Proceedings of IEEE INFOCOM'99,
pp. 1134-1143

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

S. Y. Wang and H.T. Kung
{shieyuan, htk}@eecs.harvard.edu
Division of Engineering and Applied Sciences
Harvard University
Cambridge, MA 02138, USA

Abstract[1]

This paper proposes a simple methodology for constructing extensible and high-fidelity TCP/IP simulators in BSD UNIX environments. A simulator constructed under this methodology will simulate multiple network nodes by re-entering the UNIX kernel of the simulation host multiple times. Generated simulation results are derived from executing the native TCP/IP protocol stack on the simulation host. They are thus more accurate than those generated from a TCP/IP network simulator that implements only an abstraction of a real-life TCP/IP implementation.

By using this methodology, the simulator architecture creates an illusion for the BSD UNIX kernel that the simulated network is a real network. All existing application programs such as ftp, telnet and http, and all network utilities such as route, ifconfig and tcpdump are immediately applicable to a simulated network for generating network traffic, configuring networks, gathering statistics, etc. Additionally, the network simulator provides the standard UNIX API on every node in a simulated network so that any existing or future application program can run on any node in a simulated network. This allows a network simulator to be easily extended to study high-level network architecture and application issues.

1. Introduction

Network simulators implemented in software are valuable tools for researchers to develop, test and diagnose network protocols. Simulation is economical because it can carry out experiments without the actual hardware. It is flexible because it can, for example, simulate a link with any bandwidth and propagation delay and a router with any queue size and queue management policy. Simulation results are reproducible and easy to analyze because the simulated network environment is free of other uncontrollable factors (e.g., other unwanted external traffic), which researchers may encounter when doing experiments on real networks.

It is well known, however, that network simulators also have their own limitations. Developing a complete network simulator, including associated application programs and network tools, is a large effort. Due to limited development resources, typical network simulators have the following drawbacks:

- Simulation results are usually not as convincing as those produced by real hardware and software equipment. In order to constrain their complexity, most existing network simulators can only simulate real-world network protocol implementations with limited detail.
- These simulators are not extensible. They lack the application programming interface (API) that allows application programs to be developed and run on nodes in a simulated network. 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 system (e.g., a distributed database system's response time when running on a particular network configuration) cannot be studied. But 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 for their systems. Commercial simulation systems have been developed to meet some of this need [1]. Second, the lack of API prohibits the use of these network simulators in areas where user-developed programs need to run on nodes to carry out tasks cooperatively. Examples of these areas include "Active networks" [2], "intelligent mobile agents" [3], "Mobile IP" [4] and "virtual private networks" [5].

This paper proposes a simple simulation methodology for alleviating these drawbacks. It simulates multiple

[1] Due to the paper length limitation, this is a heavily condensed version of its original version of 16 pages. Detailed discussions on addressing, address-remapping, and routing schemes in Section 2.3 are omitted. For the complete version, please visit http://www.eecs.harvard.edu/networking.

network nodes by re-entering the UNIX kernel of the simulation host multiple times. Based on this methodology, we have constructed a TCP/IP network simulator that simulates a network of BSD UNIX hosts and routers. This network simulator has two good properties: 1) It makes direct use of the real-life BSD UNIX TCP/IP protocol code, existing network application programs, and existing network tools. As a result, while being able to use existing software, it generates more accurate simulation results than a traditional TCP/IP network simulator that may abstract a lot away from real-life TCP/IP implementations; 2) It provides the BSD UNIX system's API (i.e., the standard UNIX system call interface) on every node in a simulated network and allows application programs to be developed and run on any node in a simulated network. Since a developed application program is a real UNIX program, our simulator has an important advantage that a 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 the simulation implementations. For example, a Harvard network projects course in the Fall semester of 1998 made direct use of several programs, originally designed on our simulator, for the real network testbeds in our teaching lab.

This methodology is said to be *simple* in the sense that it allows easy development of high-fidelity TCP/IP simulators with minimal time and effort, via the use of existing code. Only about one hundred lines of code addition and modification to the kernel are needed.

In this paper, we define a "high-fidelity" TCP/IP simulation as one that uses a real-life unmodified TCP/IP implementation and can correctly reflect TCP/IP's qualitative behaviors under different network configurations. Note that although it is desirable to run simulations that can predict quantitative performance results as accurate as those generated by a real hardware experiment, so far, to the best of the authors' knowledge, no known simulators (including ns [6] used by many papers on networking research) have been able to achieve this goal. This is partly due to the difficulty in simulating a real host or router's packet processing time, which contains non-deterministic, architecture-specific, and vendor-specific components. Therefore, in this paper we are satisfied with defining a "high-fidelity" simulation, not as one that can precisely predict quantitative performance results (as would be more desired), but as one that can correctly reflect TCP/IP's qualitative behaviors. Although our high-fidelity simulator may not generate precisely the same quantitative performance results as those generated from experiments on real hardware, we can use the results generated by the simulator under various network configurations to study network issues. For example, we can study the effect of different buffer allocation or packet scheduling methods on TCP/IP performance. These quantitative performance results may not be exactly the same as real performances, but their trend can still provide valuable insights and direct us toward a better design.

Our simulator is operational. Its simulation results have been validated extensively against results obtained from experiments done on real hardware, and shown to be able to correctly reflect TCP/IP network behaviors. Several institutions and companies are using the simulator to study TCP performance on various network architectures such as "TCP trunking" [7] and "mobile IP." Because each simulated node supports the standard UNIX API, the simulator can also be used to study high-level network applications such as those related to active networks [8, 9].

2. Simulator Architecture Overview

Our simulator architecture differs from traditional ones in how they integrate the various component programs that implement the following functions:

1. Links with various delays and bandwidths
2. Routers that forward IP packets
3. Hosts that use TCP/IP protocol to send and receive packets
4. Application programs that generate network traffic

Unlike traditional approaches such as REAL [10] and ns [6], our simulator architecture does not combine parts 1, 2, 3 and 4 together to form a single monolithic program. Instead, our simulator has separate and independent parts for 1, 2, 3 and 4. When these parts run concurrently in a BSD UNIX environment, together their executions simulate a network and generate network traffic.

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

2.1. Tunnel Network Interface

Tunnel network interface, available in most BSD UNIX environments, 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 a normal Ethernet or FDDI 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.

As depicted by Figure 1, we can simulate a network configuration in which two hosts are connected together by

two one-way links of any bandwidth, delay and other characteristics (e.g., packet corruption, packet dropping, reordering, and duplication). This can be done simply by writing an application program (we call it "virtual link" object) that plays the role of a link. This application would open a tunnel network interface's special file in /dev and then execute a loop in which it reads a packet from the special file, waits the link's propagation delay time plus the packet's transmission time on the link, and writes this packet to the special file. The application continues this loop forever.

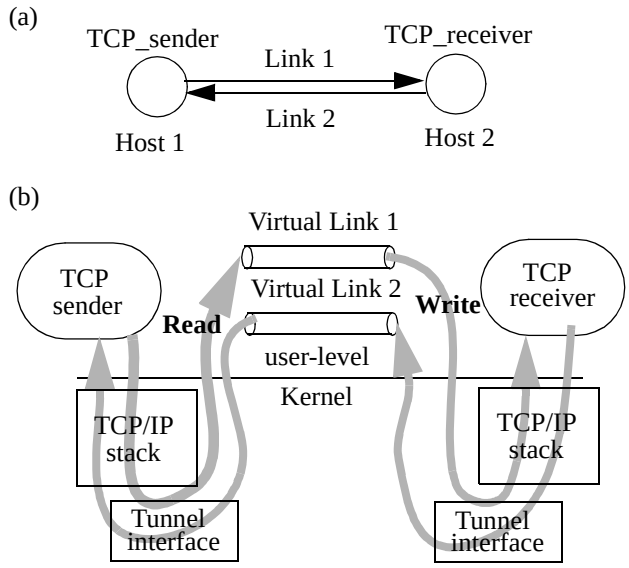


Figure 1: A network (a) is simulated using tunnel network interface (b).

2.2. 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 the 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 host can not see this packet because the packet will not enter and leave the kernel again until it finally reaches its destination host. OPNET Modeler [1] and ns [6] are examples of network simulators using the opaque network cloud simulation model.

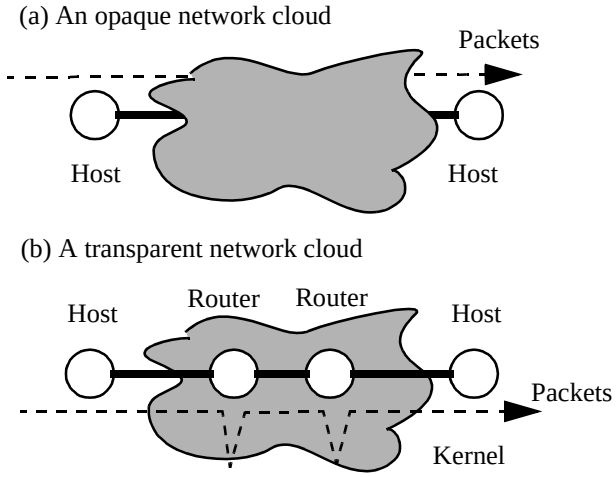


Figure 2: Opaque and transparent network clouds differ in whether or not packets traversing through routers in the simulated network are "visible" to the kernel of the simulation host.

In contrast to the traditional approach, our methodology will simulate a router's packet forwarding by sending down each packet received from a simulated link to the kernel, letting the kernel forward it toward the correct direction (i.e., put it into the correct output port's queue), pulling up the packet from the kernel (i.e., fetch it from the output port's queue), and then transmitting it on the next simulated link. We call the 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, this packet will go down (enter the kernel) and go up (leave the kernel) when going through each router on the way to the destination host. Thus the kernel will see the packet when it traverses the network cloud.

A network simulator under the transparent network cloud simulation model uses the UNIX kernel of the simulation host to implement the routers in a simulated network. As a result it has the following advantages: 1) There is no need to spend time and effort on porting kernel routing/forwarding code to a user-level program to simulate routers. 2) Because the unmodified real-world BSD UNIX routing/forwarding code is used, simulation results are more credible than otherwise. 3) The standard UNIX system call interface (API) is supported on every node. Thus all application programs available on hosts can now run on routers as well.

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

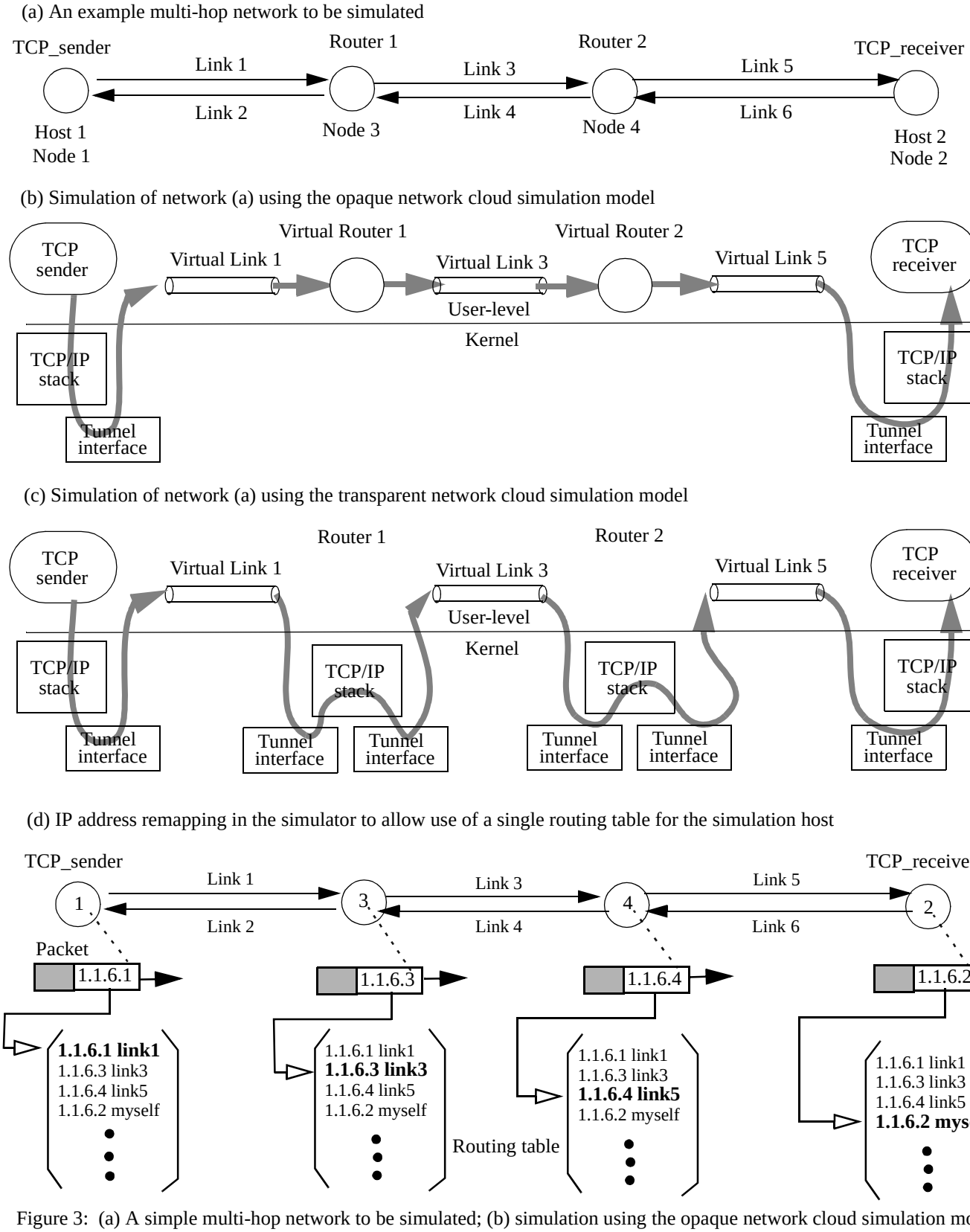


Figure 3: (a) A simple multi-hop network to be simulated; (b) simulation using the opaque network cloud simulation model;. (c) simulation using the transparent network cloud simulation model; and (d) IP address remapping in the simulator. Notice that for presentation simplicity, links 6, 4 and 2 are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

2.3. Addressing and Routing for the Transparent Network Cloud Simulation Model

This section presents our addressing, routing, and address-remapping schemes to support the transparent network cloud simulation model. As described earlier, the single simulation host will act as both hosts and routers during a simulation. Using the network of Figure 3 (a) as an example, we illustrate the operation of the simulation host when sending a packet across the network from node 1 to node 2. As depicted by Figure 4 it involves 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.

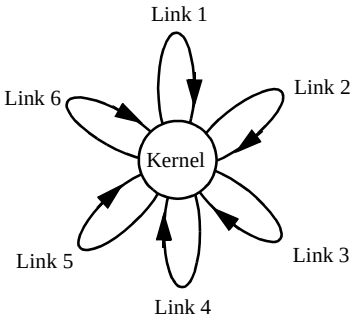


Figure 4: Routing a packet along a route in the simulated network of Figure 3 (c) is a sequence of leaving and entering the kernel operations.

For routing packets, the simulation host needs to maintain a routing table. Conceptually, since the simulation host simulates all the routers and hosts in a simulated network, by using the union of the routing tables in all the nodes, it should be able to route packets correctly. However, 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 host's routing table, the kernel will be confused and will not be able to choose the correct next hop for forwarding a packet.

One solution to this problem is to have, in the kernel, a separate routing table for every node in the simulated network and to have each packet re-entering the kernel tell the kernel which node it should simulate for the packet at this time (e.g., when passing through router 2, the packet will tell the kernel that now it should simulate router 2). The kernel then uses this information to retrieve the proper routing entry (destination IP address, next hop) from the correct routing table.

Although the above method can solve the routing problem discussed, it is not our preferred solution for the following reasons: First, maintaining a separate routing table in the kernel for every node is not the standard mechanism used in the UNIX simulation host. In order to work around, we will need to modify the kernel code related to routing processing extensively. This violates our goal of minimizing modification to real-life network protocol code in order to provide high-fidelity simulation. Second, using a different routing mechanism means that we no longer can use existing utilities in UNIX environments, such as "route," to configure routes.

Our preferred solution is to use special address-remapping and route-setup schemes. The basic idea is that by remapping the destination IP address of a packet to a new one before it arrives at and is forwarded by a router, a single routing table can be used in the simulation host without the conflicting problem. Because traffic may be bidirectional (e.g., TCP traffic), address-remapping needs to be applied to a packet's both source and destination addresses. We call the mapped version of an IP address on node i as this IP address's **"As-Seen-By-Node(i)"** address. Since before a packet arrives at a router it is transmitted along a link, it is natural that in our simulator the virtual link objects implement the address remapping. Figure 3 (d) illustrates the address-remapping idea.

3. Required Supports from Kernel and Applications

3.1. Kernel Modifications

Skip IP and UDP/TCP Checksum Tests. Because in simulation the source and destination IP addresses of a packet will change every hop (see Section 2.3), the checksums in the IP and UDP/TCP headers of the packet are incorrect, and should not be checked. Skipping these checksum tests will not affect the data integrity of packets in our simulator since all packets in our simulated network, in fact, never leave the simulation host. For situations where we need to simulate a corrupted packet, we can simply set a flag in its IP header. Nodes in the simulated network can then detect the corruption, and discard the packet.

TCP Timers Based on Virtual Time. TCP slow and fast timers will be 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 at a time which is k times smaller

than it should, if our simulator is k times slower than the real network.

Each Node Has Its Own Virtual Clock. In the real world, each machine's clock may be different from others' due to clock drift. We should simulate this phenomenon to avoid multiple TCP connections to drop packets, time out, and "slow-start" in a lock-step manner. In our simulator, we maintain a virtual clock for each node in the simulated network. The granularity of these virtual clocks can be very coarse (e.g., 100 ms) because a TCP slow timer is triggered only every 500 ms and a TCP fast timer every 200 ms. These virtual clocks are offset by some random time.

Immerse the Simulated Network into the Kernel. Significant simulation speed up is achieved by implementing virtual link objects in the kernel. Forwarding a packet now becomes an inexpensive operation of moving a pointer to a packet inside the kernel. Compared to the cost of copying a whole packet out or into the kernel, the cost of moving a pointer is minimal. Since the reduction of the required CPU time is so great, our kernel-version IP simulator can run three times faster than its original version. As a consequence of eliminating packet copy cost, now sending real data in our IP simulator is no longer a performance burden, but an asset without any overhead (some other IP simulators only send "fake" or "null" data in order to run faster).

3.2. Application Modifications

An application needs to perform the following three tasks to work with the simulator.

Associate the Application's TCP Sockets with the ID of the Node Where the Application Will Run. The simulator provides the application the identity of the node in the simulated network on which its TCP socket will be created. The application program then makes the setsockopt() system call to associate its TCP socket with the node identity provided. The TCP socket's retransmit timer can now be triggered based on the virtual clock of the node on which the application is running.

Convert the Destination Address to the "As-Seen-By-Node(i)" Address. The application normally would need to convert its packets' addresses before they are delivered to and routed by the kernel. The conversion can be avoided if the destination address provided to the application program is already the "As-Seen-By-Node(i)" address of a packet's destination address (assuming that the application is running on node i).

Use 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.

3.3. Simulator's Event Scheduler

Our system has only one kind of object to simulate, i.e., virtual link objects that simulate links. Hosts and routers are "simulated" or "run" by the BSD UNIX kernel of the simulation host; therefore, we do not need to create corresponding objects for them in the simulation system. Due to this property, our simulation system's event scheduler is both simple and efficient. It schedules 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 also 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 this event scheduler is to send periodically 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 the real time. As discussed in Section 3.2, a node's virtual clock, which is used solely for triggering TCP connections' timers, can have a coarse granularity (e.g., 100 ms in the virtual time). This means that the overhead of sending down the simulated network's virtual time into the kernel is very low (only 10 times per second in the virtual time).

The simulator is a discrete-event dynamic system. The unit of its virtual time can be set to any value as small as we would like (e.g., nanosecond) to simulate high speed links. All events can thus be precisely scheduled and triggered based on the virtual time in the simulated network. For this reason, simulation results are not affected by other activities on the simulation host (e.g., disk I/O and network I/O).

4. 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, after the three slight modifications mentioned in Section 3.2. The following are a few application examples that our simulator has used. We illustrate them using the network of Figure 3 (a).

4.1. "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 (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
```

4.2. "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 algorithms. 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 of our address remapping and route setup schemes, the output of "traceroute" in our simulation system is somewhat different from its normal output. To understand its output in our simulation system, we need only 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 rule, we know it means link 2.) This anomaly is caused by the fact that ICMP TTL_expired packets are sent back at the previous hops while an ICMP port_unreachable packet is sent back at the last hop and that BSD uses different processing for these two different kinds of packets.

4.3. "Ftp" Client and Server on Any Node

"Ftp" clients and servers can readily work on our simulation system. 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 in UNIX environments. It sinks all data without writing it to disks, thus reducing unnecessary disk I/O operations on the simulation host.)

```
# ftp -node 3 -server ftp4 1.1.4.3
Connected to 1.1.4.3.
220 reluctance.eecs.harvard.edu 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)
```

Option "-node 3" tells this ftp client to run on node 3. Option "-server ftp4" tells it to connect to the ftp server on node 4.

The above throughput report confirms that our simulation system can simulate 10 Mbps links. This is because, after removing the bandwidth consumed by the IP and TCP header overheads, we can roughly achieve a throughput of 1017.04 KB/sec 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. Indeed, our simulation system can run any application program on any node in a simulated network. This capability allows network traffic, which need not originate from edge hosts, to be generated deep inside a simulated network.

4.4. "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., [11]) 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)
```

4.5. "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"[12] is a tool in the UNIX environment 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:

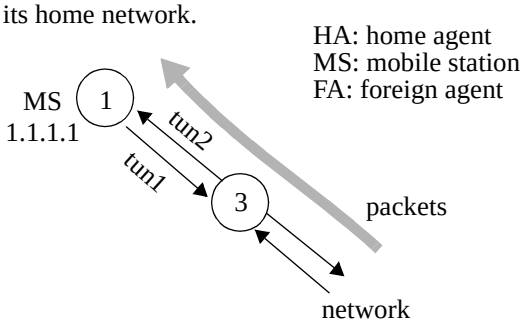
```
# trpt
631 ESTABLISHED:output
(src=1.1.3.3,3195,dst=1.1.4.3,8000)[75219d1..7521f85)@8e
754(win=8052)<ACK> -> ESTABLISHED rcv_nxt 8e754 rcv_wnd
100a4 snd_una 7512a49 snd_nxt 7521f85 snd_max 7521f85
snd_wl1 8e754 snd_wl2 74bf3c1 snd_wnd 10000 REXMT=3 (t_
rxtshft=0), KEEP=14400
```

4.6. Mobile IP Simulation Is Easy

Our simulator has been used to simulate home and foreign agents in mobile IP. Figure 5 illustrates how this simulator's architecture allows us to easily implement a home agent, which needs to intercept 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 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,

(a) Packets destined for the mobile station are transmitted on tun2 link when the mobile station is in its home network.



(b) When the mobile station is away from its home network, packets are forwarded along tun_redirect.

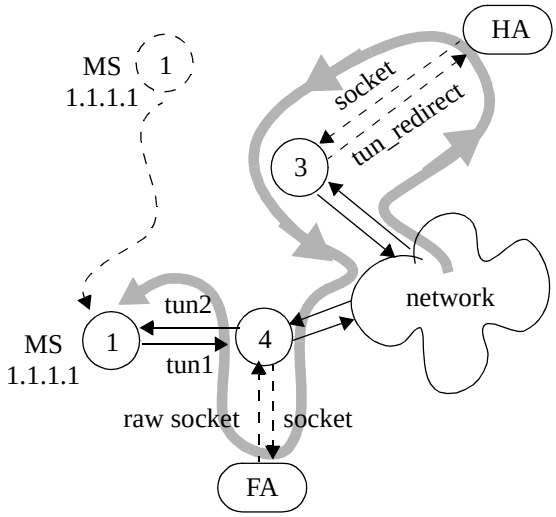


Figure 5: By changing a routing entry on node 3 from [1.1.1.3 -> tun2] to [1.1.1.3 -> tun_redirect]. the home agent can easily intercept packets destined for the mobile station. It then reads these raw packets from tun_redirect and sends them to the foreign agent via a normal datagram socket.

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. Our implementations for the home and foreign agents are simple. They 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.

5. Extension, Resource Requirements, and Performance

5.1. Support of 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, WFQ and CBQ) and/or queueing disciplines (e.g., drop-tail and RED). But normally a BSD UNIX kernel supports only FIFO and drop-tail. Thus a TCP/IP network simulator constructed using BSD UNIX and based on our methodology can not simulate this kind of network. To solve this problem, the simulator can use the ALTQ tool [13], which allows a network interface to use a different packet scheduling method and/or queueing discipline in a UNIX kernel.

5.2. Memory Requirement

Since application programs running in our IP network simulation system are all real independent programs in BSD UNIX environments, the simulation host system's memory requirement is proportional to the number of application programs running on the system. 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 BSD UNIX environment, we have found that the virtual memory mechanism provided in BSD UNIX environments 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 be present in the physical memory. For example, on a PC with 256 MB physical memory and 300 MB disk swap space, we can support up to 500 TCP connections with 1,000 ftp and ftpd programs without any page in and page out activities.

5.3. Performance

Often a simulator becomes slower and slower when simulating more and more nodes, links, and traffic generators. In order to report comparable performance, we normalize our IP simulator's speed with respect to the number of nodes it needs to simulate. On a 200 MHZ PentiumPro PC, our kernel-version IP network simulator can simulate a node faster than a real node in a network composed of 10 Mbps links. For example, when simulating a network with 40 nodes, 82 10Mbps links and 288 ftp-ftpd pair TCP traffic generators, our IP simulator is 3 times slower than the real network. This means that it can simulate a node 40/3 times faster than a real node. When comparing our IP simulator's speed with that of ns [6], our IP simulator is about 20 times slower than ns for the same configuration. Our IP network simulator is slower than ns because almost everything our simulator runs is real, and therefore it needs to execute every instruction in the real programs and in the real-life TCP/IP stack.

Although our IP simulator cannot run simulations as fast as ns, in studying TCP/IP network performance, network researchers usually need only simulate 5 to 10 minutes of the real network in order to gather stable performance data (e.g., in [14, 15]). Generally speaking, a total simulation time, in real time, on the order of a few hours is still acceptable.

Since our simulator approach can offer some unique advantages that are difficult for other simulator approaches to achieve (see the application examples presented in Section 4), our approach could serve as an alternative for those who need these unique advantages but for whom simulation speed is not their most important or sole concern.

6. Comparison with Other Approaches

Among other approaches, Dummynet [16] most resembles our simulator. Both Dummynet and our simulator use the native TCP/IP code on the simulation host. 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 host. 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, as they do not come from any link

OPNET [1], REAL [10], and ns [6] represent traditional network simulation approaches in which the thread-supporting event scheduler, application programs, host protocol implementation, and a network simulator (to simulate routers and links) are all compiled together to form a single complex program. Due to this enormous complexity, such a simulator tends to be difficult to develop, debug, verify, validate, and extend. The lack of API support between the simulator and application programs also limits its usage.

The simulator that we reported in [17] was developed to simulate an ATM network. It used the opaque network cloud simulation model because IP processing on the internal ATM switches is not required.

7. Conclusions

We have described a simple methodology for constructing a TCP/IP network simulator with minimal time and effort. The constructed simulator performs high-fidelity simulations of IP networks by executing real-life BSD TCP/IP code on the simulation host. It is extensible because any existing or future application program can be developed and run on any node in a simulated network without the need to recompile the simulator. Furthermore, because each node provides the standard UNIX API to the application programs, these programs' simulation implementations can be the same as their real implementations on a UNIX machine, and therefore the real system can reuse these programs' simulation implementations. Because of its extensibility, the simulator is able to study application-level performances of distributed systems such as "active network," "intelligent mobile agents," "mobile IP," and "virtual private network."

Acknowledgment

This research was partially supported by Nortel, Sprint, Air Force Office of Scientific Research Multidisciplinary University Research Initiative Grant F49620-97-1-0382, and National Science Foundation Grant CDA-94-01024.

References

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

[2] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden, "A Survey of Active Network Research", IEEE Communication Magazine, January 1997, p.80-86

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

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

[5] C. Scott, P. Wolfe and M. Erwin, "Virtual Private Networks", O'Reilly & Associates, 1998

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

[7] A. Chapman and H. T. Kung, "Enhancing Transport Network with Internet Protocols", IEEE Communication Magazine, May 1998, pp. 100-104

[8] L-W Lehman, S. Garland, and D. L. Tennenhouse, "Active Reliable Multicast", IEEE INFOCOM'98, 29 March - 2 April 1998

[9] U. Legedza, D. J. Wetherall, and J. Guttag, "Improving the Performance of Distributed Applications Using Active Networks", IEEE INFOCOM'98, 29 March - 2 April 1998

[10] S. Keshav, "REAL: A Network Simulator", Technical Report 88/472, Dept. of computer Science, UC Berkeley, 1988. (http://netlib.att.com/~keshav/papers/real.ps.Z). Simulator source available as ftp://ftp.research.att.com/dist/qos/REAL.tar

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

[12] Trpt, UNIX manual page (8)

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

[14] S. Floyd and V. Jacobson, Random Early Detection gateways for Congestion Avoidance, IEEE/ACM Transactions on Networking, V.1 N.4, August 1993, p. 397-413

[15] A. Romanow, and S. Floyd, "Dynamics of TCP Traffic over ATM Networks", ACM SIGCOMM '94, August 1994, pp. 79-88

[16] L. Rizzo, "Dummynet: a simple approach to the evaluation of network protocols", Computer Communication Review, Vol. 27, No. 1, p.31-41, January 1997

[17] H.T. Kung and S.Y. Wang, "Zero Queueing Flow Control and Applications", IEEE INFOCOM'98, 29 March - 2 April 1998, pp. 192-200