Design and Validation of a Software Link Monitor

Roman Chertov
Purdue University
rchertov@cs.purdue.edu

Sonia Fahmy
Purdue University
fahmy@cs.purdue.edu

ABSTRACT

This report gives a brief overview of how to monitor a link in an emulation testbed (e.g., DETER or Emulab) using a software-only approach, without the use of span ports or other hardware solutions.

Keywords

tcpdump, span ports, software hub, link monitoring

1. INTRODUCTION

Link monitoring is important when conducting experiments with denial of service (DoS) attacks, worms, congestion control protocols, and several other networking experiments. In a simulator, link monitoring can be trivially accomplished, without any performance repercussions. Real networks require a more sophisticated solution. An easy monitoring approach is to place a hub between the two endpoints of the link being monitored, and attach a third node to this hub to perform packet logging via *tcpdump*. If the nodes are connected by a switch or a router that provides a span port capability, then it is easy to mirror traffic between the two nodes to a third node. Such hardware solutions, however, are inflexible and have to be set up on an experiment by experiment basis. This is especially cumbersome in an emulation testbed such as Emulab (www.emulab.net), which is highly time- and space-shared.

In Spring 2004, we experimented with a hardware solution on the DETER testbed (www.isi.deterlab.net). Specifically, the DETER team set up a 100 Mbps hub between three test machines, in order to provide a link monitoring capability for an Intrusion Detection System (IDS) that we were evaluating with DoS attacks. The Emulab team has also reported that the Emulab testbed has a span port capability between any two nodes. However, using this capability requires manual setup before the experiment can be swapped in. Testbeds like Emulab and DETER, however, derive their strength from their flexibility and the minimal intervention required from their staff to aid with experiments. One should therefore be able to create a software-only link monitor that is easy to use and does not require special attention.

A naive software-only link monitoring solution is to run *tcpdump* on the receiver node or any node that is between the sender and receiver. As long as packet flows are not arriving at extremely high rates, and the test machines have ample capacity, this solution is the easiest to use. However, this solution becomes problematic when there is not enough capacity to log and process/forward arriving packets. In this case, the monitoring result will be a log file that does not contain all the arriving packets. In the worst case, the logging can interfere with forwarding, thus leading to packet drops and the introduction of artifacts into the experiment. In the next section, we will describe a simple approach that works well under high traffic loads, and retains the flexibility of the emulation environment.

2. SOFTWARE SOLUTION

Our approach is based upon the distribution of the forwarding and logging tasks. We observe that a machine that performs traffic forwarding should not run *tcpdump*; however, that machine will typically have enough capacity to duplicate and forward the traffic. Figure 1 depicts a logical view of how tasks are distributed. In order to monitor a link between two test nodes, two additional nodes are required. One node will act as a bridge/duplicator and the other node will log packets to a file.

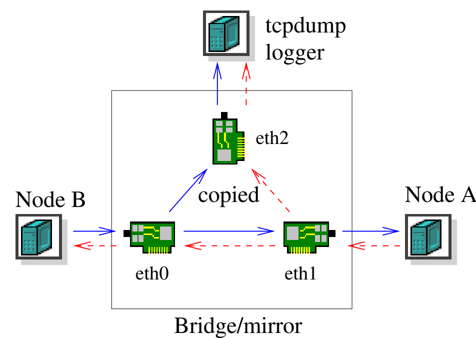

Figure 1: Anatomy of a software link monitor

Figure 2 demonstrates our setup on the DETER testbed. If the node performing logging is incapable of handling high loads but the bridge node is, then the measurements will be incomplete, but there will be no loss on the link itself. Loss at the bridging node is a more severe problem, and will introduce significant artifacts in the experiment. To address both concerns, it is essential that both the bridge and logging nodes be much faster than the sending nodes. On the DETER testbed, all of the test nodes have dual CPUs, but the default OS images use only one CPU.

To utilize this solution, our first step was to create a new OS image that was SMP capable and had a bridge module installed. The next step required modification of the bridge code to ensure that our solution worked on DETER and Emulab. The modification allows us to specify a mirrored interface which would receive all the traffic that passes through other interfaces. The switch will drop incoming packets if its *dst* MAC address is not in the set of MACs that can be reached by a port on which the packet arrives. To avoid packet drops by the backend switch, all of the mirrored packets

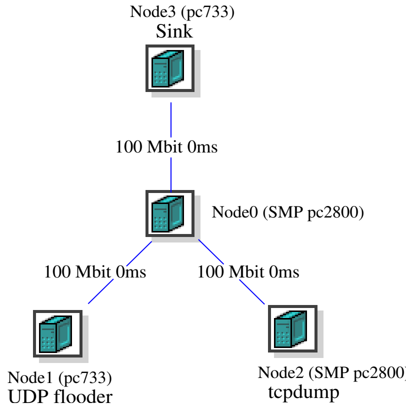

Figure 2: Validation topology used on the DETER testbed

must therefore have their destination MAC address modified to that of the logging node.

3. VALIDATION EXPERIMENT

Our baseline experiments have demonstrated that when *ttcp* (http://ftp.arl.mil/∼mike/ttcp.html) is used for file transfer over TCP, our link monitor can successfully log all of the packets at 11 Megabytes per second. Since file transfer over TCP is not as aggressive as a persistent UDP flood, we have devised UDP-based stress tests for our link monitor. Our "UDP flooder" sends UDP packets using a raw socket as fast as possible. We vary the packet payload size. The packet payload size is inversely proportional to the sending rate. High packet rates do not necessarily use up the entire bandwidth, but they impose high loads on the machines. This is because there is a certain overhead required for processing each individual packet.

To test the performance of our link monitor, we have created an experiment on DETER using the topology illustrated in Figure 2. We use *tcpdump* on the logging node and then use *tcptrace* to count the number of packets in the dump file. The reported packet count is compared to the number of *send()* calls that the flooding tool has made. The experiments are repeated for packet payload sizes of 10, 100, 500, 700, 1000, and 1400 bytes. For each payload size, the experiment is run 10 times and the average of the packet loss value recorded in the 10 runs is reported. Packet loss is computed as ((number of packets received by logging node − number of sent packets) × 100) / number of sent packets.

4. EXPERIMENTAL RESULTS

As expected, the highest packet losses occur when the packet sizes are the smallest. Figure 3 shows that for packet sizes closest to the MTU size, the percentage loss is minimal: for a payload size of 1400 bytes, the flooder can send 514691.5 packets per minute on the average, and the logger can receive 514681.3 packets per minute on the average. This implies that the logger was able to capture packets at just over 12 Megabytes a second (computed as number of packets × (packet payload size + header size) / 60 seconds per minute / 1,000,000).

At the other extreme, when the packet payload size is 10 bytes, the flooder was able to send 8978569.5 packets per minute on the average, and the logger could only capture 8908384.7 packets per minute on the average. Even though the packet rates are higher than in the case of a 1400-byte-payload, the logged throughput was only approximately 5.6 Megabytes per second since packet sizes are smaller. The loss was higher: approximately 0.8% in this case.

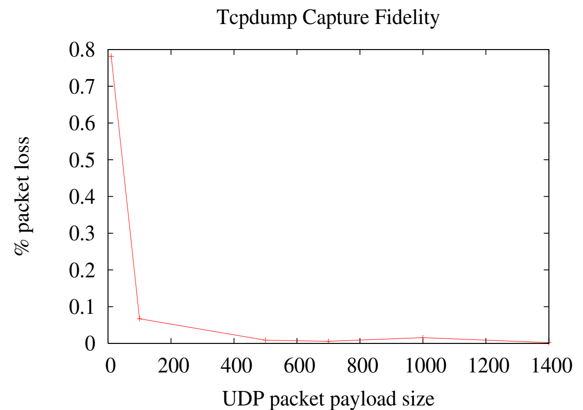

Figure 3: Tcpdump packet loss

5. CONCLUSIONS AND FUTURE WORK

Our results indicate that our software link monitoring mechanism can reasonably cope with the 100 Mbps transfer rates in our experiment. Experiments that do not expect to see extremely high packet rates can easily use the bridging solution to "tap" the links. If extremely high packet rates are expected in an experiment, additional tuning must be performed to eliminate packet loss. We have observed that *tcpdump* does receive all of the packets in our experiments, but does not log all of them. Packet logging fails when the CPU load is significant or the receive buffer overflows. As a result, the difference between received and logged packets increases as the packet rates increase.

Linux kernels 2.6.x support NAPI for the NIC device drivers. Using polling can decrease the CPU load as IRQ livelock will be eliminated. To compensate for delays in polling, the receive buffers must be configured to be large enough so as not to overflow before the next polling round. Our future work will include installation of the latest Linux kernel and creating a NIC driver configuration to eliminate packet loss during logging, even at extremely high packet rates.