International Journal of Multimedia and Ubiquitous Engineering
Vol.11, No.2 (2016), pp. 159-172
http://dx.doi.org/10.14257/ijmue.2016.11.2.17

Implementing Innovative Routing Using Software Defined Networking (SDN)

Adnan Shahid, Jinan Fiaidhi and Sabah Mohammed

Department of Computer Science, Lakehead University,
Thunder Bay, Ontario P7B 5E1, Canada
{ashahid, jfiaidhi, Sabah.mohammed}@lakeheadu.ca

Abstract

Software Defined Networking (SDN) is an open source networking framework recently introduced. It allows developers to program and reprogram the network so that intelligence and new features can be integrated to optimize and enhance the performance of the network. This paper is focused on optimizing the routing implementation of SDN (i.e. SDN Controller). We have used the Floodlight Open Source SDN Controller[1] in our experimentation. The Floodlight controller provide source Java libraries and APIs.. It uses Dijkstra's algorithm to calculate the shortest path between any source and any destination within the network. However, the default routing implementation of Floodlight Controller is such that, while calculating any path, it ignores the actual bandwidth of the link as it takes a unit value for each link. The resultant calculated path becomes a least hop path. This least hop path may be an optimal path where all the links in the network have equal bandwidth and may not be optimal where the networks have unequal link bandwidth. However, today's networks are mostly consisting of unequal link bandwidth. The goal of this paper is to re-structure the Floodlight Controller so that it can collect the actual bandwidth of all the links in the network and use this information to calculate optimal path which is the highest bandwidth path instead of the default least hop path.

Keywords*: SDN, Floodlight, Routing, Network Efficiency, Optimal Path*

1. Introduction

As said, Software Defined Network (SDN) architecture decouples the control plane from the underlying network infrastructure as shown in Figure 1.1. This is unlike traditional network infrastructure[2] where both control and forwarding functionality are tightly integrated within all network elements. The control goes to a separate entity formally known as Controller. This decoupling allows new program to be added or existing program to be customized in control program and hence allow innovation to the network. The main focus areas of SDN architecture are, 1) Achieving centralized control through the use of controller, 2) Open interface (OpenFlow [2]) between network devices (data plane) and controller (control plane) and 3) Bring innovation and intelligence through program the network (API's) [6].

[1] http://www.projectfloodlight.org/floodlight/
[2] https://www.opennetworking.org/sdn-resources/openflow

ISSN: 1975-0080 IJMUE

(a): Traditional Network (b): SDN

Figure 1.1. Control Plane & Data Plane in Traditional Network vs SDN

SDN architecture has been adopted by many prominent vendors like Cisco, HP, Big Switch Networks, *etc.* It is been implemented in few enterprises as well as in some university networks. There have been many open source projects ongoing on different components of SDN framework and researches have mostly flourished in last 5 to 6 years.

In the traditional network, all network nodes (such as, Switches in this network) used to take part is control decision. This adds a certain amount of delay both for packet controlling and forwarding functions, while the packets travel through the network device. However over the period of time, delay in network node packet processing has reduced considerably. On the other hand, in SDN framework, network nodes do not even participate in packet control functions other than forwarding. In addition to this, in today's network data transfer speed is more important. For these reasons, calculating shortest path with least hop is less demanded and required compare to the shortest path with highest bandwidth.

2. Recent Works on SDN

There have been many researches going-on on SDN. Many researchers have proposed architectures, frameworks, API's, *etc.* based on SDN targeting various network aspects such as virtualization, scalability, network management, programmability, service assurance, optical networks, transport layer, *etc.*

One of the major advantages of SDN over traditional network is the management of the network [3]. This includes changing multiple network elements in traditional network vs centralized change management in SDN, event-driven network operation management using high-level functional language (Procera), *etc.*

Several researches going on to make an abstraction layer so that the operators can extract the low level data from underlying network elements as needed without dealing on how the low level data are generated and managed. One such project is the development of Frenetic language [5].

There have been several concerns regarding SDN, some of which are scalability and resiliency. SDN controller (based on NOX controller) has max 30k flow initiation limit with 10ms/per flow installation capacity. Researchers have argued and conclude that this limitations are not platform specific rather limitation of both SDN and traditional network [7]. However, implementing network virtualization with SDN has ruled out the scalability concern. One such research is FlowN over SDN [2].

Meridian is a prototype framework built on top of SDN to bring flexibility and agility to cloud infrastructure [4] in terms of managing virtual network in cloud environment. Many cloud providers allow the users/tenants to configure their own network within cloud infrastructure. But most of these network configurations are network or device centric such as switches, VLANs, subnets, ACLs, *etc.* Additionally, users/tenants are exposed to low level network configurations. Meridian is a framework based on SDN and it integrates along with cloud infrastructure (OpenStack with Quantum plug-in) where it built network applications, API and network orchestration layer. This way tenants only work with API & applications & these convert high level user information to low level network configuration. This way any application provisioning becomes seamless and easily manageable. However several key areas are still needs to be

considered and developed as mentioned by authors such as effective network configuration conflict resolution between multiple tenants, network updates mechanism, topology discovery, recovery from failure, *etc.*

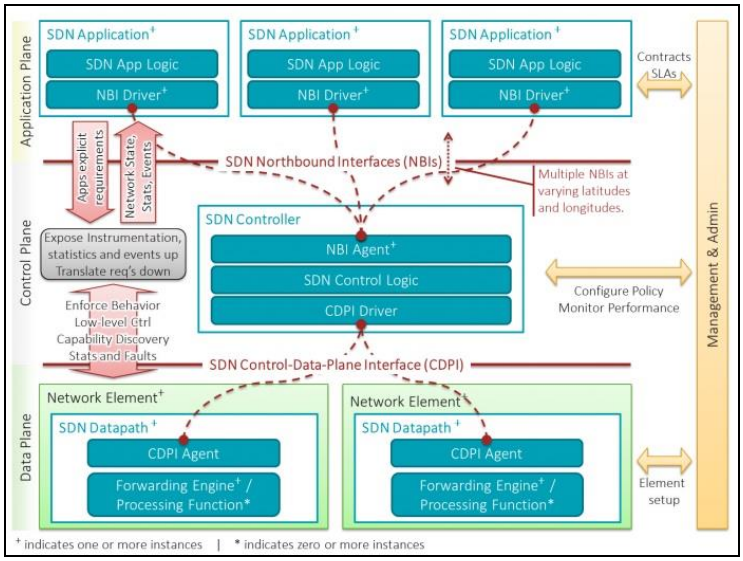

Figure 1.2. Architecture of Software Defined Networking (SDN)

It may be some times but not often possible to establish network with SDN approach. Most organization has to adopt SDN in incremental basis. Operation and performance of the overall network is always been a concern during this incremental integration of SDN with traditional network. This [8] is one of the early researches not on incremental adaptation of SDN but on the performance of SDN network segment compare to traditional network segment during this incremental deployment of SDN on traditional network.

3. Formatting Your Paper

Open Networking Foundation (ONF), which is a non-profit user driven organization, is responsible for standardizing, promoting and adopting Software Defined Networking (SDN) through open standard development, while the major focus is the wide spread commercial adaptation of SDN. Figure 1.2 shows the architecture of SDN, which has the components 1) SDN Application (SDN App), 2) SDN Controller, 3) SDN Data-path and 4) Management and Administration.

4. Benefits

Traditional networks have many limitations, such as restrictive configuration (vendor specified configuration set), complexity in managing network elements, static nature of the network, inability to scale, vendor dependency & its resistance towards change, skill workforce requirement for network management, *etc.* On the other hand SDN brings several overall benefits to today's network some of these but not all, including management simplification, increase innovation through programmability, automation, speeding-up service provisioning, increased network reliability and security, cost advantage in both Capex and Opex, increased uptime, *etc.*

5. The Required Components

In this research we have used several technologies, applications, tools, languages, *etc.* The three major components that we have used in this project were; 1) SDN OpenFlow Controller (Floodlight), 2) OpenFlow Protocol and 3) OpenFlow based Network Elements (OVS).

In addition to this, we have used some other applications and tools like, a) Eclipse (Compiler for floodlight controller code), b) Mininet (Virtual network simulator for test-bed), c) Oracle Virtual Box (Virtual machine container), d) Ubuntu (Operating System), e) Cacti (Monitoring tool for interface utilization of the virtual network).

The languages that we have used for this project development were, i) Java (as the language of the floodlight controller development) ii) JSON (as the RESTful API revocation & parsing), and iii) Python (as the script to create a virtual network in Mininet).

5.1. SDN OpenFlow Controller (Floodlight Controller)

Floodlight is an OpenFlow protocol based SDN Controller to control network traffic in SDN. It is a java based and an Apache licensed controller. It builds on the work done by David Erickson in the Beacon controller [1]. It is one of the best-in-class controller platforms that are in use in commercial products, namely, Big Switch Networks products.

5.2 Open Flow Specification

OpenFlow is an open standard managed by Open Networking Foundation, is one of the first and most widely used open communications protocol for SDN. It allows the SDN Controller to speak to the forwarding plane (switches, routers, *etc.*) of the underlying network elements to make changes to the network.

5.3. Open Flow Enable Network Elements

In this research we have used Open vSwitch (OVS) which has the support for Open Flow protocol.

6. Implementation

The diagram in Figure 1.3 shows our network which was used as a sample case to find the path between any source and destination host in the network based on highest bandwidth and compare the result with the default least hop shortest path.

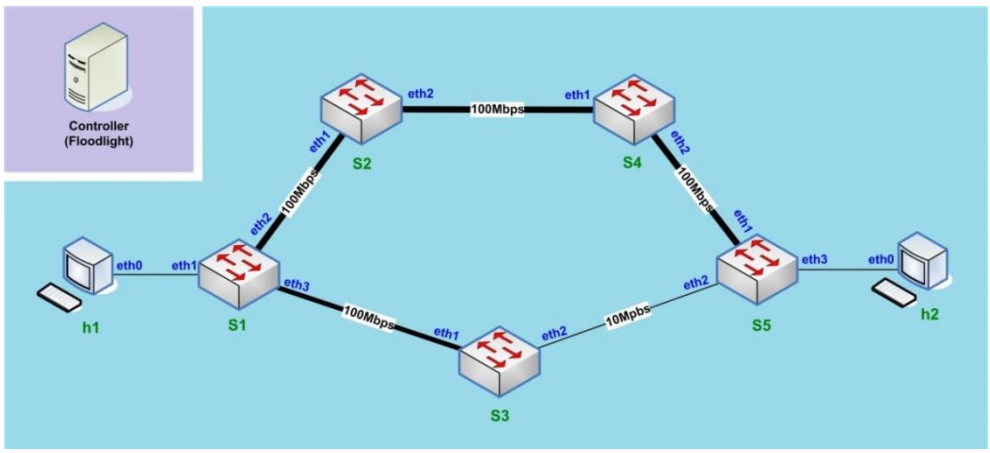

Figure 1.3. Sample Network Diagram

In our project test lab (Figure 1.4), we have used virtual machines to install all the project components. The first virtual machine (VM1) was used for Floodlight Controller and the second virtual machine (VM2) was used for network simulator (in this case, Mininet) to create a virtual network. Both the virtual machines were on the same LAN, but the virtual network elements (switches and hosts) created by Mininet network simulator, were on different LAN. In VM1 we have installed Eclipse. We also download Floodlight Controller & run it from Eclipse. In VM2, in addition to Mininet, we have also installed Cacti. This is a monitoring tool to monitor the bandwidth of the switch interfaces created from Mininet.

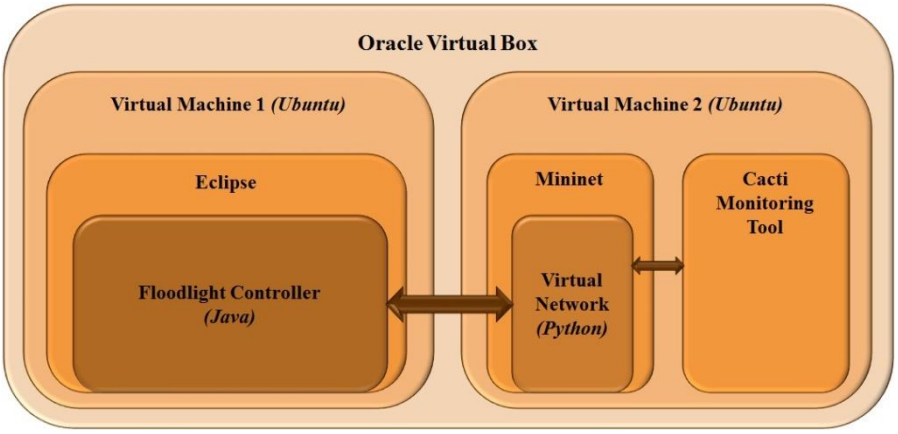

Figure 1.4. Test Lab Setup

According to the diagram Figure 1.3, the network contains two hosts (namely, h1 and h2) and five switches (namely, s1, s2, s3, s4 and s5). The specific interfaces (namely, eth1, eth2, eth3, *etc*.) of the switches and the hosts are also connected accordingly as shown in the diagram.

Floodlight Controller uses dijkstra's algorithm to calculate the shortest path between any source and destination. It takes the nodes and their associate links within the network to calculate the shortest path between any source host and destination host. However, the default implementation does not consider the actual link bandwidth (*i.e*. cost or weight) during shortest path calculation; instead it takes a unit (1) value as the bandwidth for all links. For this reason, the default behavior of the calculated shortest path is always the least hop path. According to the diagram in Figure 1.3 the shortest path between host h1 and host h2 is "host h1 -> Switch 1 -> Switch 3 -> Switch 5 -> host h2" (Total Cost = 2) instead of "host h1 -> Switch 1 -> Switch 2 -> Switch 4 -> Switch 5 -> host h2" (Total Cost = 3). In our sample network, all the links have 100Mbps bandwidth, except the link between Switch 3 & Switch 5 is 10Mbps. For this reason, the bandwidth of the default calculated shortest path (least hop) between host h1 and h2 would always be 10Mbps.

In Figure 1.3, the highest bandwidth path is "host h1 -> Switch 1 -> Switch 2 -> Switch 4 -> Switch 5 -> host h2" which is 100Mbps instead of "host h1 -> Switch 1 -> Switch 3 -> Switch 5 -> host h2" which is 10Mbps. Our implementation is exactly this *i.e*. to optimize the controller in such a way so that it can calculate the path with highest bandwidth instead of least hops.

7. Code Development

7.1. Floodlight Controller

For our implementation we have worked mainly with two files in Floodlight Controller, namely TopologyInstance.java and TopologyManager.java, both are under Topology package (net.floodlight.topology).

7.1.1. TopologyManager.Java

We made the following changes in this file.

- We first imported the libraries java IOException, MalformedURLException, and javax swing JOptionPlane.
- We have modified the startUp() function and implemented Java Dialog option that allowed the user to choose between default & optimized routing algorithm.
- We have also modified the createNewInstance() function. There were two modifications. One was the addition of try-catch structure which allowed the program to catch exception of URL and I/O. And the second one was the addition of function parameter "input" to topology instance compute() function. This input is a String variable which holds the user's choice of the algorithm as stated above. This choice is then passed to the compute() function of TopologyInstance Class by the TopologyManager Class whenever a new instance of TopologyInstance is created from TopologyManager Class.

7.1.2. TopologyInstance.Java

We have made the following changes in this file.

- We have first added the JSON Jackson Jar files (jackson-core-2.5.1.jar, Jackson-annotations-2.5.1.jar and Jackson-databind-2.5.1.jar) as our external Reference libraries in Eclipse. In TopologyInstance.java, we have used Floodlight RESTful API to query switch port information from the underlying OpenFlow switches. Once we get this information, we have then parsed the information using JSON Jackson to yield only relevant data (in this case, link bandwidth).
- We have imported JSON Jakson library files to the code which are core.JsonFactory, core.JsonParser, core.JsonToken.
- In the compute() function, we have added a function parameter which is used to carry the user choice from TopologyManager.java while it creates an instance of TopologyInstance.java class. We assigned this value to a global String parameter "in".
- We have made several optimizations to calculateShortestPathTreeInClusters() function code. The main purpose of this modification is to gather link bandwidth information of any link in the network. First we have removed the loop that checks whether any port is Tunnel port or not. Instead we have created another loop which will first check whether any port is Tunnel port or not and if the port is not Tunnel port then it checks whether this port is any interface port or not. As per OpenFlow Specification version 1.3, Tunnel port indicates any virtual interface or port such as VLAN Interface, Ether-channel Interface, VPN Interface, *etc.* The implementation of the user choice between default and optimized algorithm was our second change. If the user choose default algorithm then all the link bandwidth will be assigned to 1 (default behavior of Floodlight Controller) and if the user choose optimized algorithm then another function getLinkCost() will be called which will find the actual cost of the link in the network (Detail explanation is in the following sub-section). A linkCost HashMap is used to store all the link bandwidth information along with its associated link.

method calculateShortestPathTreeInClusters():


```
        tempCost <- 0
        linkCost = HashMap<Link, Integer>();
        for all NodePort in switchPortLinks
                    if switchPortLinks = null then exit loop
                for all Link in switchPortLinks
                     if Links = null then exit loop
                   nodeInPathTree = NodePort.getNodeId()
                   portOfNodeInPathTree = NodePort.getPortId()
                 tempCost = getLinkCost(nodeInPathTree,
.                                              portOfNodeInPathTree)
                   linkCost <- <link, tempCost>
         for all clusters
            for all node in clusters
                 tree = dijkstra(clusters, node, linkCost, true)
                 destinationRootedTrees <- <node, tree>
end method
```

- We have implemented another function called getLinkCost() function. It took two function parameters namely, NodeId (aka Switch Id which is unique in the network) and NodePort (Port of any node/switch). We have used JSON Jackson library and created a JSON factory and parser. Using JSON we have gathered port-desc information (as defined in OpenFlow version 1.3) in JSON string format for each port of the underlying switch. We have used the following URL to gather the port-desk information from the underlying switch ““http://localhost:8080/wm/core/switch/” +sw+"/port-desc/json””. In Floodlight Controller we have Rest Web service which makes possible to query any network information using URL. There are some pre-defined URL, however any one can create or implement any new URL/service as required. Once we have gathered

```
ashahid@ub-fl1:~/floodlight$
curl http://localhost:8080//wm/core/switch/00:00:00:00:00:00:00:01/port-desc/json

{"version":"OF_13","portDesc":
        [{"portNumber":"1","hardwareAddress":"7a:63:fc:fe:49:1f",
                  "name":"s1-
eth1","config":"0","state":"0","currentFeatures":"2112",
                  "advertisedFeatures":"0","supportedFeatures":"0",
                  "peerFeatures":"0","currSpeed":"10000000","maxSpeed":"0"},

          {"portNumber":"local","hardwareAddress":"0a:1b:cb:25:ce:4a",
                "name":"s1","config":"0","state":"0","currentFeatures":"0",
                 "advertisedFeatures":"0","supportedFeatures":"0",
                "peerFeatures":"0","currSpeed":"0","maxSpeed":"0"}
        ] }
```

The port-desk information we then parsed the JSON string with the provided Node (aka Switch) and Port of the node for “currSpeed” parameter. This currSpeed parameter contains the advertised speed (aka bandwidth) information of the link. This link bandwidth information was then sent back to the parent function calculateShortestPathTreeInClusters() and stored in linkCost HashMap for later use. Thirdly we have used try-catch format while doing the port-desk JSON string parsing and use URL and I/O exception because of the use of URL and Parser.

```
method getLinkCost(nodeInPathTree, portOfNodeInPathTree):
   sw <-  nodeInPathTree, portNumber <- oprtOfNodeInPathTree
   tempBWValue <- 0, inverseBWValue <- 0, maxBWValue <- 10000001
   try
      url <- "http://localhost:8080/wm/core/switch/"+sw+"/port-desc/json"
      parser <- url
      while the parser is not end
         token <- parser.nexttoken()
              if token = null exit loop
         if token = portDesk then
                  token <- parser.nexttoken()
                  if the token is not the start of an array exit loop
   while the parser is not end
                   token <- parser.nexttoken()
                   if token = null exit loop
                   if token = portNumber
                           while the token is not the start of an array
                                token <- parser.nexttoken()
                                if token = null exit loop
                          if token = currSpeed
                               tempBWValue <- currSpeed
                                     inverseBWValue = maxBWValue - tempBWValue
         catch exception of URL and IO
         return inverseBWValue
end method
```

In order to test our implementation we required multiple links from any source to destination with in the network. Figure 1.3 is a sample network with multiple links between source host h1 to destination host h2. We have used Mininet custom script using python to build this custom network. We run the script in such a way so that it build the virtual network and also contact Floodlight Controller to be its SDN Controller.

8. Test-Bed Testing Procedure

Following are the set of steps that we used to test the default and optimized routing implementation in our test-bed for the project.

- Start both the VMs, VM1 (Floodlight Controller) and VM2 (Mininet).
- Login to VM1 and start Eclipse.
- From Eclipse run the Floodlight Controller. Once the Floodlight Controller is running it will listen for any OpenFlow enable switch request or any other packets in the network. A dialog box will appear and ask to choose between “Default (Least Hop)” and “Optimized (Highest Bandwidth)” algorithm. For now we will choose Default Algorithm.
- Login to VM2. From the terminal window, go the Mininet custom folder and run the custom script. The Mininet console will show that the virtual devices (hosts and switches) are created. It will also show whether it has properly able to contact Floodlight Controller or not. We assume that it has properly communicated with the controller.
- Now, switch back to VM1. From the Eclipse console we will be able to see that the controller is getting the request of switches and links. At this stage the Floodlight Controller will create a Topology based on the Default Algorithm.
- Switch back to VM2, Mininet terminal. Establish ping between host h1 to host h2. We will able to see the ping responses.

- At VM2, we will open a browser to access Cacti. We will create a profile for local host (*i.e*. VM1) and create a graph for the virtual interfaces it creates for switches in the Mininet custom script. We will wait for 10 minutes or more, so that the graph gets populated with ping data between host h1 and host h2 and also the necessary switches in the path.
- After waiting for sometimes, we will able to see that only the switches of the path (host h1 -> Switch 1 -> Switch 3 -> Switch 5 -> host h2) has utilization and other switches has no utilization. This means that the default algorithm works and chooses the path with least hop between source host h1 and destination host h2.
- At this stage, we will stop the ping between host h1 and host h2 at VM2 (Mininet). This will flash the flows in the relevant switches. This step is mandatory in order to go to the next step of the test.
- If we wait for long and check the Cacti page, we will find out that there is no utilization in any of the switches. This is due to the fact that there is no traffic in the network.
- At this stage we will switch to VM1 (Floodlight Controller) and stop the Floodlight Controller from Eclipse. This step is mandatory in order to go to the next step of the test. This will allow us to test the other routing algorithm.
- Now, we will again start the Floodlight Controller from Eclipse. The same dialog box will appear and this time we will choose "Optimized Algorithm (Highest Bandwidth)".
- Switch back to VM2, Mininet terminal. Again establish ping between host h1 to host h2. We will able to see the ping response.
- Wait for 10 or more minutes so that the Cacti graph gets populated with data at VM2.
- Finally to verify the success of our modifications we need to wait for sometimes, to see that only the switches of the path (host h1 -> Switch 1 -> Switch 2 -> Switch 4 -> Switch 5 -> host h2) has utilization and other switches has no utilization. This proves that the optimized algorithm works and chooses the path with highest bandwidth between source host h1 and destination host h2.

Following these above steps we were able to test the different routing algorithms and effectively found the difference between the default and optimized algorithm, where the latter is the one that we have implemented in our project.

9. Test Results and Output Screenshots

We have followed the given test bed steps in previous section and have captured the port utilization report from Cacti monitoring tool. We have started both the VM's and started all the applications & tools (Eclipse & Floodlight Controller in VM1) in those VMs. The test procedure was divided into two phases. In first phase we have tested & verified the output of the default algorithm in the sample network followed by the optimized algorithm on the same network in second phase.

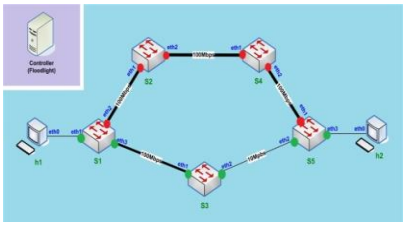

Figure 1.5. Green Dots Showing Path of Default Algorithm

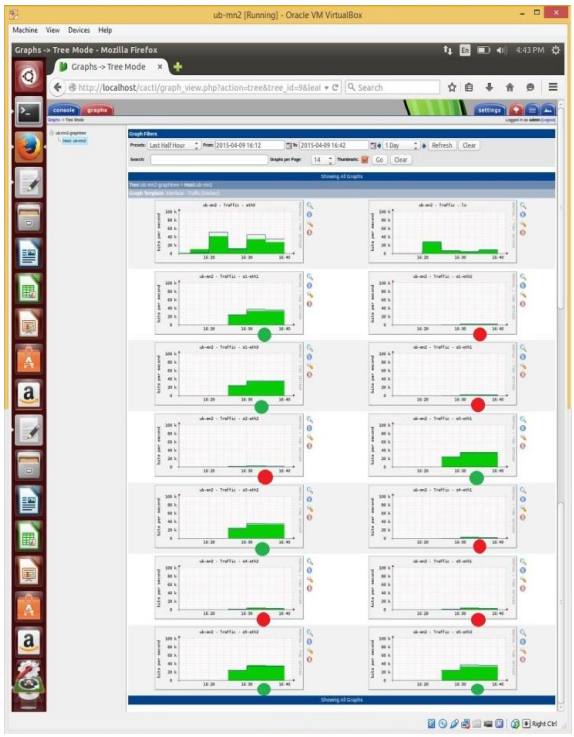

Figure 1.6. Interface Utilization during Default Algorithm

9.1. Phase1: Testing and Verifying Default Algorithm Output

When we use the default routing implementation, we have observed that the interfaces S1-eth1, S1-eth3, S3-eth1, S3-eth2, S5-eth2 and S5-eth3 (Figure 4.2 and 4.3 with green dots) of the sample network were utilized whereas interfaces such as S1-eth2, S2-eth1, S2-eth2, S4-eth1, S4-eth2 and S5-eth1 were not utilized (Figure 1.5 and 1.6 with red dots).

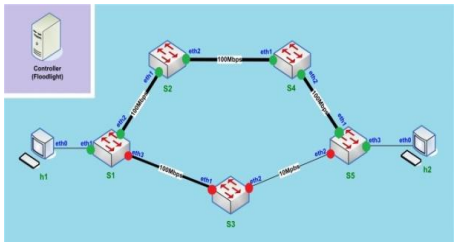

Figure 1.7. Green Dots Showing Path of Optimized Algorithm

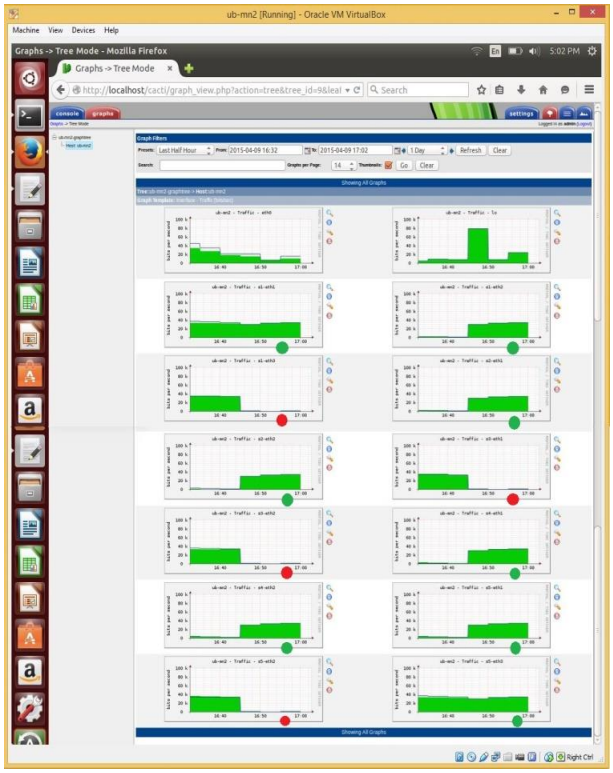

Figure 1.8. Sample Network with Green Dots Showing Optimized Algorithm Path

9.2. Phase2: Testing and Verifying Optimized Algorithm Output

However using the optimized routing implementation, we have observed that the interfaces S1-eth1, S1-eth2, S2-eth1, S2-eth2, S4-eth1, S4-eth2, S5-eth1 and S5-eth2 (as shown green in Figure 4.5 and Figure 4.6) of the sample network were utilized whereas interfaces such as S1-eth3, S3-eth1, S3-eth2 and S5-eth3 were not utilized (as shown red in Figure 1.7 and 1.8).

10. Experimentation Prerequisites

- We must ensure that all the switches flow-tables are flushed out (*i.e*. empty) each time we change our routing algorithm.
- In our implementation, we invoke "currSpeed" parameter (aka Advertised Port Speed) from the underlying OpenFlow switches and input this information to the dijkstra algorithm using RESTful API call. As long as any underlying OpenFlow switches can properly provide us this information which is synchronized with port bandwidth information, our algorithm can effectively find out the highest bandwidth path in any SDN network.
- If we choose default algorithm or none than the Floodlight Controller will proceed to calculate route between any source and destination with least hop.
- If we choose optimize algorithm than the Floodlight Controller will proceed to calculate route between any source and destination with highest bandwidth regardless of that being least hop or not.

11. Conclusion

Our research work attempt to restructure the open-source network (SDN Controller). In this direction we have optimized the Floodlight Controller so that now it can yield highest bandwidth path. However, there are so many other improvements that we are planning to do in our future work including the followings:

- In our implementation we have considered only link bandwidth while calculating any path. However we can also consider some other parameters such as history of link flapping of any particulate link, *etc.* while calculating any path.
- Another good idea is to keep a record of backup route alongside primary route between any source and destination in the network. Once the main route fails due to any fault, the backup route would automatically take over and avoid any packet drop.

References

[1] D. Erickson, "The Beacon OpenFlow Controller", 2010, http://yuba.stanford.edu/~derickso/docs/hotsdn15-erickson.pdf
[2] E. Keller, "University of Colorado; Dmitry Drutskoy", Elysium Digital; Jennifer Rexford, Princeton University - Scalable Network Virtualization in Software-Defined Networks, **(2013)**.
[3] N. Feamster and H. Kim, "Georgia Institute of Technology - Improving network management with software defined networking", **(2013)**.
[4] M. Banikazemi, D. Olshefski, A. Shaikh, J. Tracey and G. Wang, "IBM T. J. Watson Research Center - Meridian: an SDN platform for cloud network services", **(2013)**.
[5] N. Foster, A. Guha, M. Reitblatt and A. Story, "Cornell University, Michael J. Freedman, Naga Praveen Katta, Christopher Monsanto, Joshua Reich, Jennifer Rexford, Cole Schlesinger, and David Walker, Princeton University, Major Robert Harrison, U.S. Military Academy - Languages for software-defined networks", **(2013)**.
[6] S. Sezer, S. S. Hayward, P. K. Chouhan, B. Fraser, .J. Finnegan, N. Viljoen. M. Miller, D. Lake and N. Rao, "Are we ready for SDN? Implementation challenges for software-defined networks", IEEE communications Magazine, July **(2013)**.
[7] S. H. Yeganeh, A. Tootoonchian and Y. Ganjali, "University of Toronto - On Scalability of Software-Defined Networking", **(2013)**.
[8] S. Agarwal and M. Kodialam, "T. V. Lakshman, Bell Labs,Alcatel-Lucent, Holmdel, NJ, USA. - Traffic engineering in software defined networks", **(2013)**.

Authors

Adnan Shahid, He have completed Masters in Computer Science from Lakehead University, Canada, in April, 2015. He has completed his B.Sc. in CSE from CUET, Bangladesh. He has 8+ years of experience in IT Network and Security design, implementation and operation, including 6+ years of working experience in leading telecom organizations (Banglalink & Warid Telecom) in Bangladesh. He is a Cisco Certified CCNA and CCSP.

Dr. Jinan Fiaidhi, He is a full Professor and the Graduate Coordinator with the Department of Computer Science, Lakehead University, Ontario, Canada since late 2001. She is also an Adjunct Research Professor with the University of Western Ontario. She received her graduate degrees in Computer Science from Essex University (PgD 1983) and Brunel University (PhD, 1986). During the period (1986-2001), Dr. Fiaidhi served at many academic positions (*e.g*. University of Technology (Asso. Prof and Chairperson), Philadelphia University (Asso. Prof), Applied Science University (Professor), Sultan Qaboos University (Asso. Prof.). Dr. Fiaidhi

research is focused on mobile and ubiquitous and collaborative learning utilizing the emerging technologies (*e.g*. Data Mining, Cloud Computing, Calm Computing, Mobile Learning, Learning Analytics, Data Science, Social Networking, Croudsourcing, Enterprise Mashups, and Semantic Web). Dr. Fiaidhi research is supported by the major research granting associations in Canada (*e.g*. NSERC, CFI).

Dr. Sabah Mohammed, He started his career during 1977 as a Multimedia Maintenance Engineer working for Canon and Sony following his hobby in Electronics, although he completed his bachelor degree in Mathematics (HBSc 1977). From July 1979 he started his graduate studies where he received his degrees in Computer Science from Glasgow University-UK (PgD 1980, MPhil 1981) and from Brunel University-UK (PhD 1986). Since late 2001, Dr. Mohammed is a full Professor of Computer Science at Lakehead University. Formerly, from 1986-1995, Dr. Mohammed was an Assistant/Associate Professor of Computer Science at various universities including (BU, Amman University, Philadelphia University, Applied Science University and HCT). Sabah is interested in intelligent systems that have to operate in large, nondeterministic, cooperative, survivable, adaptive or partially known domains. Although his research is inspired by his PhD work on the employment of some Brain Activity-Structures based techniques for decision making (planning and learning) that enable processes (*e.g*. agents, mobile objects) and collaborative processes to act intelligently in their environments to timely achieve the required goals, Dr. Sabah extended his research vision to include constructivism and focus more on the nature of knowledge. Since knowledge is created by people and influenced by their values and culture, Sabah research stated to shift more towards net centric systems (*e.g*. Cloud Computing, Social Networking and Enterprise Systems, Web-Based Systems, Big Data, Data Analytics and Data Science).