Distributed Web-Cache using OpenFlow

Project group Number: 8

CSC/ECE 573: Internet Protocols

Fall 2016

Guide

Dr. Muhammad Shahzad

Team

Arjun Thimmareddy [athimma@ncsu.edu]
Pranesha Shashwath Kumar Kattepura Jayabheema Rao [pkattep@ncsu.edu]
Aasheesh Tandon [atandon@ncsu.edu]
Jay Udani [judani@ncsu.edu]

2nd December 2016

1. INTRODUCTION

In networks where a lot of similar requests are made by multiple clients, the network often gets congested, which results in longer delays and reduced efficiency. So, to take care of such scenarios, 'web caching' can be employed to temporarily store the web content in a local server. This helps in reducing the load on the web server and the network bandwidth.

In a Content Delivery Networks (CDN) or a data centers, sharding and data sharing among the servers is very prevalent. Having a single cache, sometimes could be inefficient as every server would dependent on that single cache. This might increase the latency for the servers located at a longer distance from the cache. Further, having a single cache becomes a single point of failure in cases the server becomes a target of an attack, say DDoS attack. This makes the overall system less robust and an easy target for an attacker to take it down. Thence, we are proposing a system where the cached data is distributed among multiple web caches.

Next, we have implemented this system using OpenFlow. It is a protocol in which switches get their forwarding rules from a server. The packet forwarding decisions are made centralized in order to maintain network programmability independent of the networking device (switch/router). So, it is different from the traditional switches in the sense that the data plane and control plane are not on the same device. The data plane is handled by a controller (server) which makes the packet forwarding decisions on the switch. The communication between the controller and switch is based on the OpenFlow Protocol. This method allows for a greater control and in turn more effective use of network resources as compared to the traditional networking systems. It also allows us to force network traffic to take specific paths for security purposes or for load balancing. OpenFlow processes packets on each hop to determine the most optimal route at each stop. So, the overall functionality is like traditional packet forwarding systems, only with the plus of having a software controlled data plane. Such untraditional networking falls under the category of 'Software Defined Networking' and it has gained a lot of momentum in applications such next-gen IP mobile networks, mission critical networks and virtual machine mobility. [1]

The objective of this project is to implement a distributed web cache using OpenFlow and OpenvSwitches on GENI testbed (Global Environment for Network Innovations) so as to enable faster access to frequently accessed content by storing them in local caches. This would reduce the load on the access link by caching the previous responses for GET requests from the internet. Finally, achieving this in a distributed way in order to load balance caching.

2. DESIGN

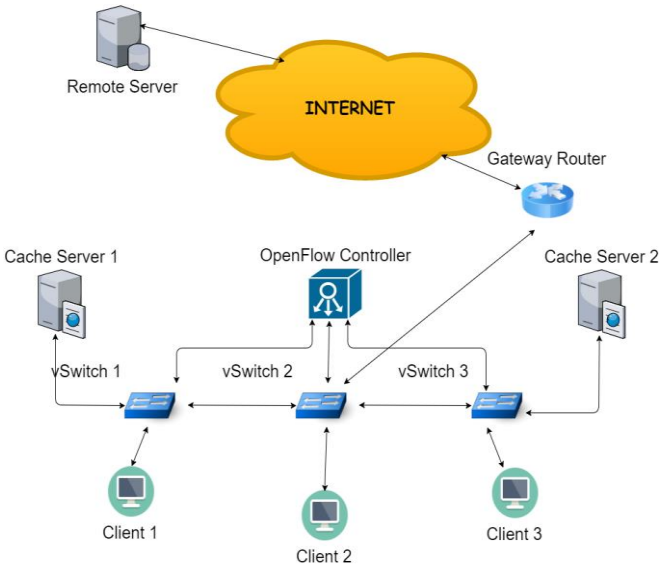

Figure 1: System Model

As can be seen in the block diagram (see figure 1), the implemented topology consists of three switches, a single controller and two local web-cache servers. The OpenFlow controller feeds the flow table entries into switches which defines the flow between clients and caches. Cached content is distributed among the two local caches. Assignment of web content to the local cache is achieved by round-robin algorithm as and when a client puts up a request.

Upon a HTTP WGET/curl request for a URL by a client, the switch checks whether a mapping already exists for that particular host in its flow table. If it does, then the request is directed to the web cache handling requests for the host. If the flow for the web cache does not exist in the switch, then the switch obtains the mapping from the controller. If the mapping does not exist in the controller, then controller chooses a web cache using the round robin algorithm. Once a cache is

selected by the controller, an appropriate flow entry to forward the client's request to the selected cache is made into the switch.

The response file from the cache is directly streamed to the client if it already exists in the cache, else it is first fetched from the internet and then streamed to the client. All the files inside the cache have a TTL element associated with them, which when expires, makes the flow entry invalid and then the switch again needs forwarding information from the controller.

Upon a cache miss, the file is obtained from the internet, then first cached in the web cache server and then streamed back to the client. A TCP connection is established between the controller and the switches. Again, TCP segments are used for fetching the file from the web-cache server to the client requesting it. TCP packets assure reliable end-to-end delivery, flow control, error control as well as congestion control. Implementation is done on GENI test bed using POX OpenFlow controller. GENI is a platform which provides a virtual laboratory for experimenting on networking and distributed systems. Using this, we obtained virtual machines which allowed us to install Openvswitch (OVS) and OpenFlow on them, thereby helping us implement our network design.

The detail algorithm can be seen using the flowcharts below:

Flowchart for client:

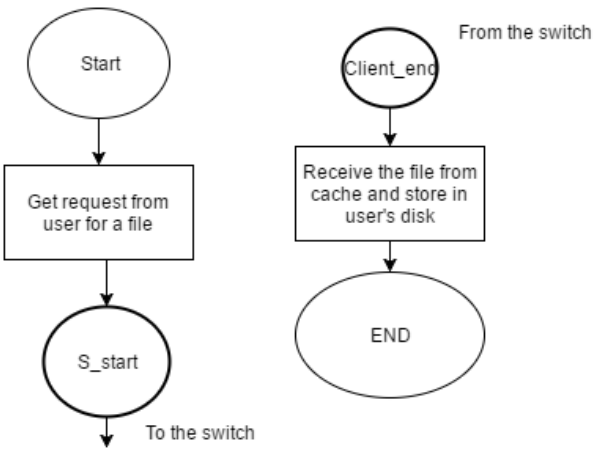

Figure 2: Client Flowchart

As can be seen in the flowchart (see figure 2), the client sends a request for a specific file by specifying the complete URL for the file. We use 'curl' command for the request. Along with the command we also mention the Ethernet interface via which the request should go, making sure that it does not go via 'eth0' which is the management interface (used for managing the VMs). We use 'eth1' interface so that the request goes via our network topology on the GENI VMs.
On successful retrieval of the file, it stores the file to the current working directory.

Flowchart for switch:

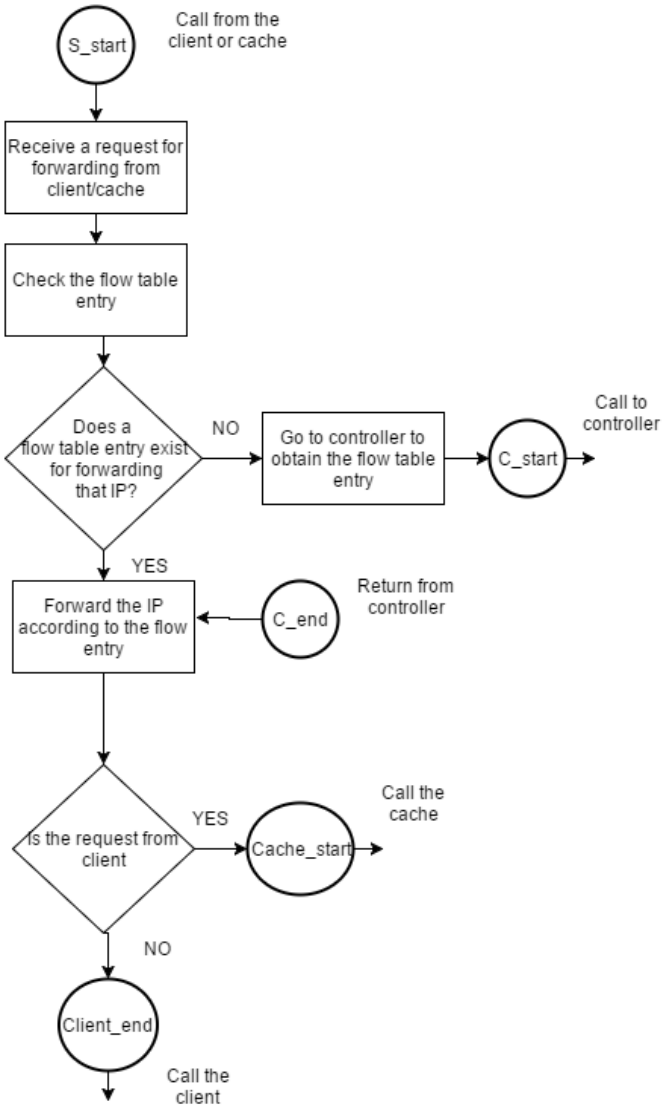

Figure 3: Switch Flowchart

As can be seen in the flowchart (see figure 3), on receiving the packet, the switch checks for the destination IP. If a flow table entry already exists for that particular IP, the switch simply forwards the packet on that specific link to the cache. If the mapping does not exist, it calls the controller to obtain the flow

table entry for that IP address. On getting back the response from the cache, the switch simply sends back the file to the client according to the flow table entry which already exists in the OVS.

Flowchart for Controller:

On receiving the request for forwarding from switch, the POX controller first checks whether a mapping exists. If yes, then simply install a flow table entry for the same in the OVS. If the mapping doesn't exist, then create a mapping for the same using a round-robin algorithm and then install a flow table entry for the same in the OVS(see figure 5).

Flowchart for Cache:

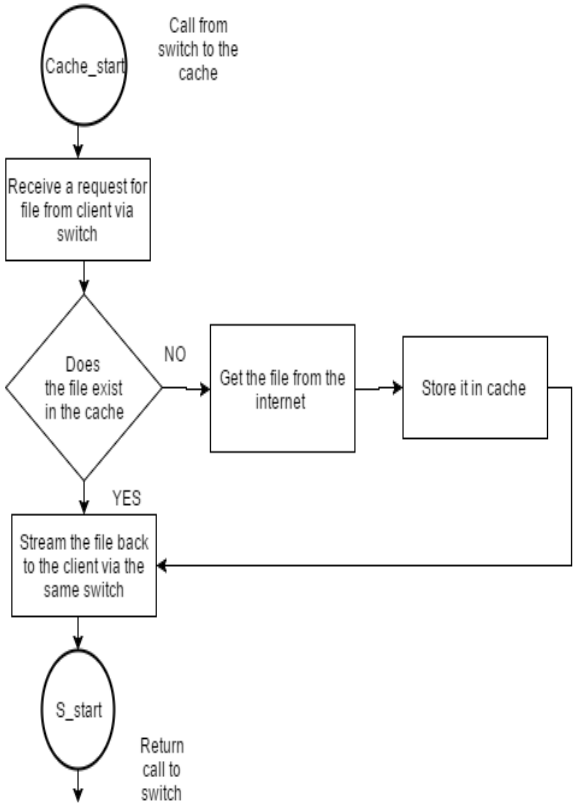

Figure 4: Cache Flowchart

On receiving a request for a file from client via the switch, the cache checks whether it has that file or not. If it does, then it simply streams the file back to the client using a TCP connection. If it does not have the file, then it simply gets the file from the internet using the curl command, stores it, and then streams the file back to the client by opening a TCP socket (see figure 4).

3. IMPLEMENTATION

We use Python as our programming language since the POX is a Python based SDN controller. We create a GENI slice using ORCA-Flukes GUI.

The User enters the followings parameters for requesting the web-content:

The user must use the curl command to get make a HTTP GET request for the specified URL. The Ethernet interface (eth1 in our case) has to be mentioned, because on the GENI VMs there is a management interface which takes the requests by default if the user doesn't mentions the interface. And in that case our designed topology is not used for servicing the user's request as the management interface has a default gateway setup which connects it to the world-wide web and the request gets served directly without going through the controller and the OVS. Further, there is no need to not mention the destination port number since by default the destination port is PORT 80.
We do not use the alternate command "wget" to make a HTTP GET request because "wget" does not allow us to mention the Ethernet interface along which the request should go.
We are using ExoGENI as our GENI testbed. "ExoGENI is based on an Infrastructure-as-a-Service (IaaS) cloud model with coordinated provisioning across multiple sites and a high degree of control over intra- and inter-site networking functions. It provides unified access to services to enable users to construct virtual network topologies on demand. The testbed software supports GENI APIs and extended APIs to enable users to create and manage a virtual network as a slice of virtualized resources within the infrastructure." [3]

Basic operations embedded in ExoGENI include:

1. It provides computing resources from the available rack resources. It enables us to supply boot images for virtualized instances.

2. It also provides functionality for both Windows and Mac apart from Linux.

3. It enables us to create, modify and destroy slices which occupy computing resources generated by rack switches and intermediate circuit providers.

4. It enables us to use the OpenFlow protocol in slices to forward the packets. But these slices are restricted to VLANs provisioned between the racks.

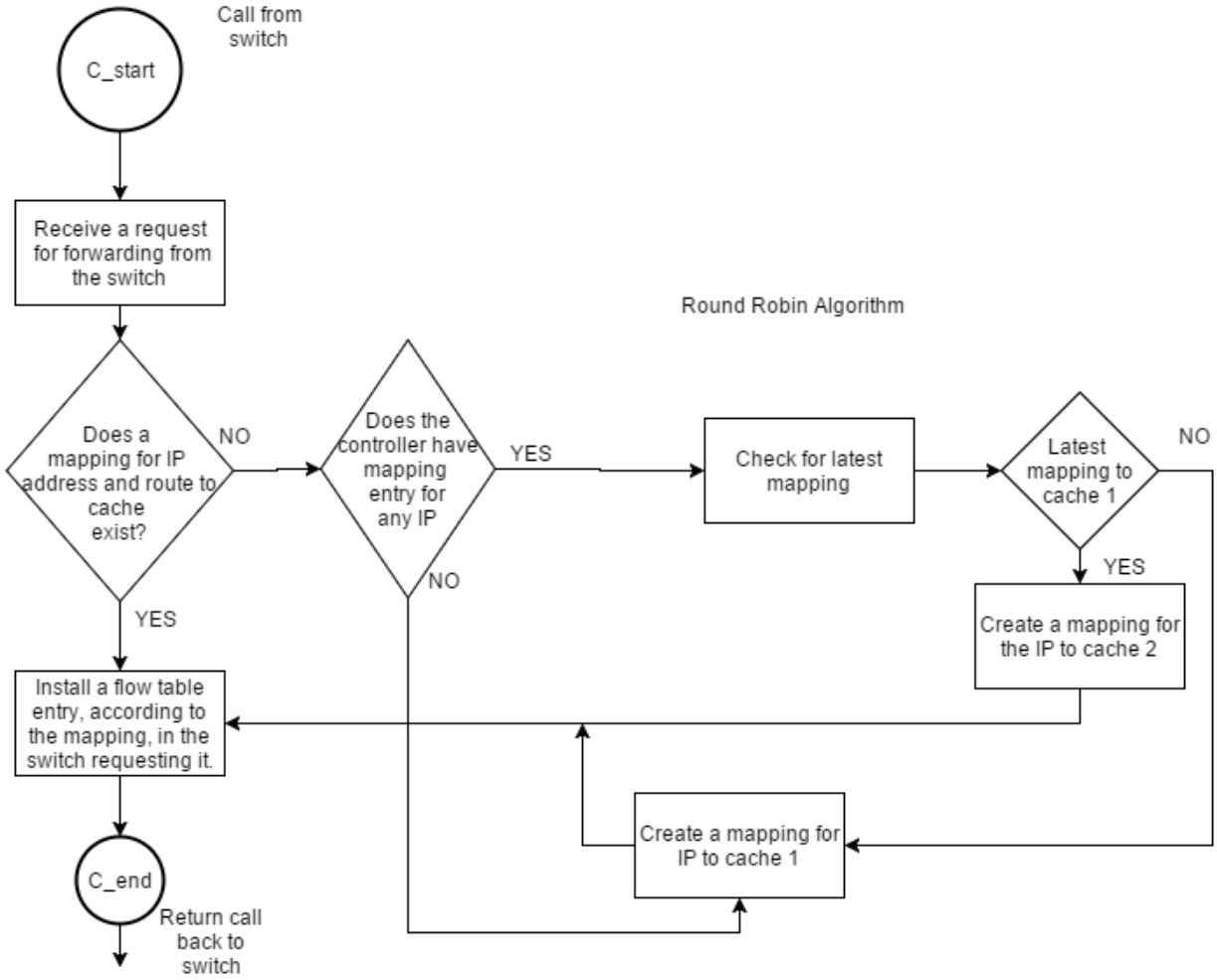

Figure 5: Controller Flowchart

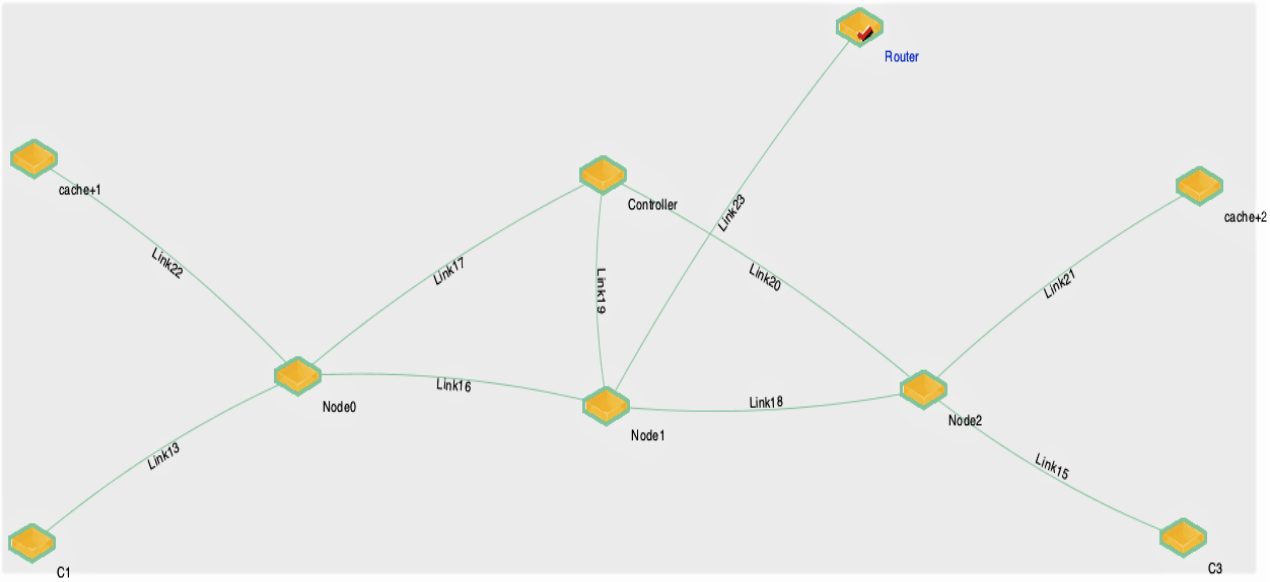

Figure 6: Topology Implemented on ExoGENI testbed

5. In the future, using the OpenFlow protocol, it will provide additional features which allow external traffic to transit existing slices.

6. It enables us to combine ExoGENI resources with other GENI resources such as WiMAX testbeds and meso-scale OpenFlow.

Configuring topology on GENI Slice:

Each topology in GENI is made on a slice. This slice is a set of VMs which can be assigned different functions by installing software as per our needs. These VMs must be reserved from one of available locations across the US which offer remote connections to these VMs. We have made the topology on a slice as shown in the figure above (see figure 6). We configured the VMs as switches and installed OpenFlow on them by the following:

1. *Open vSwitch Installation*: By logging into the VMs we Install Debian Open vSwitch packages (switch, common and controller).
2. *Configuring Open vSwitch:* Un-assign IP addresses on the interfaces that connect to clients/servers/switches.
3. *Create a new bridge* on the Open vSwitch node and add the host interfaces to the bridge.
4. *Set the Open vSwitch node to OF mode* to secure. This will make the topology secure in a way that the swtiches wont forward packets if the controller is down. For making the topology work even when the controller fails, we need to set the OF mode to 'standalone'.
5. *Add a controller to the Open vSwitch*. In this step, we specify the controller of the Open vSwitches by specifying its IP address and the port number 6633.

Source Code Descriptions:

A. Cache source code:

Our caches are configured to run on port 80. On receiving the request for a file from client, it reads the request, parses it and in turn sends a request to get it from the internet if the file does not exist in the cache. We service every new request directly. We also maintain entries in a unique file for all the URL's requested and the contents of that URL.

Our caches are configured to be multithreaded in nature meaning multiple TCP connections can be opened simultaneously. Thus, if a HTML doc indexes multiple objects, all of them can requested from the internet using parallel TCP connections.

We also associate a TTL element that acts a timestamp to check for the validity of the file. The TTL element is set appropriately from the 'Last-modified' timestamp in the HTTP response message. If the file is present but the timestamp has expired, then the cache simply gets the file back from the internet.

We also maintain the state of the cache in a unique JSON file. This provides a backup in case our cache server goes down. The cached content in the server can

be restored back by reading the JSON file.

B. Controller Source code:

POX is a python based SDN controller which manages flow control to enable intelligent inter-networking. It uses OpenFlow protocol to tell the switches where to redirect the packet.

We are imitating the behaviour of L3 learning switch

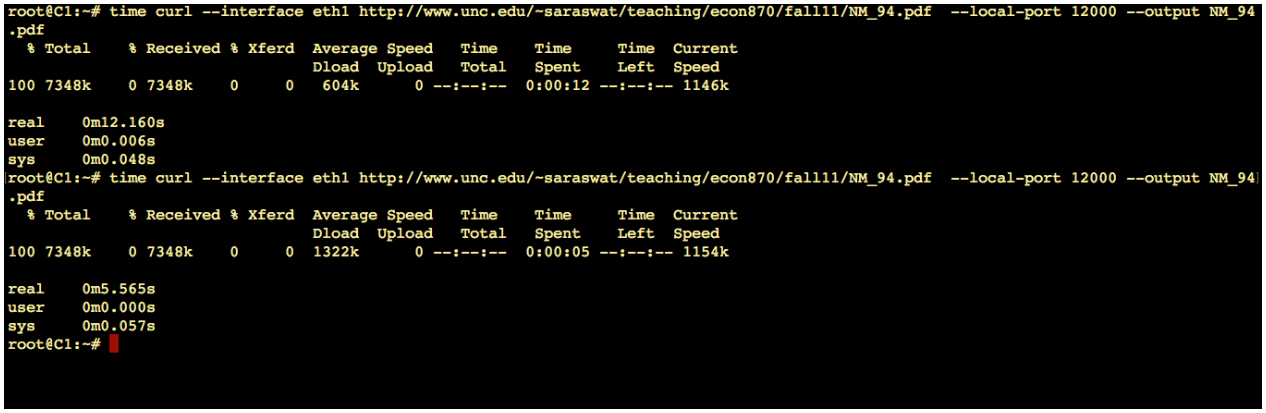

Figure 7: Curl request for URL by user. See time difference achieved in the second request.

using a module. The module has many functionalities such as:

1. It learns the mapping between IP and MAC addresses.
2. It uses this information to install a rule which replaces a destination MAC while forwarding a packet to the correct port.
3. It floods ARP request in case it doesn't know the destination
4. It also replies to the ARP request
5. It updates the ARP table of a switch with a particular DPID using 'PacketIn' handler.

4. RESULTS AND DISCUSSION

As can be seen from above see figure 7, when the requested file is not present in the cache, it takes quite some time to send the file back to the client. This is because the cache first requests the file from the internet, stores a copy of the file in itself and then streams the file back to the client.

However, when the same file is requested again, it takes lesser amount of time comparatively to send the file back to the client since the cache already has the file.

This clearly shows our functionality of distributed WEB-CACHING and the improvement in performance.

Figure 8 shows flow table entries which the controller provides to the OVS for data forwarding.

We faced the following challenges during our project:

1. Maintaining the state of the cache: In case the cache goes down, it was important to restore the contents and relevant information in a file before the process went down. This was important to maintain the state of cache and relevant information one run to next.

We resolved this issue by creating a JSON file which maintains this information.

2. Implementation of network topology on GENI testbed: Reserving the resources, creating slices, maintaining the state of the topology, assigning the IP addresses, understanding the subnet at which the GENI works were all challenging. These had a huge learning curve as we created and re - created over 10 topologies before we got a completely working topology we wanted.

3. Understanding and altering POX controller: Being new to it and also having to handle a foreign code extensively written. This controller was critical to configure as an L3_learning module that automatically maps the IP and MAC, install the flows in the switch, floods the ARP, respond to the ARP, re-route requests to external IPs to one of the caches in a distributed fashion. Moreover, make everything work in an Openflow framework was a challenge on its own. Reading the code many times and understanding what controls lied where was a huge challenge.

4. Working with 4 layers at the same time: To resolve some of the core issues, we debugged our way through all the 4 layers of TCP/IP model. We had to sniff the

packets using tools such as tshark, wireshark and tcpdump to check the MAC addresses correctness, IP addresses verification, port number compatibility, checksum verification, fragmentation verification and

Figure 8: Log on OVS showing the flow table entries

subnet configurations.

5. TCP connection with external IP: This was the biggest challenge we faced in this project. This challenge consumed more than 40% of our work effort towards this project. We dealt with all the layers and parameters mentioned in the previous point simultaneously. We eliminated the possibilities of issue one by one and independently understood that it was the issue with the subnet configuration which did not allow interactions with the external IP. So, we mapped our external IPs with one of the host addresses available in the subnet but not yet assigned to any of the components. Then we wrapped the requests for external IP inside a request for one of the subnet IPs we had mapped and sent them to cache. This did the trick

6. Flow Installation: Working with ARP wasn't a breeze. Understanding how the ARP worked and how it influenced each of the switch ports was a challenge. The switch we were working with GENI dealt with more than one layer and every action we were taking for the distributed cache dependent directly on the flows in the switch. It was critical to get this working perfectly.

7. Cache down: Handling the case of cache going down was a major challenge. We are re-routing the requests from one cache to another in case the cache is down. Ideating how to implement this was a challenge. We are disabling the ARP flow to simulate this

Thus, we were successfully able to implement our proposed project: Distributed Web-Caching using OpenFlow on GENI test-bed.

Thus, we have enabled faster access to frequently accessed content by storing them in local caches. This also in turns reduces the load on the access link by caching the previous responses for GET requests from the internet.

Finally, we also achieve load balancing by this in a distributed way.

5. REFERENCES

1. OpenFlow.http://whatis.techtarget.com/definition/OpenFlow
2. R. Tewari, M. Dahlin, H. M. Vin, and J. S. Kay, "*Beyond Hierarchies: Design Considerations for Distributed Caching on the Internet*," in ICDCS, 1998.
3. Wikipedia - Web Cache. http://en.wikipedia.org/wiki/Web_cache.
4. GENI. http://www.exogeni.net/
5. GENI: Exploring Networks of Future. http://www.geni.net/
6. OpenFlow Controllers in GENI. http://groups.geni.net/geni/wiki/OpenFlow/Controllers#OpenFlowControllersinGENI
7. OpenFlow. https://www.opennetworking.org/sdn-resources/openflow
8. Open vSwitch. https://github.com/openvswitch/ovs
9. POX. https://github.com/noxrepo/pox
10. POX WIKI. *openflow.stanford.edu/display/ONL/POX+Wiki*
11. POX:http://searchsdn.techtarget.com/definition/SDN-controller-software-defined-networking-controller

Few extra logs of Client and Switches

```
root@C1:~# time curl --interface eth1 http://www.iso.org/iso/annual_report_2009.pdf  --local-port 12000 --output report.pdf
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 10.2M    0 10.2M    0     0   336k      0 --:--:--  0:00:31 --:--:-- 1154k

real    0m31.241s
user    0m0.013s
sys     0m0.064s
root@C1:~# time curl --interface eth1 http://www.iso.org/iso/annual_report_2009.pdf  --local-port 12000 --output report.pdf
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 10.2M    0 10.2M    0     0  1265k      0 --:--:--  0:00:08 --:--:-- 1132k

real    0m8.312s
user    0m0.008s
sys     0m0.068s
root@C1:~#
```

Figure 9: Performance Improvement using Caching 1

```
root@C1:~# time curl --interface eth1 http://imaging.nikon.com:80/lineup/lens/zoom/normalzoom/af-s_dx_18-140mmf_35-56g_ed_vr/img/sample
/img_01.jpg  --local-port 12000 --output img_01.jpg
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 50654    0 50654    0     0  22620      0 --:--:--  0:00:02 --:--:-- 22623

real    0m2.246s
user    0m0.004s
sys     0m0.004s
root@C1:~# time curl --interface eth1 http://imaging.nikon.com:80/lineup/lens/zoom/normalzoom/af-s_dx_18-140mmf_35-56g_ed_vr/img/sample
/img_01.jpg  --local-port 12000 --output img_01.jpg
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 50654    0 50654    0     0  3590k      0 --:--:-- --:--:-- --:--:-- 3805k

real    0m0.020s
user    0m0.007s
sys     0m0.000s
root@C1:~#
```

Figure 10: Performance Improvement using Caching 1

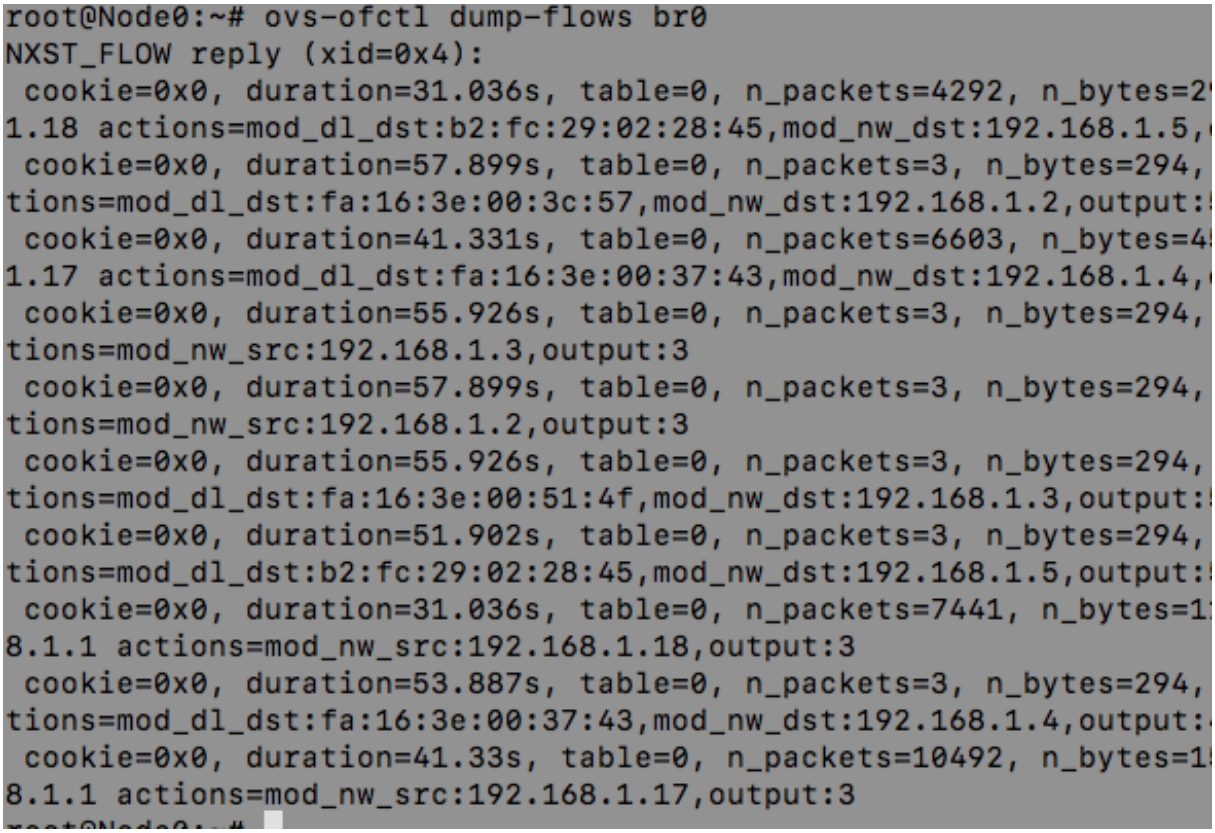

Figure 11: Flow entries on Open vSwitch