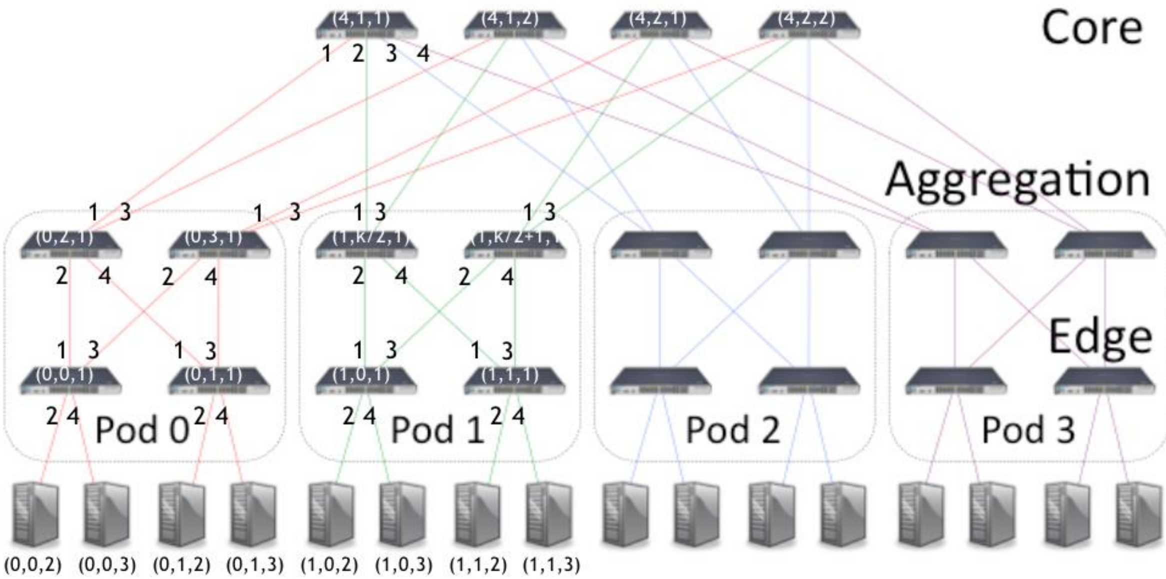
Report of FatTree project

Huimin Yan, Xinyuan Ma

In this project, we focus on the implementation of FatTree topology, and three routing algorithms: Dijkstra algorithm, ECMP algorithm, and Two-Level algorithm. At last we use the given traffic pattern to evaluate these three routing's bisection bandwidth.

Here is the topology:



1. Create topology

In the code, we implemented it as the following:

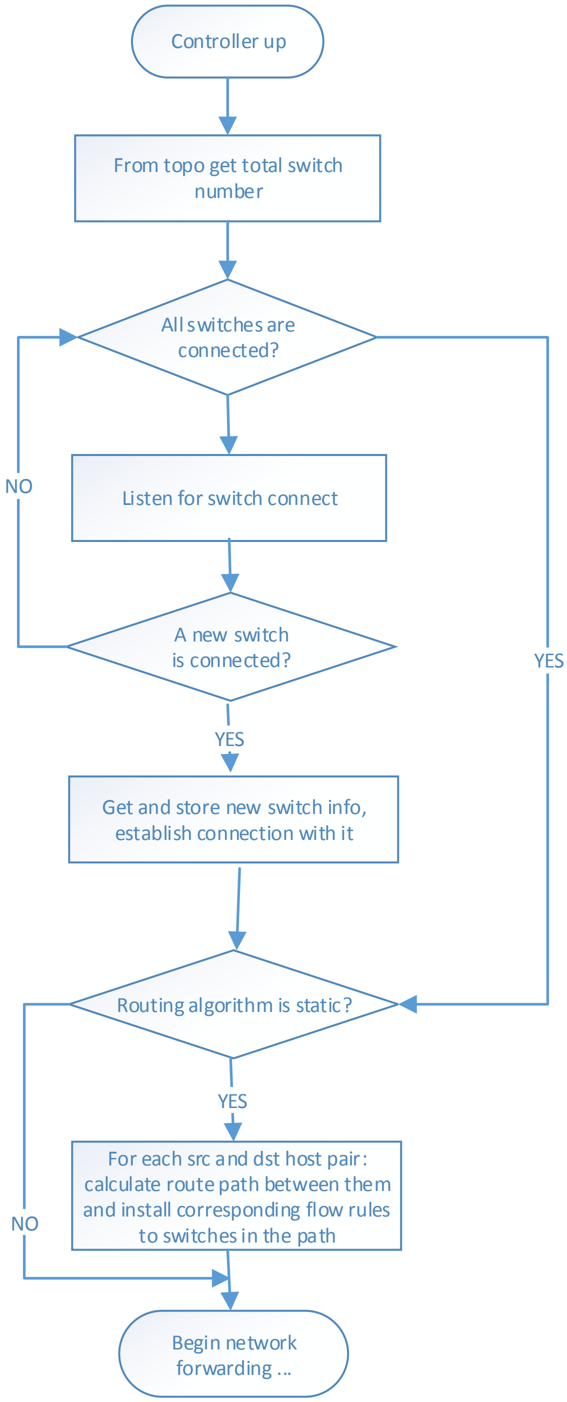
1) define four level: core level as the lowest level, and then aggregation level, and edge level and host level, which is the highest level.
2) create each level's nodes; add links between them, define the port connection as picture shows.
3) After create the topology, we have interface to get two switches' connected port number. This will be used when insert flow rules.
4) Limit the bandwidth of each link to 10Mbps.

2. DCController

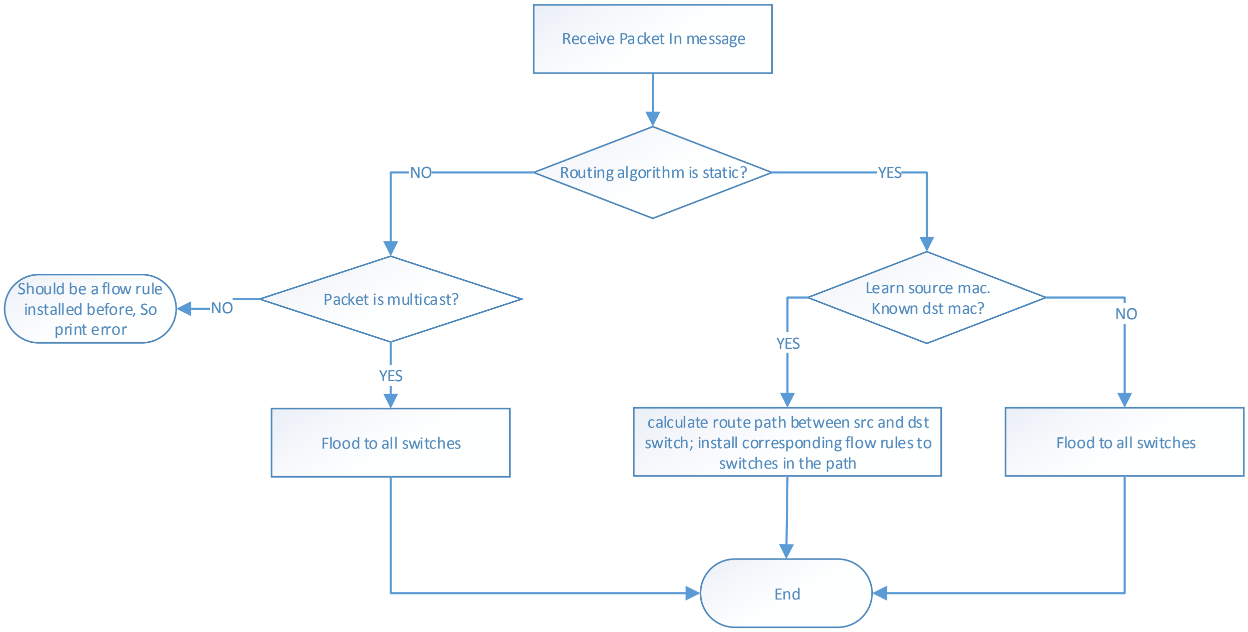
We define Dijkstra and two-level algorithm as static routing algorithm, and inserts all needed flow rules when all switches are up.

Define ECMP algorithm as non-static algorithm, when controller receive a packet In, ECMP algorithm will be invoked.

DCController's up action:

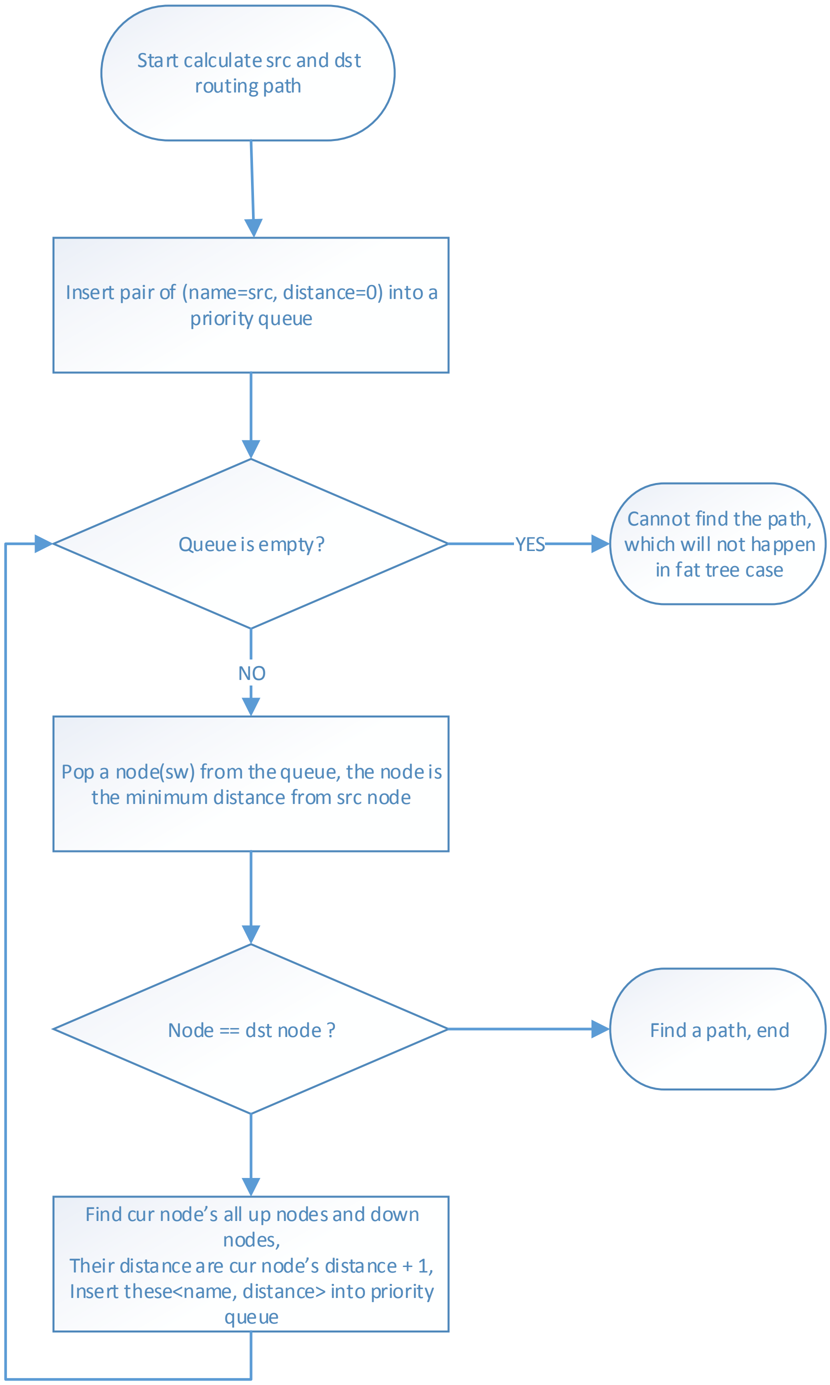


Handle when controller receive a packet in message:



3. Dijkstra algorithm

Since we assume all links have the same distance 1, so for a particular switch, we can get it's directly connected up layer switches and down switches as its neighbors, and calculate their distance as current distance + 1, the detailed algorithm is:



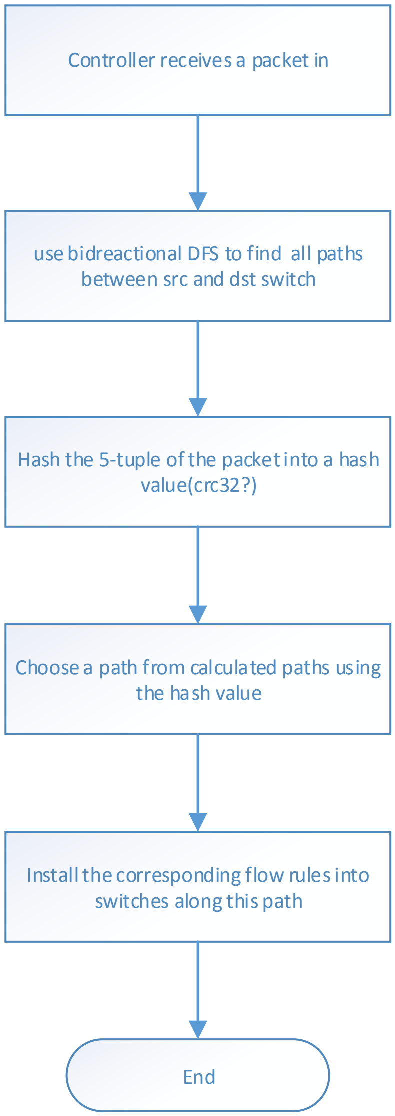
Using Dijkstra algorithm, we push all the needed flow rules into switches after all switches are connected, so it's a static routing algorithm.
For example, if host (0,0,2) want to communicate with host (1,0,2), all the switches in the path have following flow rules:

```
forwarding path:
mininet> sh ovs-ofctl dump-flows 0_0_1 | grep "dl_src=00:00:00:00:00:02,dl_dst=00:00:00:01:00:02"
 cookie=0x0, duration=70.201s, table=0, n_packets=0, n_bytes=0, idle_age=70, dl_src=00:00:00:00:00:02,dl_dst=00:00:00:01:00:02 actions=output:3
mininet> sh ovs-ofctl dump-flows 0_3_1 | grep "dl_src=00:00:00:00:00:02,dl_dst=00:00:00:01:00:02"
 cookie=0x0, duration=78.565s, table=0, n_packets=0, n_bytes=0, idle_age=78, dl_src=00:00:00:00:00:02,dl_dst=00:00:00:01:00:02 actions=output:1
mininet> sh ovs-ofctl dump-flows 4_2_1 | grep "dl_src=00:00:00:00:00:02,dl_dst=00:00:00:01:00:02"
 cookie=0x0, duration=93.054s, table=0, n_packets=0, n_bytes=0, idle_age=93, dl_src=00:00:00:00:00:02,dl_dst=00:00:00:01:00:02 actions=output:2
mininet> sh ovs-ofctl dump-flows 1_3_1 | grep "dl_src=00:00:00:00:00:02,dl_dst=00:00:00:01:00:02"
 cookie=0x0, duration=272.16s, table=0, n_packets=0, n_bytes=0, idle_age=272, dl_src=00:00:00:00:00:02,dl_dst=00:00:00:01:00:02 actions=output:2
mininet> sh ovs-ofctl dump-flows 1_0_1 | grep "dl_src=00:00:00:00:00:02,dl_dst=00:00:00:01:00:02"
 cookie=0x0, duration=317.522s, table=0, n_packets=0, n_bytes=0, idle_age=317, dl_src=00:00:00:00:00:02,dl_dst=00:00:00:01:00:02 actions=output:2
mininet>

reverse path:

...

mininet> sh ovs-ofctl dump-flows 1_0_1 | grep "dl_src=00:00:00:01:00:02,dl_dst=00:00:00:00:00:02"
 cookie=0x0, duration=480.709s, table=0, n_packets=0, n_bytes=0, idle_age=480, dl_src=00:00:00:01:00:02,dl_dst=00:00:00:00:00:02 actions=output:1
mininet> sh ovs-ofctl dump-flows 1_2_1 | grep "dl_src=00:00:00:01:00:02,dl_dst=00:00:00:00:00:02"
 cookie=0x0, duration=528.298s, table=0, n_packets=0, n_bytes=0, idle_age=528, dl_src=00:00:00:01:00:02,dl_dst=00:00:00:00:00:02 actions=output:3
mininet> sh ovs-ofctl dump-flows 4_1_2 | grep "dl_src=00:00:00:01:00:02,dl_dst=00:00:00:00:00:02"
 cookie=0x0, duration=547.829s, table=0, n_packets=0, n_bytes=0, idle_age=547, dl_src=00:00:00:01:00:02,dl_dst=00:00:00:00:00:02 actions=output:1
mininet> sh ovs-ofctl dump-flows 0_2_1 | grep "dl_src=00:00:00:01:00:02,dl_dst=00:00:00:00:00:02"
 cookie=0x0, duration=564.839s, table=0, n_packets=0, n_bytes=0, idle_age=564, dl_src=00:00:00:01:00:02,dl_dst=00:00:00:00:00:02 actions=output:2
mininet> sh ovs-ofctl dump-flows 0_0_1 | grep "dl_src=00:00:00:01:00:02,dl_dst=00:00:00:00:00:02"
 cookie=0x0, duration=574.838s, table=0, n_packets=0, n_bytes=0, idle_age=574, dl_src=00:00:00:01:00:02,dl_dst=00:00:00:00:00:02 actions=output:2
mininet>
```

Dijkstra algorithm is not suit for fattree, for example: all inter-pod traffic will choose the first found core switch, which is switch (4,1,1) in our case, or other three core switches will just standalone!

4. ECMP algorithm

Using 5-tuple (source IP, destination IP, source port, destination port, IP protocol) to hash the coming packet into a specific path.



5. Two-Level algorithm

TODO
I tried to use NXM to push some prefix and suffix route into OpenvSwitch, but the OpenvSwitch keeped core dump, didn't find the reason yet …

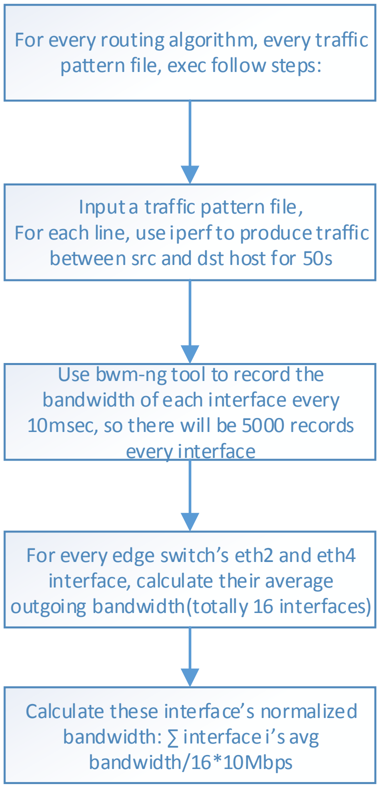
6. Given traffic patterns:

To evaluate the bisection bandwidth of the network, we need to calculate the average incoming traffic of all the hosts, here we use the average downwards traffic of all the edge switches, that is to say, all edge switch's interface 2 and 4 outgoing traffic, which is equal to all hosts' incoming traffic)

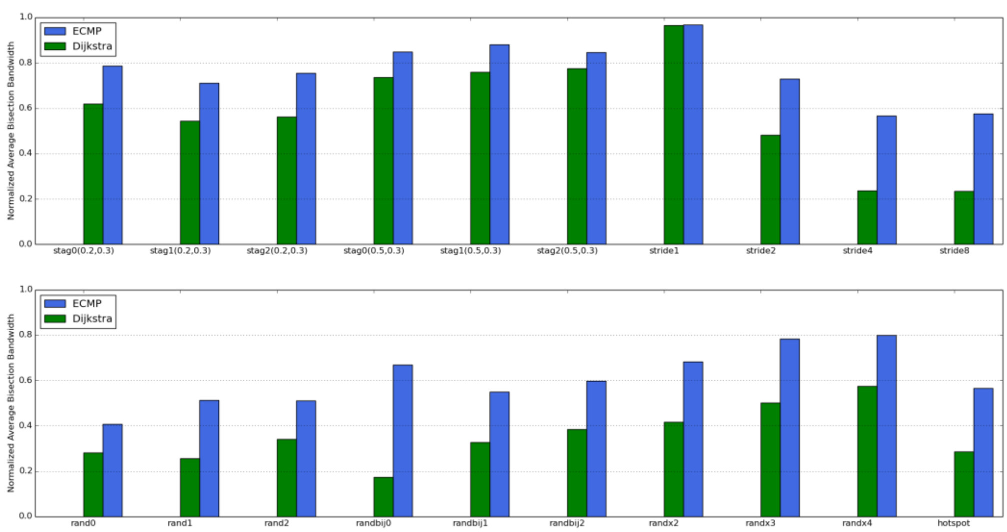
First, We use iperf to simulate traffic from source host to destination host for 50s.
Server: mnexec –*a dst_host_pid* –c –s 5001 > /dev/null &
Client: mnexec -a *src_host_pid* iperf –c *dst_ip* -p 5001 -t 50 -i 1 -yc > /dev/null &

We didn’t use the Iperf output, instead use bwm-ng to record all interface every 10 msec,
This is the output of bwm-ng, use Tx to evaluate a particular interface’s outgoing bandwidth.

```
input: /proc/net/dev type: rate
/         iface                   Rx                   Tx                Total
==============================================================================
          2_3_1:           0.00 KB/s            0.00 KB/s            0.00 KB/s
     4_1_2-eth4:        1173.17 KB/s            0.00 KB/s         1173.17 KB/s
     1_0_1-eth4:        1065.41 KB/s         1037.24 KB/s         2102.65 KB/s
     0_0_1-eth2:        1178.93 KB/s         1190.20 KB/s         2369.13 KB/s
     4_1_1-eth2:           0.00 KB/s          642.80 KB/s          642.80 KB/s
          3_0_1:           0.00 KB/s            0.00 KB/s            0.00 KB/s
     1_3_1-eth3:           4.06 KB/s         1193.47 KB/s         1197.53 KB/s
     0_3_1-eth1:           0.00 KB/s           22.34 KB/s           22.34 KB/s
     2_2_1-eth4:        1158.16 KB/s           42.50 KB/s         1200.66 KB/s
     1_2_1-eth2:        1189.31 KB/s          642.80 KB/s         1832.11 KB/s
          3_2_1:           0.00 KB/s            0.00 KB/s            0.00 KB/s
     4_2_2-eth1:           0.00 KB/s         1031.96 KB/s         1031.96 KB/s
     2_1_1-eth3:        1200.57 KB/s         1199.70 KB/s         2400.27 KB/s
     1_1_1-eth1:        1175.19 KB/s            0.00 KB/s         1175.19 KB/s
     3_0_1-eth4:        1190.96 KB/s         1188.27 KB/s         2379.23 KB/s
     2_0_1-eth2:        1196.13 KB/s         1192.42 KB/s         2388.55 KB/s
     3_3_1-eth3:          24.78 KB/s           11.43 KB/s           36.21 KB/s
     2_3_1-eth1:           0.00 KB/s         1173.17 KB/s         1173.17 KB/s
          0_0_1:           0.00 KB/s            0.00 KB/s            0.00 KB/s
          4_1_1:           0.00 KB/s            0.00 KB/s            0.00 KB/s
     4_2_1-eth4:        1173.17 KB/s         1173.17 KB/s         2346.33 KB/s
     3_2_1-eth2:           0.00 KB/s            0.00 KB/s            0.00 KB/s
     0_1_1-eth4:         573.46 KB/s         1196.25 KB/s         1769.71 KB/s
           eth0:           0.13 KB/s            1.99 KB/s            2.12 KB/s
     0_0_1-eth3:           0.00 KB/s            0.00 KB/s            0.00 KB/s
          0_2_1:           0.00 KB/s            0.00 KB/s            0.00 KB/s
     4_1_1-eth3:           0.00 KB/s           25.29 KB/s           25.29 KB/s
     3_1_1-eth1:         549.65 KB/s         1198.46 KB/s         1748.11 KB/s
     1_3_1-eth4:        1199.41 KB/s         1198.93 KB/s         2398.33 KB/s
     0_3_1-eth2:           0.00 KB/s            0.00 KB/s            0.00 KB/s
     1_2_1-eth3:           0.00 KB/s           14.12 KB/s           14.12 KB/s
     0_2_1-eth1:           0.00 KB/s         1192.44 KB/s         1192.44 KB/s
          1_1_1:           0.00 KB/s            0.00 KB/s            0.00 KB/s
          4_2_2:           0.00 KB/s            0.00 KB/s            0.00 KB/s
     4_2_2-eth2:        1199.23 KB/s            4.06 KB/s         1203.29 KB/s
     2_1_1-eth4:        1200.01 KB/s          165.00 KB/s         1365.00 KB/s
     1_1_1-eth2:         162.03 KB/s         1170.67 KB/s         1332.70 KB/s
```



And get the following picture: TODO add two-level algorithm compare



Analyze:

1)ECMP is apparently better than Dijkstra in fattree topology. The picture also shows the fact.

2) notice that in stride1, ECMP and Dijkstra are all nearly get full bandwidth.

3) for stride1, stride2, stride4, stride8, as more and more inter-pod traffic, the bandwidth is more and more small, especially Dijkstra, because it only use one core switch instead of four.
4) stag(0.5, 0.3) outperforms stag(0.2, 0.3), which is consistent with our expectation.
5)

7. run code

1) install mininet ubuntu VM as official guide
2) install necessary packets in mininet VM
apt-get install numpy
apt-get install bwm-ng
apt-get install python-matplotlib
3) ssh to this mininet VM, download the code as following:

```
root@mininet-vm:/home/mininet/pox/ext# git clone https://github.com/hmyan90/fattree.git
Cloning into 'fattree'...
remote: Counting objects: 111, done.
Receiving objects: 100% (111/111), 247.62 KiB | 0 bytes/s, done.
remote: Total 111 (delta 0), reused 0 (delta 0), pack-reused 111
Resolving deltas: 100% (64/64), done.
Checking connectivity... done.
root@mininet-vm:/home/mininet/pox/ext#
root@mininet-vm:/home/mininet/pox/ext# ls
fattree  README  skeleton.py
root@mininet-vm:/home/mininet/pox/ext# mv fattree/* ./
root@mininet-vm:/home/mininet/pox/ext# rm fattree -fr
root@mininet-vm:/home/mininet/pox/ext# ls
addresses.py.diff  DCRouting.py  fattree.pptx  INSTALL  monitor     README     run.sh     util.py
DCController.py    DCTopo.py     inputs       mn_ft.py  plot_rate.py  README.md  skeleton.py
root@mininet-vm:/home/mininet/pox/ext#
```

4) patch addresses.py file on pox

```
root@mininet-vm:/home/mininet/pox/ext# cp addresses.py.diff ../pox/lib/
root@mininet-vm:/home/mininet/pox/ext# cd ../pox/lib/
root@mininet-vm:/home/mininet/pox/pox/lib# ls
addresses.py       epoll_select.pyc  ioworker        pxpcap            socketcapture.pyc
addresses.pyc      graph             mock_socket.py  recoco            threadpool.py
addresses.py.diff  __init__.py       oui.txt         revent            util.py
epoll_select.py    __init__.pyc      packet          socketcapture.py  util.pyc
root@mininet-vm:/home/mininet/pox/pox/lib# patch addresses.py addresses.py.diff
patching file addresses.py
```

5) root@mininet-vm:/home/mininet/pox/ext# **sh run.sh**
this scripts will automatically run all traffic patterns in mininet and result a plot.png in /home/mininet/pox/ext/

6) If you don’t want to run all these patter at one time, and want to enter mininet CLI:

```
root@mininet-vm:/home/#/home/mininet/pox/pox.py DCController --topo=ft,4 --routing=ECMP
root@mininet-vm:/home/#mn --custom /home/mininet/pox/ext/DCTopo.py --topo mytopo --controller remote
```

```
mininet> 0_0_2 ping 1_0_2
PING 10.1.0.2 (10.1.0.2) 56(84) bytes of data.
64 bytes from 10.1.0.2: icmp_seq=1 ttl=64 time=78.1 ms
64 bytes from 10.1.0.2: icmp_seq=2 ttl=64 time=0.597 ms
64 bytes from 10.1.0.2: icmp_seq=3 ttl=64 time=0.074 ms
64 bytes from 10.1.0.2: icmp_seq=4 ttl=64 time=0.597 ms
64 bytes from 10.1.0.2: icmp_seq=5 ttl=64 time=0.074 ms
^C
--- 10.1.0.2 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4002ms
rtt min/avg/max/mdev = 0.074/15.890/78.112/31.111 ms
mininet>
```

this will allow you debug more …test whether ping is Okay(First packet is ARP request)

8. Reference

1) build Fattree topo and ECMP routing referred some code of ripl:
https://github.com/brandonheller/ripl

2)test bandwidth referred some code of hedera:
https://msharif@bitbucket.org/msharif/hedera.git