CSED702Y: Software-Defined Networking
Assignment 4: Design and Implement Fat-Tree using Mininet

Jian Li (gunine@postech.ac.kr)

Due at 11:59pm, April 22nd, 2015

In this assignment, we will design and implement a Fat-Tree topology and its routing protocol using Mininet and Floodlight. This assignment is consist of two parts. In the first part, you will implement a Fat-Tree topology using Mininet API. The detailed description can be found in Section 1. In the second part, you will first design an OpenFlow based Fat-Tree routing scheme, and then implement the routing scheme using Floodlight API. The resulting routing scheme can be an Floodlight application which runs on top of Floodlight controller. The detailed description can be found in Section 2.

This assignment should be finished individually and is worth a total 10% of the final mark. The Fat-Tree topology generation script should be written in Python, while the Fat-Tree routing protocol should be implemented using Java. The use of any supplemental library is permitted, which means you can freely import any Python library as well as Java library (.jar) if you need. Plagiarism is not tolerated, and once cheating is detected, students will be automatically failed in the course.

Have fun with Mininet and Floodlight!

Remark

- Before you start programming, please install Mininet network emulator and Floodlight OpenFlow controller first. Any Linux distribution will be fine, but I highly recommend the students to use Ubuntu distribution. You may use CentOS, but the installation procedure would be a bit more complicated.
- Make sure that you have read the presentation material on Mininet and Floodlight provided during the class. Also, it is strongly recommended to read the Fat-Tree paper to obtain the basic knowledge about Fat-Tree topology and its routing scheme in IP network [1].
- When implement the second part (Fat-Tree routing scheme), make sure that you exploit the Floodlight native API rather than RESTful API. No credit will be provided if the routing scheme is implemented using Floodlight's *Static Entry Pusher*, even if the resulting program works.
- With resulting program, all hosts in Fat-Tree should be pingable by simply using **pingall** command through Mininet CLI.

1 Fat-Tree Topology Generation (40 pt)

Among recently proposed Data Center Network (DCN) topologies, Fat-Tree is known as one of the most promising topologies for future DCNs. Fat-Tree originated from the Clos switching network and was adopted and introduced as a candidate for future DCNs in [1].

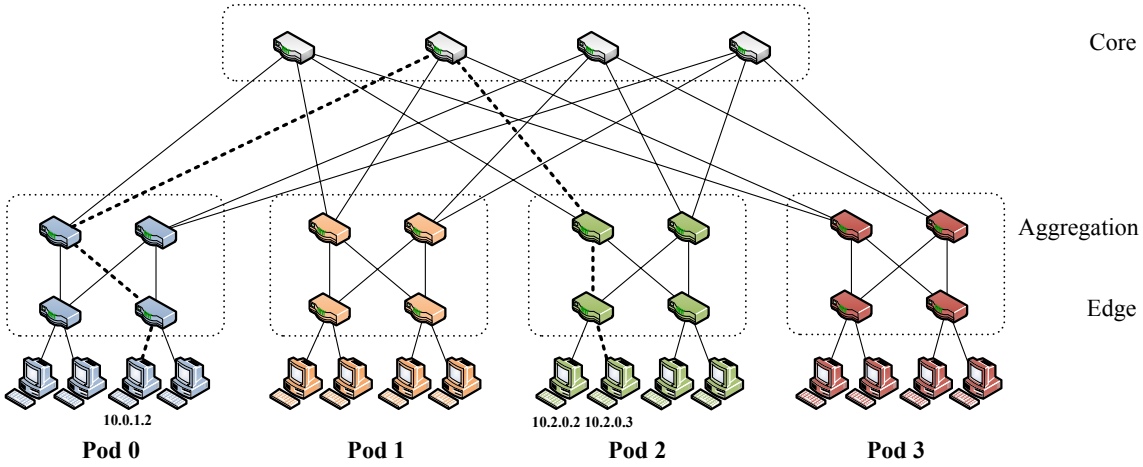


Figure 1: Simple Fat-Tree Topology with k = 4

In this part, you will implement a Fat-Tree topology generator using Mininet Python script. Fat-Tree is not a static network topology, instead it grows with respect to the value of **k**. Followings are some characteristics of Fat-Tree. A simple Fat-Tree topology with $k = 4$ has been shown in Figure 1.

- **Hierarchy:** all switches are categorized in accordance with hierarchical design. There are three levels in Fat-Tree, which are core, aggregation and edge. In other word, in Fat-Tree we can group the switches into core switches, aggregation switches and edge switches. The role of core switch is to forward traffic among aggregation switches, and that of the aggregation switch is to inter-connect core and edge switches. The edge switches reside in lowest level of Fat-Tree topology, try to forward traffic between hosts and aggregation switches.

- **Switch:** the switches in Fat-Tree should have identical port number which is specified by the value of k. E.g., with $k = 4$ Fat-Tree, all switches in Fat-Tree should have 4 ports. There are k pods, each containing two layers of $k/2$ switches. Each k-port switch in the lower layer is directly connected to $k/2$ hosts. Each of the remaining $k/2$ ports is connected to $k/2$ of the k ports in the aggregation layer of the hierarchy. There are $(k/2)^2$ k-port core switches. Each core switch has one port connected to each of k pods. The ith port of any core switch is connected to pod i such that consecutive ports in the aggregation layer of each pod switch are connected to core switches on $(k/2)$ strides. In general, a Fat-Tree built with k-port switches supports $k^3/4$ hosts.

- **Address:** the Fat-Tree follows strict IP addressing scheme, and in this assignment we will use a private IP block 10.0.0.0/8 to implement the addressing scheme. Since in

OpenFlow network, the switch cannot be assigned any IP address, we need to slightly change the original Fat-Tree addressing scheme. The pod switches (comprised of aggregation and edge switches) are given **D**ata**P**ath**ID** (DPID) of the form $00:00:00:00:00:pod:switch:01$, where pod denotes the pod number (in $[0, k-1]$), and switch denotes the position of that switch in the pod (in $[0, k-1]$, starting from left to right, bottom to top). The core switches are allocated the DPID of the form $00:00:00:00:00:k:j:i$, where j and i denote that switch's coordinates in the $(k/2)^2$ core switch grid (each in $[1, (k/2)]$, starting from top-left). The address of a host follows from the pod switch it is connected to; hosts have addresses of the form: $10.pod.switch.ID$, where ID is the host's position in that subnet (in $[2, k/2+1]$, starting from left to right). Therefore, each lower-level switch is responsible for a /24 subnet of $k/2$ hosts.

- **Port:** there are k-ports in each switch, and the index number of port is assignment in inverse-clockwise. Therefore the left and down most port is assigned port number 0, while the right and top most port is assigned port number $k-1$.
- **Link:** all network elements should be inter-connected through logical links. The ith host in a subnet should be connected to the ith port of edge switch which manages the subnet. Since each edge switch has $k/2$ number of hosts, therefore $k/2$ ports are occupied to connect to the hosts. The residual ports of edge switch are connected to aggregation switches. The $(k/2+i-1)$th port of edge switch which has DPID $00:00:00:00:00:pod:j:01$ should be connected to the jth port of aggregation switch which has DPID $00:00:00:00:00:pod:(k/2+i-i):01$.

By keeping those characteristics in mind, you need to design your topology generation program. The input of the resulting program should be k, and in accordance with the value of k, the corresponding Fat-Tree topology should be generated in Mininet.

2 Fat-Tree Routing Protocol (60 pt)

In this part, you will implement Fat-Tree's routing protocol. Assume that you have already created a Fat-Tree topology by specifying the k value. The load balancing is the most important issue in DCN topology, to keep this in mind, you will implement a routing protocol which can evenly distribute the traffic load among all switches. In [1], the authors proposed a Two-Level routing table to realize the even-distribution mechanism. The key idea of the proposed method is to allow two-level prefix lookup. Each entry in the main routing table will potentially have an additional pointer to a small secondary table of (suffix, port) entries. A first-level prefix is terminating if it does not contain any second-level suffixes, and a secondary table may be pointed to by more than one first-level prefix. Whereas entries in the primary table are left-handed (e.g., /m prefix masks of the form 1^m0^{32m}), entries in the secondary tables are right-handed (e.g., /m suffix masks of the form $0^{32m}1^m$). If the longest-matching prefix search yields a non-terminating prefix, then the longest-matching suffix in the secondary table is found and used. Figure 2 shows an example of routing table at switch $00:00:00:00:00:02:02:01$. The prefix entries are used to match and forward the packets from aggregation switch to hosts, while the suffix entries are used to match and forward the packets from hosts to aggregation switches. Note that for load-balancing purpose, the output port of suffix entries are calculated by using *mod* bit operation. For

example the switch which has DPID $00:00:00:00:00:x:y:01$, has $k/2$ suffix entries and ith suffix entry's output port number can be calculated by using Equation 1.

$$SUFFIX\_OUTPUT\_PORT\_NUM = (i - 2 + y) mod (k/2) + (k/2) \quad (1)$$

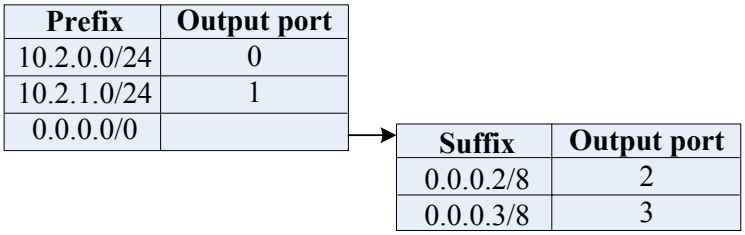


Figure 2: Fat-Tree's Two-Level Routing Scheme

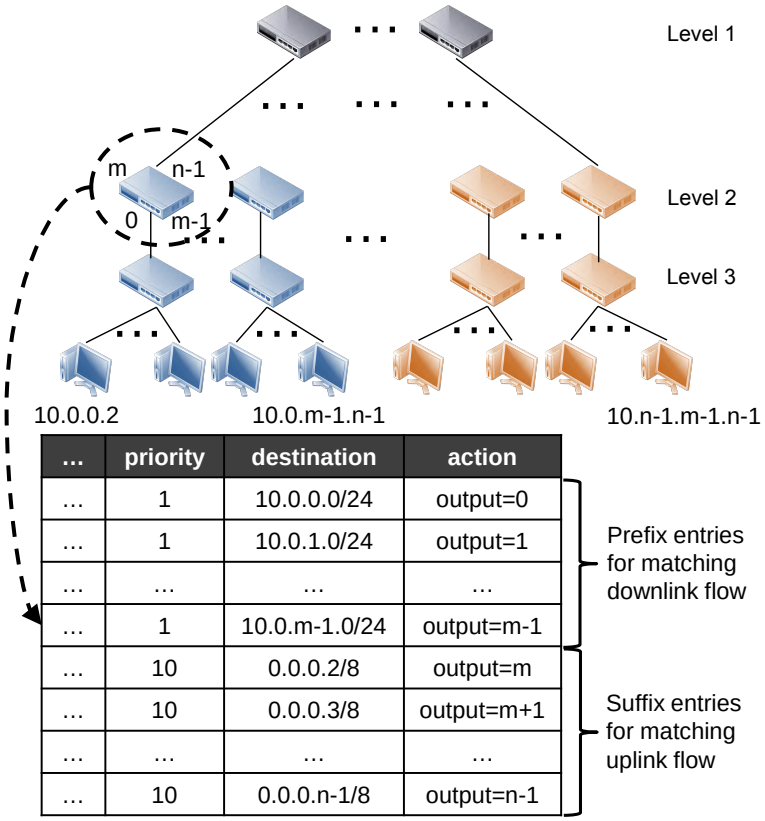
Since from version 1.1, OpenFlow starts to support multiple flow tables, you can exploit this feature to implement the aforementioned routing scheme. Or if you feel difficulties on implementing the routing scheme using multiple tables, you can exploit the priority field of flow table to implement the routing scheme. Note that for latter case, you only need to use one flow table for each switch, and by assigning higher priority to prefix entries and lower priority to suffix entries, you can mimic the behavior of two-level routing table just by using one flow table. For the detail refer to Figure 3.
The routing program should be implemented as a Floodlight module, and will be automatically loaded with the initializing the Floodlight controller. As long as the switches added in the network topology, the routing program should automatically detect the event, identify the switch and based on the switch's DPID, the corresponding flow entries should be inserted to the switch.

3 Submission Instruction

You should firmly follow the submission instruction, otherwise there will be a penalty. Late assignments may be handed in, but there will be a penalty of 20% of the mark for assignments turned in less than one day late, and an additional penalty of 10% for each day thereafter. The submitted source files must be compilable and the binaries must be executable. No credits will be provided if either the sources are failed to compile or the binaries are failed to execute.

1. Develop the Fat-Tree topology generation scripts and place in "topo" directory. All source files which related to topology generation should be located under "topo" directory. Create a script file (e.g., bash shell) which initiates the Mininet with the Fat-Tree topology. The value of k can be configure either inside source file or through CLI argument.

2. Develop the Fat-Tree's routing scheme as a Floodlight module. You need to create a new Java package (e.g., kr.ac.postech.fattree), and place all source files which related to Fat-Tree's routing scheme under the package. Note that since you do not need



...	priority	destination	action
...	1	10.0.0.0/24	output=0
...	1	10.0.1.0/24	output=1
...	...	...	...
...	1	10.0.m-1.0/24	output=m-1
...	10	0.0.0.2/8	output=m
...	10	0.0.0.3/8	output=m+1
...	...	...	...
...	10	0.0.0.n-1/8	output=n-1

Figure 3: Modified Fat-Tree's Routing Scheme using Prioritized Routing Table

to modify any Floodlight's Java code, hence, only need to submit the newly added source files contained in your package.

3. Document your program. The content should include: 1) the instruction on how to compile and run your program, 2) the detailed algorithm what you used to implement the topology generation and routing scheme. To be more specific, the compilation instruction should include the content that to be added in Floodlight configuration file, so that the Floodlight loads your module program at initialization phase. For algorithm, you may add some pseudo code with corresponding description. The length of document should be under 6 pages in LNCS paper format. The report should be formatted as "`.PDF`" and written in `English`.

4. Compress the "topo" directory and Java packages directory (e.g., kr directory) along with the document as a zip file and send to `(gunine@postech.ac.kr)` through email.

References

[1] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *ACM SIGCOMM Computer Communication Review*, vol. 38, pp. 63–74, 2008.