Distributed Software-Defined Networks in Highspeed Amateur radio Multimedia NETwork

Mamdouh Muhammad
Communication Networks Group
Master of Science in Communications and Signal Processing
P.O. Box 100565, D-98684 Ilmenau, Germany
mamdouh.muhammad@tu-ilmenau.de
Supervisor: M.Sc. Matthias Aumüller

***Abstract*—Software-Defined Networks (SDN) as technology has attracted a lot of interest in both industry and academy life. In SDN, the centralization of the control plane of a network simplifies managing and troubleshooting problems of a network. Highspeed Amateur radio Multimedia NETwork (HAMNET) [6] [3] [10] is an independent internet network consists of sites which are connected in a mesh way. These sites are using wireless links and operating mainly on 5GHz band to provide a network for radio communications and other services like data-storage and mail servers.**
Implementation of distributed SDN in HAMNET is an approach proposed to be used in HAMNET in a distributed manner.

***Index Terms*—SDN, HAMNET**

I. Introduction

The traditional networks lack filling the gap of handling the rising traffic load being processed nowadays. The vast traffic load is due to new technologies like cloud computing, Big data, and Internet of Things (IoT). The management of a decentralized network as in the traditional networks is hard, and it takes time to configure each device in a network. SDN, as a technology of separation of the control plane from the data plane, is being used to overcome the problems that exist in traditional networks easily. As in SDN, you can control your network from a single point. SDN is being investigated a lot in both academic and industry fields to prove its feasibility for current and future systems.
In legacy SDN, we use only one controller to control the whole network based on the collected data from the control plane. In opposite, in distributed SDN, we use multi-domains, in each domain, there is a controller.
One challenge in HAMNET is to deliver high QoS [2] as we can, as it mainly a voice-oriented network. Using legacy - centralized - SDN in HAMNET will lead to colossal traffic loads congestion on the uplinks to the one controller scenario, which also will lead to a notable delay.
For these reasons, using distributed SDN is being proposed as a solution. In distributed SDN, we have multiple connected domains - sites - and every domain contains one controller and many access points which operate in bridge mode.
Another issue to consider is HAMNET is to search for a suitable routing protocol to route traffic between two sites.
Because HAMNET sites are connected via wireless, which is strongly affected by the weather. So to overcome this problem, we should use a routing protocol to record all available routes and forward the data if needed to the best route in the routing table based on different metrics that differ from a routing protocol to other.

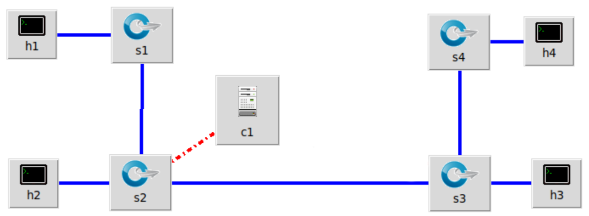

Figure 1. One-controller scenario

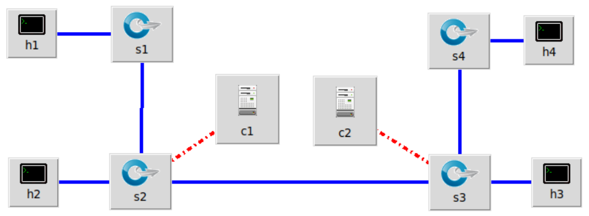

Figure 2. Multi-controller scenario

II. Legacy SDN vs Distributed SDN

A. Validate the drawbacks of legacy SDN and introducing distributed SDN

As shown in fig. 1, we used Mininet [13] as an emulator for SDN networks, and to plot this topology, we used Miniedit [8]. In this figure, we can see the SDN network contains one controller, four OpenFlow switches, and four hosts. As depicted and mentioned earlier, if C1 - the controller goes down, the whole system will breakdown.
In fig. 2, we used two controllers, so in case of an s2-s3 link failure, there will be two independent domains, and a controller is serving each domain. And if one controller goes down, the other domain members will remain in contact with each other.

Although this solution seems to be valid, but it lacks providing backup links, which can be done by using a routing protocol that can search for another available link based on routing information and will send the data through the best link.

III. RouteFlow

In order to emulate the behaviour of distributed SDN networks, we are using an open source project called RouteFlow [4] [5].
RouteFlow is being used to give the capability to run ip routing on any hardware that support OpenFlow. In general, in any RouteFlow scenario, as shown in fig. 3 there are three roles to be considered :

1- ***RF Controller*** behaves as the proxy for OpenFlow and distribute the traffic from and to the virtual machines
2- ***RFServer*** regulates the accessible virtual machines and combines the virtual machines with the corresponding OpenFlow switches.
3- ***RFClient:*** translates the Linux IP tables (routing table) into OpenFLow entries.

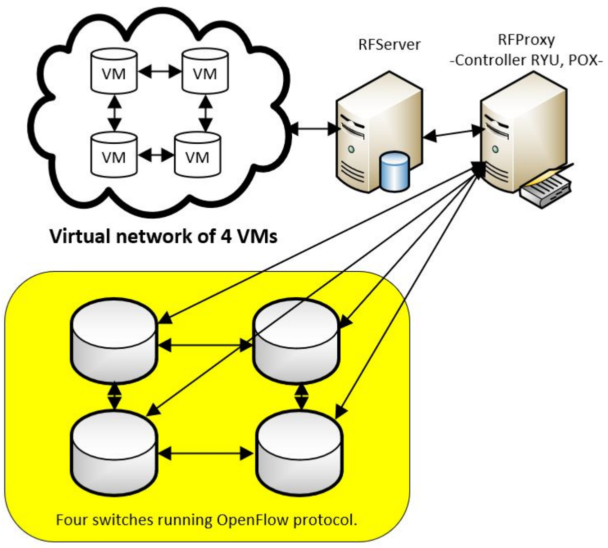

Figure 3. RouteFlow architecture

In the previously mentioned four virtual machines, we are using Quagga as a routing software to provide the routing capability.
In the four OpenFlow enabled switches, we are using Mininet as software to emulate the OpenFlow switches behavior.
The goal of RouteFlow is to run IP routing implemented in the four virtual machines over the four OpenFlow switches emulated by Mininet.
Then it is required that if the connectivity between two domains is down due to link failure, using routing capability, it is possible to find another link to provide the connectivity again.
As an RFProxy, we are using the RYU controller which is a python-written GUI-support single model controller that supports OpenFlow 1.0 to 1.5 [1] in addition to many other network protocols.

HAMNET domain is shown in fig. 3, in which there are one controller and many access points.
In our scenario, as shown in fig. 4, we will use many domains to enable routing between them and determine the system behavior in case of link failure.

Using a PC with specifications mentioned in Table I, We created four domains connected using one of the four OpenFlow switches, and in every domain, we are using two virtual machines with the specifications mentioned in Table II.

Table I
Host specifications to emulate our topology

	CPU	RAM	Hard Disk
value	I7 4910QM	16G	512 SSD

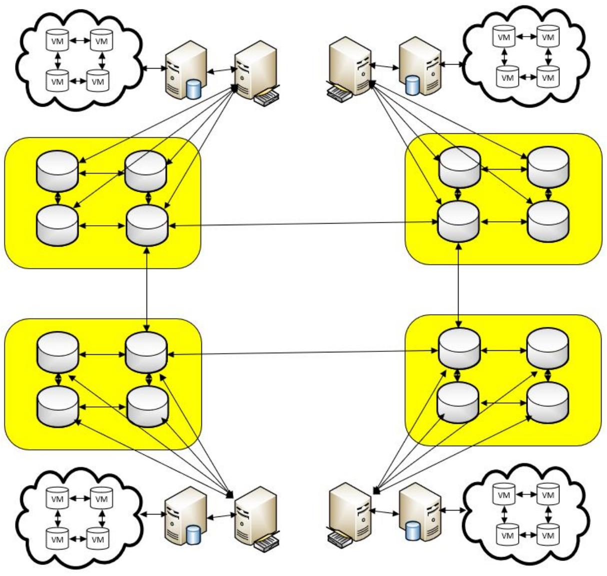

Figure 4. Our topology to emulate distributed SDN in HAMNET

Table II
One domain virtual machines Specifications

	Role	OS	virtual machine IP
VM 1	RouteFlow	ubuntu 12.04	10.0.2.4
VM 2	Mininet and RYU	ubuntu 14.04	10.0.2.5

In VM 1, we installed RouteFlow, using following steps:
1- Install Ubuntu 12.04
2- Install the dependencies using

```
sudo apt-get install build-essential git
libboost-dev libboost-program-options-dev
libboost-thread-dev libboost-filesystem-dev
iproute-dev openvswitch-switch mongodb
python-pymongo
```

3- Clone to RouteFlow:

```
git clone git://github.com/routeflow/RouteFlow.git
```

4- using following commands to build rfclient

```
cd RouteFlow; make rfclient
```

5- using following commands to create the LXC containers to work as virtual machines

```
cd rftest; sudo ./create
```

then we run **rftest2** file using the following command:

```
sudo ./rftest2
```

rftest2 file is a bash script file containing commands to Setup the management bridge and MongoDB, and to start the four virtual machines, the RFProxy, RFServer and the control plane network.

Now in virtual machine 2, we will run the following command to start our four OpenFlow switches using Mininet emulator.

```
sudo mn --custom mininet/custom/topo-4sw-4host.py
--topo=rftest2 --controller=remote,ip=10.0.2.4
,port=6633 --pre=ipconf
```

here from last command, we can see that we are using the **topo-4sw-4host.py** as a python file containing the commands to create the four OpenFlow switches in Mininet and we are using **ipconf** file which contains commands to be executed inside Mininet, these commands are used to add default gateways to the four hosts.

The content of **topo-4sw-4host.py** file as following:

```
"""
Four switches connected in mesh topology
plus a host for each switch:
       h1 --- sA ---- sB --- h2
               |   \     |
               |    \    |
               |     \   |
               |      \  |
       h3 --- sC ---- sD --- h4
"""
from mininet.topo import Topo

class rftest2(Topo):
    "RouteFlow Demo Setup"

 def __init__( self, enable_all = True ):
        "Create custom topo."

        Topo.__init__( self )

    h1 = self.addHost("h1",
        ip="172.31.1.100/24",
        defaultRoute="gw 172.31.1.1")

    h2 = self.addHost("h2",
        ip="172.31.2.100/24",
        defaultRoute="gw 172.31.2.1")

    h3 = self.addHost("h3",
        ip="172.31.3.100/24",
        defaultRoute="gw 172.31.3.1")

    h4 = self.addHost("h4",
        ip="172.31.4.100/24",
        defaultRoute="gw 172.31.4.1")

        sA = self.addSwitch("s5")
        sB = self.addSwitch("s6")
        sC = self.addSwitch("s7")
        sD = self.addSwitch("s8")

        self.addLink(h1, sA)
        self.addLink(h2, sB)
        self.addLink(h3, sC)
        self.addLink(h4, sD)
        self.addLink(sA, sB)
        self.addLink(sB, sD)
        self.addLink(sD, sC)
        self.addLink(sC, sA)
        self.addLink(sA, sD)


topos = { 'rftest2': ( lambda: rftest2() ) }
```

After executing previous commands, we can check for connectivity using **pingall** command we will get the following result confirming that every host can reach other hosts is right.

```
h1 -> h2 h3 h4
h2 -> h1 h3 h4
h3 -> h1 h2 h4
h4 -> h1 h2 h3
*** Results: 0% dropped (12/12 received)
```

to assure our topology we can use **net** command to see current network structure:

```
h1 h1-eth0:s5-eth1
h2 h2-eth0:s6-eth1
h3 h3-eth0:s7-eth1
h4 h4-eth0:s8-eth1
s5 lo:  s5-eth1:h1-eth0 s5-eth2:s6-eth2
        s5-eth3:s7-eth3 s5-eth4:s8-eth4
s6 lo:  s6-eth1:h2-eth0 s6-eth2:s5-eth2
        s6-eth3:s8-eth2
s7 lo:  s7-eth1:h3-eth0 s7-eth2:s8-eth3
        s7-eth3:s5-eth3
s8 lo:  s8-eth1:h4-eth0 s8-eth2:s6-eth3
        s8-eth3:s7-eth2 s8-eth4:s5-eth4
c0
```

As seen above from the command result, we have four hosts, and four switches. Every host is connected to only one switch and all switches are interconnected.
To see Open short Path First (OSPF) protocol routing entries that has been converted by RFClient to OpenFlow records we can use **dpctl dump-flows** command.

To understand how routing is done, the content of the **rftest** folder is shown in fig. 5

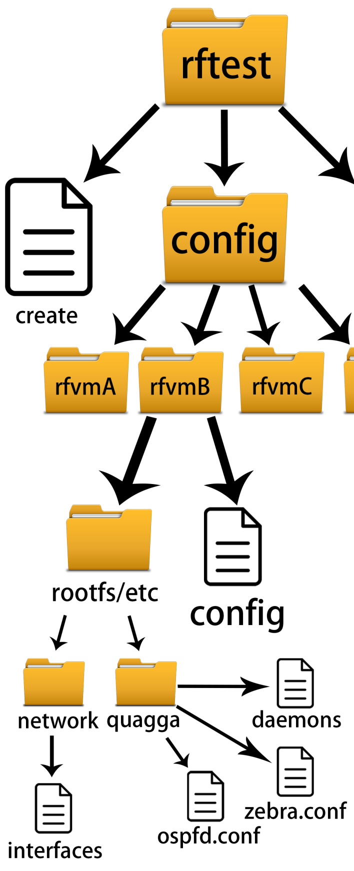

Figure 5. rftest folder structure

Following is the description of these files:
1- **create** file is to create virtual machines as previously mentioned.
2- **rftest2** file is to run start virtual machines, the RFProxy, the RFServer and the control plane network as previously mentioned.
3- **config** folder contain the configurations for the four virtual machines A, B, C and D.
4- **config** file to configure the ports and Media Access Control (MAC) address for every port (e.g. rfvmA.1, rfvmA.2 , .. etc).
5- **rootfs/etc** folder containing network configurations.
6- **interfaces** file containing the IP address for eth0 management interface. 7- **quagga** folder containing three important files regardingIP routing services.
8- **daemons** file determines which daemon to activate (ospfd in our case).
9- **ospfd.conf** file which contain the OSPF routing protocol configurations.
10- **zebra.conf** contains IP to switch interface mapping and zebra log file, for example in case of rfvmB:

```
log file /var/log/quagga/zebra.log

interface eth1
        ip address 172.31.2.1/24

interface eth2
        ip address 10.0.0.2/24

interface eth3
        ip address 40.0.0.2/24
```

In **ospfd.conf** file, you can configure networks and the area that every network belongs to. In addition, you can configure the hello and dead intervals for every interface.

Till now we were just investigating the implementation of RouteFlow in just one domain -site-.
To achieve our scenario as shown in fig. 4, we should extend our topology by integrating another site as a site 2 and then see if using OSPF protocol will achieve connectivity between two sites.
We will use a preconfigured virtual machine called **vandervecken** which contains RouteFlow and all required services to run it successfully.
Virtual machines to be used are mentioned in Table III.

Furthermore, we will configure the following:
1- Add site 2 by editing the content of **Create** file to add four virtual machines (A2, B2, C2 and D2) and connect the newly added virtual machines (site 2) to site 1 through one link between B1-B2 as shown in fig. 6.
2- Add more four virtual machines to our topology by editing the content of **topo-4sw-4host.py** file and activate the corresponding ports in the config file.
3- Assign new MAC addresses to the new ports in the **config** file.
4- Add the ospf hello and dead intervals to it in the **ospfd.conf** file.
5- Add an IP addresses to the new ports in the **zebra.conf** file.

Table III
MULTI-DOMAIN VIRTUAL MACHINES SPECIFICATIONS

	Role	**OS**	**virtual machine IP**
VM 1 (site 1,2)	RouteFlow	ubuntu 12.04	10.0.2.6
VM 2	Mininet and RYU	ubuntu 14.04	10.0.2.5

IV. RESULTS

After doing the following steps:
1- Replicating the two sites shown in 6 so we have four sites

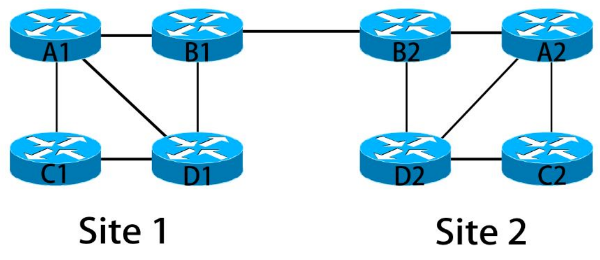

Figure 6. Two sites topology

connected as we want in 4.
2- Check connection between two sites by using **ping -t** command , then while pinging, we will use **traceroute** command to see the route between the two sites.
3- We will drop the link that carry the data of ping traffic between the two site.
4- We will notice that the ping traffic will pass through the other route that was the lowest cost route - and the only available route in our case - in the routing table.

As a result We can achieve a valid connection between any two sites using OSPF routing protocol.
Challenging now is simulate a large topology which will require high specifications laptop. Simulating a large topology is beneficial to measure how much average delay, maximum delay and load jitter we will happen in OSPF convergence phase especially in case of vast traffic.

V. Conclusion

Distributed SDN is a promising approach to be used in WWAN.
RouteFlow appeared to be a valid distribued SDN application to connect legacy IP routing protocols and any programmable physical switch [9] and can be used in one of the WWAN approaches (HAMNET).

Next to be in the future researches is to investigate the behaviour of other routing protocols and compare between all to find the most appropriate one to HAMNET.

Also investigating the reaction of other SDN controllers like ONOS, OpenDayLight, POX and NOX.
Other IP routing protocols can be investigated which are either dynamic IP routing protocols or static IP routing protocols. Dynamic IP routing protocols means nodes are moving. Here nodes are static but changing the link frequently is behaving as if nodes are moving.

Automatic Custom Generation of Topologies and Configuration of Routing protocols in SDN approach [12]can be used to simulate OSPF, Border Gateway Protocol (BGP) (interior iBGP and exterior eBGP) protocols. To measure the behaviour of any IP routing protocols, future works must simulate a full real scenario, in which, there is inter and intra-domains communications, more traffic to be simulated, other controllers to be used and different IP routing protocols.

For ease of use, i strongly recommend using the following:
1- A preconfigured SDN virtual machine [7] that can save you a lot of time in installing mininet and ryu controller and many other controllers.
2- Vandervecken [11] as a preconfigured RouteFlow iso file that can be used directly either in hardware environment or in virtual environment.

References

[1] Esmaeil Amiri, Emad Alizadeh, and Khalilollah Raeisi. “An Efficient Hierarchical Distributed SDN Controller Model”. In: Feb. 2019, pp. 553–557. DOI: 10.1109/KBEI.2019.8734982.
[2] Fetia Bannour, Sami Souihi, and Abdelhamid Mellouk. “Distributed SDN Control: Survey, Taxonomy and Challenges”. In: *IEEE Communications Surveys Tutorials* PP (Dec. 2017), pp. 1–1. DOI: 10.1109/COMST.2017.2782482.
[3] bravo37. URL: http://www.bravo37.de/hamnet/.
[4] CPqD. URL: https://sites.google.com/site/routeflow/.
[5] CPqD. *RouteFlow*. URL: https://routeflow.github.io/RouteFlow/.
[6] hamnetdb. URL: https://hamnetdb.net/.
[7] SDN Hub®. *sdnhub*. URL: http://sdnhub.org/.
[8] Bob Lantz and Gregory Gee. URL: https://github.com/mininet/mininet/blob/master/examples/miniedit.py.
[9] Marcelo Nascimento et al. “The RouteFlow approach to IP routing services on software-defined networks”. In: (Mar. 2020).
[10] RouteFlow. URL: https://github.com/CPqD/RouteFlow.
[11] RouteFlow. *RouteFlow*. URL: https://github.com/routeflow/RouteFlow/wiki/vandervecken.
[12] Apoorv Shukla, Mengchen Shi, and Anja Feldmann. “Automatic Custom Generation of Topologies and Configuration of Routing Protocols in SDN”. In: *Proceedings of the SIGCOMM Posters and Demos*. SIGCOMM Posters and Demos ’17. Los Angeles, CA, USA: Association for Computing Machinery, 2017, pp. 3–5. ISBN: 9781450350570. DOI: 10.1145/3123878.3131966. URL: https://doi.org/10.1145/3123878.3131966.
[13] Mininet Team. URL: http://mininet.org/.