ONE: The Ohio Network Emulator

Mark Allman, Adam Caldwell, Shawn Ostermann
mallman@lerc.nasa.gov, adam@eni.net
ostermann@cs.ohiou.edu

School of Electrical Engineering and Computer Science
Ohio University

TR-19972
August 18, 1997

This work sponsored in part by NASA Lewis Research Center.

Abstract

Studying network protocols and distributed applications in real networks can be difficult due to the need for complex topologies, hard to find physical channels (e.g., satellite channels), and conditions beyond the control of a researcher (e.g., queue sizes). Network emulators can provide a controlled and reproducible environment for network testing. This paper discusses *ONE*, a network emulator we have written and tested.

1 Introduction

Network emulators, like network simulators, allow researchers to create network topologies and conditions that are difficult to achieve in a reproducible manner on production networks. Studying network protocols and distributed applications in the relatively simplistic environment that simulators and emulators furnish can be helpful when investigating subtle network and protocol interactions. In addition, these environments can provide access to network environments that are not easily found in the production Internet, such as satellite links.

Network simulators do not generate real network traffic, but rather model traffic and major network components internally. The strength of network simulators (such as REAL [Kes88], NetSim [Hey90] and LBL's *ns* [MF95]) is that they allow study of complex network topologies that are difficult to create using real networks. In addition, simulators are not limited by the speed of the hardware that makes up a network. Therefore, simulators provide an environment for studying high speed networks. On the other hand, simulators usually run an independent specification of network code, rather than the code used in real networks. This can cause a simulator to fail to mimic subtleties in the real code.

Two simulators, x-Sim [BP96] and *dummynet* [Riz97], allow direct execution of production network code. The disadvantage of both of these tools is that they are limited to one operating system (x-kernel and FreeBSD, respectively) to which they require modifications. However, both of these tools provide a useful environment for testing the same code in both a simulator and over real networks.

Network emulators alter real network traffic between nodes in a physical network to model various network configurations. The strength of network emulators is that researchers can easily move network code between a real network and an emulated network. Furthermore, the code being tested can run on any type of system without modification and any testing and analysis software will work in both environments. The drawbacks of emulation are twofold: the speed of the emulated network cannot be greater than the speed of the underlying physical network, and complex topologies are difficult to create because they must be physically constructed.

The remainder of this paper is organized as follows. Section 2 describes the components of the network that are modeled by *ONE*. Section 3 discusses *ONE*'s configuration. Section 4 provides verification that *ONE* is providing an accurate model of a given network. Finally, section 5 provides conclusions and outlines future work.

2 Modeling a Network

A simple network topology is given in figure 1. A host on one network communicates with a host on the other network via the two intervening routers. The area within the dotted line is the portion of the network we wish to emulate. Figure 2 shows the emulated version of the network in figure 1. The emulator in this figure is a Sun workstation, running the Solaris operating system[1] and our emulator software. The machine has two network interfaces. *ONE* passes packets between the two networks, just as the two routers would do in figure 1.

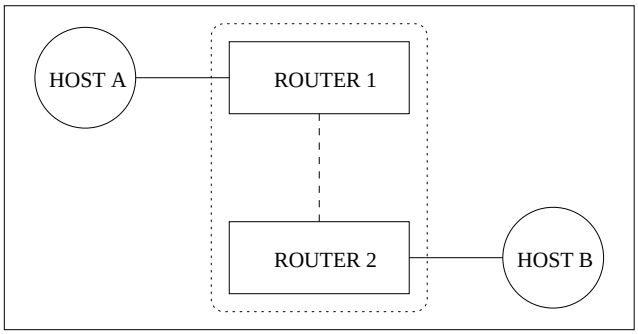

Figure 1: Simple Network Topology
Each host is connected to a physical network containing a router. The routers are connected via another channel. The two hosts communicate via the two routers.

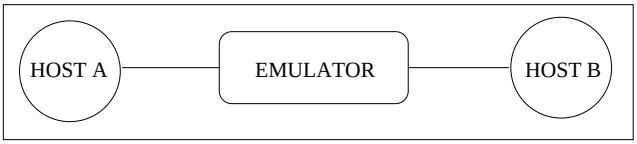

Figure 2: Emulated Network Topology
ONE has two network interfaces that connect it to two separate networks. The hosts on these networks communicate via the emulator. *ONE* alters the network traffic being forwarded based on several user-configurable parameters.

ONE models the routers and intervening network by delaying packets arriving on one network interface before forwarding them to the other network. *ONE* also provides congestion loss according to its configuration. The delay a packet experiences is based on the packet size and the configuration parameters given by the user. The following three components of packet delay are modeled.

Transmission Delay:
The transmission delay is the amount of time it takes a network node to transmit a packet onto a given channel. The transmission delay is determined using the channel bandwidth and the packet size (according to formula 1).

[1] We have tested *ONE* under Solaris 2.2, 2.4 and 2.5.

$$trans\_delay = \frac{packet\_size}{bandwidth} \tag{1}$$

Queuing Delay:
Queuing delay occurs when a packet arrives at a router which is already busy transmitting another packet. In this case, the packet is placed in a queue. The queue delay for a given packet P is dictated by the sum of the sizes of the packets in the queue when the P arrives. Formula 2 gives the queue delay from the n-th packet in the queue.

$$q\ delay = \begin{cases} 0, & n = 1 \\ \sum_{i=1}^{n-1} \frac{packet\_size_i}{bandwidth}, & n > 1 \end{cases} \tag{2}$$

Propagation Delay:
The propagation delay is the time it takes a packet to travel from one node to another along a physical channel. This component of the delay is configured by the user.

The sum of the above component delays is the amount of time *ONE* holds a packet before forwarding it. For example, assume two identical packets are sent through our emulator back-to-back. Assume the transmission delay is 1 second for each packet and the propagation delay given by the user is 0.5 seconds. The first packet will be delayed 1.5 seconds by *ONE* (1 second for transmission delay and 0.5 seconds for propagation delay). *ONE* will delay the second packet 2.5 seconds (1 second for transmission delay, 0.5 seconds for propagation delay and 1.0 seconds for queue delay).

3 Configuration

Our emulator delays packets being sent between the two networks using two user-configurable parameters.

- `linespeed`
  This is the bandwidth of the channel between the two routers in figure 1 (e.g., 100 Kilobytes/second).

- `propagation`
  This is the propagation delay of the channel between the two routers in figure 1 (e.g., 20 milliseconds).

The above parameters can be set for each network interface on the emulator. For example, setting the `linespeed` on a given interface subjects all packets arriving on that interface to the given bandwidth.

Additionally, *ONE* queues packets for transmission based on the following two user-configurable settings:

- `qsize`
  This is the size of the queue for the given interface (e.g., 50 Kilobytes).

- `memunit`
  This is the internal buffer size (memory allocation granularity) used to store packets in the queue (e.g., 256 bytes).

The queue size can be set for each interface, while the buffer size specified is for both interfaces. *ONE* uses fixed size buffers to store queued packets in order to model routers that employ a similar strategy. If, for example, `memunit` is set to 256 bytes, a 5 byte packet will use 256 bytes of queue space. Similarly, a 300 byte packet will use 512 bytes of queue space. `Memunit` can be configured to 1 byte, making the memory allocation equal to incoming packet sizes.

4 Experimental Results

The following is a presentation of several experiments to show that each aspect of our emulator is functioning properly. Also included is a realistic comparison of FTP tests run over *ONE* and equivalent tests run over the NASA ACTS satellite system.

le0 Propagation Delay (ms)	le1 Propagation Delay (ms)	Expected RTT (ms)	Observed RTT (ms)	RTT Difference (ms)
10	10	≈ 20	23.87	3.87
50	50	≈ 100	104.41	4.41
100	100	≈ 200	204.49	4.49
250	250	≈ 500	504.57	4.57
10	40	≈ 50	54.01	4.01
100	65	≈ 165	169.18	4.18
75	0	≈ 75	79.44	4.44

Table 1: Propagation Delay
This table compares the expected RTT with the values observed using the emulator for various combinations of propagation delays.

4.1 Propagation Delay Tests

To verify that our emulator correctly delays packets based on the configured propagation delay, we sent small ICMP ping packets from one network to the other via *ONE*. The emulator was configured with infinite bandwidth to remove the transmission time from the total delay experienced by the packets. Also, the interval between successive pings was large enough to ensure that no queuing delay was present. The results of our experiments are shown in table 1. The configured propagation delay was varied as shown in the first two columns of table 1. As shown in the last column of the table, the actual round trip time (RTT) the packets experienced was slightly longer than the expected RTT. This difference can be explained by the time the packet was on the network but not in the emulator (one RTT between the emulator and the sending host and one RTT between the emulator and the receiving host). Our emulator does not correct for this error, as the difference will change based on the type of physical channel used between the hosts and the emulator. If greater accuracy is needed, the user can slightly lower the propagation delay to offset the delay added elsewhere.

4.2 Transmission Delay Tests

To verify that *ONE* delays packets the proper amount of time based on the configured bandwidth, we again sent ICMP pings across our emulator. To ensure that only transmission delay was affecting the packets, the propagation delay was set to 0 ms. Furthermore, the inter-packet spacing ensured that no queuing delay was present. The bandwidth was configured to the values given in the first column of table 2 on the first network interface of our emulator. The second interface was given infinite bandwidth. The results of these tests are summarized in table 2. Again, the difference between the expected and observed RTTs can be explained by the time the packets were on the network but not in *ONE*.

Bandwidth (bytes/sec)	Packet Size (bytes)	Expected RTT (ms)	Observed RTT (ms)	RTT Difference (ms)
100	100	≈ 1000	1004.07	4.07
100	200	≈ 2000	2005.30	5.30
100	300	≈ 3000	3008.42	8.42
200	100	≈ 500	504.46	4.46
200	200	≈ 1000	1008.06	8.06
200	300	≈ 1500	1505.21	5.21

Table 2: Transmission Delay
This table compares the expected RTT with the values observed using the emulator for various combinations of bandwidth and packet size.

Packet Number	Expected RTT (ms)	Observed RTT (ms)	RTT Difference (ms)
1	≈ 1000	1005.96	5.96
2	≈ 2000	2005.51	5.51
3	≈ 3000	3006.52	6.52

Table 3: Queuing Delay
This table compares the expected RTT with the values observed using the emulator for each of three datagrams sent back-to-back.

4.3 Queuing Delay

To verify that our emulator delays segments properly based on the length of the queue, we used the UDPping[2] tool to send a burst of 3 UDP datagrams consisting of 100 bytes from a host on one side of our emulator to the echo port on a host on the other side. The emulator was configured to subject all packets arriving on its first interface to a bandwidth of 100 bytes/second. All packets arriving on the second interface were given infinite bandwidth.

[2]Available at `http://jarok.cs.ohiou.edu/software/`.

The propagation delay was set to 0 ms for all packets and the queues were large enough to handle well over 3 packets. Each packet in the burst should be delayed 1 second based on the transmission time. Furthermore, each packet after the first should be delayed another second for each packet ahead of it in the queue. The results of these tests are shown in table 3. These results are consistent with the way in which the packets should be delayed based on the configured bandwidth and the number of packets in the queue. The difference between the actual RTTs and the expected RTTs can again be explained by the time the packets spent on the network outside *ONE*.

4.4 Queue Drops

To show that *ONE* is properly dropping packets when the queue is full, we sent bursts of UDP datagrams through the emulator. We configured the queue to be 1024 bytes and the standard buffer size (`memunit`) to be 256 bytes. We configured the bandwidth such that the transmission delay was 1 second to ensure that the packets would be queued by *ONE*. We sent 6 back-to-back 100 byte UDP echo datagrams through the emulator. As expected, *ONE* consistently dropped packets 5 and 6.

4.5 Comparison with ACTS Network

While the above tests show that each aspect of our emulator works well when isolated, we wanted to ensure that all mechanisms work correctly in conjunction with each other. To show that the emulator provides a good approximation of a physical network, we repeated a set of FTP transfers over the NASA ACTS [AHKO97] satellite using the emulator. The results of these experiments are shown in figure 3. In both the ACTS and *ONE* tests, the TCP receive window was large enough to ensure that congestion loss occurred, provided that the transmission is long enough to fully open the window.

Due to the slow start algorithm [JK88], the 200 KB transfer was not able to utilize a full window and therefore, no congestion loss occurred. In the 1 MB and 5 MB transfers, TCP is able to to finish slow start and achieve a full window. This overwhelms the configured router queue. For each of these three data points, both *ONE* and ACTS showed similar throughput, indicating that *ONE* accurately models the physical network. Larger disparities for the two larger file sizes are due to subtle differences between the queueing mechanisms in the routers and *ONE*.

5 Conclusions and Future Work

We have developed a network emulator that accurately models the network configured by the user. While the emulator should not be a replacement for real network tests or simulations, we believe it is a powerful tool for studying distributed applications and network protocols in a controlled and reproducible environment. Furthermore, the emulator provides this environment to a wide variety of hardware and software systems.

Future work on the emulator includes developing a model for packet loss due to corruption. This will be useful in the study of protocols for lossy networks, such as satellite channels.

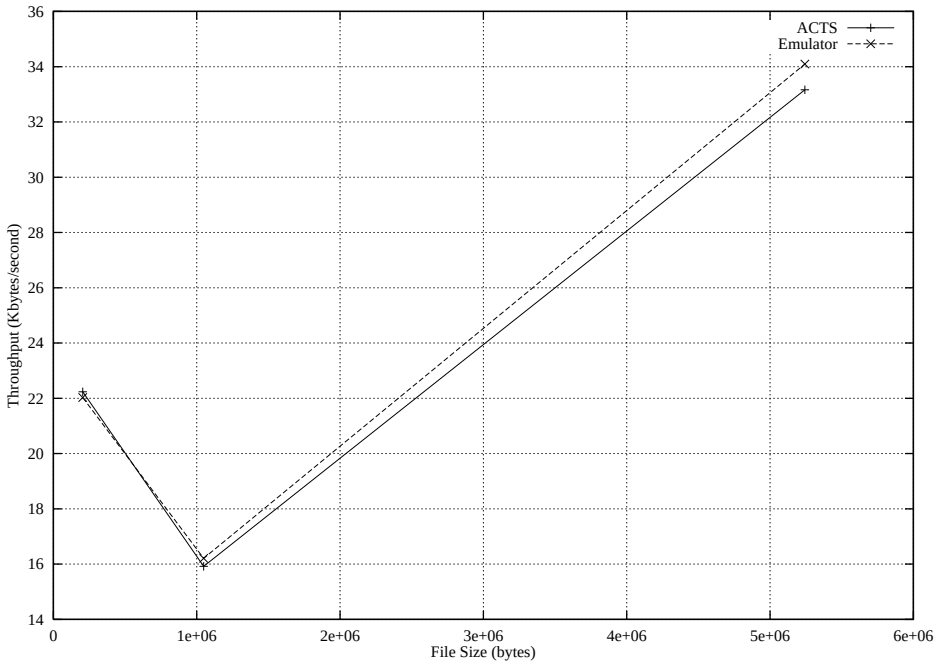

Figure 3: Emulator vs. ACTS
This figure compares throughput for file transfers of size 200 KB, 1 MB, and 5 MB between the ACTS network and *ONE*'s emulation of the network.

Additionally, alternative queuing strategies such as Random Early Detection [FJ93] should be implemented.

Acknowledgments

This work benefited from discussions with Hans Kruse and John Tysko. Edward Kroeze provided the needed hint for fixing a bug in our packet capturing routines. Finally, we would like to thank Chris Hayes for helping us work the bugs out of *ONE*.

References

[AHKO97] Mark Allman, Chris Hayes, Hans Kruse, and Shawn Ostermann. TCP Performance Over Satellite Links. In *Proceedings of the 5th International Conference on Telecommunication Systems*, March 1997.

[BP96] Lawrence Brakmo and Larry Peterson. Experiences with Network Simulators. In *ACM SIGMETRICS*, 1996.

[FJ93] Sally Floyd and Van Jacobson. Random Early Detection Gateways for Congestion Avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, August 1993.

[Hey90] A. Heybey. The Network Simulator. Technical report, MIT, September 1990.

[JK88] Van Jacobson and Michael J. Karels. Congestion Avoidance and Control. In *ACM SIGCOMM*, 1988.

[Kes88] Srinivasan Keshav. REAL: A Network Simulator. Technical Report 88/472, University of California Berkeley, 1988.

[MF95] Steven McCanne and Sally Floyd. NS (Network Simulator), 1995. URL http://www-nrg.ee.lbl.gov.

[Riz97] Luigi Rizzo. Dummynet: A Simple Approach to the Evaluation of Network Protocols. *Computer Communications Review*, 27(1):31–41, January 1997.