Computer Network Performance Project

Andrea Bernini - 2021867
Donato Francesco Pio Stanco - 2027523
Sapienza University of Rome
Master's Degree in Computer Science

***Abstract*—For the Computer Network Performance course we have developed the following project which consists in the creation of a custom Software Defined Networking (SDN). We used miniedit for the realization of the network and it allows to experiment with SDN and OpenFlow systems. The following report shows the description and analysis of four plots that show particular network cases such as the increase in network traffic, the simulation of a denial of service, a simulation of network redirection and recovery.**

The entire code developed by us can be found at the following GitHub repository: computer-network-performance-project.

***Index Terms*—SDN, OpenFlow, miniedit**

I. Introduction

Software-defined networking (SDN) technology is an approach to network management that enables dynamic, programmatically efficient network configuration in order to improve network performance and monitoring, making it more like cloud computing than traditional network management. SDN is meant to address the static architecture of traditional networks. SDN attempts to centralize network intelligence in one network component by disassociating the forwarding process of network packets (data plane) from the routing process (control plane). The control plane consists of one or more controllers, which are considered the brain of the SDN network where the whole intelligence is incorporated. The following diagram 1 depicts how, in case of switches, SDN will realize the separation of control plane into controller.

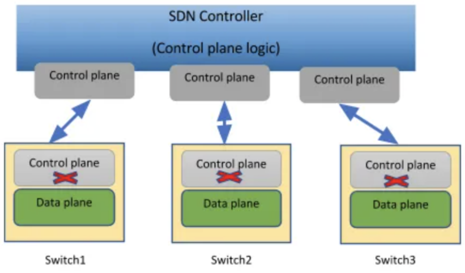


Fig. 1. SDN separation of control plane into controller

Control separation has many benefits like:

- **Central management**: You can configure, monitor, and troubleshoot the network and can also get a complete view of it (network topology) from the controller.
- **Light-weighted network equipments**: Network elements like switches and routers can be slimmed down, which in turn can help them becoming less expensive over the time. Intelligence would be at the controller where control plane (i.e. control logic) would reside, allowing control of underlying network elements by pushing rules over them through a common channel (i.e. protocols).
- **Network virtualization**: Virtualization of network leads to multi-tenancy (an architecture where-in a single software instance runs on a server and serves multiple tenants), which in turn helps leverage full potential of network elements. SDN controller can abstract underlying physical network and allow network administrators to program virtual networks corresponding to each tenant.

However, centralization has its own drawbacks when it comes to security, scalability and elasticity and this is the main issue of SDN.

SDN was commonly associated with the OpenFlow protocol (for remote communication with network plane elements for the purpose of determining the path of network packets across network switches) since the latter's emergence in 2011. However, since 2012, proprietary systems also used the term. These include Cisco Systems' Open Network Environment and Nicira's network virtualization platform.

II. Topology of the Network

We have been assigned the following network topology to build and analyze, which comprises:

- 3 hosts (*h1*, *h2*, *h3*)
- a malicious host (*hm*)
- 4 switches (*s1*, *s2*, *s3*, *s4*)

The following schema represents the topology that we implemented:

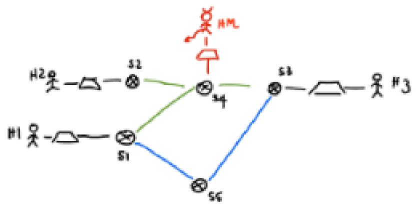


Fig. 2. Topology of the network

We used **miniedit**, that is the graphic interface of **Mininet**, to create the network using hosts and switches and then set the various values of the bandwidth, delay and queue size. The final result obtained is the following:

As per the network specifications, we set the bandwidth values of the green links to **double** those of the blue ones and we used the following parameters.

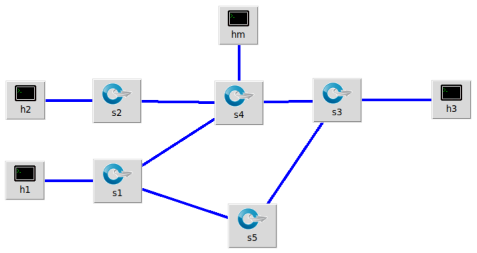


Fig. 3. Topology of the network realized with miniedit

For the *green links* we used:

- Bandwith: 10
- Delay: 5
- Max Queue size: 10

Instead for the *blue links* we used:

- Bandwith: 5
- Delay: 5
- Max Queue size: 10

Subsequently, to speed up the construction of the network, without going through miniedit every time, we have generated from it, through the appropriate function, the python script `run_network.py` for the automatic construction of the network.

The aim of the project was to use any tool to measure the performance from *h1* to *h3* assuming that the controller established that the path p1 = $\langle h1, s1, s4, s3, h3\rangle$ had to be used by all the flow rules, initially configured on each switch.

III. Latency with an increasing rate

The purpose of this plot is to consider a total traffic rate λ_{13} which shows how the latency T increases with increasing lambda.

For the realization of this plot we have made a series of changes to the `base_switch.py` script, that is, we have added a further parameter concerning the path *priority* in the `inst_path_rule()` function. While in the `flow_reroute_app.py` file, we have implemented the `set_green_flow_rule()` function, which performs a check to verify if the hosts *h1* and *h3* are up and if successful, we obtain their MAC addresses and call the `inst_path_rule()` function to define the path from h1 to h3 and vice versa.

Once the function was implemented we needed to generate traffic within our network, to do this we wrote a Python script called `ping.py`, which executes a function that at every second elapsed increases the number of pings per second performed, up to a maximum decided by us (for example 70). The ping commands are done through the use of *Threads*, this in order not to have blocking calls. The frequency with which the latter are generated is equal to `1/i`, where `i` is a counter that increases by two at each iteration, which corresponds to the number of requests to be made in a second.

Each Thread executes the ping command by calling the `ping_host()` function and the following is an example of how the ping command is executed:

```
subprocess.run(["ping", "-c", "1",
    "-s", SIZE, IP_H3],
```

Where the `-c` parameter indicates the number of packets to be sent, in our case `1` for each *Thread*, while `-s` the size of the packet that we have set to `9972` bytes (to which we add the 28 of the header, therefore `10000`), for have more visible results.

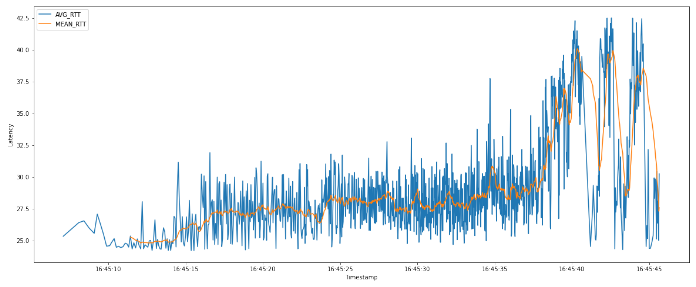


Fig. 4. Plot 1 test with max 70 packets

We have done several tests by varying the maximum number of packets sent every second, and we have decided to show the most explanatory, i.e. with the greatest number of packets (70). Specifically, we can see how as the number of requests made increases, the rate also increases until there is a loss of packets after about 40 seconds of execution, which lead to a fast decrease in latency, which however increases again, repeating this procedure more times.

IV. Attack Detection

The second part of the experiment on this network is to simulate a **Denial of Service attack** performed by the *hm* host, so that *s4* adds an intolerable delay to any communication that passes through it.

The purpose of this second plot is to show the increase in latency T between *h1* and *h3* as a consequence of the attack.

For the realization of this plot we have implemented the `set_malicious_flows()` function inside the `flow_reroute_app.py` file, which performs a check to verify if the hosts *hm* and *h2* are up and if successful, we get their MAC addresses and generate multiple identical flow rules but with different priorities for the same pair of communicating hosts (i.e. *hm* and *h2*).

However, this is not enough to obtain a DoS Attack, as to simulate it, *hm* should send streams with increasing size and/or frequency, until a certain value is reached which involves the fall of some streams passing through *s4*. To do this we use an additional python script called `dos_attack.py`, in which new *Threads* are generated with the same method as `ping.py`, each of which performs a large number of pings with a increasing frequency. Instead, as regards the `ping.py` script we have modified it (for this and subsequent plots) in such a way as to perform a ping of one packet per second (for a better visualization and understanding of the plots) and increased the size of the latter to `15000` bytes (`14972 + 28`).

As expected, the DoS attack involves an ever greater increase in latency, as we can see from the following figure.

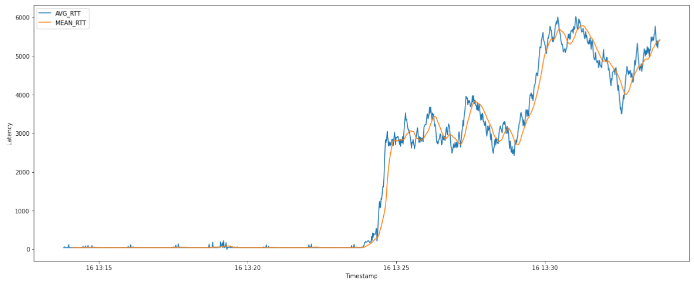


Fig. 5. Plot 2

V. Redirection

In this section we use a third timeline plot to show the behavior of the system, which when it detects the DoS attack performed in the previous section IV, reacts by redirecting the flow along the path $p2 = \langle h1, s1, s5, s3, h3 \rangle$. More specifically, along a separate channel H1 trigger the intervention of the controller, which re-routes the flow between *h1* and *h3*, along the path p2 with lower bandwidth, bypassing the attacked network components.

For the realization of this plot, our idea was to use a JSON file, in which to write the RTT values of the pings performed by the `ping.py` script, and then read them through the use of the `read_rtt_h1()` function, in the `flow_reroute_app.py` file, which performs the reroute action on p2 if it reads an RTT value greater than 1000 ms.

As we can see from the following figure the packets reach an Rtt close to 1500 ms, this is due to a delay problem, as time elapses between the writing and reading of the Rtt value in the JSON file, time in which the packet latency continues to increase.

Furthermore we can note that after the redirection the average value of the Rtt is greater than before the DoS attack, this is due to the fact that the p2 path has lower capacities (by half) than p1.

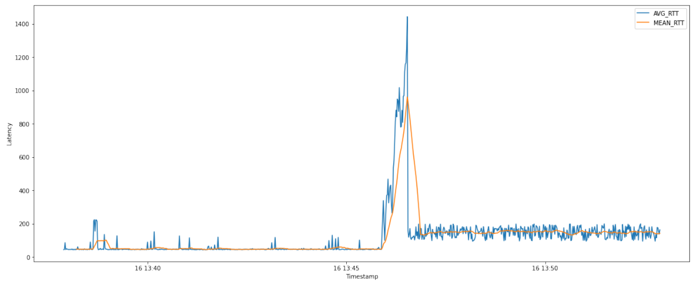


Fig. 6. Plot 3

VI. Recovery

In this last part we were asked to 'ask' the controller to block malicious traffic and dynamically re-establish the most convenient path along the p1 path. To complete this task we have modified the `base_switch.py` file in particular by working on the `_packet_in_handler()` function which allows us to analyze the origin of a package and possibly not route it if it comes from an unwanted source.

More specifically we used a boolean variable `drop_hm_packet`, internal to the `BaseSwitch` class, which by deafult is set to **False** but when the *Redirection* operation is performed, in `read_rtt_h1()` function in the `flow_reroute.py` file, it is set to **True**, so that in the `_packet_in_handler()` function we can discard the packet by checking the value of the latter (if it is equal to True) and if the source is the host *hm*.

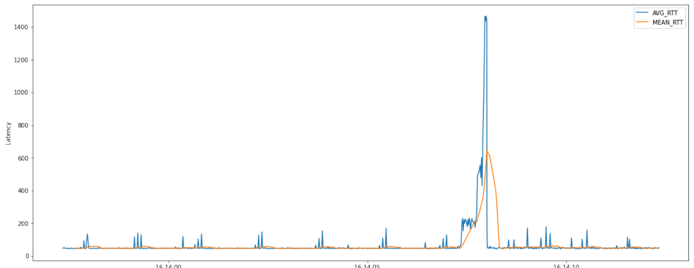
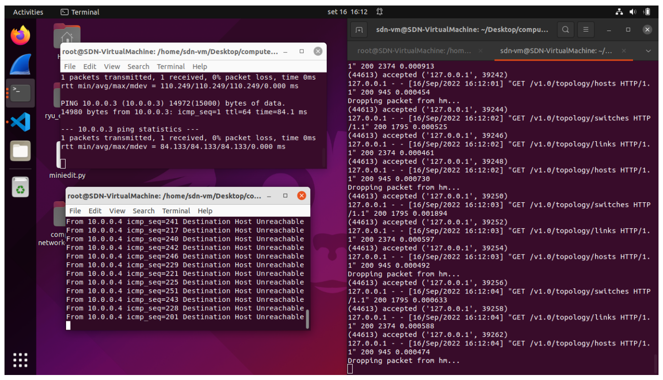


Fig. 7. Plot 4

We can see how compared to Figure 6 after the peak of latency, the latter turns out to be on average lower, precisely because the capacity of the links of the original path turns out to be double.

In the following figure we can see the setup of our environment, where it can be seen that in the *hm* terminal (bottom left) the destination host *h2* is unreachable precisely because the controller has blocked the traffic coming from it and then re-establishes the original path more rapid.


Activities
Terminal
set 16 16:12
root@SDN-VirtualMachine: /home/sdn-vm/Desktop/compute...
File Edit View Search Terminal Help
ryu_
miniedit.py
root@SDN-VirtualMachine: /home/sdn-vm/Desktop/co...
com
network
sdn-vm@SDN-VirtualMachine: ~/Desktop/compu...
root@SDN-VirtualMachine: /hom...
sdn-vm@SDN-VirtualMachine: ~/...

```
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 110.249/110.249/110.249/0.000 ms

PING 10.0.0.3 (10.0.0.3) 14972(15000) bytes of data.
14980 bytes from 10.0.0.3: icmp_seq=1 ttl=64 time=84.1 ms

--- 10.0.0.3 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 84.133/84.133/84.133/0.000 ms
```

```
From 10.0.0.4 icmp_seq=241 Destination Host Unreachable
From 10.0.0.4 icmp_seq=217 Destination Host Unreachable
From 10.0.0.4 icmp_seq=240 Destination Host Unreachable
From 10.0.0.4 icmp_seq=242 Destination Host Unreachable
From 10.0.0.4 icmp_seq=246 Destination Host Unreachable
From 10.0.0.4 icmp_seq=229 Destination Host Unreachable
From 10.0.0.4 icmp_seq=221 Destination Host Unreachable
From 10.0.0.4 icmp_seq=225 Destination Host Unreachable
From 10.0.0.4 icmp_seq=251 Destination Host Unreachable
From 10.0.0.4 icmp_seq=243 Destination Host Unreachable
From 10.0.0.4 icmp_seq=228 Destination Host Unreachable
From 10.0.0.4 icmp_seq=201 Destination Host Unreachable
```

```
1" 200 2374 0.000913
(44613) accepted ('127.0.0.1', 39242)
127.0.0.1 - - [16/Sep/2022 16:12:01] "GET /v1.0/topology/hosts HTTP/1.
1" 200 945 0.000454
Dropping packet from hm...
(44613) accepted ('127.0.0.1', 39244)
127.0.0.1 - - [16/Sep/2022 16:12:02] "GET /v1.0/topology/switches HTTP
/1.1" 200 1795 0.000525
(44613) accepted ('127.0.0.1', 39246)
127.0.0.1 - - [16/Sep/2022 16:12:02] "GET /v1.0/topology/links HTTP/1.
1" 200 2374 0.000461
(44613) accepted ('127.0.0.1', 39248)
127.0.0.1 - - [16/Sep/2022 16:12:02] "GET /v1.0/topology/hosts HTTP/1.
1" 200 945 0.000730
Dropping packet from hm...
(44613) accepted ('127.0.0.1', 39250)
127.0.0.1 - - [16/Sep/2022 16:12:03] "GET /v1.0/topology/switches HTTP
/1.1" 200 1795 0.001894
(44613) accepted ('127.0.0.1', 39252)
127.0.0.1 - - [16/Sep/2022 16:12:03] "GET /v1.0/topology/links HTTP/1.
1" 200 2374 0.000597
(44613) accepted ('127.0.0.1', 39254)
127.0.0.1 - - [16/Sep/2022 16:12:03] "GET /v1.0/topology/hosts HTTP/1.
1" 200 945 0.000492
Dropping packet from hm...
(44613) accepted ('127.0.0.1', 39256)
127.0.0.1 - - [16/Sep/2022 16:12:04] "GET /v1.0/topology/switches HTTP
/1.1" 200 1795 0.000633
(44613) accepted ('127.0.0.1', 39258)
127.0.0.1 - - [16/Sep/2022 16:12:04] "GET /v1.0/topology/links HTTP/1.
1" 200 2374 0.000588
(44613) accepted ('127.0.0.1', 39262)
127.0.0.1 - - [16/Sep/2022 16:12:04] "GET /v1.0/topology/hosts HTTP/1.
1" 200 945 0.000474
Dropping packet from hm...
```

Fig. 8. Setup for each plot

References

[1] Tarun Thakur, Software Defined Networking (SDN) explained for beginners *Software Defined Networking (SDN) explained for beginners*

[2] Lubna FayezEliyan, Roberto Di Pietro (September 2021); *Denial-of-Service (DoS) Attacks in an SDN Environment*.

[3] Rajat Kandoi, Markku Antikainen (June 2015); *Denial-of-service attacks in OpenFlow SDN networks*

[4] University of South Carolina; *Network Tools and Protocols*, "CyberTraining CIP: Cyberinfrastructure Expertise on High-throughput Networks for Big Science Data Transfers".