest of the network (i.e., it is a router), the selection of the next hop for a "As-Seen-by-Node(i)" IP address actually depends on the desired routing algorithm. For example, we can use routes generated by a shortest-distance-vector or policy routing algorithm to configure the routes for the entire simulated network.

Using "As-Seen-by-Node(i)" IP addresses and setting routes for these IP addresses effectively achieve the following two functions -- storing each node's routing table separately in the kernel and forwarding packets using their own correct routing tables. The reason is that when we set the next hop for node j's "As-Seen-by-Node(i)" IP address, we are essentially adding a route from node i to node j into node i's own routing table.

4.5. Modify A Packet's "As-Seen-by-Node(i)" Destination and Source Addresses on Every Hop Along Its Path

Suppose that before a packet arrives at its final destination node j, it arrives at node i. In our methodology, the kernel can forward the packet to the correct next hop (link) if the packet's destination IP address has been set to the "As-Seen-by-Node(i)" address of node j's IP address. In order to continuously route a packet across multiple nodes toward its destination node, this packet's destination IP address must be modified on every hop along its path.

More specifically, before a packet arrives at node i, we must change its destination IP address from its current one to the "As-Seen-by-Node(i)" address of its final destination node's IP address. This change can be easily done because we only need to change the least significant byte of this packet's destination IP address to i. For example, in Figure 3 (d), when a packet is routed from node 1, via node 3 and node 4, to node 2, its destination IP address in the IP header will be "1.1.6.1", "1.1.6.3", "1.1.6.4" and "1.1.6.2", respectively. Note that by using this scheme, when a packet arrives at its final destination node, the destination address in its IP header is the same as its destination node's (normal) IP address. This property is important because it enables the kernel to decide when to stop forwarding the packet. In Section 6.2, we will explain this property in detail.

Sometimes a node j in a simulated network may want to send reply packets back to node i after having received a packet from node i. For example, during a TCP connection's connection establishment phase, when a server receives a SYN packet from a client, it will reply with a SYN+ACK packet to that client to continue the three-way handshaking procedure. Also, in an ICMP application (e.g., traceroute), a node in the simulated network may want to send an error message (e.g., TTL expired) back to a packet's source node. Therefore, in order for a node to directly use the source IP address in a received packet's header to send back a reply packet, before this packet arrives at node i, both its destination and source IP addresses should be changed to their corresponding "As-Seen-by-Node(i)" addresses.

Since before a packet arrives at a node, it is transmitted on a link, it is natural for virtual link objects in the simulated network to modify the source and destination address. In our methodology, when a virtual link object is created to simulate a one-way simplex link, the parameters associated with it, besides standard ones such as bandwidth and delay, contain the identity (assume i for the following discussion) of this link's destination node. With this information, when receiving a packet, this link can change the least significant bytes of the packet's source and destination IP addresses to i before delivering it to the link's destination node.

5. Implementation

5.1. User-Level Event Scheduler

Our system only needs to simulate one kind of object -- the virtual link objects that simulate links. Hosts and routers need not be simulated because the simulation machine acts as them in a simulation. Due to this property, our TCP/IP network simulator is just a user-level event scheduler. (Section 7.1 presents the implementation of an in-kernel version, which greatly speeds up simulation.) This event scheduler simulates links simply by scheduling a time for a link to read a packet from the link's source node (in order to simulate the previous packet's transmission time) and a time for a link to deliver a packet to the link's destination node (in order to simulate the link propagation delay).

Another task of the event scheduler is to pass the simulated network's virtual time down into the kernel so that the timers of TCP connections in the simulated network can be triggered by the virtual time rather than by the real time. Although this can be easily implemented by periodically calling a system call to pass the current virtual time into the kernel, we implement this by mapping the memory location that stores the current virtual time in the event scheduler to a memory location in the kernel. As such, at any time the virtual time in the kernel is as precise as that maintained in the event scheduler without any system call overhead.

The virtual time in the event scheduler is maintained by a counter. The time unit represented by one tick of the counter can be set to any value as small as we would like (e.g., one nanosecond) to simulate high speed links. The current virtual time thus is the current value of the counter times the used time unit. During simulation, the counter is periodically advanced by one as soon as all events that need to be processed at current virtual time have been processed.

Events are generated in many situations. One situation is that it is time for the first packet in a tunnel interface's output queue to enter the simulated network. In each time unit, the event scheduler checks a bit map to see which tunnel interfaces have a packet to enter the simulated network at this time. (We use the memory-mapping technique to map the in-kernel bit map to a map maintained in the event scheduler. Therefore the overhead for checking this bit map in every time unit is insignificant.) When it is time for a packet in a tunnel interface to enter the simulated network, an event is generated and immediately executed. Executing this event will remove this packet from that tunnel interface's output queue and generate more events to be executed in the future. One of these events is used to simulate the link propagation delay that this packet should experience before reaching the other end of the link. Another event is used to simulate the packet's transmission time so that during this period of time the same tunnel interface cannot inject any more packet into the simulated network.

An event is triggered when the current virtual time becomes greater than its timestamp. All events are precisely scheduled and triggered based on the virtual time of the simulated network. For this reason, simulation results are not affected by other activities on the simulation machine (e.g., disk I/O and network I/O).

5.2. Kernel

Some parts of the kernel need to be modified to support our simulator.

IP and UDP/TCP Checksum Tests: Because in simulation the source and destination IP addresses of a packet will change on every hop (see Section 4.5), the checksums in the IP and UDP/TCP headers of the packet will be incorrect and should not be checked. Skipping these checksum tests will not affect the data integrity of packets in our simulator because all packets in our simulated network, in fact, never leave the simulation machine. For situations where we need to simulate a corrupted packet, we can simply set a flag in its IP header to indicate it. Nodes in the simulated network can then detect the corruption and discard the packet.

TCP Timers: TCP slow and fast timers should be modified so that they are triggered based on the virtual time of the simulated network rather than the real time. If we would use the real time, a TCP connection's re-transmit (slow) timer would prematurely expire k times earlier than when it should do, if a simulated network is k times slower than the real network.

Virtual Clocks: In the real world, each machine's clock may be different from others' due to clock drift and skew. We should simulate this phenomenon to avoid multiple TCP connections to drop packets, time out, and "slow-start" in a lock-step manner. For this purpose, we maintain a separate virtual clock for each node in a simulated network. These virtual clocks are offset by some random time.

Process Scheduler: The default UNIX process scheduler is modified so that the processes of the event scheduler and all launched traffic generators are scheduled in a controlled way. The default UNIX process scheduler uses a priority-based dynamic scheme to schedule runnable processes. As such, the order in which the event scheduler and traffic generator processes are scheduled cannot be precisely controlled. Also, the CPU cycles allocated for

each of these processes cannot be guaranteed. This may result in potential problems. For example, after getting the control of CPU, the event scheduler may use the CPU too long before releasing it to traffic generators. Because the event scheduler advances the virtual time while it is executing, if it monopolizes the CPU too long, no network traffic can be generated during this long period of time, which should not occur. To avoid this potential problem, we modified the default UNIX process scheduler so that the event scheduler process and all runnable traffic generator processes are scheduled explicitly in a round-robin manner.

5.3. Application

With our run-time system and kernel support, an application program can work with our simulator without any modification.

Associate an Application's TCP Socket(s) with the ID of the Node on Which It Runs. If a TCP application program runs on node i, its TCP timers should be triggered based on node i's virtual clock. We achieve this requirement by modifying the kernel and using a run-time system. In our simulator, launching an application program must be performed by our run-time system. After obtaining the process ID of the just launched application program, the run-time system immediately calls a system call to pass the node_ID information to the kernel. Inside the system call, the node_ID information will be stored in the process's process control block. This information will later be used when a TCP control block (socket) is created for this process.

Use the Simulated Network's Virtual Time. When an application program reports data related to time, it should use the simulated network's virtual time, rather than the real time. Examples include "ping," which reports a packet's round-trip time, and "ftp," which reports the throughput of a file transfer. Another example is an application program (traffic generator) that uses time information to generate a particular traffic pattern (e.g., constant-bit-rate or Poisson arrival). This requirement can be easily handled because our run-time system has the process IDs of all launched application programs and will pass these process IDs into the kernel. The kernel's implementation of time-related system calls such as (gettimeofday(), sleep(), alarm(), etc.) is modified so that each of these system calls will check whether the process that issues this system call is one of these launched processes. If so, it returns the current virtual time. Otherwise, it returns the default current real time.

6. Configuration and Usage Examples

We use the example network depicted in Figure 3 (a) to illustrate the configuration and usage of our simulator. The example considers a TCP sender and a TCP receiver connected through a cascade of routers. In the appendix at the end of this paper, we consider a more complex network topology (Figure 6).

6.1. Network Configuration

All links in the network of Figure 3 (a) are 10 Mbps links. The propagation delay of link i is set to be i ms. This configuration allows us to easily test whether our simulator can correctly simulate links with various delays.

6.2. Tunnel Network Interface Configuration

The commands used to configure the six tunnel network interfaces for link 1, 2, 3, 4, 5, and 6, respectively, are shown as follows:

```
ifconfig tun1 1.1.1.1 1.1.1.254 netmask 0xffffff00
ifconfig tun2 1.1.2.3 1.1.2.254 netmask 0xffffff00
ifconfig tun3 1.1.3.3 1.1.3.254 netmask 0xffffff00
ifconfig tun4 1.1.4.4 1.1.4.254 netmask 0xffffff00
ifconfig tun5 1.1.5.4 1.1.5.254 netmask 0xffffff00
ifconfig tun6 1.1.6.2 1.1.6.254 netmask 0xffffff00
```

A tunnel network interface resembles a SLIP or PPP network interface because both are used for a point-to-point link. What the first "ifconfig" command states is that tunnel network interface 1 ("tun1") will be used for a link whose local IP address is "1.1.1.1" and whose foreign address is "1.1.1.254". After executing these commands on an UNIX machine, host 1's (normal) IP address in the simulated network is "1.1.1.1", host 2's (normal) IP address is "1.1.6.2", router 1's (normal) IP addresses are "1.1.2.3" and "1.1.3.3", and finally router 2's (normal) IP addresses are "1.1.4.4" and "1.1.5.4".

To save a user's time and effort, our simulator provides a program that reads in the network configuration file and automatically generates the required tunnel interface configuration commands.

Note that because the single simulation machine acts as both routers and hosts in the simulated network, it is important for our simulator to know when to forward and when to stop forwarding a packet. Together, the following three properties enable the kernel to know when to stop forwarding a packet:

- First, as stated in Section 4.5, when a packet arrives at its destination node, the destination address in its IP header, although it may have been changed several times, will be its destination node's (normal) IP address.

- Second, a node's (normal) IP address is the local address of one configured tunnel network interface.
- Third, the local address of any configured tunnel interface is one of the simulation machine's IP addresses that can be used in the real world. Other machines in a real world network can use any of these configured tunnel interfaces' local addresses to send a packet to this simulation machine. For this reason, when a packet is received, whether from a physical network or from a simulated network, if its destination address is one of these tunnel interfaces' local addresses, the kernel will think that this packet is destined for itself and thus will stop forwarding the packet. The kernel then delivers the packet through the TCP/IP protocol stack to an application program.

6.3. Route Configuration

The commands used to configure the routes for the simulated network are as follows:

```
route add 1.1.2.1 -interface tun1
route add 1.1.3.1 -interface tun1
route add 1.1.4.1 -interface tun1
route add 1.1.5.1 -interface tun1
route add 1.1.6.1 -interface tun1

route add 1.1.1.2 -interface tun6
route add 1.1.2.2 -interface tun6
route add 1.1.3.2 -interface tun6
route add 1.1.4.2 -interface tun6
route add 1.1.5.2 -interface tun6

route add 1.1.1.3 -interface tun2
route add 1.1.4.3 -interface tun3
route add 1.1.5.3 -interface tun3
route add 1.1.6.3 -interface tun3

route add 1.1.1.4 -interface tun4
route add 1.1.2.4 -interface tun4
route add 1.1.3.4 -interface tun4
route add 1.1.6.4 -interface tun5
```

The purpose of the first block of these commands is to set routes for host 1. What the first "route" command states is that any packet whose destination IP address is "1.1.2.1" should be sent out through tunnel network interface 1 ("tun1"). The second block is for setting routes for host 2. The third block is for setting routes for router 1. The last block is for setting routes for router 2. Clearly, configuring routes for host i follows a simple and regular pattern. Let's assume that host i uses link j (tunnel network interface j) as its outgoing link to the rest of the simulated network. The route configurations for host i are as follows:

```
For each link in the simulated network, do

   let k be the identity of the link
   if (k != j) do
       route add 1.1.k.i -interface tun(j)
```

For a large network, the number of routing entries to be generated and set up can be large. To save a user's time and effort, our simulator provides a program that reads in the network configuration file and automatically generates the needed routing entries. In addition, the routing daemons provided on UNIX (e.g., routed and gated which implement RIP and OSPF routing protocols respectively) can be run on nodes in a simulated network to cooperatively and dynamically generate and set up routing entries.

6.4. Link Configuration

The simulator reads in a link configuration file to simulate a link's bandwidth and delay. The format of a line for a link in this file is (Link_ID, Destination_Node_ID, Bandwidth, Delay). Because we associate link i with tunnel network interface i when we configure tunnel network interfaces, as performed in Section 4.2, each virtual link object knows from which tunnel network interface to read packets. For a link, if it wants to deliver a packet to its destination node, after simulating the link's propagation delay and the packet's transmission time, it can use the same tunnel network interface from which it reads packets to write packets into the kernel. Actually, it does not matter which tunnel network interface a virtual link object should use to write packets into the kernel. The result will be the same. This is because, as in a real network, no matter from which network interface a packet is received, the kernel can still correctly forward the packet or deliver it to an application program. The following six lines are for links 1, 2, 3, 4, 5 and 6.

```
# link_ID    next_Node_ID     BW           delay
# ------------------------------------------------
  1           3               10Mbps       1ms
  2           1               10Mbps       2ms
  3           4               10Mbps       3ms
  4           3               10Mbps       4ms
  5           2               10Mbps       5ms
  6           4               10Mbps       6ms
```

6.5. Example Application Programs

Any existing real-world application program (e.g., the Netscape web browser and the Apache web server) can readily run on any node in our simulated network The following are just a few application examples that our simulator has used. We illustrate them using the network of Figure 3 (a).

6.5.1: "Ping" Reports Round-Trip Time

"Ping" is a useful tool to test whether our simulator can correctly simulate links with various delays and bandwidths. Usually, in a real-world network, "ping" can only be executed on a host. This means that only the round-trip time between an edge host and a node (either an edge host or a router) can be reported. In contrast, in our simulator, "ping" can report a packet's round-trip time between any two

nodes. The following example demonstrates that we can use "ping" to estimate the round-trip time between router 1 (node 3) and router 2 (node 4) of Figure 3 (a), neither of which is an edge host.

```
# ping 1.1.4.3
PING 1.1.4.3 (1.1.4.3): 56 data bytes
64 bytes from 1.1.4.3: icmp_seq=0 ttl=255 time=7.000 ms
64 bytes from 1.1.4.3: icmp_seq=1 ttl=255 time=7.000 ms
^C
--- 1.1.4.3 ping statistics ---
2 packets transmitted, 2 packets received, 0%
packet loss
round-trip min/avg/max = 7.000/7.000/7.000 ms
```

6.5.2: "Traceroute" Shows the Routing Path

"Traceroute" can test whether routes are correctly set up in our simulator. Being able to use "traceroute" to show the routing path between any two nodes, our simulator has been helpful in debugging routing protocols. In the following example, "traceroute" outputs the routing path from host 2 to host 1 of Figure 3 (a).

```
# traceroute 1.1.1.2
traceroute to 1.1.1.2 (1.1.1.2), 30 hops max, 40 byte
packets
 1  1.1.6.2  11.000 ms  11.000 ms  11.000 ms
 2  1.1.4.2  19.000 ms  18.000 ms  18.000 ms
 3  1.1.1.2  21.000 ms  21.000 ms  22.000 ms
```

Because the links used in the simulator are simplex links, the output of "traceroute" in our simulator is somewhat different from its normal output. To understand its output in our simulator, we need to look at the Link_ID field (the second least significant byte) of the IP addresses reported by "traceroute." In general, the sequence of these Link_ID values shows us how a packet is routed along these links. The only exception occurs on the last hop of the reported routing path, where the reported Link_ID gives us the reverse direction of the actual link that is used to transmit packets to the destination node. For example, in the above output, "traceroute" shows that a packet is first sent on link 6, then on link 4, and finally on link 2. (Although it reports that link 1 is the last hop, according to our explanation, we know it means link 2.) This anomaly is caused by UNIX's different processing of choosing the source IP address to be included in the error-reporting packet. On every hop, except the last hop, where the error-causing packet has not yet reached its destination node, the chosen source IP address is the error-causing packet's incoming interface's address. However, on the last hop, where the error-causing packet has reached its destination, it is the error-reporting packet's outgoing interface's address. Since an Ethernet's incoming and outgoing interfaces use the same IP address, this anomaly does not happen in a real network that uses Ethernet interfaces.

6.5.3: "Ftp" Client and Server on Any Node

"Ftp" clients and servers can readily work on our simulation machine. Since ftp clients can accept scripts to "get" and "put" files automatically, we can use them to generate network traffic in different directions automatically. The following example illustrates the use of "ftp" to "put" a file to /dev/null on a remote node. (/dev/null is a sink device on an UNIX machine. It sinks all data without writing it to disks, thus eliminating unnecessary disk I/O operations on the simulation machine.)

```
# ftp 1.1.4.3
Connected to 1.1.4.3.
220 nsl.csie.nctu.edu.tw FTP server
ftp> ls
200 PORT command successful.
150 Opening ASCII mode data connection for '/bin/ls'.
total 73408
-rw-rw-r--  1 root  wheel   2383872 Aug  8 23:53 file1
226 Transfer complete.
ftp> put file1 /dev/null
local: file1 remote: /dev/null
200 PORT command successful.
150 Opening BINARY mode data connection for
'/dev/null'.
226 Transfer complete.
2383872 bytes sent in 2.29 seconds (1017.04 Kbytes/s)
```

The above throughput report confirms that our simulator can correctly simulate 10 Mbps links. After removing the bandwidth consumed by the IP and TCP header overhead, the achieved throughput of 1017.04 KB/sec is roughly the throughput that can be achieved on 10 Mbps links with an MTU of 576 bytes.

Notice that the ftp server in the above example is on router 2 (node 4) in the simulated network, not on an edge host. Moreover, the ftp client is running on router 1 (node 3), also not on an edge host. Actually, in our simulator, a real application program can run on any node in a simulated network. This capability allows network traffic to be generated deep inside a simulated network.

By using "inetd" [17], the internet "super-server" on an UNIX machine, a ftp server on a node can be dynamically and automatically launched by "inetd" only when the ftp server is really needed. Therefore, providing a ftp server on every node in a simulated network is very resource-efficient, and invoking a ftp server on a node does not need any human effort.

6.5.4: "Tcpdump" Monitors Packets on Any Link

"Tcpdump" is a useful tool for monitoring and scrutinizing packets transmitted on a link (e.g., an Ethernet). Since "tcpdump" opens a network interface to monitor a network's traffic and since, from the kernel's point of view, a tunnel network interface is no different from a normal network interface, we can readily use "tcpdump" to monitor network traffic on any link (tunnel network interface) in a simulated network. This means that we can directly use

many useful "tcpdump" scripts (e.g., [18]) to analyze network traffic. For example, the following shows the use of tcpdump on link 3 of Figure 3 (a) to trace packets transmitted on the link from node 3 to node 4:

```
# tcpdump -i tun3
22:10:01.034208 1.1.3.3.2882 > 1.1.4.3.8000:
38232:39692(1460) ack 97 win 8192 (ttl 27, id 39326)
```

6.5.5: "Trpt" Traces Any TCP Connection

If compiled with the TCPDEBUG option, the UNIX kernel will automatically trace the state and variables associated with a TCP connection whenever certain events occur (e.g., just sent out a packet, just received a packet, and a timer just expired). The information recorded include values of many important variables, such as the current timestamp, sequence numbers, congestion window size, slow start threshold, and timers' information. This information is beyond what "tcpdump" can observe. "Trpt"[19] is a tool on an UNIX machine that can extract a TCP connection's information from the kernel to the user level for analysis. The following is a line of "trpt"'s output that contains send and receive sequence numbers, sending window size, and timer (retransmit and keep alive) information:

```
# trpt
631 ESTABLISHED:output
(src=1.1.3.3,3195,dst=1.1.4.3,8000)[75219d1..7521f85)@8e7
54(win=8052)<ACK> -> ESTABLISHED rcv_nxt 8e754 rcv_wnd
100a4 snd_una 7512a49 snd_nxt 7521f85 snd_max 7521f85
snd_wl1 8e754 snd_wl2 74bf3c1 snd_wnd 10000 REXMT=3 (t_
rxtshft=0), KEEP=14400
```

6.5.6: Mobile IP Simulation Is Easy

Our simulator has been used to study the performance of mobile IP [20]. Figure 5 illustrates how this simulator's architecture allows us to easily implement a home agent. The home agent needs to intercept arriving packets destined for a mobile station that is not currently in its home network, encapsulate them, and then send them to the mobile station's current foreign agent. To intercept a mobile station's packets, we simply redirect them to a special tunnel network interface (tun_redirect in this example) by changing an entry in the routing table. For example, in Figure 5, we change [1.1.1.3 -> tun2] to [1.1.1.3 -> tun_redirect]. The home agent then can read redirected raw packets from this special tunnel network interface in the same way as a virtual link object reads raw packets from its associated tunnel network interface. To encapsulate and tunnel these packets to the foreign agent, the home agent need only treat these raw packets as normal data and send them to the mobile station's foreign agent via a normal datagram socket.

Implementing a foreign agent on top of this simulator is equally easy. Since a mobile station's tunneled packets are received by its foreign agent via a normal datagram socket,

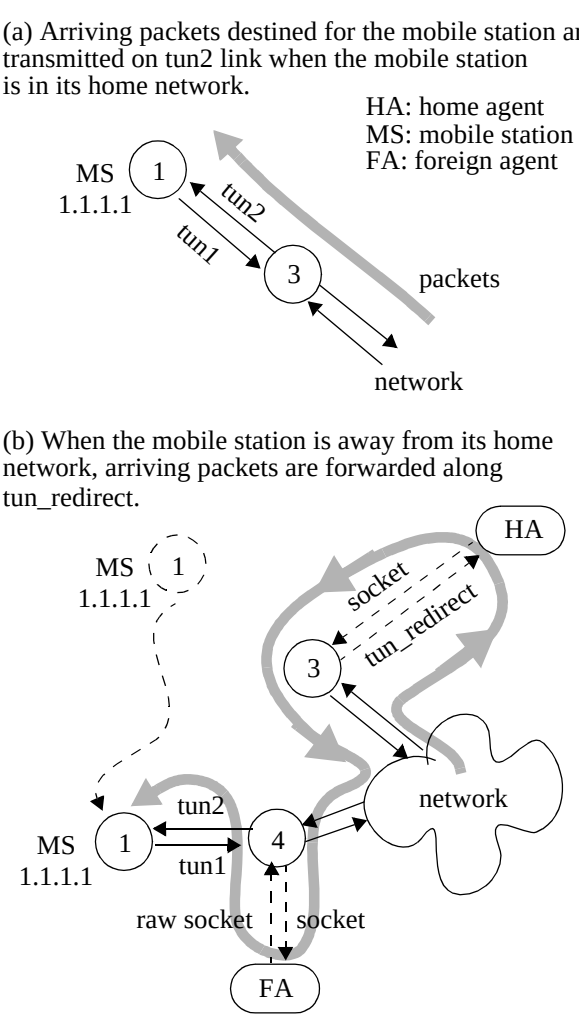

Figure 5: Simulating mobile IP. By changing a routing entry on node 3 from [1.1.1.3 -> tun2] to [1.1.1.3 -> tun_redirect], the home agent can intercept arriving packets destined for the mobile station. It then reads these raw packets from tun_redirect and sends them to the foreign agent via a normal datagram socket.

when they are delivered to the foreign agent at the user level, the packets have been automatically decapsulated in the kernel. The foreign agent uses a raw socket to send the received raw packets to the kernel, which then sends the packets on a tunnel link that is directly connected to the mobile station. A raw socket instead of a normal socket is used because the received raw packet already has its own TCP/IP headers and thus should bypass the normal TCP/IP protocol stack processing. Our implementations for the home and foreign agents contain only about 20 lines of C code for intercepting, encapsulating, tunneling, and decapsulating traffic. This would be hard to achieve if we would use a traditional simulator for this task.

7. Improvement and Extension

7.1. Improve Simulation Speed

To speed up simulations, we move the user-level event scheduler down into the kernel. Running a simulation in the kernel is implemented as a never-return system call. In the original implementation, when a packet goes through many nodes along its path, it needs to be copied out and into the kernel multiple times before reaching its destination, as shown in Figure 3 (c). Because the data volume that needs to be copied is large and the frequency of these copy operations is high, a lot of simulation time is spent on these copy operations and simulations are slow. In the new in-kernel implementation, because a packet now can always stay in the kernel on its way to its destination node, copy operations can be eliminated and replaced with cheap operations of moving pointers from one tunnel interface directly to another in the kernel. Compared to the cost of copying a whole packet out and into the kernel, the cost of moving a pointer is minimal. Since the reduction of the required CPU time is so great and it is on the performance critical path, the new in-kernel simulator runs about 500% faster than its original user-level implementation. As a consequence of eliminating packet copy cost, now sending real data in our simulator is no longer a performance burden, but an asset without any overhead. (In order to run faster, some other TCP/IP simulators only send “fake” or “null” data in packets.)

7.2. Support a Variety of Scheduling and Queueing Disciplines

Sometimes in a network to be simulated, output links may use a variety of packet scheduling methods (e.g., FIFO and round-robin) and/or queueing disciplines (e.g., drop-tail and RED [21]). But normally a UNIX kernel supports only FIFO and drop-tail. Thus a TCP/IP network simulator constructed using UNIX and based on our methodology cannot simulate this kind of network. To solve this problem, the ALTQ tool [22], which allows a network interface to use a different packet scheduling method and/or queueing discipline in a UNIX kernel, can be installed on the simulation machine. It is easy to change buffer size in ALTQ.

7.3. Emulation and Distributed Simulation

Our simulator can easily be used as an emulator to intercept live packets, let them traverse in a simulated network, and then transmit them onto a real network again. Interfacing a simulated and real network is easy for our simulator. This is because our simulator uses tunnel interfaces to construct a virtual simulated network and tunnel interfaces can also be used in the real world to construct a real network. By properly configuring tunnel interfaces and route entries on the simulation machine, live packets from a real network can be first received at on an Ethernet interface and then automatically forwarded to a tunnel interface. From now on, a live packet can traverse in a simulated network until it needs to be transmitted onto a real network again. At this time, it can be automatically forwarded from a tunnel interface to an Ethernet interface for transmission. For the same reason, in a distributed simulation, packets can be easily exchanged among the partitioned subnetworks of a simulated network.

8. Scalability Discussions

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

8.1. Number of Nodes

Because our scheme uses the kernel re-entering technique to simulate multiple nodes, there is no limitation on the maximum number of nodes that can be simulated.

8.2. Number of Links

In our scheme, because each simulated link uses a tunnel interface, the maximum number of links that can be simulated is limited by the maximum number of tunnel interfaces that a BSD UNIX system can support, which currently is 256. If needed, this number can be increased by modifying the kernel device deriver subsystem.

8.3. Number of Routing Entries

Since in our scheme the kernel routing table can be viewed as an union of each simulated node’s routing table, the size (the number of routing entries) of the kernel routing table will increase as the number of simulated nodes increases. For a network simulation, suppose that the number of nodes to be simulated is N and the number of (tunnel) interfaces used by these nodes is T, then the total number of routing entries that need to be stored in the kernel routing table will be N * T. This is because in our scheme, each tunnel interface defines an IP address and each of these IP addresses has a different “As-Seen-By-Node(i)” address for each different node.

Note that a simulated node uses at least one interface to connect to a simulated network. Therefore, T should be

greater than or equal to N. The size of the kernel routing table required thus is between [N^2, T^2]. As noted above, on current BSD UNIX systems, T cannot be greater than 256. Thus, the size of the kernel routing table required is less than 65,536 (256*256). We have tested several network configurations that need to store over 60,000 routing entries in the kernel routing table. We found that the because the BSD UNIX systems use the radix tree [23] 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.

8.4. Number of Application Programs

Since application programs running on an UNIX simulation machine are all real independent programs, the simulation machine's physical memory requirement is roughly 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 an UNIX machine, we found that the virtual memory mechanism provided on an UNIX machine together with the "working set" property of a running program greatly alleviate the problem. The reason is that, when an application program is running, only a small portion of its code related to network processing will need to be present in the physical memory. In addition, because UNIX machines support the uses of shared libraries and shared virtual memory pages, the required memory space for running the same application program multiple times can be greatly reduced. For example, on a PC with 512 MB physical memory, we can support up to 1,000 TCP connections used by 2,000 ftp and ftpd programs without any page in and page out activities.

8.5. Simulation Speed

A simulator often runs slower and slower when the size of the simulated network increases and/or the network load becomes heavier. The size of a simulated network generally is measured by its numbers of nodes and links, and the offered network load generally is measured by the number of packets that need to be exchanged through a simulated network per second.

To see how our simulator's speed changes under various network sizes, we performed a series of tests using the in-kernel implementation of our simulator. We found that large network sizes do not slow down simulations much. The simulator's speed is relatively fixed under various network sizes. We used the chain topology as the simulated network's topology (like the one depicted in Figure 3 (a)) and varied the length of the chain (the number of nodes and links) from 1 to 250. The traffic imposed on the simulated network is a greedy UDP flow originating on the first node and ending on the last node. We found that as the network size increases, the required simulation time only slightly increases. As an evidence, our results show that the simulation time for the 250-node case is only 1.8 times of the simulation time for the 2-node case. We attribute this scalability to the in-kernel implementation of our simulator. This is because, as explained in Section 7.1, the packet copy operation overhead on the intermediate forwarding nodes can be eliminated.

To see how our simulator's speed changes under various network loads, we used a 10-node chain network and varied the bandwidth of its links from 10 to 100 Mbps. Because the traffic imposed on the simulated network is a greedy UDP flow, the network load will increase when we increase the link bandwidth. We found that increasing network load from 10 to 100 Mbps had a more significant effect than increasing network sizes on slowing down simulations. The slow down factor is 3.5 when the network load was varied from 10 to 100 Mbps. We attribute this slow down to the increased number of events that our simulator needs to process per second in virtual time.

9. Limitation

Since only a single UNIX machine (with its own protocol stack) is used to simulate multiple nodes, our simulator has a limitation that it allows only one version of TCP/IP protocol stack in a simulated network. Studying interactions between different TCP versions (e.g., TCP tahoe and TCP reno) or between different TCP implementations (e.g., FreeBSD and Linux) thus cannot be done by using our simulator as is. One way to overcome this limitation is to use a distributed simulation approach discussed in Section 7.3. In such a distributed environment, a UNIX machine with a particular protocol stack can be used to simulate nodes using the same stack, while other UNIX machines with different stacks may be used to simulate nodes using different stacks.

Furthermore, we note that if a simulation study requires modification of in-kernel protocols, the kernel will need to be modified and recompiled before our simulation methodology can be used. Modifying and recompiling the kernel, however, may represent a challenge for some users.

10. Conclusions

We have described a methodology for easily constructing extensible and high-fidelity TCP/IP network simulators. Due to its unique architecture, a simulator constructed under our methodology has many important advantages that are hard to achieve by traditional network

simulators. First, the simulator uses the real-life TCP/IP protocol stack of the simulation machine. As such, its simulation results are more accurate than those generated by a traditional simulator that abstracts a lot away from the real-life implementation. Second, since the standard UNIX system call API is provided on every node in a simulated network, any existing or future real application program can run on any node in a simulated network. In addition, because the simulation implementation of a developed application program on our simulator can be directly used as the program's real implementation, much time and effort can be saved.

The proposed methodology is general and not specific to a particular platform. Although our simulator is based on FreeBSD, this methodology actually can be applied to any machine that has a TCP/IP protocol stack and supports tunnel interfaces.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable comments.

For S.Y. Wang, this research was supported in part by MOE Program of Excellence Research under contract 89-E-FA04-4, the Lee and MTI Center for Networking Research, NCTU, and the Institute of Applied Science and Engineering Research, Academia Sinica, Taiwan.

For H.T. Kung, this research was supported in part by NSF grant ANI-9710567, Air Force Office of Scientific Research Multidisciplinary University Research Initiative Grant F49620-97-1-0382, and grants from Microsoft Research and Nortel Networks.

References

[1] S.Y. Wang and H.T. Kung, "A Simple Methodology for Constructing Extensible and High-Fidelity TCP/IP Network Simulators," IEEE INFOCOM'99, March 21-25, 1999, New York, USA.

[2] S. McCanne, S. Floyd, ns-LBNL Network Simulator. (http://www-nrg.ee.lbl.gov/ns/)

[3] MIL3 Inc. home page, http://www.mil3.com/products.

[4] T. Magedanz, K. Rothermel, and S. Krause "Intelligent Agents: An Emerging Technology for Next Generation Telecommunications?," IEEE INFOCOM 1996, March 24-28, 1996.

[5] C. E. Perkins, "Mobile IP: Design Principles and Practices," Addison-Wesley, 1998.

[6] H.T. Kung and S.Y. Wang, "The Behavior of Competing TCP Connections on a Packet-Switched Ring : A Study Using the Harvard TCP/IP Network Simulator," PDPTA'99 (International Conference on Parallel and Distributed Processing Techniques and Applications), June 28 - July 1, 1999, Las Vegas, USA .

[7] Harvard TCP/IP network simulator 1.0, available at http://www.eecs.harvard.edu/networking/simulator.html.

[8] Fall and K., "Network Emulation in the Vint/NS simulator," ISCC99, July 1999.

[9] Nist net, available at http://snad.ncsl.nist.gov/itg/nistnet.

[10] Jong Suk Ahn, P. Danzig, Zhen Liu, and 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," Computer Communication Review, Vol. 27, No. 1, p.31-41, January 1997.

[12] S. Keshav, "REAL: A Network Simulator," Technical Report 88/472, Dept. of computer Science, UC Berkeley, 1988.

[13] SSF network module (SSFnet), available at http://www.ssfnet.org.

[14] PARSEC and GloMoSim, available at http://pcl.cs.ucla.edu/projects/parsec.

[15] X.W. Huang, R. Sharma, and S. Keshav, "The ENTRAPID Protocol Development Environment," IEEE INFOCOM'99, March 21-25, 1999, New York, USA.

[16] A. Meyer and L.H. Seawright, "A Virtual Machine Time-Sharing System," IBM Systems Journal, Vol. 9, No. 3, 1970, pp. 199-218.

[17] Inetd, UNIX manual page (8).

[18] Software available in the Internet Traffic Archive, http://ita.ee.lbl.gov/html/software.html

[19] Trpt, UNIX manual page (8).

[20] R. Ramjee, T. La Porta, S. Thuel, K. Vardhan, and S.Y. Wang "HAWAII: A Domain-based Approach for Supporting Mobility in Wide-area Wireless Networks," IEEE ICNP'99, October 31 - November 3, 1999, Toronto, Canada. (A modified version will appear in IEEE/ACM Transaction on Networking.)

[21] S. Floyd and V. Jacobson, "Random Early Detection Gateways for Congestion Avoidance," IEEE/ACM Transactions on Networking, Vol.1 No.4, August 1993, p. 397-413.

[22] K. Cho. "A Framework for Alternate Queueing: Towards Traffic Management by PC-UNIX Based Routers," USENIX'98 Annual Technical Conference, June 1998.

[23] Gary R. Wright and W. Richard Stevens, "TCP/IP Illustrated Volume 2," Addison Wesley, 1995.

Appendix

In this appendix we illustrate the configuration of the network simulator for a network topology that is more complex than the simple one used in Section 6.1, 6.2., and 6.3. Figure 6 depicts the mesh topology considered in this appendix.

Tunnel Network Interface Configuration

The commands used to configure the twelve tunnel network interfaces for link 1, 2, ..., and 12 of Figure 6 are shown as follows:

```
ifconfig tun1 1.1.1.1 1.1.1.254 netmask 0xffffff00
ifconfig tun2 1.1.2.4 1.1.2.254 netmask 0xffffff00
ifconfig tun3 1.1.3.3 1.1.3.254 netmask 0xffffff00
ifconfig tun4 1.1.4.6 1.1.4.254 netmask 0xffffff00
ifconfig tun5 1.1.5.2 1.1.5.254 netmask 0xffffff00
ifconfig tun6 1.1.6.5 1.1.6.254 netmask 0xffffff00
ifconfig tun7 1.1.7.4 1.1.7.254 netmask 0xffffff00
ifconfig tun8 1.1.8.6 1.1.8.254 netmask 0xffffff00
ifconfig tun9 1.1.9.6 1.1.9.254 netmask 0xffffff00
ifconfig tun10 1.1.10.5 1.1.10.254 netmask 0xffffff00
ifconfig tun11 1.1.11.5 1.1.11.254 netmask 0xffffff00
ifconfig tun12 1.1.12.4 1.1.12.254 netmask 0xffffff00
```

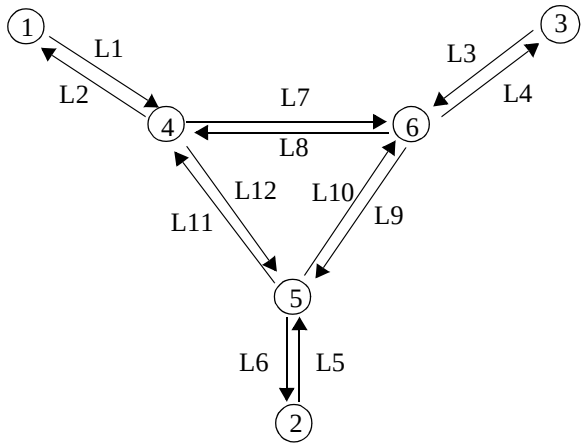

Figure 6: A mesh network topology to illustrate configuration of the network simulator.

Route Configuration

The commands used to configure the routes for the simulated network of Figure 6 are shown as follows:

```
route add 1.1.2.1 -interface tun1
route add 1.1.3.1 -interface tun1
route add 1.1.4.1 -interface tun1
route add 1.1.5.1 -interface tun1
route add 1.1.6.1 -interface tun1
route add 1.1.7.1 -interface tun1
route add 1.1.8.1 -interface tun1
route add 1.1.9.1 -interface tun1
route add 1.1.10.1 -interface tun1
route add 1.1.11.1 -interface tun1
route add 1.1.12.1 -interface tun1

route add 1.1.1.2 -interface tun5
route add 1.1.2.2 -interface tun5
route add 1.1.3.2 -interface tun5
route add 1.1.4.2 -interface tun5
route add 1.1.6.2 -interface tun5
route add 1.1.7.2 -interface tun5
route add 1.1.8.2 -interface tun5
route add 1.1.9.2 -interface tun5
route add 1.1.10.2 -interface tun5
route add 1.1.11.2 -interface tun5
route add 1.1.12.2 -interface tun5

route add 1.1.1.3 -interface tun3
route add 1.1.2.3 -interface tun3
route add 1.1.4.3 -interface tun3
route add 1.1.5.3 -interface tun3
route add 1.1.6.3 -interface tun3
route add 1.1.7.3 -interface tun3
route add 1.1.8.3 -interface tun3
route add 1.1.9.3 -interface tun3
route add 1.1.10.3 -interface tun3
route add 1.1.11.3 -interface tun3
route add 1.1.12.3 -interface tun3

route add 1.1.1.4 -interface tun2
route add 1.1.3.4 -interface tun7
route add 1.1.4.4 -interface tun7
route add 1.1.5.4 -interface tun12
route add 1.1.6.4 -interface tun12
route add 1.1.8.4 -interface tun7
route add 1.1.9.4 -interface tun7
route add 1.1.10.4 -interface tun12
route add 1.1.11.4 -interface tun12

route add 1.1.1.5 -interface tun11
route add 1.1.2.5 -interface tun11
route add 1.1.3.5 -interface tun10
route add 1.1.4.5 -interface tun10
route add 1.1.5.5 -interface tun6
route add 1.1.7.5 -interface tun11
route add 1.1.8.5 -interface tun10
route add 1.1.9.5 -interface tun10
route add 1.1.12.5 -interface tun11

route add 1.1.1.6 -interface tun8
route add 1.1.2.6 -interface tun8
route add 1.1.3.6 -interface tun4
route add 1.1.5.6 -interface tun9
route add 1.1.6.6 -interface tun9
route add 1.1.7.6 -interface tun8
route add 1.1.10.6 -interface tun9
route add 1.1.11.6 -interface tun9
route add 1.1.12.6 -interface tun8
```