

CMPE 210 - SDN and NFV



**SAN JOSÉ STATE
UNIVERSITY**

Project Report

Improved Dijkstra's Algorithm Based Load Balancing SDN Application

Under guidance of Dr. Younghee Park

Group 09

Nirbhay Kumar Singh (SID - 010693917)

Tejaswi Goel (SID - 010698623)

Gunveet Singh Arora (SID - 010641904)

Abstract

The project extends the Dijkstra's algorithm and considers not only the edge weights but also the node weights and bandwidth of each network link. We used python to develop our load balancer application, Restful APIs and networkx module to gather all the network information mininet to generate our network topology and testing tools like Iperf, ping and wireshark to compare our load balancer with the original load balancer of floodlight..

Introduction

At present, network traffic is growing fast and complex as enterprises need to purchase more equipment to handle this complex network. The online services like e-commerce, websites, and social networks frequently use multiple servers to get high reliability and accessibility. Network congestion and server overload are the serious problems faced by the enterprises network. IP services uses Open Shortest Path First(OSPF), the computation is based on Dijkstra's algorithm which calculates the shortest path within the network with disadvantage is only edge weights is considered which makes it less efficient. In our extended dijkstra's based load balancer we consider not only the edge weights but also the node weights and bandwidth.

Tools and Technologies used

Extensive efforts, research and active development in the field of SDN has given rise to many effective and easy to use tools for developing and testing SDN applications. We were fortunate to have multiple options to select from. Here are the tools which we have used in the process of development and testing of this project.

Mininet

Mininet can be most accurately described as a Network Simulation system which is used for interpretation. It runs a ton of things out of which some include switches, links and routers. The mininet is responsible for using virtualization techniques which can make a system to appear as if it is a whole new complete network. It acts in a way that is similar to a real machine. We can perform all the basic tasks that we perform on a real machine to a mininet. The difference basically lies in the fact that instead of physical hardware, mininet is made using virtualized software (Bob Lantz). Mininet is known for its various features and advantages some of which i am going to list here:

- Mininet starts up in a matter of few seconds.
- Basically anything that requires a linux operating system can be run on Mininet
- We can create various custom built topologies on mininet.
- One of the biggest advantages of using mininet is that it is an Open Source platform hence anyone can improve it and use it for free.
- We only need simple and easy python scripts to run them on mininet.

GRE tunnel and VxLAN tunnel

GRE tunnel is basically a tunneling protocol used for enclosing a lot of protocols that come under a point to point link network. This protocol is mostly deployed in between gateways or in some cases or from a gateway to an end station. We encrypt the data sent between the tunnel to secure the data. We usually use a GRE tunnel protocol if the data is sent through an insecure network and the risk is there.

VxLAN on the other hand uses UDP. VxLAN may look similar to GRE tunnel but there are plenty of differences between the two. In case of VxLAN we have the concept of uplink and downlink. The uplink is at the receiving end of the VxLAN frames. These VxLAN frames are routed through an IP Protocol (Brent Salisbury,2012).

Floodlight

To say simply, Floodlight is an SDN controller. A group of developers and experts work on it to improve it and increase it's efficiency. This is an open source software that is available to everyone. Over the time we have seen improvements in floodlight and it going to version v1.2 with all the improvements from Floodlight version v1.1. One of the main Purposes of Floodlight controller are making instructions and controlling the rules on how the traffic has to be controlled (Ryan Izard). The main advantages of using the floodlight controller are:

- 1) Scalability : The ability of the floodlight controller to manage more and more switches, links and routers at any given instant is more than what we currently use.
- 2) Versatility: Floodlight controller can easily merge with whatever environment is currently in use and can also support traditional networks that may use Non-OpenFlow switches (sdxcentral.com).

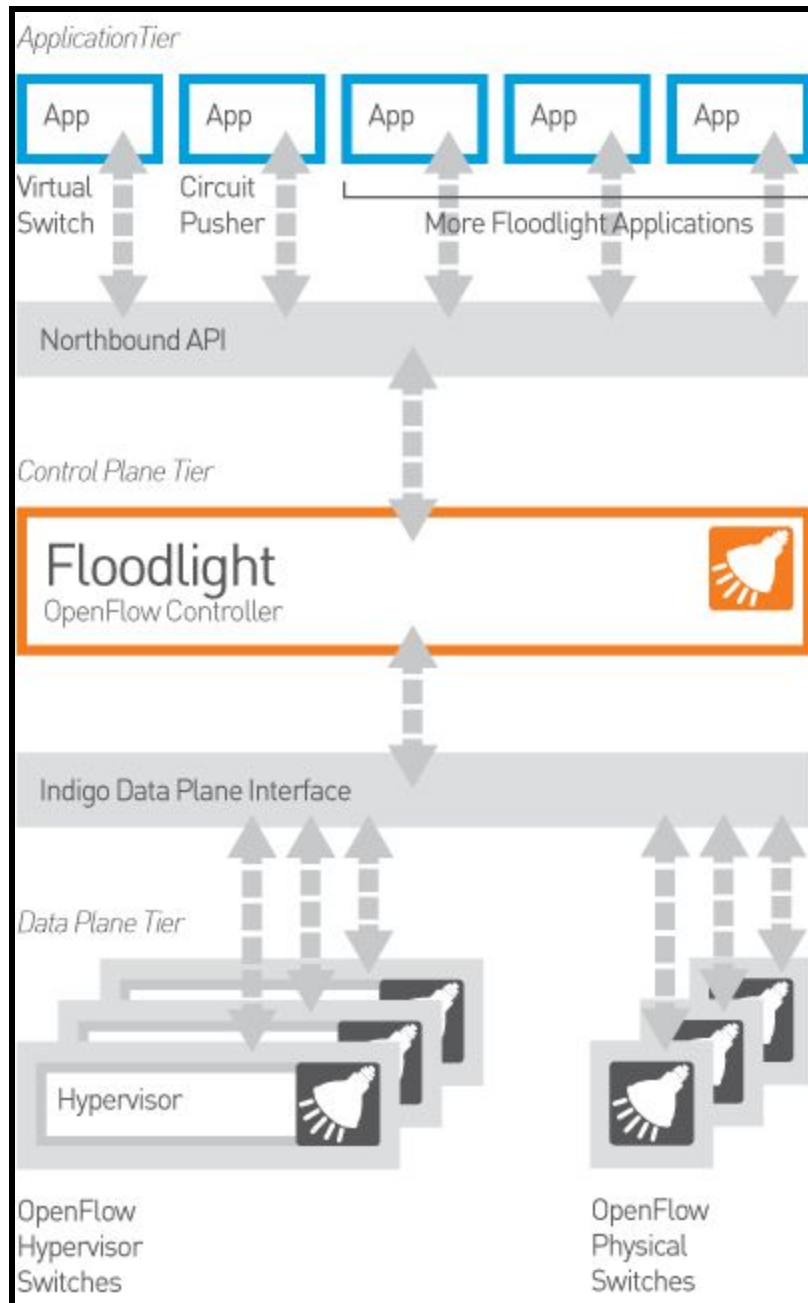


Fig 1: Floodlight controller architecture

Northbound APIs

Northbound API are used in SDN to communicate with protocols that are above the level the protocol that is trying to communicate is at. It basically is for the component of that protocol. The same goes for southbound protocol too. Northbound API is one of the most important API of the SDN environment. “They are the most nebulous component of the SDN environment” (sdxcentral.com). However, since the start of these, Northbound API’s have changed a huge deal. They have to support a wide variety of applications. “SDN Northbound APIs are also used to integrate the SDN Controller with automation stacks”

Iperf

A simple networking tool designed to test the UDP and the TCP protocol so that we can get into the details about Network Bandwidth, delay and data loss. We take a look at the characteristics of the TCP and UDP and change them in such a manner as to run tests and know more about these protocols. We punch in the commands to know what specific performance evaluation we are interested in. Now we also have Iperf 3 which is not compatible with the original Iperf (Chris Partsenidis).

Dijkstra’s Algorithm

In simple words, Dijkstra Algorithm is finding the shortest path between two nodes, which in context of networks can be two hosts connected through a mesh of Switches and links. In our case we extend the Dijkstra’s algorithm and compare it with the original to give the analysis in the form of graphs and figures. Here we have a figure of the dijkstra’s algorithm between two nodes. (Wikipedia.org)

Implementation details

Infrastructure and Topology Setup

One of the crucial part of the project was to choose a suitable topology for clear demonstration of load balancing. But even then we wanted to test our load balancing module with a more challenging infrastructure. So, we joined two different topologies created in two different mininet VMs with GRE tunnel and VxLAN tunnel. Here are the steps followed to create the mininet topology.

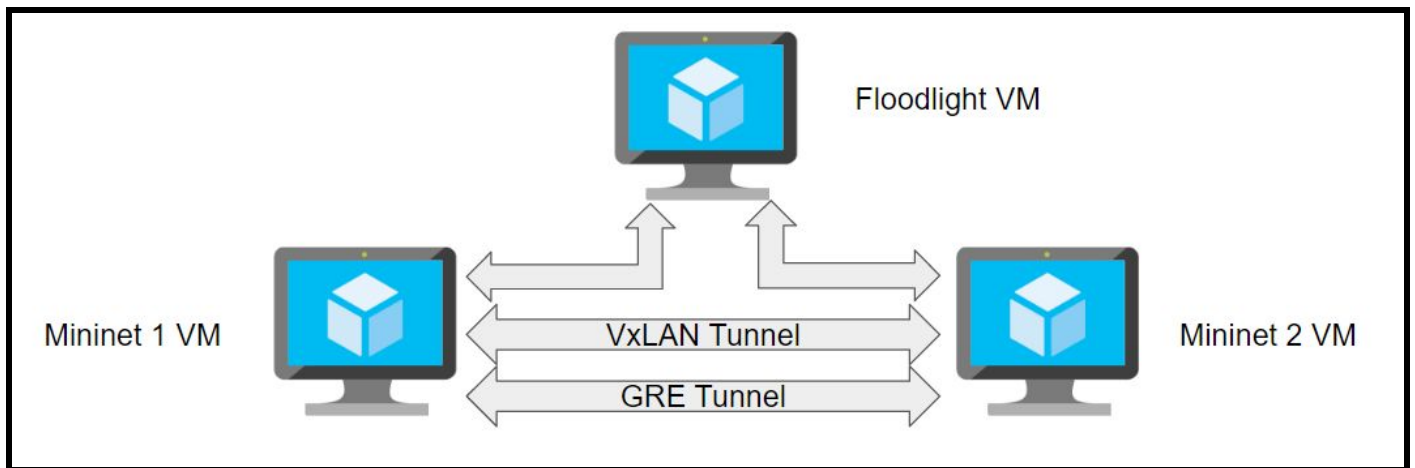


Fig 2: The system architecture

Step 1. Created the two topologies in two different mininet VMs by editing a custom topology python file. The python file was easy to edit as it contained straight forward API calls like `addLink()`, `addHost()`, `addSwitch()`. The function of each API was as expect congruent with their names.

- `addLink()` - This API was used to create a link between either two switches or one switch and one host.
- `addHost()` - This API was used to add hosts to the network. While adding a host, it was also possible to assign a static IP address to the host.
- `addSwitch()` - This API was used to add a switch to the network.

Step 2. Then run the mininet command to create the topology from the custom python file. Mention the IP address of the floodlight VM in both the command for both the mininet topology creation. For us, the IP address of the Floodlight VM was 192.168.56.101 and the name of custom topology python file in both the VMs is *topology4.py*. Following is the command to create the custom topology in both the mininet VMs.

```
sudo mn --custom topology4.py --topo mytopo--controller=remote,ip=192.168.56.101,port=6653
```

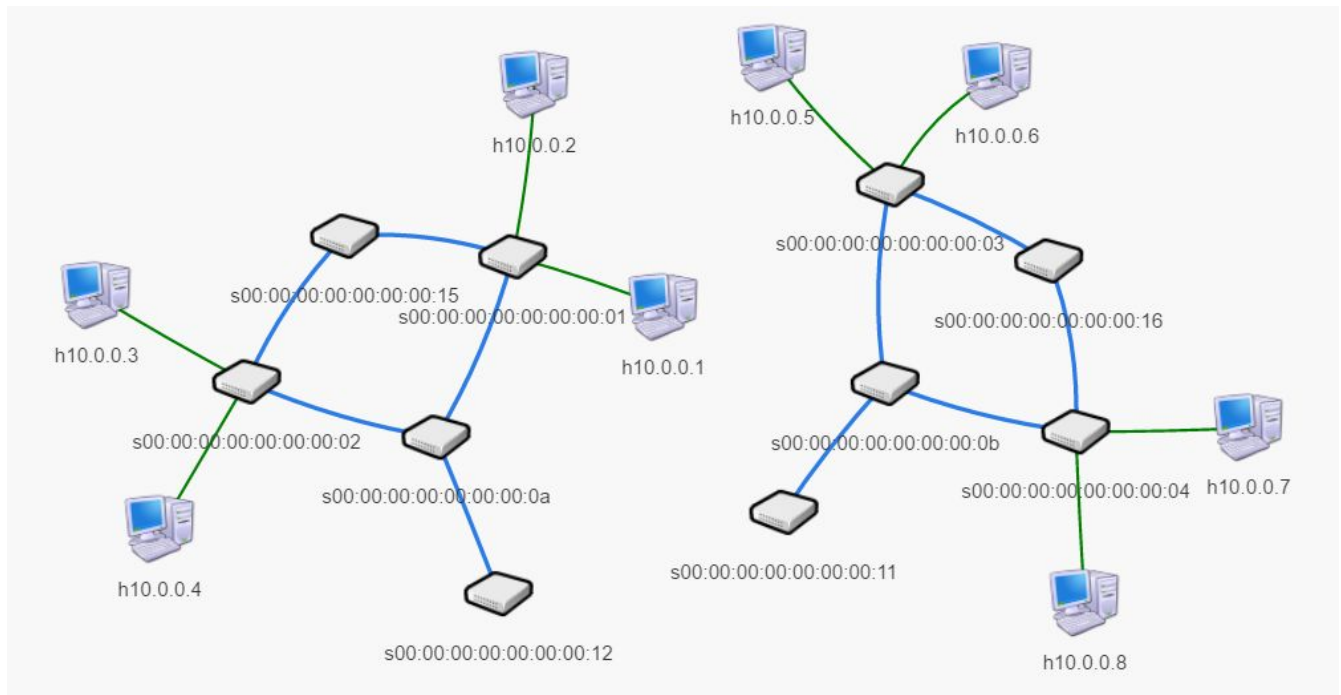


Fig 3: Topology in Floodlight UI without GRE and VxLAN tunnel

Step 3. Once both the VMs are connected to the Floodlight VM, the next step is to create a GRE tunnel and VxLAN tunnel to connect both the topologies. Here are the commands to do so:

- GRE tunnel - The IP address of Mininet_1 VM is 192.168.56.102 and of Mininet_2 VM is 192.168.56.103. The connection is established between s21 of Mininet_1 to s17 of the Mininet_2.
 - *sh ovs-vsctl add-port s21 hello2 -- set interface hello2 type=gre option:remote_ip=192.168.56.103*
 - *sh ovs-vsctl add-port s17 hello2 -- set interface hello2 type=gre option:remote_ip=192.168.56.102*

- VxLAN tunnel - The IP address of Mininet_1 VM is 192.168.56.102 and of Mininet_2 VM is 192.168.56.103. The connection is established between s18 of Mininet_1 to s22 of the Mininet_2.
- sh ovs-vsctl add-port s18 hello1 -- set interface hello1 type=vxlan option:remote_ip=192.168.56.103
- sh ovs-vsctl add-port s22 hello1 -- set interface hello1 type=vxlan option:remote_ip=192.168.56.102

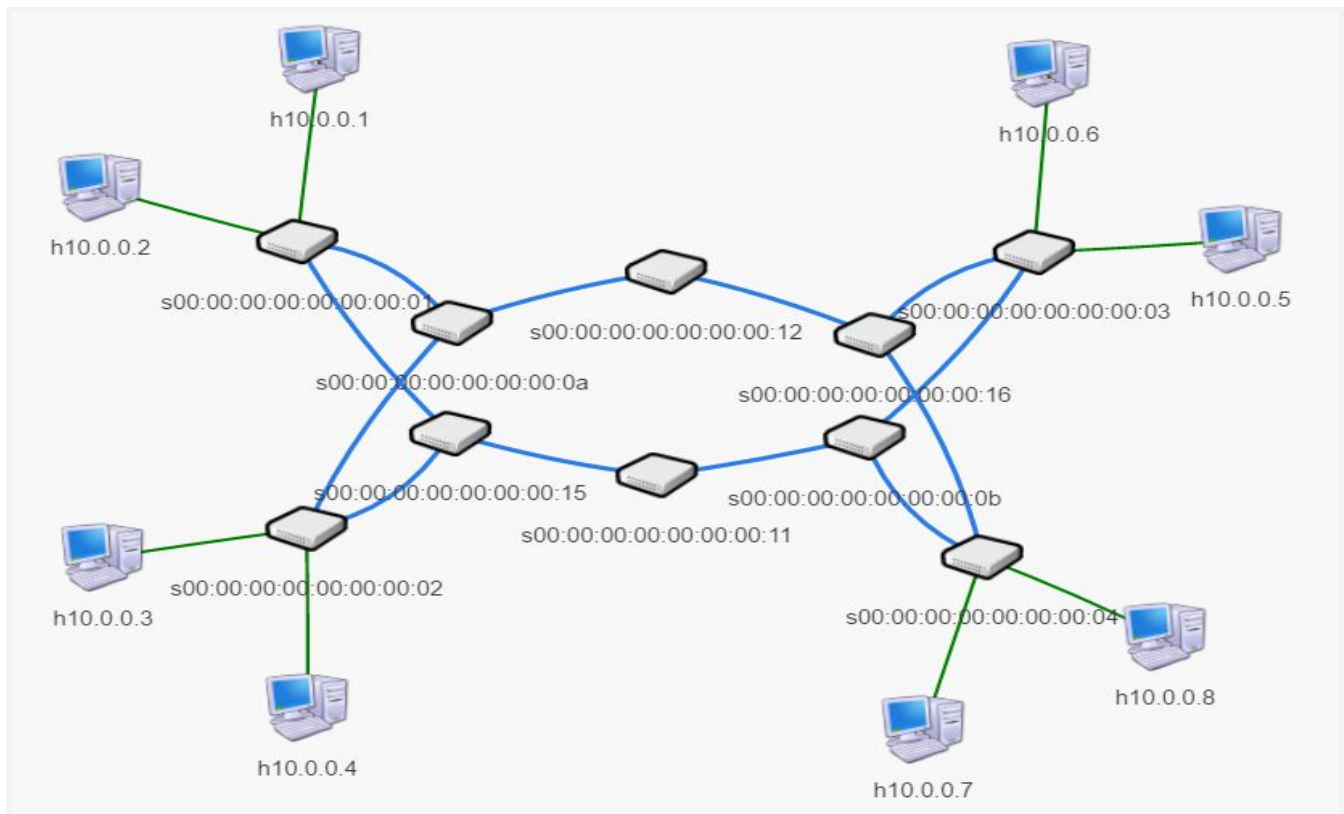


Fig 4: Topology in Floodlight UI with GRE and VxLAN tunnel

Flow of Implementation

We have implemented the load balancing module as an application which interacts with the controller through RESTful (Representational State Transfer) API calls to collect data about the underlying network. Following is a step by step process of development and testing we have followed:

- First the Floodlight VM and both the Mininet VMs are started. Different static IP addresses are assigned to all the VMs. In our case the assigned IP addresses are:

- Floodlight VM - 192.168.56.101
- Mininet_1 VM - 192.168.56.102
- Mininet_2 VM - 192.168.56.103
- Then the desired topology is created in both the Mininet VMs and connected through GRE and VxLAN tunnels.
- The current state and data regarding the topology is collected in JSON format using the API interface provided by floodlight controller.
- Then meaningful data is extracted from the JSON data pool for further calculation



```
[
  {
    "dpid": "00:00:00:00:00:00:00:0b",
    "port": "2",
    "updated": "Fri Dec 02 01:15:14 PST 2016",
    "link-speed-bits-per-second": "10000000",
    "bits-per-second-rx": "60",
    "bits-per-second-tx": "60"
  }
]
```

Fig 5: Sample API response which is used in Bandwidth scheme

```

"00:00:00:00:00:00:00:0b": {
  "flows": [
    {
      "version": "OF_13",
      "cookie": "0",
      "table_id": "0x0",
      "packet_count": "1128",
      "byte_count": "84588",
      "duration_sec": "5314",
      "duration_nsec": "629000000",
      "priority": "0",
      "idle_timeout_s": "0",
      "hard_timeout_s": "0",
      "flags": [],
      "match": {},
      "instructions": {
        "instruction_apply_actions": {
          "actions": "output=controller"
        }
      }
    }
  ]
},

```

Fig 6: Sample API response which is used in Node weight / Edge weight scheme

```

{
  "src-switch": "00:00:00:00:00:00:00:04",
  "src-port": 3,
  "dst-switch": "00:00:00:00:00:00:00:0b",
  "dst-port": 2,
  "type": "internal",
  "direction": "bidirectional",
  "latency": 15
},

```

Fig 7: Sample API response which is used in Node weight / Edge weight scheme

- The meaningful data is provided to the routing module which calculates the shortest possible route from one host to another with the help of Dijkstra's algorithm.

- The routing module gives the list of shortest path from one host to another.
- Then the load balancing module will find out the path which has least transmission cost with help of either Bandwidth scheme or Node weight / Edge weight scheme.
- The Flow creator module prepare a static flow rules.
- The flow rules obtained from the Flow creator module is pushed into the routers using the API interface provided by floodlight controller.
- At last we compare the results from three scenarios with help of Iperf test and Wireshark test:
 - Firstly, with inbuilt Dijkstra's algorithm
 - Secondly, with Bandwidth scheme, and
 - Thirdly, with Node weight / Edge weight scheme

Implemented methods of load balancing

There are many methods and parameters that can be used to perform load balancing. But, through this project, we have tried to implement two different scheme of load balancing i.e Bandwidth scheme and Node weight / Edge weight scheme.

Bandwidth Scheme

In this scheme of load balancing we have successfully improved the routing performance of the network compared to the default load balancing module performance. We do so by adding all the transmission rates of all the output port of all the switches in a link which gives the total end to end transmission rates (totalTx) of the shortest paths. Then we find the path with maximum totalTx which becomes the best current path.



Fig 8: Sample API response which gives the transmission rate of a switch

Node Weight / Edge Weight Scheme

In this scheme of load balancing we have successfully improved the routing performance of the network compared to the default load balancing module performance. Here I define terms which is very useful in this scheme.

- The *Node weight* is defined as the amount of time a switch takes to process a certain number of packets (i.e. Latency of switch).
- The *Edge weight* is defined as the amount of time a link takes to transmit a certain number of packets (i.e. Latency of links).
- Therefore, the *End-to-End latency* is equal to the sum of total Node weights and total Edge weights.

Using these

The sample API calls used to calculate the Node weight and Edge weight is as follows:

```
"00:00:00:00:00:00:0b": {
  "flows": [
    {
      "version": "OF_13",
      "cookie": "0",
      "table_id": "0x0",
      "packet_count": "1128",
      "byte_count": "84588",
      "duration_sec": "5314",
      "duration_nsec": "629000000",
      "priority": "0",
      "idle_timeout_s": "0",
      "hard_timeout_s": "0",
      "flags": [],
      "match": {},
      "instructions": {
        "instruction_apply_actions": {
          "actions": "output=controller"
        }
      }
    }
  ]
},
```

Fig 9: Sample API call to calculate the Node weight

From the figure above, a sample calculation of Node weight (considering 100 bytes of data to be standard) is as follows:

Amount of seconds required to calculate 100 byte of data = $100 * (\text{"duration_sec"} / \text{"byte_count"})$

```
{
  "src-switch": "00:00:00:00:00:00:04",
  "src-port": 3,
  "dst-switch": "00:00:00:00:00:00:0b",
  "dst-port": 2,
  "type": "internal",
  "direction": "bidirectional",
  "latency": 15
},
```

Fig 10: Sample API call to calculate the Node weight

Results and Outcomes

We have tested the load balancing application in two ways. Firstly we have tested the performance of the application by computing the bandwidth of the system in three different scenarios. First being load balancing with default load balancer module in floodlight, second being load balancing with Bandwidth scheme and third being the load balancing with Node weight / Edge weight scheme.

Iperf Test Results

We have used Iperf to determine bandwidth of our system in every scheme. For standard results we have always checked bandwidth between h1 and h7. Where, h1 is set as Iperf server and h7 is set as Iperf client. The bandwidth result is appeared on the client side.


```
xterm
root@sdnhubvm:~/mininet/custom[07:15] (master)$ iperf -s
-----
Server listening on TCP port 5001
TCP window size: 85,3 KByte (default)
-----
[  4] local 10.0.0.1 port 5001 connected with 10.0.0.7 port 42789
█
```

Fig 11: Setting up of Iperf server

```
root@sdnhubvm:~/mininet/custom[02:40] (master)$ iperf -c 10.0.0.1
-----
Client connecting to 10.0.0.1, TCP port 5001
TCP window size: 85,3 KByte (default)
-----
[  3] local 10.0.0.7 port 45076 connected with 10.0.0.1 port 5001
[ ID] Interval      Transfer    Bandwidth
[  3] 0.0-960.0 sec  63,6 KBytes  543 bits/sec
root@sdnhubvm:~/mininet/custom[02:57] (master)$ iperf -c 10.0.0.1
-----
Client connecting to 10.0.0.1, TCP port 5001
TCP window size: 85,3 KByte (default)
-----
[  3] local 10.0.0.7 port 45077 connected with 10.0.0.1 port 5001
[ ID] Interval      Transfer    Bandwidth
[  3] 0.0-939.4 sec  63,6 KBytes  555 bits/sec
root@sdnhubvm:~/mininet/custom[03:14] (master)$ iperf -c 10.0.0.1
-----
Client connecting to 10.0.0.1, TCP port 5001
TCP window size: 85,3 KByte (default)
-----
[  3] local 10.0.0.7 port 45078 connected with 10.0.0.1 port 5001
[ ID] Interval      Transfer    Bandwidth
[  3] 0.0-1029.0 sec  66,5 KBytes  529 bits/sec
root@sdnhubvm:~/mininet/custom[03:36] (master)$ iperf -c 10.0.0.1
-----
Client connecting to 10.0.0.1, TCP port 5001
TCP window size: 85,3 KByte (default)
-----
[  3] local 10.0.0.7 port 45559 connected with 10.0.0.1 port 5001
[ ID] Interval      Transfer    Bandwidth
[  3] 0.0-939.4 sec  63,6 KBytes  555 bits/sec
root@sdnhubvm:~/mininet/custom[11:51] (master)$ iperf -c 10.0.0.1
-----
Client connecting to 10.0.0.1, TCP port 5001
TCP window size: 85,3 KByte (default)
-----
[  3] local 10.0.0.7 port 45560 connected with 10.0.0.1 port 5001
[ ID] Interval      Transfer    Bandwidth
[  3] 0.0-1028.9 sec  66,5 KBytes  529 bits/sec
root@sdnhubvm:~/mininet/custom[12:23] (master)$ █
```

Fig 12: Iperf client results for default load balancing scheme

The average for the default load balancing scheme was computed to be **542.2 bits / sec**

```
root@sdnhubvm:~/mininet/custom[12:23] (master)$ iperf -c 10.0.0.1
-----
Client connecting to 10.0.0.1, TCP port 5001
TCP window size: 85.3 KByte (default)
-----
[ 3] local 10.0.0.7 port 45561 connected with 10.0.0.1 port 5001
[ ID] Interval      Transfer    Bandwidth
[ 3] 0.0-1028.7 sec  66.5 KBytes  529 bits/sec
root@sdnhubvm:~/mininet/custom[12:57] (master)$ iperf -c 10.0.0.1
-----
Client connecting to 10.0.0.1, TCP port 5001
TCP window size: 85.3 KByte (default)
-----
[ 3] local 10.0.0.7 port 45562 connected with 10.0.0.1 port 5001
[ ID] Interval      Transfer    Bandwidth
[ 3] 0.0-939.5 sec   63.6 KBytes  555 bits/sec
root@sdnhubvm:~/mininet/custom[13:22] (master)$ iperf -c 10.0.0.1
-----
Client connecting to 10.0.0.1, TCP port 5001
TCP window size: 85.3 KByte (default)
-----
[ 3] local 10.0.0.7 port 45563 connected with 10.0.0.1 port 5001
[ ID] Interval      Transfer    Bandwidth
[ 3] 0.0-935.1 sec   63.6 KBytes  557 bits/sec
root@sdnhubvm:~/mininet/custom[13:50] (master)$ iperf -c 10.0.0.1
-----
Client connecting to 10.0.0.1, TCP port 5001
TCP window size: 85.3 KByte (default)
-----
[ 3] local 10.0.0.7 port 45564 connected with 10.0.0.1 port 5001
[ ID] Interval      Transfer    Bandwidth
[ 3] 0.0-935.1 sec   63.6 KBytes  557 bits/sec
root@sdnhubvm:~/mininet/custom[14:20] (master)$ iperf -c 10.0.0.1
-----
Client connecting to 10.0.0.1, TCP port 5001
TCP window size: 85.3 KByte (default)
-----
[ 3] local 10.0.0.7 port 45565 connected with 10.0.0.1 port 5001
[ ID] Interval      Transfer    Bandwidth
[ 3] 0.0-939.1 sec   63.6 KBytes  555 bits/sec
root@sdnhubvm:~/mininet/custom[15:51] (master)$
```

Fig 13: Iperf client results for load balancing with Bandwidth scheme

The average for the load balancing with Bandwidth scheme was computed to be 550.6 bits / sec

```

root@sdnhubvm:~/mininet/custom[15:51] (master)$ iperf -c 10.0.0.1
-----
Client connecting to 10.0.0.1, TCP port 5001
TCP window size: 85,3 KByte (default)
-----
[ 3] local 10.0.0.7 port 50618 connected with 10.0.0.1 port 5001
[ ID] Interval      Transfer    Bandwidth
[ 3] 0.0-935.2 sec  63,6 KBytes  557 bits/sec
root@sdnhubvm:~/mininet/custom[16:12] (master)$ iperf -c 10.0.0.1
-----
Client connecting to 10.0.0.1, TCP port 5001
TCP window size: 85,3 KByte (default)
-----
[ 3] local 10.0.0.7 port 50619 connected with 10.0.0.1 port 5001
[ ID] Interval      Transfer    Bandwidth
[ 3] 0.0-943.2 sec  63,6 KBytes  553 bits/sec
root@sdnhubvm:~/mininet/custom[16:50] (master)$ iperf -c 10.0.0.1
-----
Client connecting to 10.0.0.1, TCP port 5001
TCP window size: 85,3 KByte (default)
-----
[ 3] local 10.0.0.7 port 50620 connected with 10.0.0.1 port 5001
[ ID] Interval      Transfer    Bandwidth
[ 3] 0.0-939.4 sec  63,6 KBytes  555 bits/sec
root@sdnhubvm:~/mininet/custom[17:24] (master)$ iperf -c 10.0.0.1
-----
Client connecting to 10.0.0.1, TCP port 5001
TCP window size: 85,3 KByte (default)
-----
[ 3] local 10.0.0.7 port 50621 connected with 10.0.0.1 port 5001
[ ID] Interval      Transfer    Bandwidth
[ 3] 0.0-939.1 sec  63,6 KBytes  555 bits/sec
root@sdnhubvm:~/mininet/custom[17:41] (master)$ iperf -c 10.0.0.1
connect failed: No route to host
root@sdnhubvm:~/mininet/custom[20:09] (master)$ iperf -c 10.0.0.1
-----
Client connecting to 10.0.0.1, TCP port 5001
TCP window size: 85,3 KByte (default)
-----
[ 3] local 10.0.0.7 port 58314 connected with 10.0.0.1 port 5001
[ ID] Interval      Transfer    Bandwidth
[ 3] 0.0-931.3 sec  63,6 KBytes  560 bits/sec
root@sdnhubvm:~/mininet/custom[20:36] (master)$

```

Fig 14: Iperf client results for load balancing with NW/EW scheme

The average for the load balancing with NW / EW scheme was computed to be 556 bits / sec

Wireshark Test Results

In wireshark test we essentially try to test if the packets are actually travelling through the best shortest path, i.e. if the flow rules are being pushed successfully. For this example we have taken two examples. In first example the source host is h1 and destination is h7. We see that there is no TCP or UDP traffic in the alternate path.

High Availability Cluster

A controller failure can quickly paralyze the entire network, the issue of control plane HA is critical as controllers are responsible for the functions of network switches. In linux the High availability project promotes reliability, availability and serviceability.

The Linux-HA main product is Heartbeat, a GPL-licenced portable cluster management program for high-availability clustering. Other Famous HA tools are, Corosync and Pacemaker.

Heartbeat Features:

1. A heartbeat signal or message at regular intervals to finds if cluster nodes working correctly.
2. Heartbeat program runs specialized scripts automatically, supporting two or more nodes clusters.

HA setup steps:

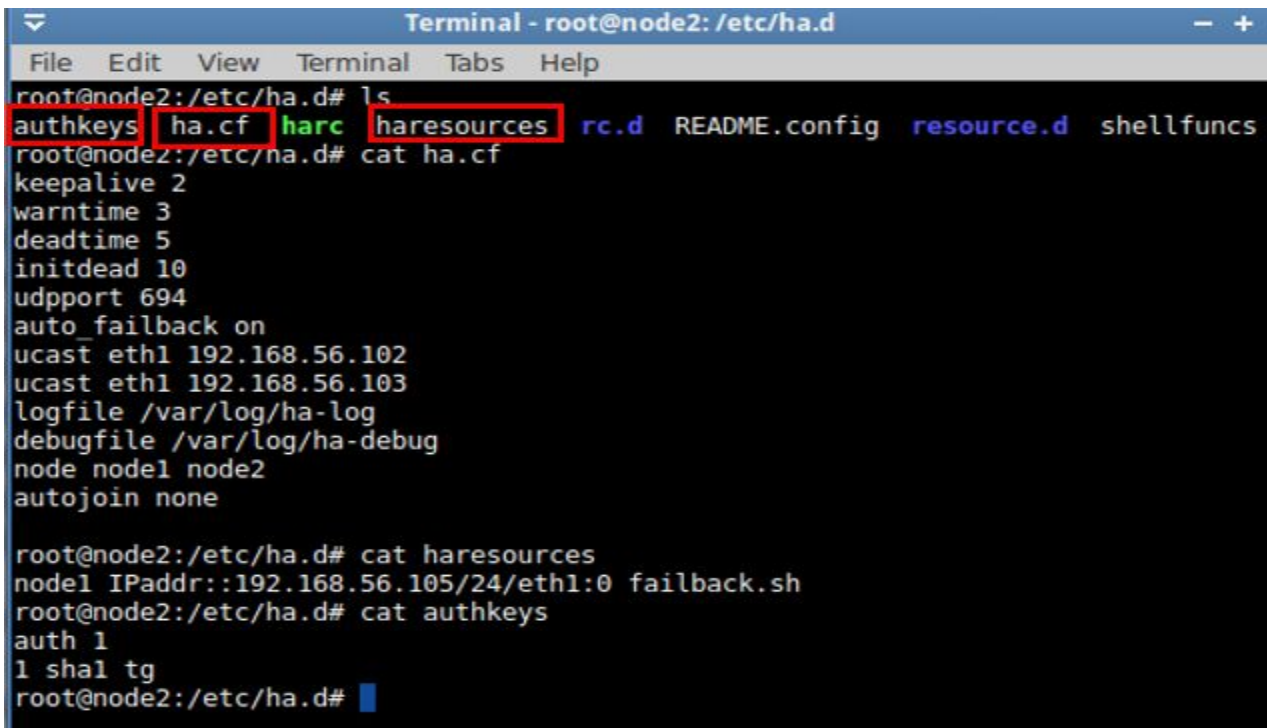
Pre-configuration Requirements:

1. Set the VM's network adapter as host only/ Bridged adapter.
2. Assign hostname to primary node with IP address to all the cluster nodes.
3. To confirm the node is dead a script called STONITH(shoot the other node in the head) is used.

Configuring heartbeat, Heartbeat has three main configuration files (on both nodes exactly same files):

1. /etc/ha.d/authkeys: The authkeys file is owned by root and chmod 600. It has two lines; auth directive with an associated method ID number and the other line is the authentication method and the key that go with the ID number of the auth directive. The three supported authentication methods are: crc, mds and sha1.
2. /etc/ha.d/ha.cf: ha.cf is the main configuration file defines the main heartbeat vairables.
 - a. Keepalive: the time between heartbeat in seconds.
 - b. Deadtime: time to wait without hearing from a cluster member before declaring it dead.
 - c. Warntime: time to wait before issuing "late hearbeat" warning.

- d. `Auto_failback`: can be set to either on or off, if set to on; after failure if primary node comes back than secondary node falls back to secondary state. If set to off, after the primary comes back, it will be secondary.
 - e. `/var/log/ha-log`: this stores all the heartbeat logs and can be used for troubleshooting when warning occurs.
 - f. List all the nodes in the cluster (use `uname -n` to find the node name)
3. `/etc/ha.d/haresources`: this files consists of the node name that will act as primary node, VIP, and the resource script. Resource script is the startup script that run the necessary commands before the heartbeat starts its operation.

A terminal window titled "Terminal - root@node2: /etc/ha.d" showing the contents of heartbeat configuration files. The user runs 'ls' and lists files: authkeys, ha.cf, haresources, rc.d, README.config, resource.d, and shellfuncs. 'ha.cf' and 'haresources' are highlighted with red boxes. Then, the user runs 'cat ha.cf' and 'cat haresources' to display their contents. The 'ha.cf' file shows settings like keepalive, warntime, deadtime, initdead, udpport, auto_failback, ucast, logfile, debugfile, node, and autojoin. The 'haresources' file shows a node configuration with IPaddr, failback.sh, and auth keys.

```
root@node2:/etc/ha.d# ls
authkeys  ha.cf  haresources  rc.d  README.config  resource.d  shellfuncs
root@node2:/etc/ha.d# cat ha.cf
keepalive 2
warntime 3
deadtime 5
initdead 10
udpport 694
auto_failback on
ucast eth1 192.168.56.102
ucast eth1 192.168.56.103
logfile /var/log/ha-log
debugfile /var/log/ha-debug
node node1 node2
autojoin none

root@node2:/etc/ha.d# cat haresources
node1 IPaddr::192.168.56.105/24/eth1:0 failback.sh
root@node2:/etc/ha.d# cat authkeys
auth 1
1 sha1 tg
root@node2:/etc/ha.d#
```

Fig 12: The heartbeat configuration files.

Commands in Heartbeat:

1. To start heartbeat: `/etc/init.d/heartbeat start`
2. To stop heartbeat: `/etc/init.d/heartbeat stop`

3. To check the logs: `tail -f /var/log/messages`

Testing SDN Floodlight HA cluster:

1. For two member cluster, start both the clusters. Use `ifconfig` and `host` to check if proper ip address and hostname are assigned.
2. Now start the heartbeat service on all nodes, you will see an alias IP on one of the nodes. That's the primary node you had set in the `haresources` file.

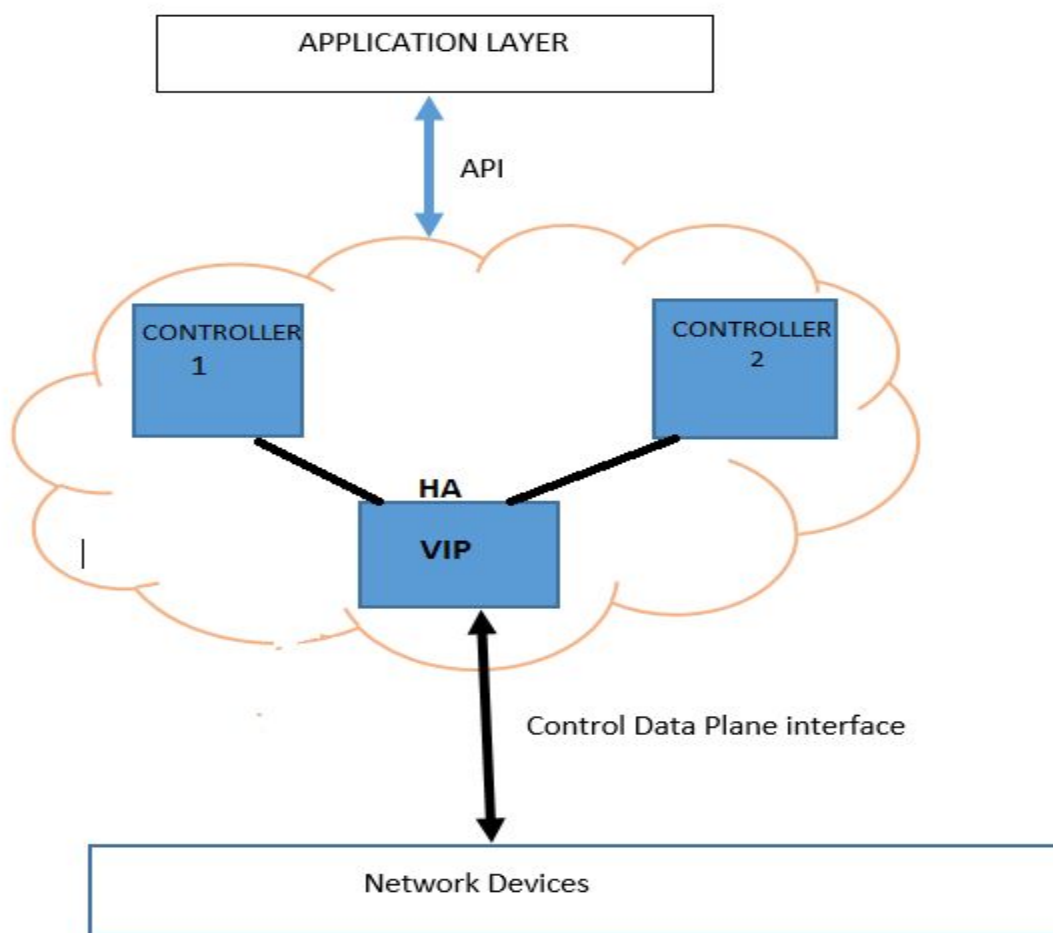


Fig 12: The HA VIP

3. To test failover, stop the floodlight and heartbeat service on primary node. You will see an alias ip comes up in the secondary node (test with ifconfig command). And the alias IP from primary is removed automatically as it is down.
4. To test failback, start again the primary node floodlight controller and heartbeat service, An alias IP will automatically will come on it, and the floodlight controller will still work with alias IP.

Conclusion

1. Load Balancer: In the original load balancer of floodlight only the edge weight is considered, we further changed its code to consider the node weight and bandwidth. After testing and comparing the results we found out that our load balancer design outperforms the original load balancer of the floodlight.
2. The working of HA in the floodlight controllers cluster has widened our scope of research to the scaling of control plane and replication of data plane.

References

Jehn-Ruey Jiang, Hsin-Wen Huang, Ji-Hau Liao, and Szu-Yuan Chen (2014). Extending Dijkstra's Shortest Path Algorithm for Software Defined Networking. Asia-Pacific Network Operation and Management Symposium (APNOMS)

Brent Salisbury (2012, Jul 2). Configuring VXLAN and GRE Tunnels on OpenvSwitch.
(<http://networkstatic.net/configuring-vxlan-and-gre-tunnels-on-openvswitch/>)

GRE and VXLAN encapsulation methods
(<http://akanda.io/tip-of-the-week/tip-5-gre-and-vxlan-encapsulation-methods/>)

What is the floodlight controller?
(<https://www.sdxcentral.com/sdn/definitions/sdn-controllers/open-source-sdn-controllers/what-is-floodlight-controller/>)

Floodlight (<http://www.projectfloodlight.org/floodlight/>)

Chris Partsenidis. What is iPerf and how is it used?
(<http://searchnetworking.techtarget.com/answer/What-is-iPerf-and-how-is-it-used>)

Dijkstra's Algorithm (https://en.wikipedia.org/wiki/Dijkstra's_algorithm)

What are SDN Northbound APIs?:(<https://www.sdxcentral.com/sdn/definitions/north-bound-interfaces-api/>)
HA:
(<https://www.digitalocean.com/community/tutorials/how-to-create-a-high-availability-setup-with-heartbeat-and-floating-ips-on-ubuntu-14-04>)