CSC/ECE 573: INTERNET PROTOCOLS
NC STATE UNIVERSITY

Secure client server communication in OpenFlow network

Final Report

https://sites.google.com/a/ncsu.edu/clientauthopenflow/

- **Team**

Name	**Unity ID**
➢ AbhishekBandarupalle	abandar
➢ Mahesh Agrawal	magrawa4
➢ Manushri	manush
➢ Sushma Poondru	spoondr

Under the guidance of Prof. Dr. RudraDutta

ACKNOWLEDGEMENT

We would like to express our gratitude to our mentor and professor Dr. Rudra Dutta for his valuable advice and constant support right from the beginning of the project. We have enjoyed learning and researching under his constant guidance.

We would also like to thank the teaching assistant Bharathkumar Ramachandra for his valuable guidance.

Table of Contents

1. Introduction
    - Description of problem

2. Components
    - Platforms for project
    - Areas for project
    - Major components decomposition

3. Design and development plan
    - High-level design
    - Low level design
    - Per-member responsibilities
    - Timeline

4. Verification and validation plan
    - Test application
    - Test plan
    - Test cases
    - Demo playbook

5. Self-Study
    - Test plan
    - Test cases

6. Reflection
    - Per-member learning experience
    - What went as expected
    - What went different from expected

7. Enhancement after interim report

8. Future scope

9. Submission Package

10. References

1. INTRODUCTION:

Description of problem

Most of the communications in the present day world require a secure connection. The objective of this project is to establish a secure connection by authenticating the clients before they can communicate with the application server. In this client server model, the client gets authenticated before initiating communication with application server. The client and server are unaware of the authentication. Openflow controller initially redirects the traffic sent by client for application server to the authenticator, where the authentication operations are performed. If the authentication is successful for the client, certain flow rules are pushed on to the switch which will allow traffic from client to server thereafter. In case of failure in authentication, the packets are dropped. The server receives connections only from the clients that are authenticated.

Our model is also efficient in avoiding DoS (Denial of Service) attacks. A denial of service attack is a malicious attempt to make a server unavailable to users, usually by temporarily interrupting or suspending the services of a host to connect. When an unauthorized client continuously tries to reach the server, the server is busy in authenticating and rejecting the connections from that host and hence will be unavailable to the other hosts trying to make a connection. Aseparate authenticator in the design helps in preventing any DoS (Denial of Service) attack on the network.

Using OpenFlow the forwarding plane operations are centralized and the clients' packets are redirected to authenticator before reaching the server. The mechanism uses the following components: Authenticator, OpenFlow Controller and an Application Server. POX Controller is used to run the OpenFlow application, Authenticator is used to authenticate the clients before contacting the server.

2. COMPONENTS:

2.1 Areas for project

The project relies on OpenFlow for redirection of flows. We are using POX as the OpenFlow controller for implementing our algorithm.

2.2 Platforms for project

- NetLabs
- Ubuntu 14.04 LTS
- Open vSwitch
- Python
- GENI (for testing operations before implementing in NetLabs)

2.3 Major Components Decomposition

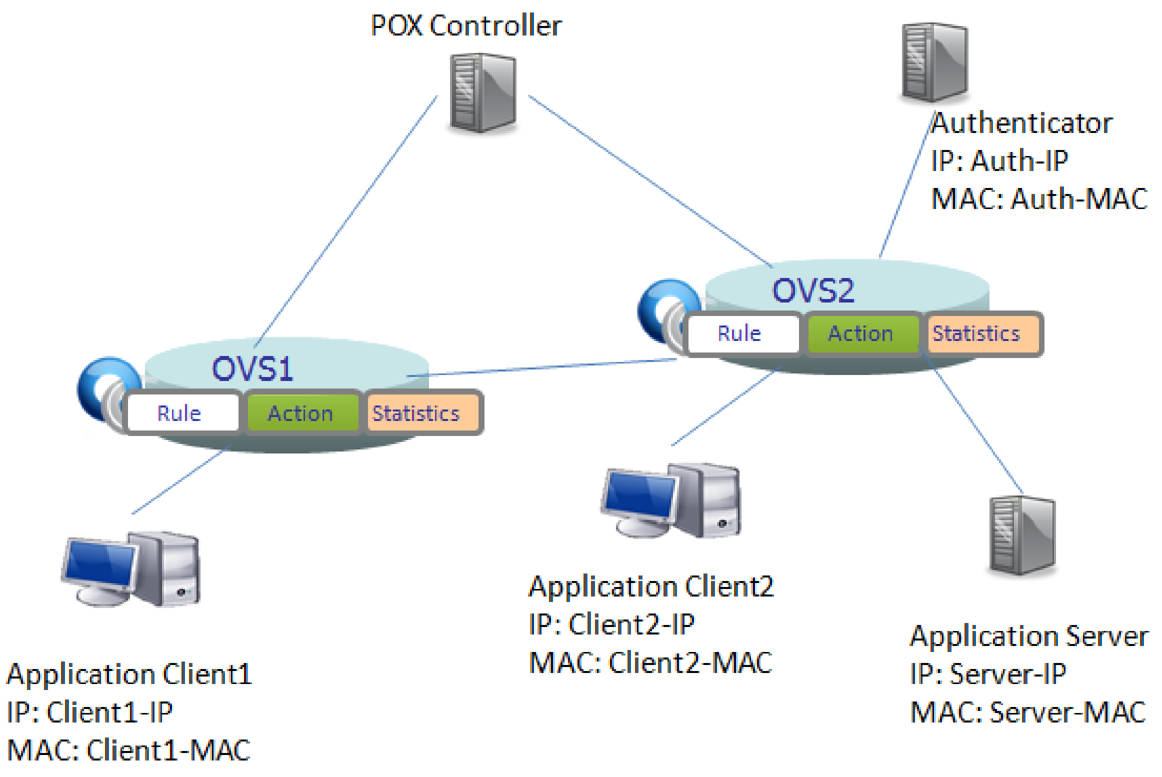
POX OpenFlow Controller: POX is an open source development platform that provides API's for Python-based Software-defined networking(SDN) control applications such as OpenFlow. POX components are additional Python programs that can be invoked when POX is started from the command line. These components implement the network functionality in the software defined network. POX comes with some stock components already available. Using these components we are making the controller in our project to push flow entries in to the flow tables of the OVS Switches based upon the response received by the Authenticator.

OpenvSwitch: Open vSwitch is a software implementation of a virtual multilayer network switch, designed to enable effective network automation. OVS Switches are used to interconnect all the devices (end-hosts, server, and authenticator) in our topology. The switch receives packet from the host application. It identifies the incoming packet by checking the flow table entries to take forwarding decision. If there is a match in the table then the packet is forwarded according to the flow rule. Any packet with no match in the flow table is forwarded to the POX controller connected to the switch. The controller finds routes, sends a flow entry to the switch and sends the packet back to the switch.

Authenticator: In this project the authenticator is used as a process that communicates with the application running over the POX controller. The application allows or denies network traffic and pushes the rules in the OpenFlow tables from a supplicant host depending on the result of authentication.

3. Design and Development Plan:

3.1 High-Level proposed Design:



The major functional components used in AuthFlow are:

- POX Controller
- OpenFlow vSwitches
- Authenticator
- Application Clients and Server

The above diagram depicts a basic topology that will be used to demonstrate this project. When an application client tries to establish contact with an Application Server, the packet is redirected to the authenticator, where it is checked if the client should be allowed to communicate with the server. If the client credentials are valid then the Authenticator sends a response back to the controller to insert flow rules in the OVS Switches, so that thereafter the client will be able to contact server without being redirected for authentication.

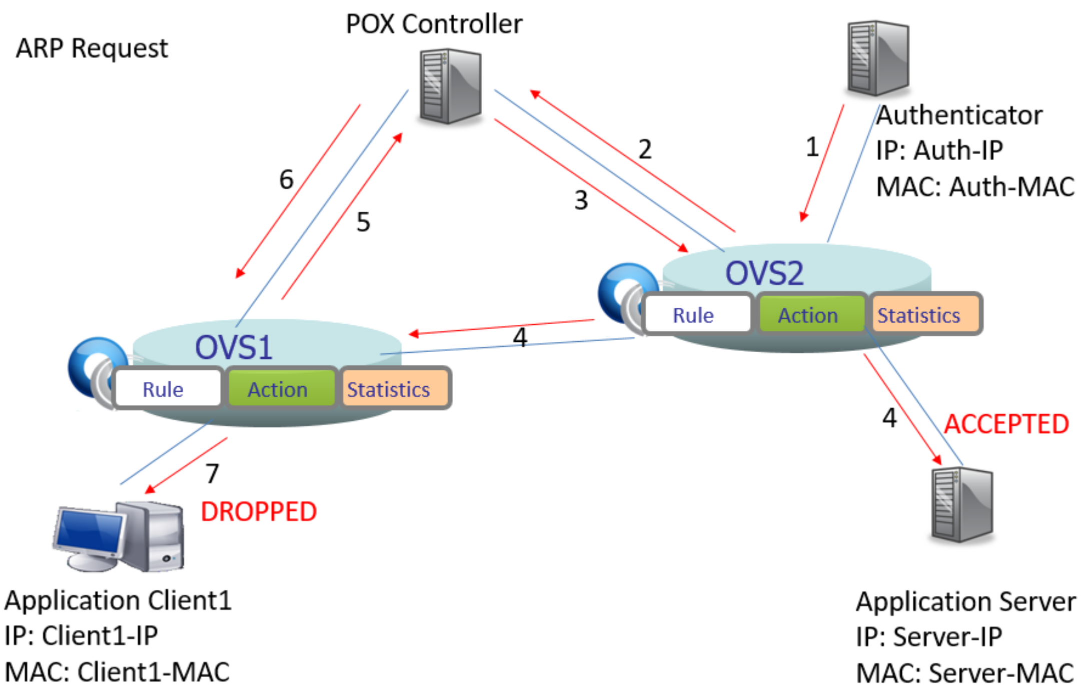
Note: Below sections depict the packet flow from just one Application Client and therefore corresponding diagrams will show only a single client (Application-Client) and the Mac and IP as Client-IP and Client-MAC.There are 3 development phases in our project:

I. Topology discovery:

Application server and the authenticator can be present anywhere in the topology and the controller should have the information of their location depicting how they can be reached from each of the OVS Switches present in the topology. To learn their locations, the authenticator sends ARP packets to the server which gets flooded in the network using which the controller keeps track of the port number in each OVS switch from where the authenticator and server can be reached.

Redirection of the client request from the server to the authenticator is required as it is reducing the load on the server. To do that dynamically controller should know about the location of the application server and the authenticator, i.e. from which port of the OVS switches the authenticator and server can be reached. This is achieved by topology discovery in this project. The controller stores the data path identifier of each OVS switch, the port number on the OVS and Mac address of the server (and authenticator) in a list to learn the network.

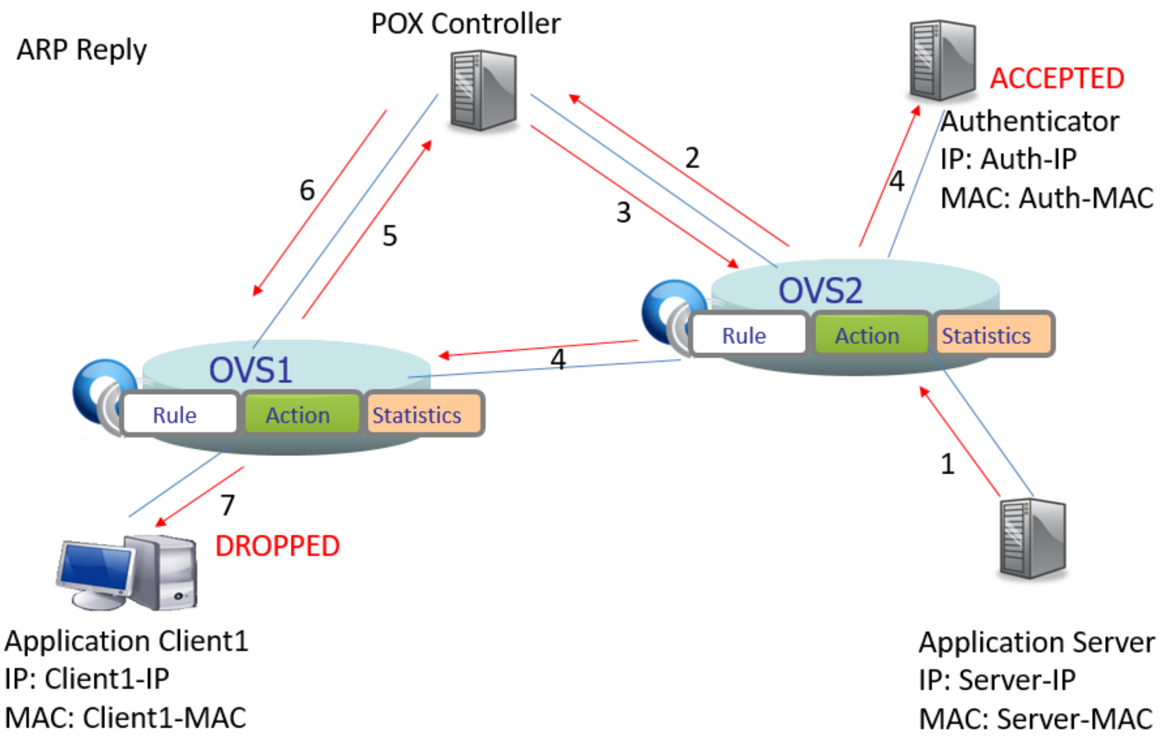
Packet Flow during Topology Discovery ARP Request:



Packet flows description:

1. Authenticator sends ARP request to the application server.
2. As there is no flow entry present in the OVS that matches this packet, the packet is encapsulated with PacketIn header (containing Data-path Id of the switch and the in_port information) and sent to the controller.
3. As the packet contains ARP request, controller keeps track of the location of the authenticator and the controller reachable from the requested OVS switch in a list called array[dpid][src_mac] =event.port using PacketIn header information. Controller then instruct the OVS to flood the packet.
4. Packet is forwarded to all ports except the incoming port.
5. As there is no flow entry present in the OVS that matches this packet, the packet is encapsulated with PacketIn header (containing Data-path Id of the switch and the in_port information) and sent to the controller whereas server accept the ARP as the requested destination IP matches its IP address.
6. Step 3 is performed on the other OVS Switch also.
7. Client drop the packet as the requested destination IP does not match with its IP

Packet flow during Topology Discover ARP Reply:



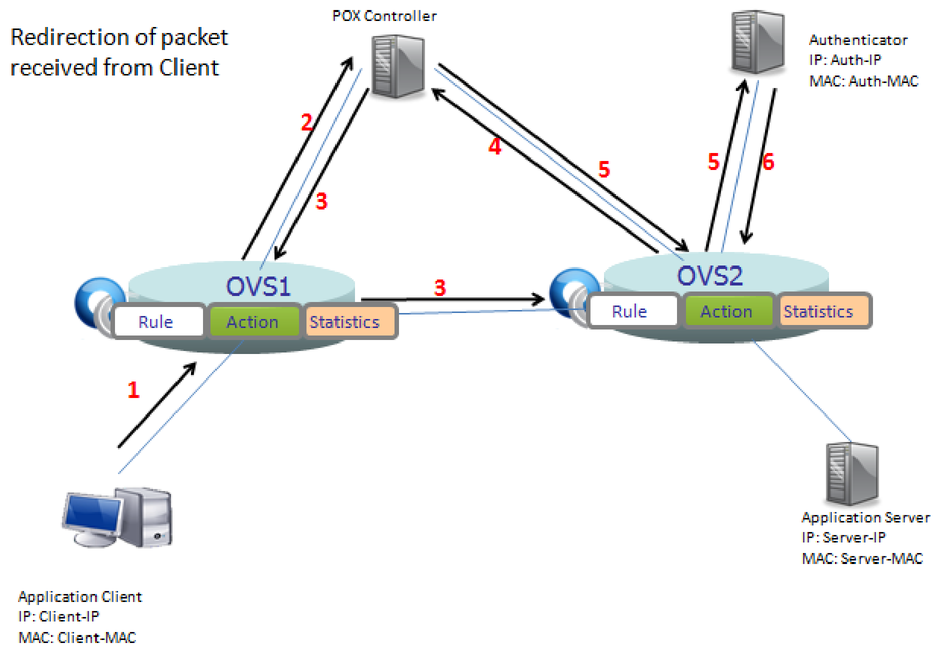
Packet flows description:

1. Server sends ARP reply to the authenticator.
2. As there is no flow entry present in the OVS that matches this packet, the packet is encapsulated with PacketIn header (containing Data-path Id of the switch and the in_port information) and sent to the controller.
3. As the packet contains ARP reply, controller keeps track of the location of the authenticator and the controller reachable from the requested OVS switch in a list called array[dpid][src_mac] =event.port using PacketIn header information. Controller then forwards the packet to other OVS switches and hosts present in the topology.
4. Packet is forwarded to all ports except the incoming port.
5. As there is no flow entry present in the OVS that matches this packet, the packet is encapsulated with PacketIn header (containing Data-path Id of the switch and the in_port information) and sent to the controller whereas client gets the ARP reply.
6. Step 3 is performed on the other OVS Switch also.
7. Client drop the packet as the requested destination IP does not match with its IP

II. Redirection for authentication:

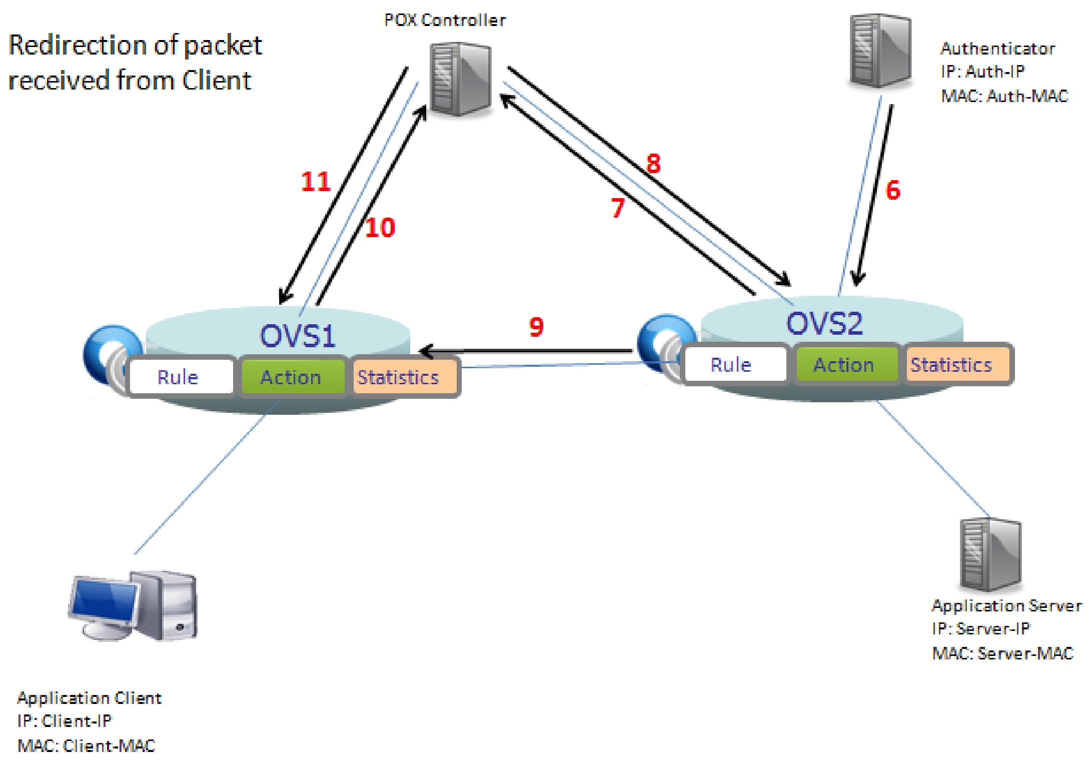
When the client tries to communicate with the server for the first time, the packets are redirected to the authenticator. The authenticator authenticates its packet and sends a response back to the controller to add flow rules for that authenticated client, such that the next time, that client can communicate directly to the server.

Packet Flow during Redirection:



Packet flows description:

1. When a client initiates a flow connection towards the Server, the packet reaches the OVS Switch.
2. As there is no flow entry present in the OVS that matches this packet, the packet is then sent to the controller.
3. As this flow is towards the server, the controller redirects this packet to the Authenticator using the information controller got from topology discovery phase. So the packet exits out the corresponding interface out of the OVS.
4. As there is no flow entry present in the OVS (OVS2) that matches this packet, the packet is again sent to the controller.
5. Again, the controller redirects this packet to the Authenticator for authentication.
6. Authenticator validates the Request IP address of the client and in case of valid client sends a response back to the controller.

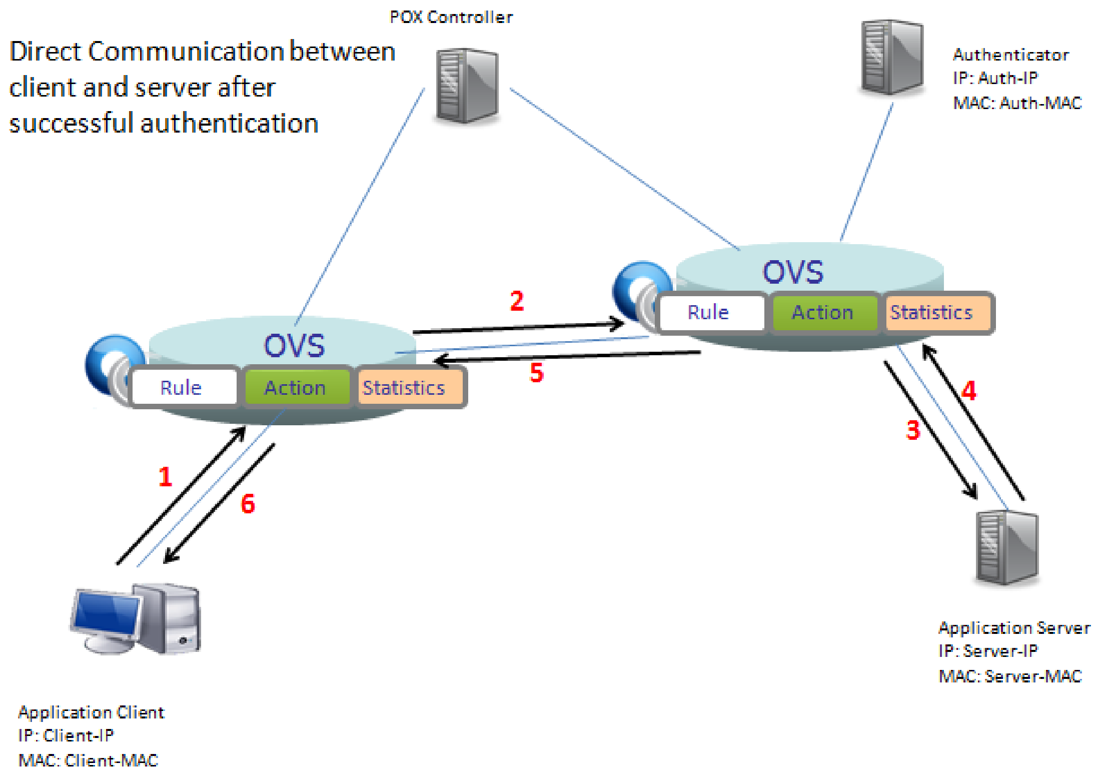


7. When this packet from the authenticator reaches OVS Switch (OVS2), as it does not have the flow entry the packet is again sent to the controller.
8. The Controller realizes that the packet received is from the authenticator and thus pushes flow entry on the OVS for the authenticated client and forward the packet to other OVS switches.
9. Same packet is forwarded to other OVS switches, to add flow entry on them if required for direct communication between the server and that authenticated client.
10. Again, the packet is sent to the controller because of the miss.
11. The Controller realizes that the packet received is from the authenticator and thus pushes flow entry on the OVS for the authenticated client.

III. Data flow between server and client after authentication

After the controller receives the response from the Authenticator, rules required for communication between server and client are added in the OVS Switches and data flow follows the direct path between the client and the server not via authenticator or the controller.

Packet Flow after authentication:



Packet flows description:

1. Client sends request to the application server and the packet reaches the OVS switch.
2. As there is a flow entry present in the OVS that matches this packet, the packet is forwarded to other OVS switch.
3. Packet again gets directed to the application server following the flow rule present in the OVS
4. Server gets the request and sends the response back to the client.
5. Packet from server to the client gets forwarded based on the action specified in the matched rule.
6. Packet finally reaches to the client following the flow rule present in the OVS

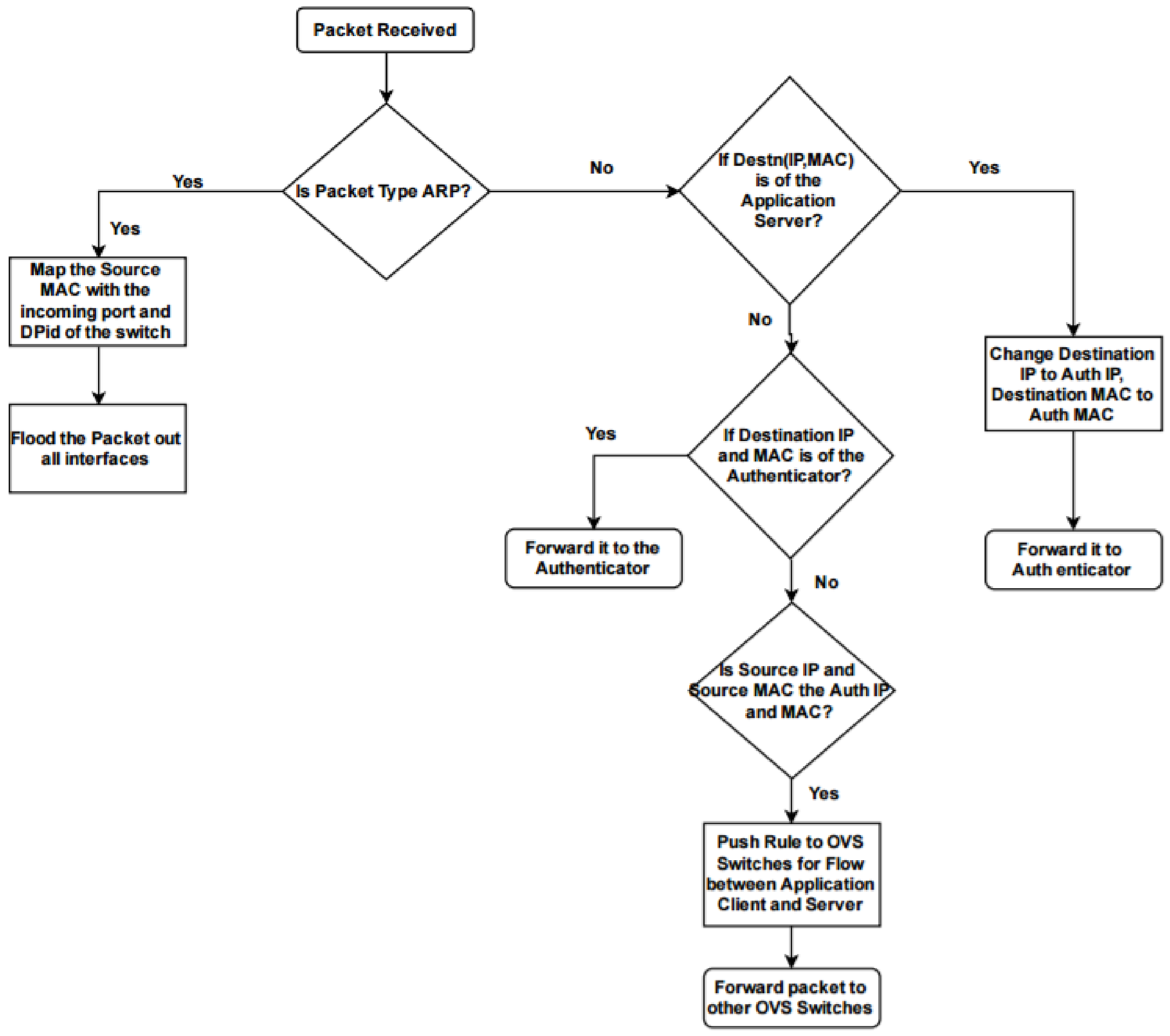
3.2 Low-Level Design:

The low level design of every component used is described in this section.

3.2.1 OpenFlow Controller:

In this project, we are using POX Controller as the OpenFlow SDN Controller. POX is an open source development platform that provides API's for OpenFlow. The controller specifies the entries that are pushed to the flow tables of the OVS Switches based upon the response received by the Authenticator.

The following flow chart is used to program the controller:


Packet Received
Is Packet Type ARP?
Yes
Yes
Map the Source MAC with the incoming port and DPid of the switch
Flood the Packet out all interfaces
No
If Destn(IP,MAC) is of the Application Server?
Yes
Change Destination IP to Auth IP, Destination MAC to Auth MAC
Forward it to Auth enticator
No
If Destination IP and MAC is of the Authenticator?
Yes
Forward it to the Authenticator
No
Is Source IP and Source MAC the Auth IP and MAC?
Yes
Push Rule to OVS Switches for Flow between Application Client and Server
Forward packet to other OVS Switches

Packet_In Module is utilized when we are implementing checks for the packet and determining the path that is to be taken by that packet.

For ARP Packets:

ARP packet is used to learn the network topology and the ARP packets is made to flood, such that host and server can get each other’s MAC Address required for the communication.

The below code snippet is used to Implement the above scenario:

```
Ifpacket.type == packet.ARP_TYPE:
```

<<From the ARP packet parsing library in POX determine the Destination IP for which the ARP was requested>>

```
arp_packet = packet.payload
dst_ip = arp_packet.protodst
```

<< Flood the packet out of all the ports of the OVS Switch – not adding a rule on the OVS Switches for this and keep a mapping of the Switch and the port over which it has come from, thus making the controller aware of the port connections of Application Server and Authenticator >>

```
msg = of.ofp_packet_out()
msg.actions.append(of.ofp_action_output(port = of.OFPP_FLOOD))
msg.data = event.ofp
self.connection.send(msg)
array[event.dpid][src_mac]=event.port
```

For IP Packets:

The controller uses the IP Packets and in particular TCP packets to redirect the flow to the authenticator and modify the flow entries on the table once a client is authenticated successfully.

<< For packet type IP and protocol used is TCP>>

```
Ifpacket.type == packet.IP_TYPE:
ip_packet = packet.payload
ifip_packet.protocol == ip_packet.TCP_PROTOCOL:
tcp_packet = ip_packet.payload
```

if (destination IP mac and port are of the Server):
<< Change the Destination IP, MAC and Port to that of the Authenticator and send it out through the corresponding switch port>>

```
        msg = of.ofp_packet_out()
      msg.actions.append(of.ofp_action_nw_addr.set_dst(auth_ip))
        msg.actions.append(of.ofp_action_dl_addr.set_dst(auth_mac))
        msg.actions.append(of.ofp_action_output(port = array[event.dpid][auth_mac]))
      msg.data = event.ofp
     self.connection.send(msg)
```

If (destination IP, MAC and port is of the Authenticator):
<< Send it out through the corresponding switch port>>

```
        msg = of.ofp_packet_out()
      msg.actions.append(of.ofp_action_output(port = array[event.dpid][auth_mac]))
      msg.data = event.ofp
      self.connection.send(msg)
```

If (Source IP, Mac and Port are of the Authenticator):
<<It means that the requested client is authenticated successfully. *Accordingly, add a Flow Entry to the OVS Switches allowing flows from Client to the Server>>*

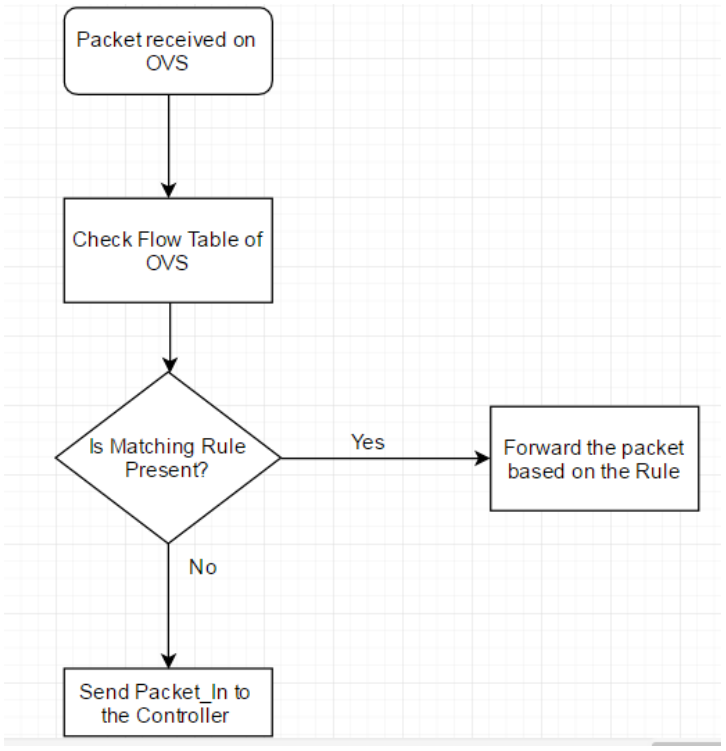
```
        self.connection.send(of.ofp_flow_mod(action=of.ofp_action_output(
port=array[event.dpid][web_mac]),match=of.ofp_match(dl_type=0x800,nw_proto=6,
                                        dl_src=dst_mac,dl_dst=web_mac,tp_dst=80)))

        self.connection.send(of.ofp_flow_mod(action=of.ofp_action_output(
port=array[event.dpid][dst_mac]),match=of.ofp_match(dl_type=0x800,nw_proto=6,
                                        dl_src=web_mac,dl_dst=dst_mac,tp_src=80)))
```

<<forwarding packet to other OVS switches>>

```
        msg = of.ofp_packet_out()
        msg.actions.append(of.ofp_action_output(port = of.OFPP_FLOOD))
     msg.data = event.ofp
     self.connection.send(msg)
```

3.2.2 OpenflowvSwitches:

The below flow chart depicts the way Openflow vSwitches work in this project:


Packet received on OVS
Check Flow Table of OVS
Is Matching Rule Present?
Yes
Forward the packet based on the Rule
No
Send Packet_In to the Controller

The OVS Switches have no entries initially. The switches forward any packet (that does not match the entries) to the POX Controller. The POX Controller decides if flow rules need to be pushed on the OVS Switches based upon the response from the Authenticator.

Flow Rules on OVS before authentication:

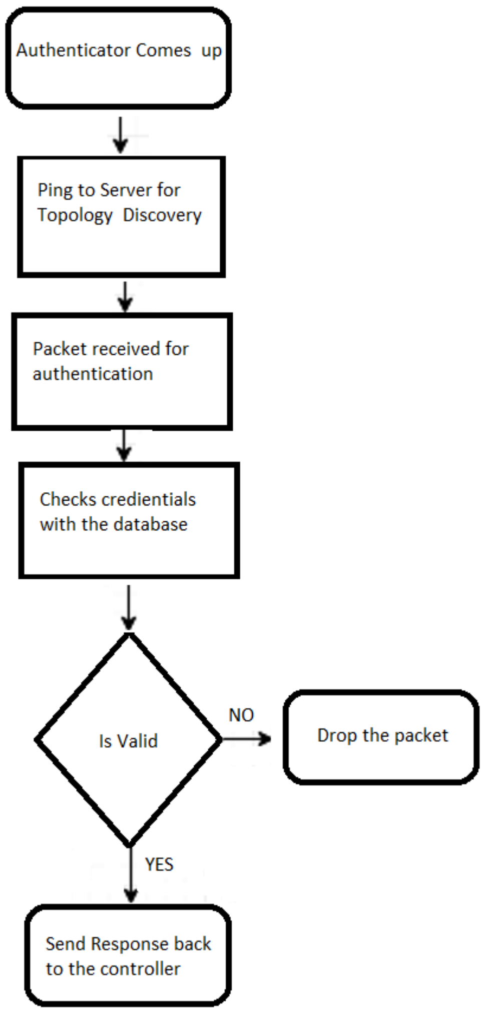
```
    root@ubuntu:/home/ubuntu# ovs-ofctl dump-flows br0
    NXST_FLOW reply (xid=0x4):
```

Flow Rules seen on OVS after successful authentication:

```
root@ubuntu:/home/ubuntu# ovs-ofctl dump-flows br0
NXST_FLOW reply (xid=0x4):
cookie=0x0, duration=368.689s, table=0, n_packets=4,
n_bytes=252, idle_age=353,
tcp,dl_src=00:21:9b:da:76:b3,dl_dst=ec:f4:bb:a0:28:d1,tp_src=80
actions=output:4
cookie=0x0, duration=368.689s, table=0, n_packets=8, n_bytes=492,
idle_age=353,
tcp,dl_src=ec:f4:bb:a0:28:d1,dl_dst=00:21:9b:da:76:b3,tp_dst=80
actions=output:3
```

3.2.3 Authenticator:

From the point of view of packet flow, the following flow chart depicts the way authenticator works.

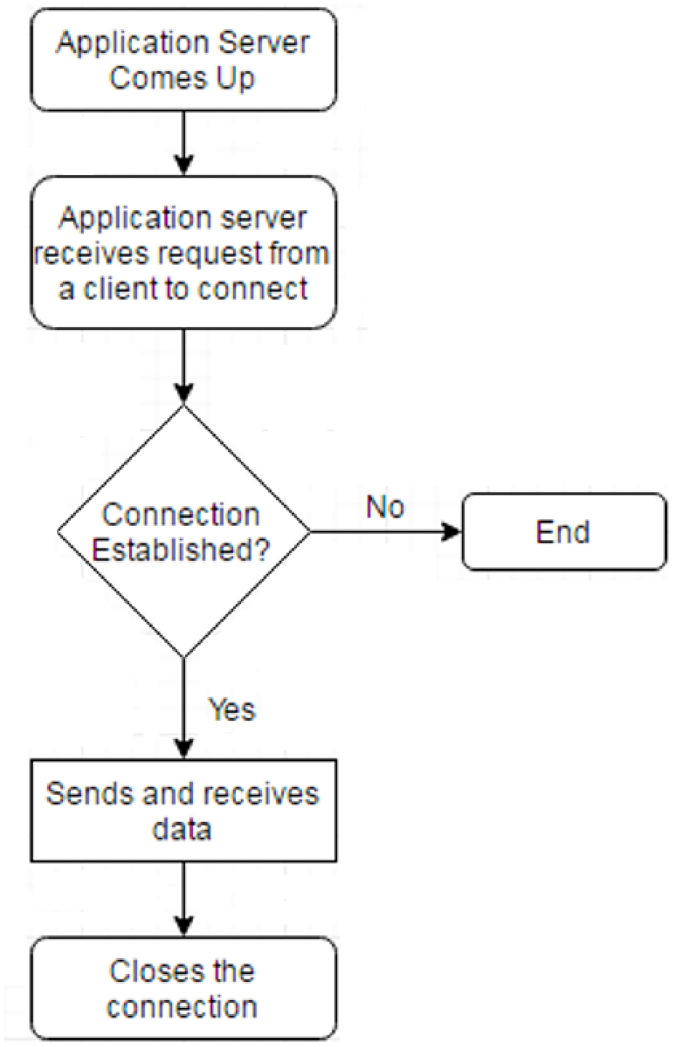

Authenticator Comes up
Ping to Server for Topology Discovery
Packet received for authentication
Checks credientials with the database
Is Valid
NO
Drop the packet
YES
Send Response back to the controller

When the Authenticator comes up, it sends ARP packet to the server present in the network, as it helps the controller discover the location of the authenticator in the network. The Authenticator basically runs a Kernel module using Netfilter that listens to packets received at an interface. When the authenticator receives a packet from Controller, it verifies whether the validity of the client IP address. In the authenticator, we have used an array to store all IP addresses that are allowed to talk to the application-server. Once the Authenticator receives the packet, it compares the requested client's IP address with the one present in its database (or array). If it matches, then the Application Client is authenticated and this response is sent back to the Controller. If the credentials (here IP Address) do not match, the Authenticator drops the packet.

The below snippet of code was used in the authenticator to compare the credentials of a TCP packet:

```
if (ip_header->protocol == 6) {
      for(i=0;i<array_size;i++){
        if (strcmp(array[i],dest_ip)==0)
           break;
      }
      if (i!=array_size){
        printk(KERN_INFO "Dropped:http request does not have valid
dst address:%s\n",dest_ip);
        return NF_DROP;
      }
   }
   return NF_ACCEPT;
```

3.2.4 Application Server:


Application Server Comes Up
Application server receives request from a client to connect
Connection Established?
No
End
Yes
Sends and receives data
Closes the connection

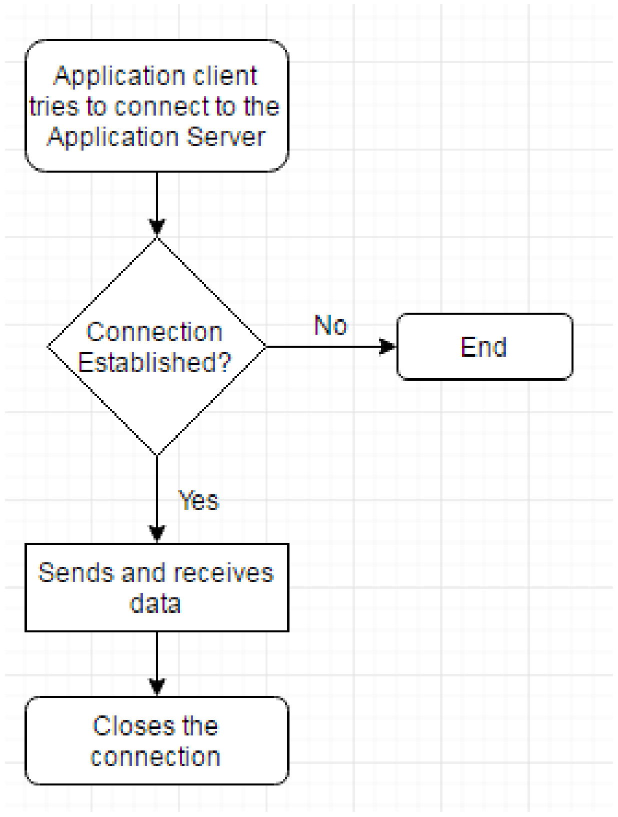
In our project, the Server runs a web application. The client and Server communicate using TCP (port 80) and the Server is unaware about the fact the client is authenticated before establishing connection with the Application Server.

When this server comes up, it sends a ping request to a topology IP. This is done so that the Controller can determine the location and MAC Address of the Application Server.

After a client establishes the connection, the server and client exchange data with each other until one of them closes the connection.

In order to implement an Application Server hosting a web page in our project, we have used an Ubuntu Server with Apache installed that hosts a web page on TCP Port 80. We have used basic PHP and HTML to code the basic web page.

3.2.5 Application Client:



The Application Client is a TCP based Client Application that sends data destined to the Application Server. As we have used a Web Application on the Server, we are using browsers on a Windows Client in order to access the server. On an Ubuntu Client we have used curl package and browser to send requests in order to connect to the Web Server.

3.3 Per-member Responsibilities:

Tasks	**Abhishek**	**Mahesh**	**Manushri**	**Sushma**
Environmental Setup in GENI and NetLabs	Setup of Topology in GENI and NetLabs.	Setup of environment in NetLabs for final testing	Setup of GENI slice for internal testing	Setup of environment in NetLabs for final testing
OpenFlow Controller	Research about various controllers and POX controller installation	Design and implementation for Topology discovery and ARP module	Design and implementation for redirection and flow entries addition module	Design and implementation for redirection and flow entries addition module
Authenticator	N/A	Implementation of Kernel Module for authentication	Researched about various authentication mechanisms	Researched about various authentication mechanisms
Application Server	Developed a simple Web application for server	N/A	N/A	N/A
Website Maintenance	Updated the website after each report phase	N/A	N/A	Updated the website after each report phase
Testing	Tested authenticator Module	Tested the topology discovery	Tested the redirection module	Tested the addition of Flow entries
Documentation	All	All	All	All

3.4 Project Timeline

Tasks	**Sub Tasks**	**Start**	**Finish**	**Duration**	**Milestones**
Research	Learn about OpenFlow and Authentication	10/1/2016	10/6/2016	6 days	Learning objectives met
Environment Setup	Bringing up network topology in GENI	10/7/2016	10/8/2016	2 days	
	Setting up OpenFlow Controller and Switches	10/10/2016	10/12/2016	3 days	
	Working on Radius server and Client for authentication	10/14/2016	10/16/2016	3 days	
Implementation	Designing own protocol for authentication	10/18/2016	10/19/2016	2 days	
	Working on Topology discovery	10/29/2016	11/2/2016	3 days	Design and Implementation of Topology discovery module successful
	Working on redirection of flows	11/2/2016	11/4/2016	3 days	Redirection of flows module successful
Unit Testing	Testing units - Authenticator, POX controller operations	11/5/2016	11/9/2016	5 days	Unit testing and some system testing done
Documentation	Work on Project Interim Report and update the website	11/11/2016	11/13/2016	3 days	
System testing	Bringing up network topology in Netlabs	11/14/2016	11/15/2016	2 days	Implementation in Netlabs
	Implement testcases of the entire system	11/16/2016	11/18/2016	3 days	
Documentation	Debugging in case of errors	11/19/2016	11/22/2016	4 days	Testing the whole system
	Work on the final Project Report	11/23/2016	11/25/2016	3 days	
	Demo of the project	11/29/2016	12/1/2016	2 days	

4. Verification and Validation

4.1 Test application

The project is tested using the client and server application that is designed by the team. Upon successful communication between client and server, we can be sure that the client has been authenticated. We can also check status of authentication in the flow tables of the OVS Switches. If the switches have updated flow entries: Authentication was successful.

4.2 Test Plan

Objective of the test plan is to prepare and run test cases that authenticate the client to communicate with the server. Following sections would give a set of test cases.

4.3 Test Cases

The following is the topology we used to run the below mentioned test cases. It consists of two clients Client C1 (valid) and Client C2 (invalid), two OpenFlow Vswitches, one authenticator, one application server and POX controller. Both the OpenFlow switches are controlled by the same POX controller.

Following are the data paths in given network topology:

1. Data Path for the Client C1(valid):

Before authentication

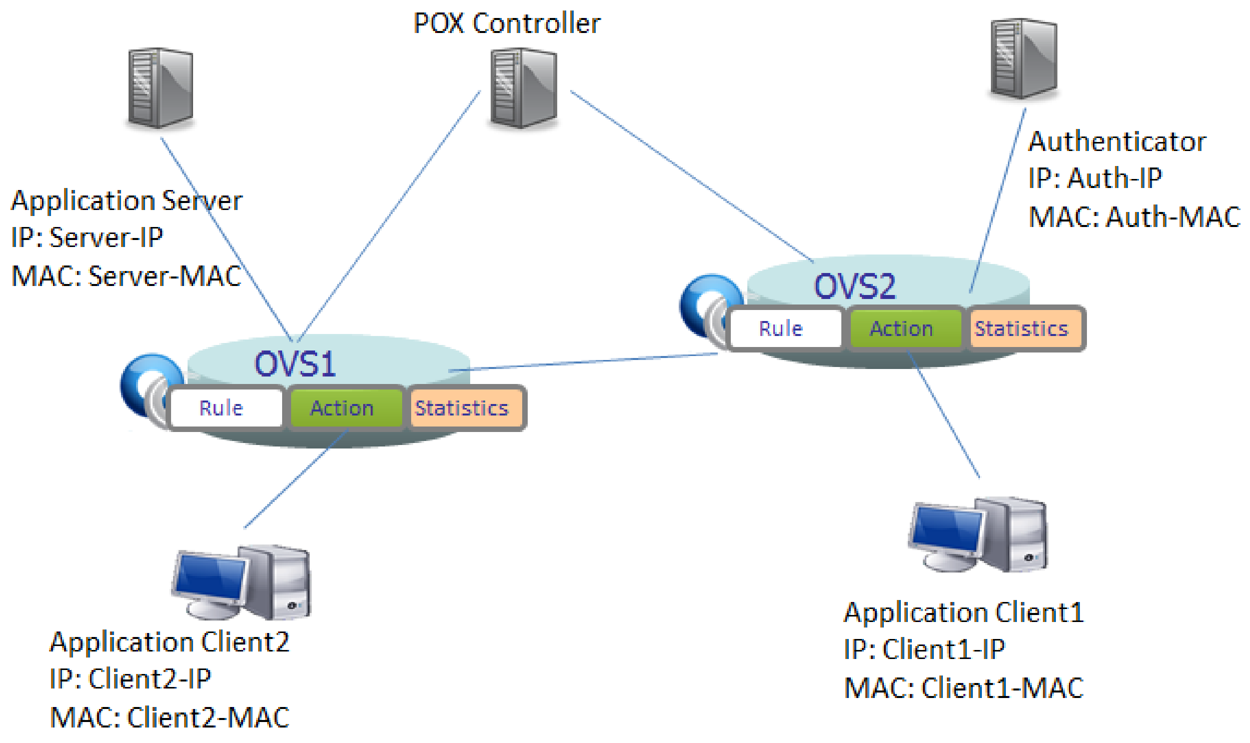
C1 -> OVS2 -> POX Controller -> OVS2 ->Authenticator ->OVS2 -> POX Controller ->OVS2 -> OVS1 -> POX Controller

After authentication

C1 -> OVS2 ->OVS1 ->Application Server ->OVS1 -> OVS2 -> C1

2. Data Path for the Client C2(invalid)

C2 -> OVS1 -> POX Controller -> OVS1 ->OVS2-> POX Controller-> OVS2 ->Authenticator -> dropped



Test ID	**Actions**	**Expected Observation**	**Conclusion**	**Result**
T1	Verification of Open vSwitches connection to the POX controller	OVS output shows the controller configuration and the logs at the controller shows that both the switches are connected to the controller	Successful connection between OVS Switches and the controller	Passed
T2	Ping from Authenticator to Application Server for topology discovery	Controller logs will show the accessibility of the server and the authenticator from each of the OpenFlow vswitches	Controller has discovered the location of both the server and authenticator in the network	Passed
T3.1	First time service request from a valid client to the server	Logs at the controller show the redirection of traffic to the authenticator	The client traffic is being redirected to the authenticator in case of first time connection.	Passed

T3.2	Service request from aninvalid client	Logs at the controller show the redirection of traffic to the authenticator	The client traffic is being redirected to the authenticator The client is authenticated and the flow entry is added for the client which allows the client to communicate with the server	Passed
T4.1	Verification of authenticator's process for Valid Client	Logs at the authenticator show the authentication status as success	Authenticator authenticates the clients with valid IP addresses	Passed
T4.2	Verification of authenticator's process for Invalid Client	Logs at the authenticator show the authentication status as failure	The clients with invalid IP addresses are not authenticated	Passed
T5.1	Verification of authenticator's reply to the controller in case of Valid Client	Logs at the controller show the packet received from the authenticator	The authenticator replies to the controller for clients with valid IP addresses	Passed
T5.2	Verification of authenticator's reply to the controller in case of Invalid Client	Logs at the controller show nothing about reply from the authenticator	The authenticator replies to the controller only for the clients with valid IP addresses	Passed
T6	Verification of flows on the switches after reply from the authenticator	Logs on the controller show modification of flow tables and logs on the switches show updated flow rules	The flow rules are updated on the switches for valid client to reach the web server	Passed

Unit Test Cases implemented in NetLabs:

Switch	DPID (String)	DPID (Long)
OVS 1	00-1b-21-83-a2-8c	116526391948
OVS 2	00-1b-21-76-34-a0	116525511840

DPIDs of the switches in the topology

Node	IP address
Client 1	192.168.2.4/24
Client 2	192.168.2.3/24
Web Server	192.168.2.1/24
Authenticator	192.168.2.2/24
Pox Controller Interface connected to OVS 1	192.168.3.1/24
Pox Controller Interface connected to OVS 2	192.168.1.1/24

IP addresses of nodes in the topology

Test case 1:

Objective: Verify Open vSwitches connection to the POX controller after configuring the appropriate controller details on OVS.

Procedure:

1. Start POX controller module on the controller node using the command "./pox.py app.py"
2. Configure controller IP address on OVS 1 using the command "ovs-vsctl set-controller tcp:192.168.3.1:6633"; on OVS 2 using the command "ovs-vsctl set-controller br0 tcp:192.168.1.1:6633"
3. Verify OVS connection to the controller using "ovs-vsctl show br0"
4. Verify log on controller node that detects the switch as connected

Results:

OVS output shows the controller configuration and both the switches are shown connected in the controller log

Logs:

OVS 1

```
root@ubuntu:/home/ubuntu# ovs-vsctl show
bcfabc1a-3026-436a-80da-d8fc77d3682b
    Bridge "br0"
```

```
        Controller "tcp:192.168.3.1:6633"
        fail_mode: secure
        Port "br0"
            Interface "br0"
                type: internal
        Port "eth2"
            Interface "eth2"
        Port "eth0"
            Interface "eth0"
        Port "eth1"
            Interface "eth1"
        Port "eth3"
            Interface "eth3"
    ovs_version: "2.0.2"
root@ubuntu:/home/ubuntu#
```

<u>OVS 2</u>

```
root@ubuntu:/home/ubuntu# ovs-vsctl show
d3196fb1-4ffd-49a5-aabb-8043c90c0920
    Bridge "br0"
        Controller "tcp:192.168.1.1:6633"
        fail_mode: secure
        Port "eth1"
            Interface "eth1"
        Port "eth0"
            Interface "eth0"
        Port "br0"
            Interface "br0"
                type: internal
        Port "eth2"
            Interface "eth2"
        Port "eth3"
            Interface "eth3"
    ovs_version: "2.0.2"
root@ubuntu:/home/ubuntu#
```

<u>POX Controller</u>

```
root@ubuntu:/home/ubuntu/pox# ./pox.py forwarding.final_redirection
POX 0.2.0 (carp) / Copyright 2011-2013 James McCauley, et al.
INFO:core:POX 0.2.0 (carp) is up.
INFO:openflow.of_01:[00-1b-21-83-a2-8c 1] connected
INFO:openflow.of_01:[00-1b-21-76-34-a0 2] connected
```

Test case 2:

Objective: Verification of topology discovery module

Procedure:

1. Start POX controller module on the controller node using command "./pox.py app.py"
2. Configure controller IP address on OVS 1 using the command "ovs-vsctl set-controller tcp:192.168.3.1:6633"; on OVS 2 using the command "ovs-vsctl set-controller br0 tcp:192.168.1.1:6633"
3. Ping the web server IP address from authenticator
4. Verify the controller log showing port locations of authenticator and web server on switches
5. Verify the successful ping to web server from the authenticator

Result:

Topology discovery is successful, the authenticator and the web-server is reachable via port-1, port-3 of OVS2 (116525511840); port-2, port-4 of OVS1 (116526391948) as can be seen from the logs.

Logs:

<u>POX controller</u>

```
defaultdict(<type 'dict'>, {116525511840: {EthAddr('00:1b:21:23:35:28'): 1,
EthAddr('00:21:9b:da:76:b3'): 3}, 116526391948: {EthAddr('00:1b:21:23:35:28'): 4,
EthAddr('00:21:9b:da:76:b3'): 2}})
```

<u>Authenticator</u>

```
untu@ubuntu:~/Desktop/firewall$ ping 192.168.2.1
PING 192.168.2.1 (192.168.2.1) 56(84) bytes of data.
64 bytes from 192.168.2.1: icmp_seq=1 ttl=64 time=40.6 ms
64 bytes from 192.168.2.1: icmp_seq=2 ttl=64 time=23.4 ms
64 bytes from 192.168.2.1: icmp_seq=3 ttl=64 time=6.81 ms
--- 192.168.2.1 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3005ms
rtt min/avg/max/mdev = 6.810/30.989/53.090/17.481 ms
```

Test Case 3:
Objective: Verification of redirection of traffic to authenticator

Procedure:
1. Start POX controller module on the controller node using command "./pox.py app.py"
2. Configure controller IP address on OVS 1 using the command "ovs-vsctl set-controller tcp:192.168.3.1:6633"; on OVS 2 using the command "ovs-vsctl set-controller br0 tcp:192.168.1.1:6633"
3. Ping the web server IP address from authenticator
4. Assign IP addresses to the hosts connected one each to the OVS
5. Send a service request from both the clients (one valid and one invalid) to the server
6. Verify the logs on controller showing redirection to authenticator

Results:
The traffic got redirected to the authenticator as can be seen in the controller logs.

Logs:

<u>POX controller</u>

```
1. Valid Client:
Request reached controller with destination mac as web-server and redirected to authenticator
('dpid of the ovs:', 116525511840)
('source IP & Destination IP', IPAddr('192.168.2.4'), IPAddr('192.168.2.1'))

2. Invalid Client:
Request reached controller with destination mac as web-server and redirected to authenticator
('dpid of the ovs:', 116526391948)
('source IP & Destination IP', IPAddr('192.168.2.3'), IPAddr('192.168.2.1'))
```

Test Case 4:
Objective: Verification of packet received and processed by authenticator

Procedure:

1. Start POX controller module on the controller node using command "./pox.py app.py"
2. Configure controller IP address on OVS 1 using the command "ovs-vsctl set-controller tcp:192.168.3.1:6633"; on OVS 2 using the command "ovs-vsctl set-controller br0 tcp:192.168.1.1:6633"
3. Ping the web server IP address from authenticator
4. Assign IP addresses to the hosts connected one each to the OVS
5. Send a service request from both the clients (one valid and one invalid) to the server

6. Verify the logs on authenticator showing authentication of clients

Results:
The authenticator authenticates the client with valid IP address and drops the traffic of clients with invalid IP address

Logs:
Authenticator

```
1. Valid client
[166002.958439] Accepted:http request have valid dst address:192.168.2.4

2. Invalid client
[166004.072671] Dropped:http request does not have valid dst address:192.168.2.3
[166004.072808] Dropped:http request does not have valid dst address:192.168.2.3
[166004.338786] Dropped:http request does not have valid dst address:192.168.2.3
```

Test Case 5:
Objective: Verification of authenticator's reply to the controller

Procedure:
1. Start POX controller module on the controller node using command "./pox.py final_redirection.py"
2. Configure controller IP address on OVS 1 using the command "ovs-vsctl set-controller tcp:192.168.3.1:6633"; on OVS 2 using the command "ovs-vsctl set-controller br0 tcp:192.168.1.1:6633"
3. Ping the web server IP address from authenticator
4. Assign IP addresses to the hosts connected one each to the OVS
5. Send a service request from both the clients (one valid and one invalid) to the server
6. Verify the logs on POX controller showing the broadcast from authenticator

Results:
The reply from authenticator has been forwarded to both the Open vSwitches.

Logs:
POX Controller

```
1. Valid Client:
Reply from authenticator
('dpid of the ovs:', 116525511840)
('source IP & Destination IP', IPAddr('192.168.2.2'), IPAddr('192.168.2.4'))
('source MAC & Destination MAC', EthAddr('00:1b:21:23:35:28'), EthAddr('ec:f4:bb:a0:28:d1'))
authenticated packet Broadcasting
Reply from authenticator
```

```
('dpid of the ovs:', 116526391948)
('source IP & Destination IP', IPAddr('192.168.2.2'), IPAddr('192.168.2.255'))
('source MAC & Destination MAC', EthAddr('ec:f4:bb:a0:28:d1'), EthAddr('ff:ff:ff:ff:ff:ff'))
authenticated packet Broadcasting

2. Invalid Client:
Empty
```

Test Case 6:

Objective: Verification of flows on the switches after authenticator's reply

Procedure:

1. Start POX controller module on the controller node using command "./pox.py final_redirection.py"
2. Configure controller IP address on OVS 1 using the command "ovs-vsctl set-controller tcp:192.168.3.1:6633"; on OVS 2 using the command "ovs-vsctl set-controller br0 tcp:192.168.1.1:6633"
3. Ping the web server IP address from authenticator
4. Assign IP addresses to the hosts connected one each to the OVS
5. Send a service request from both the clients (one valid and one invalid) to the server
6. Verify flow entries on both the switches being updated after controller receives reply from authenticator

Results:

After authenticating the valid client, the POX controller pushes flow rules on to both the switches as can be seen from the controller log below. Also, the logs on the switches

Logs:

<u>POX Controller</u>

```
1. Valid Client:
Modifying flow-table
('dpid of the ovs:', 116525511840)
('source IP & Destination IP', IPAddr('192.168.2.2'), IPAddr('192.168.2.4'))
('source MAC & Destination MAC', EthAddr('00:1b:21:23:35:28'), EthAddr('ec:f4:bb:a0:28:d1'))
Modifying flow-table
('dpid of the ovs:', 116526391948)
('source IP & Destination IP', IPAddr('192.168.2.2'), IPAddr('192.168.2.255'))
('source MAC & Destination MAC', EthAddr('ec:f4:bb:a0:28:d1'), EthAddr('ff:ff:ff:ff:ff:ff'))

2. Invalid Client:
Empty
```

OVS 1

1. Valid Client:

```
root@ubuntu:/home/ubuntu# sudo ovs-ofctl dump-flows br0
NXST_FLOW reply (xid=0x4):
cookie=0x0, duration=1476.874s, table=0, n_packets=4, n_bytes=252, idle_age=1461,
tcp,dl_src=00:21:9b:da:76:b3,dl_dst=ec:f4:bb:a0:28:d1,tp_src=80 actions=output:4
 cookie=0x0, duration=1476.874s, table=0, n_packets=8, n_bytes=492, idle_age=1461,
tcp,dl_src=ec:f4:bb:a0:28:d1,dl_dst=00:21:9b:da:76:b3,tp_dst=80 actions=output:2
root@ubuntu:/home/ubuntu#
```

2. Invalid Client:

Empty

OVS 2

1. Valid Client:

```
root@ubuntu:/home/ubuntu# ovs-ofctl dump-flows br0
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=368.689s, table=0, n_packets=4, n_bytes=252, idle_age=353,
tcp,dl_src=00:21:9b:da:76:b3,dl_dst=ec:f4:bb:a0:28:d1,tp_src=80 actions=output:4
 cookie=0x0, duration=368.689s, table=0, n_packets=8, n_bytes=492, idle_age=353,
tcp,dl_src=ec:f4:bb:a0:28:d1,dl_dst=00:21:9b:da:76:b3,tp_dst=80 actions=output:3
```

2. Invalid Client:

Empty

4.4 Demo playbook:

Setup for the Demo

1. Build a network topology in NetLabs, the topology would be same as mentioned in previous section.
2. Configure the network

   Install Open vSwitch on all the switches

   Install the POX controller

   Bringing up application server and authenticator

   Assigning IP addresses in the network topology
3. Walk through the modules implemented in authenticator

Below are the test cases/scenarios which will be executed during the Project Demonstration

Demo ID	Demo Scenario	Expected Observation	Conclusion
D1	Discovering authenticator and application server locations by the controller using ping request from authenticator to server	Controller log showing the reachability of authenticator and server from both the switches. The log will show the binding of MAC address and DPID with the port number of each OVS	Learning of authenticator and server locations by the controller is successful
D2	First time service request from valid client	As there is no flow entry in the OVS for the requested client, the traffic is redirected to the authenticator as can be seen from the controller logs. Logs on authenticator show the validity of the client. Logs at the controller shows the reply from authenticator and the modification of flow entries on switches required for the later direct communication between client and server	The packet has been redirected to authenticator and upon successful authentication flow entry has been added
D3	Service request from valid client after authentication	As can be seen from the output of "ovs-vsctl dump-flows br0" command, flow entries required for communication between client and server are already present on the OVS. Hence, the traffic is directly sent to the web server.	Valid client is able to communicate with the server without being redirected to the authenticator
D4	Service request from an invalid client	As there is no flow entry in the OVS for the requested client, the traffic is redirected to the authenticator as can be seen from the controller logs. Logs on authenticator show the invalidity of the client and hence the packet is dropped. Logs at the controller show no reply from authenticator and hence no modification of flow entries is done on switches	Invalid client is not able to communicate with the server and it's packets are dropped at the authenticator

5. Self-Study:

Following are the performance tests carried out as part of self-study:

Self-study ID	Performance Test Scenario	Expected Observation
1	Authentication performed by the OpenFlow controller	More processing timeon the controller for handling authentication process hence delay in data forwarding operations
2	Authentication performed by the server	More processing time on the application server for handling authentication process. Also there's a possibility of DoS (Denial of Service) attack
3. Proposed model	Authentication performed by a separate system called authenticator	Less processing time on application server and in the controller, less delay in the data path forwarding operations as well because of a separate machine handling authentication function. No possibility of DoS (Denial of Service) attack

5.1 Authentication on Controller vs. Authentication on separate machine

Description:
We examined the scenario when authentication is done on the controller instead of having a separate authenticator. In this case the controller handles the functions of both the data flow in network and authentication as well. As the controller takes time in processing the authentication requests there will be a delay in pushing forwarding rules to the switches. We are using processing time on the controller in both cases as a metric for comparison. Processing time here is the time taken between the events of request received by the controller from OVS and packet is forwarded to the authenticator pushed on to OVS.

Observation:
We observed that the time taken by controller to redirect the traffic is very less compared to having the authentication functionality on the controller.

Results:
We have implemented a POX application which performs authentication instead of redirecting to a separate machine for authentication. Following readings are recorded using “time.clock” function of Python for two different http service requests.

Controller	Processing Time on the controller(m sec)
With authentication	1.195, 1.141
Without authentication	0.251, 0.201

5.2 Authentication on Web Server vs. Authentication on separate machine

Description:
We are using Processing timeat the application server as a metric for comparing client server communication model with the proposed authentication mechanism, against a base case of client getting authenticated by the server itself without any redirection of its traffic to the authenticator. Processing time measured here is the time interval between the events of request from client received by web server to web page (valid/invalid) displayed to client.

Observation:
We observed that the processing time of the web server for a client request in case of authentication done in the web server is much higher as compared to our proposed model.

The possibility of a DoS attack is avoided by having a separate authenticator, different from server as in this case the unauthorized clients are dropped before reaching the server itself.

Results:
We have implemented a code which gives the processing time of the web server with and without authentication. Following readings are recorded using “microtime” function of PHP for five different http service requests.

Web-server	Processing Time on the web server(µ sec)
With authentication	241,212,157,208,153
Without authentication	72,64,78,67,92,78

6. REFLECTIONS:

6.1 Per-member learning experiences

Abhishek:

- I researched about the various controllers that were available for us to use and chose POX Controller for the project. This was because POX uses Python language, which is easier to code in as it is very close to the spoken language. POX also had more material available which explains how to code the necessary functionality that we needed to implement.
- I have implemented the web server for proposed model which hosts an application that the client requests access to. We have also modified the web server to implement authentication for our self-study case. I obtained a basic understanding of how to code a simple web page using PHP and how to use Apache to host a simple Web page.
- As part of this IP Project, I worked along with Sushma to bring up the topology in NetLabs. This setup of topology in Netlabs gave me an experience of working in an actual lab and hardware devices and how to troubleshooting the errors while bringing up the topology.

Mahesh:

- As a part of project, I implemented the topology discovery and the handling of ARP Packets in POX. What I learnt from this project is how the controller handles topology discovery by default and how to improvise on this topology such that the location of the Web Server and Authenticator can be determine dynamically. This was done by storing the mapping of DPID and MAC address with each OVS switches port such that the controller knows how to reach authenticator and server from each OVS switches.
- I had also implemented Kernel code for performing authentication on the Authenticator. What I learnt while implementing this feature was the number of hooks that were available for me to use while designing this module. I used the Netfilter hook “POST_ROUTING” to implement this feature. This helped me understand how to use this hook for implementing a simple authenticator for our project and how to code a simple module for the Linux Kernel.

Manushri:

- For this project, Sushma and I designed and implemented redirection on the controller and pushing flow entries on the OVS.
- The tricky part was to come up with logic to determine the required switches that are needed for the flow entries to be pushed after successful authentication. This was done by modifying the packet that is received from the authenticator and broadcasting this response packet such that the controller can decide on which switches the flow entries needs to be pushed. It helped me understand how to use the flow modification and packet_In functionality at the POX to modify flow entries and packet contents.
- This project also gave me an insight into the various stages that occur in the software development cycle of a project.

Sushma:

- I have implemented redirection of TCP traffic to authenticator and flow entry changes along with Manushri.
- This redirection was achieved by using handle_Packet_In function of POX. When a packet_In is received by the controller we modified the packet's destination IP and MAC Address such that the packet can be redirected to the authenticator. This project gave me a better understanding of data path and how the various entities in our project interact with each other. Also, as I had no experience of coding in Python and POX, this project also gave me a good starting point to learn the language.
- I have also brought up the topology in NetLabs, which enabled me to even consider the physical layer level problem, closer to real world scenario.

6.2 What went as expected

- Redirection:As a group, our work on redirecting the flows from the client towards the server went as expected. That is, when the client is unauthenticated, flows from the clients were supposed to be redirected towards the authenticator. Once the client is authenticated, appropriate flows were pushed into the only the required OVS for direct communication between server and authenticated client.

- Server unaware of authentication:In this project, as we proposed earlier the client and server are unaware of the authentication. This went as expected sincein our model, the client TCP/HTTP session initiationpacket is redirected towards the authenticator without the knowledge of web server. The client is validated by the authenticator using its IP address alone, hence not requiring client to send additional credentials, making client also unaware of authentication.

- Performing Authentication on a different server:Implementing authentication on a different machine not on the controller helps reduce the time taken to process a packet on the controller as shown in our self-study results. The time that a controller takes to process a Packet_In is more in the case where authentication is also performed at the controller.

6.3 What went different from expected

- As a group, our work on topology discovery was different from what was expected. POX does not learn the end hosts' location in a network when a port_up (or LLDP) notification is received from the OVS. To resolve this, we used our own mechanism such that POX knows the location of the hosts i.e., how to reach authenticator and web server from each OVS in the topology.
- Also, as per our initial proposed design, using EAP protocol for 802.1x authentication did not go as per expectation. This was because 802.1x frames used multicast IP address to signify successful authentication. It was difficult to determine which host was being authenticated using these multicast addresses in the reply. Hence, we decided to drop this design of authenticator and used a kernel code for authenticator.
- Considering the hardware limitations in the Netlabs facility, we had to make small changes to our design topology. The Linux machines on which we loaded OVS have a maximum of 4 Ethernet ports but our original design required an OVS with 5 ports. The Linux machine with 7 ports on the pod which we are working on is labeled "Do not Use". Also, in the other pods which are available, we are unable to boot the PCs using our flash drives. The test cases that we recorded in the interim report were according to the previous topology which was implemented on GENI. As we are implementing the test cases in Netlabs now, there is a slight change in the topology, but using the same codes and approach.

7. Enhancement after Interim Report:

Earlier in the project the concept of Topology IP was used,where both the authenticator and server had to ping some arbitrary IP belonging to the same subnet but unassigned to any node. This is required to discover the topology i.e. reachability of authenticator and server from all the OVS switches present in the topology using mapping between (MAC address, DPID) with the port number of each OVS switches. But now a single ping request from authenticator to server is enough for discovering the network topology. As we kept working on the code, we realized this simplification.

8. Future Scope:

The proposed work can further be extended to use multiple web servers and authenticators. In the case when we use multiple web servers and authenticators the controller and authenticator codes need to be modified to ensure that the packet sent to and received from the specific authenticator that has information about the web server to which the client is requesting access to.

Also, the efficiency of the algorithm used to authenticate a client could be improved and be made more secure by using username/password as credentials.

9. Submission Package:

The code package which will be submitted on the day of demo will have 3 folders and 1 readme file. The readme file gives the instructions how to run all the codes. The 3 folders, namely:

1. Authenticator contains

- firewall.c
- Makefile
- test-logs-auth.pdf

2. Controller contains

- app.py
- self-study/app-auth.py
- self-study/self-study-logs netlab.pdf
- test-logs-ovs1.pdf
- test-logs-ovs2.pdf
- test-logs-pox.pdf

3. WebServer contains

- index.php
- image.jpg
- self-study/self-study-server-logs.txt
- self-study/server-auth.php
- self-study/server-no-auth.php

10. References:

➢ http://www.cisco.com/c/en/us/support/docs/security-vpn/remote-authentication-dial-user-service-radius/12433-32.html

➢ http://www.gta.ufrj.br/ftp/gta/TechReports/MFD14.pdf

➢ https://openflow.stanford.edu/display/ONL/POX+Wiki

➢ http://sdnhub.org/tutorials/pox/