

# **COSC 6376: Cloud Computing**

## **Final Project Report**

### **Title**

#### **Network Virtualization in Cloud**

**Creation of an Open Source Testbed using Software Defined Networking**

### **Advisor**

**Dr. Weidong (Larry) Shi**

### **Team**

#### **‘Cloud Weavers’**

**Tushar Chaudhary**

**Rohan Gupta**

**Anil**

**Chaitanya Narayanavaram**

**University Of Houston**  
**Department Of Computer Science**

**Index:**

1. Abstract.....	3
2. Introduction.....	3-5
3. Design and Implementation.....	6
4. Installation.....	7
5. Results and Analysis.....	13
6. Challenges.....	14
7. Conclusions.....	15
8. Individual Contributions.....	16
9. References.....	17
10. Code appendix.....	18-21

## **Abstract:**

This project discusses and implements a networking testbed in the cloud. The testbed is nothing but a virtualized network using remote resources. It serves to enable complex topologies cutting across multiple virtual machines. These virtual machines are in turn laid out across multiple or same physical hosts. Such a highly nested architecture of virtualization is immensely flexible. We next discuss the architecture of the testbed and dissect the reasons behind choice of tools.

To demonstrate the merits of our design we discuss possible applications of our virtual network. We do this by way of an example where the prior knowledge of topology helps mitigate a Distributed Denial of Service (DDoS) attack on a web server running in our topology. The strategy of mitigation is simplistic where we assume that the attacker will be discouraged by repeated throttling of his malicious flows.

We have made heavy use of Software Defined Networking concepts introduced over the duration of the coursework to implement the testbed in its current form. It is the SDN design philosophy that guarantees a centralized view of the network topology. This view of topology can then be utilized by applications running on top of it to amazing effects.

## **Introduction:**

A testbed for virtualization of networks is much needed not just by the academia but by the industry. A testbed in all its glory that is capable of implementing all the communication protocols out there in their entirety, without compromises that often accompany such tools, is invaluable. Such a testbed should be able to mimic even the most complex of topologies without resource constraints and should also allow one to expand without any more effort than the click of a button.

The question asked time and again is 'Why do we need a testbed and since we already have solutions available in this domain where is the need to reinvent the wheel?' A testbed even when not implemented in cloud is an important tool for testers and researchers to enhance their understanding of systems that can be. They give you the liberty to commit and learn from your mistakes without reprisals. Such freedoms are unthinkable in networks of the real world. Secondly they are inexpensive and mimic the real world systems to degrees dependent on the specific testbed being used. Setting up large testbeds for researching say Data Centre Networks (DCN) is beyond means for large institutes let alone curious lone researchers. The same underlying resource from the testbed-host can be reused and reconfigured at the click of a few buttons. This increases the overall hardware resource utilization under the hood and thereby increases efficiency. When implemented in cloud the testbeds offer an extra layer of virtualization and thus control to the community using

it. Clouds when brought into the mix, offer all the efficiency and utilization as well economies of scale advantages, frequently associated with it.

Any testbed to be a meaningful tool for researchers and testers alike, must have the following attributes:

1. Complete protocol support only limited by imagination.
2. Closely mimics the functionality of a real network without the hassle of device by device setup and configuration.
3. The network must be able to expand independent of the underlying host implementation.
4. Crunch for physical resources must never be allowed to happen under reasonable conditions of network load and size.
5. Open Source implementation allowing unrestricted research and reverse engineering.
6. The network must allow customization options for different scenarios.
7. Users should be able to setup and tear down the network remotely.
8. All the configurations should be centralized but not without proper hierarchy of access and authorization.
9. API access for application developers.
10. Ease of use for non-experts.

We will discuss these attributes one by one in context of our design upon introducing the building blocks of this testbed in fine detail. The concept of Software Defined Networking (SDN) is a great enabler. SDN is a revolutionary way of network management wherein a centralized controller or group of controllers in tandem have a complete view of the network topology and activity. This centralization offers a lot to network application developers who can use this information in real time to make intelligent applications. The SDN Controller(s) dictates the behavior of an array of switches using a specialized protocol called Openflow. Openflow allows remote traffic flow management in an SDN. It is the de-facto standard adopted by the industry to broker communication between the traditional switches and SDN controllers. The switches should have Openflow support built-in to allow remote management using SDN paradigm. There comes the biggest bottleneck in the expansion of this idea. Legacy equipment which is widespread has to be overhauled in entirety to support SDN end to end. But such measures are extreme and SDN paradigm can productively co-exist with the traditional networks as well. An example of this is discussed in later sections where we talk about the SDN APIs and case of DDoS attacks.

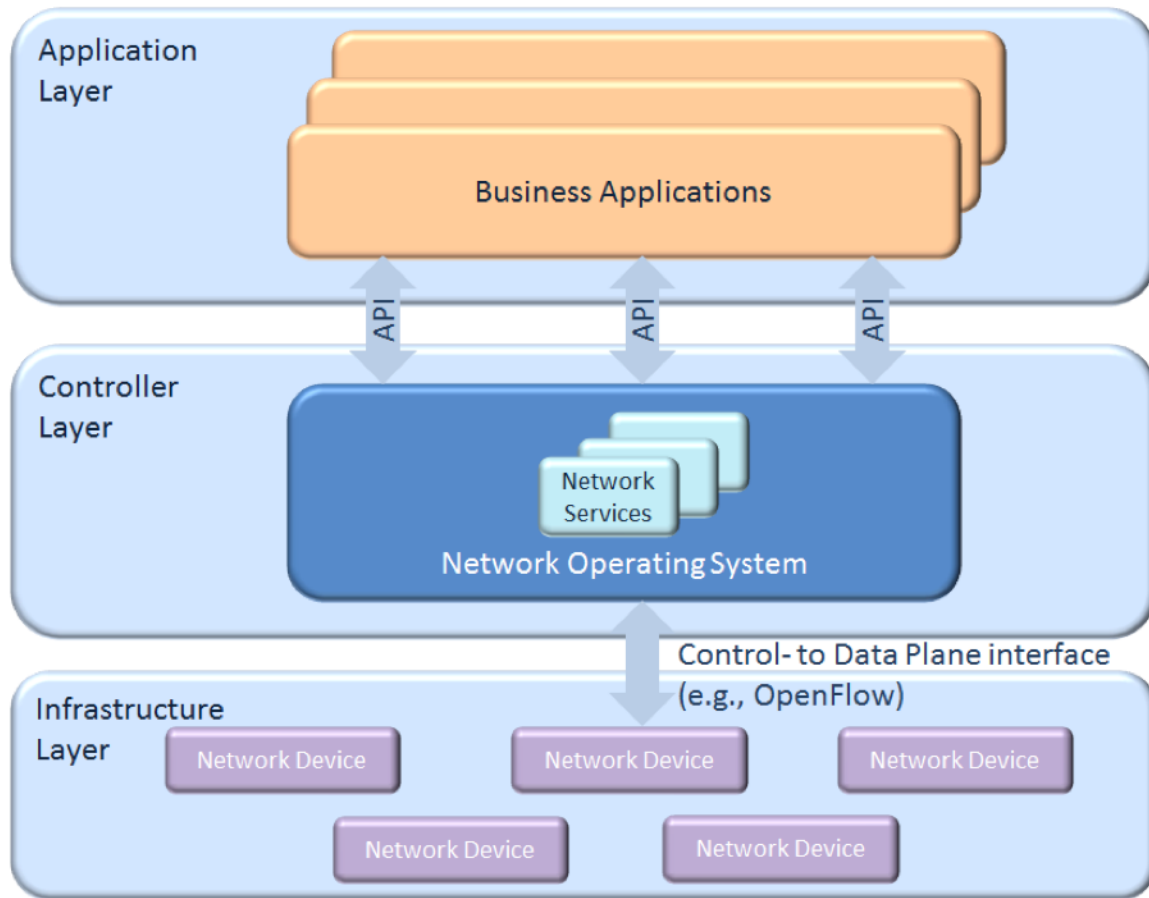


Figure 1: SDN Architecture

The organization of this document is such that it introduces the concepts for a complete novice before moving on to specialized use cases implemented in the project. We first introduce the utility of a testbed and the networking paradigm being virtualized in the project. Next we lay bare the design and architectural building blocks of the project. We then talk about an example application in our virtual testbed that is concerned with DDoS detection and mitigation. We conclude the report with appropriate future work and conclusions of our project endeavor. Finally we talk a little about ourselves and the contributions individual team members made to achieve the goal of the project.

## Design and Implementation:

**Design:** The architecture referenced in the figure 2 is a high level view of virtualization layers within our implemented system.

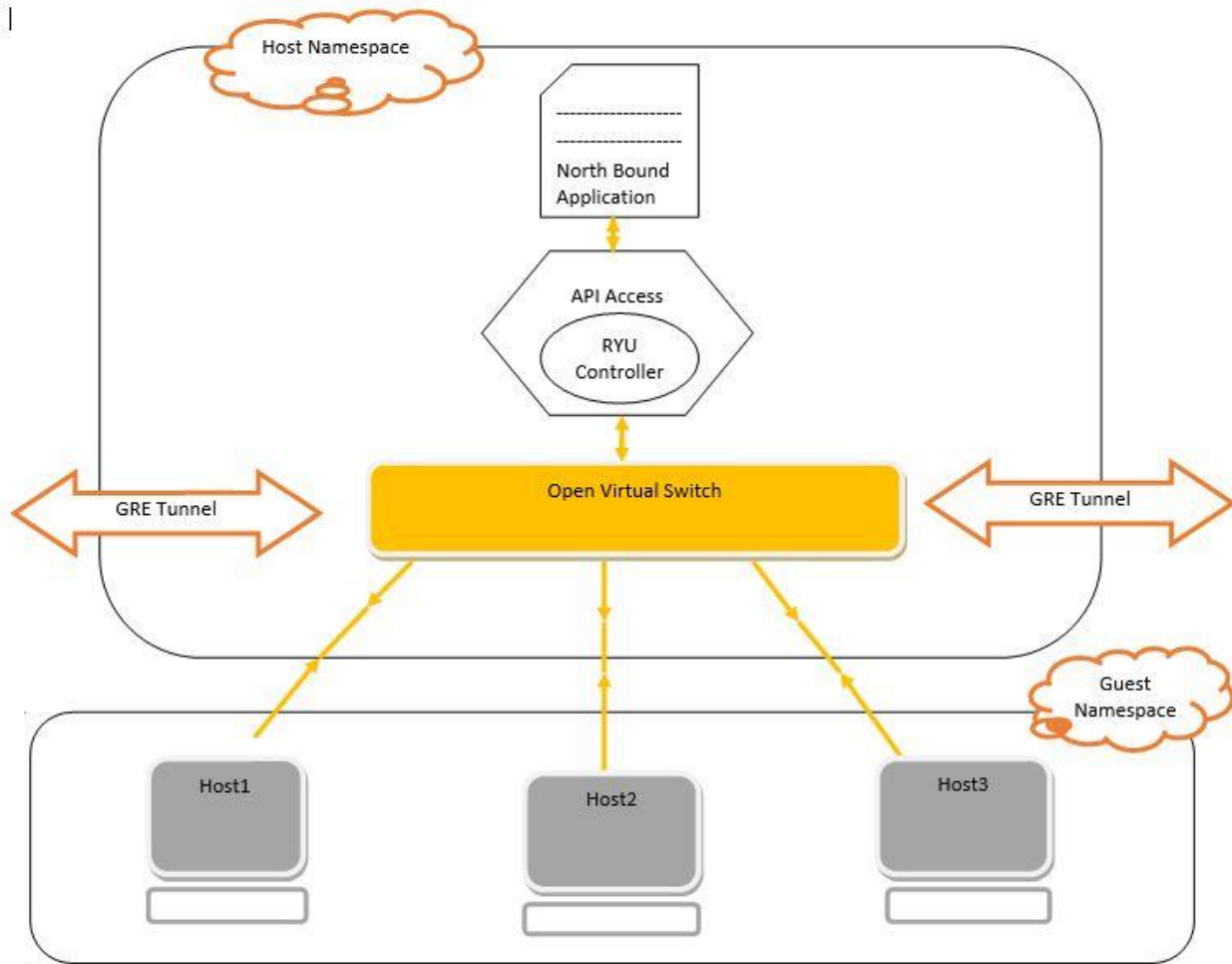


Figure 2: Architecture of Testbed

Let us discuss this architecture element by element. The exhaustive set of elements used top-down in the design are mentioned below:

1. *Physical Machines or Hardware:* We have used our individual notebook PCs for this purpose but we can as well use large server farms or hosting farms being offered by Amazon or Google.
2. *Emulation Platforms:* If the case is of an individually owned machine then we can use Virtual box / VM ware as the emulation platform. If a server farm or a cluster is being used we may have XEN server running on top of it. If no infrastructure is available we may jump directly to the next component.

3. *Multiple Guest VMs*: On top of physical server farms or real hardware we run VM instances. The role played by these instances is that of a backup capacity in case we run out of resources on a VM while expanding our test topology. When no kind of physical infrastructure is available multiple Amazon EC2 instances can be used.
4. *Network Nodes in Mininet*: Each node in the network topology can be simulated in software using Linux IP-routing namespaces. These namespaces can be created using a tedious process in command line mode or a wrapper program such as 'Mininet' can be utilized to easily do this for you on larger scale.
5. *Open Virtual Switch (OVS)*: A fully virtual software emulated switch implemented in Linux kernel.
6. *Openflow Controller or SDN Controller*: The controller of choice in this case is the python based 'RYU'. Since Mininet topologies are already implemented in Python, therefore using RYU makes the system gel together seamlessly.

## Installation

1. The environment setup includes installing a virtual Linux environment (VM Virtual Box) in our case) as the first step. Download and install this software on the system.
2. To download & run Mininet in the virtual box:

```
git clone git://github.com/mininet/mininet
cd mininet
mininet/util/install.sh -a
```

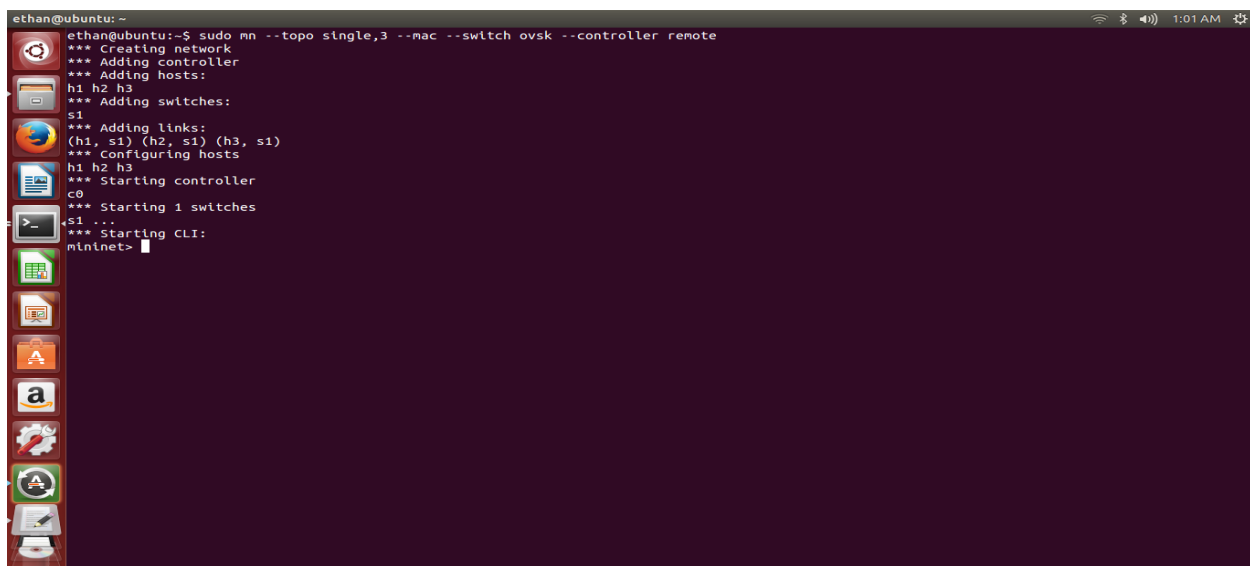


Figure 3: Running Mininet

### 3. Installing RYU controller on Mininet:

```
git clone git://github.com/osrg/ryu.git **download from source**  
cd ryu  
PYTHONPATH=. ./bin/ryu-manager ryu/app/simple_switch.py **Running RYU**
```

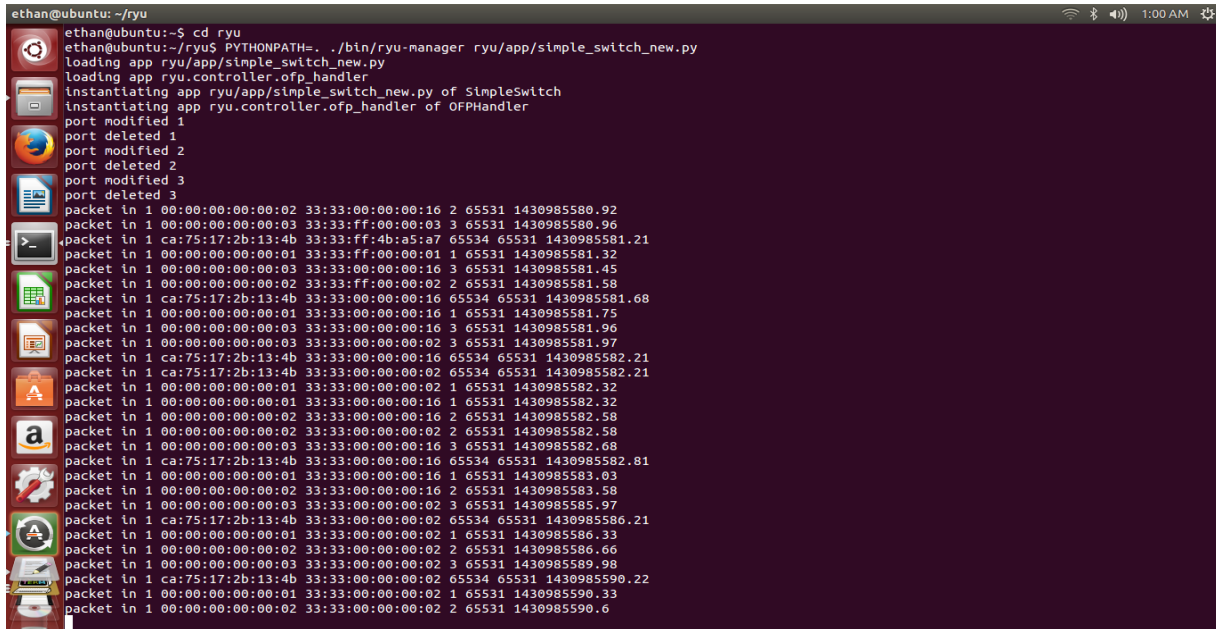


Figure 4: Running RYU Controller

The overall design has a Virtual Box Platform installed on a physical machine. The guest OS for the Virtual Box being Ubuntu Linux 14.04. The same setup can be replicated on an Amazon-EC2 instance without having to setup a virtual box host first. Once the setup is ready we installed Mininet and Ryu SDN controller on the Guest OS. Mininet wrapper can generate multiple switch and host topologies on the Guest OS. Each Mininet topology can be tunneled using GRE with another topology running elsewhere. This scheme of network virtualization is thus extensible.

### 4. Installation of Apache Web Server Version 2.13:

```
Download $ lynx http://httpd.apache.org/download.cgi  
Extract  $ gzip -d httpd-NN.tar.gz  
           $ tar xvf httpd-NN.tar  
           $ cd httpd-NN  
Configure $ ./configure --prefix=PREFIX  
Compile  $ make  
Install   $ make install  
Customize $ vi PREFIX/conf/httpd.conf  
Test      $ PREFIX/bin/apachectl -k start
```



Once Apache Web server is running you can check the status of HTTPd service by opening the link: <http://localhost/> in your browser, if successful the following message should be displayed:

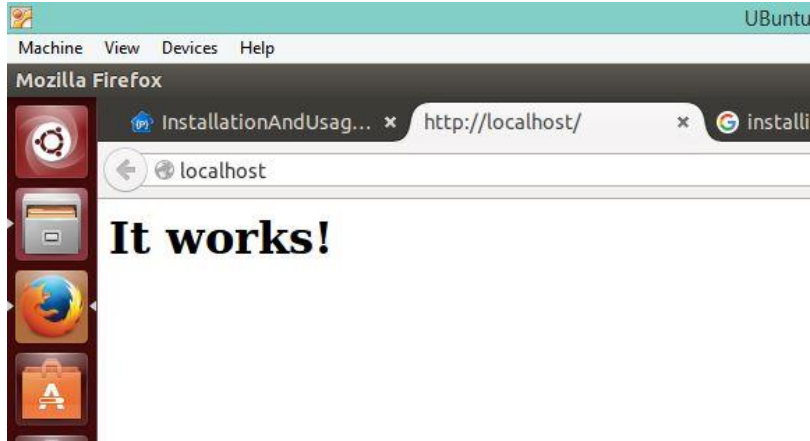


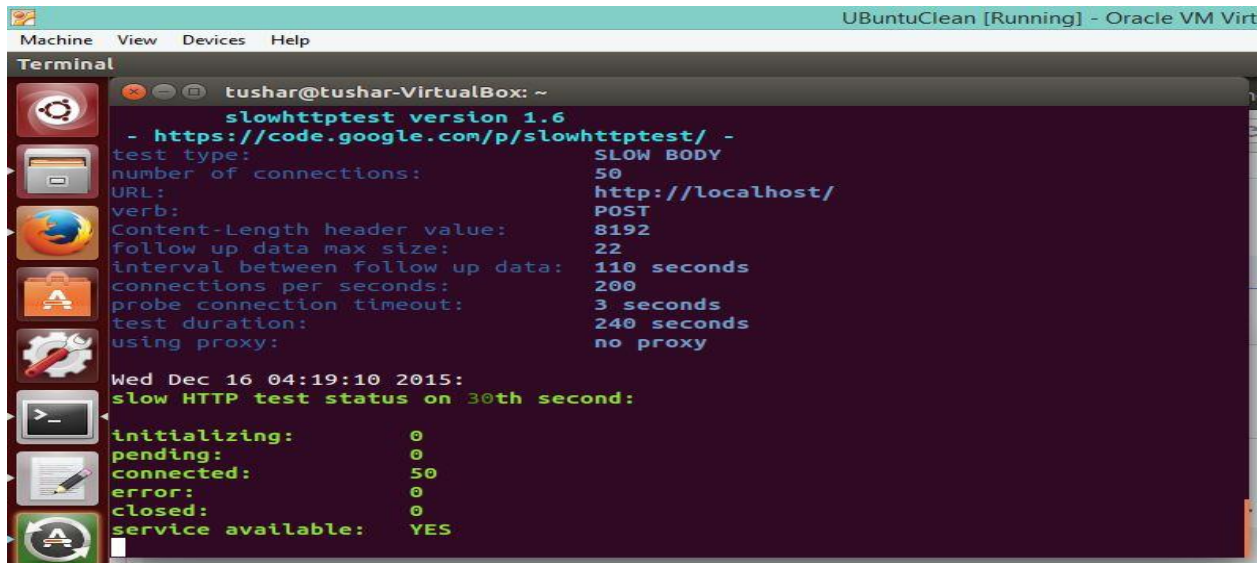
Figure 5: Confirmation of Apache Webserver 2.13 Running in Background

#### 5. SlowHTTPtest Tool ver 1.6 Installation:

```
$ tar -xzf slowhttptest-x.x.tar.gz
$ cd slowhttptest-x.x
$ ./configure --prefix=PREFIX
$ make
$ sudo make install
```

Once installed the tool can be run as follows:

```
Sudo slowhttptest -c 1000 -B -g -o my_body -i 110 -r 200 -s 8192 -u
https://localhost/ -x 10 -p 3
```



```
tushar@tushar-VirtualBox: ~  
slowhttptest version 1.6  
- https://code.google.com/p/slowhttptest/ -  
test type: SLOW BODY  
number of connections: 50  
URL: http://localhost/  
verb: POST  
Content-Length header value: 8192  
follow up data max size: 22  
interval between follow up data: 110 seconds  
connections per seconds: 200  
probe connection timeout: 3 seconds  
test duration: 240 seconds  
using proxy: no proxy  
  
Wed Dec 16 04:19:10 2015:  
slow HTTP test status on 30th second:  
  
initializing: 0  
pending: 0  
connected: 50  
error: 0  
closed: 0  
service available: YES
```

Figure 6: SlowHTTPtest tool running for 240 seconds window

A typical workflow in our testbed can be explained in context of figure 3.

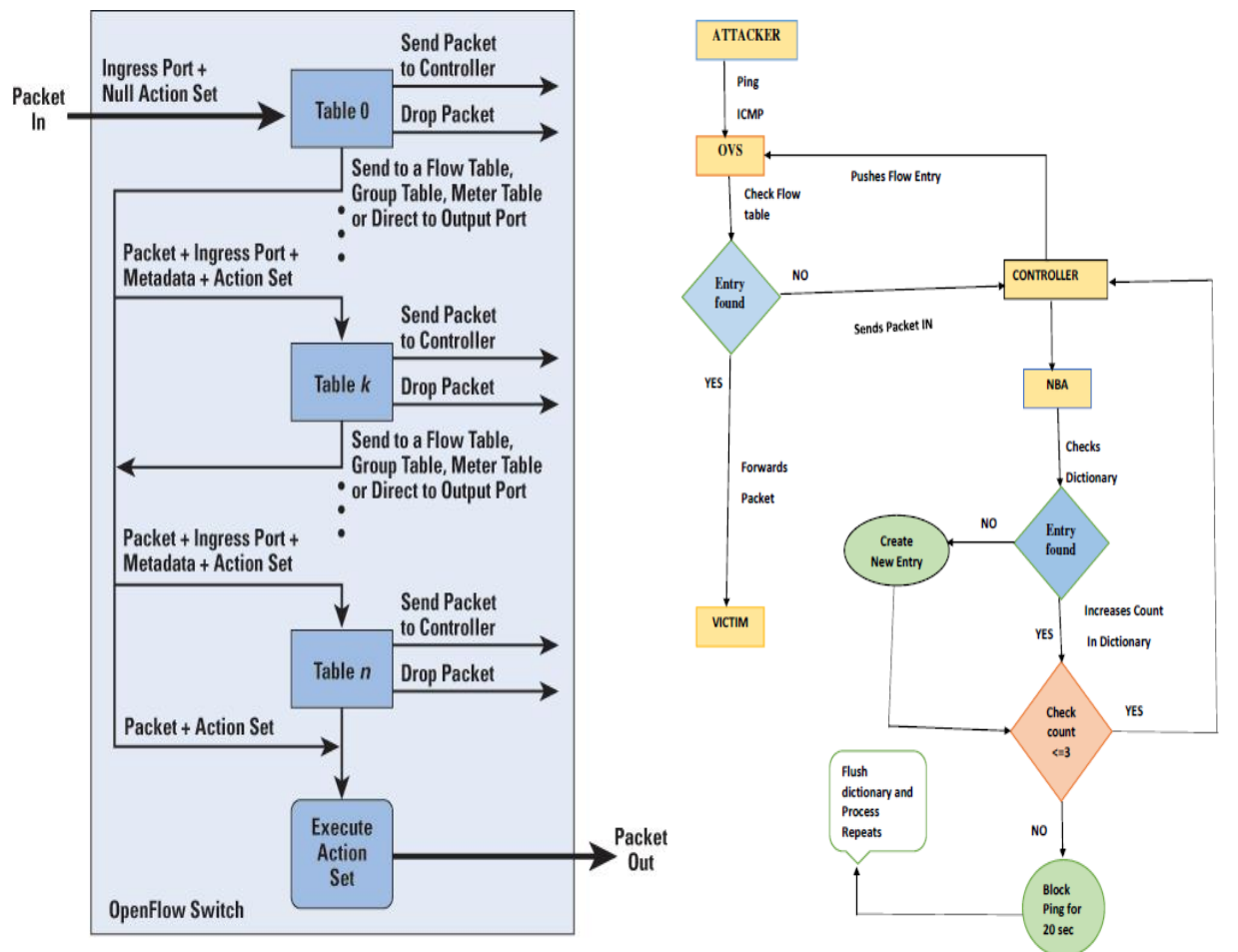
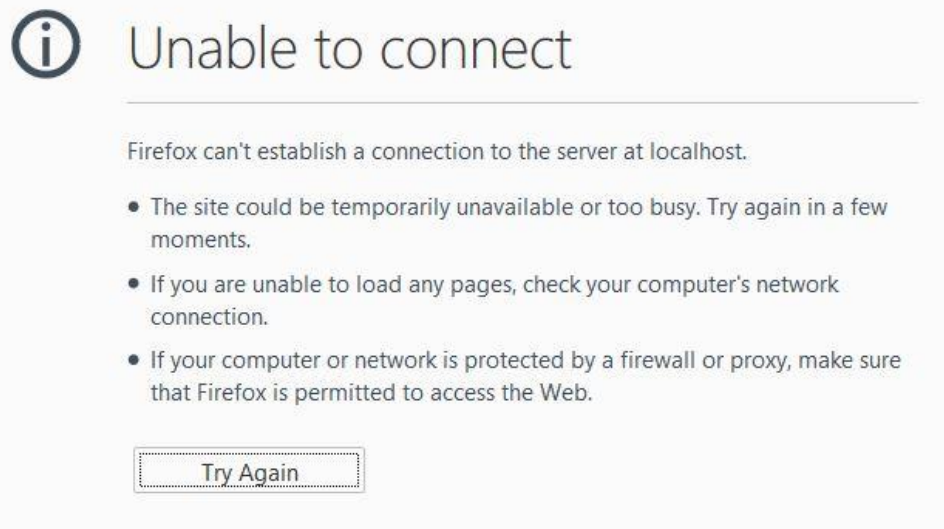


Figure 7: Standard SDN Workflow

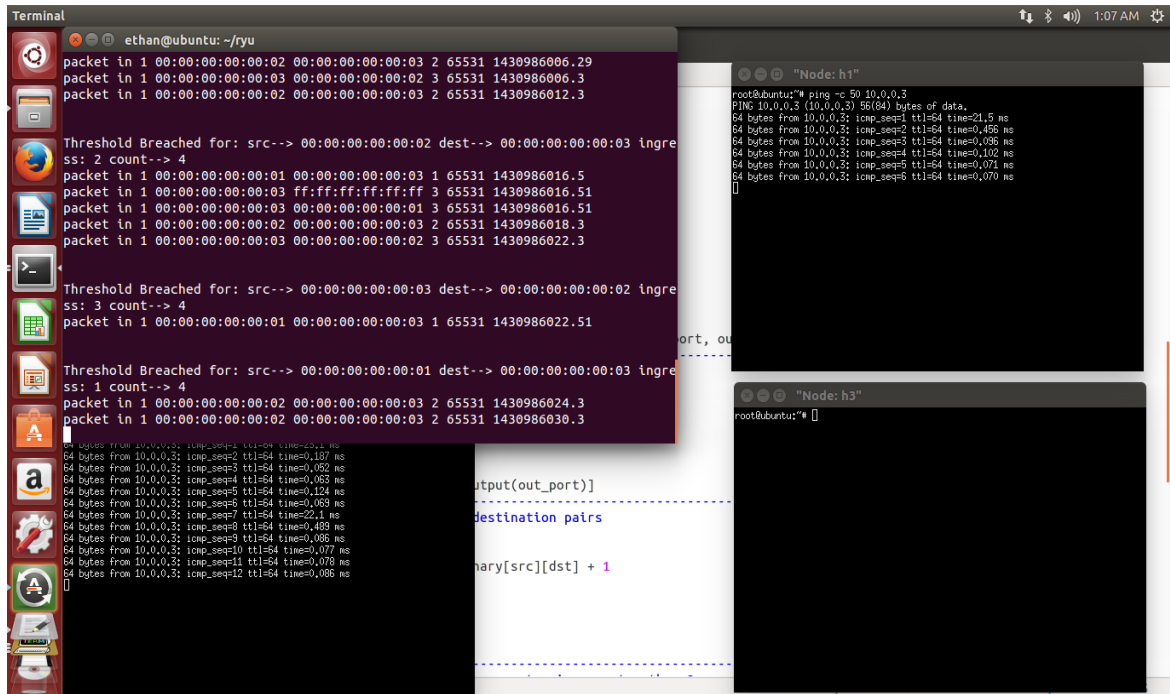
**Implementation:** We have used Mininet as an emulator to implement the network topology as shown in figure 1. The scenario consist of an OVS switch 1.0, RYU controller 3.2 and 3-hosts. Host 1 and 2 act as attackers while host 3 will act as a victim. The entire scenario is implemented on a V-BOX (Virtual Box Instance).

- Host 1 and Host 2 (attackers) directing a flood of requests at Host 3 (victim) using the tool slowHTTPtest. Host 1 and Host 2 are running instances of slowHTTPtest tool version 1.6. The Host 3 on the other hand is running an Apache Webserver Httpd service instance on Localhost: Port 80. Host 3 responds to each of the connection requests received from the slowHTTPtest tool and if there are more requests per second than can be handled by this server it becomes unresponsive to all further requests effectively making it in accessible. Hence the DDoS simulation.



- A Packet IN generated for the initial ping from host1 and host2 (through the OVS) directed towards the victim; the NB application extracts following data (record): Source IP + Source MAC + Port Requested and Connection Request Count. Store this record to a dictionary with an associated Timestamp.
- We create a condition such that the flow entry for the http request is pushed to the OVS for only a short period of time (in milliseconds) and then the flow expires. The pings directly pass through the switch during this time.
- This process repeats for subsequent http requests (Packet Ins).
- The NB application updates the dictionary every time a Packet In arrives at the controller and allows a maximum of 3 entries for a specific Source IP, avoiding a possible flooding attack. Any request from that same IP after the count equals 3 is denied by the controller for the next 5 minutes (=hard timeout) and simply dropped. The dictionary is flushed every 5 minutes.

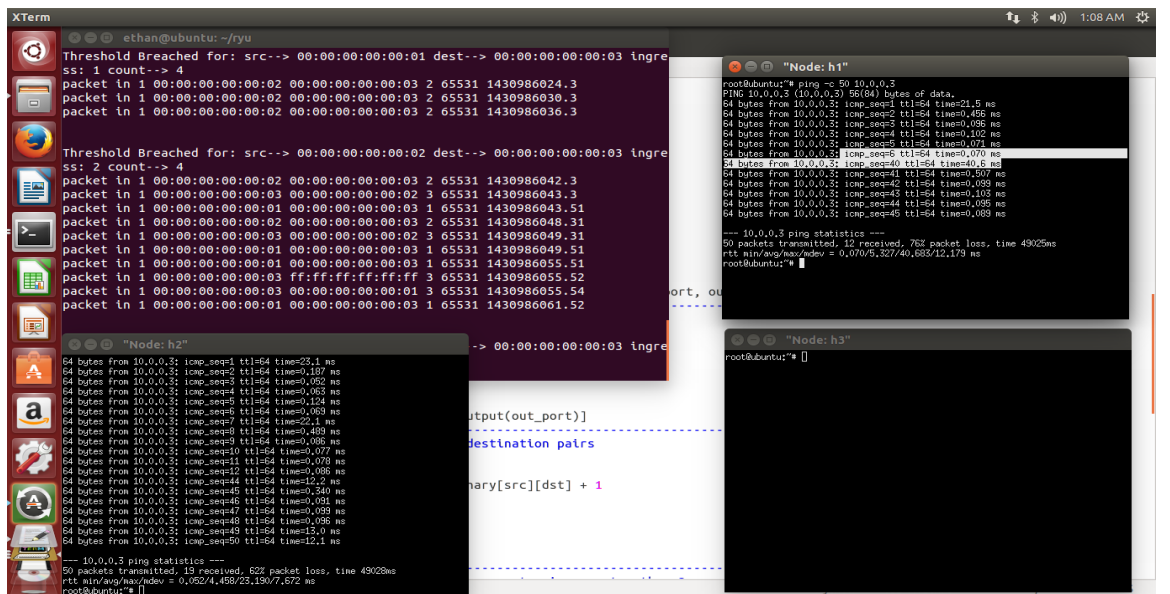
The following figure 8 shows the application running on top of RYU controller in real time during the attack. It is clearly visible that the threshold decided by the administrator is being breached by the attacking hosts. It thereby blocks all traffic coming from those sources to the attacked port 80 on our localhost server running on victim.



The terminal window displays network traffic logs and threshold breach notifications. The logs show packets from various sources (e.g., 00:00:00:00:00:02, 00:00:00:00:00:03) and their corresponding sequence numbers and ports. The threshold breach notifications indicate that the threshold has been breached for specific source-destination pairs, and the count of breaches is increasing (e.g., ss: 2 count--> 4, ss: 3 count--> 4, ss: 1 count--> 4).

Figure 9: Threshold Breach Detected by Northbound Application running on RYU

- After the dictionary is flushed out the procedure repeats all over again.



The terminal window displays network traffic logs and threshold breach notifications after a dictionary flush. The logs show packets from various sources (e.g., 00:00:00:00:00:01, 00:00:00:00:00:02, 00:00:00:00:00:03) and their corresponding sequence numbers and ports. The threshold breach notifications indicate that the threshold has been breached for specific source-destination pairs, and the count of breaches is increasing (e.g., ss: 1 count--> 4, ss: 2 count--> 4, ss: 1 count--> 4).

Figure 10: Attackers Blocked After Breach

Distributed Denial of Service (DDoS) flooding attacks are one of the biggest concerns for security professionals. DDoS flooding attacks are typically explicit attempts to disrupt legitimate users' access to services. Developing a comprehensive defense mechanism against identified and anticipated DDoS flooding attacks is a desired goal of the intrusion detection and prevention research community. However, the development of such a mechanism requires a comprehensive understanding of the problem and the techniques that have been used thus far in preventing, detecting, and responding to various DDoS flooding attacks.

## Results and Analysis:

The figure 5 depicts status of request handled by apache web server under DDoS attack from Host 1 and Host 2. Python application running on top of Ryu Controller keeps track of port requested by the slowHTTPtest tool and blocks traffic originating from the hosts H1 and H2 if the number of requests received per second is breached. See Appendix for the python script.

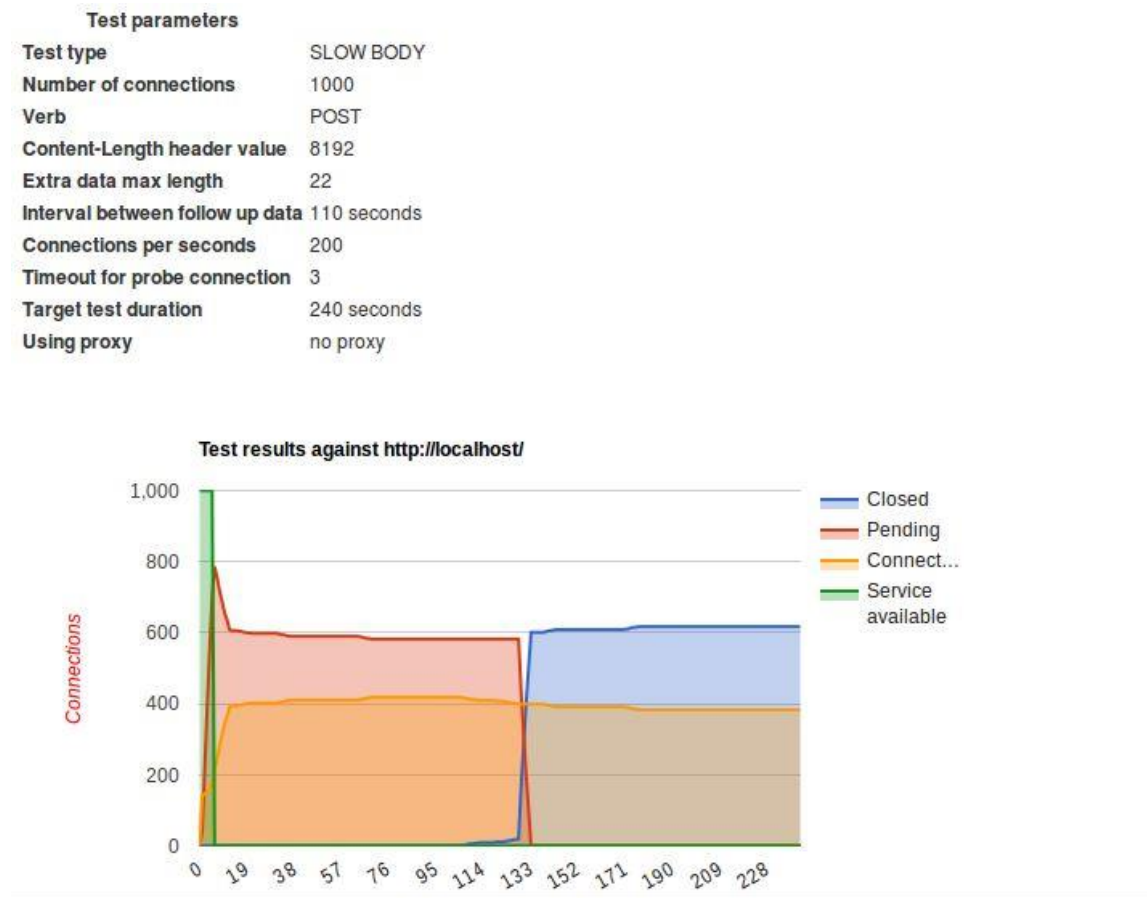


Figure 11: Results without monitoring by Northbound Application

When the script runs on top of Ryu controller monitoring the traffic flow through OVS switch it keeps the overflowing http requests under check by blocking sources of those requests for 20 seconds at a time. The following graph in figure 6 thus results:

Test parameters	
Test type	SLOW HEADERS
Number of connections	50
Verb	GET
Content-Length header value	4096
Extra data max length	68
Interval between follow up data	10 seconds
Connections per seconds	50
Timeout for probe connection	5
Target test duration	240 seconds
Using proxy	no proxy

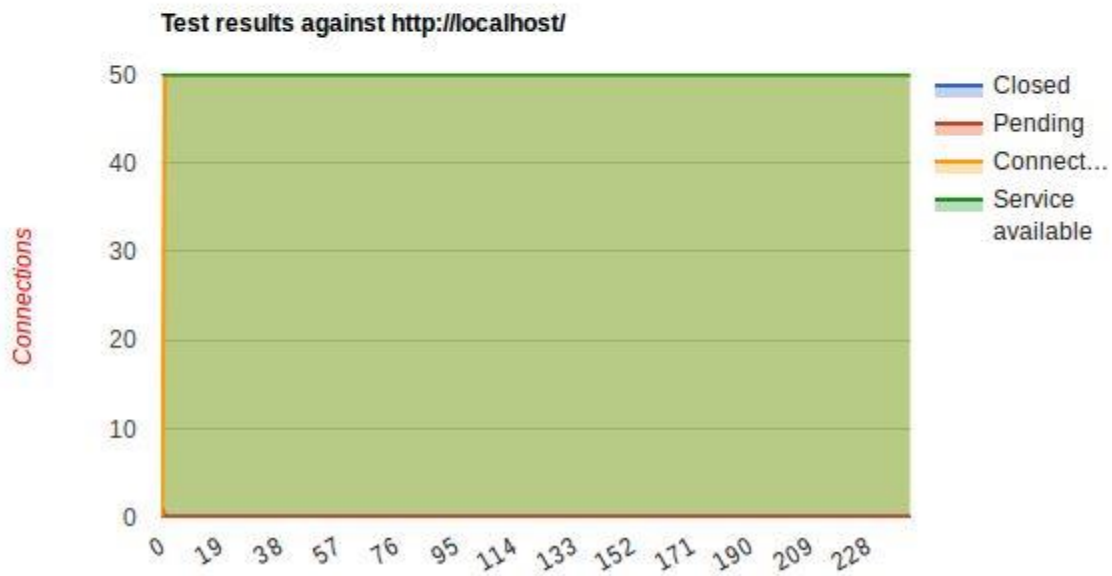


Figure 12: Results when the Northbound Application blocks the DDoS Sources

## Challenges:

1. Setting up GRE Tunnel between two Mininet instances took time. But it made the project extensible to other physical or Virtual Machines.
2. Setting up Apache Web Server version 2.13 since the latest apache web servers are impervious to slowHTTPtest tool originating attacks.
3. The DDoS script in the previous form was unable to detect protocol headers. It was fixed after study of Ryu APIs in detail.
4. The hard timeout is limited to 1 second lower limit which makes it impossible to handle DDoS attacks which last for a fraction of a second.

5. The detection of source of attack is right now through OVS' port number and mac address. This may cause generalization and may block more interfaces on OVS than required in case of hierarchical topology.
6. Also the biggest challenge faced was in setting up of slowHTTPtest tool on Mininet shells.

#### **Security Constraints:**

- There is a chance of losing legitimate traffic in this approach. Flows that are long enough such as file download requests can be mistook as an attack and blocked.
- A man-in-the-middle attack can occur generating spurious packet streams with ever changing packet headers to avoid detection.
- If there are sufficient number of compromised nodes in the botnet then the attacker can play with the timeout values by rotating between the available attacker pools in a round robin fashion.

### **Conclusions:**

The testbed and DDoS application example proves the merit of the architecture against the attributes listed at the outset. The testbed so developed uses exclusively open source components. The testbed is free of cost and free for modification by anyone. The project also takes care of the fact that functionality of the underlying network in the testbed must mimic the real world setup closely as possible. Protocol support is complete and no compromises have been made on that front.

The project helped us grasp the concept of virtualization in greater detail.

- The whole project was based on open source implementation.
- This project is not limited by hardware limitations.
- Extensible beyond physical machine boundaries because of the implementation using virtual machines.
- Protocol support limited only by imagination.
- SDN standard is widespread already.



## Individual Contribution:

Task	Responsibility	Timeline
Project Proposal	Tushar	September 15
Literature Review	Anil Chaitanya Rohan Tushar	September 15 - October 5
Project Plan and Timeline	Rohan	October 10
Initial Design of Testbed	Tushar	October 10 – October 30
Tool Selection	Chaitanya	October 30 – November 10
Configuration and Setup	Tushar Rohan	November 10 - November 20
Changes after Advisor Feedback	Anil	November 21
Python Northbound Application Development	Tushar	November 25
Python Script Debugging	Rohan Anil Chaitanya	November 25 - November 30
Presentation Slides	Rohan Anil	December 5
Demo Recording	Tushar Chaitanya	December 5 - December 7
Feedback Modification (SlowHTTPtest Tool)	<i>SlowHTTPtest Tool Installation:</i> Tushar  <i>Apache Webserver:</i> Tushar Rohan	December 10
Project Report Draft	Tushar	December 12
Project Report Finalizing and Proof-Reading	Anil Chaitanya Rohan	December 16



## **9. References:**

1. <http://ryu.readthedocs.org/en/latest/index.html>
2. <https://github.com/mininet/mininet/wiki/Documentation>
3. <https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.0.0.pdf>
4. [http://ryu.readthedocs.org/en/latest/library\\_packet.html](http://ryu.readthedocs.org/en/latest/library_packet.html)
5. [https://github.com/osrg/ryu/wiki/OpenFlow\\_Tutorial](https://github.com/osrg/ryu/wiki/OpenFlow_Tutorial)
6. [https://www.virtualbox.org/wiki/User\\_HOWTOS](https://www.virtualbox.org/wiki/User_HOWTOS)

## Appendix I:

### Code:

```
# Copyright (C) 2011 Nippon Telegraph and Telephone Corporation.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing,
software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
# implied.
# See the License for the specific language governing permissions
and
# limitations under the License.

"""
An OpenFlow 1.0 L2 learning switch implementation.
"""

import logging
import struct
import time
from ryu.base import app_manager
from ryu.controller import mac_to_port
from ryu.controller import ofp_event
from ryu.controller.handler import MAIN_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.ofproto import ofproto_v1_0
from ryu.lib.mac import haddr_to_bin
from ryu.lib.packet import packet
from ryu.lib.packet import ethernet
from ryu.lib.packet import icmp

counter = 0 #defunct code variable
dictionary = {}
class SimpleSwitch(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_0.OFP_VERSION]
    def __init__(self, *args, **kwargs):
        super(SimpleSwitch, self).__init__(*args, **kwargs)
        self.mac_to_port = {}
#-----
#-----
    #actual function that adds flows
    def add_flow(self, datapath, in_port, dst, actions, timeout):
        ofproto = datapath.ofproto
```

**University Of Houston**  
**Department Of Computer Science**

```
match = datapath.ofproto_parser.OFPMatch(
    in_port=in_port, dl_dst=haddr_to_bin(dst))

mod = datapath.ofproto_parser.OFPFlowMod(
    datapath=datapath, match=match, cookie=0,
    command=ofproto.OFPFC_ADD, idle_timeout=0,
hard_timeout=timeout,
    priority=ofproto.OFP_DEFAULT_PRIORITY,
    flags=ofproto.OFPFF_SEND_FLOW_REM, actions=actions)
datapath.send_msg(mod)
#-----
-----
    @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
#decorator for event association
    def _packet_in_handler(self, ev):
        global dictionary #data structure for source destination mac
pairs
        h_timeout=5
        h2_timeout=20
#-----
-----
        # generating time stamp in terms of "seconds since epoch"
        ts=time.time()

        msg = ev.msg #message data structure
        datapath = msg.datapath # switch details
        ofproto = datapath.ofproto # negotiated protocol version
        print "protocol",ofproto

        pkt = packet.Packet(msg.data)
        eth = pkt.get_protocol(ethernet.ethernet)
        dst = eth.dst
        src = eth.src
        dpid = datapath.id #dpid extraction of connected switch

        self.mac_to_port.setdefault(dpid, {})

        out_port = ofproto.OFPP_FLOOD

        self.logger.info("packet in %s %s %s %s %s %s", dpid, src,
dst, msg.in_port, out_port, ts)
#-----
-----
        # learn a mac address to avoid FLOOD next time.
        self.mac_to_port[dpid][src] = msg.in_port

        if dst in self.mac_to_port[dpid]:
            out_port = self.mac_to_port[dpid][dst]
        else:
            out_port = ofproto.OFPP_FLOOD
        actions =
[datapath.ofproto_parser.OFPActionOutput(out_port)]
```

**University Of Houston**  
**Department Of Computer Science**

```
#-----
#-----
# Dynamic data structure to store the source destination pairs
if src in dictionary.keys():
    if dst in dictionary[src].keys():
        dictionary[src][dst] = dictionary[src][dst] + 1
    else:
        dictionary[src][dst] = 1
else:
    dictionary[src]={dst:1}
'''print "---->",dictionary'''
#-----
#-----
# Traversing Dictionary to find out flow push request-pairs
greater than 3
for i in dictionary.keys():
    for j in dictionary[i].keys():
        if dictionary[i][j] > 3:
            print "\n"
            print "Threshold Breached for: src--
>",i,"dest-->",j,"ingress:",msg.in_port,"count-->",dictionary[i][j]
            if out_port != ofproto.OFPP_FLOOD:
                actions = []
                self.add_flow(datapath, msg.in_port, j,
actions, h2_timeout)
                dictionary[i][j] = 0
            else:
                if out_port != ofproto.OFPP_FLOOD:
                    self.add_flow(datapath, msg.in_port, j,
actions, h_timeout)
#-----
#-----
# Defunct Code
'''if counter <= 6:
    if out_port != ofproto.OFPP_FLOOD:
        self.add_flow(datapath, msg.in_port, dst,
actions,h_timeout)
        counter=counter+1
if counter > 6:
    if out_port != ofproto.OFPP_FLOOD:
        actions = []
        self.add_flow(datapath, msg.in_port, dst,
actions,h2_timeout)
        counter = 0'''

data = None
if msg.buffer_id == ofproto.OFP_NO_BUFFER:
    data = msg.data
#-----
#-----
# Packet_Out parser
out = datapath.ofproto_parser.OFPPacketOut(
```

**University Of Houston**  
**Department Of Computer Science**

```
        datapath=datapath, buffer_id=msg.buffer_id,
in_port=msg.in_port,
        actions=actions, data=data)
    datapath.send_msg(out)
#-----
-----
    @set_ev_cls(ofp_event.EventOFPPortStatus, MAIN_DISPATCHER)
    def _port_status_handler(self, ev):
        msg = ev.msg
        reason = msg.reason
        port_no = msg.desc.port_no

        ofproto = msg.datapath.ofproto
        if reason == ofproto.OFPPR_ADD:
            self.logger.info("port added %s", port_no)
        elif reason == ofproto.OFPPR_DELETE:
            self.logger.info("port deleted %s", port_no)
        elif reason == ofproto.OFPPR_MODIFY:
            self.logger.info("port modified %s", port_no)
        else:
            self.logger.info("Illegal port state %s %s", port_no,
reason)
```