Lab 2: *Renaissance*, a self-stabilizing control plane

General Instructions

In this lab and the next one, we will go more in-depth into SDN and we will investigate a self-stabilizing control plane. We are giving you two controllers: a *global* controller and a *local* controller. if you have not done it yet, please read the paper *Renaissance: Self-Stabilizing Distributed SDN Control Plane* [2] present in pingpong.

The successful completion of the labs requires your provision of a report for each lab. All exercises in the instructions must be answered unless stated otherwise. Your answers should be concise; you don't need more than a few lines to answer each question. Questions that needs to be answered are within boxes. Some exercises ask you to discuss with your lab partner. You do not need to provide written answers to those questions.

You should complete the labs in groups of two persons — use the group you've created in pingpong! You are of course encouraged to discuss with other groups, but all your submissions must be the results of your own work. Once finished, upload your solution as a PDF document to pingpong, and don't forget to identify both members of the group.

Additional documents, such as source code, are available on pingpong.

It is assumed that you run the labs in the windows environment at Chalmers. We use a virtual machine and VirtualBox to have access to a linux environment. You may use your own computers, however we might not be able to provide support in that case.

In-band and Out-of-Band Controllers

Deploying a software-defined network can take two forms: with an *in-band* control plane, where both the control and data planes are using common physical links, or with an *out-of-band* control plane, where dedicated physical links are used to connect the switches and the controllers. The latter is more expensive, since an operator must now maintain two physical networks, but is usually a more reliable solution. In in-band deployments, the control traffic and the data traffic are competing for the bandwidth, and link failures can cause a loss of connectivity for the control plane, which cannot operate the necessary forwarding to support topology changes.

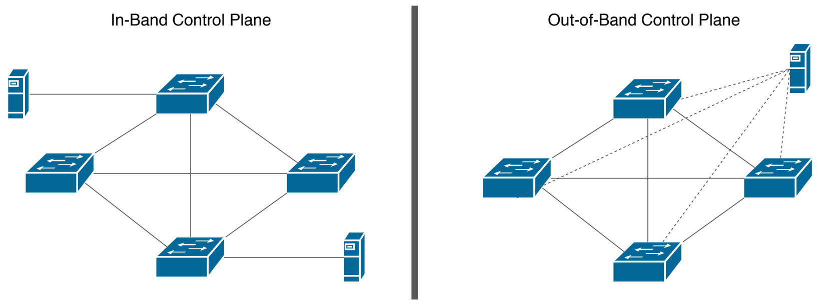

Figure 1: Architecture of Software Defined Networks (SDN)

In this lab, we will take a look at an in-band control plane, i.e. controllers are running on hosts within the network, unlike the precedent lab where the controller was running *outside* the network.

Renaissance, Local and Global Controllers

This time, you will have at your disposal a modified version of the Floodlight controller. The Renaissance algorithm has been (partially) implemented, and you will have to complete it. Your modification should be consistent with the algorithm, i.e. it should guarantee that every switch will be managed by at least one non-faulty controller and eventually that every non-faulty controller will reach every switch in the network, and this within a bounded delay of communication with a bounded number of failures.

We give below a high-level description of the algorithm. For more details, please see the paper in pingpong.

For simplicity, our implementation has split the task of discovering and computing paths in the network from actually installing them at the switches by introducing a local and a global controller. The global controller takes the role of making decisions of which rules should be installed and keeps track of all switches in the network, while the local controller translates the instructions that the switches receive from the global controller into instructions that they understand.

We will focus for now on the global controller. You can assume that the local controller implementation is correct, but feel free to check how it works.

Part 1 - Global Controller

In the floodlight_global/src/main/java/net/floodlightcontroller/globalcontroller/ directory, open GlobalController.java. You will find here the (partial) implementation of Renaissance.

Algorithm 1: Renaissance, high-level code description. [2]

```
1  Local state: $responses \subseteq \{m(j) : p_j \in P\}$ has the most recently received query replies;
2  $currTag$ and $prevTag$ are $p_i$'s current and previous synchronization round, respectively;
3  Interface: $myRules(G, j, tag)$: returns the rules of $p_i$ on switch $p_j$ given a topology $G$ on
     round $tag$;
4  do forever begin
5      Remove from $responses$ any reply from unreachable senders or not from round $prevTag$
         or $currTag$. Also, remove from $responses$ any response from $p_i$ and then add a record
         that includes the directly connected neighbors, $N_c(i)$;
6      if responses includes a reply (with tag currTag) from every node that is reachable
         according to the accumulated local topology, G, in responses then
7          Store $currTag$'s value in $prevTag$ and get a new and unique tag for $currTag$;
8      foreach switch $p_j \in P_S$ and $p_j$'s most recently received reply do
9          if this is the start of a new synchronization round then
10             Remove from $p_j$'s configuration any manager $p_k$ or rule of $p_k$ that was not
                 discovered to be reachable during round $prevTag$;
11         Add $p_i$ in $p_j$'s managers (if it is not already included) and replace $p_i$'s rules in $p_j$
             with $myRules(G, j, tag)$;
12     foreach $p_j \in P$ that is reachable from $p_i$ according to the most recently received replies in
         responses do
13         send to $p_j$ (with tag $currTag$) an update message (if $p_j \in P_S$ is a switch) and query
             $p_j$'s configuration;
14 upon query reply $m$ from $p_j$ begin
15     if there is no space in responses for storing $m$ then
16         perform a C-reset by including in $responses$ only the direct neighborhood, $N_c(i)$
17     if $m$'s tag equals to $currTag$ then include $m$ in $responses$ after removing the previous
         response from $p_j$;
18 upon arrival of a query (with a $syncTag$) from $p_j$ begin
19     send to $p_j$ a response that includes the local topology, $N_c(i)$, and $syncTag$
```

The goal is to familiarize yourself with the code. You should be able to understand and explain what is the main function of each section. Discuss them with your partner. You do not need to write your discussion in the report.

Attributes

Lines 70 to 94 contains the different attributes used throughout the algorithm, such as the tags associated to the iteration (here called *previousLabel* and *currentLabel*), the set of all nodes discovered so far (*discoveredNodes*), and the set of nodes to query for this iteration (*querySet*).

Let us go a bit further down in the code.

a. Main Task

```
public void startUp(FloodlightModuleContext context) throws
    FloodlightModuleException {
```

```
installRulesTask = new SingletonTask(ses, new Runnable() {
    @Override
    public void run() {
```

This is the main task of the algorithm. It is rescheduled and executed periodically. It corresponds to the **do forever** loop of Alg. 1.

b. Network Queries

```
private void queryNetwork () {
```

The **queryNetwork** method implements the **foreach** loop, line 12, of Alg. 1.

c. Topology

```
private void createTopology(Response r, List<SwitchNode> sNodes)
```

The **createTopology** method allows the controller to build the topology only with the directly connected neighbors of queried switches.

d. Paths

```
private String findPaths(SwitchNode dpid1, SwitchNode dpid2, List<
    SwitchNode> pathNodes)
```

As you can see, the implementation of this function is missing.

To keep the network operational, the global controller must compute the port used for forwarding packets from one switch to any other point of the network. *findPaths* construct a tree and do a Breadth-First Search (BFS) to obtain the path.

The function returns the port to use on the switch *dpid1* to contact the switch *dpid2*.

e. Receive

```
public net.floodlightcontroller.core.IListener.Command receive(IOFSwitch
    ofSwitch, OFMessage message, FloodlightContext context)
```

The controller executes *receive* upon reception of a packet. If the message can be decoded as a control packet, a switch statement calls the correct method. If the message is not for the control plane, e.g. a ping, it must be forwarded to the correct destination.

Part 2 - Running *Renaissance*

Now, you'll try to run the controllers in a network. Three different topologies are given in pingpong:

- B4, Google's wide-area SDN [3];
- Clos, a fat-tree data-center architecture [1]; and
- the backbone of Telstra, an Australian ISP [4].

To run a network, you must first start with the controller, like in the previous lab. Start the *local* controller. As you can see, it takes a few seconds for the controller to boot. Now, start the network with the command:

```
$ sudo python b4.py
```

The local controller should register all the new switches directly.

We now want to start the global controller *in-band*. To do so, you need to open a terminal on one of the hosts (xterm h1). Navigate to the location of the global controller, and start it.

The controller will now periodically query all the switches he is aware of. Because the implementation is incomplete, the controller cannot discover the entire network.

Question 1. Implement the *findPaths* method.
You need to use a Breadth-First Search (BFS) from the first switch *dpid1* towards the second node *dpid2* using the list *pathNodes*. Look at the implementation of SwitchNode to find the relationship between switches. The method *must* return the port of dpid1 that connects towards dpid2, followed by a "/" character.
For example, if the path from switch A to switch B passes through the port 1 of switch A, your implementation should return "1/".
To compile your code, use the *ant* command from the floodlight directory.
Question 2. Show that your implementation is correct by demonstrating a few paths on the B4 topology.
Note: The UI of floodlight has been deactivated. If you want, you can run the topology with the floodlight from lab 1 to see the topology. You can also look at the mininet script. To show that your path is correct, you can simply print the path once it is created, and show with the topology that it is indeed the shortest path.

Part 3 - Evaluating *Renaissance*

Run once more a network topology and both controllers. If your implementation is correct, your global controller should now be able to discover the entire network. With B4, you should have a querySet with 12 elements.

We will now evaluate *Renaissance*. We choose two performance metrics: the discovery time, i.e. the time required by a global controller to receive a reply from all nodes in the network, and the number of messages exchanged.

Question 3. Evaluate the discovery time distribution of *Renaissance* for three network topologies: B4, Clos, and Telstra. The average time is not enough. Use a violin plot or a box plot. The experiment must be repeated enough times for significant results. After each experiment, you need to completely stop both the network and the local controller.
Remember, if Mininet did not quit correctly, use sudo mn -c.
Question 4. Evaluate the number of messages exchanged for three network topologies: B4, Clos, and Telstra.
For an easier comparison between different topologies, divide the total number of messages by the number of nodes present in the network.
Again, use a violin plot.

Conclusion

In this lab, we studied *Renaissance*, a self-stabilizing distributed control plane. We looked at the relationship between the algorithm and its implementation, and tested it on three real-world topologies: B4, Clos, and Telstra. We then completed the implementation with a BFS-tree building algorithm. Finally, we evaluated the performance of our modified implementation.

In the next lab, we will go even further with Renaissance. We will implement backup paths for improved fault-tolerance, allow hosts to ping each other and use multiple global controllers concurrently.

References

[1] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*, SIGCOMM '08, pages 63–74, New York, NY, USA, 2008. ACM.

[2] Marco Canini, Iosif Salem, Liron Schiff, Elad Michael Schiller, and Stefan Schmid. Renaissance: Self-stabilizing distributed SDN control plane. *arXiv*, abs/1712.07697, 2017.

[3] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, et al. B4: Experience with a globally-deployed software defined wan. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 3–14. ACM, 2013.

[4] Neil Spring, Ratul Mahajan, David Wetherall, and Thomas Anderson. Measuring isp topologies with rocketfuel. *IEEE/ACM Trans. Netw.*, 12(1):2–16, February 2004.

Authors

This series of labs were created by Valentin Poirot <poirotv@chalmers.se> and Emelie Ekenstedt <emeeke@student.chalmers.se>.