Traffic Steering Application for Datacenter Networks

The Hebrew University of Jerusalem

School of Computer Science and Engineering

Laith Abu-Omar, Abed Ewiwi

This project was done as part of the Course "Workshop in Communication Networks" at the Hebrew University of Jerusalem.

1. INTRODUCTION

The use of middleboxes in networks became an elementary need, especially when it comes to datacenter networks. However, designing Networks that support and make use of middleboxes is not an easy task. Fortunately, Software Defined Networking (SDN) provides a bunch of astonishing chances and alternatives for such implementations, but also has its own challenges.

In this project, we aim to utilize the capabilities offered by the use of SDN in order to implement an advanced Traffic Steering Application (TSA) for datacenter networks. Our implementation targets a specific type of networks called Folded Clos Networks (or Fat Trees). A Clos Network is a multistage switching network first formalized by Charles Clos in 1953, and was used for building telecommunication networks. A Folded Clos Network (or Fat Tree) is a special type of Clos Networks where the topology of such a network takes the shape of a tree (may vary in number of levels) where links closer to the root intend to have higher bandwidth than those closer to the leaves of the tree. It was invented by Charles Leiserson in 1985, and since then, it has been widely used as a reference topology for datacenter networks.

Building the project required the use of the following:

1) Mininet – provided the capability of demonstrating the Network virtually.
   See http://mininet.org/ for more on Mininet.
2) Openflow POX – SDN Controller written in Python.
   See https://openflow.stanford.edu/display/ONL/POX+Wiki for more on the POX controller.
3) Click – provided the capability to create the middleboxes for the Network as click elements, and implement the NFs.
   See http://read.cs.ucla.edu/click/click for more on Click.

Our implementation supports a Topology with several Network Functions (NFs) and policy chains that require matching packets to go through the specified NFs within the datacenter network before reaching its destination. We do so by accepting a configuration file that contains a list of policies (matching fields to check for using POX Match structure), and the corresponding NFs that matching packets should be directed through. (See "policyConfig" file for an example of the format)
Supported NFs are any Click elements implemented to act as a Virtual NFs.

• Initial construction:
We knew from the beginning that the several connections between the levels of switches had something to do with the efficiency of the network and load balancing. In addition, we knew that the splitting into levels is mainly needed for supporting a huge number of hosts while maintaining a decent performing network. So we understood that we need to configure a way to deal with the loops in the network without eliminating any links. Otherwise, we will be missing the whole point of the Fat Tree topology. So to sum up, we started thinking of a strategy on how to deal with the topology we have in order to install the rules in a specific way so to enforce the policies chains in the network.

We were trying to figure out some assumptions and rules regarding the type of networks we would like to support. After considering several implementation issues, we decided that the best way is to start with a simple naïve construction and then widen it to include the full functionality by sticking the pieces carefully and slowly. We started by dealing with Fat Tree topology consisting of 2 levels and full connection between the two levels of ToR and Core switches (i.e. a full bipartite graph). So we built our TSA based on these assumptions and ended up with a basic TSA that supports any Fat Tree consisting of 2 levels containing any number of switches in each level as long as the two levels are fully connected. This TSA can be found in the python file **"basic2LevelTSA.py"**. Further explanation regarding this TSA is mentioned in the METHOD section.

• Building the fully functional advanced TSA:
Now it was time to take the naïve TSA built previously, and improve it so that it supports any provided datacenter network topology. We now changed our perspective and started figuring out how to support any 2 or 3 level Fat Tree consisting of ToR, Spine (if 3 level), and Core switches, which doesn't necessary have full connection between levels, **but assuming there is at least one way from each host to another, and to every NF in the network**.

The first step was creating a basic TSA that is capable of doing the required in a rather naïve way, having less rules installed, and referring to the controller on several occasions. The latter TSA can be found in the python file **"basic3LevelTSA.py"**. This TSA does the job required in a very basic way.

The next step was to improve our previously created TSA in every possible way. The implementation of this step ended in an Advanced TSA that achieves higher and better performance and functionality. Further details on the improvements done in this part is mentioned in the METHOD section. The advanced TSA can be found in the python file **"advancedTSA.py"**.

2. METHOD

We will give a detailed explanation on each one of the three TSAs that we created. But first let us give a bird's eye view of the general common principles used in the 3 TSAs.

All of the 3 TSAs have 3 main parts, the first is responsible for parsing the policies file and extracting the information from it.

The second part is responsible for discovering the Topology of the network (This is the Discovery class in all the TSAs python files).

The third and last part is responsible for applying the policies by installing rules on all switches accordingly and interacting with switches in dealing with packets that don't match the installed rules (This is the Tutorial class in all the TSAs python files).

In the rules installation process, there are things in common between all of the three TSAs, so we mention them now briefly, and emphasize on each one later.

We **separate the switches into categories**, ToR with no NF (i.e. don't have NFs connected to them), ToR with NF (i.e. have one or more NF connected to them), Core, and Spine (added in basic3LevelTSA and onwards). When it comes to installing rules, we first determine to which category from the mentioned ones the switch belongs, and only then we install the rules for it.

As a part of the rules, and packets tracking, we use **VLAN tagging**. The standard we took in tagging the packets is the one we saw in the fourth paper (SIGCOMM 13). We give a packet the VLAN tag of a NF when it successfully completes that NF (i.e. authorized by that NF to move on).

The VLAN tags are given to the NFs at the moment of parsing the policyConfig file, and in the order we meet the NF.

The VLAN tags start from 2, because 1 is reserved for special use in our TSA to tag packets that are done (i.e. passed all of the NFs in the policy chain).

We chose **the moment to install all the rules on the switches** to be the moment when we find the location of the last NF, which is crucial for setting the rules as we need to know for each NF to which ToR switch it is connected. In addition, we keep track of the discovered edges in the network at each time an LLDP packet is received (this was achieved by setting a timer on each switch to flood an LLDP packet to all its neighbors every 1 second).

See https://en.wikipedia.org/wiki/Link_Layer_Discovery_Protocol for more on the LLDP protocol.

In addition, whenever a new edge is found, we delete all the rules from all switches and install them again, and that's because the order of rules in switches is important, and changing it might result in an incorrect behavior that doesn't maintain the policies as expected.

Regarding **the location of the NFs**, we found the best way to do it is to order ToR switches to send ARP requests to the IP addresses of the all NFs given to us in the policyConfig file with the default "00" IP and MAC address. Then we catch the ARP reply in the first receiving ToR switch in order to determine the location of the NF.

<u>**Important Note:**</u> Due to the mechanism we used, policy enforcing rules are installed on switches as early as possible by our TSA. This was designed mainly to allow for quicker response of the TSA. However, this requires some caution when using the "ping" command, as policy rules are already there when attempting to ping, this might cause conflict with rules and leads to undesired behavior of the TSA, so **please don't ping from\to NF serving hosts especially when there are policies that match the ICMP packets of the ping**. In addition, performing **pingall** in the beginning will also cause problems for the same reason.

- **basic2LevelTSA:**

Let us follow the order we took above in dividing the tasks into 3 main categories, and explain each only briefly.

➢ Discovering the Topology:

This part was done using LLDP packets as mentioned earlier. As a result, we got a clear image of the network we are dealing with in terms of switches and links between them.

<u>**Note:**</u> It is important to emphasize on the difference from the Spanning Tree Protocol (STP) which applies Kruskal's Algorithm to get rid of loops in the network, which is irrelevant here as we mentioned above, since in Fat Tree topologies we utilize the presence of several possible ways from one host to another (i.e. Loops) in order to achieve higher efficiency, so we deal with loops in a different way explained later, but we don't eliminate a single link.

➢ Discovering the Hosts:

This task was done by maintaining a global dictionary named hostsLocation where the keys are the MAC addresses of the hosts, and the value is a list containing the DPID of the ToR switch, and the port which the host is connected through.

This dictionary is filled on each time a new host sends a packet to its gateway ToR switch.

Later on, we modify the way we fill and maintain this dictionary as we support moving hosts in our TSA.

➢ Parsing the PolicyConfig file:

As mentioned above, this part is done with the help of 2 classes (RulesTable, and Policy), and a global function (parseConfigFile).

The function parseConfigFile reads the file named policyConfig assuming it is always present under "/home/mininet", and extracts the relevant information from it as follows. It inserts all the NFs into a global dictionary called "services", where the keys are the names of the NFs as they appeared in the policyConfig file, and the value is a list containing a Boolean value indicating whether the NF was found or not yet, the IP address of the NF, the VLAN tag assigned to the NF, the ToR switch connected to the NF, and the port of the ToR switch that connects it to the NF.

The function also extracts the policies from the file, creates instances of the Policy class, and adds those to a global list of policies objects called "policies". The Policy object has 2 attributes, one is the match list, and the other is the chain list of the policy.

➢ Enforcing Policies (Installation of the rules):

This part is the hardest and most complicated one, so we will do our best to make it clear.

After deep thinking, we found out that the best way is to divide switches in the network into 3 categories, ToR switches with no NFs (i.e. don't have NFs connected to them), ToR switches with NF (i.e. have one or more NF connected to them), and Core switches.

Let us consider each case separately:

❖ ToR switches with no NFs:

This category is the simplest. Only one type of rules is installed on these switches. For each policy in the policies list mentioned above, we install a **rule** for each policy that matches on all of the policy's match list elements, and in addition matches on VLAN tag to be equal to None.

(done using the instruction "fm.match.dl_vlan = of.OFP_VLAN_NONE")

The **action** for the rule is to output to a Core switch (it doesn't matter which Core switch it is, since we assume full connectivity between the levels - This assumption is relaxed later on in the basic3LevelTSA)
- The function responsible for this part is "installNonNFToRRules()".

❖ ToR switches with NFs:
This category is the hardest. It has several types of rules that need to be installed according to a specific order so that we get the desired behavior from the network. The latter rely on the fact that rules installed on Mininet switches are matched according to their order (i.e. If there are two rules matching a packet, then the first one's action is done as it appears before the second).
The installation process of the rules in this part is done by iterating over the policies list twice.

The first loop has two parts, the first checks whether the last NF (i.e. the last NF in the chain of the current policy) is connected to the switch being figured, if so we install a **rule** that matches on all of the policy match list elements, and on the in port of the switch to be the one connected to the NF. The **action** for the rule is to refer to the controller. The reason we needed to add this rule is the fact that we still don't know the place of the destination host, so we thought on referring to the controller first in an attempt to modify the rule or delete it, and add a new rule for outputting to the host. But after some more thinking, we paid attention that we actually still need that rule for future packets sent to other hosts, and unfortunately we couldn't solve this problem even in the advanced TSA as we will explain later on.
Now back to our rules, after checking the latter and adding the rule to refer to the controller when the packet completes its policy chain, we iterate over the chain list of the policy while considering each successive pair of NFs at the same time. We check whether the switch is connected to the first NF, if so we continue to check if it is also connected to the successive NF.
If the switch is connected to both NFs, then we install a **rule** that matches on all of the policy match list elements, and on the in port of the switch to be the one connected to the first NF. The **action** for the rule is to output the packet to the second NF which is also connected to the same switch (the port of that NF can be found in the "services" dictionary explained earlier)
However, if the switch is not connected to the second NF (i.e. It is connected to the first NF, but the second NF is placed under a different ToR switch), then we install a **rule** that matches on all of the policy match list elements, and on the in port of the switch to be the one connected to the first NF. The **action** for the rule is to output the packet to a Core switch (again it doesn't matter which Core switch it is for the same reason as before).

The second Loop iterates over the policies once again, but this time installing a different type of rules. For each policy in the list, we iterate over its chain list and do the following.
Check whether the first NF in the chain is located under a different switch than the one being figured, if so then we install a **rule** that matches on all of the policy match list elements, and in addition matches on VLAN tag to be equal to None. The **action** for the rule is to output to a Core switch (again it doesn't matter which Core switch it is for the same reason as before).
While iterating over the chain list of the policy, when we encounter a NF that is connected to the current switch, we install a **rule** that matches on all of the policy match list elements, and on the VLAN tag to be the appropriate one according to the situation (i.e. if the NF is the first, then the match is on VLAN None, otherwise the match is on the tag of the previous NF). In addition, there is a match on the in ports in order not to confuse with packets incoming from the NF itself. The **action** for the rule is to output through the port connecting the switch to the NF.
- The function responsible for this part is "installNFToRRules()".

❖ Core switches:
Assuming that all Core switches are connected to all ToR switches in the lower layer, the task of installing rules for the Core switches consists of only one loop that iterates over the policies list, and for each policy iterates on its chain list, installing a **rule** that matches on all of the policy match list elements, and on the VLAN tag to be the appropriate one according to the situation (i.e. if the NF is the first, then the match is on VLAN None, otherwise the match is on the tag of the previous NF). The **action** is to output to the ToR switch that is connected to the desired (next) NF.
- The function responsible for this part is "installCoreRules()".

It is important to emphasize on the need to install rules in the mentioned order. Otherwise, we might end up with an undesired behavior of the network. Consider for example installing the rule for inserting packets that match a certain policy to the target NF on a ToR switch before installing the rule for packets incoming from the NF. This will cause a back and forth sending of the packet between the switch and the NF, which is not what we meant. Therefore, we found out that it is very important to maintain the order of installation of the rules on switches.

Now it remains to explain the behavior of our TSA when the packet completes its security checks with all the NFs chain or incase the packet doesn't match any policy and therefore needs to be forwarded to its destination host. We divide this into two cases.

The first is when the location of the destination host is previously known by the TSA controller. In such a case, we order each switch to forward (send) the packet to a specific port computed by us at the controller without setting a rule. We chose not to install rules as we initially thought it would raise the level of complexity by too much, and won't help by much. However, we later discovered a way to reduce the overhead of this issue by installing additional rules (further details in advancedTSA).

The second case is when the location of the destination host is unknown by the TSA. In such a case, we order the ToR that is connected to the last NF of the chain (reminder: a rule was installed on these switches to refer to the controller when the packet arrives from the last NF of the matching policy) or to the ToR that received a packet that doesn't match any policy at all to tag the packet with VLAN 1 (meaning the packet is DONE with the policy).
On all Core switches, we installed rules at the end of the installation process to flood packets with VLAN tag 1. And installed rules on all ToR switches to strip the VLAN tag (i.e. set it to None), and flood packets with VLAN tag of 1 to all hosts connected to the switch.
It is important to mention that ARP requests, which are sent to destination MAC address "FF" fell exactly on those flood rules. And that's how we made sure that ARP requests arrive at their destination.

That was all of it for the basic2Level TSA which was rather simple. Now let us move on one step further to a more general TSA.

- **basic3LevelTSA:**

We modified our TSA so that it supports any Fat Tree network consisting of 2 or 3 levels (it is important to note that this TSA does everything which is done by the previous basic2Level TSA and has more functionality, so it doesn't only support any 3 level Fat Tree, but rather any 2 or 3 level Fat Tree networks). The tasks can still be divided into 3 main categories, so we will follow the same order as above while mentioning the main differences compared to the previous TSA.

➢ Discovering the Topology:
This part works in the exact same way as in the basic2Level TSA, so we will skip it.

➢ Discovering the Hosts:
This part also works similarly to that of the 2 level TSA.

➢ Parsing the PolicyConfig file:
This part also works similarly to that of the 2 level TSA.

➢ Enforcing Policies (Installation of the rules):
In a similar way to what we have done in the basic2Level TSA, here we divided the switches in the network into 4 categories, ToR switches with no NFs (i.e. don't have NFs connected to them), ToR switches with NF (i.e. have one or more NF connected to them), Core switches, and Spine switches.
Let us consider each case separately:

❖ ToR switches with no NFs:
As in the basic2Level TSA, only one type of rules is installed on these switches. For each policy in the policies list, we install a **rule** that matches on all of the policy match list elements, and in addition matches on VLAN tag to be equal to None. The only difference is in the action, which now becomes a bit more complicated. The **action** for the rule is to output to a Spine (or Core in case of a 2 level Fat Tree) switch, but not any Spine (Core), we first needed to calculate the list of all Spine (Core) switches that can reach the target ToR switch. The latter is done with the help of the function "getWayPort()".
- The function responsible for this part is "installNonNFToRRules()".

❖ ToR switches with NFs:
Again, the overall structure remains similar to the one in the basic2Level TSA. It has several types of rules that need to be installed according to a specific order so that we get the desired behavior from the network.
The installation process of the rules in this part is done by iterating over the policies list twice.
The first loop has two parts, the first checks whether the last NF (i.e. the last NF in the chain of the current policy) is connected to the switch being figured, if so we install a **rule** that matches on all of the policy match list elements, and on the in port of the switch to be the one connected to the NF. The **action** for the rule is to refer to the controller, similarly to the previous TSA. The reason we needed to add this rule is also the same as above.
The second part of the first loop iterates over pairs of NFs in the same way as in the basic2Level TSA. The case where the switch is connected to both NFs is handled similarly, the **rule** that matches on all of the policy match list elements, and on the in port of the switch to be the one connected to the first NF. The **action** for the rule is to output the packet to the second NF which is also connected to the same switch (the port of that NF can be found in the "services" dictionary explained earlier).
However, the case where the second NF is connected to a different ToR switch has a slight modification in the **action** which is outputting to a specific Spine (Core) that can reach the target ToR switch and not to a random Core switch as before.
Regarding the second Loop, the **rule** installed for the first NF in the chain matches the same fields as before, but the **action** is now again to output to a specific Spine (Core) that can reach the target ToR switch. Such a switch is found with the help of the function "getWayPort()". The part that installs the rules for forwarding packet to NFs that are under the switch remained similar.
- The function responsible for this part is "installNFToRRules()".

❖ Core switches:
As in the basic2Level TSA, the task of installing rules for the Core switches consists of only one loop that iterates over the policies list, and for each policy iterates on its chain list, installing a **rule** that matches on all of the policy match list elements, and on the VLAN tag to be the appropriate one according to the situation (i.e. if the NF is the first, then the match is on VLAN None, otherwise the match is on the tag of the previous NF). The difference is in the **action** which is now to output to the ToR switch that is connected to the desired (next) NF in case of 2 level Fat Tree, and to output to a Spine switch in case of 3 level Fat Tree. The process of finding that specific switch is done by the function "getWayPort()".
Another crucial point is the fact that we wanted to support a topology were levels aren't fully connected, in such a case there might be some ToR switches that aren't reachable from this specific Core switch but reachable from others. So before installing the rule on a Core switch, we first check whether the target ToR switch is reachable from it or not.
- The function responsible for this part is "installCoreRules()".

❖ Spine switches:
The case of Spine switches is somehow similar to the Core switches. The job of the rules is to secure the forwarding of policy packets between the NFs, so they are able to find their way to the target ToR switch. The installation of the rules for the Spine switches consists of one loop that iterates over the policies list, and for each policy iterates on its chain list, installing a **rule** that matches on all of the policy match list elements, and on the VLAN tag to be the appropriate one according to the situation (i.e. if the NF is the first, then the match is on VLAN None, otherwise the match is on the tag of the previous NF). The difference is in the **action** which is now to output to the ToR switch that is connected to the desired (next) NF in the case where the ToR switch is directly connected to the Spine switch, and to a suitable Core switch in the case where the ToR switch is not directly connected to the Spine switch. The process of finding that specific switch is done by the function "getWayPort()". It is also possible that the target ToR switch is not reachable from the Spine switch, in such a case we simply don't install the rule for that Spine switch as we did with the Core switches.

Regarding the part at the end of the rules installation process, where we installed the rules for flooding packet destined to unknown hosts. This part was replaced by regular sending of the packet and continuous referring to the controller in this TSA. That was done mainly due to the problems that we encountered when we attempted to set rules for auto-forwarding of packets to hosts when they are done with the policy. The problem was caused with policy rules where a more specific policy (i.e. higher priority) has a shorter chain than a less specific one (i.e. lower priority).
This behavior was partially improved in the advanced TSA which is explained below.

- **advancedTSA:**

This TSA represents Part 3 of the project. It stands as the improved version of the basic3Level TSA mentioned above. It has additional features that makes it better than the basic3Level in terms of both performance and functionality.

The implementation of the different parts in the TSA are pretty much the same as in the basic3Level TSA. The changes made in each section are due to the improvements. So we think it is better to explain each improvement, how it was done, and the changes that were done in order to allow for it.

✓ Modification of rules system:

We felt that the idea to drive the packets after they have done their policy tour is not the best we can do, and were worried about the performance of our TSA (which we will later show how incredibly important it was). So we did our best to add as much rules as possible in order to minimize the number of references to the controller at the end of the policy tour (or if the packet doesn't match any policy). We ended up installing rules for Core, Spine, and ToR switches to forward packets to specific MAC address (host) if they have special VLAN tag.

We reused the special VLAN for done packets, but this time not for flooding packets. Whenever a ToR switch refers to the controller, we check whether the location of the destination host is known, if not, then we simply flood the packet without setting a rule. Otherwise, we calculate the way to reach it, and install a **rule** on the ToR switch to set the VLAN tag to SPECIAL (which is a special value not equal to any other VLAN tag) and the **action** is to output the packet from the calculated port.

The same trick is done with the Core, Spine switches, and destination ToR switches, but here the **rule** is to match on the VLAN tag to be SPECIAL, and on the destination host, while the **action** is the same as for the sending ToR. We separate between the sending and receiving ToRs by checking whether the incoming port is one of the ports discovered by us in the Discovery class (since LLDP packets aren't forwarded to hosts).

Thus, in each time a packet that matches some policy is sent, we let exactly one ToR switch (which is the one connected to the last NF) refer to the controller for only one time, and the rest of the way is handled by rules. The reason we couldn't get rid of that one referring to the controller is the possibility to have policies as described above, and in the link. In addition, we needed that reference to the controller in order to detect moving hosts (see below). The case where the packet doesn't match any policy is now handled completely with rules when the location of the destination host is known.

✓ Random outputting:

Seeking better performance lead us to a simple idea of dividing the load between upper layer switches instead of congesting some of them while leaving others free.

We made a small change in the way we calculate the output port for a switch when we install rules for policy enforcing and for done packets.

Instead of always choosing the first encountered port from the available ones, we place all of them in a list, and randomly choose one of them to be the output port. This results in splitting the load on the links, thus yielding better numbers.

✓ Moving hosts:

Instead of looking only at hosts that weren't seen before, we check hosts that we already know their places, and compare the previously known location (i.e. ToR switch) with the current one. If there is a change we update the location of the host, delete all rules installed on switches in the network, and reinstall the rules on all switches. The fact that we have a rule installed on every ToR switch to refer to us on packet completion allows us to detect change of location for hosts sending packets without policy. And for packets with policy, we modified the rule installed on the ToR switches, which was previously sending the packet to the NF, and we added another action of referring to the controller while setting a special VLAN tag called HOST, which let us know that this is only for checking change in host location. After we have done our check, we simply ignore the packet sent to the TSA (the controller). The case where the packets doesn't match any policy is simple, as the switch certainly refers to the controller, and we do the rest.

It is important to note that we only detect the change in the location of a host when the host itself sends a packet from its new location. Otherwise we remain blinded to the that change. So if a host changes its location, all the packets destined to that host will be lost until a packet is received from that host. We could have modified the mechanism in a way similar to that of the moving NFs (see next part) in order to detect change in location without the need to receive a packet from the host, but felt there is no need as hosts send packets all the time, and this might cause extra load on the network.

Note: Due to the fact that we delete all rules and reinstall them, there might be some transient instability in the network which might lead to loss of some packets when hosts move, but this will disappear as soon as the new rules are installed.

✓ Moving NFs:

In this part, we benefited from the assumption that we have the IP addresses of the NFs (due to the format of the configuration file). We adjusted the ARP sending mechanism of the ToR switches explained earlier that was used to detect the location of the NFs, and created a timer for each ToR switch upon its connection in the Discovery class, that sends ARP requests to all NFs from that switch every 5 seconds. In addition, we added a part in the _handle_packetIn() function that determines whether there is a change in the location of a NF or not. If a change in location is detected, we delete all rules from all switches in the network, and reinstall all rules once again. The timer is stopped in the _handle_connectionDown() function (i.e. when the switch shuts down).

It is important to note that unlike the host case, here we do detect the change in location of the NF without the need to receive a packet from it (which is not acceptable to assume), and that's due to the continuous sending of the ARP requests.

Note: Due to the fact that we delete all rules and reinstall them, there might be some transient instability in the network which might lead to loss of some packets when NFs move, but this will disappear as soon as the new rules are installed.

✓ Multiple instances of the same NF:

In this part, we improved the Parser of the configuration file so that we support a Network containing multiple instances of the NF (for example: 2 Firewalls, and 3 Intrusion Detection Systems in the same network).

The main advantage gained was the ability to lower the load on a NF that is used frequently by dividing packets between the instances of that NF. In other words, we balance the load on a specific NF by sending a portion of the packets to one instance of that NF, and sending the remaining packets to other instances. In this way, we minimize the delay caused by overloaded NF, by giving the Network manager the ability to add more instances of that overloaded NF, and apply the mechanism of load balancing between multiple NF instances implemented by the TSA.

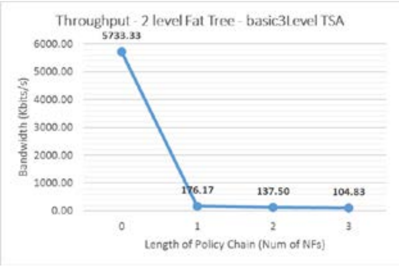
✓ CIDR prefixes:
This one is rather simple. We added a function that takes as input the IP address, and the mask as an integer. It computes and returns a dotted format of the matching subnet address.
In the part of installing the rules, we modified the part that fills the ipv4_src, and ipv4_dst fields. It now detects whether the string contains “/”, and if so calculates the matching subnet address and uses the wildcard functionality in the POX match structure in order to satisfy the required.

3. RESULTS

We measured the performance of the basic3Level TSA, and the improved version of it, the advanced TSA. We performed two tests on the TSAs. The tests are the throughput (measured using **iperf**), and latency (measured using **ping**). The experiment was held twice for each TSA, once with a 2 level Fat Tree (see Appendix A), and then with a 3 level Fat Tree (see Appendix B). In measuring the throughput (bandwidth), we took 6 results by running iperf 6 times in its default arrangement with the default interval of 5 seconds, and calculated the average of the 6 results collected. While measuring the latency was done by pinging for 20 times and taking the average over the results.
We will now provide results from the experiments as tables and graphs, and discuss these in the next section.

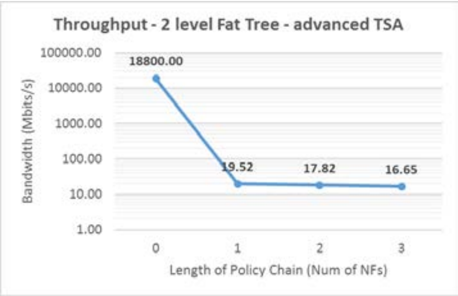
Throughput analysis for basic3Level TSA with 2 level Fat Tree (Appendix A):

2Level - basic3LevelTSA				
	Policy Chain length			
	0	1	2	3
Bandwidth (Kbits/s)	5300	130	115	99
	5700	118	116	99
	5400	230	210	103
	5700	223	125	100
	5500	130	130	109
	6800	226	129	119
AVG	5733.33	176.17	137.50	104.83

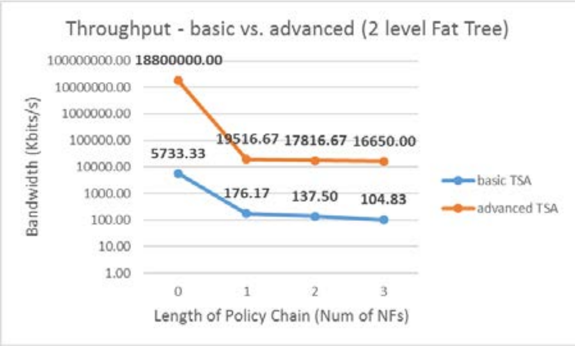


Throughput analysis for advanced TSA with 2 level Fat Tree (Appendix A):

2Level - advancedTSA				
	Policy Chain length			
	0	1	2	3
Bandwidth (Kbits/s)	11800	19.2	17.5	17
	20600	19.4	17.9	17.1
	20900	19.6	17.8	16.9
	19700	19.3	17.7	15.6
	20300	19.7	17.9	16.1
	19500	19.9	18.1	17.2
AVG	18800.00	19.52	17.82	16.65

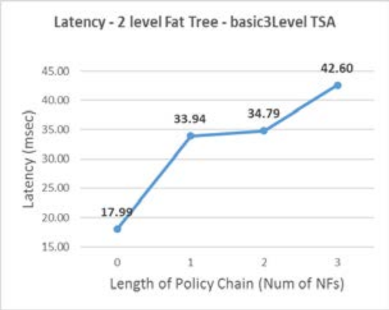


Comparison between the performance of the basic3Level TSA and the advanced TSA:



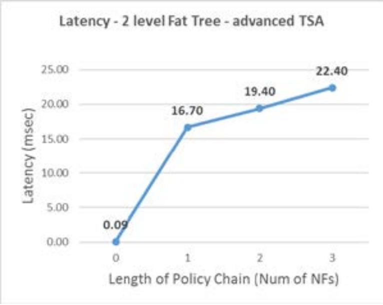
Latency analysis for basic3Level TSA with 2 level Fat Tree (Appendix A):

2Level - basic3LevelTSA				
	Policy Chain length			
	0	1	2	3
Latency (msec)	17.99	33.94	34.79	42.60



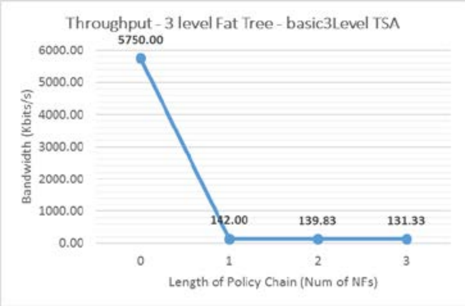
Latency analysis for advanced TSA with 2 level Fat Tree (Appendix A):

2Level - advancedTSA				
	Policy Chain length			
	0	1	2	3
Latency (msec)	0.09	16.70	19.40	22.40



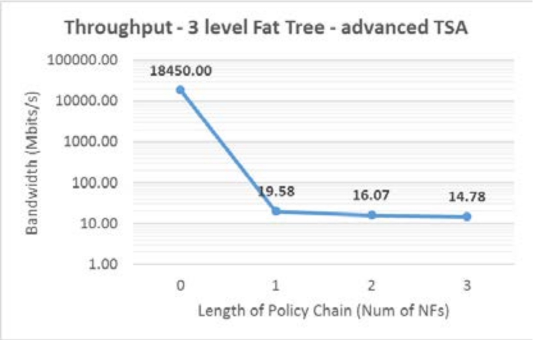
Throughput analysis for basic3Level TSA with 3 level Fat Tree (Appendix B):

	3 Level - basic3LevelTSA			
	Policy Chain length			
	0	1	2	3
Bandwidth (Kbits/s)	5900	186	185	188
	5900	122	170	160
	5700	98	182	108
	5500	108	113	112
	5700	160	84	137
	5800	178	105	83
AVG	5750.00	142.00	139.83	131.33

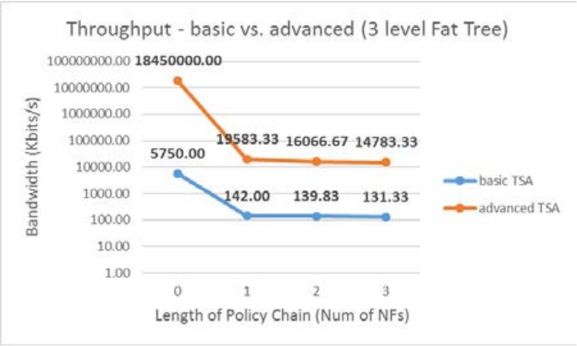


Throughput analysis for advanced TSA with 3 level Fat Tree (Appendix B):

	3 Level - advancedTSA			
	Policy Chain length			
	0	1	2	3
Bandwidth (Kbits/s)	20500	19.6	15.4	15
	11300	19.9	15.7	15.2
	20000	19	16.6	13.2
	20500	20.3	16.4	14.3
	19900	19.4	15.8	15.6
	18500	19.3	16.5	15.4
AVG	18450.00	19.58	16.07	14.78

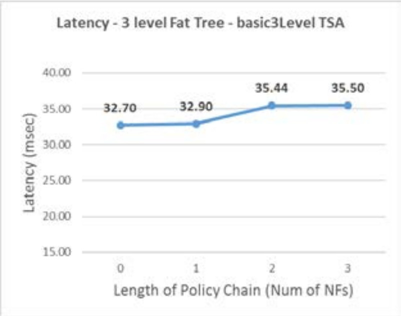


Comparison between the performance of the basic3Level TSA and the advanced TSA:



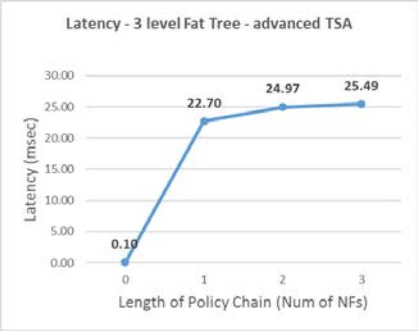
Latency analysis for basic3Level TSA with 3 level Fat Tree (Appendix B):

	3 Level - basic3LevelTSA			
	Policy Chain length			
	0	1	2	3
Latency (msec)	32.70	32.90	35.44	35.50



Latency analysis for advanced TSA with 3 level Fat Tree (Appendix B):

	3 Level - advancedTSA			
	Policy Chain length			
	0	1	2	3
Latency (msec)	0.10	22.70	24.97	25.49



4. DISCUSSION

As mentioned above, we performed the analysis twice. Once with a 2 level Fat Tree topology using the setup shown in Appendix A, and then with a 3 level Fat Tree topology using the setup shown in Appendix B.
In both experiments we had 3 NFs running under 2 ToR switches with 2 NFs sharing the same ToR. In both experiments the Firewall was at host 1, the Proxy was at host 7, and the IDS was at host 8. For simplicity, we actually ran 3 firewalls created using click, but only named them differently in the policyConfig file. We defined policies such that packets from h2 to h3 require no NF, packets from h2 to h4 require passing through Proxy, packets from h2 to h5 require passing through Proxy and Firewall (in that order), while packets from h2 to h6 require passing through all NFs in the following order, Proxy, Firewall, and IDS. We wanted all packets to finish at the same place in order to have a fair comparison as much as possible.
Recall that after the modification of the rules system, packets that don't require any policy are handled completely by rules, while packets that require some policy have only one reference to the controller which is at the end of the chain tour (i.e. after returning from the last NF). We couldn't improve any more than that for the reasons explained above.

We ran a small test using the default network topology and controller provided by mininet just to get an idea of the normal range of throughput reached by iperf. The results were pretty much high (around 20 Gbit/s). And when we first tested the basic3Level TSA, we were shocked, as we only achieved several Kbits/s, which was a big reason why we thought on improving our rules system in part 3 of the project.
There were still big differences than the normal bandwidths of the mininet, but we were happy with the huge leap we managed to make between the basic3Level and the advanced TSAs. You can clearly see the large impact our new rule system made on the TSA in the comparison graphs (the graphs with orange and blue lines). After the modification we noticed that the case where the policy chain length is zero (i.e. no NFs) approached the numbers of the default mininet configuration, which was great. The impact of the NFs was clear, the fact that packets had to travel some way in order to pass through the NF, in addition to the processing time within the NF itself affected the throughput of the network heavily. But we were able to rise the levels up to 14-20 Mbits/s in the advanced TSA in both the 2 and the 3 level Fat Trees, compared to the 100-180 Kbits/s in the basic3Level TSA. Overall, as we expected to see, **the longer the policy chain is, the lower the bandwidth rate.** We can easily notice the exponential decrease in the Throughput graphs.

Same goes for the Latency. We did a small test using ping on the default mininet network topology and controller, and the average time was 0.08 msec. Again, we were not satisfied with the large numbers we got from the basic3Level TSA. After we upgraded our TSA to the advanced one and repeated the measurements, we got a decent result for the empty policy chain case (no NFs), which was an average of 0.09 msec taken over 20 pings. However, the results for longer policy chains (i.e. 1, 2, and 3) were higher (around 16-25 msec), but they were still much lower (by ~10 msec) than the basic3Level TSA. Overall, as we expected to see, **the longer the policy chain is, the higher the latency (the more time it takes – higher RTT).** We can easily notice the increase in the Latency graphs.

In conclusion, the most important thing that we learned was how expensive it is to refer to the controller and how much it affects the performance of the network. In addition, we discovered that it is not always an easy task to achieve the maximum possible link utilization. We also saw that small changes can have a big impact sometimes, which was the case when we updated our rules system in the advanced TSA. Last but not least, we learned that Flooding packets in a Clos network (Fat Tree) kills you, as it completely misses the point of load balancing and providing a decent performance and link utilization levels.

5. USER MANUAL

First, let us provide a short list of all attached files, and briefly explain each one of them.

Attached Files:
o basic2LevelTSA.py – the 2 level supporting TSA
o fatTreeTopology2Level.py – 2 level Fat Tree topology used
o MakefileBasic2Level – makefile for running the basic2Level
o basic3LevelTSA.py – the 3 level supporting TSA
o fatTreeTopology3Level.py – 3 level Fat Tree topology used
o MakefileBasic3Level – makefile for running the basic3Level
o advancedTSA.py – the improved 3 level supporting TSA
o MakefileAdvanced – makefile for running the advanced TSA
o Makefile – makefile for transferring all the files to Mininet
o utils.py – Helper file. Used by all TSAs
o parser.py – Helper file. Used by all TSAs
o helper.py – Helper file. Used by all TSAs
o policyConfig – An example of the NFs' policies file
o mobility.py – Used for testing Moving hosts and NFs (Based on mobility.py from Mininet GitHub - https://github.com/mininet/mininet/blob/master/examples/mobility.py)

Placing the files in the mininet directories:
• All TSAs (i.e. Controller files – basic2LevelTSA.py, basic3LevelTSA.py, advancedTSA.py) should be placed in "~/pox/pox/samples/".
• The "utils.py" file should also be placed in "~/pox/pox/samples/".
• The topology files (fatTreeTopology2Level.py, fatTreeTopology3Level.py) should be placed in the home directory (i.e. "~" or "/home/mininet/"), but their names should be changed according to use (see below).
• The makefiles (MakefileBasic2Level, MakefileBasic3Level, MakefileAdvanced) should be placed in the home directory (i.e. "~" or "/home/mininet/"), but their names should be changed according to use (see below)
• The "mobility.py" file should also be placed in the home directory (i.e. "~" or "/home/mininet/").

Important Note (regarding the policies file):
Our TSAs assume that the policies file is always under "/home/mininet/" and is named "policyConfig". So please change the name of your policies file accordingly, or modify the constant "CONFIG_FILENAME" in the code, which is line 20 in all TSAs (basic2LevelTSA.py, basic3LevelTSA.py, advancedTSA.py)
The line is the following:
CONFIG_FILENAME = '/home/mininet/policyConfig'

Now, let us explain briefly how to run each TSA, and how to perform the small Moving hosts and NFs test that we built (using mobility.py).

basic2LevelTSA:

▪ Copy the TSA file “basic2LevelTSA.py” to “~/pox/pox/samples/” using scp.

▪ Copy the Topology file “fatTreeTopology2Level.py” to the home directory “~” using scp.

▪ Copy the “utils.py” file to “~/pox/pox/samples/” using scp.

▪ Copy the policies file **after renaming** it to “policyConfig” to “/home/mininet/”. Alternatively, you may change the constant in the code as explained above.

▪ If you use the Makefile then **rename** the Topology file to “fatTreeTopology.py”, and **rename** the Makefile “MakefileBasic2Level” to “Makefile”.

▪ Copy the Makefile “MakefileBasic2Level” **after renaming** it to “Makefile” to the home directory “~” using scp.

▪ Connect to the mininet (using “ssh”) from two terminals.

▪ From one terminal, run the controller (TSA) by simply typing “make”. Alternatively, you can use the full classic command if you don’t want to use the Makefile.

▪ From the other terminal, start the Fat Tree Topology by typing “make topo”. Alternatively, you can use the full classic command if you don’t want to use the Makefile.

basic3LevelTSA:

▪ Copy the TSA file “basic3LevelTSA.py” to “~/pox/pox/samples/” using scp.

▪ Copy the Topology file “fatTreeTopology3Level.py” or “fatTreeTopology2Level.py” to the home directory “~” using scp.

▪ Copy the “utils.py” file to “~/pox/pox/samples/” using scp.

▪ Copy the policies file **after renaming** it to “policyConfig” to “/home/mininet/”. Alternatively, you may change the constant in the code as explained above.

▪ If you use the Makefile then **rename** the Topology file to “fatTreeTopology.py”, and **rename** the Makefile “MakefileBasic3Level” to “Makefile”.

▪ Copy the Makefile “MakefileBasic3Level” **after renaming** it to “Makefile” to the home directory “~” using scp.

▪ Connect to the mininet (using “ssh”) from two terminals.

▪ From one terminal, run the controller (TSA) by simply typing “make”. Alternatively, you can use the full classic command if you don’t want to use the Makefile.

▪ From the other terminal, start the Fat Tree Topology by typing “make topo”. Alternatively, you can use the full classic command if you don’t want to use the Makefile.

advancedTSA:

▪ Copy the TSA file “advancedTSA.py” to “~/pox/pox/samples/” using scp.

▪ Copy the Topology file “fatTreeTopology3Level.py” or “fatTreeTopology2Level.py” to the home directory “~” using scp.

▪ Copy the “utils.py” file to “~/pox/pox/samples/” using scp.

▪ Copy the policies file **after renaming** it to “policyConfig” to “/home/mininet/”. Alternatively, you may change the constant in the code as explained above.

▪ If you use the Makefile then **rename** the Topology file to “fatTreeTopology.py”, and **rename** the Makefile “MakefileAdvanced” to “Makefile”.

▪ Copy the Makefile “MakefileAdvanced” **after renaming** it to “Makefile” to the home directory “~” using scp.

▪ Connect to the mininet (using “ssh”) from two terminals.

▪ From one terminal, run the controller (TSA) by simply typing “make”. Alternatively, you can use the full classic command if you don’t want to use the Makefile.

▪ From the other terminal, start the Fat Tree Topology by typing “make topo”. Alternatively, you can use the full classic command if you don’t want to use the Makefile.

Moving Hosts and NFs test:

▪ Copy the TSA file “advancedTSA.py” to “~/pox/pox/samples/” using scp.

▪ Copy the Topology file “fatTreeTopology3Level.py” or “fatTreeTopology2Level.py” **after renaming** it to “fatTreeTopology.py” to the home directory “~” using scp. Notice that here you need to rename the topology file even if you are not using the Makefile.

▪ Copy the “utils.py” file to “~/pox/pox/samples/” using scp.

▪ Copy the “mobility.py” file to the home directory “~” using scp.

▪ Copy **an empty** policies file (or a file **containing no rules**) after renaming it to “policyConfig” to “/home/mininet/”. Alternatively, you change the constant in the code as explained above.
The reason for requesting a policies file without rules is due to the mechanism the we use in starting the network and conducting the automatic Mobility tests. We weren’t able to run the NFs using the script “mobility.py”, so in order to use this test that we provided please make sure you provide a policies file without rules.

▪ If you use the Makefile then **rename** the Makefile “MakefileAdvanced” to “Makefile”.

▪ Copy the Makefile “MakefileAdvanced” after renaming it to “Makefile” to the home directory “~” using scp.

▪ Connect to the mininet (using “ssh”) from two terminals.

▪ From one terminal, run the controller (TSA) by simply typing “make”. Alternatively, you can use the full classic command if you don’t want to use the Makefile.

▪ From the other terminal, type “sudo python ./mobility.py” and wait for the test to end.

The test starts the network topology, and connects to the controller running in the second terminal. Then it excutes “pingall” to test connectivity. Next, it moves host “h1” from switch “s300” to “s301”, and then to “s302”. At each time it moves “h1” it again executes “pingall” to make sure connectivity is maintained. You can notice that the movement of the host is detected by seeing the printings in the controller terminal.

6. REFERENCES

[1] CS5413 – High Performance Systems and Networking, Cornell University (2014, September 22)

[2] CS6452 – Datacenter Networks and Services, Cornell University (2012, April)

[3] Konstantin S.Solnushkin: Fat Tree Design (2013, April)

[4] Ivan Pepelnjak: Data Center Fabrics – What Really Matters, ProIdea, ipSpace.net, NIL Data (2012)

[5] Cafarella M. & Ann Arbor: Datacenters EECS485, University of Michigan (2013, April 20)

[6] Ankita Mahajan: A Scalable, Commodity Data Center Network Architecture (2013, September 22)

[7] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat: A Scalable, Commodity Data Center Network Architecture – University of California, San Diego

[8] VL2: A Scalable and Flexible Data Center Network – Microsoft Research

[9] Zafar Qazi, Rui Miao, Cheng Tu, Vyas Sekar, Luis Chiang, and Minlan Yu: SIMPLE, SIGCOMM 2013

[10] Clos Networks, Wikipedia (https://en.wikipedia.org/wiki/Clos_network)

[11] Fat Tree Topology, Wikipedia (https://en.wikipedia.org/wiki/Fat_tree)

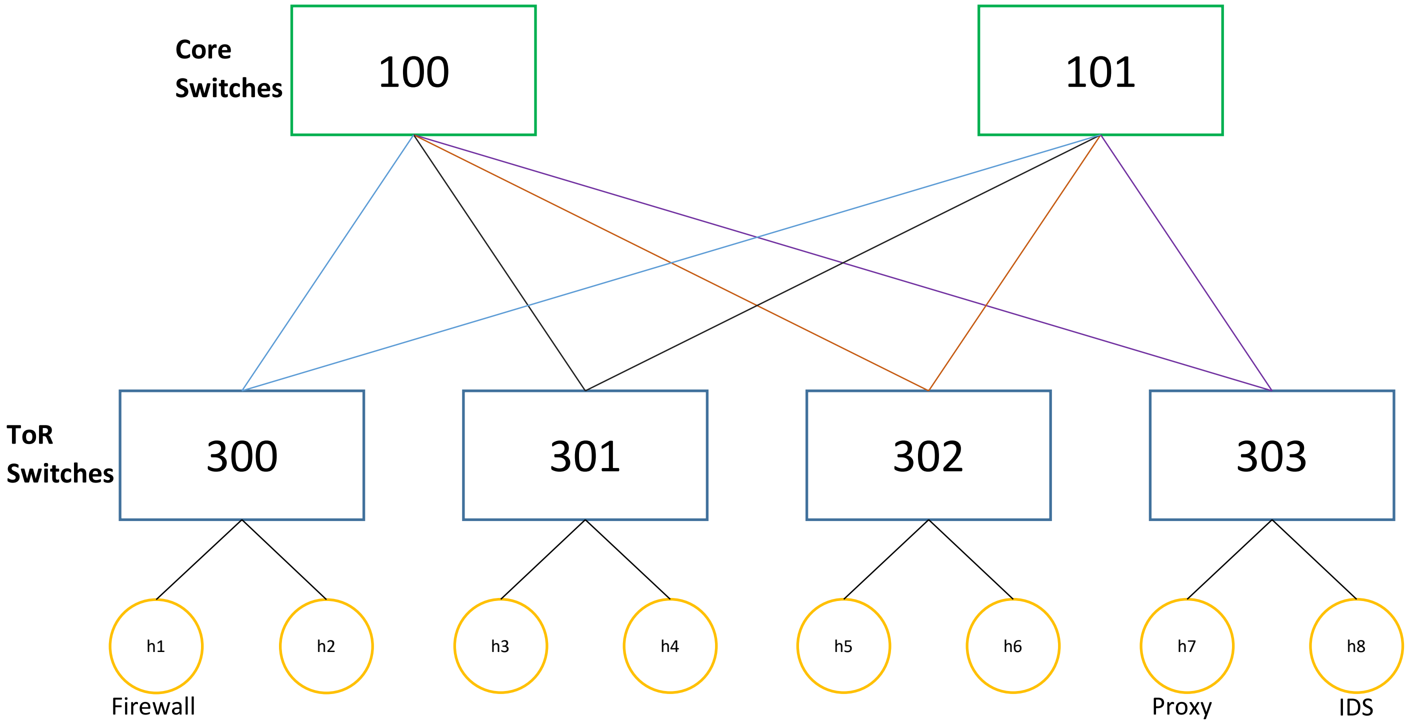
[12] Mininet – Walkthrough (http://mininet.org/walkthrough/)

[13] POX wiki, Openflow at Stanford (https://openflow.stanford.edu/display/ONL/POX+Wiki)

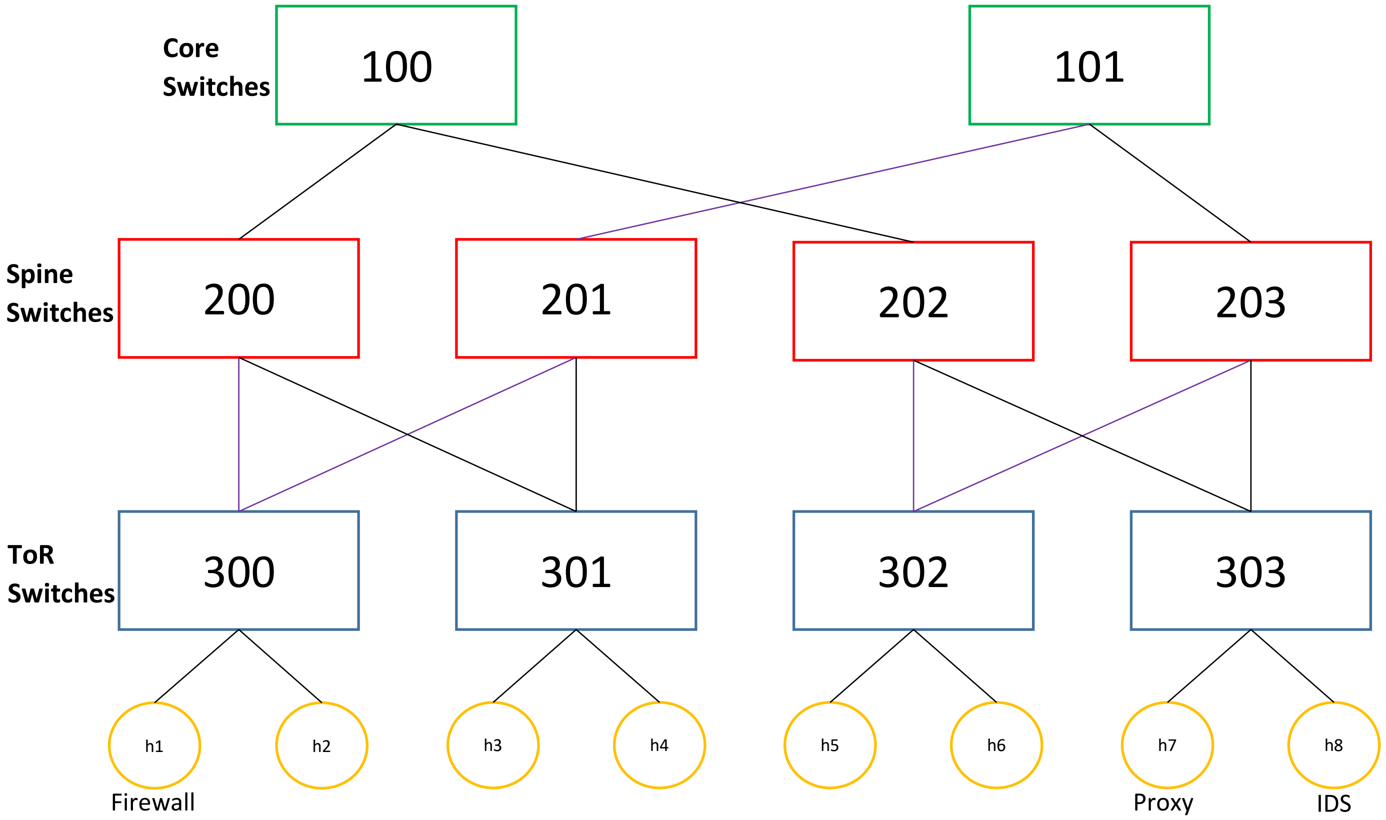
[14] The POX Controller, GitHub (https://github.com/noxrepo/pox)

[15] Click Modular Router (http://read.cs.ucla.edu/click/click)

Appendix A – 2 Level Fat Tree Network



Appendix B – 3 Level Fat Tree Network


Core Switches
100
101
Spine Switches
200
201
202
203
ToR Switches
300
301
302
303
h1
h2
h3
h4
h5
h6
h7
h8
Firewall
Proxy
IDS