



Frankfurt University of Applied Sciences

Master Information Technology

Course: Mobile Computing

Network virtualization with Containernet: Network slicing based on Network Service Header

Sang Nguyen

1185021

Long Nguyen

1067534

March 8, 2021

Disclaimer

I declare that this report is a product of my own work, unless otherwise referenced. I also declare that all opinions, results, conclusions and recommendations are my own and may not represent the policies or opinions of Frankfurt University of Applied Sciences.

Sang Nguyen

Long Nguyen

Abstract

This project is intended to be used for evaluation of the module “Mobile Computing” of master program “Information Technology” at Frankfurt University of Applied Sciences.

In conventional networks with traditional routers and switches, improvement in terms of network performance was mainly achieved through the improvement of hardware resources (higher link capacity, routers and switches with better performance,...). Nowadays, in a networking environment where topology is becoming increasingly dynamic and versatile through the power of many virtualization technologies, a good networking approach requires not only high-quality hardware but also efficient utilization of those hardware resources. Software-defined networking (SDN) is one such approach. SDN is a networking approach that deploys a dynamically and programmatically managed network topology. It allows for optimized utilization of a network resources while keeping or even surpassing the performance of normal static networking approach.

One of the applications of SDN is network slicing. One can virtually divide a physical network into different logical networks. These logical networks will use the same hardware resource but they belong to different administrative domains and are managed by separate entities. There are many approaches to realize network slicing, e.g. virtual LAN, MPLS,... The main point is to implement a mechanism to differentiate between different flows and to treat them correspondingly.

This report describes the prototype of one approach in realizing such networks. Specifically, in this project, the slicing is achieved through the utilization of Network Service Header (NSH). This approach is conceptually similar to VLAN or MPLS approach, only the representation is different (instead of using VLAN tag or MPLS tag to differentiate traffics between different slices, NSH Service Path Index (SPI) is used). In general, every SPI belongs to one certain network slice and the transport node will base on these SPI values to isolate traffic flows of one network slice from those of other slices.

The tools used in this project includes: Containernet, which is an extension from Mininet, is used to emulate virtual network topology, Open vSwitch is used in the emulated network as SDN switch, POX controller platform is used to design controllers to handle NSH related functions and implement network slicing. Since these tools are still under development, the result of this project is also experimental and only serves to demonstrate the concept of network slicing using NSH.

Keyword: SDN, NSH, mininet, containernet, POX controller, Open vSwitch

Contents

Disclaimer	2
Abstract	3
Contents	4
Table of figure	6
1. Introduction	8
1.1. Purpose and Scope of the document	8
1.2. Main project requirements	8
1.3. Approach overview	8
1.4. Task assignment to project participants	8
2. Utilized tools	9
2.1. Containernet (Docker container-extended Mininet)	9
2.2. Open vSwitch	9
2.3. POX controller	9
2.4. Python file sharing server – Droopy	9
3. Description	10
3.1. Network Topology by Mininet	10
3.1.1. <i>Docker hosts</i>	10
3.1.2. <i>OVS switches</i>	10
3.2. POX controller and Network Slicing	11
3.2.1. <i>Common database and the slicing mechanism</i>	14
3.2.2. <i>Instance-database</i>	17
3.3. Extension: Dynamic topology adaptation	17
3.4. File sharing service components	19
4. Testing	20
4.1. Naming convention	20
4.1.1. <i>IP addresses of hosts</i>	20
4.1.2. <i>Access-side IP, access-side MAC and core-side MAC of access switches</i>	20
4.2. Experiment setup	21
4.2.1. <i>Set up environment</i>	21
4.2.2. <i>Download experiment specific files</i>	21
4.2.3. <i>Pull Docker images for file sharing service</i>	21
4.3. Network slicing using NSH	21

4.4.	Network slicing and dynamic topology adaptation	25
4.5.	Sliced network with file sharing service	26
5.	Discussion	32
5.1.	Problems encountered during developing phase	32
5.2.	Evaluation of the project's result	32
5.2.1.	Some justifications	32
5.2.2.	Project evaluation	32
5.2.3.	Further suggestion on improvement	33
6.	Conclusion	34
	Reference	35

Table of figure

Figure 1 Network topology intended for two logical network slices (red: slice 1, blue: slice 2)	10
Figure 2 POX controller connections to OVS switches.....	11
Figure 3 Summarizing on controller's runtime programming structure for topology in Figure 2	12
Figure 4 Decision tree of a sub-module instance, which controls one specific access switch (without extension for dynamic topology adaptation)	13
Figure 5 Sample common database of the controller	14
Figure 6 Some important databases (e.g. for an IP packet with IP source "10.1.1.1" and destination "10.2.1.3", from router_ip_db, an ordered pair of switch indices is deduced, which is 1 and 2, this corresponds to SPI value of 9 (both host from "slice-1"))	15
Figure 7 Logically defined service paths for network slice 1	16
Figure 8 Logically defined service paths for network slice 2	16
Figure 9 Sample instance-database of the controller (every access switch should be represented by one such database, with name equal to the switch's DPID and value equal a dictionary of parameters)	17
Figure 10 nsh_forwarding_table database of access switch with DPID 1	17
Figure 11 Upon losing connection to one of its hosts, the switch will report to the controller.....	18
Figure 12 When detecting a new connection, the switch will report to the controller	18
Figure 13 UI of the file sharing service.....	19
Figure 14 Network topology intended for two logical network slices (red: slice 1, blue: slice 2) (replicated).....	20
Figure 15 Naming convention for IP addresses used in testing phase (please note that the IP addresses should be written without any leading zeros)	20
Figure 16 Naming convention of the access switch's MACs in testing phase (please note the hex format of MAC)	21
Figure 17 Reachability test of unsliced network.....	22
Figure 18 Database initiated and reported by the controller	23
Figure 19 Reachability test in sliced network	23
Figure 20 Original ping message from host "d1"	23
Figure 21 NSH-encapsulated ping message in core network	24
Figure 22 Decapsulated ping message arriving at host "d2"	24
Figure 23 SPI assignment process.....	24
Figure 24 Database recalculated after detecting network change.....	25
Figure 25 Reachability test in sliced network after topology change.....	26
Figure 26 Original ping message from host "d1"	26
Figure 27 NSH-encapsulated message from in core network.....	26
Figure 28 Decapsulated ping message arriving at host "d3"	26
Figure 29 Check socket to access "d1" graphical user interface.....	27
Figure 30 VNC Viewer interface	28
Figure 31 Opening web browser on "d1"	28
Figure 32 Accessing file sharing service from "d1"	29
Figure 33 "d1" accesses file service	29
Figure 34 "d3" accesses file service and uploads files	30
Figure 35 "d1" reloads the site and notices the changes	30

Figure 36 "d4" does not see any changes because it is served by a different server and has no access to service in the other slice	31
--	----

1. Introduction

1.1. Purpose and Scope of the document

This report includes concepts, diagram, explanation, captured test data, discussion on a quick prototyping of network slicing based on Network Service Header (NSH) [1] with dynamic topology adaptation extension. Also included is demonstration of an auxiliary file sharing service with client-server model, which is called Droopy [2] and serves as an example of service deployment in the sliced network's environment.

1.2. Main project requirements

There are a dozen of requirements in the project description. However, three main functional requirements that need to be achieved are:

1. Network slicing based on NSH
2. Extension to the sliced network's capability
3. File sharing service implementation

1.3. Approach overview

In general, these requirements are satisfied as following:

1. Network slicing based on NSH: network slicing happens at the access switches, and carried out by a centralized controller. This means all the traffic will be forwarded back to the controller. The controller has one centralized database and it also creates different controlling module instances (with their own database) to manage each switch. Traffic from each switch will be handled by the corresponding controlling module instances. Basically, every incoming packets, whose source and destination are in the same slice, will be forwarded, otherwise, will be dropped.
2. Extension: When the network topology changes, the controller will be noticed. It will then update the database and recalculate all flow tables.
3. File sharing service: The simple Python-based file sharing service is reused from an existing open source project on Github, which is called Droopy [2]

1.4. Task assignment to project participants

- Sang Nguyen: Network slicing using NSH and extension for dynamic topology adaptation
- Long Nguyen: File hosting and sharing service

2. Utilized tools

In this project, there are three main software tools, which are used to create the virtual network and implement the NSH-related function for network slicing. The file sharing service is reused from an open source project called Droopy [2].

2.1. Containernet (Docker container-extended Mininet)

Mininet [3] is a network emulation software, whose functions are facilitated by many isolation and network stack management features of the Linux operating system. It helps to model quickly and compactly realistic networking scenarios in practice. Since Mininet is capable of emulating a large amount of network nodes, the hardware resource capability should also be taken into account when trying to emulate a complicated network with high load of computing requirements.

Containernet [4] is an extension of a Mininet, which adds the functions to use Docker containers as emulated hosts in Mininet virtual network. In this document, the terms Mininet and Containernet will be used interchangeably.

2.2. Open vSwitch

Open vSwitch [5] (OVS switch) is a multilayer virtual switch. This switch supports OpenFlow versions as well as its Nicira extension. This switch is utilized by Mininet in this project. Actually, in this project, many features of OVS switch are not utilized since network processing is concentrated to the controller. OVS switch only acts as a forwarding node to transport network traffic.

2.3. POX controller

POX controller platform [6] is a Python-based controller development kit, which is used to design a controller. There are many built-in functions of POX, which helps to accelerate the developing process. However, this is still an open project under development. Thus the reliability of the tool is not high. Some components are not yet developed, requiring users to come up with their own solution. In this report, the term “POX controller” will often be shortened to “controller”.

2.4. Python file sharing server – Droopy

This is a very simple implementation of a file sharing service based on Python [2]. This service follows the client-server model. The communication protocol between hosts and servers is HTTP. Hosts will be able to access the server’s service through a normal web browser.

3. Description

3.1. Network Topology by Mininet

Following is the depiction of the network topology used in this project.

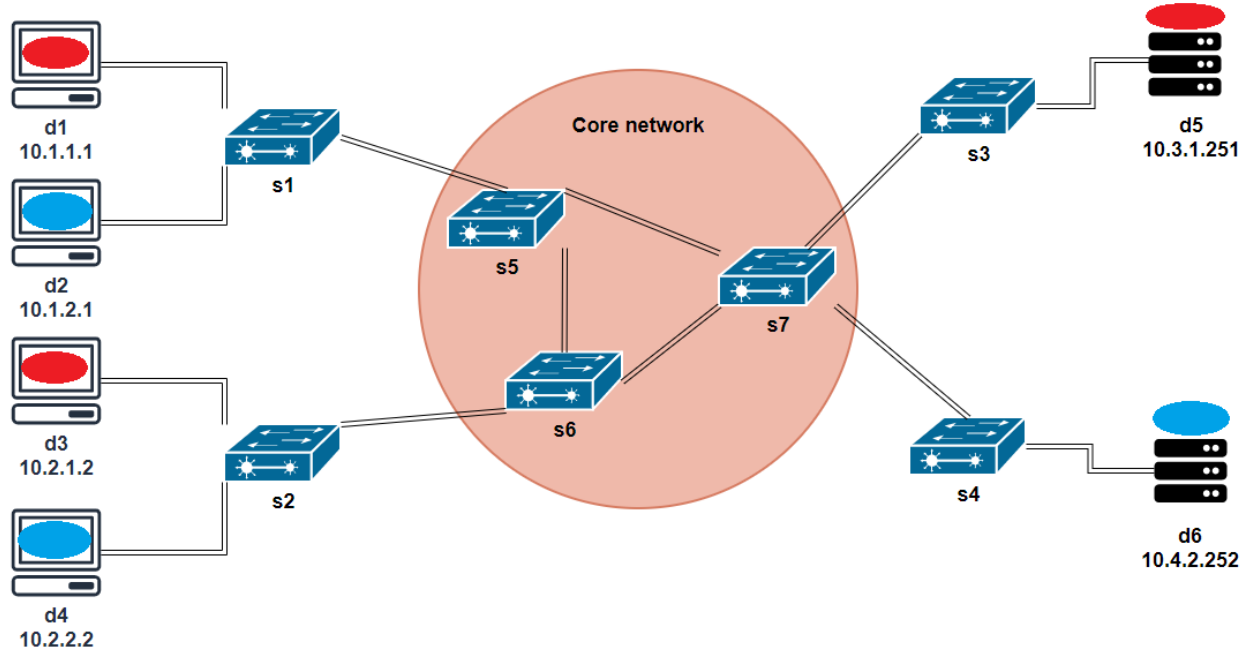


Figure 1 Network topology intended for two logical network slices (red: slice 1, blue: slice 2)

From a physical point of view, the test network topology used in this project is relatively simple and small. It consists of 6 hosts in the form of Docker containers. There are four clients and two servers for file sharing service. These hosts are connected to 4 access OVS switches. There are three core switches, which form a loop.

From a logical point of view, this network topology includes two logical slices, which are marked with red or blue eclipses. Red eclipse represents slice 1 and blue eclipse represents slice 2.

3.1.1. Docker hosts

There are four client containers (d1, d2, d3, d4) each with a light-weight web browser [7] already installed on it and two server containers (d5, d6) with running file sharing services (listening on port 80).

3.1.2. OVS switches

Every OVS switch in the Containernet topology is identified by a *DPID value*. This value could be set beforehand and queried at runtime. In this project testing phase, the naming convention makes it easy to deduce this value from host name e.g. host with name “d1” will have DPID equal to 1, “d2” will have DPID 2 and so on...

Access switches (s1, s2, s3, s4) will be connected to a POX controller and communicate with the controller through OpenFlow protocol. These access switches will forward all their received traffic to the controller in an OpenFlow PacketIn message. The controller has sub-module for each switch. These sub-modules will then process the packet based on rules set by the controller (derived from the main database, which contains information about physical and logical description of the network topology). After finishing processing the packet, the packet (if not dropped) will be sent back to the switch in an OpenFlow PacketOut message with an instruction which port to send the packet out.

Core switches (s5, s6, s7) will not be connected to any controller and will operate in standalone mode. In this mode of operation, the switches will act as a normal STP-enabled layer 2 learning switch.

3.2. POX controller and Network Slicing

Following is the depiction of the connection between OVS switches and the controller.

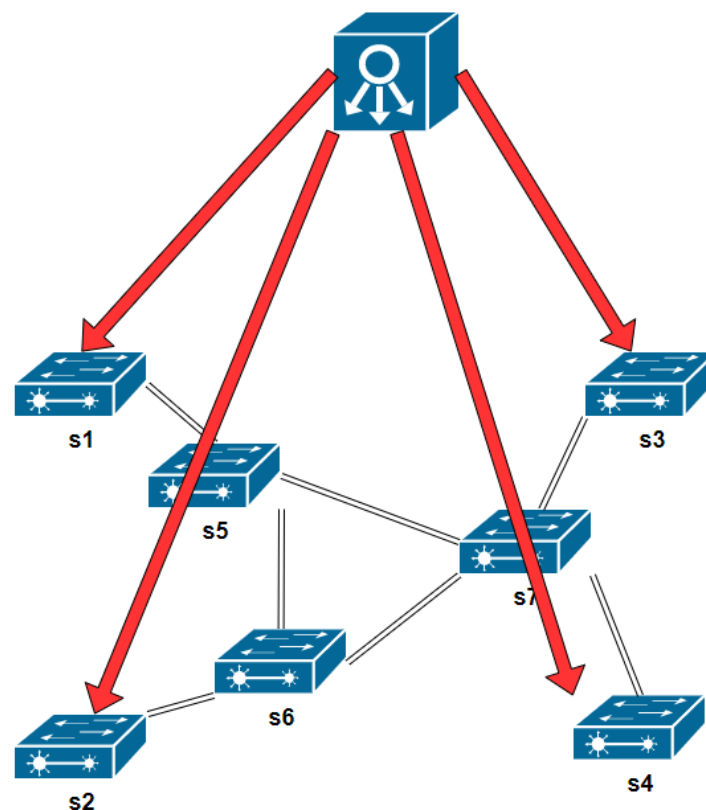


Figure 2 POX controller connections to OVS switches

The core switches will operate in standalone mode (they are not connected to a controller) and act as a normal Spanning Tree Protocol (STP)-enabled layer 2 learning switch. Therefore, routing inside core network depend only on layer 2 protocols. The access switch will not need IP address on the interface that connects to core network. Using STP and layer 2 switches is just a work-around for routing strategy inside core network (since this project does not focus on the problem of routing and forwarding). In practical scenario, routing inside core network should be implemented with conventional routing approaches (RIP, OSPF,...).

All access switches are controlled by one POX controller [6]. POX controller platform was written in Python [6]. The controller is basically a Python module. The designed controller defines a class for the sub-module (Python class named “accessSwitch”), each of whose instances will directly manage one access switch. Upon initiating, the controller will calculate the necessary databases and then listen for connection establishment from access switches. When a connection is set up, the controller will create one instance of sub-module class for that switch. This sub-module instance will have its own associated instance-database (related to the switch) and listen to that one switch exclusively. Upon receiving a packet from the switch, the sub-module instance will lead the packet through a decision chain to decide the actions to take on this packet.

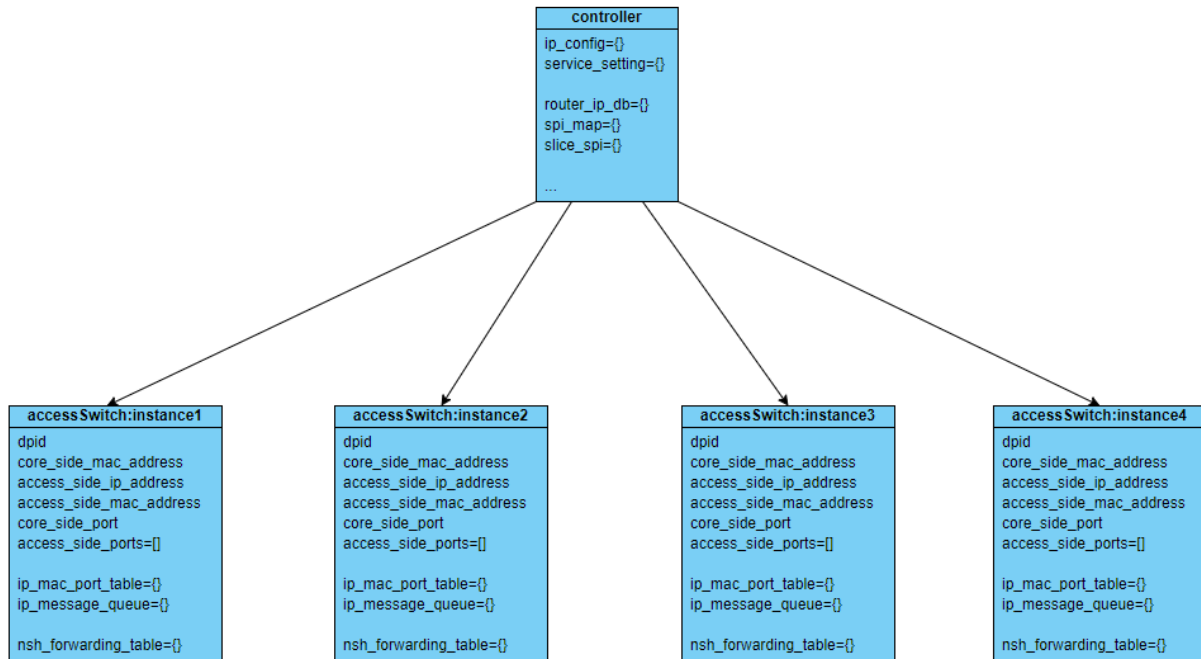


Figure 3 Summarizing on controller's runtime programming structure for topology in Figure 2

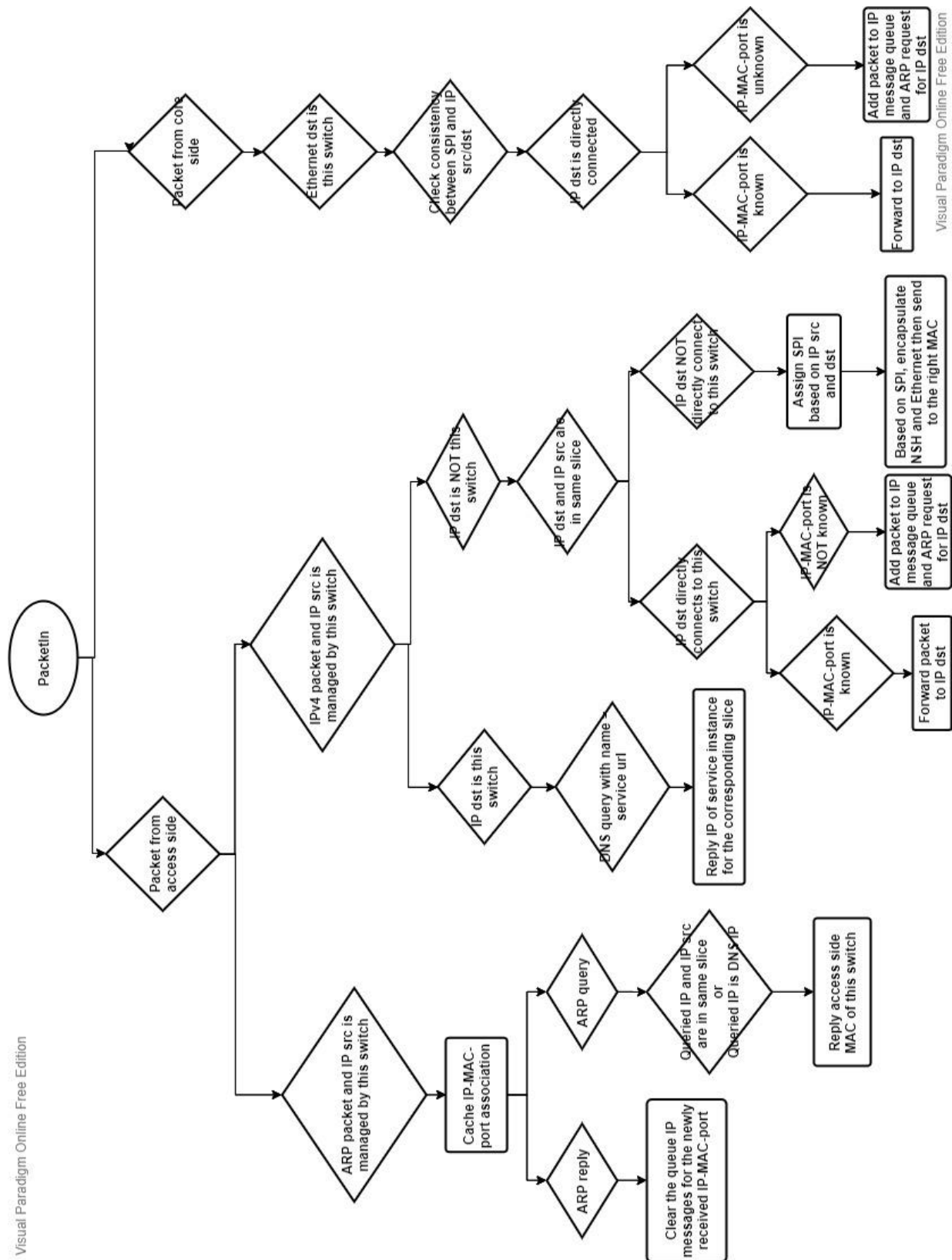


Figure 4 Decision tree of a sub-module instance, which controls one specific access switch (without extension for dynamic topology adaptation)

As can be seen from the above figure, to facilitate the function of a sub-module controlling an access switch, there are some databases that need to be created and managed. They could be categorized into common database and instance-databases.

3.2.1. Common database and the slicing mechanism

This data base is used and shared between different sub-module instances. This helps to provide a clearer look at the network topology and provide a unified and easy-to-managed control strategy (this is needed for the extension). Information contained in this database includes: physical and logical description of the network topology, common parameters to define the service utilized in the network.

```
# Main configuration area for admin
ipconfig = {
  "10.1.1.1": {"router-dpid":1, "slice":"slice-1"}, # NOTE: no leading zeros e.g. "10.01.001.01"
  "10.1.2.1": {"router-dpid":1, "slice":"slice-2"},
  "10.2.1.2": {"router-dpid":2, "slice":"slice-1"},
  "10.2.2.2": {"router-dpid":2, "slice":"slice-2"},
  "10.3.1.251": {"router-dpid":3, "slice":"slice-1"},
  "10.4.2.252": {"router-dpid":4, "slice":"slice-2"}
}

service_setting = {
  "service-url":"filesharing.frauas",
  "slice-service-ip-map":{
    "slice-1": "10.3.1.251", # NOTE: no leading zeros e.g. "10.03.001.251"
    "slice-2": "10.4.2.252"
  }
}
```

Figure 5 Sample common database of the controller

As can be seen from the above figure, every host in the network needs to be defined in the common database ipconfig. DPID is the data path identifier of a switch in general. This parameter could be deduced from the name of the switch in Mininet e.g. a switch with name “s1” will have DPID 1 and so on... For each host entry in ipconfig database, “router-dpid” is the DPID of the access switch that this host directly connects to, “slice” is the name of the slice this host belongs to. The controller designed in this project only supports hosts with single connection to a single access switch.

Every host, which offers the file sharing service, must also have another entry in the service_setting database. This entry specifies the slice that this service belongs to. “service-url” parameter used to set the domain name of the file sharing service. This database is mainly used for the purpose of a simple ad-hoc DNS function. With this DNS function, user of each host only needs to type in the domain name and will be directed to the correct file sharing server’s service.

From the main common database some other auxiliary databases will be deduced. One important derived database is the spi_map database, which represents the mapping required to assign SPI to every IP packet.

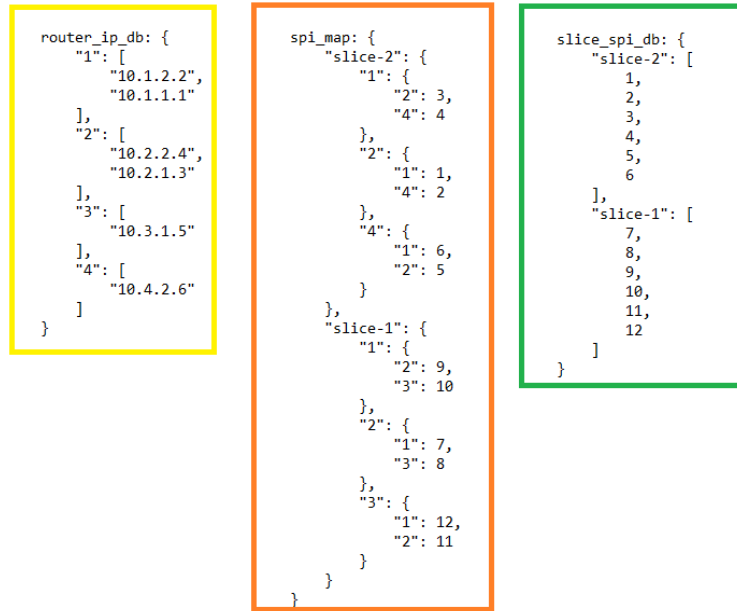


Figure 6 Some important databases

(e.g. for an IP packet with IP source "10.1.1.1" and destination "10.2.1.3", from router_ip_db, an ordered pair of switch indices is deduced, which is 1 and 2, this corresponds to SPI value of 9 (both host from "slice-1"))

To realize network slicing, the controller will define logical service path for every pair of access switches, which manage hosts from the same slice. For example, for the network topology used in this project, there will be three service paths defined for traffic of slice 1 and three service paths defined for network slice 2. For slice 1, there will be service paths from access switch s1 to s2, s2 to s3 and s1 to s3. For slice 2, there will be service paths from s1 to s2, s2 to s4 and s1 to s4. Every logical service paths is identified by Service Path Index (SPI) of NSH header, which encapsulates the original IP packet (from access side) once it sets out to the core network (after being classified).

When a packet is classified, it will be checked if the source and destination IP addresses are in the same slice. After it is made sure that, they are eligible traffic from hosts in the same slice, the IP addresses will be converted to access switch indices. Based on these indices and spi_map, the packet will be assigned the appropriate SPI. The packet will then be encapsulated in NSH header with the assigned SPI and sent to the NSH Service Function Forwarder (SFF) module (in this project, it is pretty plain and simple, consisting of only a packet forwarding function and a small NSH forwarding table). At the NSH SFF, the SPI will be checked and the packet will be encapsulated in an Ethernet header with destination MAC corresponding to the SPI in NSH header (based on the so-called NSH forwarding table – nsh_forwarding_table).

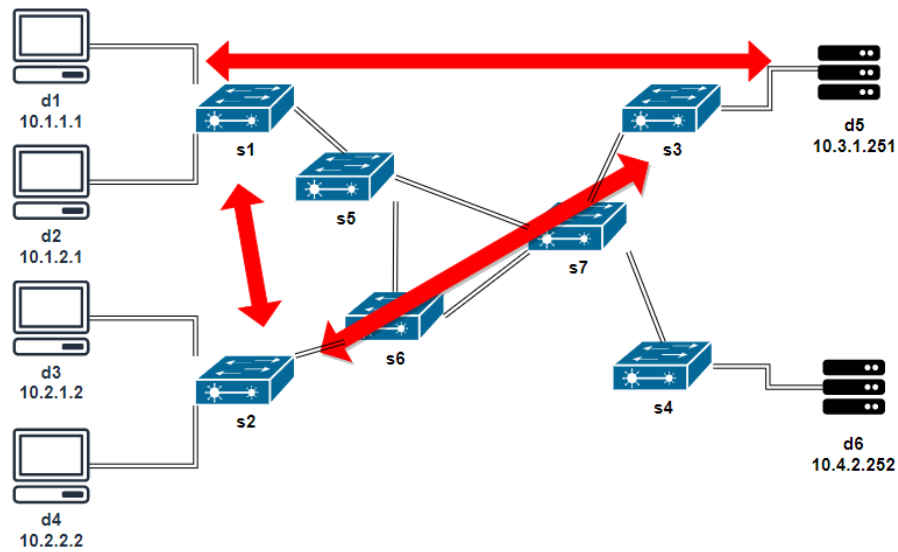


Figure 7 Logically defined service paths for network slice 1

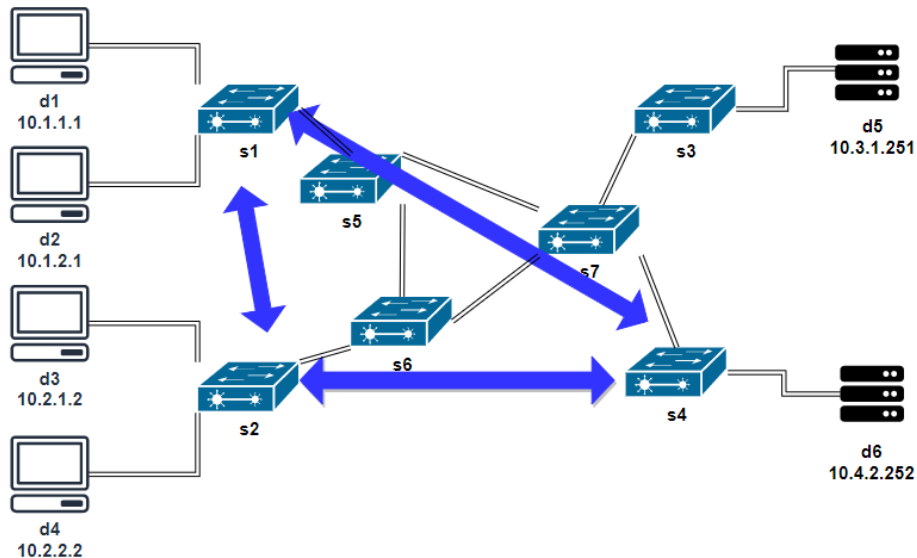


Figure 8 Logically defined service paths for network slice 2

Inside core network, packets are not actually routed. The core switches act as STP-enabled layer 2 switches, therefore traffic in the core network will not subject to loop but they are not routed in an efficient manner either. Transportation and security in the core network are based on mutual trust. Only packets with destination MAC of one certain access switch will be processed by that switch, otherwise ignored.

3.2.2. Instance-database

The instance-databases carry switch-specific networking parameters. The `access_side_ports` will be monitored and informed to the controller in case of any dynamic changes. The `nsh_forwarding_table` will be derived from the main common database, it facilitates the interaction between NSH and Ethernet (service layer and transport layer) as described in the above section.

```
router_setting = {
  1: {
    "access_side_ip_address": "10.0.0.1", # NOTE: no leading zero e.g. "10.00.000.01"
    "access_side_mac_address": "00:00:00:00:00:01",
    "core_side_mac_address": "00:00:01:00:00:00",
    "core_side_port": 1,
    "access_side_ports": [],
    "nsh_forwarding_table": {}
  },
}
```

Figure 9 Sample instance-database of the controller (every access switch should be represented by one such database, with name equal to the switch's DPID and value equal a dictionary of parameters)

```
nsh_forwarding table of DPID 1: {
  "1": {
    "255": "00:00:04:00:00:00"
  },
  "2": {
    "255": "00:00:02:00:00:00"
  },
  "3": {
    "255": "out-of-sfc"
  },
  "5": {
    "255": "out-of-sfc"
  },
  "8": {
    "255": "out-of-sfc"
  },
  "10": {
    "255": "out-of-sfc"
  },
  "11": {
    "255": "00:00:03:00:00:00"
  },
  "12": {
    "255": "00:00:02:00:00:00"
  }
}
```

Figure 10 `nsh_forwarding_table` database of access switch with DPID 1

3.3. Extension: Dynamic topology adaptation

When there is a host suddenly disconnected from the network, the controller will delete all local information related to that host on the corresponding switch. When the host reconnects to the network at a different location, an OpenFlow message `PortStatus` will be sent to the controller. It will then update and recalculate all databases based on the new network topology information.

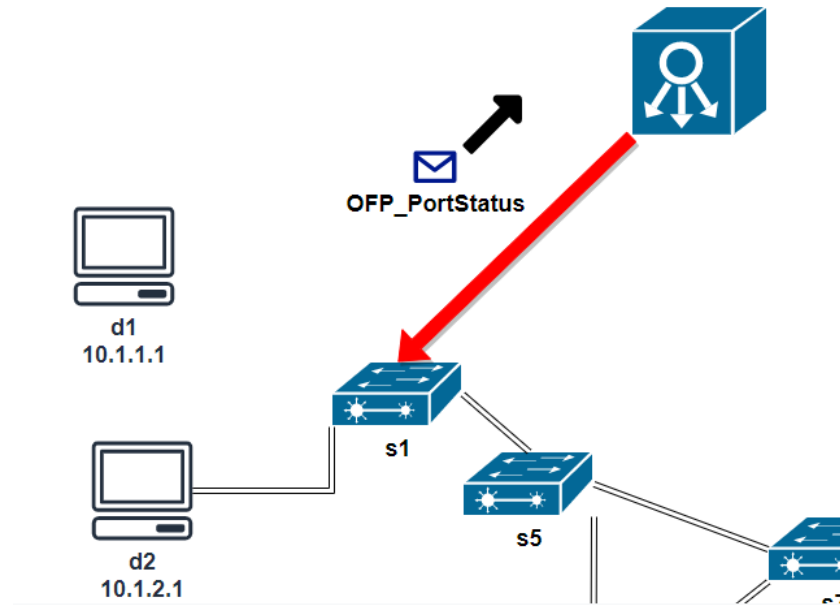


Figure 11 Upon losing connection to one of its hosts, the switch will report to the controller

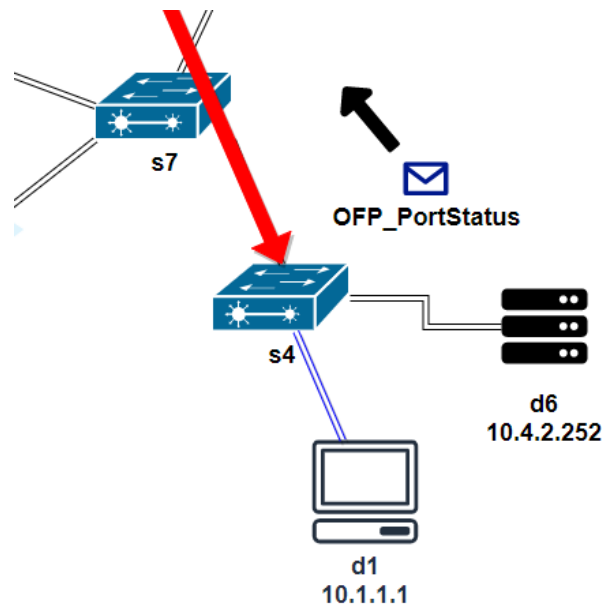


Figure 12 When detecting a new connection, the switch will report to the controller

When there is an OpenFlow PortStatus, which announces the disconnection of a host, arriving at the controller, the corresponding sub-module controlling instance will check its database for related information to this host and clear them.

When an OpenFlow PortStatus announcing the connection of a host arrives, the corresponding sub-module instance will try to carry out host discovery (by sending an ARP request and possibly ICMP echo request for every possible IP in the current network topology). Upon having obtained the IP address of the newly connected host, it will modify the main common database and make the controller recalculate all the databases. The new databases reflect the current topology of the network.

3.4. File sharing service components

The file sharing service Droopy [2] follows client-server model. The client can access the file sharing service through a web browser. The server will listen on famous web service port 80. They communicate with each other using HTTP protocol.

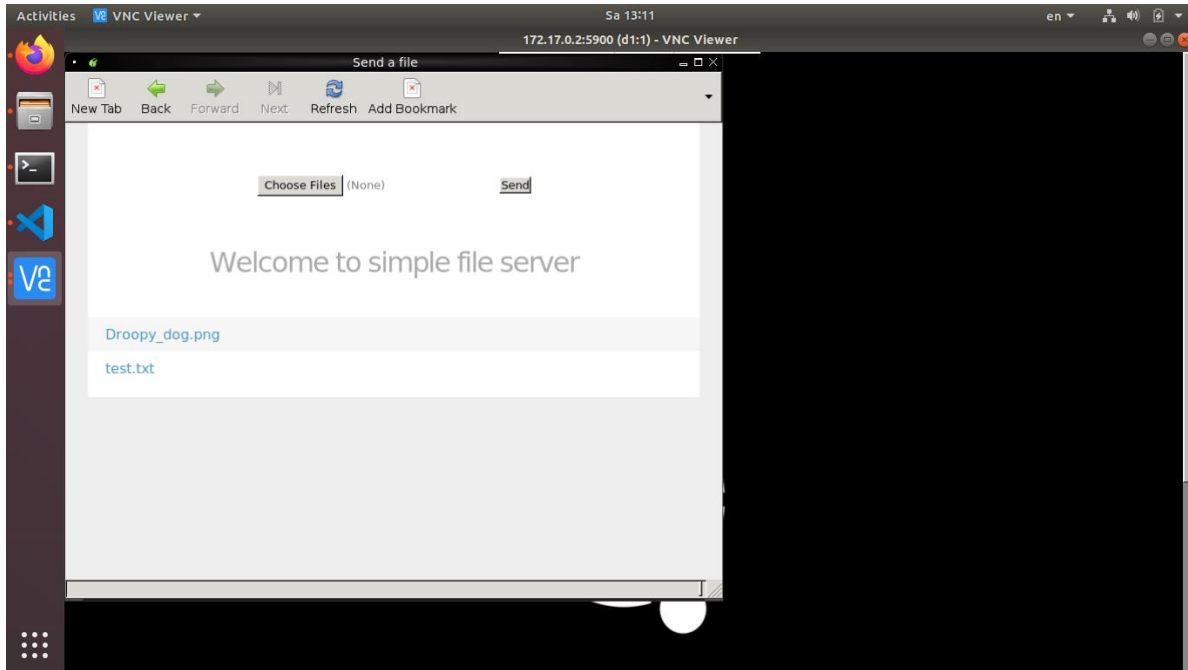


Figure 13 UI of the file sharing service

4. Testing

The network topology and slicing structure used in this test is the same as in Figure 1.

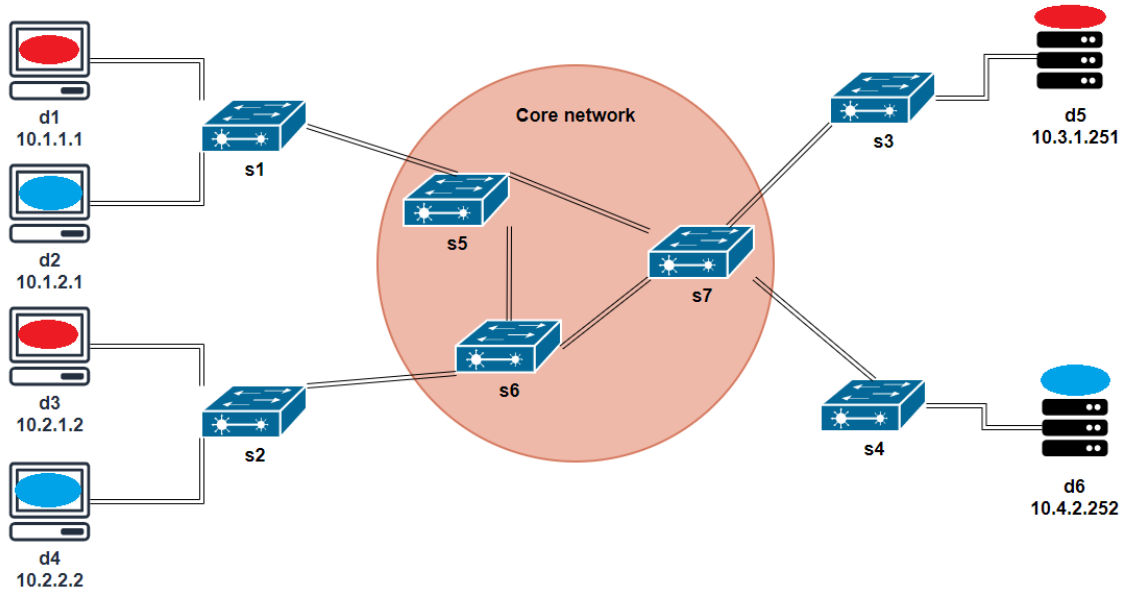


Figure 14 Network topology intended for two logical network slices (red: slice 1, blue: slice 2) (replicated)

4.1. Naming convention

In order to make testing phase become less complicated, consistent and meaningful naming convention is necessary.

4.1.1. IP addresses of hosts

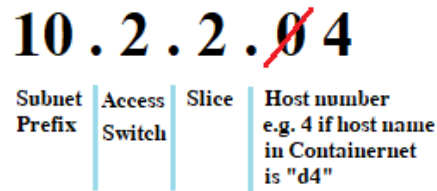


Figure 15 Naming convention for IP addresses used in testing phase (please note that the IP addresses should be written without any leading zeros)

For hosts IP addresses, first byte represents subnet prefix and is fixed (equal 10). Second byte is the DPID of the access switch that directly connects to this host. Third byte is the slice number, which this hosts belongs to. Fourth byte is the host number, e.g. this number is equal to 4 if the name of this host is "d4" in the Containernet topology file.

4.1.2. Access-side IP, access-side MAC and core-side MAC of access switches

Access-side IP address of all access switch is fixed (equal "10.0.0.1").

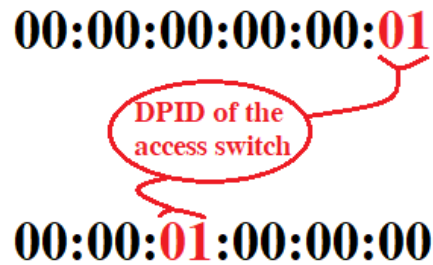


Figure 16 Naming convention of the access switch's MACs in testing phase (please note the hex format of MAC)

There is no core-side IP address because traffic in core network is routed only by layer 2 protocols. Again, this is a work-around since this project does not concern with routing and forwarding strategies.

4.2. Experiment setup

Before being able to run the experiment, some preparations need to be carried out. This experiment requires Docker desktop, both Python 2 and 3.

4.2.1. Set up environment

1. Download and install Containernet [4].
2. Download POX controller [6]. Notice that there are two versions of POX controller, please use the "fangtooth" version. This version could be found in branch "fangtooth" on POX repository.
3. Download VNC Viewer [10]. This allows us to access the Docker host through user interface. This is necessary since our file service servers and clients use web-based communication.

4.2.2. Download experiment specific files

4. Download the file named "my-network-topology.py" from Github repository of this project [8] and copy it to the directory ".../containernet/examples".
5. Download the file named "my-controller.py" from [8] and copy it to the directory ".../pox-fangtooth/ext".
6. Download the file named "nsh.py" from [8] and copy it to the directory ".../pox-fangtooth/pox/lib/packet"

4.2.3. Pull Docker images for file sharing service

7. Pull docker images of the file sharing service [2] using the following command:
docker pull sangnguyenfrauas/midoricontainer:v1.1
docker pull sangnguyenfrauas/droopy:v1.1

4.3. Network slicing using NSH

First, we need to start the network:

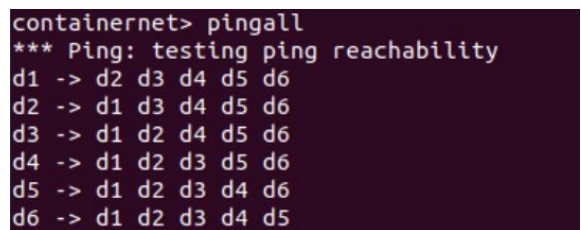
1. Change directory to ".../containernet/examples" and run the following command (with super-user privilege):
python3 my-network-topology.py

2. After the network successfully starts, run the following command:
`# sh time bash -c 'while ! ovs-ofctl show s1 | grep FORWARD; do sleep 1; done'`
Wait until this command returns.

Now our network is up and converges. To verify the network topology, run commands “links” and “dump”. A useful tool to quickly visualize the network topology is Narmox [9]. This tool graphically represent the network topology from data returned by “links” and “dump”. *However, it should be noticed that this tool was built for Mininet, the naming convention is not compatible with Containernet. Simply changing the names to those supported by Mininet will solve the problem.*

We could use the built-in Mininet tool “pingall” to check the reachability of all hosts in the network. Type the following command in the Containernet terminal:

`# pingall`



```
containernet> pingall
*** Ping: testing ping reachability
d1 -> d2 d3 d4 d5 d6
d2 -> d1 d3 d4 d5 d6
d3 -> d1 d2 d4 d5 d6
d4 -> d1 d2 d3 d5 d6
d5 -> d1 d2 d3 d4 d6
d6 -> d1 d2 d3 d4 d5
```

Figure 17 Reachability test of unsliced network

The result is that every host is reachable from the others.

Now let us start the POX controller:

3. Open another terminal, change directory to “.../pox-fangtooth/ext” and run the following command:
`# ./pox.py log.level --DEBUG my-controller`

Now both the controller and the virtualized network are up and running. Looking at the terminal of the POX controller, the first few lines are the calculated common databases.

```

sang@sang-Inspiron-5558:~$ cd containernet/pox-fangtooth/
sang@sang-Inspiron-5558:~/containernet/pox-fangtooth$ ./pox.py log.level --DEBUG my-controller
POX 0.6.0 (fangtooth) / Copyright 2011-2018 James McCauley, et al.
DEBUG:my-controller:router_ip_db: {
  "1": [
    "10.1.2.2",
    "10.1.1.1"
  ],
  "2": [
    "10.2.2.4",
    "10.2.1.3"
  ],
  "3": [
    "10.3.1.5"
  ],
  "4": [
    "10.4.2.6"
  ]
}
DEBUG:my-controller:slice_map: {
  "slice-2": {
    "1": {
      "2": 3,
      "4": 4
    },
    "2": {
      "1": 1,
      "4": 2
    },
    "4": {
      "1": 6,

```

Figure 18 Database initiated and reported by the controller

Run the reachability test again.

```

containernet> pingall
*** Ping: testing ping reachability
d1 -> X d3 X d5 X
d2 -> X X d4 X d6
d3 -> d1 X X d5 X
d4 -> X d2 X X d6
d5 -> d1 X d3 X X
d6 -> X d2 X d4 X
*** Results: 60% dropped (12/30 received)
containernet> 

```

Figure 19 Reachability test in sliced network

This time only hosts in the same network slice (according to the database printed out in the POX controller terminal) are reachable to each other. It means our network slicing works. Next, let us examine the NSH encapsulation of traffic in the core network. Use ping utility to ping host “d3” from host “d1”.

```

> Frame 149: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface s1-eth2, id 1
> Ethernet II, Src: 26:a2:d8:0e:d1:ab (26:a2:d8:0e:d1:ab), Dst: 00:00:00_00:00:01 (00:00:00:00:00:01)
> Internet Protocol Version 4, Src: 10.1.1.1, Dst: 10.2.1.3
> Internet Control Message Protocol

```

Figure 20 Original ping message from host “d1”

```

> Frame 145: 106 bytes on wire (848 bits), 106 bytes captured (848 bits) on interface s1-eth1, id 0
> Ethernet II, Src: Xerox_00:00:00 (00:00:01:00:00:00), Dst: Xerox_00:00:00 (00:00:02:00:00:00)
▼ Network Service Header
  00.. .... = Version: 0 (0x0)
  ..0. .... = O Bit: 0
  ...0 .... = C Bit: 0
  .... 1111 11.. .... = Time to live: 0x3f
  .... .... ..00 0010 = Length: 2 (0x02)
  MD Type: 2 (0x02)
  Next Protocol: IPv4 (1)
  SPI: 9 (0x000009)
  SI: 255 (0xff)
> Internet Protocol Version 4, Src: 10.1.1.1, Dst: 10.2.1.3
> Internet Control Message Protocol

```

Figure 21 NSH-encapsulated ping message in core network

```

> Frame 151: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface s2-eth2, id 2
> Ethernet II, Src: 00:00:00_00:00:02 (00:00:00:00:00:02), Dst: 8e:a1:c7:4c:a6:19 (8e:a1:c7:4c:a6:19)
> Internet Protocol Version 4, Src: 10.1.1.1, Dst: 10.2.1.3
> Internet Control Message Protocol

```

Figure 22 Decapsulated ping message arriving at host "d2"

The NSH header is now presented in-between the Ethernet header and the IPv4 header. The SPI value is consistent with the database. Here we have host "d1" with IP 10.1.1.1 and host "d3" with IP 10.2.1.3. For ping message from "d1" to "d3", the corresponding ordered pair of indices is 1-3. These hosts belong to slice "slice-1" so the SPI would be 9.

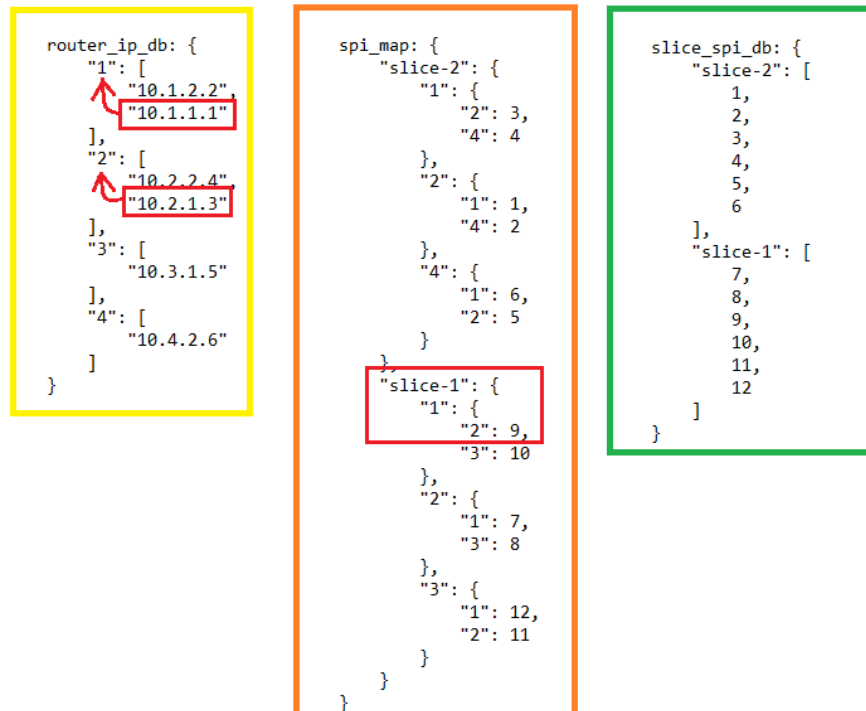


Figure 23 SPI assignment process

4.4. Network slicing and dynamic topology adaptation

Next, to test the ability to adapt to network changes, run the following command in Containernet terminal to move host “d1” from access network of “s1” to access network of “s4”:

```
# py s1.moveHost(d1,s1,s4)
```

Looking at output of the POX controller, new databases is already calculated.

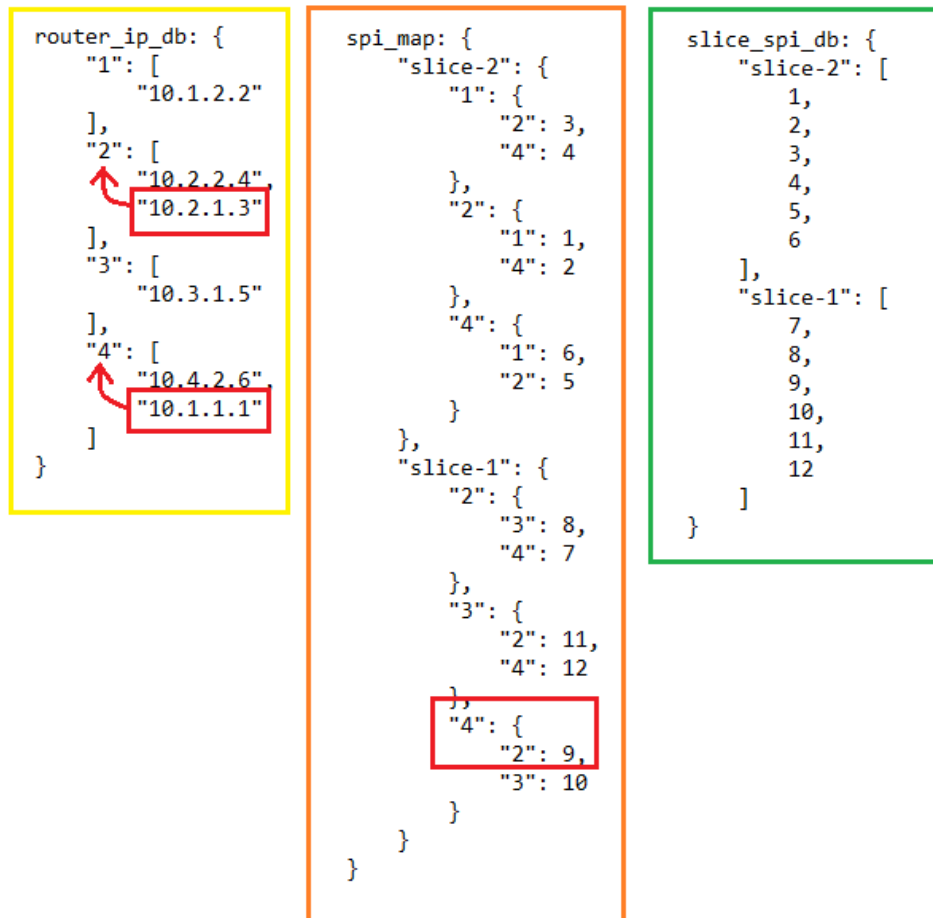


Figure 24 Database recalculated after detecting network change

Now use the “links” and “dump” to visualize the network topology. After that, run “pingall” to check reachability of the network again (sometimes, “pingall” need to be re-run one or two times to see the slicing effect).

```

containernet> pingall
*** Ping: testing ping reachability
d1 -> X d3 X d5 X
d2 -> X X d4 X d6
d3 -> d1 X X d5 X
d4 -> X d2 X X d6
d5 -> d1 X d3 X X
d6 -> X d2 X d4 X
*** Results: 60% dropped (12/30 received)
containernet>

```

Figure 25 Reachability test in sliced network after topology change

The network is sliced. Use ping utility to ping host “d3” from host “d1” again.

```

> Frame 16: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface s4-eth3, id 2
> Ethernet II, Src: 26:a2:d8:0e:d1:ab (26:a2:d8:0e:d1:ab), Dst: 00:00:00_00:00:04 (00:00:00:00:00:04)
> Internet Protocol Version 4, Src: 10.1.1.1, Dst: 10.2.1.3
> Internet Control Message Protocol

```

Figure 26 Original ping message from host "d1"

```

> Frame 11: 106 bytes on wire (848 bits), 106 bytes captured (848 bits) on interface s4-eth1, id 0
> Ethernet II, Src: Xerox_00:00:00 (00:00:04:00:00:00), Dst: Xerox_00:00:00 (00:00:02:00:00:00)
▼ Network Service Header
  00.. .... = Version: 0 (0x0)
  ..0. .... = O Bit: 0
  ...0 .... = C Bit: 0
  .... 1111 11.. = Time to live: 0x3f
  .... .... ..00 0010 = Length: 2 (0x02)
  MD Type: 2 (0x02)
  Next Protocol: IPv4 (1)
  SPI: 9 (0x000009)
  SI: 255 (0xff)
> Internet Protocol Version 4, Src: 10.1.1.1, Dst: 10.2.1.3
> Internet Control Message Protocol

```

Figure 27 NSH-encapsulated message from in core network

```

> Frame 10: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface s2-eth2, id 1
> Ethernet II, Src: 00:00:00_00:00:02 (00:00:00:00:00:02), Dst: 8e:a1:c7:4c:a6:19 (8e:a1:c7:4c:a6:19)
> Internet Protocol Version 4, Src: 10.1.1.1, Dst: 10.2.1.3
> Internet Control Message Protocol

```

Figure 28 Decapsulated ping message arriving at host "d3"

The SPI value is correctly assigned according to the common database of the controller.

Notice: Sometimes, after the host has been moved, it does not immediately start and response to incoming traffic. This makes the current controller’s adaptability less reliable and delays the re-calculation of the databases at the controller.

4.5. Sliced network with file sharing service

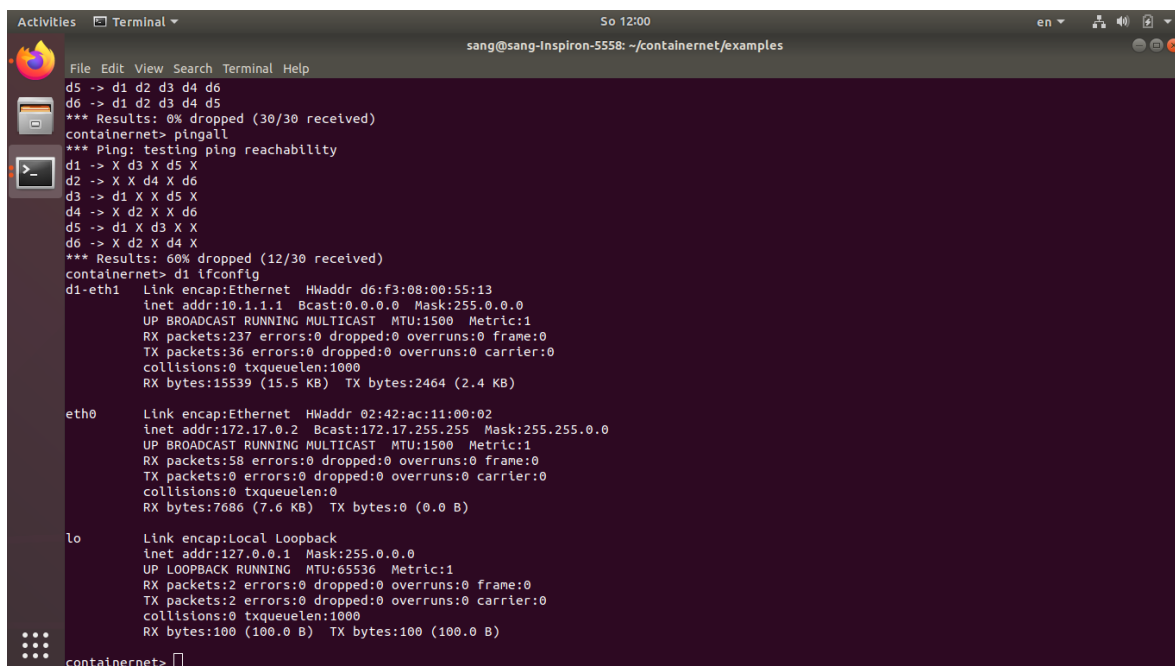
The above section already demonstrated the slicing and adapting capability of the network topology with controller.

For file service testing, however, there is one important point to be noticed. It is the problem about MTU size of links and switches used by Containernet. Default MTU size is 1500 bytes. Normally, for file service, the traffic will be transported inside packet of maximum MTU size. When we introduce an additional encapsulation (NSH header of 8 bytes), the packet size exceed the MTU of the links, thus it will not be forwarded. To mitigate this problem, we tried to increase the MTU of every link inside the virtualized network. In practice, this problem needs to be taken into consideration when utilizing NSH to encapsulate traffic in the network.

Integration of file sharing service to the current network topology is straightforward. Some pictures are provided here only for completeness.

To access file service through web browser on service client, we need to use VNC Viewer software to establish graphical user interface connection with the running Docker container host. All the necessary software and bootstrap are already included in the Docker image of client in this project. The following steps demonstrate how to use host “d1” user interface.

- From Containernet terminal, type in the following command:
containernet> d1 ifconfig



```

Activities  Terminal  So 12:00  en  sang@sang-Inspiron-5558: ~/containernet/examples

File Edit View Search Terminal Help

d5 -> d1 d2 d3 d4 d6
d6 -> d1 d2 d3 d4 d5
*** Results: 0% dropped (30/30 received)
containernet> pingall
*** Ping: testing ping reachability
d1 -> X d3 X d5 X
d2 -> X X d4 X d6
d3 -> d1 X X d5 X
d4 -> X d2 X d6
d5 -> d1 X d3 X X
d6 -> X d2 X d4 X
*** Results: 60% dropped (12/30 received)
containernet> d1 ifconfig
d1-eth1  Link encap:Ethernet  HWaddr d6:f3:08:00:55:13
         inet addr:10.1.1.1  Bcast:0.0.0.0  Mask:255.0.0.0
         UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
         RX packets:237 errors:0 dropped:0 overruns:0 frame:0
         TX packets:36 errors:0 dropped:0 overruns:0 carrier:0
         collisions:0 txqueuelen:1000
         RX bytes:15539 (15.5 KB)  TX bytes:2464 (2.4 KB)

eth0     Link encap:Ethernet  HWaddr 02:42:ac:11:00:02
         inet addr:172.17.0.2  Bcast:172.17.255.255  Mask:255.255.0.0
         UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
         RX packets:58 errors:0 dropped:0 overruns:0 frame:0
         TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
         collisions:0 txqueuelen:0
         RX bytes:7686 (7.6 KB)  TX bytes:0 (0.0 B)

lo       Link encap:Local Loopback
         inet addr:127.0.0.1  Mask:255.0.0.0
         UP LOOPBACK RUNNING  MTU:65536  Metric:1
         RX packets:2 errors:0 dropped:0 overruns:0 frame:0
         TX packets:2 errors:0 dropped:0 overruns:0 carrier:0
         collisions:0 txqueuelen:1000
         RX bytes:100 (100.0 B)  TX bytes:100 (100.0 B)

containernet>

```

Figure 29 Check socket to access "d1" graphical user interface

- Open VNC Viewer and access the following socket: ["d1" IP on eth0]:5900
e.g. 172.17.0.2:5900 in this case

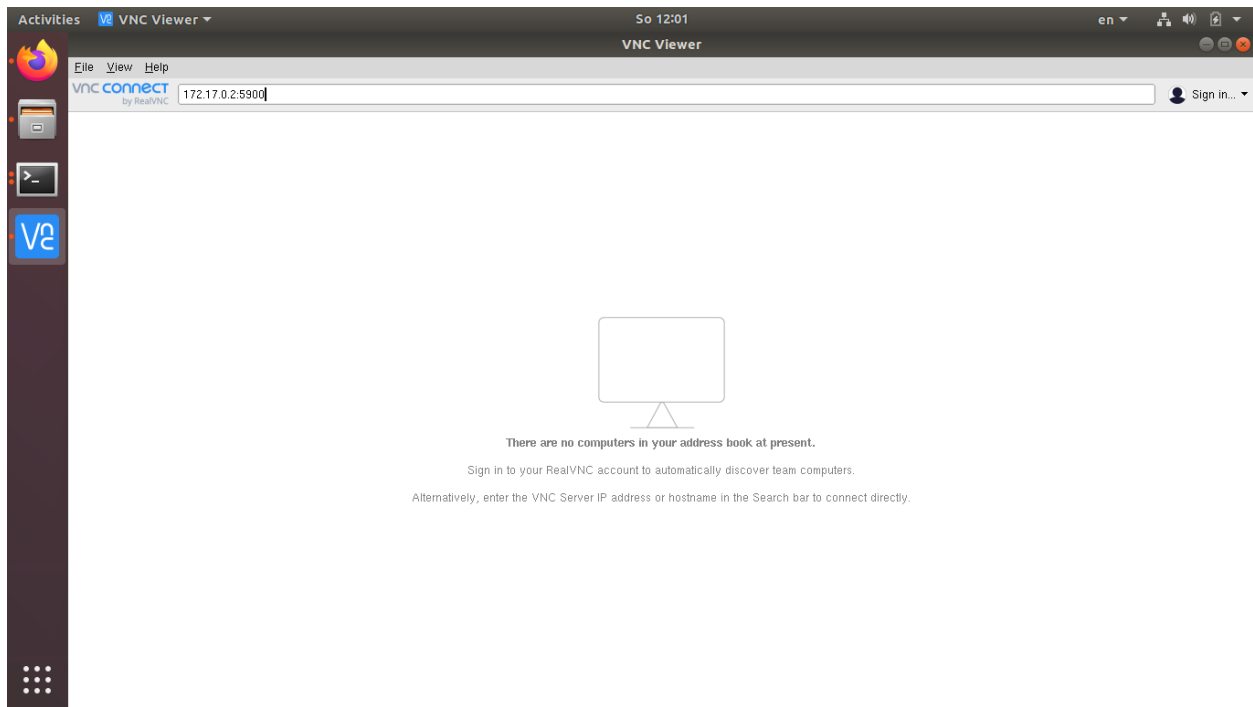


Figure 30 VNC Viewer interface

- Ignore all warning, right click on the screen and open Midori web browser

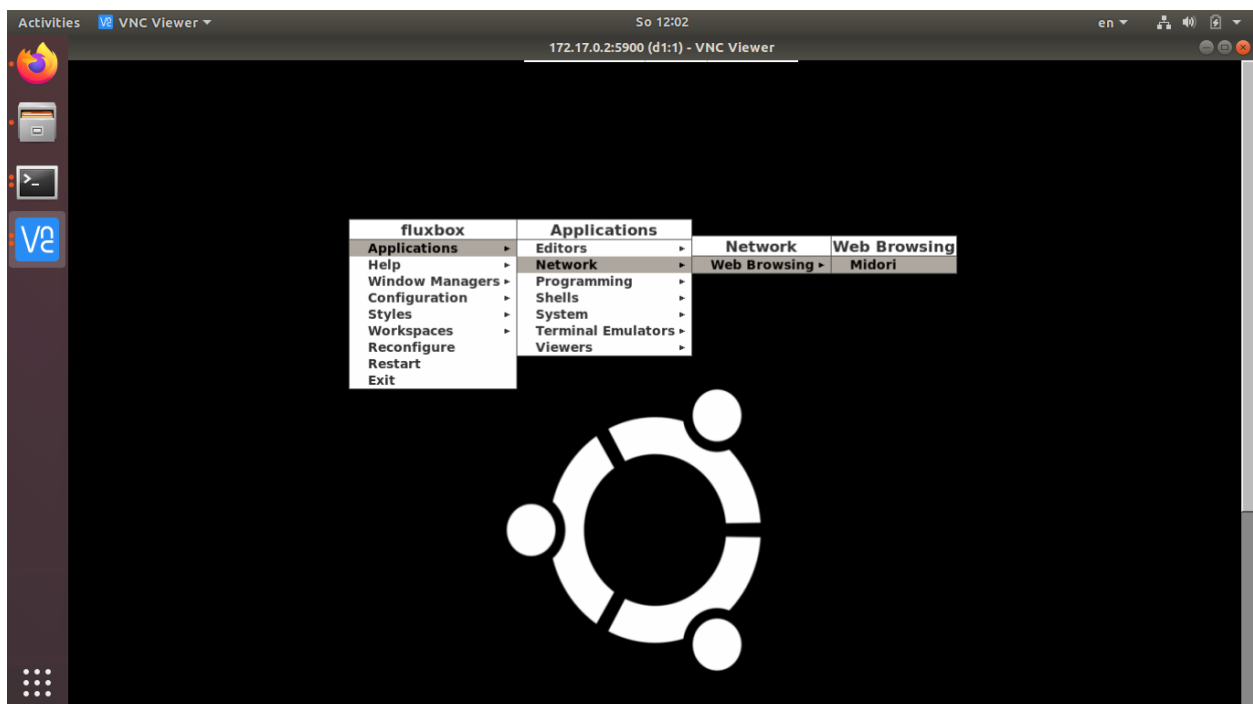


Figure 31 Opening web browser on "d1"

- Access the file service with domain name “filesharing.frauas”

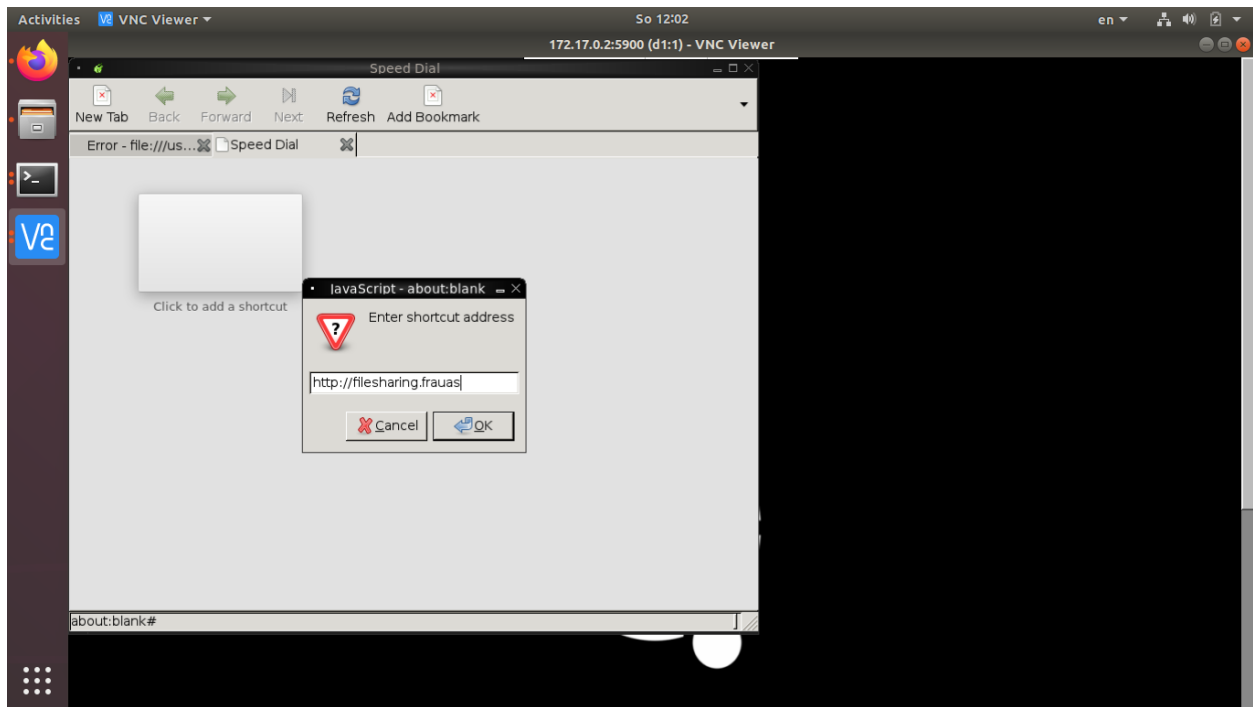


Figure 32 Accessing file sharing service from "d1"

- Next check slicing of the network's service

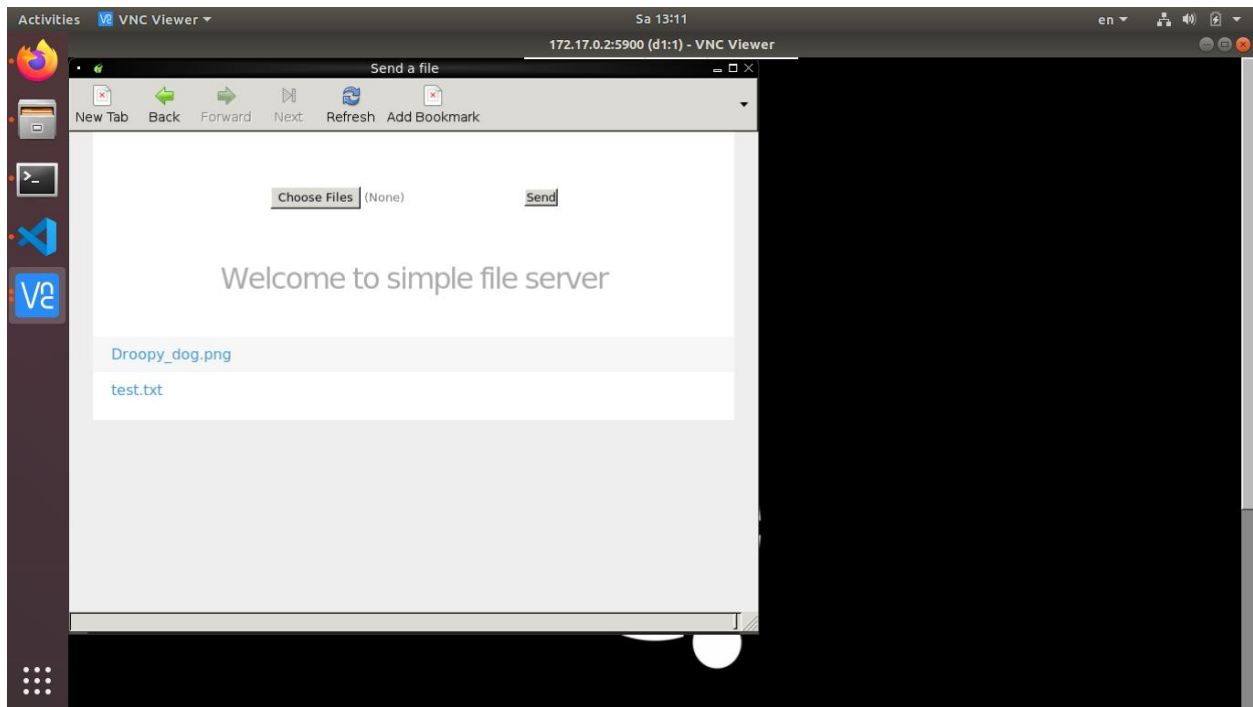


Figure 33 "d1" accesses file service

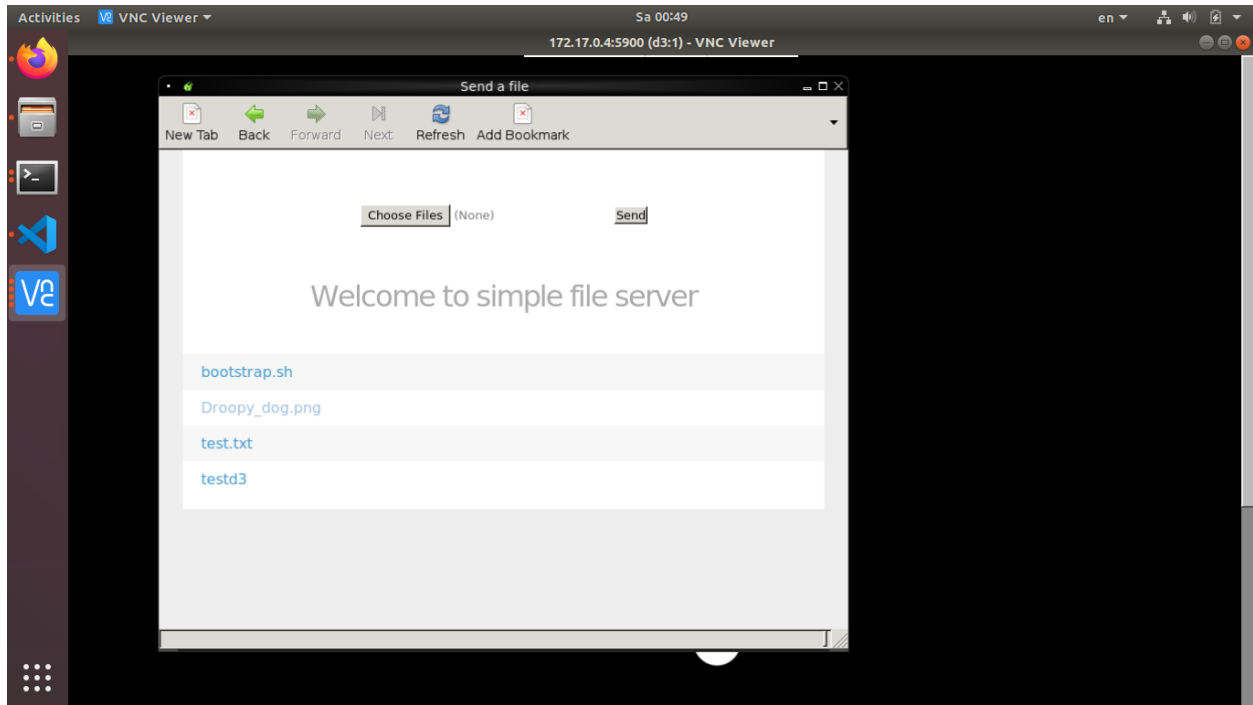


Figure 34 "d3" accesses file service and uploads files

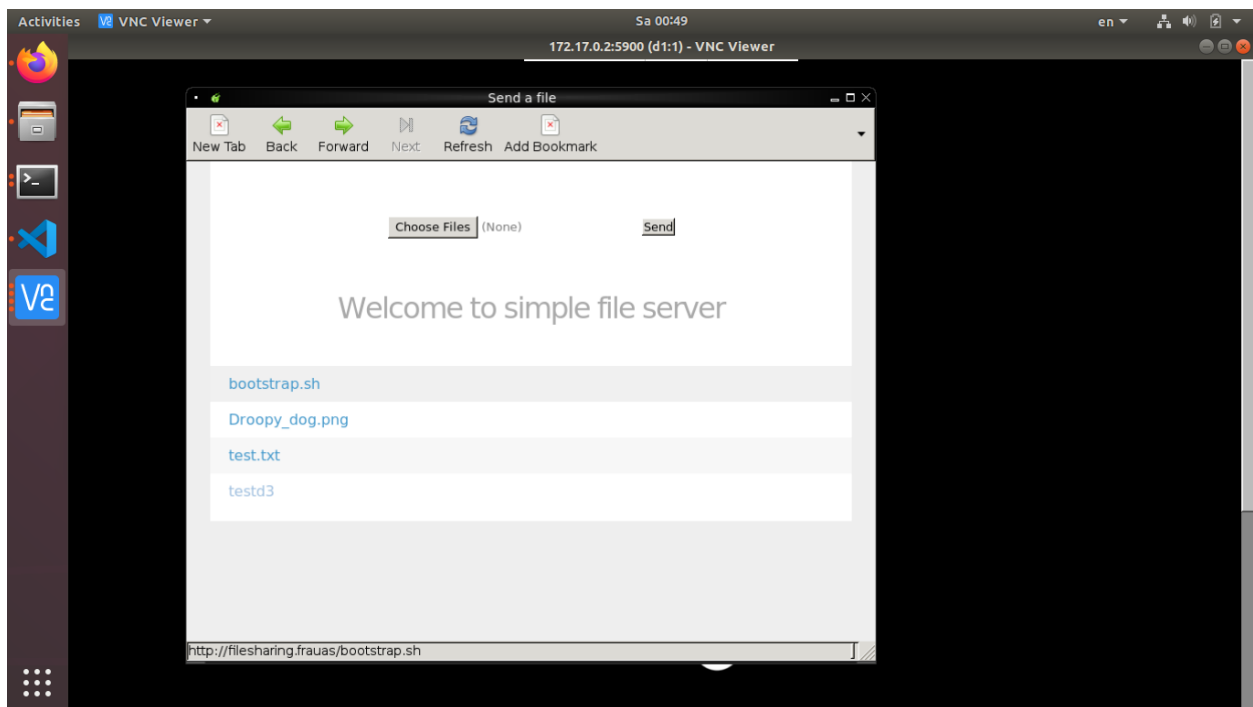


Figure 35 "d1" reloads the site and notices the changes

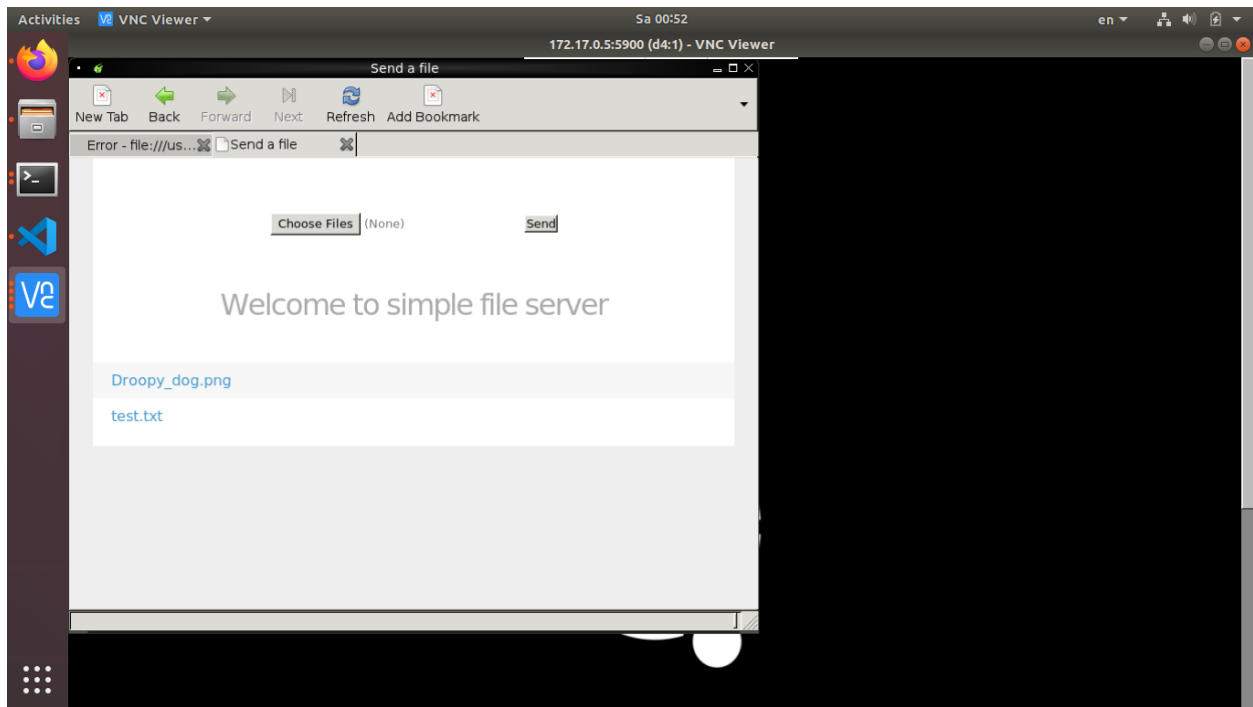


Figure 36 "d4" does not see any changes because it is served by a different server and has no access to service in the other slice

5. Discussion

5.1. Problems encountered during developing phase

During the developing phase of the project, the following problems were presented:

1. So many choices, it took quite some time to search for an appropriate approach.
2. POX controller platform is still under development and does not have NSH support. The NSH header library used in this project is self-made and its function is very limited.
3. Many useful features of Mininet and OVS are not obvious and not easy to find.

5.2. Evaluation of the project's result

Before the evaluation of the project, some justifications are necessary. They are followed by the project's evaluation. And finally are some suggestion for further improvement of this project.

5.2.1. Some justifications

OVS switch is used but only as a forwarding node (no flow is set up on the OVS switch): it is possible to improve the performance of the controller by delegating traffic classification and forwarding to the OVS switch by setting up appropriate flows at the switches. Although we could statically set the flows for the switches using “ovs-ofctl” tool from terminal, it would be difficult to introduce the dynamic topology adaptation feature. The reason is that in order for the network to automatically adapt to the network topology without the need of any manual configuration at runtime, a mechanism for the controller to communicate and configure the OVS switches is necessary (OpenFlow protocol). However, the capability to create (remotely through OpenFlow) a flow, which can encapsulate packet in NSH is not part of the official OpenFlow specification (it is in fact an extension of OVS switch to OpenFlow 1.3). Therefore, POX controller platform (which only supports OpenFlow 1.0) does not have support for this feature and the idea cannot be realized in this project.

Using STP and layer 2 forwarding as routing strategy inside core network: there are two main reasons why we decided to use only layer 2 forwarding in routing of core network. The first reason is, we do not focus on routing and forwarding aspect of the network in our project, the main concern is network slicing. The second reason is, the packet capturing software Wireshark can only parse NSH packets that is encapsulated inside Ethernet header. The testing phase becomes easier with Wireshark so we decided to use layer 2 as the transport layer of NSH traffic. Anyway, the routing strategy inside core network could be easily altered, since the core network and access networks are almost independent.

Tested network topology is small and simple: The developing and testing phase was carried out on a laptop so the available computing resource was constrained. Moreover, during developing, many other tools must also be in use e.g. Wireshark, code editor, web browser, containers,... thus further reduced the available resource.

5.2.2. Project evaluation

Following is the evaluation of the project's result:

- Network slicing based on NSH is implemented.

- Extension for simple dynamic network topology scenario is implemented. However, sometimes, it takes time for the controller to adapt to the new topology.
- File sharing service is integrated into the sliced network.

5.2.3. Further suggestion on improvement

Some possible improvements that can be made in this project are:

- Using efficient routing strategy inside core network (built-in RIP function of POX controller, using another controller with layer 3 routing for core network e.g. Floodlight,...).
- Current NSH header library only support MD Type 2 header with default length 8 bytes. It could be extended to a full-fledged library support for NSH. This will open the possibility for smooth integration of new service into the network.
- The structure of the controller's source code is quite messy and needs to be cleaned and refactored thoroughly.
- NSH-related as well as Service Function Chain-related logical components in this project (Classifier, Service Function Forwarder, NSH Forwarding Table) is very simple and not implemented as module components.
- DNS function of the controller is truly plain and ad-hoc. A well-designed DNS server would be a good improvement.

6. Conclusion

This project serves only as a quick prototyping of the network slicing approach based on NSH. The utilized tools in this project are Containernet (extension of Mininet), POX controller and simple open source file sharing server Droopy.

The network slicing mechanism in this project is carried out mainly through the introduction of SDN controller into the access network. Many implemented elements are very simplified and only offer very specific functions required in this project.

The most crucial project requirements are satisfied. In the defined network topology of this project, the designed controller is able to enforce network slicing based on NSH. It also has the ability to adapt to simple topology changes. The file sharing service is integrated into the sliced network.

Many further developments are still needed for the improvement in reliability of the implemented sliced network.

Reference

- [1] Quinn, P., Ed., Elzur, U., Ed., and C. Pignataro, Ed., "Network Service Header (NSH)", RFC 8300, DOI 10.17487/RFC8300, January 2018, <<https://www.rfc-editor.org/info/rfc8300>>.
- [2] Stackp, "stackp/Droopy," GitHub. [Online]. Available: <https://github.com/stackp/Droopy>. [Accessed: 06-Mar-2021].
- [3] B. Lantz et al., "mininet/mininet," GitHub. [Online]. Available: <https://github.com/mininet/mininet>. [Accessed: 06-Mar-2021].
- [4] M. Peuster, H. Karl, and S. v. Rossem: MeDICINE: Rapid Prototyping of Production-Ready Network Services in Multi-PoP Environments. IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN), Palo Alto, CA, USA, pp. 148-153. doi: 10.1109/NFV-SDN.2016.7919490. (2016)
- [5] "Production Quality, Multilayer Open Virtual Switch," Open vSwitch. [Online]. Available: <https://www.openvswitch.org/>. [Accessed: 06-Mar-2021].
- [6] MurphyMc, "noxrepo/pox," GitHub. [Online]. Available: <https://github.com/noxrepo/pox/tree/fangtooth>. [Accessed: 06-Mar-2021].
- [7] "Midori Browser," Astian, 25-Jan-2021. [Online]. Available: <https://astian.org/en/midori-browser/>. [Accessed: 06-Mar-2021].
- [8] S. Nguyen, "SangNguyen-97/mobilecomputing-frauas," GitHub. [Online]. Available: <https://github.com/SangNguyen-97/mobilecomputing-frauas/tree/main>. [Accessed: 06-Mar-2021].
- [9] "Mininet Topology Visualizer," Mininet Topology Visualizer - Online Tool. [Online]. Available: <http://mininet.spear.narmox.com/>. [Accessed: 06-Mar-2021].
- [10] "RealVNC," Download VNC Viewer for Linux | VNC® Connect. [Online]. Available: <https://www.realvnc.com/en/connect/download/viewer/linux/>. [Accessed: 07-Mar-2021].