CSci551 Spring 2015 Project C

Assigned: April 13, 2015. Project C Due: 6pm, Monday, May 11, 2015.

You are welcome to discuss your project with other students at the conceptual level. You may reuse algorithms from textbooks. You may possibly reuse functions from libraries, but you *must* check any packages/libararies that you plan to use with the TA's. Otherwise, each student is expected write *all* of his or her code independently. All programs will be subject to automated checking for similarity. Any cases of plagiarism will result in an F for the entire course. **If you have any questions about policies about academic integrity, please talk to the professor.**

Please note: you should expect to spend a significant amount of time on the class projects this semester, probably at least 20 hours or more when they're all put together (or that much per project if you're new to Python and shell and network programming). Please plan accordingly. If you leave all the work until the weekend before it is due you are unlikely to have a successful outcome. That said, Project A is a "warm-up" project. Projects B and C are be much larger. Although Project C is smaller than Project B, you should expect it will still take a significant amount of time.

1 Overview

The purpose of this project is to get some hands-on experience designing a congestion aware routing protocol for data centers by using Software Defined Networks. This homework will be done as three separate but related projects. First (Project A), you need to demonstrate that you are able to build a simple topology on top of Mininet and setup a dumb routing protocol. (Project A does not really evaluate anything advanced, but it should confirm that everyones on the same page and is comfortable with our software environment.) Project A has an early due date. Project A should be relatively short for students used to working under Linux and Python; if not, it should help get you up to speed.

In Project B you will implement an *arbiter*. We will supply traffic generators at end hosts, and you will program the communication between hosts and the arbiter application that you will write. The purpose is to rate limit traffic and select different paths for different traffic. It will probably be due just after the midterm. Project B will be much larger than Project A; you should plan your time accordingly. Project B will be assigned later and partially overlaps with Project A.

Project C will be assigned later. It will be smaller than Project B but bigger than Project A. It will build on Project B. Project C will involve extending your Project B arbiter implementation with additional features, e.g., congestion aware routing and monitoring.

Related papers

FatTree, SIGCOMM'08

http://ccr.sigcomm.org/online/files/p63-alfares.pdf

FastPass, SIGCOMM'14

http://dspace.mit.edu/handle/1721.1/88141

2 How grading works?

Since automated tools grade your projects, you are expected to follow the requirements and specifications of the project as they are described. Failure to do so can affect your initial grade, so please make sure that your program works with the given commands.

3 Project A and B

(We do not reproduce Project A and B here, but you may want to refer to them for background.)

4 Project C

In Project C, we will try to improve the throughput and delay performance for the centralized scheduling solution we built in project B. Here are a few suggestions on how to improve the performance. These suggestions are just to give you some directions on how to improve the performance. Since we have not tested these ideas, they may not be correct.

Add retransmission support: You might already experience the hardness to tune UDP to ensure no packet losses. Instead, if we can support fast retransmission, we can significantly improve the performance. (see this paper for why this would improve performance: http://conferences.sigcomm.org/sigcomm/2013/papers/sigcomm/p435.pdf). To support this, one needs: - At the sender: add a sequence number to the packet header, e.g., IPID field - At the receiver: check the list of received packets, and send ACKs about its received packets - At the sender, if a packet is not ACKed in a short timeout (e.g., 1 ms), it will retransmit the packet.

Another way could be the receiver to notify the arbiter about the received packets, and the arbiter can then tell the sender to resend.

We will grade if all the bytes of the flow that we specified are transmitted. Note, the receiver may receive a few more duplicate bytes due to spurious retransmission (i.e., the packet you end up with retransmission actually has arrived). Spurious retransmission can easily happen from 1) *out-of-order arrival* when the arbiter schedules different trunks of one flow into different paths, or 2) *receiver side congestion* so that it sends ACK late after sender side timeout, or 3) *ACK lost*. You may use the retransmission timeout to detect loss rather than arrival order like TCP does. The timeout should neither be too large that FCT is enlarged nor too small that the receiver is overwhelmed with spurious retransmission. Ideally, we should use TCP and allow TCP/IP stack to handle loss. However, there is no way to specify VLAN ID using standard TCP/IP socket.

In project C, we will accept those flows with spurious retransmission but you still want to minimize spurious retransmissions since they waste the link bandwidth. The retransmission is not mandatory requirement but we expect adding this scheme will significantly make more flows complete/correct. The guideline is, we expect to achieve nearly loss-free networking using your centralized algorithm. And we encourage you to integrate retransmission to improve your tail performance once the loss inevitably happens.

As shown in the example code of raw socket programming, it uses recv() function to receive packet from the raw socket at the receiver side. From what your have received at receiver, you can either acknowledge sender or notify the arbiter.

Host-arbiter communication: Instead of sending one request to the arbiter for each packet, we might batch the packet requests to reduce the overhead of sending these requests at the hosts and the time the arbiter use to process the packets.

Incast: You might be aware of the incast problem discussed in the DCTCP paper (we will read in the class). If all the packets are scheduled to arrive at the same switch output port at the same time, it may cause significant performance collapse.

Avoid head-of-line blocking: You may postpone or pace the traffic scheduling for one flow if a particular link is contented. However, this "queued" flow request should not affect the other flows if the paths for the other flows are less congested. So instead of creating head-of-line blocking (HoL blocking. http://en.wikipedia.org/wiki/Headofline_blocking) in the network, you may allocate traffic at per-flow or per-link basis so that the whole network is evenly utilized rather than polarized by a few hot-spots.

5 Grading

VM setting: We will test your program on our server on a VM with 4 cores and 4 GBs of memory. As before, the VM is sandboxed and if you need access to any libraries that need installation you should discuss it with the TAs. In order to login, you may access the server at `fastpass.omid.io`, please use your github ID and the last 4 digits of your USC-ID to view the dashboard.

Modifications to the arbiter network In this part of the project, we have made a small modification to the arbiter_network script. The links in the FatTree network are now capped at **1Gbps**, and the queues on each port of the switch are limited to **200** packets – the bandwidth and the queue sizes for the connections to arbiter are still unlimited. Note that with this change, you can experience a lot more packet drops from project B, so you would need to be smarter about bandwidth and timeslot allocation.

Traces: We will test your program with three types of traces:

1. *Incast*: This is many-to-one traffic pattern similar with project B but with short flows.

2. *Permutation*: In the permutation traffic, each host sends bulk data to one other host and no host receives from more than one host. For example, we have hosts h1, h2, h3, and h4 and the permutation traffic, for example, could be $h1 \rightarrow h2$, $h2 \rightarrow h4$, $h3 \rightarrow h1$,$h4 \rightarrow h3$. Ideally, there will not link contention given the full bi-section bandwidth in Fattree topology. We will use long flows to evaluate how well your algorithm can spread traffic across the network.

3. *Mixed*: We have mixed long flows and short flows. The source and destination of each flow are randomly chosen. We will measure the throughput of long flows (throughput sensitive, see below) and the latency of short flows (latency sensitive, see below). The point is to make sure your algorithm neither creates HoL blocking by throttling short flow nor reserve too much bandwidth for short flow.

In our trace specification, we assign destination port **6XXX** (i.e., [6000, 7000) like 6001, 6200) for throughput sensitive flows, **7XXX** for latency sensitive flows, and **5XXX** for both throughput and latency sensitive flows. In evaluating *Mixed* trace, we would measure throughput only for throughput sensitive flows and latency only for latency sensitive flows.

We will give one example trace for each type for you (put in "traffic/sample/" directory). However, the evaluation on the server will be based on **ONLY** hidden traces of the same type but not the example trace. The hidden traces may have larger flow size or more flows, but the traffic pattern exactly follows trace types above. Therefore, you are encouraged to generate your own traces of these types (with various flow size and number of flows) to debug and test the performance further.

Metrics We give grades to project C based on three criteria: *(1) Throughput:* We note the time when we start the "evaluation.py" as the start time and the last packet captured on wired as the end time. We then calculate the throughput for each flow based on the elapsed time. For Project C, we will use the average throughput of all flows for each trace, except for Mixed trace we will use the average throughput only for long flows.

(2) Tail flow Completion Time (FCT): We use the elapsed time between start time and flow end time as the flow completion time. We may compare the FCT across flows to calculate the network fairness; if one flow has experienced significantly delay. For project C, we will use the largest FCT of all flows for each trace, except for Mixed trace we will use the largest FCT only for short flows.

(3) Overall: For each trace, we calculate throughput (in Mbps) + (flow size (in MB) * 8 / tail FCT (in seconds); The flow size is the average flow size in each of our traces(except for Mixed trace where the flow size is the average flow size of short flows). We will sum up the scores from all the traces with the same weights.

For each criteria, we will calculate the percentile of your rank in the class (e.g., your number is better than X% of the class). We will then pick the best percentile among the three criteria as your final score.

Similar to part A/B of your project, once you are satisfied with your solution, please push your code to Github for grading. We will take a snapshot of your code at the deadline time, so please make sure the latest version at that time is the one you'd like to

5.1 Deliverables

You are expected to commit the following documents to projectB's branch in Github as you did for Project A. For the submission procedure, please refer to Project A's assignment. To push your code, you can use the following command:

```
git push origin projectC:projectC
```

1.Code You are expected to submit all the source files and we will generate executable files "cperf" and "arbiter" using make. We will use tcpdump to log all the packets you send traversing the network to verify the correctness, i.e., if all the traffic are properly generated and send to the destination without any drops. Make sure the traffic you send uses the correct ports.(e.g., clients communicating with the Arbiter should use port 5000, and flows should use the ports we specified in the traffic files.) We provide a script called "evaluate.py" to invoke "cperf" and "arbiter" programs in hosts and arbiter, respectively. We provide 16 sample traffic specification files with each host for you to test your code. We will invoke your program by using the following commands:

```
./arbiter_network.py --custom=fat.py --topo=fattopo
(in a separate terminal after arbiter_network.py is running)
./evaluation.py
```

2.Makefile You are expected to submit a Makefile so we can generate executable files "cperf" and "arbiter" from your source files. We will issue "make" directly to do that. If you are using python, you can make an empty Makefile. If you are using third party tools, make sure all the dependencies and libraries will be automatically installed via that Makefile. Any failure to generate the executable files will be your responsibility.

3. README.txt Please include a "README.txt" file answering the following questions:

1. Did you use code from anywhere for your project? If not, say so. If so, say what functions and where they are from. (Also mark these codes with a comment in the source code.)
2. List all the detailed changes you have made from Project B to Project C. And why each change helps the performance