Scalability Testing of SDN Controllers and Switches

Masters Thesis Project Stage I Report

submitted in partial fulfilment of the requirements for the degree of

Master of Technology

by

Chinmaya Mhatre
Roll No: 143050063

under the guidance of

Prof. Mythili Vutukuru

Department of Computer Science and Engineering
Indian Institute of Technology, Bombay

October 2015
Acknowledgements

I would like to express my sincere gratitude to my advisor Prof. Mythili Vutukuru for her continuous support and motivation. I would also like to thank my colleagues Sadagopan, Aman and Sunny for sharing their knowledge and helping me out. Also I thank my fellow lab-mates and my family for their constant support and motivation.
## Contents

1 Introduction .................................................. 5  
  1.1 Motivation ................................................. 5  
  1.2 Contributions .............................................. 5  

2 Background ................................................... 6  

3 Controller Performance Testing ............................... 8  
  3.0.1 Experiment Scenario and Setup ......................... 9  
  3.0.2 Results & Analysis ....................................... 9  

4 Switch Performance Testing .................................. 11  
  4.1 OpenvSwitch without OpenFlow Rules ..................... 11  
  4.1.1 Experiment using Virtual Machines ..................... 11  
  4.1.2 Experiment using Network Namespaces .................. 13  
  4.2 OpenvSwitch with Openflow Rules ......................... 15  
  4.2.1 Experiment with One Flow ............................... 15  
  4.2.2 Experiment with Multiple Flows ......................... 17  
  4.3 Flow Initiations Handled by OpenvSwitch ................. 18  
  4.3.1 Experiment ................................................ 19  

5 Conclusion & Future Work .................................. 21
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Structure of OpenFlow Environment</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>Components and Interfaces of OpenvSwitch</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>Scenario for Controller Testing</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>Throughput and Latency Graphs with Number of Switches Fixed to 32</td>
<td>10</td>
</tr>
<tr>
<td>5</td>
<td>Throughput and Latency Graphs with Number of Unique MAC Address Fixed to 100000</td>
<td>10</td>
</tr>
<tr>
<td>6</td>
<td>Scenario for OpenvSwitch testing without flow rules using virtual machines</td>
<td>12</td>
</tr>
<tr>
<td>7</td>
<td>Through Latency and System Performance Graphs for Experiment using Virtual Machines</td>
<td>13</td>
</tr>
<tr>
<td>8</td>
<td>Scenario for OpenvSwitch testing without flow rules using network namespaces</td>
<td>14</td>
</tr>
<tr>
<td>9</td>
<td>Through Latency and System Performance Graphs for Experiment using Network Namespaces</td>
<td>15</td>
</tr>
<tr>
<td>10</td>
<td>Scenario for OpenvSwitch Testing with Flow Rules</td>
<td>16</td>
</tr>
<tr>
<td>11</td>
<td>Through Latency and System Performance Graphs for Experiment with One Flow Through OpenvSwitch</td>
<td>17</td>
</tr>
<tr>
<td>12</td>
<td>Through Latency and System Performance Graphs for Experiment with One Multiple Through OpenvSwitch</td>
<td>18</td>
</tr>
<tr>
<td>13</td>
<td>Scenario for OpenvSwitch testing for Flow Initiations</td>
<td>20</td>
</tr>
<tr>
<td>14</td>
<td>Graphs for Experiment to Test Flow InitiationsHandled by OpenvSwitch</td>
<td>20</td>
</tr>
</tbody>
</table>
1. Introduction

Software Defined Networking (SDN) and OpenFlow have emerged as a new paradigm of networking. Software Defined Networking is changing how we design, build, and operate networks to achieve business agility. With SDN, networks are no longer closed, proprietary, and difficult to program. SDN gives network owners and operators more control of their infrastructure, allowing customization and optimization, and reducing the overall capital and operational costs. SDN model is to move network’s intelligence out from the packet switching devices and to put it into the logically centralized controller. The forwarding decisions are done first in the controller, and then moves down to the overseen switches which simply execute these decisions. This gives us a lot of benefits like global controlling and viewing whole network at a time.

1.1. Motivation

Considering the benefits, as SDN networks are being implemented in industry for different applications, we wanted to test different SDN network components for their scalability. The SDN network has controller which takes all the decision about forwarding packets in the network and OpenFlow compatible switches which forward the packets according to the flow rules fed by the SDN controller. So, unlike normal networks where there are many different types of switching devices having both control and data plane, SDN network has only SDN controller and OpenFlow compatible switches. So, we wanted to test these both SDN network devices to see how they perform when the network get scaled up. There are few papers which test for performance of either the controller or the OpenFlow compatible switches but we didn’t found any work on the testing of the SDN controllers and OpenFlow switches working together. So, we wanted to test these different controllers and OpenFlow compatible switches independently, as well as when they work communicating between each other as in most of the practical scenarios.

1.2. Contributions

So first we tested different SDN controllers to see how they perform, in terms of requests satisfied per second and in terms of the time required to serve a request, when SDN network controlled by that controller scales. We then tested OpenvSwitch, a well known open source software switch, for the amount of data traffic it can process. Then to test more practical scenarios we wanted to test how OpenvSwitch performs when it is connected to SDN controller. So, we performed experiments to see how OpenvSwitch performs when a different amount of data traffic load is passed through
it and also when different number of flows are being handled by OpenvSwitch. Then we tested OpenvSwitch for many messages to the controller it can generate, to ask for new flow rules, when large number of packets corresponding to new flows for which there are no flow rules in flow tables, comes to the OpenvSwitch. From the experiments we got that all of the controllers can satisfy millions of requests from the switches. These number are different for different controllers and we have compared them in below sections. For OpenvSwitch though we couldn’t succeed in making it bottleneck due to limited CPU available on our testing machine, we observed that OpenvSwitch can handle atleast around 27Gbps of data traffic.

2. Background

SDN/NFV environment have OpenFlow enabled switches and controller as the basic networking components. End hosts in form of virtual machines or physical machines are connected to switches. Switches are connected to each other to form the network forwarding plane. Each switch talks to the controller about how to forward different packets in the network. Figure 1 gives the basic structure of the OpenFlow environment.

![Figure 1: Structure of OpenFlow Environment](image)

End user devices that are source and destination of packet flows are connected to the OpenFlow compatible switches. When a particular source host initiates a
flow to some other destination host in the network, the switches along the path doesn’t know how to forward these packets as no control logic is implemented at these switches. So, the first packet for each new flow for which there is no matching flow table entry at the switch, will be sent to the controller. All OpenFlow switches are connected to controller and they talk to each other using OpenFlow protocol running over secure socket layer (SSL). A central SDN controller performs all complex functions, including routing, naming, policy declaration, and security checks. This plane constitutes the SDN Control Plane. The SDN controller decides how the packet should be forwarded and sends OpenFlow switch reply with flow-mod entry. This flow-mod entry form controller is saved in flow table at the switch. Now, when the subsequent packets of that flow comes at the OpenFlow switch, it knows how to forward them, and takes the according action mentioned in the action part of flow entry corresponding to that flow. Typically each switch has series of switches to manage the flows of packets through the switch.

![figure 2: Components and Interfaces of OpenvSwitch](image)

OpenvSwitch[1] is a open source implementation of a OpenFlow compatible virtual switch. OpenvSwitch contains two major components to direct the packet forwarding. First component is a userspace daemon ovs-vswitchd, which is same for all operating systems. The other major component is datapath kernel module which is written specific for each operating system for performance. Figure 2 gives basic components and interfaces of OpenvSwitch. The datapath module in the kernel receives the packet first. If already instructed by ovs-vswitchd about how to handle packet of this type it takes the corresponding action, but if not, it sends the packet to the ovs-vswitchd. ovs-vswitchd figures out how that packet should be handled by contacting SDN controller and then instructs datapath kernel module about same. ovs-vswitchd also tells datapath kernel module to cache actions for that flow to handle future packets corresponding to that flow.
3. Controller Performance Testing

SDN controller is an application which manages flow in the software defined networking. SDN controller constitutes control plane of the network. So, SDN controller has different controller applications running on one side and different network devices on the other side. Controller basically runs in two modes, reactive and proactive. In proactive mode rules are already installed in the switches.

In reactive mode when a request corresponding to new flow comes for the controller in form of PACKET_IN request, all the application modules corresponding to that PACKET_IN request are executed to generate a reply PACKET_OUT message. Then the controller feeds flow rules corresponding to that flow using PACKET_OUT message in the OpenFlow compatible switch. Controllers by default run in reactive mode, and all testing done by us is with controllers running in reactive mode. There are many open source SDN controller implementations available for use such as, Beacon, MUL, Maestro, floodlight, OpenDaylight. We have tested for few of these in my work.

So, focusing on the controller in Figure 1 we can observe that at the controller we get PACKET_IN requests from different openflow compatible switches corresponding to different flows and controller sends PACKET_OUT reply message to the OpenFlow compatible switch for each PACKET_IN request message. So, to test controller we can send some PACKET_IN requests per second to the controller and see how many requests it can handle.

![Diagram of SDN controller setup](image)

**Figure 3: Scenario for Controller Testing**

In our work we have tested and compared three controller, namely Beacon[6], Maestro[7], OpenMUL[8]. Beacon and Maestro are stable, modular Java based implementations of SDN controller. As, written in Java they can be run on many platforms from high end Linux servers to android phones. Beacon also provides
web based user interface to manage and monitor the controller. OpenMUL is a lightweight implementation of a SDN controller written in C. It is known for it’s low CPU resource requirements.

In our work we have used a tool cbench[3], which is commonly used for controller benchmarking. Cbench takes number of switches and unique end host MAC addresses per switch as input parameters, and emulate the PACKET_IN requests for end hosts from all switches to the controller. Figure 3 describes the manner in which cbench emulates switches and end hosts per switch.

Cbench has two modes of operation, throughput mode and latency mode. In throughput mode for a given number of unique MAC addresses per switch, cbench continuously keep on sending a stream of PACKET_IN requests to the controller. Controller will respond to as many PACKET_IN requests it can handle. So, cbench counts no of reply messages from the controller to get the maximum number of requests a controller can handle for given number of switches and unique MAC addresses per switch. In latency mode for a given number of unique MAC addresses per switch, cbench sends a next PACKET_IN request only after reply for the previous PACKET_IN request comes from the controller. So, this way we will get the number of packets a controller can serve if sent serially. So, taking inverse of this count we will get the average time required by the controller to serve a request.

3.0.1. Experiment Scenario and Setup

All the controllers we used for testing are open source. So, to run the controllers you need to get source from their official website and have to follow the instructions given to compile, build and configure the source code for each controller. Some if the controllers give few basic controller application implementations with them. We have tested all controllers by the layer 2 switch functionality application on each switch.

To test all controllers we have used cbench. Controllers and cbench run on same machine. They communicate over a loop-back address. So, to test throughput and latency of each controller we have written some scripts which will run controller, and then test that controller using cbench for different number of switches and unique MAC addresses per switch. Script also monitors resource utilisations.

3.0.2. Results & Analysis

While performing experiments we tested three controller Beacon, Maestro and OpenMUL, by varying the number of switches and the number of unique MAC addresses per switch in the emulated setup created by cbench. We also performed the experiments by varying the number of CPU cores given to the controller. After performing all these experiments we got throughput and latency graph of all controllers for different number of CPU cores given to the controller. So to compare all these controller performances we have plotted a few graphs for throughput and latency
varying either number of switches or number of unique MAC addresses per switch and keeping other parameter constant.

In the graph 4a, to compare throughput of different controllers we have kept number of switches constant as 32 and varied number of unique MAC addresses per switch to see how throughput get affected. So we can see from graph that varying number of MAC addresses per switch does not affect throughput much, but at the same time we can clearly observe that throughput of Maestro controller is very less compared to that of Beacon or OpenMUL. In the graph 4b we have compared latency by keeping number of switches constant and varying number of unique MAC addresses per switch. Form graph we can clearly see that latency of Beacon controller is least and also that the latencies of all the controllers are not varying much with change in number of unique MAC addresses per switch.

![Graph 4a: Throughput Graph](image)

![Graph 4b: Latency](image)

Figure 4: Throughput and Latency Graphs with Number of Switches Fixed to 32

Then to observe the effect of varying number of switches on both throughput and latency we plotted the graphs by keeping number of unique MAC addresses per switch constant as 100000 and varies the number of switches. So looking at the graph 5a we can see that for small number of switches throughput increases with increase in number of switches, but as the number of switches increase beyond 16 the throughput doesn’t changes much. Also from the graph 5b we can observe that

![Graph 5a: Throughput](image)

![Graph 5b: Latency](image)

Figure 5: Throughput and Latency Graphs with Number of Unique MAC Address Fixed to 100000
for smaller number of switches the latency is higher and it decreases with increase in number of switches. But then for the number of switches beyond 16 the latency for all controllers remain approximately same. We observed that for all controllers, by increasing the number CPU cores for the controller, we get relatively higher throughput compared to the throughput for the same controller running on less number of CPU cores. We also observed that ac. After observing all throughput and latency graphs we found that Beacon controller is performing better than the other two controllers.

4. Switch Performance Testing

OpenvSwitch is widely used OpenFlow compatible virtual switch. We have already given brief introduction about the OpenvSwitch components and it’s basic working. So, if we look at OpenvSwitch, at higher level it has two functionalities. One and major function of the OpenvSwitch is a switching functionality, where it have to forward incoming packets to appropriate ports after modifying the packet according to the action part in the flow entry for the flow to which that packet belongs. The second functionality is to generate the PACKET_IN request to the controller if there is no matching flow entry for an incoming packets in the flow tables at the switch. So to test both functionalities for OpenvSwitch We have run some experiments described below.

4.1. OpenvSwitch without OpenFlow Rules

When OpenvSwitch is created, if you do not connect it to the controller it has only one rule in it’s flow tables matching all packets and the action part as ‘normal’. So, if you do not connect a OpenvSwitch to a controller, by default acts as a layer 2 switch in the normal network. So, in this part we have performed experiments to find out the maximum data traffic an OpenvSwitch can handle when controller is not connected to it and it is acting as layer 2 switch in normal network.

4.1.1. Experiment using Virtual Machines

Scenario and Setup
To test OpenvSwitch for getting maximum data traffic it can handle we connected OpenvSwitch to two virtual machines. Virtual machines are created using KVM hypervisor and they are connected to OpenvSwitch using tap interface. Figure 10 describes experiment scenario.

Now, to generate the traffic between these two virtual machines we used Iperf. Iperf is well known network bandwidth measurement tool, which send specified
bandwidth of data from iperf client to iperf server to measure effective bandwidth. We used iperf to generate data traffic. So we have written a script using iperf to gradually increase the input bandwidth and measure how much of that input traffic is handled by OpenvSwitch.

**Results & Analysis**

After collecting the values of throughput, latency and resource utilisations for different input bandwidth we have plotted graphs. So, if we observe throughput graph 7a we can see that throughput is getting saturated at around 1600Mbps. But if we look at the CPU utilisation graph 7d,7e of OpenvSwitch, we are not able to overload OpenvSwitch and the CPU utilisation of iperf client and iperf server is also not high. We also observe that memory and disk utilisation of the all the processes is low and not much changing. If we observe in graph 7c, network data written at client side, we can see that it is approximately same as the throughput. Network data read at server side is also approximately same as the network data written at client side. So we can say that whatever data being written by client side virtual machine is forwarded to server side virtual machine. So, the virtual machines are not able to send large amount of data to the OpenvSwitch to get OpenvSwitch bottlenecked. So, to resolve this problem we made use of network namespaces as described in following experiment.
Figure 7: Through Latency and System Performance Graphs for Experiment using Virtual Machines

4.1.2. Experiment using Network Namespaces

Scenario and Setup
As from the above experiment results we got that a virtual machine was not able to write more data than 1600Mbps. So, we haven’t found out the maximum data traffic handled by the OpenvSwitch. So, to eliminate the limitation of virtual machine not able to write out enough data we made use of network namespaces to send data traffic through the OpenvSwitch. Network namespace[4] is another copy of network stack with its own routes, firewall rules and network devices. So, with network namespaces we can have different and separate instances of network interfaces and routing tables that operate independent of each other. We can use virtual ethernet interface or OpenvSwitch ports to connect network namespaces to OpenvSwitch.

So, to perform this experiments we created two network namespaces, connected those network namespaces to the OpenvSwitch using OpenvSwitch internal ports and then configured them by assigning IP addresses to the interfaces inside network namespaces, as Figure 8 describes. Now with this setup, using iperf we sent different data traffic through the OpenvSwitch and monitored how much of that data traffic can be handled by OpenvSwitch, along with monitoring system resources.
Results & Analysis

Throughput, latency and resource utilisation graphs for the above experiment are shown below. If we look at the throughput graph 9a, we can see that the maximum amount of data that is handled by the OpenvSwitch is about 27000Mbps. This is much larger than the throughput we got in the previous experiment. But if we observe the CPU utilisation of the OpenvSwitch in graph 9e, it is still very low. If we look at the CPU utilisation of the iperf server in graph 9d, we can see it is getting 100% and after the point CPU utilisation hits 100% we can see for corresponding input bandwidths in the throughput graph, there is no increase in the throughput. So, we can say that, OpenvSwitch is still not overloaded, but as iperf program is not able to send more amount data through OpenvSwitch, after input bandwidth of 30000Mbps we can’t see throughput graph flattened out. So from these results we can say that OpenvSwitch can handle at least 27000Mbps of data traffic.
4.2. OpenvSwitch with Openflow Rules

In this set of experiments we wanted to test how OpenvSwitch would perform in SDN network environment. So, the OpenvSwitch will be connected to the SDN controller and the SDN controller will feed rules in OpenvSwitch for packets going through the OpenvSwitch.

4.2.1. Experiment with One Flow

Scenario and Setup
To start with this part of testing we wanted to test OpenvSwitch for how much data traffic it can handle when it is connected to the SDN controller. So, we created three network namespaces and connected them to the OpenvSwitch using OpenvSwitch ports. In one of the three network namespaces we ran floodlight SDN controller and among other two one will be running iperf client program and other will be running iperf server program. OpenvSwitch is connected to the SDN controller and then different data traffic is sent through the OpenvSwitch while monitoring system resource utilisations. Figure 10 gives the complete picture of the testing scenario.
Results & Analysis
We plotted throughput, latency and resource utilisation to get the following graphs. The graphs for this experiment are quite similar to previous experiment graphs. We can see maximum throughput of around 2700Mbps and OpenvSwitch CPU utilisation is low. CPU utilisation of iperf server process in graph 11d, is getting 100%, resulting in not able to send more data traffic through the OpenvSwitch and we see throughput in graph 11a, flattening out. So can conclude that, having OpenvSwitch connected to the controller doesn’t affect the amount of data handled by OpenvSwitch, when number of flows passing through OpenvSwitch are less.
4.2.2. Experiment with Multiple Flows

Scenario and Setup
Now, we wanted to test how openvSwitch performs when many number of data flows are going through the OpenvSwitch. So, to test this scenario we created similar setup as the previous experiment, with three network namespaces connected to OpenvSwitch using OpenvSwitch ports. In one of the namespaces we will be running SDN controller and the OpenvSwitch will be connected to the SDN controller. Now to emulate many number of flows we made use of parallel connection streams to the iperf server. But there is limit of 128 for creating such parallel connection streams to the iperf server. To emulate more number of flows we created multiple instances of such iperf client-server pairs each sending parallel connection stream to their respective servers. So the scenario is same as that explained in Figure 10, but now instead of single instances of iperf client and server, there will multiple instances of iperf client and server running in network namespaces. We have written scripts to create more number of instances of iperf client-server pairs, maintaining input bandwidth per flow same, as we gradually increase the number of parallel flows passing through the switch.
**Results & Analysis**  Throughput, latency and system resource utilizations are plotted with different data traffic input bandwidths, for different number of flows passing through OpenvSwitch. We can see in throughput graph 12a, that as the input bandwidth to the OPenvSwitch increases there is a degradation in throughput for higher number of flows. But, at the same time if we look at the CPU utilisation of the OpenvSwitch in graph 12e, it is not much changing with increase in input bandwidth for any number of flows passing through OpenvSwitch. But looking at the CPU utilisations of the iperf clients in graph 12c and iperf servers in graph 12d, they both are getting hit to approximately 200% for higher number of flows as input bandwidth increases. As we are testing on four core machine, after around 30000Mbps of input bandwidth collectively iperf client and iperf server are consuming almost all of the CPU and we can say iperf is not able to generate more data traffic as it can’t get more CPU, throughput graphs are flattening out after around 30000Mbps of input data traffic.

![Throughput](image1.png) ![Latency](image2.png) ![Iperf client cpu util](image3.png) ![Iperf server cpu util](image4.png) ![OpenvSwitch cpu util](image5.png)

**Figure 12**: Through latency and system performance graphs for experiment with one multiple throughput OpenvSwitch

### 4.3. Flow Initiations Handled by OpenvSwitch

In this experiment we wanted to test OpenvSwitch functionality of generating PACKET_IN requests for each packet, which doesn’t have flow table entry in the flow table. To achieve this, SDN controller should be able to count the number of PACKET_IN request received from switch but it shouldn’t send reply message with
flow table entry to the switch, so that OpenvSwitch flow table will never have flow table entry corresponding to any flow and it will always generate a PACKET_IN request for each packet coming to it. So, to achieve this we wrote an application module for the floodlight controller which counts PACKET_IN requests coming in at the controller and doesn’t send any reply back to OpenvSwitch. To generate large number of flow initiation requests we have written a program ‘raw_udp’, which uses raw socket to send large number of UDP packets which will serve as flow initiation requests.

4.3.1. Experiment

Scenario and Setup
To test this scenario we created five network namespaces h1-h5. All network namespaces are connected to the OpenvSwitch using virtual ethernet interface pairs(veth pairs). Now, controller is running in network namespace h1, and raw_udp send program which sends flow initiation packets is running at nework namespace h2. Iperf client and iperf server runs in network namespace h4 and h5 respectively, which does the job of generating data traffic load. Figure 13 shows the experiment setup scenario. So, we initially sent different number of flow initiation packets to the OpenvSwitch and every time counted for how many of them OpenvSwitch generated PACKET_IN requests. But we observed that sending different number of flow initiations packets doesn’t change OpenvSwitch CPU utilisation much. Because whatever may be the number of packets sent, the send program takes large amount of CPU and every time it sends packets with same packet rate. Then we observed that if we limit the CPU usage for the send program, the rate at which it sends packets changes. So to change the packet rate we decided to change the CPU allocation of the send program, so that the rate of sending packets will change. To automate this experiment we wrote a script which will take reading for different load data traffic and limiting the send program CPU utilisation to different percentages. But to make different components of script, which run in every network namespace, communicate and work properly we have to add some flow rules to the OpenvSwitch so that messages between these script components pass OpenvSwitch. But we have make sure that for all flow initiation requests sent to OpenvSwitch there will be no matching flow table entry at OpenvSwitch and OpenvSwitch will have to generate flow initiation request for all such packets sent by ‘raw_udp’ program form network namespace h2.

Results & Analysis
So looking at the packet rate graph 14a, we can clearly observe that as we give more CPU to send program, the rate at which packets are being sent is increasing. Also looking at the OpenvSwitch CPU utilisation graph 14b, as the CPU limit for the send program increases there is increase in CPU utilisation of the OpenvSwitch. So, looking at both the packet rate graph and CPU utilisation of OpenvSwitch Graph,
we can say that as the rate of sending flow initiation packets to the OpenvSwitch increases, there is increase in the CPU utilisation of OpenvSwitch. Though we can see the increase in CPU utilisation of OpenvSwitch, it is not even utilising one core on CPU. We can also observe that, as CPU is not bottlenecked, the number of PACKET\_IN requests generated by the OpenvSwitch remains approximately constant. So, we can expect to see number of PACKET\_IN requests getting generated reduced, as the OpenvSwitch get bottleneck.

```
(a) Packet Rate
(b) OpenvSwitch cpu util
(c) PACKET\_IN request Count at Controller
```

Figure 14: Graphs for Experiment to Test Flow Initiations Handled by OpenvSwitch
5. Conclusion & Future Work

The number of maximum PACKET_IN requests handles by each controller we tested are pretty high with controller beacon satisfying around 4.5 million requests/sec, controller OpenMUL satisfying around 4.5 million requests/sec and controller maestro satisfying around 0.2 million requests/sec when run on three CPU cores. So we can conclude that performance of Maestro controller is poor compared to both Beacon and OpenMUL. In future we will also add some more controller in this comparison and also try to compare how different controller applications affect the number of PACKET_IN requests satisfied by different controllers.

In many of the experiments while testing Openvswitch, we have found that the load generation programs were consuming large amount of CPU and whatever load traffic were generated by programs like iperf were easily handled by the OpenvSwitch. So, we are not able to get OpenvSwitch CPU utilisation to 100% as not enough load traffic is generated. So, we will perform these experiments on machine with mush larger CPU cores and try to get OpenvSwitch CPU utilisation to 100%, so that we can get how much traffic can be handled by OpenvSwitch per core.
References


[2] OpenvSwitch Support
http://openvswitch.org/support/

[3] Controller Benchmarking Tool
http://archive.openflow.org/wk/index.php/Oflops


[5] iPerf - The network bandwidth measurement tool
https://iperf.fr/

[6] Beacon Controller
https://openflow.stanford.edu/display/Beacon/Home

https://code.google.com/p/maestro-platform/

[8] OpenMUL Controller
http://www.openmul.org/openmul-controller.html

[9] Application Module for Floodlight
https://floodlight.atlassian.net/wiki/display/floodlightcontroller/How+to+Write+a+Module