A TCP/IP Network Emulator

K. W. Lien, J. S. Reeve
Communications Research Group, School of Electronics and Computer Science
University of Southampton SO17 1BJ, United Kingdom
Emails: kwl02r@ecs.soton.ac.uk, jsr@ecs.soton.ac.uk

Abstract

In this paper, a Linux based framework of TCP/IP network emulator is introduced. Several advantages can be noted. Firstly, the maintenance of large numbers of processors is unnecessary. Secondly, compared with simulators constructed with conceptual codes, our emulator framework makes it easier to test the interaction and behaviour of TCP/IP in real Linux network environments. Thirdly, the wired network topology is fully controlled by a single processor, enabling us to separate TCP/IP behaviour over the wireless network, which helps distinguish performance functions that occur due to noisy wireless links. The framework was tested on two Linux processors over an IEEE 802.11b wireless network. The simulated outputs showed that the complex topology of the heterogeneous network was "realistically" constructed. Hence, the emulator could help TCP research in the future.

Keywords: TCP, Network Emulator, Simulation

1. Introduction

Current wireless networks usually incorporate wired backbone networks, and applications use the client server model, in which the server (file storage) is located in the wired network (fixed host) and the service file is requested from the mobile host. The role of TCP under the application layer is to provide stable and correct end-to-end data transmission, so that TCP interaction with other wireless protocols has been the subject of widespread study to enhance data access speed in wireless communication. Performance evaluation of TCP is usually a challenging task in heterogeneous environments, so network simulation tools are widely used for this purpose.

Our emulator separates the simulated topology into two parts: wired network simulation and wireless network emulation[1]. In the wired network simulation, the complex topology of wired network is simulated in a Linux processor (simulated host) by reusing the real Linux TCP/IP stacks multiple times. Hence, an almost "realistic" environment of wired networks is created. The network traffic is then redirected from the simulated host and connected with the real wireless network. Due to the high variation in wireless networks, a real wireless network is chosen to form a wireless network topology. We claim that these two parts assemble into an almost "realistic" simulated environment. It is especially useful for the end-to-end performance estimation of wireless applications or TCP/IP at which data transmission from the fixed host crosses the complex network cloud and ends at the mobile host. The emulator is easily constructed and a fully open source under the GNU General Public Licence (GPL)[2]. It can contribute the future TCP simulations over the wireless network.

This paper is organized as follows. In section 2, we first investigate features of numbers of simulators in detail. In section 3, we introduce the main features of our emulator and in section 4 we describe the architecture of the emulator. In section 5, we introduce the main concepts of our implementation and execution results. In section 6, we conclude our work.

2. Background Work

Network simulation tools can be classified into two types: the network simulator and the network emulator. Generally, a simulator is constructed in a single processor. High flexibility and easy extension are two main features of the simulator. It is usually used for the testing of new network protocols. In contrast with the simulator, the emulator can be

[1] The "emulation" is described as the network simulation over the real network environment.
[2] The emulator source code is currently available by email request.

constructed in a processor or multi-processors. It uses for the research what is considered realistic output.

Two technologies are mainly used to construct the simulator or the emulator: using the "conceptual network protocol" [1], [2] and reusing "the real network kernel" of the operating system (OS) [3]-[7]. Simulators designed with conceptual protocol are usually implemented as an individual program, just like a network application. It is easily set up and integrated. The main drawback of this kind of simulator is that the investigation of the interaction and the behaviour of the protocols over the real network are highly restricted. Compared with the simulator constructed with the conceptual protocol, it is easier to observe the protocol behaviour of the real network in the simulator designed with the "real network kernel". However, this kind of simulator is OS or OS kernel version specific. Mahrenholz [8] has designed a new type of emulator combining the ns-2 simulator [1] with the real network for wireless emulation. The restriction of this emulator is that the conceptual wireless modules are implemented for the wireless simulation, so the interaction of TCP/IP with the real wireless environment might not be truthfully demonstrated.

3. Emulator Features

Our simulation environment is separated into two parts: a simulated wired network and a real wireless network as shown in Fig. 1. A simulated wired network is fully constructed on a Linux processor (simulated host), this host acts as a "wired network box" in which wired network topologies are simulated by reusing the real Linux TCP/IP stack of the simulated host multiple times. In this way, the fixed host (FH), routers and links are simulated inside this "network box". There are two main advantages. Firstly, wired network parameters, such as link features, queuing disciplines and TCP settings are controlled through the simulated host. Secondly, the complex topology of wired networks is simulated at one processor; hence, the maintenance of large processor arrays is unnecessary.

The simulated host then connects with the real wireless network, instead of the simulated wireless network because of the emulator is designed to realistically construct a heterogeneous environment for the TCP/IP performance evaluation and behaviour observation. Therefore, through the connection of real wireless network, wireless features, such as quick handover and radio jamming are naturally included. Hence, the emulator could closely imitate the real network environment. Moreover, through the connection with different types of wireless network, the emulator can be used for different TCP performance estimates over different wireless protocols.

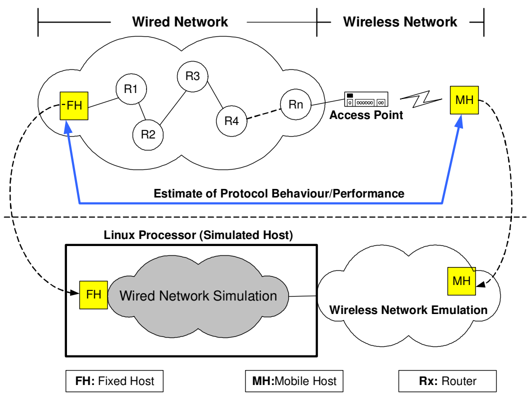

Fig. 1 The generality view of how an emulator forms a heterogeneous network environment.

The emulator is Linux module containable. Even though a "virtual wired network" is constructed inside the simulated host, the Linux kernel of TCP/IP architecture is fully maintained. This means that the Linux network modules are containable with our emulator framework. For instance, different fair queuing disciplines could be selected for specific link simulations. Most of them have been implemented as Linux modules. Hence, our emulator is flexible for different simulated topologies by only loading the selective modules and without compiling the new kernel.

Numbers of simulated network parameters are integrated through the standard Linux network interface. For instance, optional TCP mechanisms, such as selective acknowledgement (SACK), TCP timestamps and the window size, are adjustable through the system call interfaces under the directory "/proc/sys/net/". Two main advantages are displayed: firstly, users learn effortlessly to set the parameters for the emulator; secondly, existing Linux "ManPages" are naturally included as documentations of the emulator. Hence, both user learning time and emulator development time are reduced.

Linux network applications and utilities are implemented for simulation and analyzing tools. For instance, "ftp" is used to generate a simulated traffic flow and "ping" is used for round-trip time (RTT) measurements. The advantage of this feature is that the tools used are precisely those used in monitoring real networks.

4. Emulator Architecture

The emulator consists of five main parts: 1. packet re-routing, 2. packet arrival, 3. packet departure, 4. link simulation and 5. packet drop generation. Fig. 2 shows the simple relationship among these five parts, which are described in detail below.

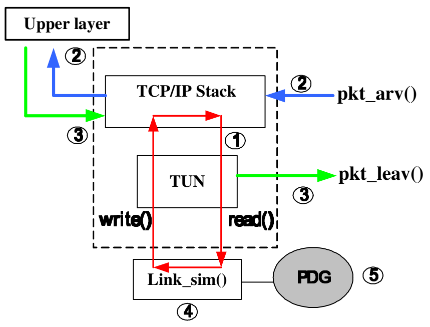

Fig. 2 The five main parts of the emulator.

4.1. Packets Route In-Out the TCP Stack

The emulator forms a virtual wired network by re-routing packets in and out of the TCP/IP stacks of the Linux simulated host multiple times. The framework presented in the Harvard network simulator [3] is re-designed here. Based on this framework, the "Private virtual IP address" mechanism is implemented to construct the virtual nodes. The "As-seen-by-node(i)" algorithm is used to form multiple routing paths between virtual nodes and TUN virtual devices. Each TUN device acts as an individual physical network device. Through the above mentioned mechanisms, a network topology is built up and network traffic is re-routed between the TCP/IP stacks and TUN devices, as displayed in Fig. 2 (label 1). Therefore, a complex wired network is constructed in one Linux processor.

Even though the concept of using virtual nodes and virtual devices inside the kernel for network simulation is not novel and has been implemented in [3]-[7], our framework differs from other projects in the following points: firstly, the mechanisms presented in [3] were implemented in our research from the FreeBSD to the Linux operating system; secondly, we further enhance the mechanisms connecting with the real wireless network through two additional functions described in section 4.2 and section 4.3; thirdly, the architecture of the Linux TCP/IP stacks is fully maintained so that the emulator can incorporate with Linux network modules for function extensions.

4.2. Packet Arrives at the Simulated Host

The pkt_arv() function deals with the emulated traffic incoming from the real network. It starts when the packet is received by the Linux Network Address Translation (NAT) module and ends when the packet is received by the application. The flowchart summarizing the algorithm for packet arrival in Fig. 3 is described below.

1. Simulation of non-executing state: If the emulator is not executed on the simulated host, the packet is processed as normal processes under the Linux processor.

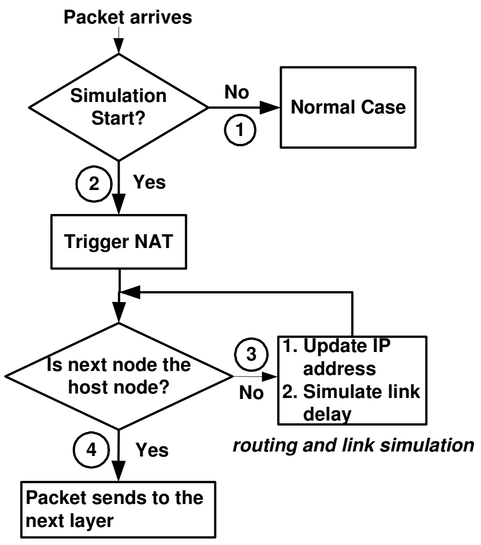

Fig. 3 The flow chart for a packet arriving at the simulated host.

2. Simulation of executing state: If the simulation is executed, nat_trig() function is called to invoke the Linux NAT module and delivers the "Private virtual IP address" information into the NAT module. Then, the function of the NAT prerouting chain is triggered to rewrite the packet's IP address as a specific "virtual address". In this way, incoming packets from a real network can be connected with the "As-seen-by-node(i)" routing mechanism and go to the "route state" as described below. It is important to note here that the emulator reuses any of the existing Linux functions instead of rewriting them, so that the development time is reduced and additional Linux functionalities can be used.

3. Route state: When the packet goes into the route state, the packet starts to re-route between TCP/IP stacks and TUN devices as mentioned in section 4.1. Each time the packet arrives at the TCP/IP stacks, the packet's next routing path is referenced from the "As-seen-by-node(i)" mechanism setting in the kernel routing table. If the next node is the virtual host, the pkt_arv() function leaves the route state directly and goes into the host state. If the next node is a virtual router, the packet re-routes again. Hence, if 10 virtual routers are set in the network topology, route state will be executed 10 times. Each time the packet arrives at the TUN device, the link_sim() function is invoked to deal with the link relative simulation. A detailed description of the link_sim() function is given in section 4.4.

4. Host state: The packet is prepared to pass up to the upper layer (i.e. TCP, UDP or ICMP).

4.3. Packet Leaves the Simulated Host

The pkt_leav() function deals with the emulated traffic outgoing from the application layer. It starts when the packet is received by the Linux TCP/IP stacks and finishes when the packet is received by the Linux NAT module. The flowchart summarising the algorithms for pkt_leav() in Fig. 4 is described below.

1. Route state: It deals with the same tasks of route state as described in the pkt_arv() function.

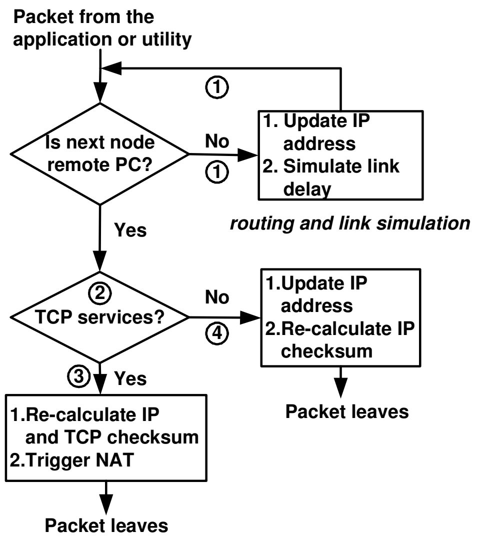

Fig. 4 The flow chart for a packet leaving the simulated host.

2. Judge state: When the packet leaves the route state, it means that the next node is the remote PC. Hence, additional tasks must be done before the packet is injected into the real network. A main task here is to differentiate varieties of service types. For instance, if the network service type belongs to TCP, the packet is delivered to the TCP state for additional processes. If the service type belongs to other protocols, the packet is delivered to the relative state for additional processes. Here, the Internet Control Message Protocol (ICMP) service is used as an example.
3. TCP state: When the packet goes into the TCP state, the IP and TCP checksum of the packet is first re-calculated. Otherwise, the packet will be dropped by the router or remote PC due to the checksum error. Then, the nat_trig() function is implemented again to invoke the Linux NAT module and asks for the IP address redirection. nat_trig() delivers the remote IP information into the NAT module, in which the function of NAT postrouting chain is triggered to change the packet's IP address from the "Private virtual IP address" to the remote PC's address. After leaving the NAT module, the packet is injected into the real network.
4. ICMP state: In this state, the packet's IP address is first updated from the "Private virtual IP address" to the remote PC's address. Then, the packet's IP checksum is re-calculated and injected into the real network. Two points are worth discussing here: firstly, the Linux NAT module is not implemented here because the function of NAT postrouting chain is not allowed to serve addresses update of ICMP packets; secondly, since the packet belonging to the different protocols is separated, it is unnecessary to re-calculate the TCP checksum here. Hence, the emulator overload is reduced, especially when long term simulation is demonstrated.

4.4. Link Simulation

The link parameters, of which there are many types, such as link delay and link bandwidth, are simulated in the link_sim() function, which is triggered when packets arrive at the TUN device. After packets arrive at each TUN device, the user space program invokes the read() system call to gather packets from the TUN device and then store packets into user space buffers. The simulations of link bandwidth and link delay are executed at this moment. When the link simulation is done, the write()system call is implemented to pick up packets from buffers and restored packets to the TUN device. The link simulation is accomplished between each read() and write() event as shown in Fig. 5.

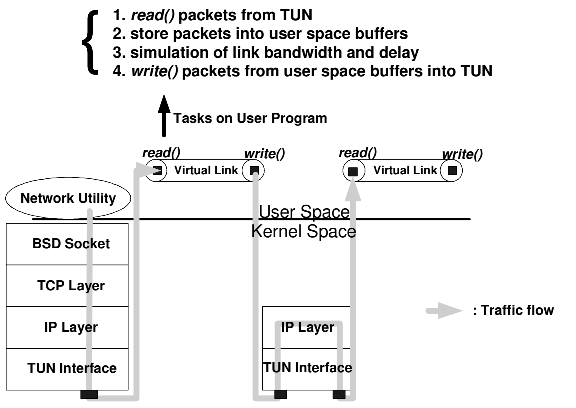

Fig. 5 A packet stored time between the read() and write() system call is used to do the link simulation.

4.5. Packet Drop Generator

The Packet Drop Generator (PDG) simulates packet loss by dropping packets on the link. When the packet is received by the virtual link, the link PDG is triggered to decide whether this packet is passed to the next node or is dropped. In the emulator, each link can be defined by the different PDG to simulate the specific simulated topology. A uniform distribution variable is used to determine the dropped packet, in which all packets have equal lost probability to simulate the topology whereby drop occurs randomly due to the environment. Usually in TCP simulations, packet loss during the TCP three-way handshaking step is avoided. Our PDG model uses a flag (S_flag), which needs to be set if SYN packets are not to be dropped. The following pseudo-code is implemented at virtual links to simulate the packet drop situation.

Fig. 6 shows the PDG demonstration on two SACK TCP hosts over 30 seconds in which different packet drop rates are set at the specific link, and “ftp” application is used to generate the traffic. The high packet drop rate could cause packet retransmission; hence, the lower sequence number is presented.

```
random_value=uniform_dist(100)
get protocol type from pkt header
if (protocol=TCP)
   if (S_flag)   /*skip SYN packets*/
      if ((SYN is set at the packet header)
         or (random_value>link_error_rate))
           pass packet to the next node
      else
           drop the packet
   else
      if (random_value<link_error_rate)
         drop the packet
else   /*non-TCP protocol*/
   if (random_value<link_error_rate)
      drop the packet
```

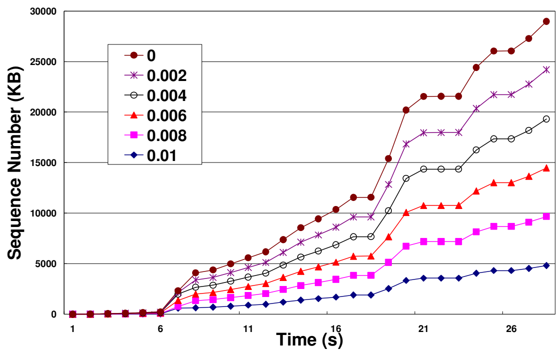

Fig. 6 PDG implementation - The observation of sequence number based on SACK TCP at different packet drop rates.

5. Emulator Execution Results

The emulator has been tested on two Redhat Linux processors with kernel version 2.4.20. One Linux processors is constructed as a simulated host and the other acts as a mobile host. Three virtual nodes (one host, two routers) are implemented inside the simulated host as wired networks. The bandwidth and delay of each virtual link are set at 10Mbps and 7 milliseconds (ms). The wireless network is implemented over the IEEE 802.11b. The implemented network topology is shown in Fig. 7.

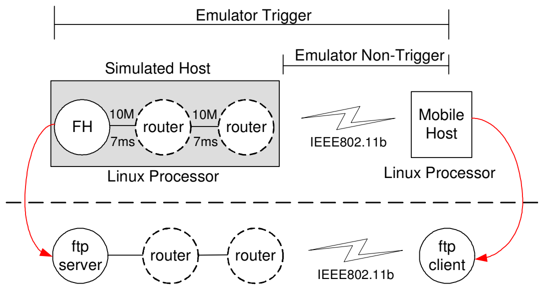

Fig. 7 Network topology for experiences.

5.1. Detection of Round-Trip Times

“ping” utility is the most popular way to test the round-trip time (RTT) between two hosts. It is also used to test whether the wired networks setting of the simulated host is correct or not. The first “ping” example below demonstrates the estimates of RTT between MH and the simulated host without executing the emulator. Hence, the reported outputs are the RTT through the real wireless network between the simulated host and MH.

```
# ping 152.78.X.X -w 100
PING 152.78.X.X (152.78.X.X): 56 octets data
64 octets from 152.78.X.X:icmp_seq=0 ttl=62 time=0.65 ms
64 octets from 152.78.X.X:icmp_seq=1 ttl=62 time=0.82 ms
64 octets from 152.78.X.X:icmp_seq=2 ttl=62 time=0.77 ms
……
--- 152.78.X.X ping statistics ---
100 packets transmitted, 100 packets received, 0% loss
round-trip min/avg/max = 0.63/0.78/0.92 ms
```

The second “ping” example below demonstrates the estimates of RTT between MH and the simulated host when the emulator is triggered.

```
# ping 152.78.X.X -w 100
PING 152.78.X.X (152.78.X.X): 56 octets data
64 octets from 152.78.X.X:icmp_seq=0 ttl=62 time=32 ms
64 octets from 152.78.X.X:icmp_seq=1 ttl=62 time=33 ms
64 octets from 152.78.X.X:icmp_seq=2 ttl=62 time=35 ms
……
--- 152.78.X.X ping statistics ---
100 packets transmitted, 100 packets received, 0% loss
round-trip min/avg/max = 30.5/33/35.5 ms
```

Since the propagation delay is set as 7ms for each virtual link, total RTT between FH and MH should be around 29ms (7*4+0.78). However, the reported RTT is larger than this value as shown in the second “ping” example. There are two reasons. Firstly, the extra transmission and processing delay inside the Linux kernel is necessary. Secondly, extra time overload is taken to re-calculate the IP checksum before packets are injected into the real network.

5.2. ftp Application for Traffic Generation

The emulator simply implements "ftp" to generate the traffic flow. The example below shows that a successful ftp connection has been created between the server (FH) and the client (MH). The "get" command is used to request the file "testfile.ps" from the server. In this way, the traffic transmission is generated easily on the emulated network. The "Passive FTP" is implemented here instead of "Active FTP" to solve port mismatch problems due to the fact that ftp server of the emulator is behind the Linux NAT module.

```
# ftp 152.78.X.X
Connected to 152.78.X.X.
230 User logged in.
Passive.
Into passive mode
ftp > get testfile.ps
152.78.X.X.3115 connection.
150 Opening BINARY mode data connection for
'testfile.ps' (3036456 bytes).
226 Transfer complete.
3036456 bytes received.
```

5.3. Monitoring Network Status

"tcpdump" utility is used to monitor packet transmission on a link. The example below shows that the packets are monitoring on a MH eth0 device. The traffic flow is generated by "ftp" connection, as described in section 5.2. Through outputs from "tcpdump", TCP parameters, such as the packet sequence number, the TCP window size, the packet drop situation and packet size, are clearly observable and gathered for statistics. This is useful for performance evaluation and behaviour observation of TCP over heterogeneous networks. "tcpdump" is not only used for traffic monitoring on an eth0 device. We further implement "tcpdump" to monitor traffic flow at each TUN device. Hence, we could advance the study of TCP behaviours inside inner networks.

```
# tcpdump -i eth0
tcpdump: listening on eth0
10:37:27.465155 152.78.X.X.32776 >
```

6. Conclusion

In this paper, a simple framework for a TCP/IP emulator has been introduced, in which only two Linux hosts are necessary to form almost "realistic" mixed wired and wireless networks. The wired network topology is fully constructed and controlled from a Linux host. We further implement two additional functions to connect the simulated host with another Linux host located in the wireless network. The emulator framework also incorporates Linux module interfaces so that development costs are reduced and additional functionalities readily included. The simulation parameters are adjusted through the normal Linux network interfaces. We have incorporated a packet drop generator so that we can simulate wireless links that have a given bit error rate. Graphs of packet sequence number against time show that the introduction of wireless link errors still results in a connection with constant but reduce bandwidth.

The emulator has been demonstrated through numbers of popular network utilities. The results show that a complex topology of heterogeneous network is realistically constructed and the simulated outputs are easier to monitor. Hence, the emulator can contribute to future TCP/IP study over heterogeneous networks.

Acknowledgment

The authors would like to thank Dr. S. Y. Wang. who gave us detailed descriptions of his simulator under BSD UNIX.

References

[1] S. McCanne and S. Floyd. ns-LBNL Network Simulator. http://www.isi.edu/nsnam/ns/.

[2] OMNet++ Object-oriented Discrete Event Simulation System. http://www.omnetpp.org/.

[3] S. Wang and H. Kung, "A Simple Methodology for Constructing Extensible and High-Fidelity TCP/IP Network Simulator," *in Proc.of IEEE INFOCOM'99*, New York, USA, Mar. 1999, pp. 1134--1143.

[4] L. Rizzo, "Dummynet: A Simple Approach to the Evaluation of Network Protocols," ACM Computer Commun. Review, vol. 27, no. 1, pp. 31-41, 1997.

[5] L. Brakmo and L. Peterson, "Experiences with Network Simulator," *in Proc. of SIGMETRICS*, Philadelphia, USA, May 1996, pp. 80--90.

[6] S. Wang *et al.*, "The Design and Implementation of the NCTUns 1.0 Network Simulator," Computer Networks, vol. 42, no. 2, pp. 175-197, June 2003.

[7] M. Carson and D. Santay, "NIST Net: a Linux-based Network Emulation Tool," ACM SIGCOMM Computer Communication Review, vol. 33, no. 3, pp. 111-126, 2003.

[8] D. Mahrenholz and S. Ivanov, "Real-Time Network Emulation with ns-2," *in Proc. of 8-th IEEE International Symposium on Distributed Simulation and Real Time Applications*, Budapest Hungary, Oct. 2004.