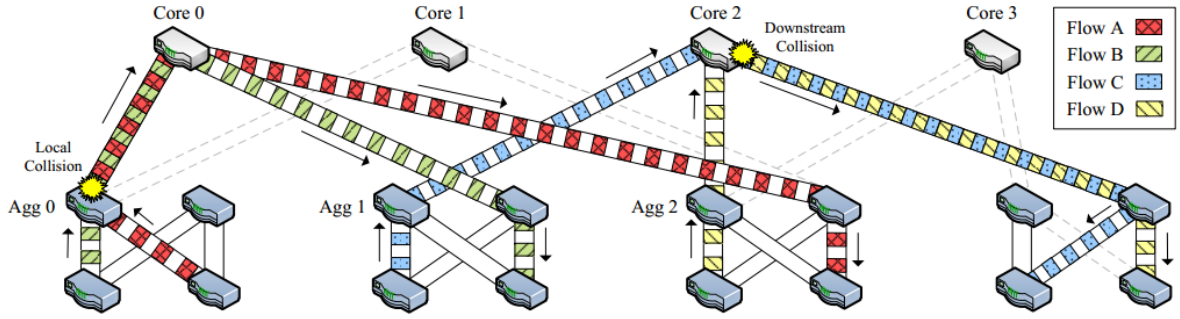
Workshop In Communication Networks - Final Project

Introduction

1. **Under what circumstances ECMP fails?**
   We have seen in class and in the paper that ECMP is good for little flows. As the flows grow, ECMP might route two “elephants” through the same link, creating a collision and slowing down the traffic. Here is an image example from the paper:



   In this example, we can see how Flow A and Flow B are colliding in one link, where they could be separated in the network (Flow A could go through the Core 1 switch instead of Core 0).
2. **How can you detect mice and elephants?**
   Using Mininet’s Python API, we can request statistic on flows that we install. We install flows rules for each 5-tuple (source IP address, destination IP address, source transport port, destination transport port, transport protocol). Those flow rules are saved in a map, and each 5 seconds we request statistics for each flow in the map. If a flow reaches 700 Bytes (**can be changed in …**), we call it an elephant. In this case, the flow requires special attention.
3. **When should you do something?**
   Generally, we have to reroute the flow to another path if we detect the flow demand estimation does not suite the path that is taken at the moment (in terms of with how much other flows does it share the same link).
4. **What should you do?**
   Again, if we detect a collision between two big flows we have to reroute one of them to a more free route.
5. **How should you balance the load across links?**
   First, we estimate each flow’s demand. After that, according to the demand we

estimated, we use Global First Fit to “place” the flow in a suiting route.

6. **How can you measure your success in mininet?**
   When creating the network, we have full control over each host and switch. That means that we can generate sample traffic and monitor it. Each Mininet host and switch run an emulated Ubuntu based OS, so we can use the built in tools to measure the traffic. In this project, we mainly used ping to check basic connectivity and iperf to generate large flows and monitor their speed (mainly on TCP, but UDP works the same way…).

Methods

Little problems that stood in our way

As we started the project, we encountered a problem which had us thinking **a lot of time** (partly the reason that we couldn’t manage to get to the presentation). In order to create a flow in TCP/UDP level, we have to create flow rules that bring into account the protocol, IP, ports etc. The problem was that if we let the discovery messages (the ones that are sent during the network startup automatically) install new rules, that would ruin the higher level rules, as they do not care about the protocol and ports. The solution (couldn’t believe that it took us that long) was to forward all ethernet packets without installing new rules. Any other packet that is in level 4 (TCP/UDP) is a subject for rule installation.

We had another problem, which was understanding POX Python API. When installing a 5-tuple rule, we forgot to add `fm.match.dl_type` which is required if we want to set a protocol version. This problem led us to the “solution” of not installing rules at all, making the controller decide for each packet like it was a single router. The performance was terrible. Links were super slow, the controller threw a bunch of exceptions and worst of all, it was a pain for us. Luckily, we’ve managed to find the cause of the problem and move on.

When implementing part 2, we’ve encountered a problem in rerouting flows: when the process of rerouting took place while the packets were going through, it ruined the session and the remaining packets didn’t get to their destination. The trick we did is giving priority for changing flow rules, i.e flow

```
ESTIMATE-DEMANDS()
1   for all i, j
2       M_{i,j} ← 0
3   do
4       foreach h ∈ H do EST-SRC(h)
5       foreach h ∈ H do EST-DST(h)
6   while some M_{i,j}.demand changed
7   return M

EST-SRC(src: host)
1   d_F ← 0
2   n_U ← 0
3   foreach f ∈ ⟨src → dst⟩ do
4       if f.converged then
5           d_F ← d_F + f.demand
6       else
7           n_U ← n_U + 1
8   e_S ← (1.0 − d_F) / n_U
9   foreach f ∈ ⟨src → dst⟩ and not f.converged do
10      M_{f.src,f.dst}.demand ← e_S

EST-DST(dst: host)
1   d_T, d_S, n_R ← 0
2   foreach f ∈ ⟨src → dst⟩
3       f.rl ← true
4       d_T ← d_T + f.demand
5       n_R ← n_R + 1
6   if d_T ≤ 1.0 then
7       return
8   e_S ← 1.0 / n_R
9   do
10      n_R ← 0
11      foreach f ∈ ⟨src → dst⟩ and f.rl do
12          if f.demand < e_S then
13              d_S ← d_S + f.demand
14              f.rl ← false
15          else
16              n_R ← n_R + 1
17      e_S ← (1.0 − d_S) / n_R
18  while some f.rl was set to false
19  foreach f ∈ ⟨src → dst⟩ and f.rl do
20      M_{f.src,f.dst}.demand ← e_S
21      M_{f.src,f.dst}.converged ← true
```

rule for elephant receives higher priority than rule for mice, such that when we add new rule, it starts to run at appropriate moment such that no packets are lost. At the last moment we found the problem: direction. The links are bidirectional, however at the start of project (and so in exercises during the semester) we ignored it, that cause to following problem: if host 1 is transmitting at full rate - it can't receive packets sent from other hosts. To solve this issue, we doubled the capacity of each link (as it has normal capacity for each direction) and link is stays to be unidirectional.

Demand Estimation

For flow demand estimation we implemented the demand estimation algorithm described in the paper. And after demand estimation we also update the usage for each link - such that the path will provide the maximum possible capacity without hurting other flows.

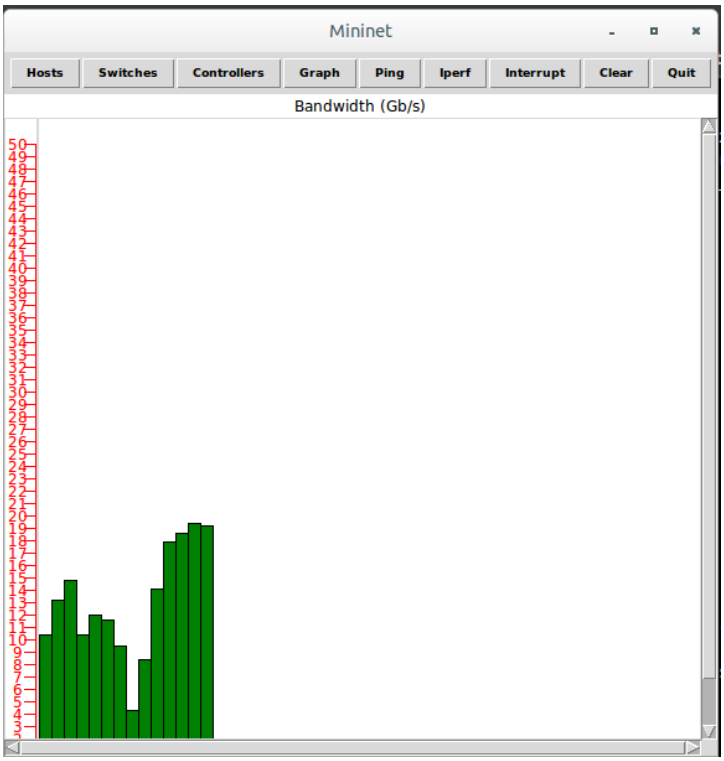
Load Balancing

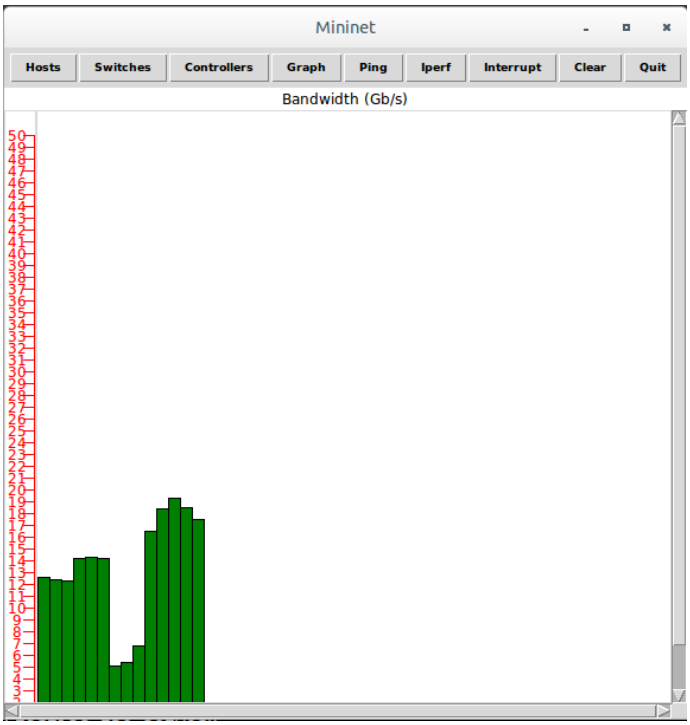
For load balancing we chose Global First Fit over Simulated Annealing because at the time of the implementation, our normal ECMP performance was already bad, so the lighter algorithm was a better choice for us. We also think that because Mininet is running on a virtual machine and not natively on the computer, it wouldn't be so smart running heavy tusks on it.

```
GLOBAL-FIRST-FIT(f: flow)
1   if f.assigned then
2       return old path assignment for f
3   foreach p ∈ P_src→dst do
4       if p.used + f.rate < p.capacity then
5           p.used ← p.used + f.rate
6           return p
7   else
8       h = HASH(f)
9       return p = P_src→dst(h)
```

Results

For measuring the results we mainly used iperf, which is a built in tool in Ubuntu (and in other Linux distributions in general) to measure network performance. What it does, basically, is sending sample data in TCP or UDP, for specified time, number of packets or bytes between a client and a server. Mininet has a built in function that opens the client in one host, and the server in the other. It also checks the traffic the other way. After the loops are done, server and client speeds are presented. For easy monitoring, we've found a script that shows a GUI with all the running hosts, and several commands that run on all of them simultaneously. The GUI also has a graph window, where it shows the





overall bandwidth of the network. Clicking on ping or iperf would establish connection between each host and the next one (h1 -> h2, h2 -> h3 and so on).

ECMP TCP Results

Max total bandwidth: 20 Gb/s
Average bandwidth: 12.93 Gb/s

Enhanced ECMP TCP Results

Max total bandwidth: 20.3 Gb/s
Average bandwidth: 13.47 Gb/s

Discussion

We can see that our approach has some positive effect on the network bandwidth. It is clear that the solution that was described in the paper definitely generates better results in most cases. We could test the network a bit more, but our computer are quite old and could handle to much traffic in the virtual machine. We should have also tested more than 2 flows per host and add a random flow generator so we could see differences between small flows and big ones. Another thing that we could do (but didn't) is generating TCP and UDP data randomly (again, because of performance issues).

When does it help

We have seen already that ECMP does a good job as long as the flows are small. In today's standards, looking at very busy data-centers (like Google's and Facebook's) small flows can combine into large ones (having the same 5-tuple) and maintaining good balance on the network is crucial for speed and user usability. We worked with this kind of company and product where the result should be accurate and good, so that's why we tested mostly with TCP.

User manual

In order to run the project, follow the next steps:

1. Run the Mininet virtual machine (Ubuntu 14.04 based is recommended). We assume that it's IP will be `192.168.56.101`.
2. When the virtual machine is loaded, open Terminal with 3 windows/tabs:
    a. First tab would copy the files into the virtual machine, run:

```
$ cd <project_directory>
$ scp ./Discovery.py ./of_learning_switch_spanning_tree.py
./SpanningTree.py ./utils.py priority_dict.py
ecmp_enhancer.py mininet@192.168.56.101:~/pox/pox
$ scp ./consoles.py ./ft-topo1.py
```

`mininet@192.168.56.101:~/mininet/custom`

b. The second tab would run the POX controller, run:

```
$ cd ./pox
$ ./pox.py log.level --DEBUG
of_learning_switch_spanning_tree
```

c. The third tab would run the Mininet script, from here we have to options:

   i. In order to have full control over the network with Mininet's CLI, run:

   ```
   $ sudo mn --custom ~/mininet/custom/ft-topo1.py
   --topo FatTree,4,6,2 --mac --controller remote
   ```

   ii. In order to run our GUI script, run:

   ```
   $ cd ~/mininet/custom
   $ sudo python ./consoles.py
   ```

   Using the GUI is straightforward: it shows all host's terminals and some action buttons including the graph display, ping, iperf and interrupt. Changing the topology can be done by replacing or modifing the `FatTreeTopology` class in the `consoles.py` file.