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:

```
1 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:

```
1 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:

```
58 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>.