Assignment 3 - Buffer Sizing

Goal

In this assignment, you will use Mininet to understand buffer sizing and how Mininet can be used for modeling real world research [1].

All Internet routers contain buffers to hold packets during times of congestion. The size of the buffers is dictated by the dynamics of TCP's congestion control algorithm. The goal is to make sure that when a link is congested, it is busy 100% of the time, which is equivalent to making sure the buffer never goes empty. Until 2004, the common assumption was that each link needs a buffer of size RTT x C where RTT is the average round-trip time of a flow passing across the link, and C is the data-rate of the bottleneck link.

A paper published at Stanford outlines a new "rule of thumb" for buffer sizing showing that a link with N flow requires a buffer size at most (RTT x C) / sqrt(N). This means smaller buffer sizes for links with large numbers of flows and potentially cost savings. The original paper included results from simulation and measurements from a real router, but not for a real network. Later, Neda Beheshti created a hardware testbed to test the buffer sizing results in the Internet2 backbone, and demonstrated it at Sigcomm 2008. We highly recommend watching the five-minute YouTube video of the experiment.

You will reproduce this experiment using Mininet.

Overview

Hardware Setup

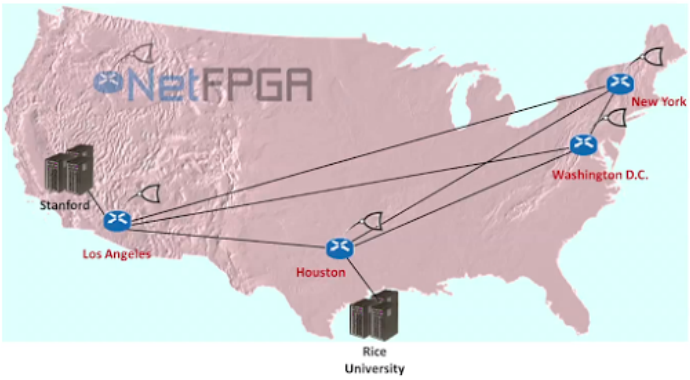

Figure 1: Hardware testbed on Internet2

In the hardware experiments, a number of TCP flows were started from two end-hosts at Stanford University to a server at Rice University (Houston, Texas), via a router in the Internet2 POP in Los Angeles (see Figure 1). The link from LA to Houston was constrained to 62.5Mb/s to create a bottleneck, and the end-to-end RTT was measured to be 87ms. Once the flows were established, a script ran a binary search to find the buffer size needed for 99% utilization on the bottleneck link.

Mininet Setup

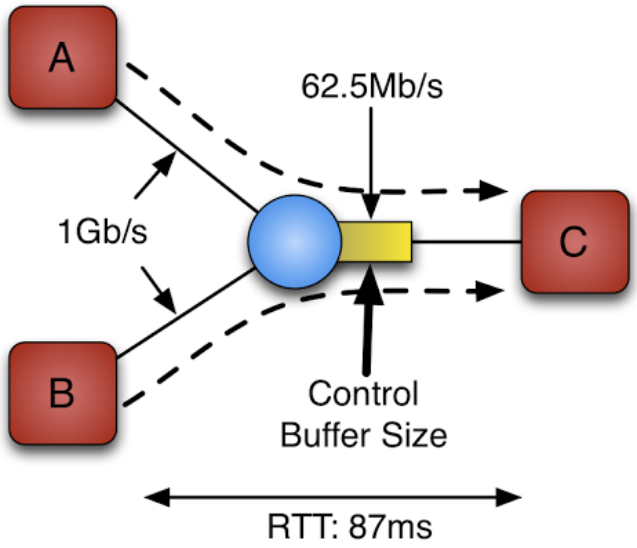

Figure 2: Software topology in Mininet

Since you don't have access to the hardware setup, your goal is to recreate key characteristics of the hardware topology in Mininet (see Figure 2). Mininet uses software rate limiters (htb from the "tc" suite) and software delay queues (netem) to emulate wide-area network links.

The goal is to show how the link utilization at the bottleneck link varies with the buffer size allocated to it. Your submitted code will use the following algorithm, and the provided code implements most of the steps for you:

For each number of flows N=1,2, ... [large number, up to 800]:

1. Start with a buffer size packets on the bottleneck link.
2. Generate N flows from A and B to C.
3. Wait until all flows start and reach full throughput.
4. Binary search the buffer size to determine the smallest buffer size (Bmin) to ensure at least X% link utilization on the bottleneck link (the constant TARGET_UTIL_FRACTIONin the code). For the hardware experiments this was 99%, but the starter code comes configured to 98% to increase the tolerance to measurement error. Repeat until the experiment has determined the minimum buffer size that leads to above-threshold throughput (say, just above 99%), to a precision of one packet.
5. For each candidate queue size: Record the throughput. Then choose the next buffer size. It should be the average of the smallest buffer size that was above-threshold (that, is maintained "full" rate), and the largest buffer size that was below threshold (that is, did not maintain "full" rate). If this text is confusing, just think binary search.
6. Plot (N, Bmin), with N on the x-axis and Bmin on the y-axis.

You will use many instances of iperf, one per flow, to generate equal numbers of long-lived TCP flows from clients A and B to server C. For example, to generate 400 flows total, you must start up 200 iperf clients on host A and then the remaining 200 on host B. The iperf code has been modified in two small ways: (1) the maximum number of connections a server can handle has been increased to 1024, and (2) the code waits for a few seconds after the handshake completes to start sending data.

Directions

1. The binary search test in this assignment requires a modified iperf. Download and compile a modified iperf as follows (from within the assignment-3 folder): `./build-patched-iperf.sh`

2. Your task is to fill the missing pieces in buffersizing.py and buffersizing-sweep.sh. Here's a summary of the functions in `buffersizing.py`, which includes the set of functions you need to fill, marked by `TODO` comments in the code. Once you have filled in the below functions, please follow the steps below to complete your task.

`StarTopo`: The definition of the topology used in the assignment. Make sure to add the links to the switch in sequential order i.e. add link from host A to the switch, then from host B, and finally host C. This will ensure the interface from the switch to host C is -eth3, where is the string name you assigned to the switch. Note that when you set the delay you will split the total equally across both sides of the switch such that the delay for any link is 43.5ms and the total round trip time is 87ms as in the original topology.
`do_sweep`: Does a binary search across sizes to find the minimum buffer size that achieves 98% utilization. It uses the following helper functions:
`set_q`: set buffer size
`set_speed`: set the link bandwidth
`get_rate`: get current aggregate throughput of the flows over the bottleneck link
`ok`: check if achieved throughput is greater than 98% of the available bandwidth
`verify_latency`: Verify the latency settings of the topology you've created
`verify_bandwidth`: Verify the bandwidth settings of the topology you've created
`main`: The main function where the execution begins
`buffersizing-sweep.sh`: you will have to set the value of iface to the name of the interface you change the buffer size on.

3. Run buffersizing.py for N=1. The cwnd and bottleneck buffer occupancy are automatically plotted and saved in the output directory for you:
```
sudo python buffersizing.py --bw-host 1000 --bw-net 62.5 --delay 43.5 --dir test --nflows 200 -n 3 --iperf ~/iperf-patched/src/iperf
```
4. Increase N and repeat the above experiment. What happens to cwnd and buffer occupancy?
5. Using `buffersizing-sweep.sh`, binary search for the minimum buffer size required for 98% utilization. Compare against hardware results by viewing the `result.png` in the log output folder.
6. Submit your topology file `buffersizing.py` and also the `buffer-size-result.txt` file created in your log folder on the assignment submission page.
7. Before you move on, take a stab at answering the quiz questions below and feel free to discuss your responses in the forum.

Note: When you run `buffersizing.py`, you may see an error message about `tcp_probe`. Don't panic! `tcp_probe` is a utility that the program uses to track congestion window of your flows. The program tries to make sure it removes all existing instances of `tcp_probe` before loading a new instance. The error message shows up if there was no previously running instance of `tcp_probe`.

Quiz Questions

- Briefly describe key/qualitative differences between Mininet and the actual hardware setup for the buffer sizing experiment.
- In the binary search algorithm, we adjust the buffer size while the flows are running. How long will it take for the TCP flows to adjust their windows when the buffer size is reduced to half its value?
- In the experimental setup, we use a modified iperf because it is hard to establish new connections after long lived TCP flows have started. Why is it so?
- Look at the plot of cwnd and buffer occupancy for N=1 flow. Based on your understanding of TCP and the buffer sizing paper, explain why you need buffering at the bottleneck switch. For N=1 flow, what is the minimum amount of buffering required to keep the bottleneck link 100% utilized? Why?
- What happens to cwnd and minimum required buffer size as you increase N? Why?
- Describe potential hurdles when using Mininet to reproduce research results. How can this knowledge guide you when choosing a research project for the next programming assignment?

Understanding and Thinking About Variability

Mininet is not a simulator. Hardware is not a simulator. Neither of these use synchronized virtual time, like a simulator. Any process can temporarily block something time-critical, like a byte-counter read or a packet send, which a simulator might run

at perfectly equal intervals. You cannot expect identical results between runs when there is noise in the measurements as well as variability in the behavior of TCP. That said, we have written the starter code to add some robustness to variability. In particular, `get_rates()` waits before reading values, corrects for variability in the precision of the sleep timing, takes multiple samples spaced a configurable period apart, and returns those to you to filter as necessary. We provide `median()` and `avg()` functions for you to do this filtering to yield more stable results in the presence of measurement noise in `do_sweep()`.

Consider: when using binary search, how does random noise affect the distribution of queue thresholds? Would you expect that the determined queue threshold might vary by one or two occasionally, or by more than that? Is the median or mean going to be more tolerant to noise?

Notes

[1] Based on CS 244 assignment.

This page was last edited on 2014/02/03 04:54:53.