NODE AND LINK FAILURE DETECTION

S&R PROJECT, ACADEMIC YEAR 2020-2021

Stefano Maxenti
stefano.maxenti@mail.polimi.it

Marco Bottenghi
marco.bottenghi@mail.polimi.it

Elisabetta Cainazzo
elisabetta.cainazzo@mail.polimi.it

REQUEST

We are asked to design a ring network and monitor the switches and links status using OpenFlow functions, by saving them every specific time interval; to implement two different approaches: the reactive one (looking at OpenFlow messages) and the proactive one (checking the switches/links status and react accordingly); to test the implementations with various traffic generators; to display the results; to verify our software on the physical SDN testbed in the Bonsai Lab.

INTRODUCTION

As requested, we implemented two possible versions of the RYU controller, one based on reactive behavior and the other one on proactive. Starting from the given template, we modified it accordingly to our needs, introducing new features particulary useful for the real-life scenario.

The original code was reformatted to make it more extensible: we created a specific function for path discovery and rules installation, because we needed to call it from other parts of the code (for example to restore the shortest path in case a link was brought back to life).

We also implemented the $buffer\_id$ option in the rules installation instead of relying on RYU to create a packet (it is more efficient to use the packet memorized in the switch instead of creating it again by composing the header and serializing the data).

A small modification to the ARP function was made as to overcome the problem of missing gratuitous ARPs from the host.

We also implemented an important function to solve a well-known problem in OpenFlow (fixed in version 1.4). We found out that rules installation is not atomic: in particular, it is possible (and it does happen a lot) that a packet reaches the second switch before the rule is installed and then it is sent back to the controller, triggering a new path discovery and subsequent rule installation. This creates a variety of problems linked to the fact that the association MAC-DPID becomes wrong, and all subsequent rules may not work because based on a wrong localization of hosts. This problem is well known in the literature; in OpenFlow 1.4, a specific message was implemented so that rules installation is atomic ($BundleMsg$). After some trials, including setting a timer between rules installation and packet sending, we found in the documentation a specific message type that could solve the problem. It is the $Barrier$ message, used to force the switch to suspend all activities until the rules are installed. By doing this, we avoid the synchronism problem in an elegant way, without introducing an artificial timer delay. The main disadvantage to this approach is the fact that internal switch optimization mechanisms may be bypassed, and rules installation becomes a bit slower, however this small downside is necessary to have a correct behavior and not to increase load on the controller itself.

REACTIVE APPROACH

The main idea is to listen for OpenFlow messages sent by the switches to the controller and react accordingly.

The OpenFlow environment sends messages any time there is an event; by using a decorator in the controller code ($@set\_ev\_cls$), the RYU controller is triggered.

The main trigger function is of course the one called when a link is disconnected (link_del_handler_reactive). This function creates a thread for each one of the failed link: we used this approach because, during the LAB testing, we found out that it is possible that the application halts when a switch is abruptly disconnected. Threads allows to parallelize the work and overcomes the waiting-for-an-answer problem.

PROACTIVE APPROACH

The main idea is to regularly check the link status to find out whether a link has been disconnected. Every t seconds (in our simulation, every 1 second, but this parameter can be adjusted to specific needs), get_list is used to find all links; we compare this list with the previously memorized one. If there are differences, we can easily discern whether a link (or even a switch) was removed. We create a thread for each link to be removed to avoid the halting of the system (as in the reactive scenario). The function that deals with the removal of the rules is almost the same as the reactive one (the main difference is the input parameters, a RYU event object in the reactive, a list in the proactive).

Another peculiarity is the contemporary installation of main rule and backup rule with lower priority (since we are in a ring, the shortest path, and the path in the other direction). This approach reduces the delay because a secondary path is already available when needed. We differentiate primary and backup both by using different priority and different cookie.

Another possibility would be to pre-install rules for every possible combination host-host. We didn't implement this because it would be very resource consuming (in the Mininet simulation we tested an 18-host and 18-switch scenario, and we would need to install before-hand $18 \cdot 17 \cdot 2 = 612$ rules). Installing, on request, primary and backup path seems to be a much better compromise.

LINK REMOVAL ALGORITHM

In the link removal function, we have the following mechanism of action:

1. We extract link information from the input parameter (RYU event in reactive, list in proactive) and we check if the link is present in the link list:

this is useful to not call the deleting function again when the opposite link's failure message is received (links are bidirectional and thus always two failure messages are generated). If this condition is not satisfied, the application returns.
2. If the link is present in the links list, we ask to the connected Datapaths their installed rules. By using this approach, we can avoid resetting all rules in the switch, but just the one that include the failed link.
3. We remove the link from the graph that represents the structure of the net.
4. We start to analyze the rules requested from the Datapaths: we get source IPv4, destination IPv4, and we send a delete request to all switches (but not the destination Datapath, to avoid a pending request in case the switch was abruptly removed) using those parameters.
5. After that, we verify if all switches are still available by calling a specific RYU function (*get_links*); in case we find that a switch is disappeared, we remove it from the switches list and from the net topology.

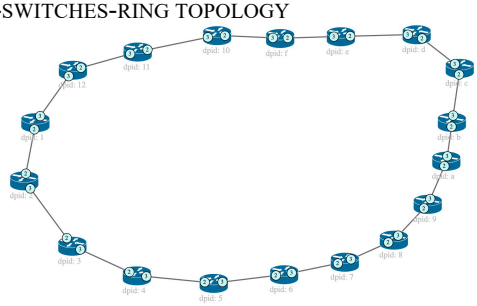
LINK ADDITION ALGORITHM

We started from the template but modified this function as well.
The main idea is to react to the message that announces that a link has been added. We simply update the topology in the reactive one, while on the proactive we implement the possibility to restore the shortest path for all rules. The separate function created for path calculation and rules installation comes in handy.

MININET RESULTS

As a baseline, we can consider a generic Ryu controller that does not react to link failure. In this case, there is no automatic way to recover from a problem. A very native approach would be to use the *idle_timeout* and/or *hard_timeout* parameter in flow installation, to define a period of time after which a rule is automatically removed, let it be for inactivity or by default. Reaction and restoration would take *idle_timeout* seconds if the flow is short-lived, *hard_timeout* if it is long-lived. Of course, it would be necessary to reset the topology every so often, otherwise even after the timeout the wrong path would be calculated.
Let's consider our implementations with the following Mininet topologies.

18-SWITCHES-RING TOPOLOGY



We use a big topology: 18 switches with one host each. This massive network is made up of 36 links. A *pingall* command shows perfect reachability between each one of the 306 combinations of hosts.

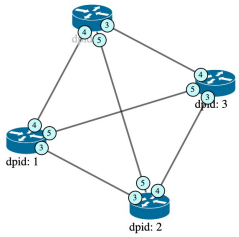
In the reactive implementation, flow installation is quite fast. Since we have already computed a lot of flows with the *pingall* command, the reaction time to failure is quite slow. If we test a simpler configuration (as the one in the lab), reaction time is infinitesimal. For this reason, we don't provide actual data.
The same happens in the proactive implementation. However, it is interesting to notice that flow installation is definitely slower since each flow has its own backup. On the other hand, even in the case of the *pingall* command, reaction to failure is fast and mostly depends on the timer. Restoring to the shortest path takes some time because the number of flows is large; if we consider simple scenarios, the restoring time is very litte.

BIGGER SWITCHES WITH MORE HOSTS EACH - EXTRA

By modifying the NUMBER_OF_SWITCH_PORTS, we can try if it works even in the case there are more than one hosts per switch. We will not try this situation in the testbed since Zodiac switches are limited in the number of ports.
This small extension works correctly from our trials.

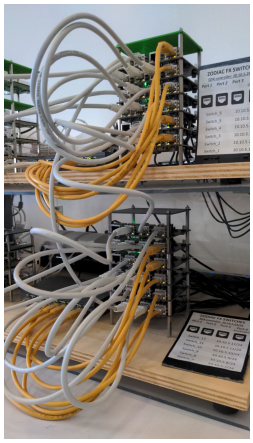
CROSSED TOPOLOGY - EXTRA

The template provided us with a generic function to find the shortest path. This means that our code is extensible to a crossed topology as the one below. As backup path we pick a random path. This could be optimized by selection of the K-shortest path.



REAL IMPLEMENTATION RESULTS

We instisted that our project work not only in Mininet, but also in the Bonsai Lab.



The task was not trivial and the final implementation surely needs more testing before being put into production: while link removal and addition worked

from the almost beginning, node failure was very tricky. In Mininet, there is no way to simulate the "death" of a link or a switch: the commands "link s1 s2 down" and "switch s1 stop" are gracefully way to remove a link or turn off a switch, always sending some messages to the controller that reacts accordingly.
In real life, pulling the electrical plug from a switch doesn't allow it to communicate to the controller that it is being shut down, and in the first version of our code this was a big problem, since the removal of a switch would halt the program because the controller didn't realize that the switch was off.
The RYU function *get_switches* provided to be useless, since this switch would still be considered alive until the TCP keepalive between switch and controller is over (and this could take a while).
For this reason, we don't use RYU functions (except *get_links*, since we found out that it behaves correctly even in real-life) but implemented our owns.
What we notice from the LAB is that everything is a bit slower: calculating the path and installing the rules takes some time, thus the first packet of the flow may be lost due to a timeout.
Of course, notification time is better in reactive mode, but reaction time (deleting the old rule) is faster in proactive since there is no need to calculate anything, thus the only downtime is the status timer (in our case 1 second).
For the reactive implementation, we get a mean down-time of about 1,3 seconds.
For the proactive implementation, we get a mean down-time of about 1,8 seconds. It is interesting to know that this is mostly due the frequency of the timer, and not flow installation, since we already have a backup available.
Iperf3 works correctly as can be seen in the video. We limited the speed to 5Mbps in the reactive video as to avoid to stress the controller. For the proactive approach, this was not needed since pre-installed flows reduce the number of PacketIn.
We also tried ITG. In the reactive implementation, we get around 7% packet loss in a 10-second interval (73 packets over 919). This corresponds to about 0,7 seconds of downtime.
In the proactive implementation, using the same parameters we get a 2% packet loss (21 packets over 971). This corresponds to just 0,2 seconds of downtime (as before, due to the already installed backup flow).

CONCLUSION AND FUTURE WORK

We tried to write an easily extensible code: we tested in Mininet that the topology is not relevant.
Creating small functions makes it easy to use a black-box approach that could be useful is other functions are to be implemented in the future.
We find out that both implementations have pros and cons and the choice between the twos must consider the specific traffic scenario.

Future work may be about the following points:

- The implementation in the lab still needs some testing to be completely reliable.
- Even though the deletion rule is quite optimized (we delete only the relevant rules as we don't nuke all switches), we send the message to delete the specific to all switches, even to the ones that surely don't have it. An even better algorithm could filter out the switches that don't contain it by traversing the graph. However, in a small network as ours, the advantage would be negligible.
- Using OpenFlow 1.4 or higher and the $BundleMsg$ option instead of Barrier would be faster and more optimized: however, not only Ryu has to be updated, but also the switches (and the Zodiac FX switches in the physical lab may not be compatible).
- A hybrid approach might be a viable solution, even though it duplicates function and increase the complexity of the solution.
- More powerful switches could be used: for example, it would be possible to use generic routers with the custom firmware OpenWRT that allows the use of OpenFlow, provided the package is correctly installed.