INFO8011: Network Infrastructures

Software-Defined Networking in Data Centers

Maxime Goffart
180521

Olivier Joris
182113

Academic year 2021 - 2022

1 General overview

1.1 Spanning Tree Controller

This controller, as requested, builds a spanning tree in order to handle cycles in the network. To implement this solution, we first needed to discover the topology of the network. To do so, we used the functions `get_host`, `get_link`, and **get_switch** of Ryu.
Then, based on the discovered topology, we built a graph of the topology where the vertices are the switches and the edges are the links between the switches. Afterwards, to compute the minimal spanning tree, we used Prim's algorithm.

When packets are received at the controller, through `switch_in_handler` method, we check the source and the destination of each packet.
If the source is one of the host and the destination is broadcast, we broadcast the packet to the switches connected to the one that received it in the minimal spanning tree. Thus, the switches to which we sent the packet will send the packet to the controller that will, once again, do the same processing. At the same time, we set flows in the different switches so the controller will be contacted only once for a particular type of packets and it will reduce the RTT for similar requests.
If the source and the destination are both hosts, we compute a path between them using a depth-first search algorithm. Of course, the path is limited to links in the minimal spanning tree. When the path is computed, we set flows in the switches along the path so the controller will be contacted only once for a particular type of packets and it will reduce the RTT for similar request.

1.2 Using VLANs

The implementation of this controller is based on the one of the spanning tree: everything explained in the previous section is valid except that we are now separating the traffic into VLANs.

This controller is building VLANs by mapping each port of the switches to one or several VLANs. This repartition is represented in figure 1. It is composed of 4 VLANs and the main idea is that each of the four core switches are responsible of the traffic corresponding to one VLAN. Only hosts belonging to the same broadcast can communicate with each other and the broadcast domain is also limited to one VLAN.

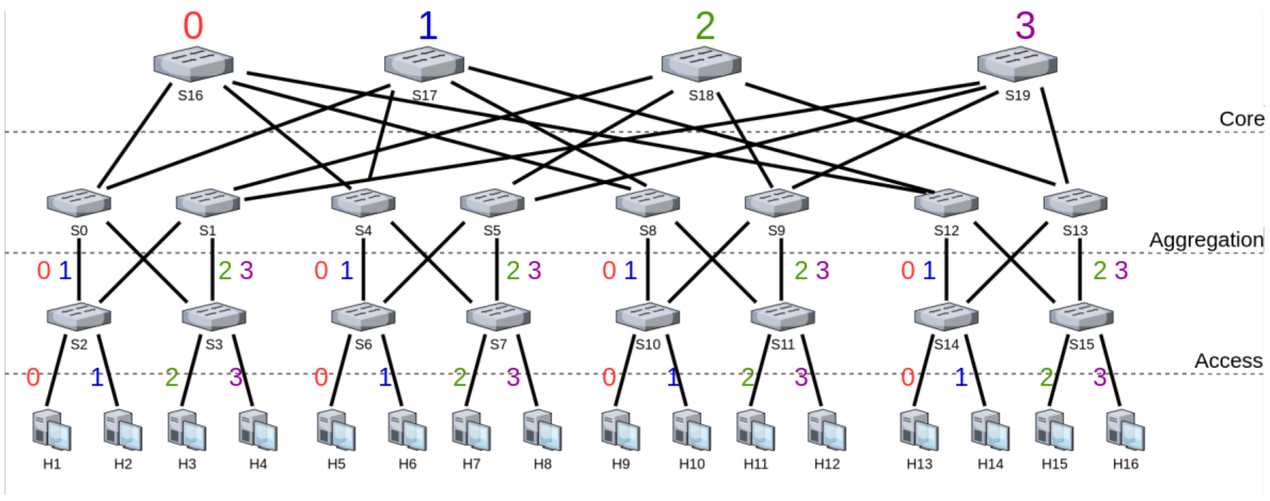


Figure 1: Map between links and VLANs.

2 Tests

2.1 Spanning Tree Controller

The simple commands used are available in the file `test_spanning_tree.sh`.

2.1.1 Tests in normal conditions

If we ping 10 times each host from `h0`, we obtain the results of table 1.

Source-Dest	Mean	MDEV	1st ping
H1-H2	272ms	201ms	877ms
H1-H3	466ms	124ms	840ms
H1-H4	460ms	124ms	830ms
H1-H5	695ms	192ms	1274ms
H1-H6	693ms	188ms	1258ms
H1-H7	691ms	208ms	1317ms
H1-H8	707ms	198ms	1300ms
H1-H9	715ms	210ms	1347ms
H1-H10	707ms	200ms	1306ms
H1-H11	703ms	200ms	1300ms
H1-H12	678ms	192ms	1256ms
H1-H13	705ms	181ms	1247ms
H1-H14	640ms	15ms	649ms
H1-H15	694ms	197ms	1287ms
H1-H16	701ms	200ms	1302ms

Table 1: Results of 10 pings between each host and `h0`

We can see that the first ping always take a lot of time. This is explained because the first ping we will have as consequence to send multiple packets to the controller. First, it will send packets to the controller related to the ARP request to learn the MAC address of the destination.

Then, it will send packets to the controller related to the icmp messages which will require the computation of paths between the host and the destination. All the processing in the controller is done in Python which is well known to be a slow language. Furthermore, these tests were performed on Maxime's computer which had the vm lock with an execution cap at 10% (see in section 3 the explanation).
Yet, the pings, after the first one of each test, have a RTT around the mean value. The high values of MDEV are explained by the fact that the first ping take a lot of time thus it increases the values of MDEV.
Another reason for the high value is the fact that each link has a delay of 50ms as set in the Mininet topology provided. Finally, we should keep in mind that we are running in a virtual environment and not on physical devices thus the measurements are not the most precise. Also, real physical links would not have a delay of 50ms (except for very long distance). tree[i][0] == None or tree[i][1] == None.

If we measure the bandwidth between `Source` and `Dest`, we obtain the results expressed in table 2. We can observe that, when the distance between the host is increasing, the bandwidth is decreasing which is logical because more switches have to be travelled in the tree. Moreover, if we perform mutliple `iperf` from multiple hosts at the same time, the bandwidth is increasing more.

Source-Dest	**Min**	**Max**
H1-H2	2.13 Mbits/sec	2.5 Mbits/sec
H1-H3	1.02 Mbits/sec	1.31 Mbits/sec
H1-H4	923 Kbits/sec	1.21 Mbits/sec
H1-H5	670 Kbits/sec	871 Kbits/sec
H1-H6	670 Kbits/sec	870 Kbits/sec
H1-H7	671 Kbits/sec	872 Kbits/sec
H1-H8	623 Kbits/sec	872 Kbits/sec
H1-H9	622 Kbits/sec	870 Kbits/sec
H1-H10	671 Kbits/sec	969 Kbits/sec
H1-H11	663 Kbits/sec	872 Kbits/sec
H1-H12	670 Kbits/sec	792 Kbits/sec
H1-H13	670 Kbits/sec	870 Kbits/sec
H1-H14	669 Kbits/sec	969 Kbits/sec
H1-H15	671 Kbits/sec	872 Kbits/sec
H1-H16	669 Kbits/sec	869 Kbits/sec

Table 2: Results of iperf between `Source` and `Dest`

2.1.2 Tests when a switch/link fails

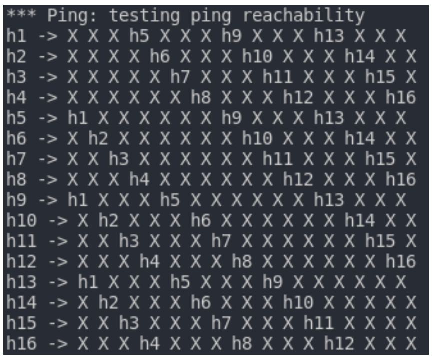
First, we can try to ping `h5` from `h1` and `h15` from `h1`. Both pings work as expected. Now, let us simulate a switch failure by typing `switch s17 stop` in Mininet. Then, ping `h5` from `h1` and `h15` from `h1`. Both work because our controller react to the change of topology. Now, let us simulate the switch going back up by typing `switch s17 start` in Mininet and do the same 2 pings as before. Both work because our controller react to the change of topology.
This simple test is available in the file `test_spanning_tree_failure.sh`.

2.2 Using VLANs

The scripts used to performed these tests are available in the `test_vlans.sh` shell script.

2.2.1 Reachability

The reachability can be verified using the `pingall` command which give the input represented on the 2. We observe that hosts can communicate each other only if they belong to the same VLAN.



```
*** Ping: testing ping reachability
h1 -> X X X h5 X X X h9 X X X h13 X X X
h2 -> X X X X h6 X X X h10 X X X h14 X X
h3 -> X X X X X h7 X X X h11 X X X h15 X
h4 -> X X X X X X h8 X X X h12 X X X h16
h5 -> h1 X X X X X X h9 X X X h13 X X X
h6 -> X h2 X X X X X X h10 X X X h14 X X
h7 -> X X h3 X X X X X X h11 X X X h15 X
h8 -> X X X h4 X X X X X X h12 X X X h16
h9 -> h1 X X X h5 X X X X X X h13 X X X
h10 -> X h2 X X X h6 X X X X X X h14 X X
h11 -> X X h3 X X X h7 X X X X X X h15 X
h12 -> X X X h4 X X X h8 X X X X X X h16
h13 -> h1 X X X h5 X X X h9 X X X X X X
h14 -> X h2 X X X h6 X X X h10 X X X X X
h15 -> X X h3 X X X h7 X X X h11 X X X X
h16 -> X X X h4 X X X h8 X X X h12 X X X
```

Figure 2: Reachability of hosts using VLANs.

2.2.2 Measures

If we ping 10 times from `Source` to `Dest`, we obtain the results of table 3. We observe that the VLANs are measures are quite similar for each VLAN which is normal because all VLANs are distributed with the same number of switches and hosts.

Source-Dest	**Mean**	**MDEV**	**1st ping**
H1-H5	671ms	210ms	1301ms
H1-H9	666ms	188ms	1229ms
H1-H13	664ms	188ms	1229ms
H2-H6	664ms	187ms	1225ms
H2-H10	662ms	184ms	1215ms
H2-H14	663ms	188ms	1229ms
H3-H7	668ms	200ms	1268ms
H3-H11	664ms	186ms	1223ms
H3-H15	664ms	187ms	1227ms
H4-H8	670ms	208ms	1294ms
H4-H12	662ms	184ms	1216ms
H4-H16	664ms	189ms	1232ms

Table 3: Results of 10 pings between `Source` and `Dest`

If we measure the bandwidth between `Source` and `Dest`, we obtain the results expressed in table 4. We observe that the VLANs are measures are quite similar for each VLAN which is normal because all VLANs are distributed with the same number of switches and hosts. These measures stay stable when combining `iperf` from different VLAN at the same time which is the principal interest to use this controller with respect to the spanning tree controller.

Source-Dest	**Min**	**Max**
H1-H5	621 Kbits/sec	791 Kbits/sec
H1-H9	670 Kbits/sec	872 Kbits/sec
H1-H13	669 Kbits/sec	872 Kbits/sec
H2-H6	670 Kbits/sec	871 Kbits/sec
H2-H10	670 Kbits/sec	872 Kbits/sec
H2-H14	667 Kbits/sec	873 Kbits/sec
H3-H7	667 Kbits/sec	872 Kbits/sec
H3-H11	670 Kbits/sec	871 Kbits/sec
H3-H15	721 Kbits/sec	1.03 Mbits/sec
H4-H8	671 Kbits/sec	872 Kbits/sec
H4-H12	663 Kbits/sec	871 Kbits/sec
H4-H16	621 Kbits/sec	870 Kbits/sec

Table 4: Results of iperf between `Source` and `Dest`

3 Feedback

3.1 Main difficulties

We encountered multiple difficulties when working on this assignment. Here is a list of the difficulties we encountered:

- Instabilities of the virtual machine, Mininet, and Ryu. For instance, Maxime had to cap the execution of the processor for the virtual machine at 10% or the code would not work. He lost some time trying to understand the issue.
- Most documentations, even the official book, on how to use Ryu is for OpenFlow 1.3 while we were blocked to OpenFlow 1.0. The differences between the 2 versions are minors but they are resulting in lose time that could be used to improve our understanding of SDN. Also regarding documentations, they are not always very well done.

These difficulties have the consequence that we have the impression that we spent more time on details related to Ryu and different versions of OpenFlow than really improving our understanding of SDN. Furthermore, these difficulties increased the difficulty of the project uselessly.

3.2 Time spent on the project

To be filled

3.3 Possible improvments

Here is a list of possible improvements for the project:

- Switching to OpenFlow 1.3 because, as mentioned previously, most documentations on Ryu are using OpenFlow 1.3 and we will not lose time on details related to which version of OpenFlow we are using.
- Maybe using something else than Ryu because we had issues related to it (e.g. needing to cap the execution of the processor or some functions of Ryu would return anything).