Graduate School ETD Form 9
(Revised 12/07)

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By Santhan Pamulapati

Entitled

Link Failure Detection In OSPF Network Using OpenFlow Protocol

For the degree of Master of Science in Electrical and Computer Engineering

Is approved by the final examining committee:

Dr. Dongsoo Stephen Kim
Chair

Dr. Brian S. King

Dr. Maher E. Rizkalla

To the best of my knowledge and as understood by the student in the *Research Integrity and Copyright Disclaimer (Graduate School Form 20)*, this thesis/dissertation adheres to the provisions of Purdue University's "Policy on Integrity in Research" and the use of copyrighted material.

Approved by Major Professor(s): Dr. Dongsoo Stephen Kim

Approved by: Dr. Brian S. King 11/20/2013
Head of the Graduate Program Date

LINK FAILURE DETECTION IN OSPF NETWORK USING OPENFLOW PROTOCOL

A Thesis

Submitted to the Faculty

of

Purdue University

by

Santhan Pamulapati

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science in Electrical and Computer Engineering

December 2013

Purdue University

Indianapolis, Indiana

To my grandfather who is battling against cancer.

ACKNOWLEDGMENTS

First and most importantly, I would like to thank my thesis advisor Dr. Dongsoo Stephen Kim for giving me an opportunity to work on this thesis. He has been an immense support and motivation behind my thesis and undoubtedly an invaluable experience working under him. Secondly, I would like to thank my friend Shreya for her encouragement and support.

I express my sincere thanks to members of InCNTRE, Ron Milford for giving me an opportunity to work on SDN/OpenFlow and Uwe Dahlmann, for his guidance. My heart felt thanks to Sudhakar Venkatesh, who has been a great support during my internship at Juniper Networks. Special thanks to Christian Esteve Rothenberg of RouteFlow community and Martin Ivanov for their help.

I would like to thank my thesis committee members Dr. Brian King and Dr. Maher Rizkalla for their precious time. I would also like to thank Sherrie Tucker and Summer Layton for their help.

TABLE OF CONTENTS

LIST OF FIGURES

Figure Page

ABSTRACT

Pamulapati, Santhan. M.S.E.C.E., Purdue University, December 2013. Link Failure Detection in OSPF Network using OpenFlow Protocol. Major Professor: Dr. Dongsoo Stephen Kim.

The study of this thesis is focused on reducing the link failure detection time in OSPF network. When a link failure occurs, OSPF protocol detects it using RouterDeadInterval time. This timer is fired only after a predefined time interval, thus increasing the time of convergence after the link failure. There are previous studies to reduce the RouterDeadInterval time, but they introduce other effects which are discussed later in the thesis. So, a novel approach is proposed in this thesis to reduce the link failure detection time with the help of emerging network architecture Software Defined Networking (SDN) and OpenFlow Protocol.

1. INTRODUCTION

Using the Internet has become an inevitable part of our daily lives. The Internet is a collection of multiple independent networks that are joined together into a single virtual network [1]. It makes users believe being connected to a single, identical network. It carries datagrams or so called data traffic from one end-point (source) to other end-point (destination) and in this process the data traverses through multiple paths. The process of path selection is called routing [2]. To aid the process of routing and to route the datagrams, routers are used, that are building blocks of the Internet.

A router is a network device that connects two or more networks. It receives datagrams from hosts on one network and forwards them to the routers or hosts on other networks. Routers make use of routing protocols to gain the knowledge of topology of the network. Also, these routing protocols learn the networks that are currently reachable and the apt next-hop to use in order to reach a given destination. This routing information (next hop, the cost associated, etc.) is stored in the form of route tables in routers memory. Routing protocols generally fall into two classes: Exterior Gateway Protocols (EGP) and Interior Gateway Protocols (IGPs). IGPs deal with routing within an Autonomous System (AS) [3] and on the other hand EGP handles routing outside an AS. Border Gateway Protocol (BGP) is an example of an EGP.

IGPs are classified into two categories: Distance Vector routing protocol and Link State routing protocol. Distance vector routing protocol is based on Bellman-Ford algorithm [4] which computes the shortest paths from a single source vertex to all other vertices in a weighted digraph. Each router running distance vector routing protocol relies on its neighbors for the routing information, which the neighbors in

turn might have learned from their neighbors. In this way routing tables are periodically broadcasted to all the neighbors in the network. Routing Information Protocol (RIP) is an example for the distance vector routing protocol.

On the other hand link state routing protocol operates by flooding the information related to state of the link in to the network periodically and when state of the network change is sensed. The link state information is stored in the form a database called topology database and each router holds an identical copy of such database. Link state routing protocol is based on Dijkistra's algorithm [5] which also computes the shortest path between the source vertex to all other vertices. Vertices in Dijkistra's algorithm contain whole information of the network topology. Open Shortest Path First protocol (OSPF) stands as an example for link state routing protocols and the focus of this paper is comparison between current implementation of OSPF and OpenFlow based OSPF.

In its current implementation OSPF as a distributed routing protocol works properly for the most of normal situation. But, there are few issues faced by it. When there is link or node (router) failure in the network, OSPF protocol would take some time to detect the failures. The amount of time to detect the failures plays a crucial role in the convergence of the network. Also, it takes time to re-build the path, so that each node in the network has the same view of the new topology. During this transitory, the data meant for the failed device will be thrown down. Also, such scenarios might lead to routing loops in the network which would impel artificial congestion in the network. In the other case where whole network is running OSPF, and one link within it is being flapping every few seconds, OSPF updates would dominate the network by notifying every other router every time the link changes its state. The outcome from it would be Shortest Path First (SPF) calculations for every Link State Advertisement (LSA) that is being propagated.

Also, every time a router has to perform SPF calculations, it would result in high CPU consumption, which could result in the performance degradation. For relatively small networks with few routers the network would converge immediately. But as the size of the network grows, the time taken for network to converge could significantly increase because of link flapping scenarios. Also, when a new OSPF router attaches to the prevailing topology, this event has to be spread throughout the AS and every routers link state database should be synchronized with respect to this change. So, every OSPF router has to go through all the process over and over until entire AS is synchronized.

Considering the above issues faced by OSPF, it would be helpful to have a global view of the network state. A logically centralized mechanism can be deployed over the existing distributed OSPF routing protocol that would help to have a better understanding of the topology. This mechanism would be able to detect the link failures in the network in very short amount of time and thus saves time in terms of convergence. An approach is proposed in this paper to have such central mechanism with the aid of emerging network architecture-Software Defined Networking (SDN) and a new protocol called OpenFlow.

With invent of Software Defined Networking (SDN) [6] in computer networking, it is possible to have a logically centralized software program that would control an entire network. With SDN it is also possible to separate the control and data plane of device, which are tied together in traditional network devices. The decoupled control plane can be directly programmable for having much control over the network. Also, it is flexible to control from high level without touching the low level device configuration through SDN. In Fig 1.1, an approach to SDN is shown, where there are few custom network devices.

The control plane from these devices is abstracted in the form Network Operating System. Just like a normal OS supports many features over it, one can have features like firewall, load balancing or routing running on Network OS.

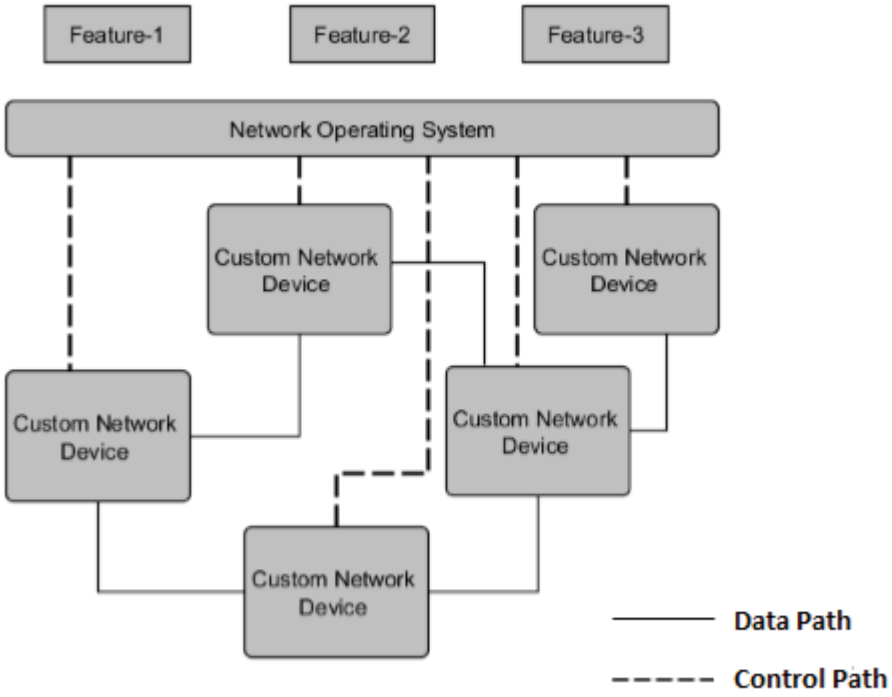

Fig. 1.1. SDN architecture

OpenFlow [6] is the first standard communications interface defined between the control and data plane layers of the SDN architecture. OpenFlow protocol is desirable in order to move the control out of the network devices. The forwarding or Data Plane of the network devices like a switch or a router can be easily managed through the OpenFlow interface. OpenFlow based SDN is currently being implemented in variety of network devices.

2. RELATED WORK

This section would brief the working of OSPF and OpenFlow protocol. Also, this section would discuss the previous work that has been done related to the link failure detection in OSPF networks. OSPF routing protocol is designed to be run internally to a single AS [3]. The successful functioning of an OSPF protocol in an AS depends on: Formation of Link State Database (LSDB), calculation of SPF tree and populating the route table with route entries.

LSDB in an OSPF router holds the information, which describes the topology of the AS. Each such piece of LSDB that belongs to a particular router represents the local state (e.g., neighbors which are reachable from the router and state of the routers interface) of that router. A LSDB is created using LSAs. LSAs often describe native state of router. This description includes the current situation of the routers interfaces and its adjacencies. Router shares this information throughout the AS by flooding. Thus LSAs collected from routers and networks aid in forming LSDB.

After the formation of LSDB its information is used by the router to construct shortest path. For the construction of shortest path, Dijkistras algorithm is used which gives the least cost path to each other router in the AS. Since least path calculation is carried out by each router, the shortest path tree varies from router to router.

Once SPF calculations are completed, the information is used in building routing tables. Routing table consist of route entries for each network in the AS. Each destination network in an AS is reached by performing a route table lookup for matching route entries.

Suppose there is a link failure or a router failure in the network, OSPF protocol dynamically recalculates the routes with respect to the change that took place after a node (router) or link failure.

For exchanging information related to LSDB, LSAs and routing information between neighbor routers, OSPF protocol establishes adjacencies (not every neighbor router will be an adjacent router). Forming adjacencies between routers is a crucial part of OSPF protocol. Only after this step the LSDB are synchronized in the network. For establishing the adjacencies, OSPF uses Hello protocol. This protocol would make sure the communication between neighbors is bi-directional by sending hello packets out of routers interface at a pre-defined interval. In particular a bi-directional link is formed when the router sees itself listed as neighbor in the neighbor field in the hello packet that is generated by other router in the same physical segment. Once the adjacencies are formed, the information related to LSDB are exchanged between the adjacent routers using Database Description (DD) Packets. After receiving these packets, a router may find that parts of the LSDB are out of date and may request for LSDB that are up-to date. This is done with the help of a request packet called Link State Request (LSR). Instead of sending a single LSA, a bundle of LSAs are grouped together and sent in the form of Link State Update (LSU) packets. And for each LSA sent an acknowledgment is sent back in the form of Link State Acknowledgment (LSAck) packet. All these packets are identified by using a type field in OSPF header. All the communication and exchange process depends on which network topology or network type OSPF is operating.

OSPF recognizes four diverse network topologies or network types: Broadcast multi-access, Point-to-Point, Non-Broadcast Multiaccess and Point-to-Multi point. In Broadcast and Non-broadcast networks such as Ethernet or Frame Relay, a large amount of bandwidth is consumed, when OSPF routers have to form adjacencies. To prevent this, an election takes place among the OSPF routers, to elect a Designated

Router (DR). The hello packet contains ROUTER PRIORITY and ROUTER ID field which helps in electing a DR. The value of the priority field ranges from 0-255, and higher the value, a router gets close to be elected as DR. A router with priority set to 0 cannot participate in the election. After a DR is decided, other routers will send updates only to it. DR will then use the multicast address of 244.0.0.6 to send these updates to all other routers in the network. If a DR fails a Backup Designated Router (BDR) takes its place. BDR is the second best router after a DR in the last election. But in Point-to-Multipoint and Point-to-Point networks there is no such election which decides a DR.

The information provided in the above paragraphs described the abstract working of OSPF protocol. In general OSPF would run as a process inside the route processor (refer to Fig 2.1) which is present in the router. When a router receives an LSU, OSPF uses these messages to build a link-state database. SPF calculations are run on this database to build a Forwarding Information Base (FIB) or so called routing table. In order to store these routing tables, routers use their memory. When a network interface card sees the data, it would refer the routing table. This way routing table helps to pick the next hop for the data to traverse. The switching fabric helps to pass the data from one interface to another.

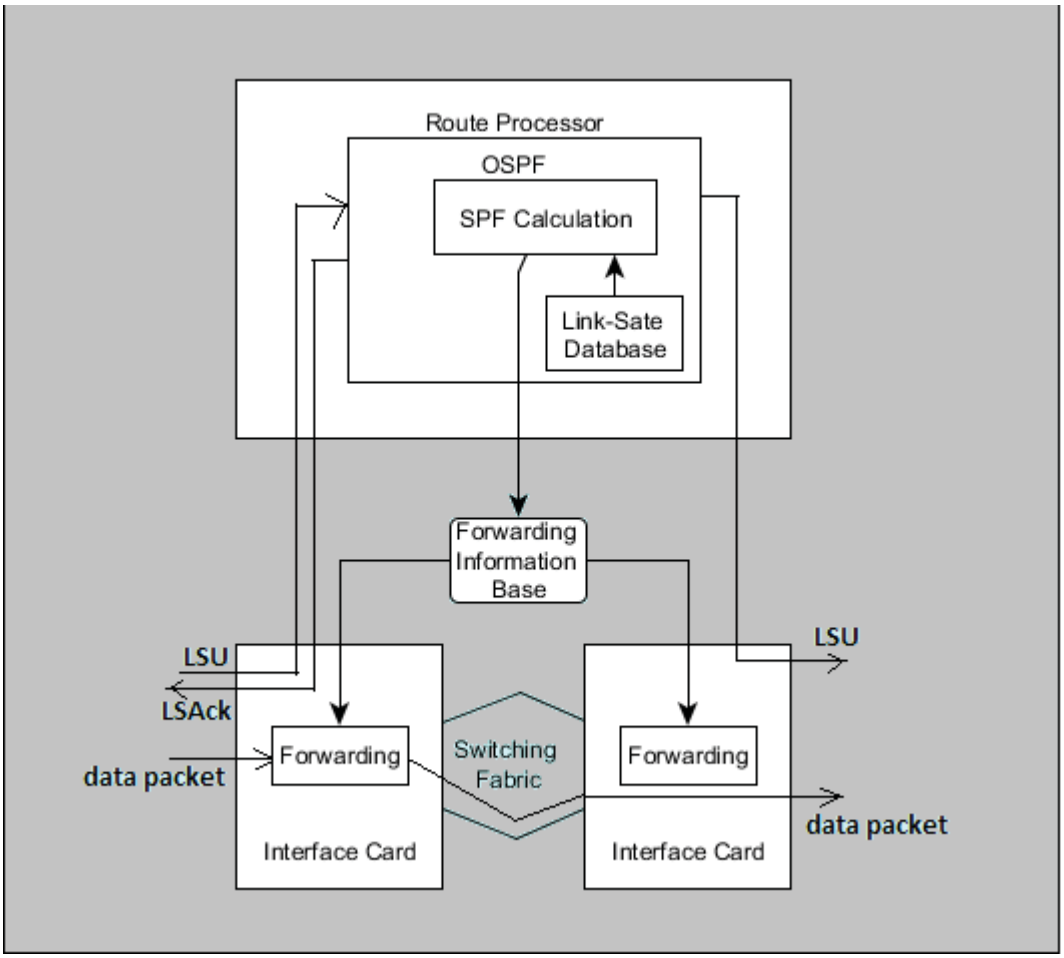

Fig. 2.1. Inside OSPF router

In the above Fig 2.1, the control plane and the data plane of the router are tied together and this is the kind of setup that traditional or legacy network devices have in them. With this kind of setup it is difficult to have control over the FIB, because the control plane is vendor dependent. If the user tries to add his/her own applications, it might break existing processes that are already running. One of the possible solution to this problem is to separate control and data plane so that the user can program the control plane according to the needs. This idea has led to a new architecture called SDN and OpenFlow is the first standard based interface defined between the control and data plane.

2.1 OpenFlow protocol

Today's network devices like switches or routers have their control and data plane tied together, as show in Fig 2.2. The control plane would build information and make the forwarding/routing decisions and populate the forwarding or routing table and data plane would forward the data according to it. With this kind of setup, it is difficult to have high-level control over these network devices.

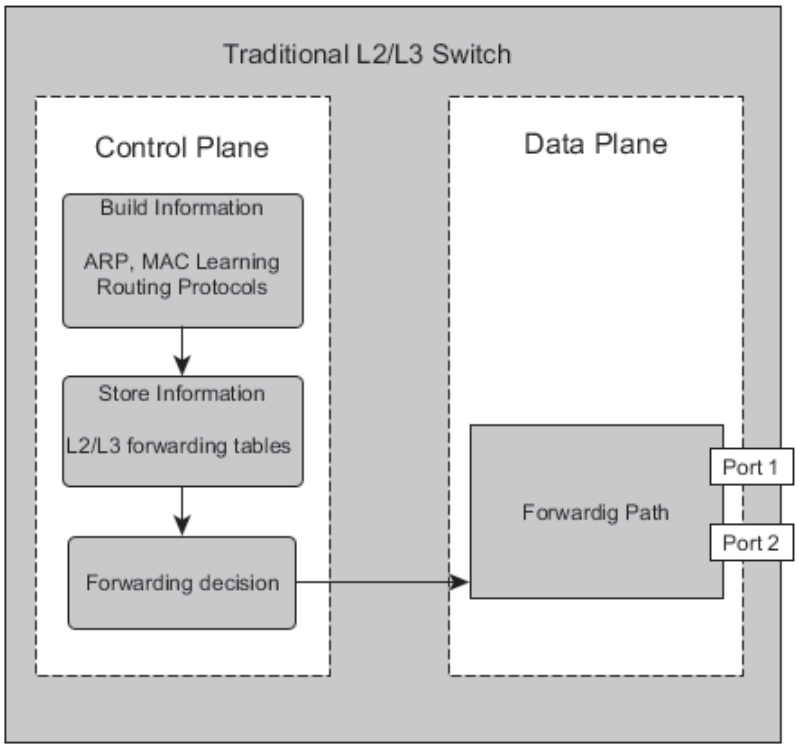

Fig. 2.2. Traditional network device with control and data plane tied together

In contrast, an OpenFlow enabled device as in Fig 2.3 has its control plane hosted inside a server (often called as OpenFlow controller) that would run applications defined for definite purpose and can be monitored by a network operator. One of the controller applications can be a learning switch module (which helps in learning MAC address of the hosts that are connected to the switch). The data plane still resides

in the device and is abstracted in the form a flow table, which can be programmed. Controller manages OpenFlow switch over a separate secure channel using the OpenFlow protocol. This protocol is implemented on both sides i.e. on controller side and also in the devices.

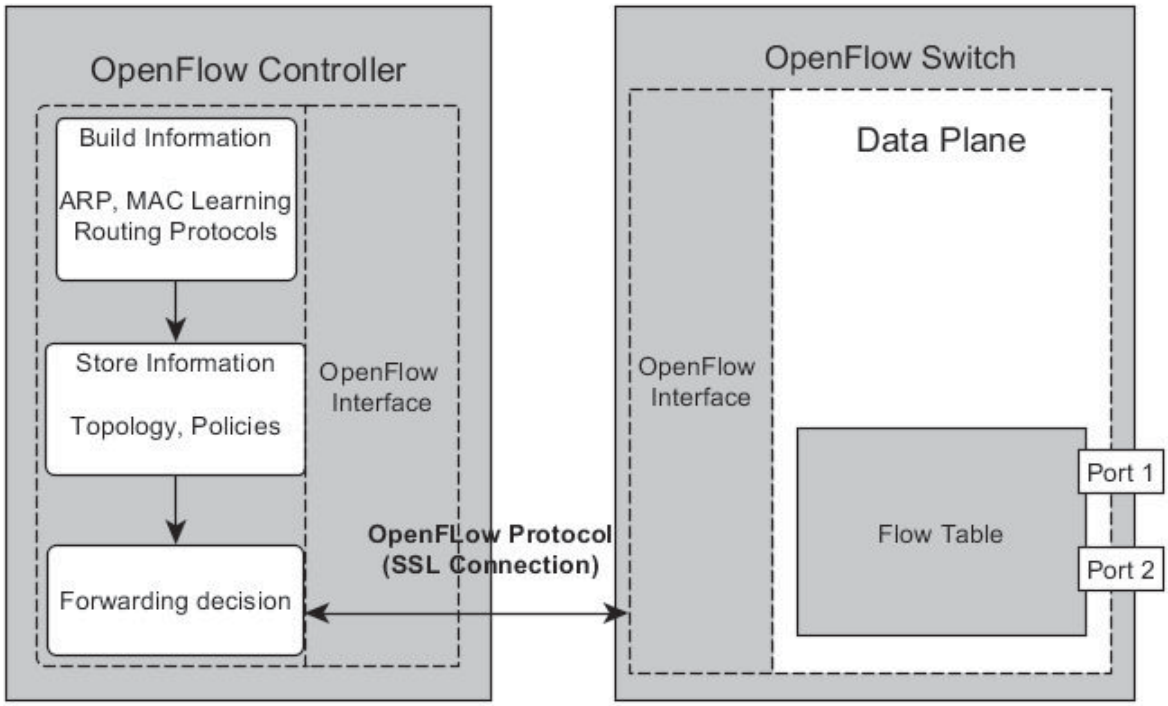

Fig. 2.3. OpenFlow controller and switch

An OpenFlow switch maintains a flow table (instead of forwarding table), which contains set of flow entries as shown in Fig 2.4 which are inserted in the flow table using the controller. Each flow entry consist of header fields (to match against incoming packets), actions (to apply to matching packets) and counters (to update for a matching packet). The incoming packets are compared against the flow entries and if a matching entry is found, respective actions are applied and counters are updated accordingly. The power of OpenFlow protocol lies in the set of actions it supports. For example, a packet matching a flow entry can be modified by its layer2 and layer

3 header fields. If there is flow entry in the switch with no action specified, that packet would be dropped. Also, if there is no matching entry, then the entire packet or a part of the packet is sent to the controller. All the communication between the controller and OpenFlow device is in the form of OpenFlow messages and all these messages carry OpenFlow header.

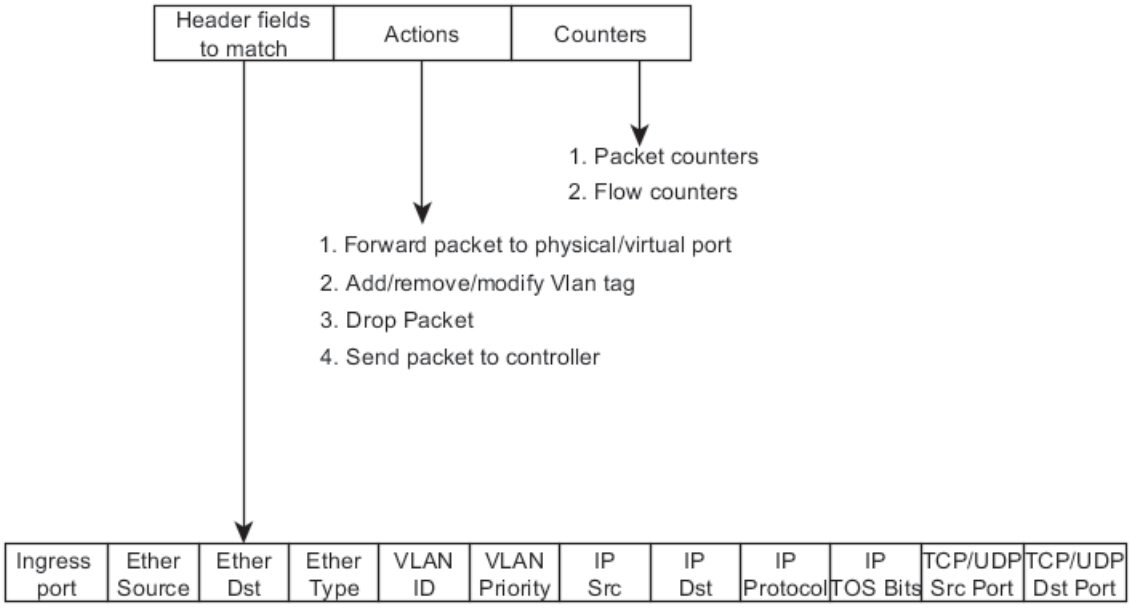

Fig. 2.4. A typical flow entry inside an OpenFlow device

The format and the fields of OpenFlow header are shown below in Fig 2.5. The version field represents the OpenFlow version being used. Both controller and the switch need to negotiate the OpenFlow version at the connection establishment. In this paper OpenFlow version is limited to 1.0. The type field represents the OpenFlow message type and OpenFlow supports three message types: Controller-to-switch messages, Asynchronous messages and Symmetric messages.

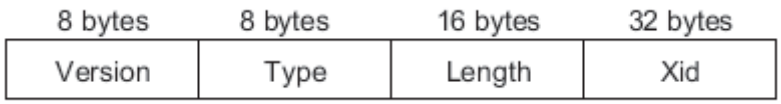

Fig. 2.5. OpenFlow header format

Controller-to-switch messages are purely initiated by OpenFlow controller and may or may not require a response from the OpenFlow device. Some of the important Controller-to-switch messages are

1. Features: With this message the controller is able to query the switch for the capabilities that it supports (like the number of physical and virtual ports supported by the switch, actions supported, etc.)

2. Modify-State: With the help of this message controller can add, delete or modify a flow entry in the flow table.

Asynchronous messages are sent by the OpenFlow switch to the controller in order to notify the arrival of packet, change of switch state (port up/down) or to report an error. Some of the important asynchronous messages are

1. PACKET_IN: If there is no matching flow entry in the flow table, then the entire packet or a part of the packet is sent to the controller. This is sent as Packet_In message to the controller, which contains the information of the port, on which the frame was receive and also the reason for sending it to the controller (no matching flow or action was to output to controller). Only the first packet of the flow is sent to the controller. Except the first packet, the rest of them are compared against the flow entry inserted by the controller and are forwarded according to the action(s) specified.

2. Flow removed: Every flow entry has a timeout associated to it. When a flow expires due to a time out, the switch notifies the controller using the FlowRemoved message.

Symmetric messages are OpenFlow messages like Hello and Echo request/reply. These are initiated either by a controller or an OpenFlow switch. Hello messages are used in version negotiation and Echo request/reply messages are used in maintaining the connection.

The length field in OpenFlow header represent the length of the OpenFlow packet. The length includes the OpenFlow header and the data sent. Xid specifies the transaction id associated with the respective packet. If a request is sent, the corresponding reply will share the transaction id. Simply put together, OpenFlow uses the idea of flows to classify network traffic based on pre-defined match rules that can be statically or dynamically programmed by the control software [6].

2.2 Previous work

The major concern for today's network infrastructure is fast rescue from the link failures and fast convergence to topology changes. Apart from these aspects, in OSPF network, processing and bandwidth requirements also draw equal importance. When a device (router) or a link(s) failure occurs, OSPF protocol needed several tens of seconds to recuperate from the failure and real time applications like Voice Over IP (VoIP) [7] cannot stand the breakdown ranging such duration. There were previous research works related to the failure identification and improving convergence time in OSPF networks. In the IP network failures occur due to hardware/software issues in the routers or due to fiber cuts. These issues may result in single or multiple links or router(s) failures. Markopoulou et al. [8] in their study on failures related to operational IP backbone networks found that nearly 70 percent of the failures were single link failures. Often, defective hardware/software leads to flapping behavior of the links. This kind of activity has severe effect on data traffic.

Failures in OSPF are identified by using the default hello protocol. After the establishment of adjacency with the neighboring routers, a HelloInterval is set with a default value of 10 seconds. Hello packets are exchanged in this interval. Whenever a router receives a hello packet, inactivity timer which is associated with the router is reset. This timer is triggered after the RouterDeadInterval (which is generally four times the HelloInterval). With this kind of setup it would take 30-40 seconds of time to detect the failure which is highly undesirable. So, efforts have been made to reduce the HelloInterval. Alaettinoglu et al. [9] suggested to reduce the HelloInterval to millisecond range to attain sub-second failure detection. But, this approach would effect the CPU of the OSPF router to be overloaded. The lowering of HelloInterval would lead to loss of consecutive Hello messages, which in turn results in false breakdown of the adjacency link. Due to this false failure, LSAs are generated leading to new routing table calculations. Basu and Riecke [10] from their observation reported that reducing the HelloInterval to 500ms would not overload the CPU. However there is an increase in the number of route flaps when the interval was further reduced to 250ms. Feng et al. [11] observed that, the frequency at which these false alarms would trigger depend on the congestion in the network and also on the links. In this paper an approach is proposed to detect the link failures effectively (without using the Hello protocol) using OpenFlow protocol and there by avoid the packet loss during the event of link failures.

3. PROPOSED IDEA

The Hello messages (refer to Fig 3.1) in OSPF are used to maintain the link adjacency and these messages are also used to detect the link failures. HelloInterval and RouterDeadInterval are two important fields inside a hello messages which are used in maintaining adjacencies and detecting link failures. In general, the default HelloInterval for point-to-point and broadcast networks is 10 seconds. With this configuration, a router running OSPF protocol would take at least 40 seconds (because RouterDeadInterval for above network types is four times the HelloInterval) to detect the link failure. So, there is a delay involved from the time actually a link went down to the time it is actually detected.

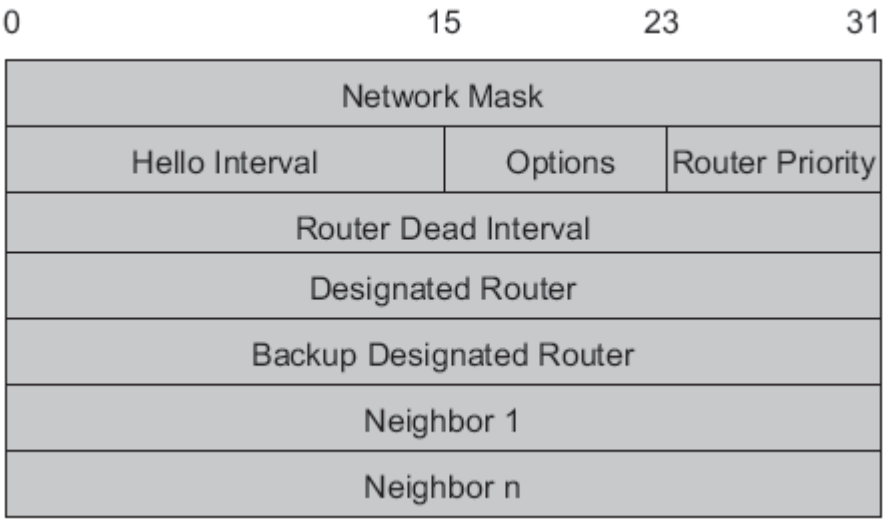

Fig. 3.1. OSPF hello message

Before the network converges again after the link failure has been identified, there are other delays involved. The link failure insists the router to generate an LSA and flood it across all its interfaces. The LSA flooding time comprises of the propagation delays (typically in milliseconds). After receiving a new LSA, router schedules an SPF calculation. As SPF calculation uses Dijkistras algorithm [5] it constitutes significant processing load. For this reason the router will wait for some time (spfDelay - typically 5 seconds) for other LSAs arrive before performing an SPF calculation. In additional to spfDelay, routers govern the frequency of SPF calculations through spfHoldTime (typically 10 seconds between successive SPF calculations) which introduces further delays. All these delays contribute to the failure recovery time of the OSPF network. Clearly among all the delays, the link failure detection time plays a major role in the failure recovery. Hence reducing the failure detection time would help the network to recover from the failure quickly.

As discussed earlier, in SDN, the control and data plane are separated and OpenFlow protocol is used as the medium for communication between control and data plane. For reducing the link failure detection time in OSPF network, OpenFlow protocol and OpenFlow network can be used. An OpenFlow network consists of switches that support OpenFlow protocol and this kind of network can be easily emulated using an open source tool called Mininet [12]. An OpenFlow switch consists of ports on which the data traffic can be sent and received. The link status of the port (if the port is up/down) can be monitored using one of the OpenFlow messages. Precisely, OFPT_PORT_STATUS [13] message is used to notify the controller of any change in the link status. Refer to Fig 3.2 for the signaling process.

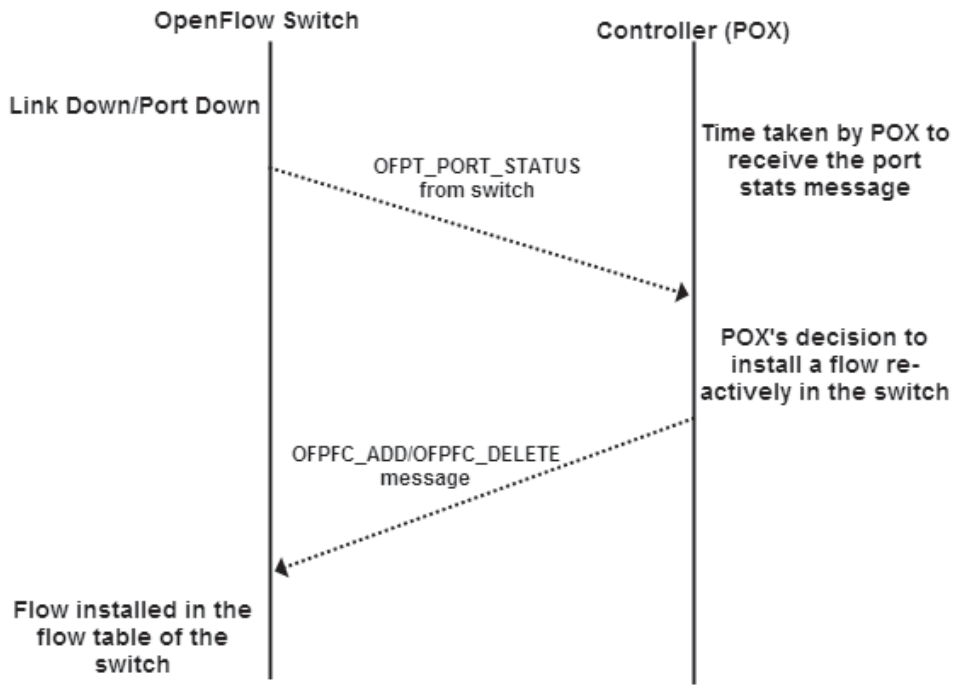

Fig. 3.2. Signalling of port down event

In OFPT_PORT_STATUS message, the reason for the change of state of the link is appended and is sent to the controller. For example, consider two OpenFlow switches connected with a link. If one of the port connecting both the switches goes down, then immediately an OFPT_PORT_STATUS message is generated with reason as OFPR_DELETE (which means a port is down). The time taken by the controller to receive this message can be considered as the time to detect the failure of the link. To estimate the time taken for controller to receive the OFPT_PORT_STATUS message, a simple test bed with two OpenFlow switches and controller has been setup (refer to Fig 3.3).

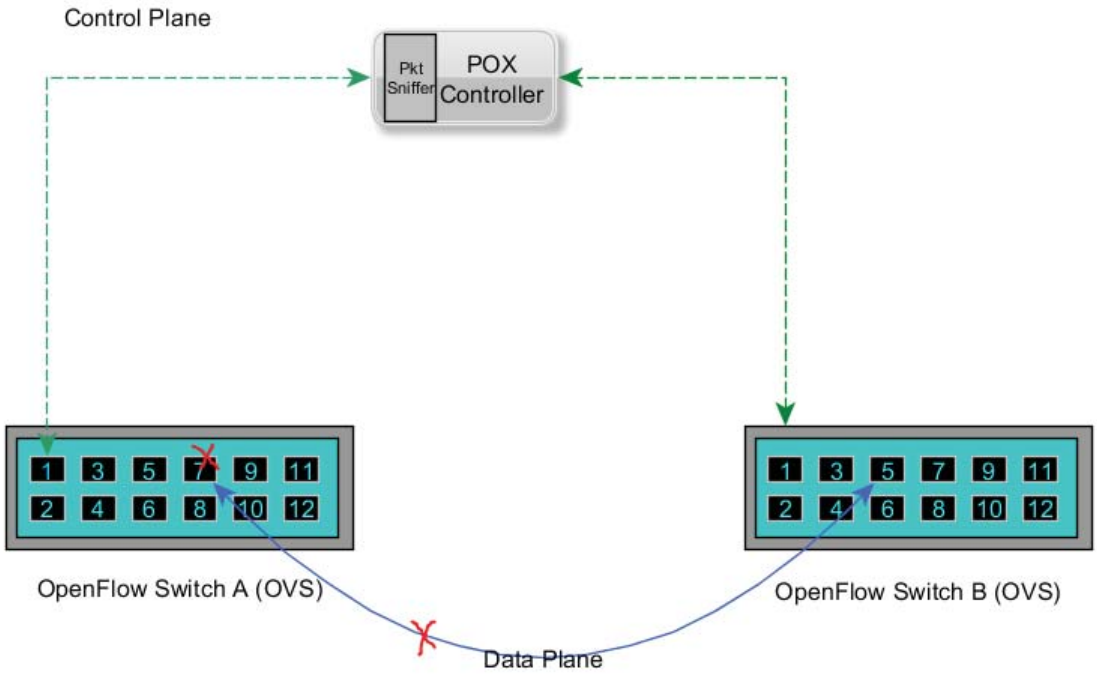

Fig. 3.3. A simple testbed of OpenFlow switches

In the Fig 3.3, two OpenFlow switches are created using Mininet, that implements software version of OpenFlow switch called Open Virtual Switch (OVS) [14]. These two switches are then hooked to an open source controller POX [15] (written in Python language) via a dedicated control channel. Now, when the 7th port of switch A fails, the link between both the switches is broken. Both the switches detect this failure and immediately send an OFPT_PORT_STATUS message to the controller. At the controller these OpenFlow messages are captured using a packet sniffer application called Wireshark [16]. These messages are time stamped so that their arrival time at controller can be estimated. From the results it was observed that the time taken to receive the OFPT_PORT_STATUS by the controller is approximately 0.02 milliseconds. Once the failure has been detected, the controller can make the decision to add or delete a flow using OFPC_ADD or OFPC_DELETE message.

The time taken by the controller to make this decision introduces delay. This delay can be compared with spf delay that OSPF introduces to calculate new routes. The delay introduced by POX is calculated using a tool called Cbench [17]. It is the current standard for evaluating OpenFlow controller performance. A latency test is performed using Cbench which emulates one OpenFlow switch. This test sends a single packet to the controller and waits for a reply. It repeats this process as quickly as possible. The total number of responses received at the end of the time period can be used to compute the average time for the controller to process each message. From the results it was observed that, POX takes almost 0.06 milliseconds to process the flow. This delay is much smaller than spfdelay. But, the disadvantage of POX is, as the network scales further delay may be introduced. This is because a single controller has to handle all the flows. Also, the decision made by controller is entirely dependent on the application (L2 Forwarding switch, firewall, load balancer, etc.) that runs over it.

From the above discussion, OpenFlow protocols capability to instantaneously detect the link failures can help OSPF routers to reduce the link failure detection time. When a routers interface goes down, OpenFlow protocol can quickly notify this event to the controller and the decision to install a flow entry in the data plane can be made instantaneously. In this way, OSPF router don't have to wait till the RouterDeadInterval time to detect the failure. In order to test this approach, RouteFlow [18] has been used which makes OpenFlow switches behave as routers.

3.1 Brief description of RouteFlow architecture

RouteFlow [18] is an open source project which provides virtualized IP routing services over OpenFlow networks. It stores the control logic (control plane) of the OpenFlow switches in a virtual network, composed of virtual machines (VM). These VMs can be interconnected to form a logical topology that mirrors the discovered

physical topology (OpenFlow switches) accordingly. The Network Interface Card (NIC) of the VMs are connected to a software switch like OVS, for their management. Also, these VMs run open source routing engines (ex: Quagga, which supports routing protocols like OSPF, BGP, etc.) that generate FIB in the Linux IP table, but not directly in the OpenFlow switch. RouteFlow converts these IP table entries into OpenFlow flow entries and installs them in the respective OpenFlow device. As a result of this architecture, control is centralized and it stays logically distributed. But, still the flow entries are upheld in the data plane to specify how traffic must be handled [13] (i.e port forwarding, MAC re-writing, TTL decrement, etc.).

The architecture of RouteFlow (refer to Fig 3.4 consists of three important components: RouteFlow Client (RF-Client), RouteFlow Server (RF-Server) and RouteFlow Proxy (RF-Proxy).

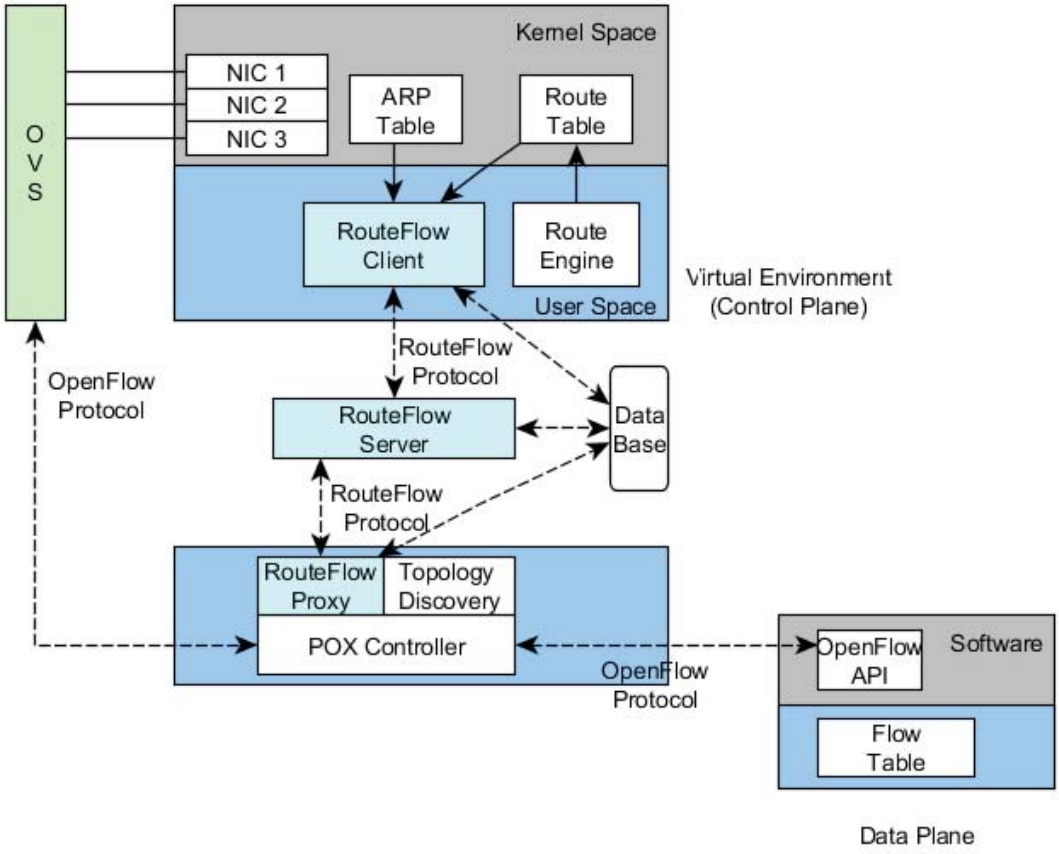

Fig. 3.4. Architecture of RouteFlow

RF-Client sits as a daemon in the VM instance. It keeps track of changes in the Linux ARP and routing tables and sends the collected routing information to the RF-Server. The VMs running the RF-Client daemons are managed by RF-Server. The RF-Server maintains the mapping between RF-Client VM instances and interfaces and the corresponding OpenFlow switches and ports. It also instructs RF-Proxy to configure flows to be installed in the OpenFlow switch as needed. RF-Proxy is an application for POX and other controllers. It controls the interactions with the OpenFlow switches (identified by DPID) through OpenFlow protocol. It takes instructions from the RF-Server and notifies it about events in the network. For all the

communication between RF-Client, RF-Server and RF-Proxy, a simple RouteFlow protocol has been defined. This protocol is based on a set of Inter Process Communication (IPC) messages. These messages carry information like flow modification and Packet_In. In this way RouteFlow provides routing over OpenFlow network.

In its current implementation, RouteFlow does not support the events of link failures. This means, when a link between OpenFlow switches in data plane is down, the failure is not replicated in the VM of control plane. Also, the failure is detected only according to the configured RouterDeadInterval which results in delayed link failure detection, which in turn results in delayed convergence. So, the code has been added to notify the link failure events to RouteFlow. When the RF-Proxy is capable of receiving link failure events, it can instruct the RF-Client daemon which sits in the VM to shutdown the respective interface of the VM. When OSPF finds that interface went down, it will not wait for the RouterDeadInterval to be fired. Rather it notices this change in the state of the interface and immediately generates an LSA and floods it out. In this way, the link failure is detected quickly (in millisecond range) and efficiently with the help of OpenFlow and RouteFlow.

4. IMPLEMENTATION

This chapter presents the implementation of traditional OSPF and OpenFlow based OSPF. The first Section 4.1 would discuss about emulating an OSPF network environment over a Linux machine. Later in Section 4.2, we extend the same environment to support OpenFlow based OSPF with the help of RouteFlow architecture. It would also discuss the implementation of the proposed idea from Chapter 3.

4.1 Linux based traditional OSPF:quagga

Quagga is an open source routing software suite and is licensed under GPL. It provides implementations of various routing protocols like OSPFv2, OSPFv3, Routing Information Protocol (RIP) and Border Gateway Protocol (BGP). All the protocols inside Quagga are implemented according to the IETF standards. There are other open source routing software suites like XORP [19] and BIRD [20] which can serve the same purpose. But, for this study we have decided to go with Quagga because of its active development community. Currently Quagga can be implemented over the platforms like GNU/Linux and BSD. Each protocol running on Quagga is a separate process. This gives the feasibility to modify any protocol, without effecting the other. This way, the failure of one process would not affect the other processes. Each protocol is handled by different routing daemons and each daemon has its own routing table. For example OSPFv2 is handled by ospfd and BGP by bgpd. Zebra daemon serves as a moderator for allocation and distribution of services and resources to various daemons. It is responsible for interacting and installing routes in the kernels routing table. The architecture of Quagga is show in Fig 4.1

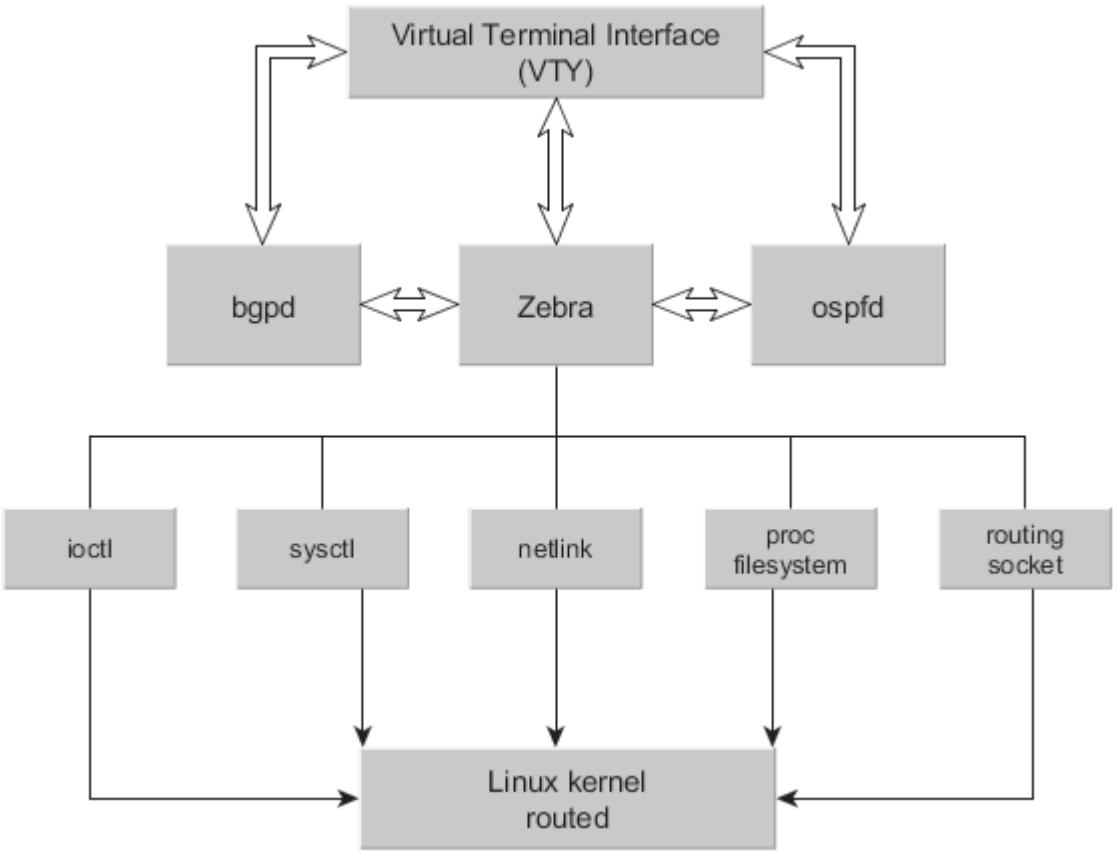

Fig. 4.1. Architecture of quagga routing suite

Also, each daemon has its own configuration file and a terminal interface to interact. When configuring an OSPF network, it must be specified in ospfd configuration file. If the user wishes to change a particular aspect of the OSPF routing protocol (e.g. to change the HelloInterval time) it can be done with the help of the terminal interface. For example, if the user wants to see the SPF tree built by OSPF protocol, he can simply issue a command (in privilege mode) from the interface as show in Fig 4.2.

```
R1# show  ip ospf  database

       OSPF Router with ID (80.0.0.1)

                Router Link States (Area 0.0.0.0)

Link ID         ADV Router      Age  Seq#       CkSum  Link count
50.0.0.1        50.0.0.1         994 0x80000009 0x74ad 3
60.0.0.1        60.0.0.1         949 0x80000008 0x856b 3
80.0.0.1        80.0.0.1         959 0x8000000b 0x98b8 4

                Net Link States (Area 0.0.0.0)

Link ID         ADV Router      Age  Seq#       CkSum
20.0.0.1        80.0.0.1         994 0x80000002 0x0a64
30.0.0.1        80.0.0.1         959 0x80000002 0xf564
```

Fig. 4.2. Using virtual interface to interact with ospfd

4.1.1 Configuring OSPF in quagga

In order to make a Linux machine behave as an OSPF router, there are few parameters that have to be configured. First, the zebra and ospfd daemons have to be enabled. This can be done by editing the daemons file under /etc/quagga. Then ospf parameters are set either by editing the ospfd configuration file or through command line (virtual interface).

We can start with setting the time of HelloInterval. The minimum possible value for HelloInterval is 1 sec and the default is 10 sec. Then, the RouterDeadInterval is set to four times the HelloInterval time. Each OSPF router should identify in the network with a RouterID. If no RouterID is configured, then ospfd uses the highest IP address of physical or virtual interfaces of the Linux machine. Finally, the router should belong to an area and two OSPF routers can communicate only if they belong to same area. In this study all the routers belong to special area called backbone area or area 0.0.0.0. A sample configuration with all the OSPF parameters set is shown in Fig 4.3.

```
!
! Zebra configuration saved from vty
!   2013/10/20 09:20:24
!
hostname ospfd
password zebra
log stdout
!
!
!
interface eth0
!
interface eth1
 ip ospf hello-interval 5 20.0.0.2
 ip ospf dead-interval 20 20.0.0.2
!
interface eth2
 ip ospf hello-interval 5 40.0.0.1
 ip ospf dead-interval 20 40.0.0.1
!
interface eth3
 ip ospf hello-interval 5 50.0.0.1
 ip ospf dead-interval 20 50.0.0.1
!
interface lo
!
router ospf
 network 20.0.0.0/8 area 0.0.0.0
 network 40.0.0.0/8 area 0.0.0.0
 network 50.0.0.0/8 area 0.0.0.0
```

Fig. 4.3. A sample of ospf config file used in quagga

After all initial configuration parameters, finally the OSPF process has to be enabled for that particular ospfd. Then, the daemon has to be restarted for the changes to be effective. Once the daemon comes up, it starts sending Hello messages out of its interface(s). A sample wireshark trace is provided in the Fig 4.4. With this, a Linux machine is turned in to an OSPF router.

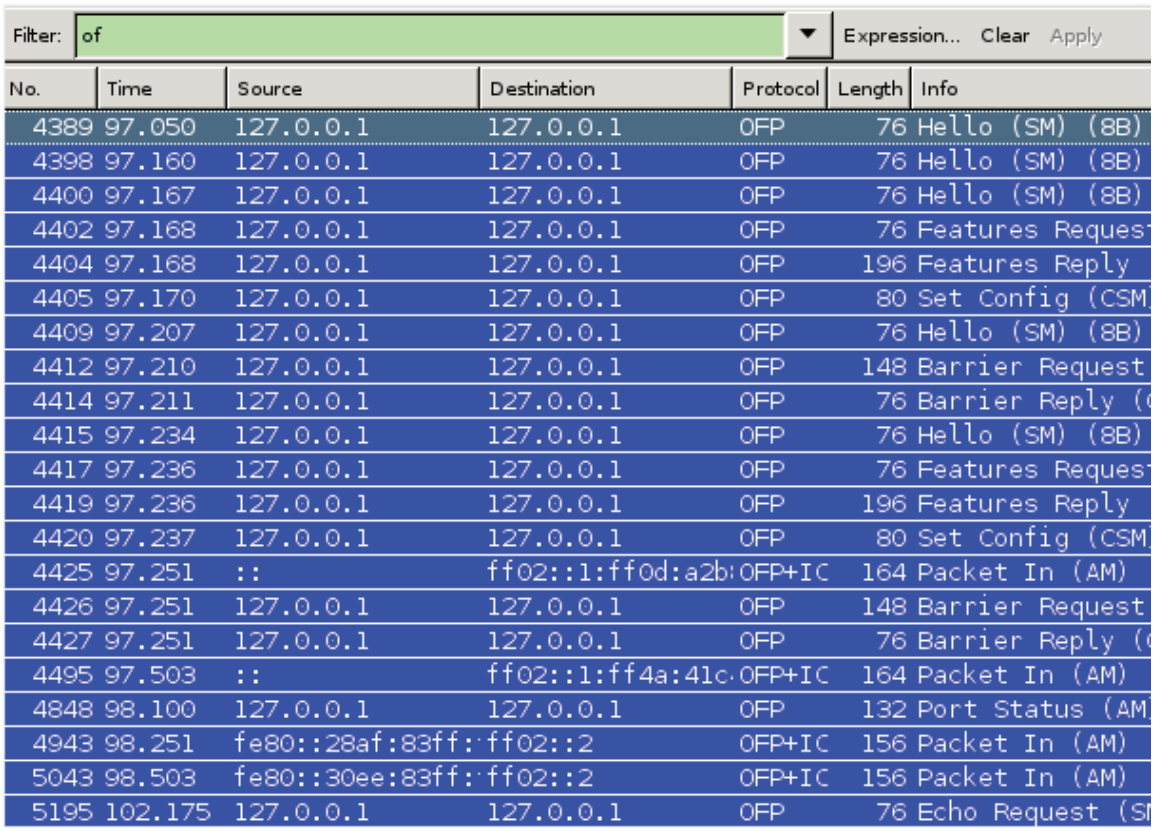

Fig. 4.4. Wireshark trace showing OpenFlow messages

4.1.2 Setting up multiple OSPF routers

An OSPF network consists of two or more routers and this network environment can be emulated using different Linux machines. But, to save resources we have used virtual Linux machines (LXC) running inside single Ubuntu (12.04) server to emulate the required OSPF network. This also makes debugging and management of the network much easier.

LXC (Linux Containers) [21] is an operating system-level virtualization method for running multiple isolated Linux systems (containers) on a single control host. It does not provide features of a full virtual machine.But it provides a virtual environ-

ment that is close to that of a fully virtualized Linux machine. Also, each Linux system (container) has its own process and network space. LXC provides resource management through the control groups aka process containers and resource isolation through the namespaces [22].

LXCs are created using a single command and each container has its own configuration file. This file is used to define the network and mount parameters. To ease the task of creating multiple containers, LXC provides the option of cloning. A single container can be created with all the required packages installed. Later that single container can be cloned for any number of containers using a LXC command. Refer to Appendix LXC commands for the list of commands used in creating multiple containers.

After creating multiple containers, Quagga routing suite is installed in each of them. Then, the process of turning a Linux system in to an OSPF enabled router is followed as discussed in Section 4.1.1.

4.1.3 Connecting OSPF routers together

For connecting the virtual interfaces of the emulated OSPF routers together, a Linux bridge can be used. In general a hardware bridge [23] is a way to connect two Ethernet segments together in a protocol independent way. A Linux bridge is more usefull than its hardware counterpart as it allows to filter and shape the traffic passing through it. Since, the containers are based on virtual environment, a Linux bridge might not be suitable to manage the virtual interfaces. For this reason, we have decide to go with Open Virtual Switch (OVS) [14]. OVS is a software switch and is designed to be used in virtualized environments.

It forwards the traffic between different virtual machines on the same physical host. It is designed specially to manage virtual machine network configuration. It can be managed by a Command Line Interface and OpenFlow protocol.

We connect every virtual interface of emulated OSPF routers to the OVS. This is done by creating an OVS bridge and its ports. Now, the virtual interfaces are attached to the created ports. OVS bridge and ports are created using ovs-vsctl program which is a part of OVS software.

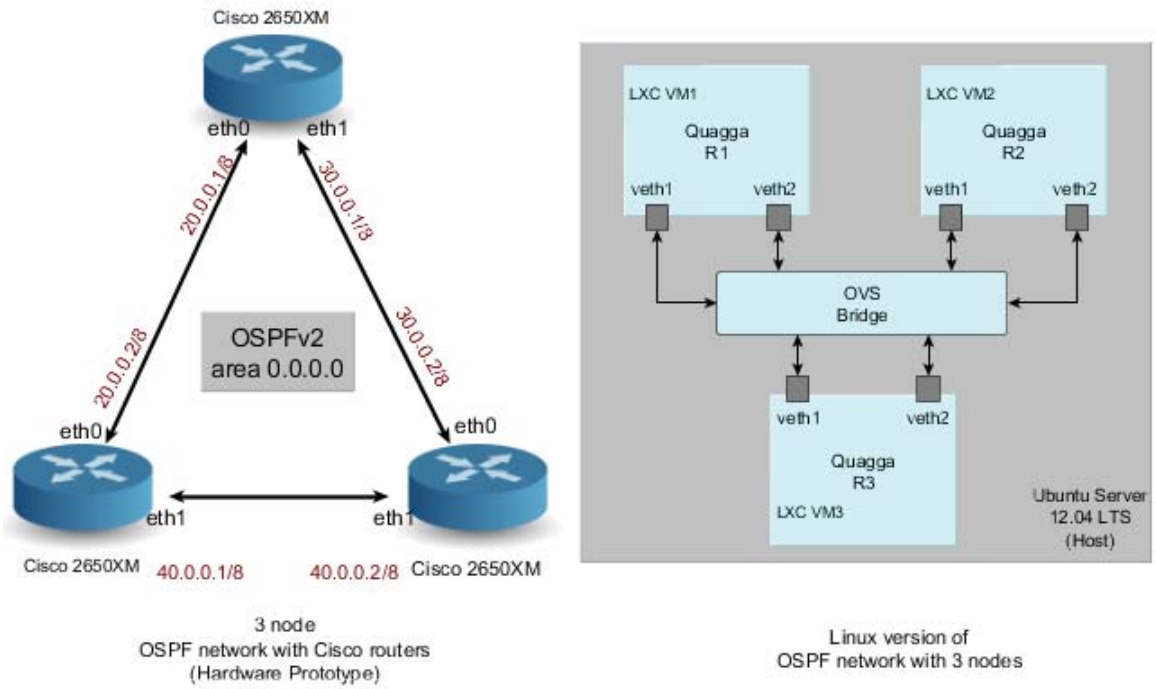

Fig. 4.5. Traditional OSPF network - hardware prototype vs. linux environment

The entire setup of OSPF network in Linux environment is compared with the hardware setup in Fig 4.5. The figure shows a simple three node topology of both the setups. Linux based OSPF routers form adjacencies and exchange Database Description (DD) packets and finally converge after a few seconds. Wireshark traces

are provided in the Fig 4.6 for the reference. Thus from above sections a traditional OSPF network can be emulated in the Linux environment with the help of LXC, OVS and Quagga routing suite.

4.2 Wirehark trace of OSPF packets

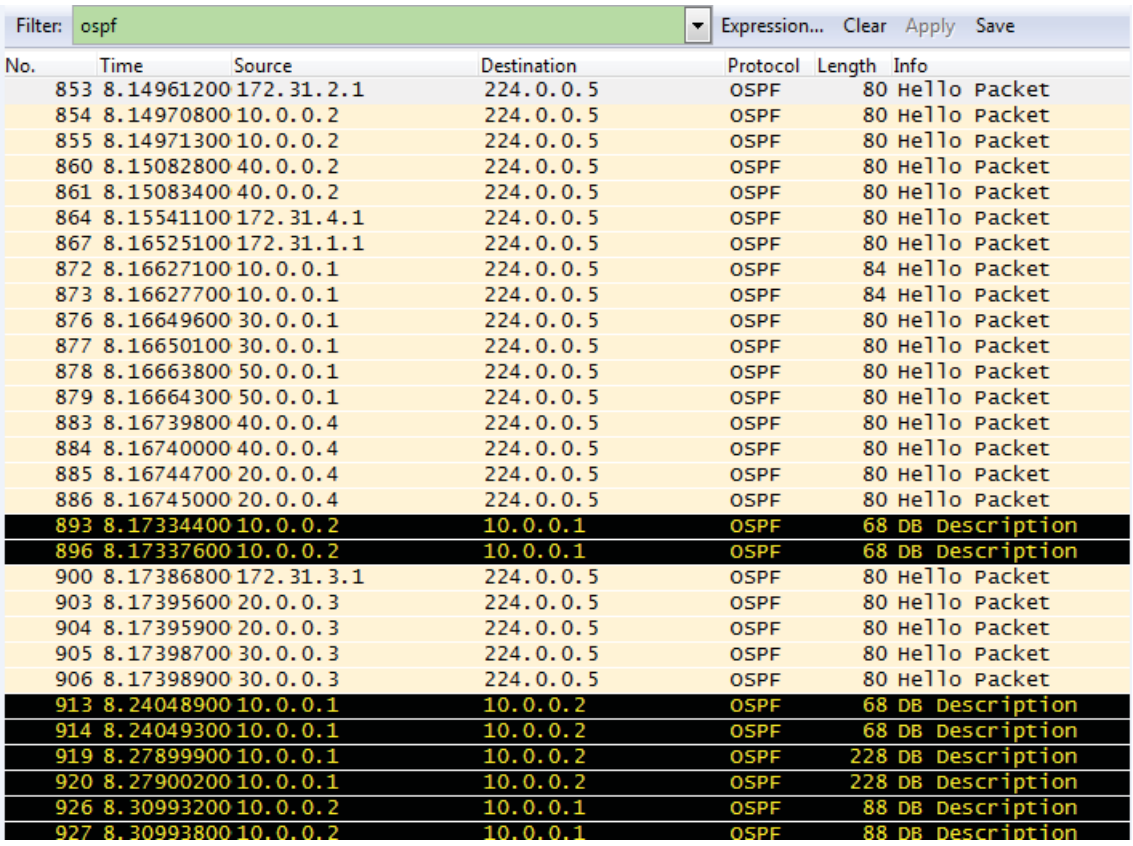

Fig. 4.6. Wireshark trace showing hello and DD packets from one of the virtual interfaces of VM in control plane

4.3 OpenFlow based OSPF environment

In order to test the proposed idea, we emulate OpenFlow based OSPF network in Linux environment with the help of RouteFlow architecture. This is done by emulating control and data plane separately and later connecting both of them together with OpenFlow and RouteFlow.

4.3.1 Emulating control plane

The Control Plane formed by VMs are emulated in the Linux environment with the help of Quagga and LXC. We follow the same procedure as discussed in the Section 4.1 for creating VMs and loading Quagga routing suite in them. Additionally, the RF-Client application is copied into every VM. RF-Client code utilizes Netlink API [24], which is used for Inter Process Communication (IPC) between the Kernel and User space process. Here the Kernel processes refers to the ARP and Routing tables of the VM and the user space processes refers to the RF-Client application. Simply put together, RF-Client gathers the updates of the routing and ARP table via Netlink API. RF-Client informs these changes to the RF-Server application for further processing.

Once the VMs of Control Plane are up and running, its interfaces are connected to an OVS. The OVS has an ability to support OpenFlow protocol and behaves as an OpenFlow switch. Every OpenFlow switch identifies itself with a special ID called Datapath ID (DPID). So, OVS is also given a DPID and is connected to the OpenFlow controller POX. This controller listens on a port 6633 for incoming connections (from OF switches). The controller and OVS exchange initial OpenFlow Hello messages and establish a connection. Now, the controller is responsible for forwarding routing protocol messages from OSPF like Hello and LSA between the interfaces of VMs. It is also responsible for forwarding packets between the data and control plane.

The whole process of starting VMs of control plane, loading RF-Client application and connecting to the controller is automated using a shell (bash) script. This script is also used in stopping the VMs and clearing the routes that are inserted by ospfd. The script can be found in the Appendix (Script 1).

4.3.2 Emulating data plane

The Data Plane in the OpenFlow based OSPF environment is formed by a network of OpenFlow (OF) switches. Such network can be emulated using Mininet. Mininet is a network emulator, which is capable of emulating any number of OF switches depending on the system resources [12]. It uses lightweight virtualization technique to make a single system look like a complete network. The behavior of such network elements is similar to their hardware prototypes.

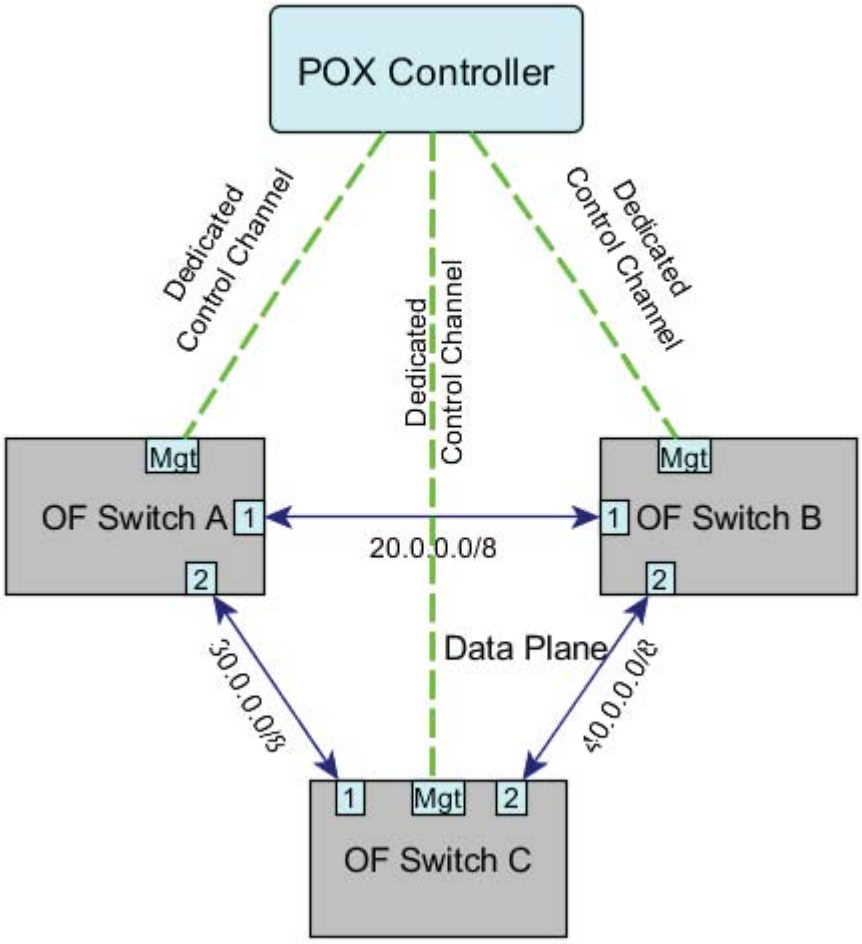

Fig. 4.7. Data plane emulated with the help of mininet

Mininet is entirely written in Python language. Using Python scripts we can emulate custom topologies of an OF network. A sample script is provided in the Appendix (Script 2) which creates four OF switches and connects them to an OpenFlow controller POX. All the necessary OF switches and links between them are created using node and net modules of Mininet. The links used to connect the OF switches are virtual Ethernet pairs, which live in Linux kernel. The created OF network is

connected to a controller using RemoteController class from node module. We connect the OF network to the same controller that we used for connecting the virtual interfaces of the VMs to OVS. With this, the emulation of Data Plane is completed with the help of Mininet.

4.4 Connecting control and data plane

This section would brief the procedure followed in connecting the Control and Data Plane together with the help of OpenFlow and RouteFlow architecture.

The POX controller used in this study is loaded with an extra module from RouteFlow architecture called RF-Proxy and is parallely started. It acts as a direct channel between the Data Plane and virtual environment (Control Plane). This eliminates the need to pass through RF-Server and RF-Client. RF-Proxy can also control the behavior of the OVS apart from controller itself. The responsibilities of RF-Proxy are: to notify RF-Server about the network events such as switch joining or leaving the network, informing RF-Server about the ports of the OF switch and taking instructions from RF-Server to configure OF switches with OpenFlow rules. It also sends packets to Data and Control Plane, with the help of OVS. The packets that need to be delivered to the Control Plane from Data Plane are received by RF-proxy through the event (packet-in). This packet arrives with the information of sender at the OVS. The packet is then stored in a buffer and an event of packet entry is sent to the RF-Server.

RF-Server is the core component of RouteFlow architecture. It acts as a mediator between RF-Client and RF-Proxy. It collects the routing updates from RF-Client and adapts this to specified routing control logic. It then instructs the RF-Proxy to install the OpenFlow rules in the OF switches in Data Plane, based on the routing control logic. RF-Server is also responsible for mapping between ports of OF switches in Data Plane to the virtual interfaces of the VMs in Control Plane. For example,

an OSPF LSU packet destined to a particular interface of a VM arrives at the OVS. POX sees this packet and raises an OFPT_PACKET_IN event listened by the RF-Proxy. RF-Proxy informs this event to RF-server for solving the mapping between VM interface and the port of the OVS. RF-Server solves this mapping by looking at the .CSV file in which it holds mapping information and then instructs RF-Proxy to send that packet to respective port as an OFPT_PACKET_OUT message.

The Control Plane VMs, controller (POX) and all of the RouteFlow components are started first. Then the Data Plane is connected to the emulated virtual environment with the help of OpenFlow and RouteFlow as described above.

5. TESTING METHODOLOGY AND RESULTS

In order to test the proposed idea of link failure detection using OpenFlow protocol in OSPF network, we simulate the link failures in Data Plane. The link failures are simulated using Mininet commands and Python scripts. A sample script is provided in the Appendix (Script 3) which takes down link between switches A and B in the Fig 4.7. But, the current version of RouteFlow does not handle link failures of Data Plane. So in RF-Proxy module, a code has been added to handle the PortStatus (generated when the link/port goes up/down of an OF switch) events of the Data Plane. As soon as RF-Proxy gets an OFPT_PORT_STATUS message from the respective switch ports, it instructs the RF-Client daemon to shutdown the mapped interface of the respective VM in Control Plane. This way the link failure detection time is reduced without waiting for the RouterDeadInterval to be fired.

5.1 Testing methodology

All the experiments were performed on an Ubuntu 12.04 (LTS) server which has 4GB of RAM, 80GB of QEMU Hard-disk and 4 CPU cores. We perform the experiments on traditional OSPF and repeat the same set of experiments for OpenFlow based OSPF. In both the cases the time taken to detect the link failure are determined.

The topology used for this experiment is shown in the Fig 5.1. The figure shows three Routers R1, R2 and R3 and the networks they handle. The emulation of all these routers and connections between them are created as per Section 4.1. First, we perform the experiment on the traditional OSPF. The HelloInterval and RouterDeadInterval time are set for 1 and 4 seconds respectively in the ospfd configuration file.

The ospfd daemon is started and all the routers start sending the Hello packets out of their interfaces. The neighbors are discovered dynamically by the Hello protocol in OSPF and adjacencies are formed. After few seconds routers exchange DD packets, LSU packets and finally converge. We open wireshark to capture OSPF packets by applying an OSPF filter. At this point, we simulate the link failure situation by deleting the link between R1s virtual interface eth1 and port of the OVS Bridge connected to it. Also, the time when the link failure occurred is noted. Now, we wait for the first LSU message to arrive at the wireshark after the link failure. The time when the LSU arrives is noted and the time difference between both the times is the time taken by OSPF to detect the link failure.

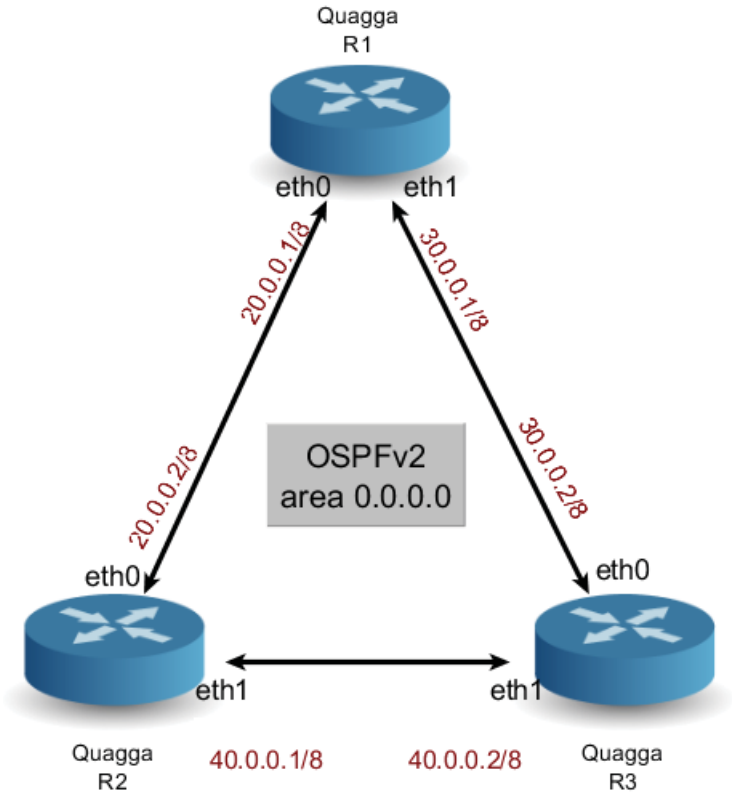

Fig. 5.1. Topology used in the experiment

The same experiment is conducted with OpenFlow based OSPF network. The three routers are now part of Control Plane and the Data Plane is emulated with three OF switches. Both the Control and Data Plane are connected together according to the procedure in Section 4.2. After the initial convergence in the Control Plane, the Flow rules are installed in the OpenFlow switches with the help of RouteFlow. At this point, we simulate the link failure between switches S5 and S6 with Mininet command link S5 S6 down. This commands breaks the link between both the switches and immediately OFPT_PORT_STATUS messages are received at the POX. These OF messages are logged along with their timestamps. Now, RF-Proxy sees this event and signals RF-Client to shutdown the mapped interface of the VM in the Control Plane. With this, OSPF is forced to send LSU packet out of its interface immediately and the time it generates this message is captured in the wireshark. Once again the time is measured for the link failure to be detected.

The experiment is repeated five times and the average time to detect the link failure is noted in both the cases. Also, the HelloInterval and RouterDeadInterval time is increased sequentially from 1 second to 10 seconds and 4 seconds to 40 seconds respectively.

The same experiment is performed with four and five routers with 5 and 7 links between them. The same steps are followed as mentioned in the above experiment and the topology used for these experiments are provided in the Fig 5.2 and Fig 5.3 respectively.

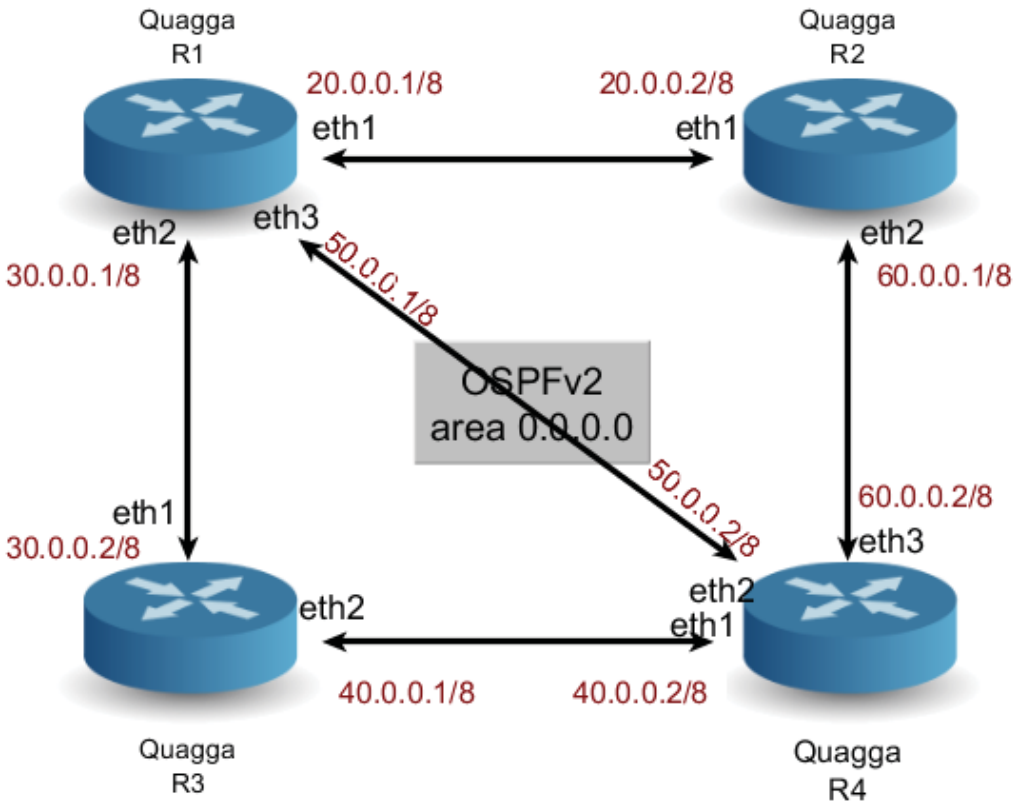

Fig. 5.2. 4 node topology used in experiment

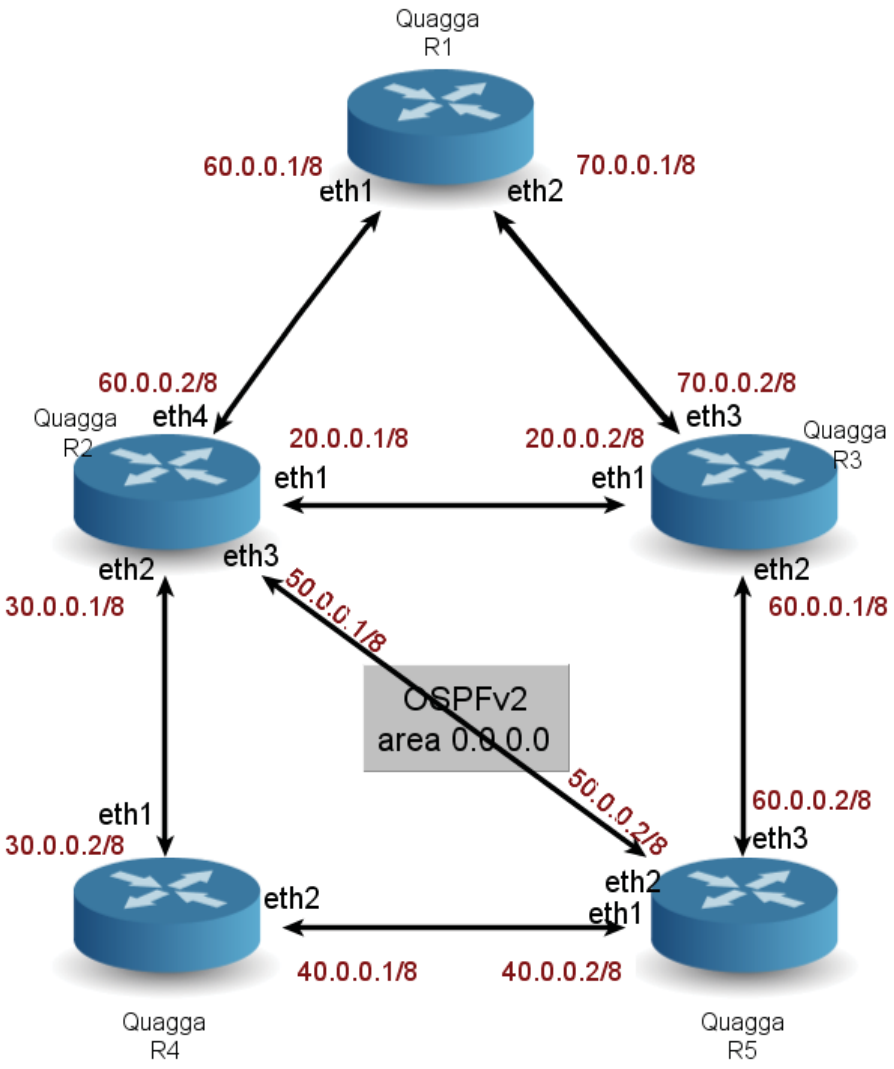

Fig. 5.3. 5 node topology used in experiment

5.2 Results

With the information from above experiments, graphs are plotted to compare the link failure detection time between traditional OSPF and OpenFlow based OSPF.

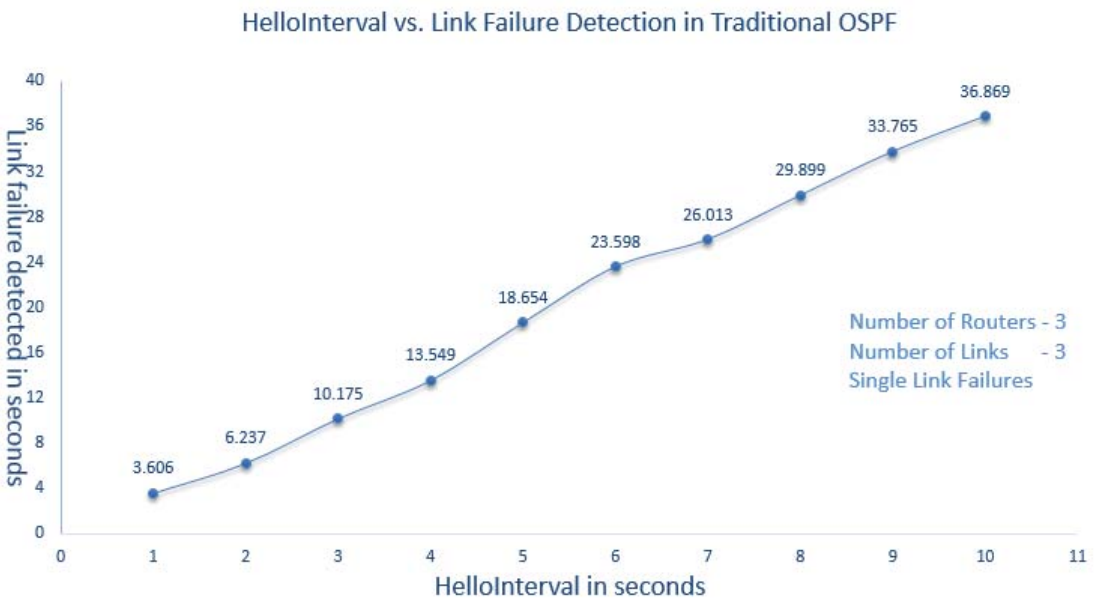

Fig. 5.4. Traditional OSPF

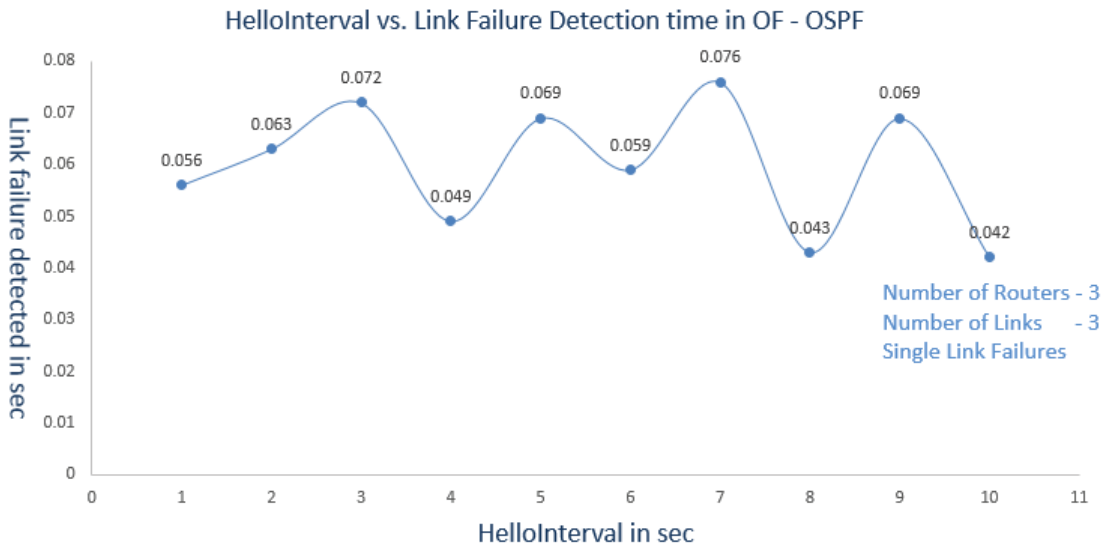

Fig. 5.5. OpenFlow based OSPF

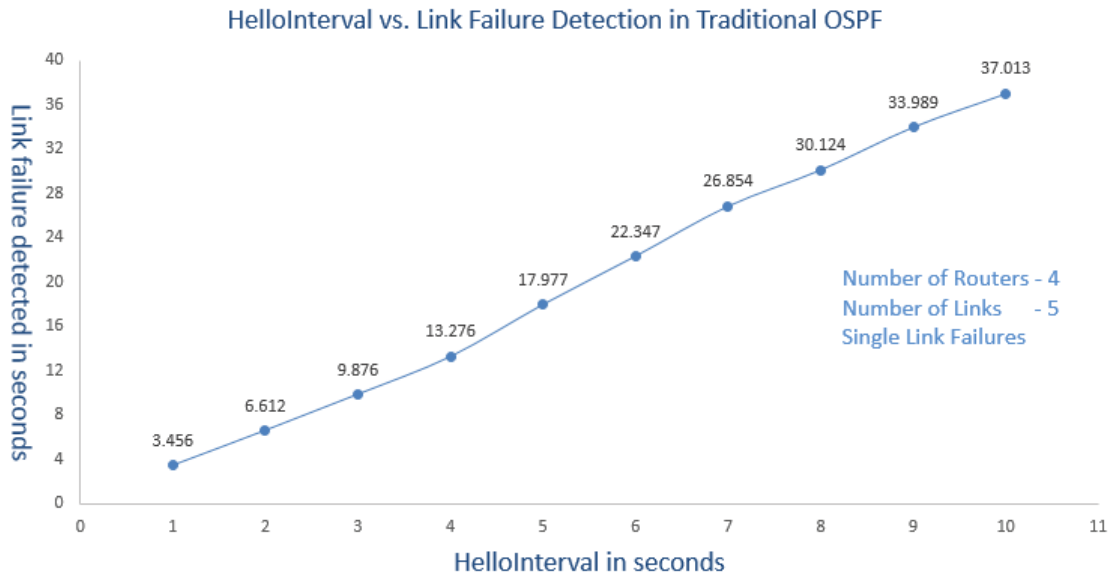

Fig. 5.6. Traditional OSPF

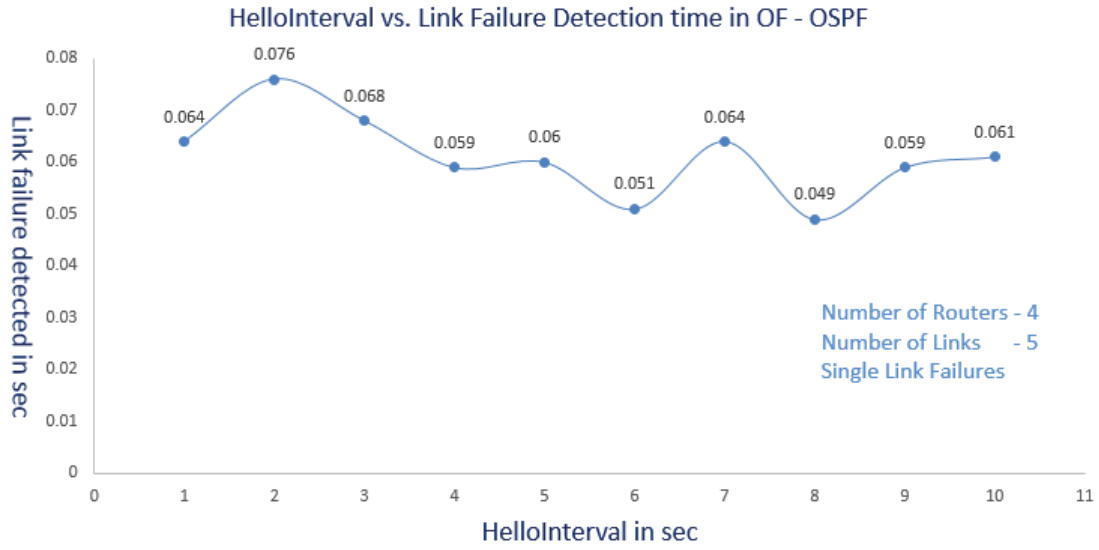

Fig. 5.7. OpenFlow based OSPF

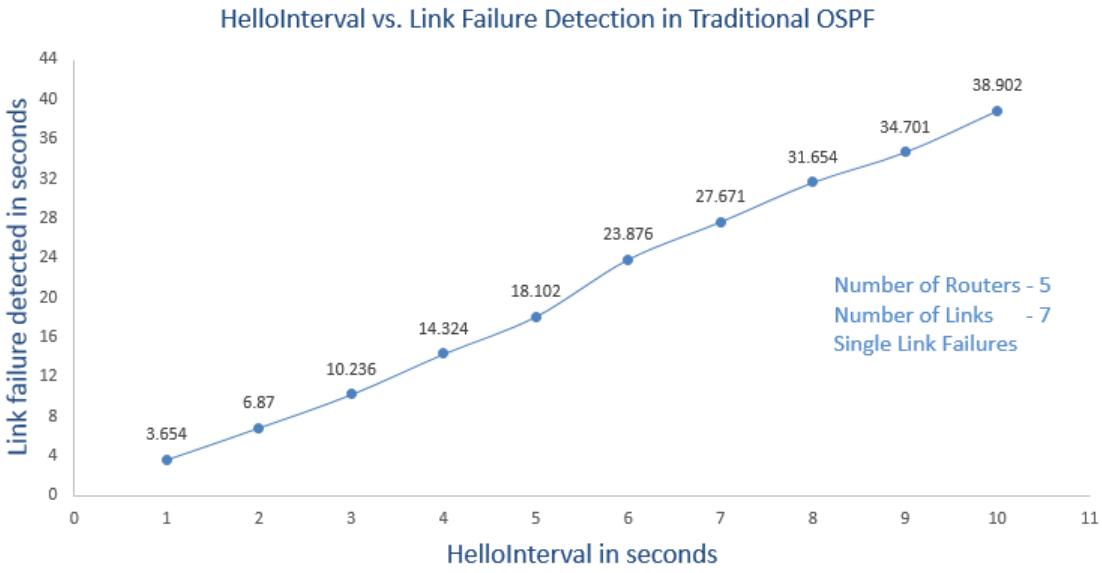

Fig. 5.8. Traditional OSPF

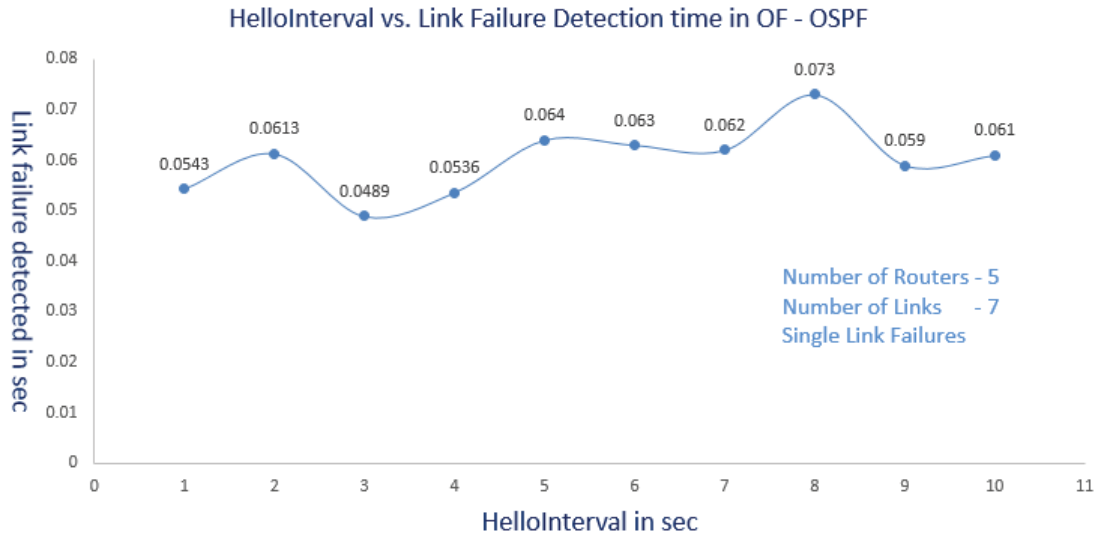

Fig. 5.9. OpenFlow based OSPF

The X-axis in graphs corresponds to HelloInterval time in traditional OSPF and OF based OSPF network and Y-axis is the time taken to detect the link failure.

Fig 5.4 and 5.5 shows the result from the first experiment that was performed with three Routers and three links. As the HelloInterval duration was increased from 1 second to 10 seconds, the time taken by the traditional OSPF to detect the failure is increased linearly (refer to Fig 5.4). Whereas in the proposed approach of OF based OSPF (refer to Fig 5.5) there is a significant improvement in detecting the link failures. The variation of time that is observed in the Fig 5.5 is due to the controller receiving OFPT_PORT_STATUS message at different instance of times. This is because OVS is software switch and the processing of OpenFlow packets is not done at line rate. Also, the processing delays of RF-Proxy, RF-Client and POX controller were also added to the failure detection time which resulted in the variation of time.

Similarly, Fig 5.6 and 5.7 shows the results from the experiment done with four routers and five links. Same set of single link failures were simulated and the results were close to what was observed in the previous experiment. And the results from the last experiment that was performed with five routers and seven links can be seen in the Fig 5.8 and 5.9.

From the graphs it is evident that the time taken by OpenFlow based OSPF network to detect the link failures is far less than the time taken by the traditional OSPF.

6. FUTURE SCOPE AND CONCLUSION

The approach proposed in this thesis has reduced the link failure detection and thereby reducing the overall time of the network convergence. But, there are few disadvantages that can be rectified in the future. Though RouteFlow architecture provided routing capability to OpenFlow network, it has lot of components and signaling involved. As the number of routers and links between them are scaled, the amount of processing time introduced by these components might increase the time to detect the link failure further. Also, controller POX is a single point of failure in the network. If it fails, then the impact would be on the entire network. This is because OpenFlow 1.0 does not support multiple controllers for the same network. But in future with OpenFlow 1.2 and later, a network can have multiple controllers. In this way, even if one of the controllers fail, the backup controller would still be able to see all the network elements.

In future, with the help of OpenFlow enabled routers, the need to use RouteFlow architecture can be totally reduced. This way, the delays involved in processing time can be greatly reduced. Moreover with OpenFlow enabled routers, the controller can take care of routing services of the OpenFlow network directly. This gives the network administrator greater flexibility to manage and program the networks Control Plane according to the varying needs.

The use of SDN and OpenFlow in the industry is growing rapidly. Their services are used by technology giants like Google in their Data centers across the world. In a presentation at Open Networking Summit, Google presented the advantages of using OpenFlow protocol, which helped them to have improved manageability over

the network. With the help of OpenFlow based SDN architecture the network can be more programmable and manageable by decoupling network Control and Data plane.

LIST OF REFERENCES

LIST OF REFERENCES

[1] T. Narten, "Internet routing," in *Symposium proceedings on Communications architectures & protocols*, SIGCOMM '89, (New York, NY, USA), pp. 271–282, ACM, 1989.

[2] "Routing." http://en.wikipedia.org/w/index.php?title=Routingoldid=579584360. Date last accessed: November 9, 2013.

[3] J. Moy, "Ospf version 2," 1998.

[4] C. E. Leiserson, R. L. Rivest, C. Stein, and T. H. Cormen, "Introduction to algorithms," pp. 588–591, 2001.

[5] E. W. Dijkstra, "A note on two problems in connexion with graphs," *NUMERISCHE MATHEMATIK*, vol. 1, no. 1, pp. 269–271, 1959.

[6] ONF, "Software-defined networking architecture: The new norm for netwoks." http:www.opennetworking.org/ images/stories/downloads/sdn-resources/white-papers/wp-sdn-newnorm.pdf. Date last accessed: July 19, 2013.

[7] G. Scheets, M. Parperis, and R. Singh, "Voice over the internet: A tutorial discussing problems and solutions associated with alternative transport," *Communications Surveys Tutorials, IEEE*, vol. 6, no. 2, pp. 1–10, 2004.

[8] A. Markopoulou, G. Iannaccone, S. Bhattacharyya, C.-N. Chuah, Y. Ganjali, and C. Diot, "Characterization of failures in an operational ip backbone network," *Networking, IEEE/ACM Transactions on*, vol. 16, no. 4, pp. 749–762, 2008.

[9] C. Alaettinoglu, I. Draft, H. Yu, and V. Jacobson, "Towards milli-second igp convergence," 2000.

[10] A. Basu and J. Riecke, "Stabilitu issues in ospf routing," *Computer Communication Review*, vol. 31, no. 4, 2001.

[11] M. Goyal, K. Ramakrishnan, and W. chi Feng, "Achieving faster failure detection in ospf networks," in *Communications, 2003. ICC '03. IEEE International Conference on*, vol. 1, pp. 296–300 vol.1, 2003.

[12] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: rapid prototyping for software-defined networks," in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, Hotnets-IX, pp. 19:1–19:6, ACM, 2010.

[13] ONF, "Openflow switch specification 1.0." http:www.opennetworking.org /images/stories/downloads/sdn-resources/white-papers/wp-sdn-newnorm.pdf. Date last accessed: July 19, 2013.

[14] Horman, "Open vswitch." http://openvswitch.org/slides/openvswitch.en-2.pdf. Date last accessed: September 4, 2013.

[15] M. Murphy, "Pox controller." https://github.com/noxrepo/pox. Date last accessed: August 20, 2013.

[16] "Wireshark: Network protocol analyzer." http://en.wikipedia.org/Wireshark. Date last accessed: September 26, 2013.

[17] "Cbench." http://www.openflowhub.org/display/floodlightcontroller/Cbench. Date last accessed: October 28, 2013.

[18] M. R. Nascimento, C. E. Rothenberg, M. R. Salvador, C. N. A. Corrêa, S. C. de Lucena, and M. F. Magalhães, "Virtual routers as a service: the routeflow approach leveraging software-defined networks," in *Proceedings of the 6th International Conference on Future Internet Technologies*, CFI '11, (New York, NY, USA), pp. 34–37, ACM, 2011.

[19] M. Handley, O. Hodson, and E. Kohler, "Xorp: An open platform for network research," in *ACM SIGCOMM Computer Communication Review*, pp. 53–57, 2002.

[20] L. Surhone, M. Tennoe, and S. Henssonow, *Bird Internet Routing Daemon*. VDM Publishing, 2011.

[21] "Lxc." http://en.wikipedia.org/w/index.php?title=LXColdid=580678290. Date last accessed: November 4, 2013.

[22] "Lxc-linux containers." http://lxc.sourceforge.net/man/lxc.html. Date last accessed: November 4, 2013.

[23] "Bridging (networking)." http://en.wikipedia.org/wiki/Bridging_(networking)/. Date last accessed: November 5, 2013.

[24] "Netlink." http://man7.org/linux/man-pages/man7/netlink.7.html. Date last accessed: October 28, 2013.

APPENDIX

APPENDIX

Script 1

```
#!/usr/bin/python

"""
This code links two switches S1 ----- S2 and then hooks it to a
remote controller POX.
This code is used in calculating the time taken by
OFPT_PORT_STATUS message to be detected by controller
"""

from mininet.log import setLogLevel, info
from mininet.net import Mininet
from mininet.node import RemoteController
from mininet.cli import CLI
from time import sleep
from mininet.topo import Topo
import os

class MyTopo( Topo ):
    "Simple topology example."

    def __init__( self ):
        "Create custom topo."
```

```
        # Initialize topology
        Topo.__init__( self )

        # Add switche
        leftSwitch = self.addSwitch( 's3' )
        rightSwitch = self.addSwitch( 's4' )

        # Add links
        self.addLink( leftSwitch, rightSwitch )

topo = MyTopo()
net = Mininet(topo=topo, controller=lambda name:
RemoteController( name, ip='127.0.0.1' ))

# Start wirehark
os.system("wireshark &")

net.start()

s3, s4 = net.get('s3', 's4')
comm = "s3 s4 down"
arg = comm.split()
sleep(15)
net.configLinkStatus( *arg )
os.system("/home/ospf/Bashscripts/dite.sh
 > /home/ospf/timelogs/mininetlinkdown.txt")
CLI(net)
net.stop()
```

```
#Kill wireshark
os.system("kill $(ps -ef | grep '[w]ireshark' | awk '{print $2}')")

LXC commands
/*
Type the following commands on a bash shell
1. To create a LXC container:
     lxc-create -n R1 -t ubuntu
where -n is the name of the container and -t is the template
to be used for creating container.
2. To clone a container:
     lxc-clone -o R1 -n R2
where -o is the container to be cloned to create a new
container named R2
3. To start a contiaer:
     lxc-start -n R1 -d
where -d will start the container as a daemon and user can log into
container with "lxc-console -n R1" command
*/

LXC network config file
/* Network config file used to boot a container */


lxc.network.type=veth
lxc.network.link=lxcbr0
lxc.network.flags=up
lxc.network.hwaddr = 00:16:3e:10:f5:54
lxc.utsname = R1

lxc.network.type=veth
```

```
lxc.network.link=lxcbr0
lxc.network.flags=up
lxc.network.hwaddr=00:00:00:00:00:01
lxc.network.veth.pair=R1.1

lxc.devttydir = lxc
lxc.tty = 4
lxc.pts = 1024
lxc.rootfs = /var/lib/lxc/R1/rootfs
lxc.mount  = /var/lib/lxc/R1/fstab
lxc.arch = amd64
lxc.cap.drop = sys_module mac_admin
lxc.pivotdir = lxc_putold

lxc.cgroup.devices.deny = a
# Allow any mknod (but not using the node)
lxc.cgroup.devices.allow = c *:* m
lxc.cgroup.devices.allow = b *:* m
# /dev/null and zero
lxc.cgroup.devices.allow = c 1:3 rwm
lxc.cgroup.devices.allow = c 1:5 rwm
# consoles
lxc.cgroup.devices.allow = c 5:1 rwm
lxc.cgroup.devices.allow = c 5:0 rwm
#lxc.cgroup.devices.allow = c 4:0 rwm
#lxc.cgroup.devices.allow = c 4:1 rwm
# /dev/{,u}random
lxc.cgroup.devices.allow = c 1:9 rwm
lxc.cgroup.devices.allow = c 1:8 rwm
```

```
lxc.cgroup.devices.allow = c 136:* rwm
lxc.cgroup.devices.allow = c 5:2 rwm
# rtc
lxc.cgroup.devices.allow = c 254:0 rwm
#fuse
lxc.cgroup.devices.allow = c 10:229 rwm
#tun
lxc.cgroup.devices.allow = c 10:200 rwm
#full
lxc.cgroup.devices.allow = c 1:7 rwm
#hpet
lxc.cgroup.devices.allow = c 10:228 rwm
#kvm
lxc.cgroup.devices.allow = c 10:232 rwm
```

OVS commands

```
/* These commands are used to create OVS bridge
   and ports in the bridge. OVS should be installed
  prior to this */
```

1. ovs-vsctl add-br cd0

Createa a bridge with name cd0

2. ovs-vsctl add-port cd0 R1.1

Adds a port to the bridge and connects an
interface of VM to the port

3. ovs-vsctl set Bridge cd0 other-config:datapath-id=1245678

Sets the DPID of the switch (OpenFlow specific parameter)

```
4. ovs-vsctl set-controller cd0 tcp:127.0.0.1:6633
Attaches OVS to an OpenFlow controller

5. ovs-ofctl show cd0
Prints the information of OF switches and its flow table and ports

Script 2
/* This script starts ospfd, Vitual Machines of Control Plane,
starts the controller POX along with RF-Proxy,
Starts RF-Server, loads RF-Clinet application to all VM's and
waits for the DataPlane connections. This script emulates
3 VM's and coonects it OVS*/

echo_bold "-> Configuring the virtual machines..."
# Create the rfclient dir
mkdir /var/lib/lxc/rfvmA/rootfs/opt/rfclient
mkdir /var/lib/lxc/rfvmB/rootfs/opt/rfclient
mkdir /var/lib/lxc/rfvmC/rootfs/opt/rfclient

# Copy the rfclient executable
cp build/rfclient /var/lib/lxc/rfvmA/rootfs/opt/rfclient/rfclient
cp build/rfclient /var/lib/lxc/rfvmB/rootfs/opt/rfclient/rfclient
cp build/rfclient /var/lib/lxc/rfvmC/rootfs/opt/rfclient/rfclient


# We sleep for a few seconds to wait for the interfaces to go up
echo "#!/bin/sh" > /var/lib/lxc/rfvmA/rootfs/root/run_rfclient.sh
echo "sleep 3" >> /var/lib/lxc/rfvmA/rootfs/root/run_rfclient.sh
echo "/etc/init.d/quagga start" >> /var/lib/lxc/rfvmA/rootfs/
root/run_rfclient.sh
```

```
echo "/opt/rfclient/rfclient > /var/log/rfclient.log" >> /var/lib/
lxc/rfvmA/rootfs/root/run_rfclient.sh

echo "#!/bin/sh" > /var/lib/lxc/rfvmB/rootfs/root/run_rfclient.sh
echo "sleep 3" >> /var/lib/lxc/rfvmB/rootfs/root/run_rfclient.sh
echo "/etc/init.d/quagga start" >> /var/lib/lxc/rfvmB/rootfs/
root/run_rfclient.sh
echo "/opt/rfclient/rfclient > /var/log/rfclient.log" >> /var/lib/
lxc/rfvmB/rootfs/root/run_rfclient.sh

echo "#!/bin/sh" > /var/lib/lxc/rfvmC/rootfs/root/run_rfclient.sh
echo "sleep 3" >> /var/lib/lxc/rfvmC/rootfs/root/run_rfclient.sh
echo "/etc/init.d/quagga start" >> /var/lib/lxc/rfvmC/rootfs/
root/run_rfclient.sh
echo "/opt/rfclient/rfclient > /var/log/rfclient.log" >> /var/lib/
lxc/rfvmC/rootfs/root/run_rfclient.sh

chmod +x /var/lib/lxc/rfvmA/rootfs/root/run_rfclient.sh
chmod +x /var/lib/lxc/rfvmB/rootfs/root/run_rfclient.sh
chmod +x /var/lib/lxc/rfvmC/rootfs/root/run_rfclient.sh

echo_bold "-> Starting the virtual machines..."
lxc-start -n rfvmA -d
lxc-start -n rfvmB -d
lxc-start -n rfvmC -d

echo_bold "-> Starting the controller and RFPRoxy..."
cd pox
./pox.py log.level --=INFO topology openflow.topology
```

```
openflow.discovery rfproxy rfstats &
cd -
wait_port_listen $CONTROLLER_PORT

echo_bold "-> Starting RFServer..."
./rfserver/rfserver.py rftest/rf3config.csv &

echo_bold "-> Starting the control plane network (dp0 VS)..."
ovs-vsctl add-br dp0
ovs-vsctl add-port dp0 rfvmA.1
ovs-vsctl add-port dp0 rfvmA.2
ovs-vsctl add-port dp0 rfvmA.3
ovs-vsctl add-port dp0 rfvmB.1
ovs-vsctl add-port dp0 rfvmB.2
ovs-vsctl add-port dp0 rfvmB.3
ovs-vsctl add-port dp0 rfvmC.1
ovs-vsctl add-port dp0 rfvmC.2
ovs-vsctl add-port dp0 rfvmC.3
ovs-vsctl set Bridge dp0 other-config:datapath-id=7266767372667673
ovs-vsctl set-controller dp0 tcp:127.0.0.1:$CONTROLLER_PORT
```

Script 3

```
"""  Script of Data Plane "rf3" topology
Three OF switches connected in mesh topology
and a each switch serves a host in different subnet.
This is connected to POX which is already running.
Controller must be started first.
```

```
        h1 --- sA
               |  \
               |   \
               |    \
               |     \
       h2 --- sB ---- sC --- h3
"""

from mininet.topo import Topo

class rftest2(Topo):

   def __init__( self, enable_all = True ):
        "Create custom topo."

        Topo.__init__( self )

        h1 = self.addHost("h1",
                          ip="172.31.1.100/24",
                          defaultRoute="gw 172.31.1.1")

        h2 = self.addHost("h2",
                          ip="172.31.2.100/24",
                          defaultRoute="gw 172.31.2.1")

        h3 = self.addHost("h3",
                          ip="172.31.3.100/24",
                          defaultRoute="gw 172.31.3.1")
```

```
        sA = self.addSwitch("s5")
        sB = self.addSwitch("s6")
        sC = self.addSwitch("s7")

        self.addLink(h1, sA)
        self.addLink(h2, sB)
        self.addLink(h3, sC)

        self.addLink(sA, sB)
        self.addLink(sC, sA)
        self.addLink(sB, sC)

topos = { 'rftest2': ( lambda: rftest2() ) }
```