CS 538: Advanced Computer Networks Fall 2012

Assignment 1: Experimental Data and Tools

Assignment 1 Due: 3:30 PM CT, September 18, 2012

1 Introduction

The main goal of this assignment is to get hands-on experience with tools for experimental networking research. In addition, the assignment will hopefully remind you about some core networking material. Specifically we will get our hands dirty in three parts:

- Analysis of public Internet BGP routing traces (§2)
- TCP throughput experiments with Mininet, a network emulator (§3)
- OpenFlow functionality with Mininet (§4)

You may **choose one** of the above three parts to do for this assignment, according to your interests. We encourage you, however, to check them all out. We're providing them because they should be useful for research projects in this class, and checking out what experiments you *can* do may also give you ideas about what research you might *want* to do.

Submission instructions. This assignment is due at the start of class on the date listed above. Submit by email to the TA, Giang Nguyen (nguyen59@illinois.edu). Acceptable formats are PDF (preferred), or plain text email, with attached figures and code files. (Word .doc is not preferred. Please export to PDF.) The subject of the email should be:

```
CS538 hw1 submission: firstname lastname NetID
```

Collaboration policy. You're encouraged to discuss the assignment, solution strategies, and coding strategies with your classmates. However, your solution and submission must be coded and written yourself. Please see the policy on cheating stated in the course syllabus.

2 The Global Internet

On January 25, 2011, a popular uprising began in Egypt that would ultimately bring an end to the 29-year regime of Hosni Mubarak. On January 27, 2011, attempting to inhibit the Facebook- and Twitter-organized protests, the Egyptian government shut off essentially all Internet service to the country of 82 million people — a unique event in the history of the Internet.

Your mission is to answer the question, *How long does it take to sever all global networked communications of the 15th largest country in the world?* Because of the open and decentralized nature of the Internet, you can answer this question using publicly-available Internet routing information.

This document guides you through the process, including a sample parser for the BGP data. However, the main point of this section is to get a feel for the BGP data by visualizing an interesting event in the data set. If you would like to write your own parser rather than using ours, or explore an interesting aspect of the data other than what we suggest, or even explore a completely different event (there are many besides the Egyptian disconnection), you are welcome to do so. Check with us first if you have any doubts about appropriateness.

2.1 Getting started

The Route Views Project maintains data of routing behavior on the live Internet, and stores these traces for later analysis. Multiple years of data sets are available. To produce the data, Route Views maintains a number of *collectors*. Each collector has BGP connections to several ISPs' routers. The collectors log two types of data:

- occasional snapshots of the collector's entire routing table (Routing Information Base, or RIB); and
- continuous logs of the BGP update messages received from the neighboring routers.

You can download Route Views data from: `http://archive.routeviews.org/`

Download the package of code for this part: `http://courses.engr.illinois.edu/cs538/assignments/a1-bgp.tar.gz` Inside, you'll find a tool called `libbgpdump` which processes both kinds of Route Views data; they both use the "MRT" format. Compile this by running `./configure` and then `make` in the `libbgpdump-1.4.99.11` directory. This builds a program called `bgpdump`, which, when given some input file (`./bgpdump routeviews_data_file`) will produce a human-readable text version of the data. Note the input file can be either a bzipped (`.bz2`) MRT file like you will get directly from the Route Views repository, or an un-bzipped MRT file. Take `bgpdump` and the Route Views data for a spin as follows:

1. From Route Views, download the first RIB snapshot from January 27, 2011, from the London Internet Exchange (LINX) collector.
2. Run `bgpdump` on the RIB snapshot. What does each field mean?
3. Search the `bgpdump` output to find any[1] RIB entry associated with your computer's public IP address. (Hint: you could do this by finding associated IP prefixes or AS numbers. You can use a whois database (e.g., `http://whois.arin.net/ui/`) to find the IP prefix and origin AS number associated with an IP address.)
4. Can you use the above entry to determine the sequence of ISPs through which packets will flow when traveling from one of the LINX routers in London to your personal computer?

What to submit: The sequence of ISPs (AS numbers and/or business names) from the last step.

2.2 Measuring Egyptian route withdrawals

Next, we want to use the Route Views data to figure out how long it took Egypt to leave the Internet. To do this, we will use an imperfect but simple approach: We will count how many Egyptian-related prefix withdrawals we have seen over time. Due to the dynamic nature of the Internet, there are continually announcements and withdrawals even under normal conditions. But we'll see *one period of time with a very high rate of Egyptian withdrawals*, and that period will correspond to Egypt's disconnection from the Internet.

Although you're welcome to write your own parser if you prefer (or use the `libbgpdump` library), we have written an incomplete parser for you. In the package of code included with this problem set, you'll find a program called `simple_bgp_parse.c`. This program expects to receive, on its standard input, the output of `bgpdump`. It scans through the BGP RIB entries or updates, and does two things. First, when it sees "interesting" entries, it remembers that the associated IP prefixes are interesting. Second, it keeps track of how many withdrawals of "interesting" prefixes it has seen, and prints out a running total.

What is "interesting" is a matter of opinion. The default version of the program is quite dull and thinks nothing is interesting. As described below, you will need to decide how to pick out the "interesting" entries, in order to learn which prefixes are associated with Egypt.

[1] There may multiple RIB entries for a single IP prefix, because each Route Views collector reports prefix advertisements received from multiple routers.

1. From Route Views, download the London Internet Exchange (LINX) collector's Updates data, from near the time of Egypt's departure from the net. That happened sometime between 21:00 and 23:00 UTC on January 27, 2011, so you'll want to grab all the LINX Updates data at least in that interval. Process these files with `bgpdump`, and send the output of all that to the (unmodified) `simple_bgp_parse` program. It should output the total number of updates seen. (For your own interest: In the data you downloaded, what time did the first and last update messages occur, and what was the average rate of updates per second during this period?)

   Tip: You might find the command `bzcat` useful. For new Unix users: to send multiple files to a program's standard input, you can run a command like: `cat file1 file2 file3 | my_program`.

2. Modify `simple_bgp_parse` so that it thinks BGP RIB entries for advertisements that *originated* at Egyptian ASes are "interesting". (Hint 1: this only takes a few lines of code. Hint 2: the following Autonomous System (AS) numbers belong to Egyptian ISPs: 5536, 8452, 24835, 24863, and 36992. Hint 3: Think about what part of the BGP RIP entry information you can to figure out which advertisements were originated by Egyptian ASes.)

3. Now, run your modified `simple_bgp_parse`. This time, on standard input, feed it the output of `bgpdump` from the RIB file that you downloaded, concatenated with the output of `bgpdump` on the Update files. (The RIB output lets us learn which prefixes are interesting, and then we can count the occurences of interesting withdrawals in the updates.) Note that the output of `bgpdump` from the RIB file is large (about 2 GB), so if you are running on university machines, you might want to put this in `/tmp` rather than in your home directory. And clean up when you're done.

   The output should now be a list of pairs of numbers; read the comment near the end of `simple_bgp_parse.c` for a description. Using that data, draw a plot showing *time* on the x axis, and *total number of Egyptian prefix withdrawals seen so far* on the y axis.

   You can use any plotting tool you want, but we have included an example of how to use Gnuplot in the package of code included with this problem set. Gnuplot is very useful and easy to get started with. In the `gnuplot_example` directory, run the command `gnuplot example.gpl` which will produce `example.pdf`. You should be able to inspect `example.dat` and `example.gpl` and modify them to suit your purposes.

What to submit: Your plot from the last step. If everything worked, it should show a slowly increasing number of withdrawals seen, and then it should increase quickly for a period — a "withdrawal storm", corresponding to the disconnection of Egypt from the Internet – and then the rate of withdrawals should slow down again. Based on your plot, how long did that high-rate "withdrawal storm" last?

3 Mininet TCP experiment

Experimentation is an important part of networking research. However, large-scale experiments can sometimes be hard to achieve, e.g., due to lack of machines. In this section, you will learn how to use Mininet[2], a relatively new experimental platform that can scale to hundreds or more emulated "nodes" running on a single machine. Mininet takes advantage of Linux support for *network namespaces*[3] to virtualize the network on a single machine, so that different processes on the same machine can see their own network environments (like network interfaces, ARP tables, routing tables, etc.), distinct from other processes. Combined with the Mininet software, this enables a single machine to emulate a network of switches and hosts. The emulated processes, however, do see the same real/physical file system. Mininet is designed with OpenFlow[4] in mind, and in the second part, you will also learn to use it.

[2] `http://yuba.stanford.edu/foswiki/bin/view/OpenFlow/Mininet`
[3] `http://lwn.net/Articles/219794/`
[4] `http://www.openflow.org/`

3.1 Prepare the VM

1. Install Virtualbox from `https://www.virtualbox.org/wiki/Downloads`. VMware should also work; adjust your VM configurations accordingly. (It is possible to install Mininet directly on your Linux system, but for simplicity we'll use the virtual machine here.)
2. Download and unzip the VM with Mininet already installed from `https://github.com/downloads/mininet/mininet/mininet-vm-ubuntu11.10-052312.vmware.zip`
3. In VirtualBox, create (not "Add") a new VirtualBox VM
   (a) For "Operating System" and "Version," select "Linux" and "Ubuntu," respectively.
   (b) At the "Virtual Hard Disk" page, select "Use existing hard disk," and select the `Mininet-VM.vmdk` file just unzipped.
   (c) For the newly created machine, go to "Settings" → "Network" and make "Adapter 1" a "Bridged Adapter". This assumes your VM is allowed to obtain an IP address from your local network. Alternately, you can use "NAT Adapter." For more information on networking with VirtualBox, see `http://www.virtualbox.org/manual/ch06.html`
4. Start the VM
5. Log in with **openflow** for both username and password
6. Make sure `eth0` is up:
   (a) run the command:
   `ifconfig eth0`
   (b) check the `inet addr` field. If it does not have an IP address, then run the command:
   `sudo dhclient eth0`
   and repeat step a.
7. Install the Mininet modifications and skeleton code we created for this assignment (this requires an external connection, i.e., a working bridged or NAT adapter)
   (a) Obtain the tarball
   `wget http://courses.engr.illinois.edu/cs538/assignments/hw1.tgz`
   (b) Install with command
   `sudo tar xfz hw1.tgz -C /`
   (c) Note the following files:
      i. `hw1a.py`
      ii. `hw1b.py`
      iii. `pox/pox/samples/mycontroller.py`
8. Install `polipo` web proxy, and `lighttpd` webserver (this requires an external connection, i.e., a working bridged or NAT adapter)
   `sudo apt-get install polipo lighttpd -y`
9. Create a 40MB random file:
   `sudo dd if=/dev/urandom of=/var/www/bigfile bs=4k count=10k`
10. Install a GUI in the VM:
    (a) Install the GUI
    `sudo apt-get install openbox xinit -y`
    (b) Start it
    `startx`

(c) Right-click on the desktop and select "Terminal emulator"

Alternately you may use SSH to log in to the VM remotely, with GUI (X11) forwarding. With SSH, you will need to enable X-forwarding (e.g., `ssh -X` on *NIX hosts) when you ssh into the VM. NOTE: this requires you have an X server running on the host. See a description of how to do this on various platforms at `http://www.openflow.org/wk/index.php/OpenFlow_Tutorial#Download_Files`. Alternative for Mac OS X: install the Developer Tools (a free download from the App Store) and open `/Applications/Utilities/X11`.

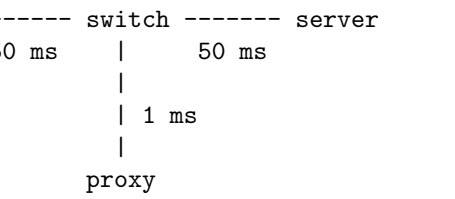
3.2 TCP throughput vs. latency

Suppose you read a paper wherein the evaluation shows that if you split a TCP connection over a high RTT path into two separate connections — by using a proxy placed approximately between the two hosts, each with half the original RTT — then you will achieve higher throughput than you would without the proxy. Let's replicate this experiment using Mininet.

First, you can familiarize yourself with Mininet by following `http://yuba.stanford.edu/foswiki/bin/view/OpenFlow/MininetWalkthrough`, with particular emphasis on the "Custom Topologies" section. The Wireshark and XTerm parts of the walkthrough are why we installed the GUI above.

The setup of our experiment will be a web client and a web server seperated by a 200ms RTT path. Additionally there is a web proxy on the path from the client to the server. The proxy is approximately 100ms RTT from the client as well as the server. The client will fetch a large file from the server in two ways: (1) directly, and (2) by using the proxy.

You will be working off the `hw1a.py` skeleton. Create the following network topology, using the same Python API as in the "Custom Topologies" section in the walkthrough. Optionally use the "name" parameter when calling `add_node()` — see `hw1a.py` for example usage. (This optional parameter is a change we made to the stock Mininet code.) Otherwise the host will use Mininet's default naming scheme. The host's name is most useful if/when you use Mininet's interactive interface.



Set link delays (not RTTs) as specified in the topology above using the "delay" parameter when calling `add_edge()`. See `hw1a.py` for example usage. (This parameter is another change we made to the stock Mininet code, but an upcoming release of Mininet should allow controlling link latency.)

NOTE: You might see errors `link dst status change failed: RTNETLINK answers: File exists`. These are harmless and can be ignored.

Once you have created the topology and started the network, you will start the web proxy and web server on the respective hosts, by using the host's `cmd()` API — see `hw1a.py` for example — to execute these commands:

- On the server host, start the `lighttpd` web server like so:
  `killall -9 lighttpd ; lighttpd -f /etc/lighttpd/lighttpd.conf`
- On the proxy host, start the `polipo` proxy like so:
  `killall -9 polipo ; polipo proxyAddress=<proxyIP> proxyPort=<proxyport> &`
  where `<proxyIP>` is the proxy's IP address (as assigned by Mininet), and `<proxyport>` is some port number of your choice, e.g., 8080.
- On the client, to fetch the file directly from the web server:
  `time wget -q -O /dev/null http://<serverIP>/bigfile`

- On the client, to fetch the file via the proxy:
  `time http_proxy=<proxyIP>:<proxyport> wget -q -O /dev/null http://<serverIP>/bigfile`
  where `<serverIP>` is the server's IP address (as assigned by Mininet).

For each scenario (direct v.s. via-proxy), fetch the file 20 times.

To run your experiment: `sudo python hw1a.py`

What to submit:

1. Your completed `hw1a.py`
2. A CDF plot comparing the file-fetch-time performance of the two scenarios
3. Your interpretation/explanation of the result

4 OpenFlow in Mininet

In this exercise, you will gain a basic understanding of OpenFlow and create a custom OpenFlow controller to control your switches. This section assumes you have already set up Mininet (§3.1).

Quite simply, OpenFlow allows for "programmable" network devices, e.g., switches. With Mininet, each switch will connect to the controller specified when the switch is launched. When the switch receives an Ethernet frame, it consults its forwarding table for what to do with the frame. If it cannot determine what to do with the frame, the switch sends the frame (and some extra information such as the input switch port) to the controller, which will then instruct the switch on what to do with the frame. To avoid this extra work on every such frame, the controller can install a new rule/match in the switch's forwarding table, so that the switch can forward future similar frames without having to contact the controller.

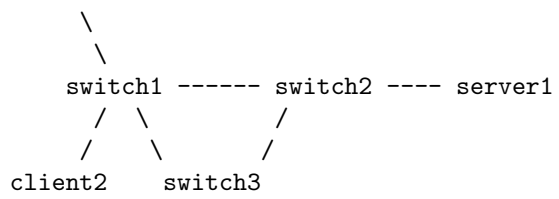
The controller framework you will use for this exercise is POX[5] (in Python), with the current "best" documentation at `https://openflow.stanford.edu/display/ONL/POX+Wiki`

The two files you will work on are:

- `hw1b.py`
- `pox/pox/samples/mycontroller.py`

NOTE: You will start your controller (instructions below) before starting the Mininet experiment. We have made the controller listen on the localhost port `12345` (hardcoded). When your Mininet network starts, its switches will connect to the controller on this port. This is already done for you in the provided files.

In this exercise, you will create this topology:



where these three links have specified delays:

- switch1 – switch2: 20ms
- switch1 – switch3: 20ms
- switch3 – switch2: 20ms

[5] `http://www.noxrepo.org/pox/about-pox/`

The switches will direct client1's flows along the shorter switch1 – switch2 path (in *both* directions), and client2's flows along the longer switch1 – switch3 – switch2 path (again, in *both* directions).

For most intents and purposes, switch3 does not exist. It exists only for switch1 to forward frames with srcMAC=client2 dstMAC=server1 to it, and for switch2 to forward frames with srcMAC=server1 dstMAC=client2 to it. For instance, in cases where switch1 or switch2 needs to flood a frame, it should not flood that frame to switch3.

See the files for more information.

How to Run Your Experiment:
You will want to open two terminals: one for the controller, the other for Mininet.

1. Clear Mininet:
   `sudo mn -c`
   Do this before starting the controller. Otherwise, switches still hanging around from a previous run might be trying to connect to the controller, which might, or might not, confuse your controller logic.

2. Start your controller:
   `python pox/pox.py --no-cli samples.mycontroller`

3. Run your Mininet in the other terminal window:
   `sudo python hw1b.py`

The `hw1b.py` skeleton already contains "verification" code at the end, where it uses `ping` to check the RTTs between client1 and server (should be approx. 40ms), and client2 and server (should be approx. 80ms). If your experiment shows these RTTs, then you have correctly implemented the exercise.

What To Submit: Your `hw1b.py` and `mycontroller.py` files.