INFO8011: INFO8011: Network Infrastructures Software-Defined Networking

Team 12: Maxime AERTS, Vincent JACQUOT

***Abstract*— Today, the prohibitive cost of fat tree based data centers led profits to another family of topology, the clos network. Such topology, relies on the deployment of commodity switches and redundancy of links. However, such topology requires more developed policies to handle the traffic. This report aims to present a few of these ones.**

I. Introduction

During the project, we experimented different algorithms and techniques to handle the traffic in a software-defined network with a Clos topology. This report presents 3 solutions to configure dynamically the network from the control plane: a simple spanning tree, a VLANs tree and an adaptive routing. We implemented and tested these solutions assuming an initial knowledge of the network such as the ids of the core switches or the port numbers forwarding to these core switches.

II. General testing methodology

Considering a network of m core switches fully connected to n edge switches, the number of links is thus m*n. Thus the network can support in the best case $\frac{m*n}{2}$ clients exchanging data at full bandwidth with $\frac{m*n}{2}$ servers. For each policy, a test running such configuration of clients/servers is run. Each pod contains m hosts, half of them are clients and each of them generates 5 flows of 2 Mbps to a server. This aims at checking the capacity of every policy to use fully the network.

On this idealistic case, variations are provided. Such as increasing the number of hosts and the bandwidth they require to check the fairness and the possible loss of performance when there are concurrent.

Finally, the size of the flows are also modified to check the impact of them on the performance.

III. Simple tree

A. General overview and implementation

The first proposed policy is to deploy a spanning tree over the topology to achieve connectivity and avoid cycles. However, such solution is sub-optimal since many of ports are blocked to avoid loops and thus some links are unused.

The algorithm to compute this minimal spanning tree is the reverse-delete algorithm. [1] To sum up, the links are listed in a decreasing order of cost (in the case of a clos network, the link costs are identical) and iteratively remove the links that can be removed without affecting the connectivity of the graph. Anyway, we still have to ensure that no edge switch is used as a relay (see Fig. 1). This is solved by selecting a fully connected core switch and trying to block its links lastly. An example of such obtained tree is presented in Fig. 2. Note that the case presented in the first figure is still possible in the case where no core switch is fully connected to every edge switch. Performance will degrade but the connectivity is still provided.

Finally, the switches' behavior is exactly the same as regular switches. It forwards packets when it knows the mapping or floods in the other case.

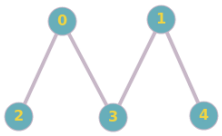

Fig. 1

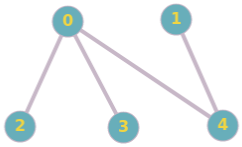

Fig. 2

B. Topology live changes

The controller is configured such it detects the changes of topology. When a link is added or removed, the whole spanning tree is recomputed. In addition, the mappings between the MAC addresses and the port for the switches are reset. To achieve recovery, another fundamental concept is the hard timeout added in the flows.

Of course, more efficient solutions could be designed. For example, we could avoid to compute again the spanning tree if the deleted link is one between two blocked ports. The hard timeout also induces a delay in the recovery since it may need some times to drop the flows (by default 10 s.). Another solution could be to drop the flows that are no more in accordance with the new tree.

C. Tests and results

We tested the simple tree several times using the `clos-topo/test.py` default topology, that is 2 core switches connected to 3 edge switches in a full mesh:

TABLE I

	Total bandwidth measured		Theoretical limit
Link bandwidth	Mean	Standard deviation	
10 Mbps	9.17 Mbps	0.43 Mbps	30 Mbps
5 Mbps	5.51 Mbps	0.28 Mbps	15 Mbps

The results shown in table I are very close to a network with only one core switch and this is not a surprise as the algorithm avoids one of the two core switches to eliminate broadcast storm risks. Therefore, the simple tree method does not use all the resources available and delivers poor results.

Note that the theoretical limit is the number of links between edge and core switches divided by 2 as a flow between two edge switches take at least 2 links.

IV. VLANs tree

A. General overview and implementation

This implementation relies on what we call rooted trees. A list of core switches that are fully connected to the edge is generated. The different VLANS in "tenants.py" and a default VLAN are distributed over these core switches. The default VLAN is used by the hosts whose address is not in "tenants.py". The rooted trees are not spanning tress in the sense that there do not provide full connectivity. In a rooted tree, only a core switch has its ports open. The other ones are not connected to the edge.

The switches are configured like in the previous algo. They forward in the destination is known and flood if not. But the flooding depends on the VLAN to which the packet belongs and the rooted tree assigned to this VLAN.

B. Topology live changes

The same method is applied here than in the previous section. When links events are triggered, the list of fully connected core switches and the rooted trees are generated again. The remarks from the previous section are the same.

C. Tests and results

We tested the VLANs tree several times using the `clos-topo/test.py` default topology, that is 2 core switches connected to 3 edge switches in a full mesh:

TABLE II

	Total bandwidth measured		Theoretical limit
Link bandwidth	Mean	Standard deviation	
10 Mbps	16.26 Mbps	1.45 Mbps	30 Mbps
5 Mbps	11.21 Mbps	0.68 Mbps	15 Mbps

The table II shows very good results in regards of theoretical limits. This is explained by a relatively good homogeneity of the traffic between the hosts, a case where VLANs tree is very great at.

V. Adaptive routing

A. General overview and implementation

The adaptive routing algorithm tries to balance the traffic load equally between the core switches. This task is handled by the edge switch controllers which, on each new flow, choose the "best" core switch for forwarding the flow. In order to avoid broadcast storm, an edge switch cannot send a packet coming from a core switch to a core switch. This seems simple and efficient, what it is, but is implies some drawbacks in case of a link failure that we will show in the next sub-section.

In term of traffic balancing, the controller collects network statistics periodically at a given interval and make a decision based on bytes transmitted on each link within the interval. Some optimizations can be done by selecting different measurements or different intervals as it is shown in the tests and results sub-section.

B. Topology live changes

Like the others algorithms, the controller reacts to link events and take the current topology into account when it defines a new flow. However, the only possibility for modifying existing flows is waiting for a hard timeout which can be very costly.

C. Tests and results

We tested the adaptive routing several times using the `clos-topo/test.py` default topology, that is 2 core switches connected to 3 edge switches in a full mesh. The interval value is set to 0.1 seconds:

TABLE III

	Total bandwidth measured		Theoretical limit
Link bandwidth	Mean	Standard deviation	
10 Mbps	15.95 Mbps	1.73 Mbps	30 Mbps
5 Mbps	9.01 Mbps	0.92 Mbps	15 Mbps

The results in table III are interesting as we expected better performances than VLANs tree. This bad surprise can be explained by a lot more computation happening in the controller which can increase the response delay and slow down the network traffic.

VI. Possible improvements

A lot of improvements could have been done with more time but one which comes first in mind is sharing the redirection knowledge between core switches.

VII. Feedback

The main difficulty encountered was the documentation of pox. The POX wiki only provides few explanations and do not cover everything. Furthermore, since the code itself is not documented, it can be very difficult sometimes to start with POX.

We spent approximately 20 hours each on this project. Most of this time was spent in developing new algorithms, trying to find a better way and start again. This project is not trivial, much could have been done if more time would have been allocated to it.

We have no suggestion to improve this project. The script of tests provided was very helpful and saves us time.

VIII. Conclusion

This project has allowed us to discover and understand software-defined networking in a more practical way than the theoretical lessons. Moreover, we learned that network measurement and management are not easy tasks and have to be taken seriously.

References

[1] J. B. Kruskal, “On the shortest spanning subtree of a graph and the traveling salesman problem,” *Proceedings of the American Mathematical Society*, vol. 1, p. 48–50, 1956.