Lab 1: Introduction to SDN

General Instructions

In this series of labs, we will look at more advanced concepts in computer networks. We will focus on Software Defined Networks (SDN), a new paradigm in network management used nowadays in many large-scale deployments.

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.

If you haven't done it yet, please watch the videos on SDN by David Mahler. A link is given on pingpong.

Software Defined Networks

In conventional networks, the control and management are done locally by dedicated equipment, such as routers and switches. Special algorithms, rules sets and specific hardware (ASICs) are used to perform operations like packet routing and flow forwarding. Configuration is device-centric, and topology changes, require human intervention or long transition phases to come back to normal performance. The lack of flexibility can especially hinder performances in high traffic networks such as Internet Service Providers backbones and within datacenters.

In SDN, the control and data planes are decoupled. The forwarding logic can now be *programmed* and is not integrated into the hardware as before. The control plane can therefore fully observe the system and sends commands to the elements of the data plane to configure them.

The control plane is composed of one or more *controllers*. A controller centrally defines the behavior of the network, by computing forwarding rules on request. Switches compose most of the elements of the data plane (along with routers) and are called *learning switches*.

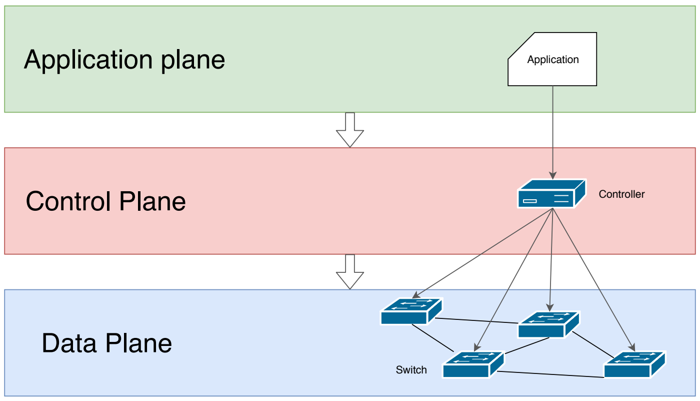


Figure 1: Architecture of Software Defined Networks (SDN)

Setup

We will experiment with Software Defined Networks using Mininet[1]. Mininet is a network emulator that can emulate switches, controllers and execute application code. It is available for many Linux distributions. If you are already using Linux, you can install it directly, although we recommend you to use a virtual machine (VM). Mininet makes updates to your network interfaces to work and might affect your system.

We will use VirtualBox[2] to run our virtual machine. VirtualBox is available for Windows, Mac OS, and Linux. You can use another hypervisor of your choice, but be aware that we will only support you for technical problems if you use VirtualBox.

A virtual machine has been prepared for you. Download it on pingpong or at `http://www.cse.chalmers.se/~poirotv/files/EDA387_sdn.ova.zip`.

Start VirtualBox and use File → Import Appliance. Import the VM we gave you. It might take a few minutes. Once it is done, you can start the VM. The credentials are given in the following table:

User	*Password*
sdn	sdn

Table 1: Virtual Machine Credentials

Please refer to the appendix if you want to create your own virtual machine.

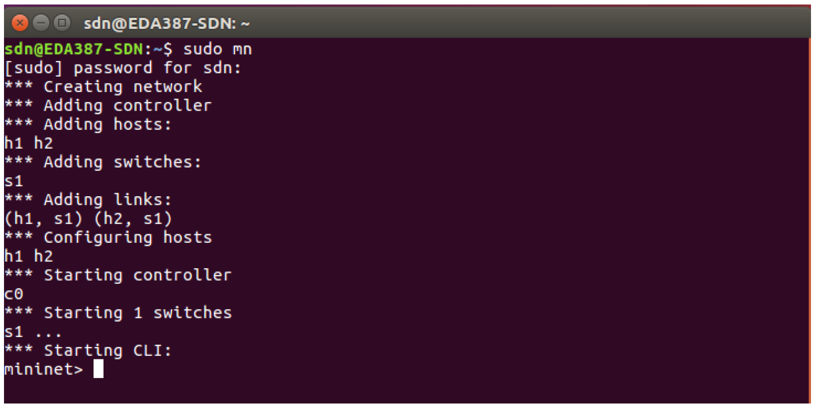
Testing your environment

You should now have a working environment. You can try to start a very simple network with the command:

[1]`http://mininet.org/`
[2]`https://www.virtualbox.org/wiki/Downloads`

```
$ sudo mn
```

If everything is working, you should see something like this:



```
sdn@EDA387-SDN: ~
sdn@EDA387-SDN:~$ sudo mn
[sudo] password for sdn:
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2
*** Adding switches:
s1
*** Adding links:
(h1, s1) (h2, s1)
*** Configuring hosts
h1 h2
*** Starting controller
c0
*** Starting 1 switches
s1 ...
*** Starting CLI:
mininet>
```

Figure 2: Starting Mininet

You can now interact with the newly created network using the Command Line Interface (CLI). By default, Mininet creates a network composed of two host machines, *h1* and *h2*, and a switch in between, *s1*. A native controller *c0* is also present. You will now try the connectivity between the two hosts. Try the command *pingall*. You should see that no packets are dropped. You can also try *iperf* to see the available bandwidth. Now, ping from one specific host by using *h1 ping h2*. Finally, you can execute any bash commands on a specific host by using *xterm h1*, where you replace *h1* by the name of the host. Once the new terminal is created, check the network interfaces with *ifconfig*. Once you are done, use the command *exit* in the mininet CLI to stop the network.

Important!
Later on, Mininet might crash if you have errors in your scripts. Mininet won't be able to start anymore if that's the case. To correct this behavior, use:
$ sudo mn -c
The -c option will clean Mininet's internal files.

Part 1 - Network Discovery

Your environment is now ready and you should know how to interact with mininet. In this first part, we will take a closer look at how OpenFlow creates the forwarding rule.

From a terminal, go to the floodlight directory (in the VM we gave you, floodlight is located in ~/labs/lab1/) and starts your controller with the command:

```
$ java -jar target/floodlight.jar
```

It takes a few seconds to initialize the controller. Once it is ready (you should see *Sending LLDP packets out of all the enabled ports* printed regularly), open a second terminal and start the custom network topology we gave you, with the command:

```
$ sudo python topo_lab1.py
```

If everything is working correctly, you should see that the controller registered a switch.

Question 1. In a conventional network, how can a switch know where to forward an incoming packet? Please explain briefly the mechanism.
Question 2. Ping from one host to another. You should see that at the very beginning, the RTT is significantly longer. Why? Stop pinging and wait for about 30 seconds, and try again. Is the RTT now stable? Why?
Question 3. Plot the Round Trip Time (RTT) evolution over time. You need to produce a plot with the runtime as the x-axis and the RTT time as the y-axis. You need two curves, the RTT from h1 to h2, and the RTT from h2 to h1. One measurement is not enough! You will need to restart both the network and the controller many times and average over multiple measures.
Remember, if Mininet crashes or has any problem, use the command: **$ sudo mn -c**

Part 2 - Understanding OpenFlow

We saw just before that a switch must learn what action it should execute if it receives a new packet. We will now take a closer look at how OpenFlow behave in the presence of new flows.

Open a new terminal. Start Wireshark in superuser mode with the command sudo wireshark (at startup, a message error should appear, this is normal). Start capturing incoming packets on the loopback interface (lo). To only display the important packets, use *openflow_v4* as the display filter (don't forget to press enter to activate the filter). In the other terminals, start the controller and the mininet topology as before. You should see new traffic appearing in Wireshark.

Question 4. Explain the roles of the different packets you see. You can use `http://flowgrammable.org/sdn/openflow/message-layer/` to understand them. We are using Openflow 1.3. Do not describe all the packets, but only the most important ones! Typically at the very beginning, once the switch contacts the controller, and when a new packet is received.

We will now directly look into the memory of the switch. Open vSwitch, the software used to run virtual switches, offers a command to look into the details of OpenFlow, *ovs-ofctl*. As with Wireshark, you can see the communications between your switch and your controller. Run

```
$ sudo ovs−ofctl snoop s1 −O OpenFlow13
```

You can also just check the switch's state with

```
$ sudo ovs−ofctl show s1 −O OpenFlow13
```

Start pinging between the two hosts again. Now execute the command

```
$ sudo ovs−ofctl dump−flows s1 −O OpenFlow13
```

You should see three flows.

Question 5. Explain what each flow represents. Briefly explain what each field is used for.
Question 6. Imagine now that a web server is running on h1. h2 starts an HTTP request to h1. What will happen? Describe briefly the messages exchanged between the switch and the controller, and the flows that would appear in the dump-flows result. Note: you can try to emulate the behavior by opening a terminal on h2 and by using the *wget* command and *python -m SimpleHTTPServer*.

Part 3 - Floodlight's User Interface

We have seen how SDN work from the user perspective, and from the switch perspective. We will now investigate how the controller views the network.

Floodlight has two interfaces we can use: an API, and a user interface. While both the network and the controller are running, connect to `http://localhost:8080/ui/index.html`.

You should see a dashboard appearing, with the switches and the hosts *discovered* by the controller. Look at the topology and discuss with your partner the different elements. Go then to the Switches tab, and select the switch. Start pinging again. In the Flows table, do you see the same flows as before? Discuss with your partner.

Stop the network and the controller. We will now try a different topology. Start again the controller, and start a new network with the command:

```
$ sudo mn −−topo=tree,depth=3 −−controller=remote,ip=0.0.0.0,port=6653
```

Question 7. How many switches are present? What kind of topology is it? How many switches must be programmed to ping from h1 to h8?

We try again with a different topology, linear this time.

```
$ sudo mn −−topo=linear,n=1,k=7 −−controller=remote,ip=0.0.0.0,port=6653
```

Question 8. How does the first RTT evolve with the number of switches to cross?

Conclusion

In this lab, we looked at SDN as a new paradigm for network management. We introduced the concepts of controllers and learning switches, and how OpenFlow allows us to *program* our switches for flow forwarding. In the next lab, we will introduce you to the concept of self-stabilizing control plane, and how we can tolerate failures in SDN.

For the next lab, please read *Renaissance: Self-Stabilizing Distributed SDN Control Plane* (the paper is present on pingpong). We do not require you to fully understand all the details of the paper or to be able to do the proofs. But pay close attention to algorithm 1, you will work with an implementation of that system during the next session.

Appendix

Creating your own VM

On your favorite OS, use the command:

```
$ sudo apt-get install git ant wireshark
```

Git is a version control system, we will use it to download and install a specific version of Mininet. Ant is used to build Java applications, and Wireshark[3] is a packet analyzer that listens and records all communications in the network.

Once this is done, we will clone the Mininet repository from Github. Use the command:

```
$ git clone https://github.com/mininet/mininet.git mininet
```

This will create a directory and copy all the files. You then need to install Mininet. From the same location, execute the following command:

```
$ ./mininet/utils/install.sh -rmf -V 2.5.5 -3
```

By doing so, you are installing mininet and OpenFlow version 1.3.

Finally, we will use Floodlight[4] as our OpenFlow controller. It is open-source and written in Java. Go to `http://www.projectfloodlight.org/download/` and download the version **1.2**. To build the controller, just use

```
$ ant build
```

from within the floodlight repository.

Authors

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

[3]`https://www.wireshark.org/`
[4]`http://www.projectfloodlight.org/floodlight/`

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.

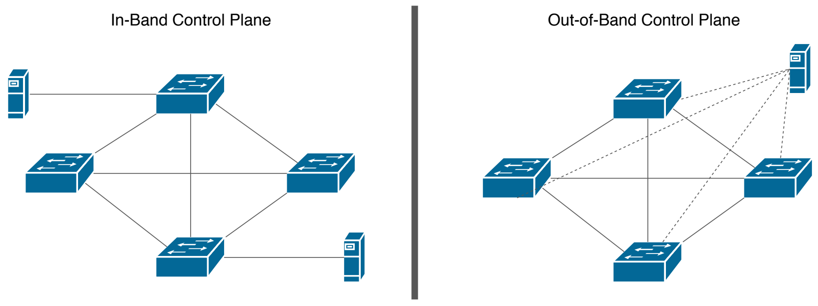


Figure 1: Architecture of Software Defined Networks (SDN)

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

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

```java
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

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

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

c. Topology

```java
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

```java
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

```java
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>.

Lab 3: Adding Fault Tolerance to *Renaissance*

General Instructions

In this lab, we will continue to work on *Renaissance*. This time, we will compute backup paths to add fault tolerance to the system.

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.

Backup Paths and Fault Tolerance

Our current implementation is not very resilient against node failures. If a node fails, the global controller must recompute the topology and install new rules on the switches. This causes downtime for the system, during which communication is disrupted.

The simplest way to resolve node failures and add fault tolerance is to use redundancy. If a failure happens along a path, a backup path can be used to forward traffic while the problem is taken care of. We now speak of primary and secondary paths.

We introduce the *findBackupPath* method:

```java
private String findBackupPath(SwitchNode dpid1, SwitchNode dpid2, List<
    SwitchNode> excludeNodes, List<SwitchNode> sNodes)
```

To be correct, the secondary path must not use the same output port as the primary path, i.e., avoid the next node along the path if it has failed. An *excludeNodes* list allows us to keep tracks of already used nodes (i.e., the output port of the source node).

Question 1. Implement the *findBackupPath* method.
You need to use a Breadth-First Search (BFS) from the first switch *dpid1* towards the second node *dpid2* using the list *pathNodes*. Your path cannot use any node present in *excludeNodes*.
Like for *findPaths* in lab 2, the method must return the port to use followed by a "/" character, e.g. "2/" if the switch dpid1 must use port 2 to contact dpid2 with the backup path.

We now need to add this information to the rules. Luckily for us, the local controller's implementation is already able to handle backup paths. All we have to do is to concatenate the result of *findBackupPath* to the String returned by *findPaths*.

Question 2. Update your *findPaths* method to include the result of *findBackupPath*. If the primary path uses port 1 and the secondary path uses port 4, your function must return "1/4/".
Do not forget to add the node used by your primary path in the *excludeNodes* list before computing the backup path!
Question 3. Show that your implementation is correct by demonstrating a few paths on the B4 topology.

Your implementation should now install both a primary path and a secondary, backup path from every switch to every other node in the network. The network should now be resilient to link failures. We will now demonstrate that backup paths are indeed used when the primary link is down.

In mininet, run the command:

```
link s1 s2 down
```

Check the B4 topology and choose one link to stop. If you want to stop more than one link, remember that the network must remain connected, i.e. you shouldn't cause network segmentation.

Now, look at the output of the global controller. You should see the current synchronization label. The label increases every time the global controller receives replies from its entire query set. If one node is not reachable, the label should not increase.

Question 4. Are all switches reachable by the global controller, even in the presence of link failures?

You should now be convinced that the global controller can still reach all nodes in the network, even if one link is down.

But the implementation is still limited to at most one failed link per switch. We can make it even more reliable by computing more backup path. We simply need to call *findBackupPath* multiple times, until there is no additional path.

Question 5. Update your *findBackupPath* to find all existing backup paths.
One way to do this is to recursively call the method and add the used ports to the *excludeNodes* list at each iteration. If no more paths are available, the method should return an empty string.
If switch A can contact switch B through the ports 1, 3 and 7, then your *findPaths* function should output “1/3/7/”.

Allowing Data Packets Through

To ensure self-stabilization, *Renaissance* considers all incoming packets as control packets by default. The content is then matched to a set of well-defined set of possible control messages. If no matches are found, the implementation discards the packet.

We will change this behavior to forward non-control packets. To simplify the problem, we will assume that all packets on the data plane are using IPv4. Even more, all packets are using ICMP, i.e. they are ping from one node to another.

Question 6. Upon reception of an ICMP message, forward the packet to its correct destination.
You need to update the *receive* method of the global controller.

You can take inspiration from the way other TCP messages are handled. ICMP messages can be tested if the protocol is equal to *IpProtocol.ICMP*.

To be able to test the ping, you need to run two global controllers in the network. Only a host with a running global controller can ping other hosts.

Make a copy of the global controller directory. Open *src/main/resources/floodlightdefault.properties*. Modify the line:

```
net.floodlightcontroller.globalcontroller.GlobalController.controllerIP =
    10.0.0.1
```

To the IP of the second host, e.g. 10.0.0.2.

Once both global controllers are running and they have discovered the entire network, you should be able to ping from one host to another.

Adding Global Controllers

We ran two global controllers in the same network to enable pinging. Remember that, to recover from any faulty configuration, a global controller must erase the memory of all the switches it contacts. One might wonder how two controllers can work in the same network without competing.

Question 7. What mechanism allows multiple controllers in the same network without competition?
Question 8. How does the algorithm deal with failed controllers? Check the meaning of manager in the implementation.

Conclusion

In this lab, we build upon our implementation of *Renaissance*. We added fault tolerance with backup paths and enabled pinging through the network.

More generally, this series of labs introduced you to the concepts of software defined networks. You used OpenFlow, the most frequently used protocol for switch and flow programming. You also interacted with Mininet, a powerful network emulator often used in research on SDN.

Finally, you had the chance to experiment with an implementation of a self-stabilizing algorithm. You saw how an implementation differs from pseudocode, how to evaluate a system and how self-stabilization can be implemented.

Authors

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