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 Shreya Pandita

Entitled

OpenFlow Based Load Balancing And Proposed Theory For Integration In VoIP Network

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

OPENFLOW BASED LOAD BALANCING AND PROPOSED THEORY FOR INTEGRATION IN VOIP NETWORK

A Thesis

Submitted to the Faculty

of

Purdue University

by

Shreya Pandita

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 family for their eternal love and care

ACKNOWLEDGEMENTS

To begin with, I would like to express my sincere gratitude to my advisor Dr. Dongsoo Stephen Kim for his continuous support, guidance and patience while I was completing my thesis. I would also like to thank my friends Sumit Raina, Santhan Pamulapati and Amit Saini for being great critics and lending me a helpful hand whenever I was stuck.

Secondly, I would like to thank InCNTRE for giving me an opportunity as an Intern, to learn about SDN/OpenFlow which is a cutting edge technology. Special thanks to my committee members Dr. Brian King and Dr. Maher Rizkalla for their valuable time.

To conclude with, I would acknowledge Sherrie Tucker and Summer Layton for their time, efforts and advice on linguistics and formatting.

TABLE OF CONTENTS

Page

LIST OF FIGURES v

ABSTRACT vii

1 INTRODUCTION 1

1.1 OpenFlow background 2

1.1.1 OpenFlow header 6

1.1.2 OpenFlow message types 7

2 RELATED WORK 10

3 PROPOSED ARCHITECTURE 14

4 EXPERIMENTAL SETUP AND RESULTS 20

5 PROPOSED THEORY – INTERGRATION OF VOIP NETWORK AND OPENFLOW 29

5.1 SIP background 29

5.1.1 Network elements 30

5.1.2 Overview of operation 32

5.2 Insight to current scenario – PSTN-SIP interworking architecture . 37

5.3 PSTN-SIP architecture integrated with OpenFlow 40

6 FUTURE SCOPE AND CONCLUSION 48

LIST OF REFERENCES 49

APPENDIX 51

LIST OF FIGURES

Figure		Page
1.1	An example of a traditional network	3
1.2	Basic functional diagram of an OpenFlow switch and controller	3
1.3	An example of an OpenFlow network	5
1.4	Structure of a flow-entry table	6
1.5	Openflow header format	7
3.1	Round robin load balancing with OpenFlow	14
3.2	Load based load balancing with OpenFlow	16
3.3	Flowchart explaining various steps in load based algorithm	18
3.4	Flowchart explaining getLeastLoadedMember process	19
4.1	Architecture of floodlight controller	21
4.2	OpenFlow wireshark trace showing OF messages	23
4.3	Experiment setup	24
4.4	Received load by servers at an instance of time for fixed data transfer rate (round robin algorithm)	25
4.5	Received load by servers at an instance of time for random data transfer rate (round robin algorithm)	25
4.6	Received load by servers at an instance of time for fixed data transfer rate (load based algorithm)	26
4.7	Received load by servers at an instance of time for random data transfer rate (load based algorithm)	26
4.8	Average jitter variation with increase in number of clients (comparison between round robin and load based algorithm)	27
4.9	Average dropped count variation with increase in number of clients (comparison between round robin and load based algorithm)	27
5.1	Registrar offering location service to proxy server	31
5.2	A basic redirection scenario in SIP	31

Figure Page

5.3 SIP basic call flow 33
5.4 SIP invite body and SDP message enclosed 36
5.5 A PSTN and SIP network interworking 37
5.6 Functional description of SIP-PSTN gateway 38
5.7 SIP-PSTN architecture integrated with OpenFlow load balancing . . . 40
5.8 Event diagram for SIP-PSTN architecture integrated with OpenFlow load balancing [Contd..] 43
5.9 Event diagram for SIP-PSTN architecture integrated with OpenFlow load balancing 44
5.10 Redirection of call from a heavily loaded media gateway to least loaded gateway 47

ABSTRACT

Pandita, Shreya. M.S.E.C.E, Purdue University, December 2013. OpenFlow based load balancing and proposed theory for integration in VoIP network. Major Professor: Dongsoo Stephen Kim.

In today's internet world with such a high traffic, it becomes inevitable to have multiple servers representing a single logical server to share enormous load. A very common network configuration consists of multiple servers behind a load-balancer. The load balancer determines which server would service a clients request or incoming load from the client. Such a hardware is expensive, runs a fixed policy or algorithm and is a single point of failure. In this paper, we will implement and analyze an alternative load balancing architecture using OpenFlow. This architecture acquires flexibility in policy, costs less and has the potential to be more robust. This paper also discusses potential usage of OpenFlow based load balancing for media gateway selection in SIP-PSTN networks to improve VoIP performance.

1. INTRODUCTION

With fast growth of Internet users and applications, the underlying networks need the ability to scale performance to handle enormous volumes of client requests and data without creating unwanted delays. Load balancing is a promising solution for increasing network scalability and service availability. When we think of load balancing we imagine expensive devices which can be placed in the network for distributing the load across these servers. Unfortunately, these devices become single point of failures in the network. In a heavily loaded network they can become bottle necks [1]. Also, there may be various scenarios where our network topology is so complicated that a single load balancer may not serve the purpose. Keeping these constraints in mind we thought OpenFlow protocol might be a solution to these problems. With OpenFlow protocol, load balancing becomes a network primitive as the network is programmed to load balance its load and thus there is no need for load balancers. The paper would explore how effectively we can perform load balancing using OpenFlow protocol and also extend a theory of its use in VoIP networks.

Recent years have witnessed a fast growth of Internet transferring voice. Network researchers have been continuously working on improving Voice over IP (VoIP) to meet public demands. But, VoIP services still experience large delay, jitter and degraded voice quality. VoIP uses Session Initiation Protocol (SIP) to perform its signaling. Introduction of SIP protocol has made a profound impact on VoIP performance. SIP media gateways are translation devices that convert digital media streams between dissimilar networks. If SIP media gateways would operate in heavily loaded conditions then it decreases the network throughput [2]. Hence, introducing load balancing can lead to improved throughput in VoIP performance.

1.1 OpenFlow background

OpenFlow is a communication protocol which implements software defined networking (SDN). It is the first standardized communication interface defined between the control and forwarding functionality of SDN architecture allowing us to manipulate the forwarding plane of network devices [3].

Commercial switches and routers do not provide an open software platform [4]. The vendors hide the devices internal exibility and also these internals differ from one vendor to another. Unfortunately, the researchers are left with no standard platform to conduct their ideas. Using OpenFlow, researchers can partition trac into production and research ows. They can control their own ows by deciding the routes and processing for their test traffic. This gives an opportunity to the researchers to try new protocols or create advanced applications like network firewall and load balancers. With a concept called network virtualization the production trac can be isolated from research traffic [5].

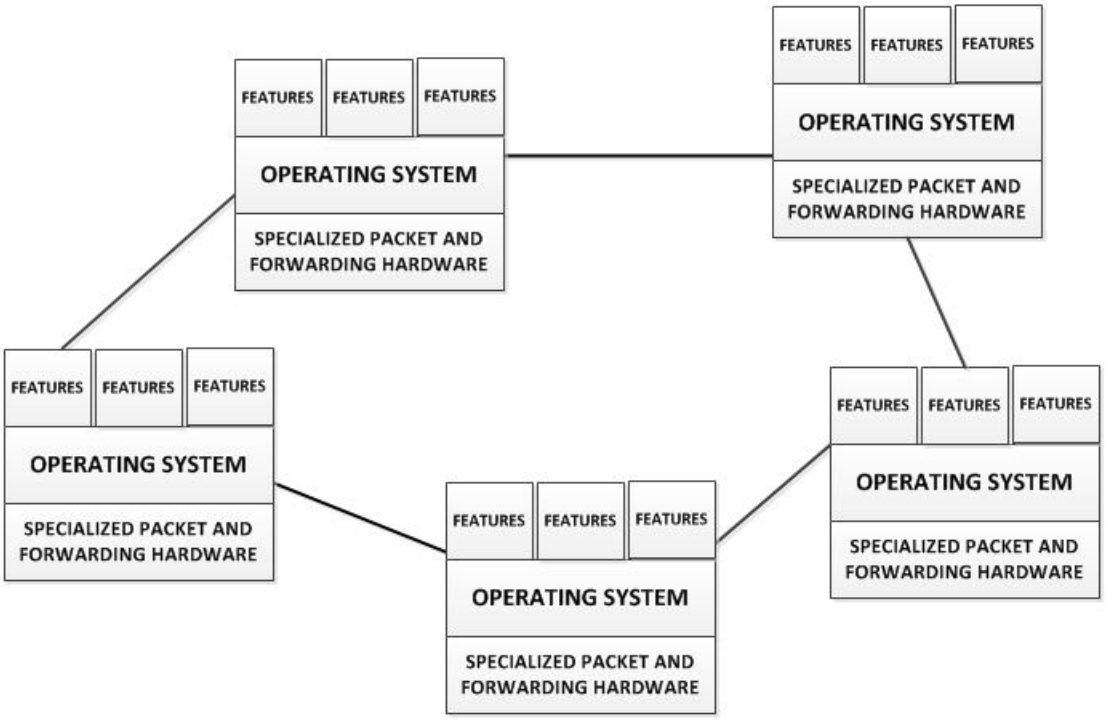

Fig. 1.1. An example of a traditional network

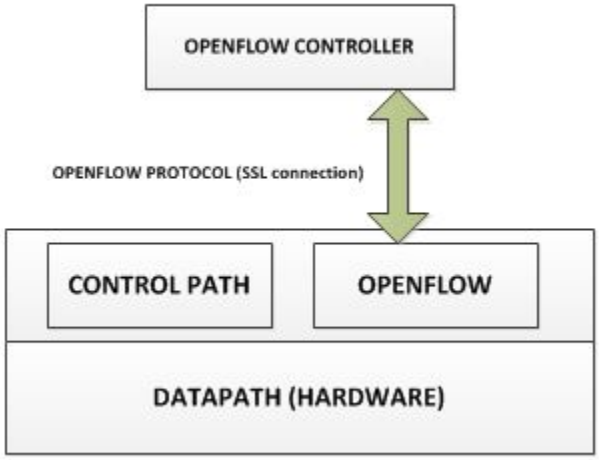

Fig. 1.2. Basic functional diagram of an OpenFlow switch and controller

In traditional network devices, the control processes and forwarding functionality resides in the device itself as shown in Fig 1.1. In an OpenFlow network device the data path (packet forwarding) and control path (high-level routing) of a network device are separated. The data path portion resides in the switch itself; a separate controller process makes high-level routing decisions. The switch and controller communicate using the OpenFlow protocol. The connection between switch and controller is Secure Sockets Layer (SSL) as shown in Fig 1.2. The control interface of the devices is connected to a Network Operating System (Nox, Floodlight etc.) via the secure channel as shown in Fig 1.2. This Network Operating System is called as a controller.

OpenFlow protocol defines a standardized API and communication method between this Network Operating System and OpenFlow process in a networking device. A traditional device has control process running inside; it is capable of building its own forwarding table. For e.g. a traditional L2 switch learns its hosts by broadcasting the incoming packets. These broadcasts help the switch to build its MAC table (host mac address, port connected to host). Further, incoming packets are sent directly to the destined hosts by doing a look-up in MAC table. Whereas, in an OpenFlow network as shown in Fig 1.3, the control process is running in a controller. The incoming packet that hits the device is encapsulated into an OpenFlow packet and sent to the controller. The controller would learn the host and insert a flow entry in the flow table. The packets that match the flow entry will not be sent to controller; instead it will follow the action specified in the matching flow entry. Forwarding tables in traditional L2/L3 network devices are replaced by flow tables in OpenFlow.

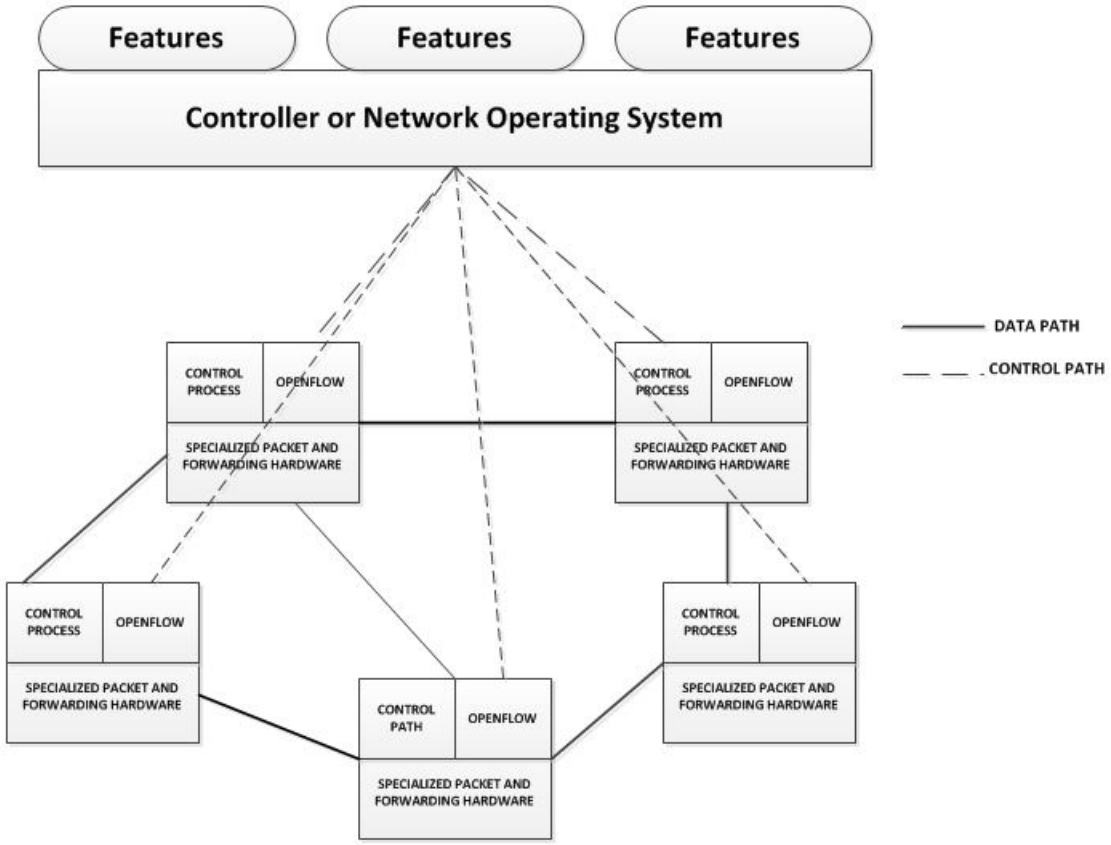

Fig. 1.3. An example of an OpenFlow network

OpenFlow Flow-Tables contains:

1. Header Fields: These are the fields against which a packet will be matched. There are 12 match fields defined in OpenFlow. These OpenFlow 12-Tuples are listed in Fig 1.4.

2. Counters: These are the statistics maintained by the switch. OpenFlow version 1.0, defines various types of counters (per-flow, per-queue, per-table, per-port).

3. Actions: They define how a packet should be treated (e.g. Forward, Drop, Modify). Refer Fig 1.4 for detailed set of actions as specified in OpenFlow switch

specifications 1.0. Some actions are grouped as mandatory and some as optional. For a switch to be considered as OpenFlow compliant, it must support all the mandatory actions [3].

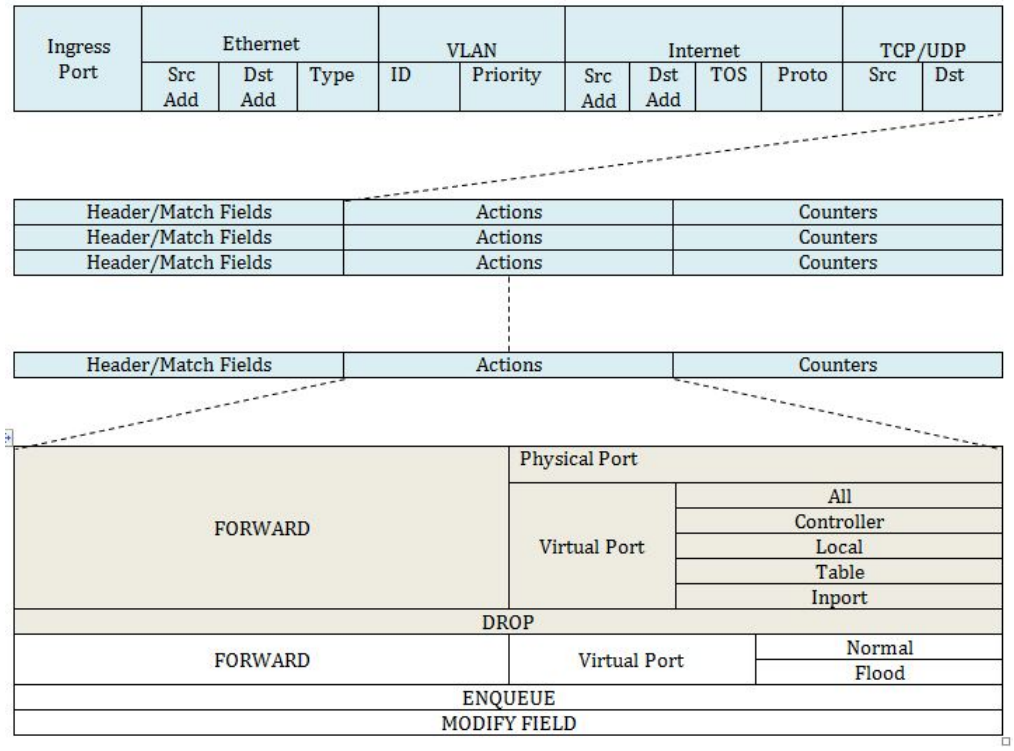

Fig. 1.4. Structure of a flow-entry table

1.1.1 OpenFlow header

OpenFlow header is 8 bytes and comprises of the following fields [6]:

1. Version: Version field specifies the type of the OpenFlow version is running of the device. The controller and switch need to negotiate on version at the connection establishment.

2. Type: This field specifies the type of message.

3. Length: This field denotes the length of the OpenFlow packet (includes header length as well).

4. XID: This field specifies the transaction ID associated with the respective packet. If a request is sent, the corresponding reply will have the same transaction ID.

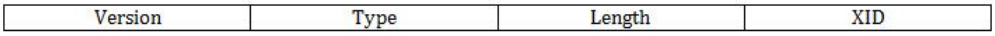

Fig. 1.5. Openflow header format

1.1.2 OpenFlow message types

Controller-to-Switch: Controller sends these messages to the switch and does not necessarily need a response [6]. Following are the different types the controller-to-switch messages [6]:

1. Features: Features request message is sent by the controller to the switch, when a Transport Layer Security (TLS) session is established between them. Features reply will be send in response to this request and it contains the capabilities, actions and what counters are supported by the switch.

2. Configuration: This is a group of messages which is used by the controller is able to query and set configuration parameters in the switch.

3. Modify-State: This comprises of a group of messages that are sent by the controller to maintain the state of an OpenFlow switch. Their prime purpose is to add,

delete or modify flows. They can also be used to set switch port state properties, for e.g. modifying the action specified for a flow or modifying the match fields of a flow.

4. Read State: This also comprises of a group of messages. It is used by the controller to collect switch statistics. There are various types of statistics defined under OpenFlow switch specifications.

5. Send Packet: This message is used by the controller to send packets out from any port on the switch.

6. Barrier: This is used by the controller to receive notifications for completed operations. When controller has sent a number of flow entries, it can then send a barrier message to ensure if flow entries were successfully installed or not.

Asynchronous messages: OpenFlow switches send asynchronous messages to the controller to notify about an event like packet arrival, error or state change. The four main asynchronous message types are described below [6]:

1. Packet-In: When an incoming packet does not match any flow entry then a packet-in event is triggered i.e. a packet-in message is sent to the controller. If the OpenFlow switch has enough memory to buffer packets, the packet-in message would contain only a fraction of this unmatched packet.

2. Flow Removed: When a flow is removed from the switch, the switch can send flow removed message to notify the controller.

3. Port Status: The OpenFlow switch will send port status message to the controller whenever the port state changes.

4. Error: The switch notifies the controller of any problems using an error message. For e.g. when switch fails a version negotiation, runs out of flow-table memory, receives an unexpected message type.

Symmetric messages: These messages are mainly used to by either switch or controller to identify the other party or to check the health of the connection [6]:

1. Hello: These messages are exchanged between the switch and controller for establishing the connection.

2. Echo: These are used to indicate the bandwidth, latency and aliveness of a controller-switch connection. These messages can be sent from either switch or controller side. An echo request must be replied with an echo reply message.

3. Vendor: These messages are used by the OpenFlow switch vendors to offer additional functionality to an OpenFlow switch.

2. RELATED WORK

The VoIP networks are spreading out and becoming prodigious; the reason being the sharp increase in users that are using VoIP over traditional telephony services and hence the VoIP networks have expanded. To meet the demands of the users there has to be an efficient usage of the network resources. Hence, network load balancing becomes a major concern. Network load balancing allows a network to become more scalable i.e. as and when traffic increases more servers can be added and incoming load can be distributed efficiently across them. Network load balancing can also provide high availability by detecting the failovers of any server and redistributing traffic among the ones that are active.

There are numerous ways of distributing load in a network. Round robin DNS is one such simple technique [7]. When the request is received by the DNS to resolve the domain name, it gives out one of the many mapped IP addresses from the server list. This redirects the request to one of the servers in a server farm. Subsequent requests from that client are sent to the same server. The major drawback of this scheme is that it does not take care of availability of the servers. Hence, the round robin DNS can be considered as a load distribution mechanism rather than a load balancing mechanism [8]. Profound works have been done in order to load balance SIP messages but not many considerable researches have been done to load balance the voice load itself.

Jenq-Shiou Leu et.al in [9] proposed a concept wherein SIP messages can be evenly distributed to all healthy SIP proxy servers without the necessity of an additional load balancer. To achieve this a Domain Name Resolution Load-Balancer (DNRLB) was

developed which periodically issues dummy SIP messages to SIP proxy servers in a SIP sever farm. These dummy SIP messages help in detecting health of the SIP proxy servers. The dummy messages are sent by DNS and the response it receives is an echo message sent by the SIP servers. DNS then picks the SIP server with least recently set SIP proxy server. DNS thus becomes an IP translator and also a service checker hence ensuring server availability. Probe waiting time increases with increase in the number of servers in the server farm. Also, an average failure rate decreases with increase in number of servers in the farm. One of the limitations in such an implementation is that these dummy messages might get sometimes lost in a congested network and hence DNRLB might not receive these dummy messages as expected. Resending of these dummy messages might increase the overall network load in congested conditions.

Hongbo Jiang et.al in [10] introduced and evaluated several novel algorithms for distributing SIP requests and voice load across SIP servers. Call Joint Shortest Queue (CJSQ) algorithm keeps a track of the number of calls which are allocated to servers and routes any new SIP calls to the server that has the least number of active calls. Transaction-Join-Shortest Queue (TJSQ) routes a new call to the server that has fewest number of transactions (Invite and Bye) instead of calls. This algorithm recognizes the fact that calls in SIP are composed of two main transactions INVITE and BYE. Completion of these transactions is tracked and estimates of server load are maintained. This algorithm provides better results since calls have variable lengths and server load estimates are better parameter for load balancing than mere requests. Transaction Least Work Left (TLWL) routes a new call to the server with least load. Load is calculated based on transaction costs. This cost calculation is based on the fact that Invites are more costly than Bye. The algorithms TJSQ and TLWL load balance SIP servers unlike CJSQ where only SIP requests are distributed across SIP servers. These algorithms might not be very effective in production networks since the voice load does not generally traverse through the SIP servers rather SIP servers

are used only for initial signaling after which the media stream goes directly between the end parties. In SIP-PSTN networks the media streams through the SIP media gateways for the protocol conversion. Consequently, load balancing becomes necessary in media gateways. This paper presents a proof of concept of how we can achieve load balancing in SIP media gateways.

We try to integrate OpenFlow protocol to the existing SIP-PSTN architecture to load balance on SIP media gateways. Some studies have already been done on load balancing using OpenFlow protocol. Nikhil Handigol et.al in [11] and Hardeep Uppal et.al in [12] proposed load balancing web traffic using OpenFlow. The former concentrates load balancing on an unstructured network using OpenFlow and tries to minimize the average response time. They developed an algorithm called Load balancing Over Unstructured Networks (LOBUS) which runs over the Plug-n-Serve Controller. Their controller views the topology and appropriately shares the load over the network devices. In the latter paper [12], three load balancing algorithms were running on Nox controller platform: Random, Round Robin and Load Based. In the random algorithm, for each new request forwarded to Nox, Nox randomly selects from a list of registered servers which server will handle the request.

In the round robin algorithm, for each new flow Nox rotates which server is the next server in line to service a request. Whereas in load based algorithm, servers wait for Nox to register and then report their current load on some scheduler similar to the Listener Pattern. Load is determined in terms of the number of pending requests in the servers queue. NOX listens in a separate thread on a UDP socket for heartbeats with reported loads from the servers and maintains an array with the current load of all servers. When a new request is received, it chooses the server with the current lowest load and increments that servers current load.

A conclusion was made that if OpenFlow switches are improved in hardware i.e. the flows are processed in hardware instead of software, then the flow processing time will be improved and we will be able to extract more benefits from OpenFlow.

Koerner et.al in [13] proposed load balancing of multiple services using OpenFlow. For load balancing multiple services, they slice or virtualize the network using Flow Visor and then use different Nox controllers to handle load of different services. Load balancing algorithm used is a simple round robin policy. Though, bandwidth attained was not high but usage of multiple OpenFlow controllers provided them with redundant network paths for reliability and also enhanced network scalability.

3. PROPOSED ARCHITECTURE

This paper uses floodlight controller as the OpenFlow controller platform to run the load balancing application. The floodlight controller has a very basic load balancer module already implemented on it. This module can be accessed via REST API. Following architecture is currently supported by floodlight load balancer application refer Fig 3.1.

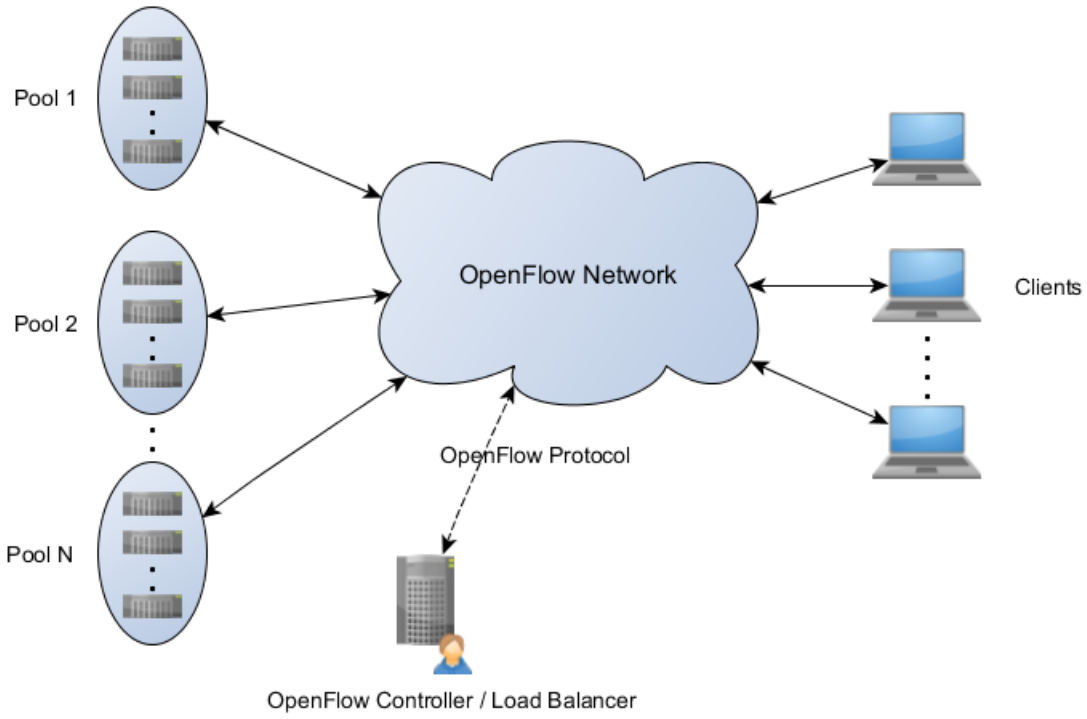

Fig. 3.1. Round robin load balancing with OpenFlow

A cluster of servers will form a pool. The servers are the members of the pool and each member has its own member ID. Each pool will be assigned a virtual IP and a pool ID. We can define multiple pools for load balancing different services like UDP, TCP and ICMP. Virtual IP will be known to the clients. Clients will send requests or data traffic to the virtual IP address instead of physical IP address of the servers. The controller will decide which server will handle the client connection. This decision is a very critical step and will depend on the load balancer algorithm running on the controller. This paper has focused on two primitive algorithms:

1. Round Robin: For each incoming connection the controller will assign a server in a round robin manner, refer Fig 3.1. This approach is very simplistic. It is helpful in scenarios where servers are handling requests for e.g. SIP proxy servers and DNS. But, this will not do a justified load balancing in scenarios where each connection can have different load involved. Also, this algorithm does not account for service availability. A server which is inactive for some reasons, may still be assigned to a client because the controller is unaware of the health of the servers.

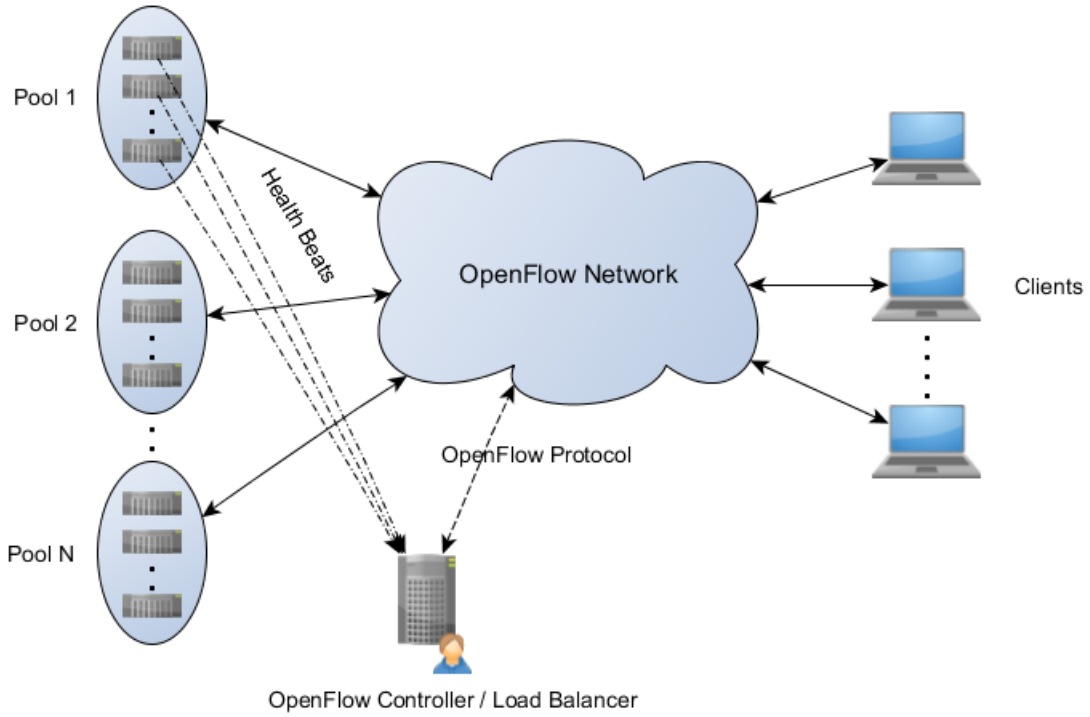

Fig. 3.2. Load based load balancing with OpenFlow

2. Load Based or Dynamic: The servers in the pool will send their health beats to the controller at periodic intervals, refer Fig 3.2. These health beats contain the server statistics. The server statistics sent to controller can vary from network to network depending on the requirements. Parameters considered in this paper are average load, CPU used, memory free and received bytes. A trade-off is made between all these parameters and finally a least loaded server is picked up by the controller. Refer flowchart 3.3 which gives a schematic view of how this algorithm will run on the floodlight controller. Load monitor process runs on the servers/members of the pool and sends health beats to the controller. Health listener implemented on the controller listens on port 8111 and grabs statistics from the health beats. Member analyzer process makes sure that any inactive member i.e. the member that does not send heart beats for 20 sec, is removed from the map file. When an packet in event arrives to the controller, the controller will insert flow entry in the switch such that

the packet is handled by the least loaded member. The information of least loaded member is returned by getLeastLoadedMember process. Flowchart 3.4 explains the getLeastLoadedMember process in detail.

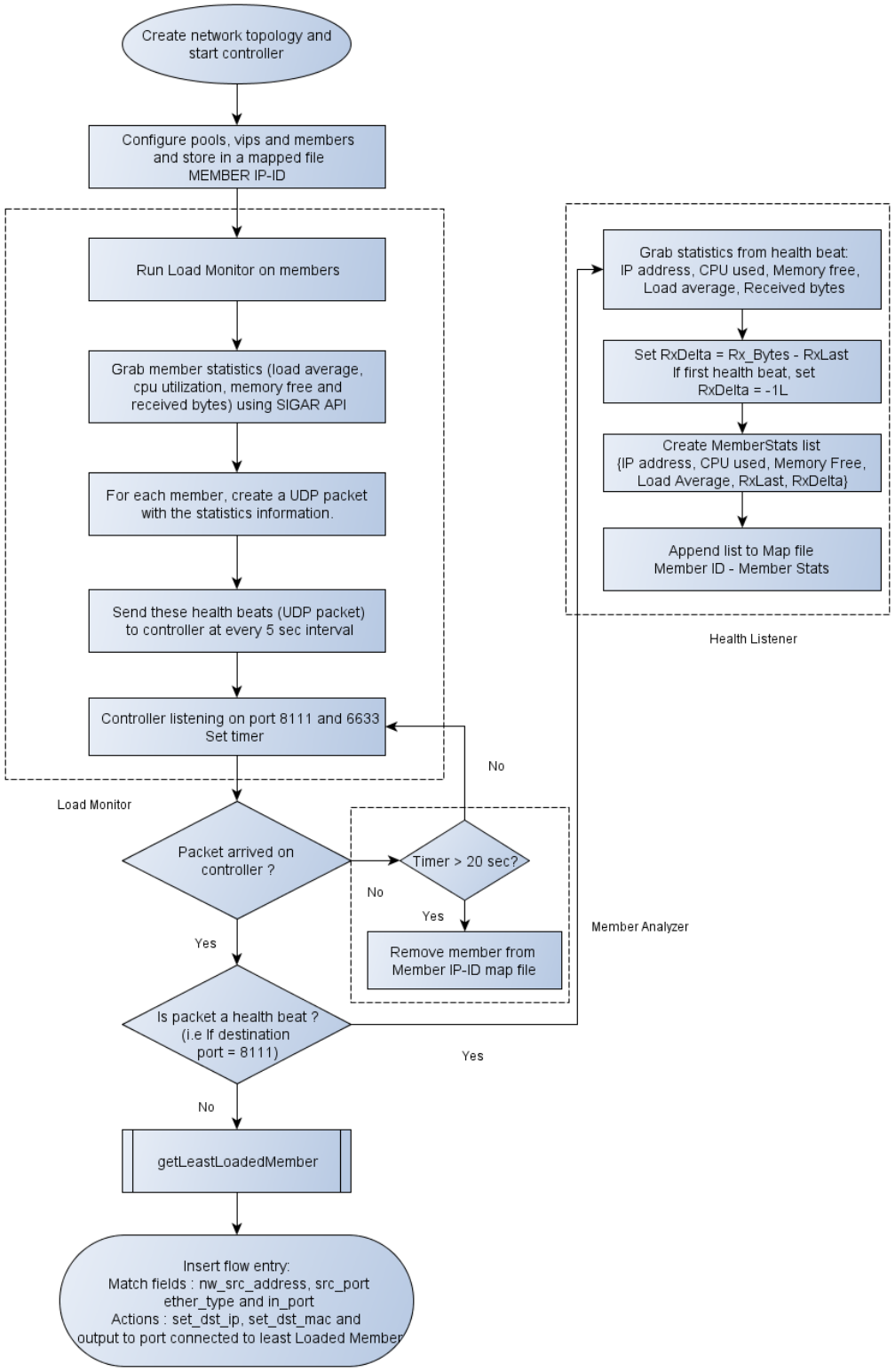

Fig. 3.3. Flowchart explaining various steps in load based algorithm

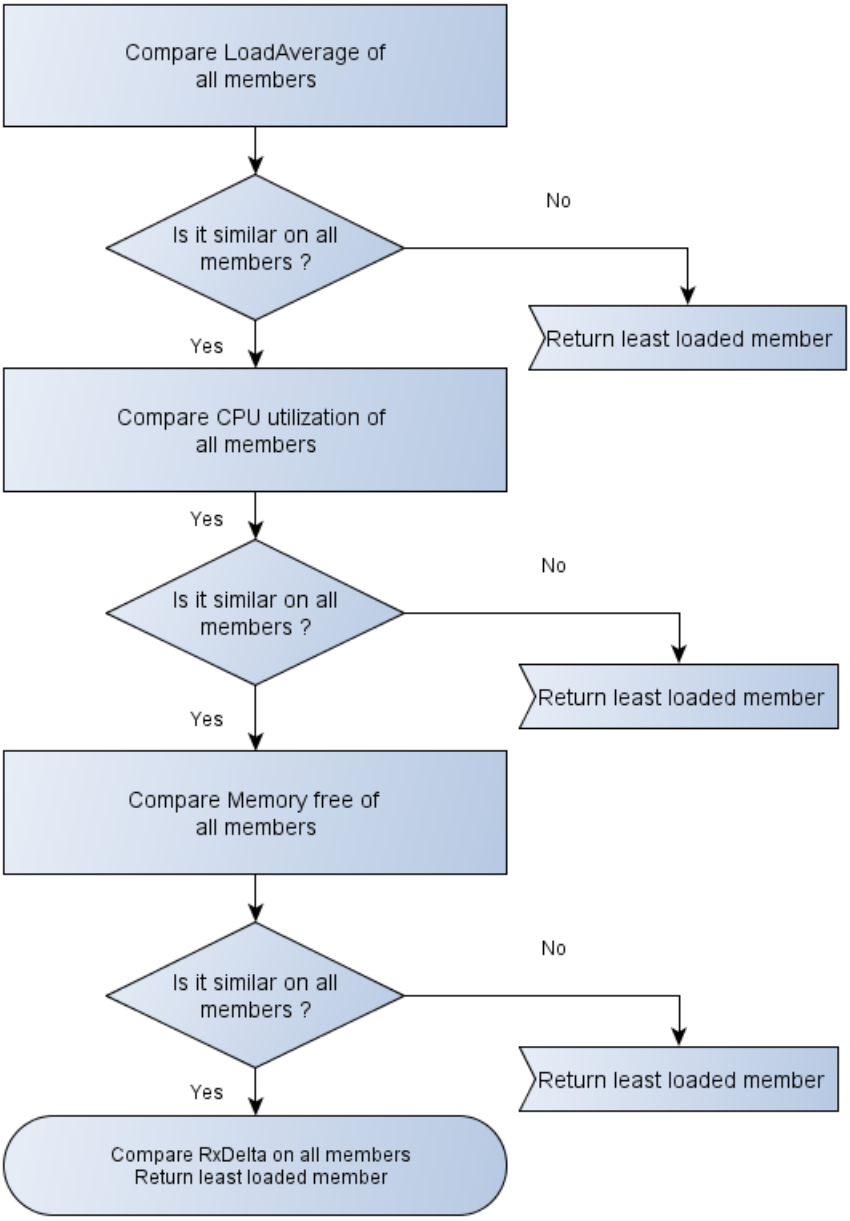

Fig. 3.4. Flowchart explaining getLeastLoadedMember process

4. EXPERIMENTAL SETUP AND RESULTS

The key components involved for experimental setup are: Floodlight OpenFlow Controller, Mininet, Iperf and Wireshark.

1. Floodlight Controller: Floodlight is a controller platform and it has a collection of applications built on top of it, refer to Fig 4.1. It realizes a set of functionalities to control an OpenFlow network while applications on top of it realize different features to fulfill user needs over the network. When you run floodlight, the controller with the set of Java module applications starts running. The REST APIs are exposed by all running modules and are available via the specified REST port (8080 by default) [14].

In this paper, Floodlight Controller is used as the controller platform to run round-robin and load based algorithms.

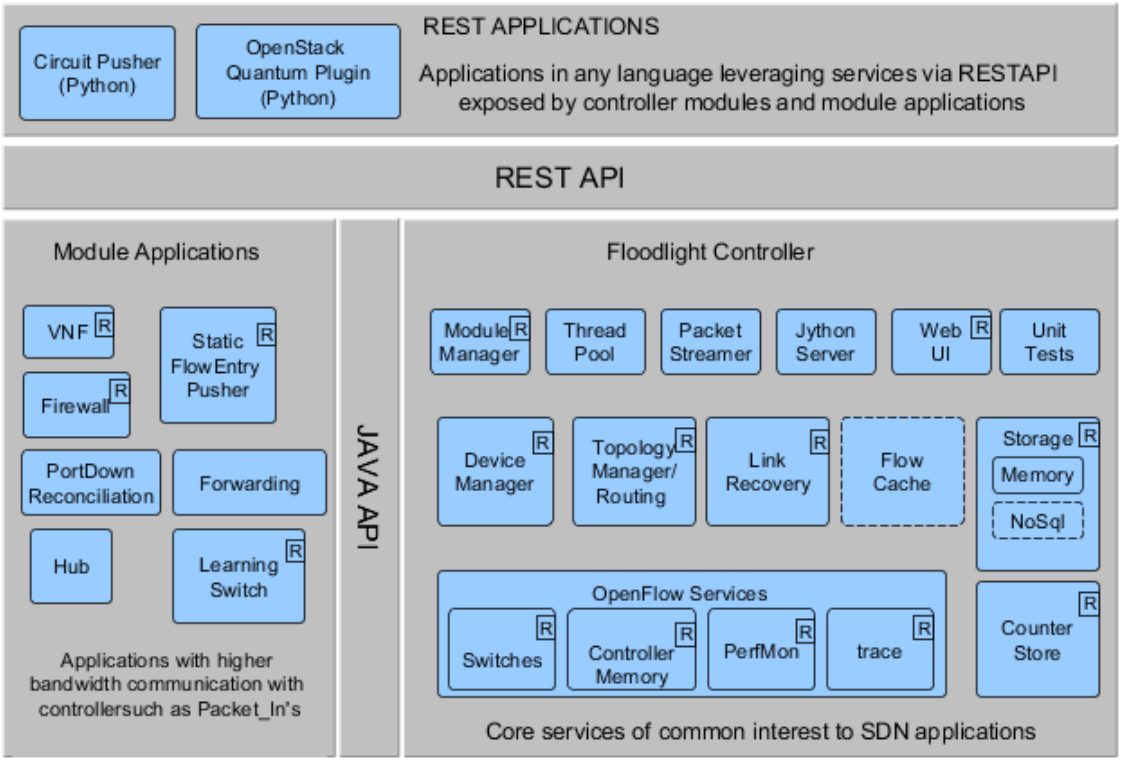

Fig. 4.1. Architecture of floodlight controller

2. Mininet:Mininet is an open source network emulator. It runs a collection of end-hosts, switches, routers and links on a single Linux kernel. It uses lightweight virtualization to make a single system look like a complete network, running the same kernel, system and user code [15]. In this paper, Mininet is used to create an OpenFlow network which is connected to the Floodlight controller.

3. Iperf:Iperf is a tool which utilizes the client/server architecture. It sends a selected amount of data from an iperf client to a listening iperf server and helps us to measure the time that it takes to transmit/receive the data. It also helps us to measure other performance parameters like jitter, bandwidth and dropped packets [16].

In this paper, Iperf is installed on the clients and on members of the pool. Iperf clients are transmitting UDP traffic and servers are listening on port 5001.

4. Wireshark: Wireshark is a network protocol analyzer for UNIX and windows systems. In this paper, Wireshark is used to verify connectivity between servers, OpenFlow switch and Floodlight controller.

The Fig 4.3 shows a basic setup that is used for experimentation. This network topology is emulated using Mininet. Floodlight controller is started and connectivity between the OpenFlow network and controller is verified. OpenFlow switch and controller exchange series of messages for connection establishment and setup (refer to Fig 4.2). Pools, virtual IP and members of the pool need to be configured on the controller (refer to Appendix Script 2). In this setup, only one pool is defined to load balance UDP traffic. This pool has 3 members or servers which are running Iperf on them. There are 25 clients which are also running Iperf on them. A virtual IP address is assigned to this pool.

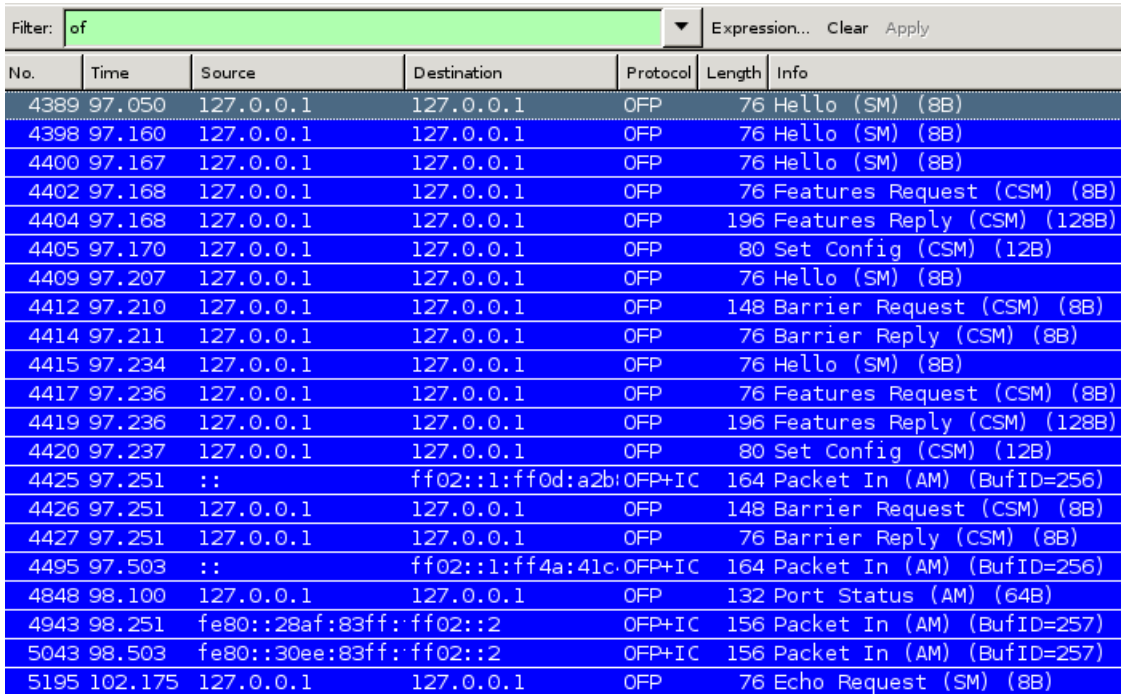

Fig. 4.2. OpenFlow wireshark trace showing OF messages

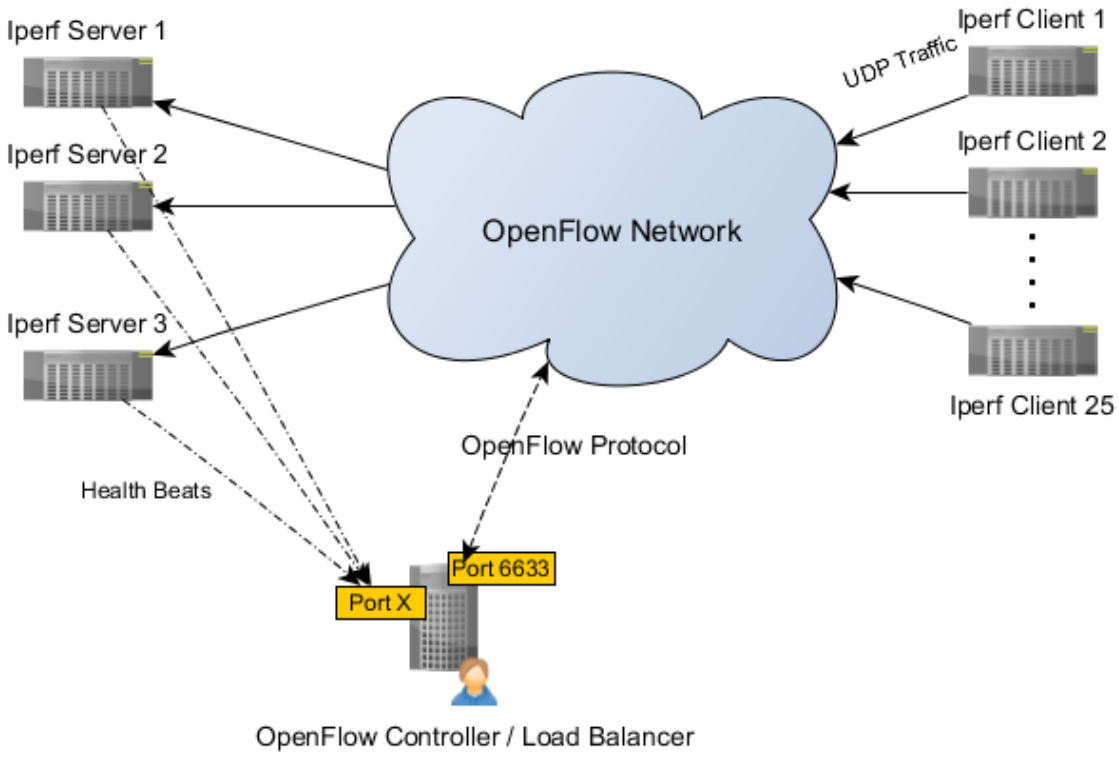

Fig. 4.3. Experiment setup

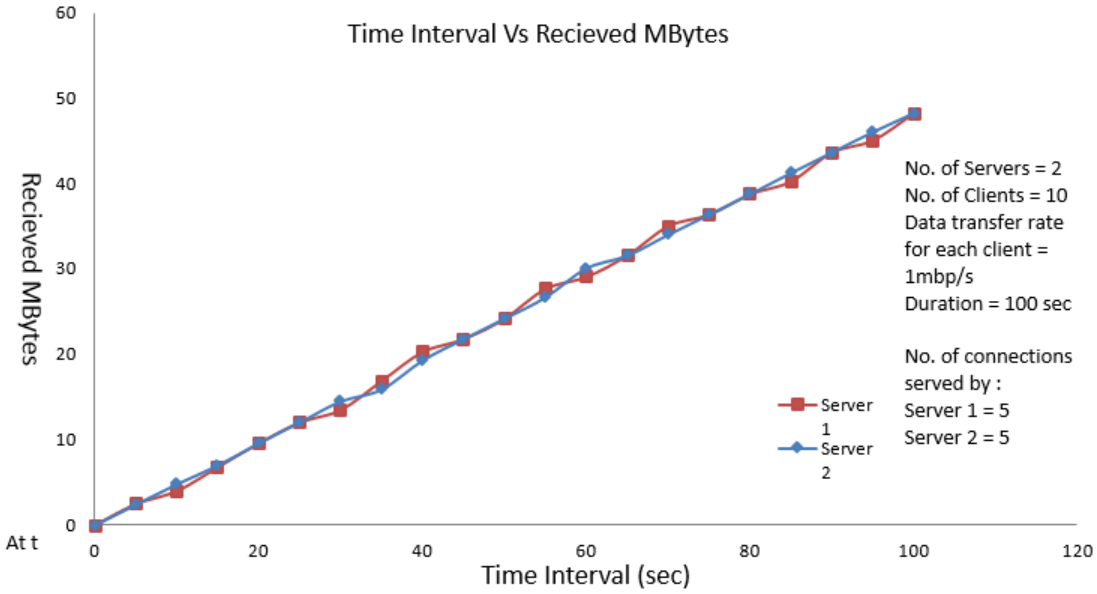

Fig. 4.4. Received load by servers at an instance of time for fixed data transfer rate (round robin algorithm)

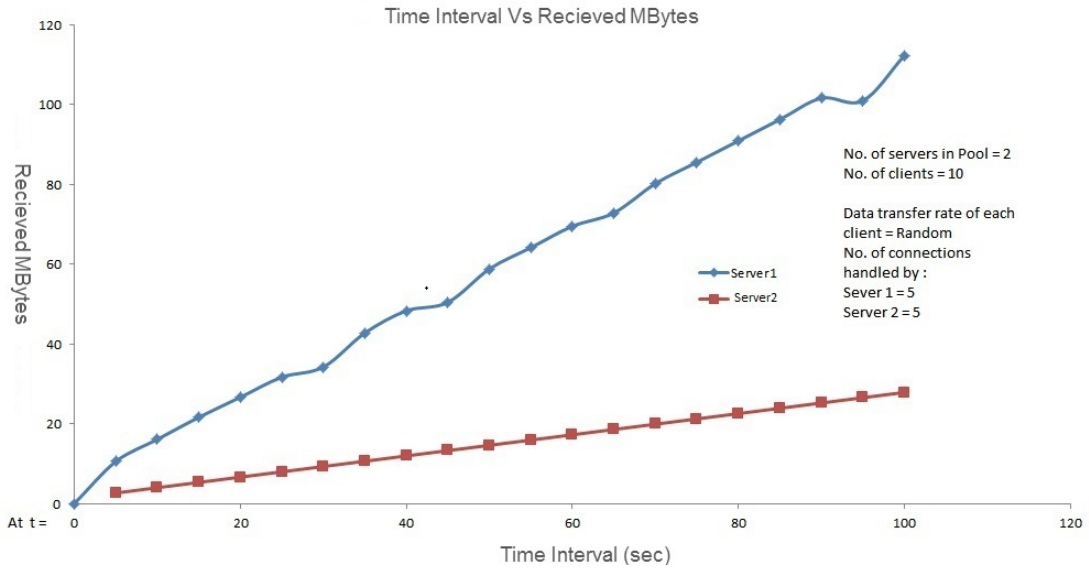

Fig. 4.5. Received load by servers at an instance of time for random data transfer rate (round robin algorithm)

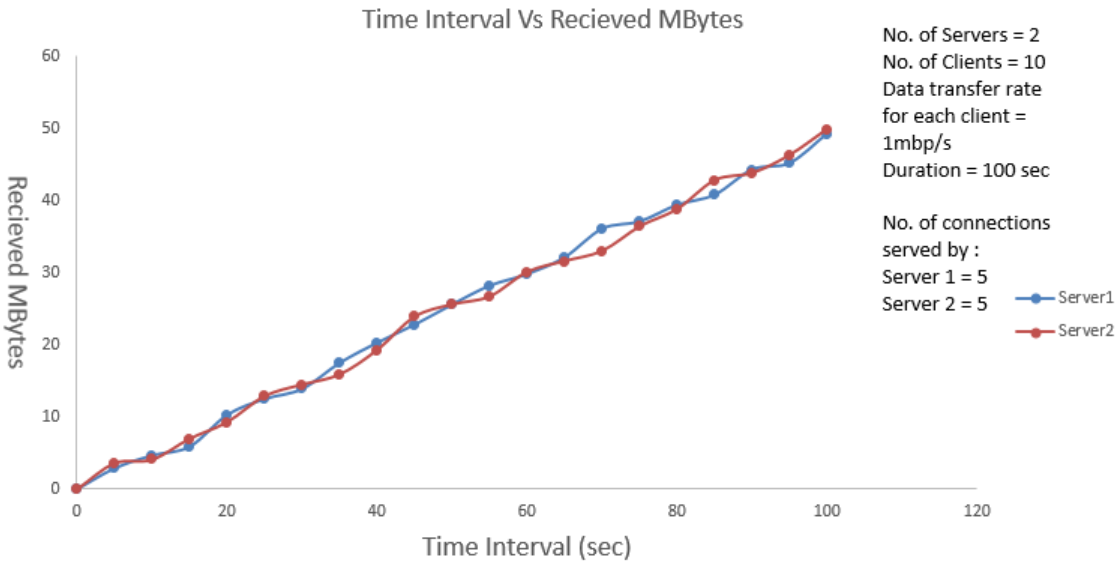

Fig. 4.6. Received load by servers at an instance of time for fixed data transfer rate (load based algorithm)

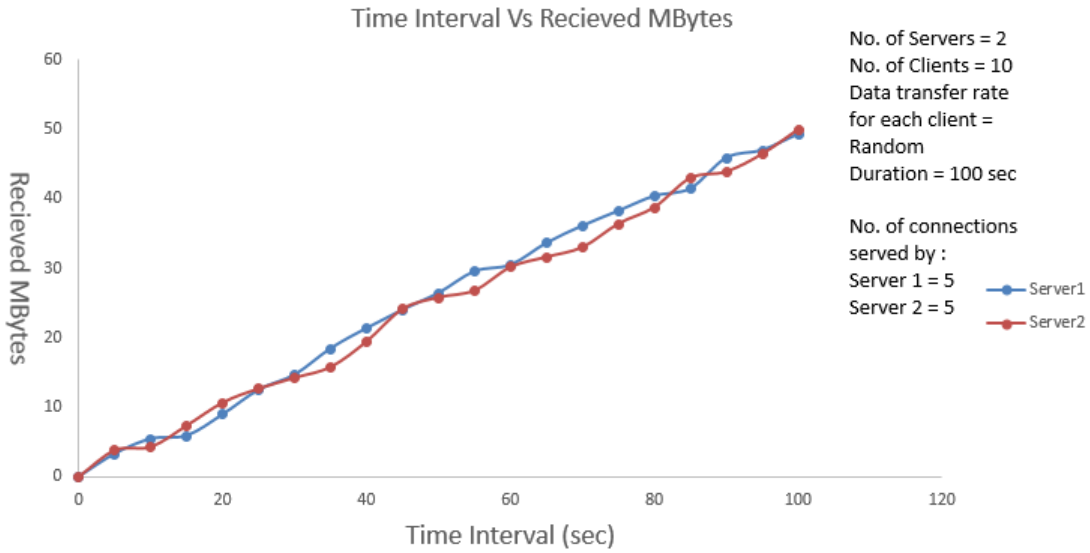

Fig. 4.7. Received load by servers at an instance of time for random data transfer rate (load based algorithm)

Fig. 4.8. Average jitter variation with increase in number of clients (comparison between round robin and load based algorithm)

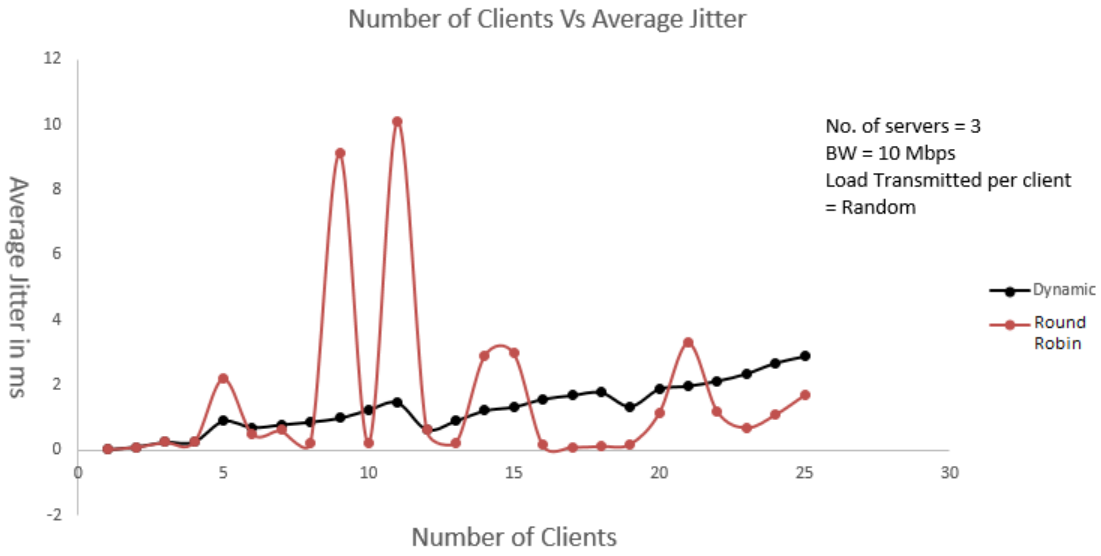

Fig. 4.9. Average dropped count variation with increase in number of clients (comparison between round robin and load based algorithm)

For the first part of the experimentation, the floodlight controller runs the round-robin load balancer application on it. All the clients transmit UDP traffic to the virtual IP address. The load balancer picks the servers in a round-robin fashion and assigns the incoming connection to the server. Refer to Fig 4.4 wherein clients are sending at a fixed data rate of 10 Mbps for a fixed duration. The servers are sharing the load almost equally and each server. When the same experiment is repeated with random data rate, refer to Fig 4.5 the server1 is heavily loaded as compared to the server 2. This variation in the load received by server1 and server2 is because round-robin algorithm is connection based and not load based.

For the second part of the experimentation, the floodlight controller runs the load based balancer application on it. All the clients transmit UDP traffic to the virtual IP address. The load balancer picks the least loaded server for each incoming connection. Every 5 sec the least loaded member is revised based on the latest statistics. Refer to Fig 4.6 and 4.7 here the servers are almost load balanced irrespective of data rate of the clients. Refer to Fig 4.8 and 4.9 which show a comparison between both the algorithms. Here, number of clients is varied and average dropped % and average jitter is plotted. The clients are transmitting at a random data rate. The average dropped and jitter is increasing linearly in case of load based algorithm but we can see sudden spikes in case of round-robin algorithm; the possible reason being that at certain instances of time the servers are heavily loaded as compared to other instances.

5. PROPOSED THEORY – INTERGRATION OF VOIP NETWORK AND OPENFLOW

5.1 SIP background

Session Initiation Protocol is a signaling protocol designed to set up, modify and terminate media sessions between two end-points [17]. A session in SIP is an end to end relationship between two parties involved in a media exchange; in VoIP a session corresponds to a phone call. This is also called a dialog in SIP. Throughout a dialog a state is maintained on the SIP server [10]. SIP does not manage or allocate network resources as does a network resource reservation protocol RSVP. Instead, SIP provides the following basic capabilities:

1. It finds the end users or agents. Endpoints or end users are addressed in email like formats i.e. user@InternetAddress like shreya.com.

2. Different end-points have different capabilities. Hence, in SIP message each endpoint lists the set of codecs that it supports. A protocol called SDP (Session Description Protocol) is used to define such capabilities. SDP message is carried in the body of a SIP message. It performs negotiation between end points for media type, media format and all other related properties.

3. SIP also manages sessions. SIP messages (like INVITE and BYE) are used to set up and teardown a session or a dialog. These type of control messages are exchanged on a separate channel from media.

5.1.1 Network elements

Following are the Elements that comprise SIP basic architecture:

1. User Agents: They initiate or terminate calls. Some of the hardware user agents are IP phones, cell phones, pagers and hard VoIP phones. Examples of software user agents are media mixers, Instant Messaging (IM) clients and soft phones.

2. Proxy Server: Proxy server handles routing. It provides dynamic association of SIP endpoints by exchanging transactions between the end users. A proxy interprets and forwards the request message. If required it can also rewrite some portion of request message before it forwards the request.

3. Registrar: A user agent has to register with a registrar and also provide its current IP address. The registrar saves this information to a database called the location server. A registrar is generally co-located with a proxy or redirect server and may also offer location services. Refer Fig 5.1 which illustrates message exchange between user agent and registrar.

4. Redirect Server: Redirection allows servers to reroute a request to a new server. When the redirect server receives a request, it sends the routing information of the new server in the response to the request. This takes this server out from the loop. If requester receives the redirection, it will create a new request based on the URI it has received from the redirect server [17].

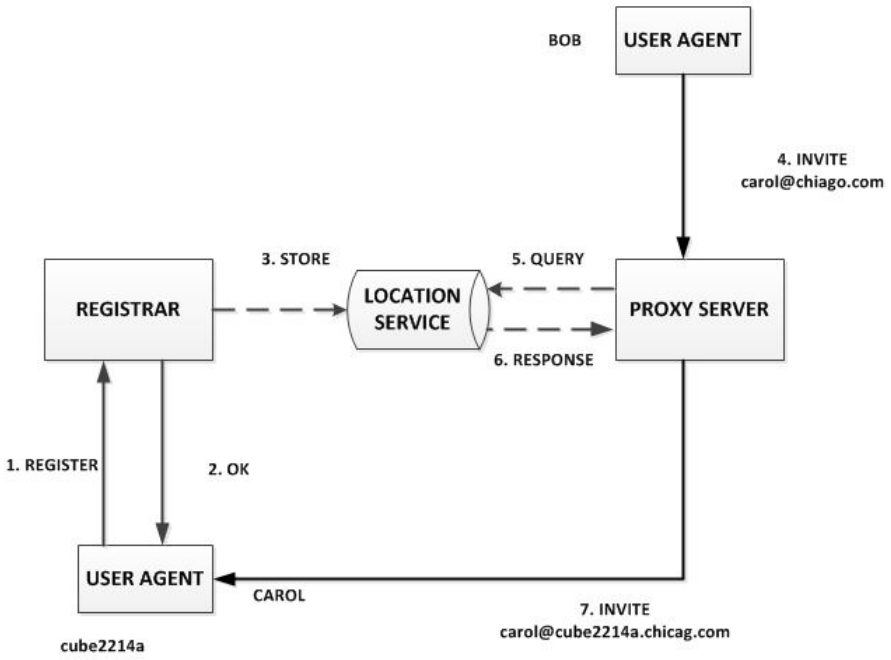

Fig. 5.1. Registrar offering location service to proxy server

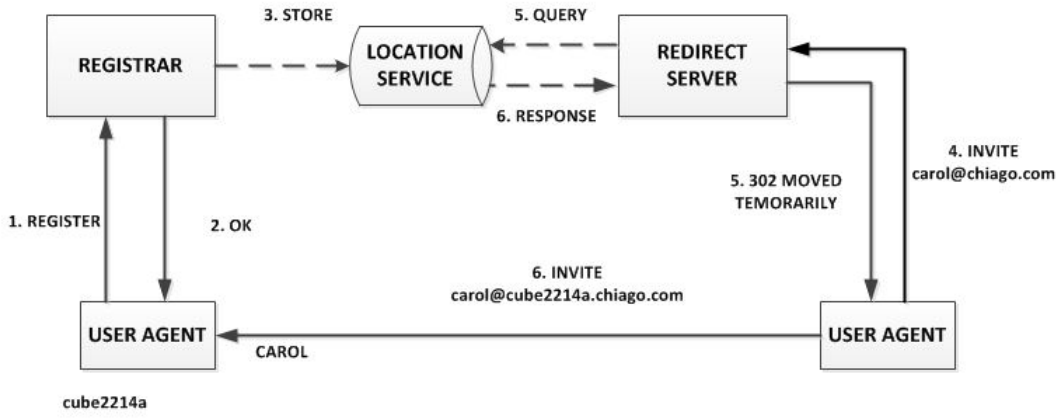

Fig. 5.2. A basic redirection scenario in SIP

Here, in this example of redirection the INVITE message is first sent to the redirect server. A 302 Moved Temporarily reply which contains a contact header is returned

by the server. Henceforth, the user agent is able to contact callee directly. The distinction between these SIP server types can be only logical and not necessarily physical [18]. Often proxy server itself may contain proxy, registrar and redirect functionality [17].

5.1.2 Overview of operation

SIP basic operation can be illustrated with an example of a call set up between two end parties: Alice and Bob. Alice calls Bob using her softphone. Two SIP proxy servers are present in this example. These servers will perform session establishment, management and tear-down. This type of configuration is referred to as "SIP trapezoid. Refer Fig 5.3, Alice sends an INVITE request to SIP URI of Bob. INVITE specifies what actions the server (Bob) should take. The INVITE is an important SIP method and contains a number of header fields. The header fields in INVITE include a unique number for identifying the call, source address, the destination address and the information about the type of session which is decided by the requester.

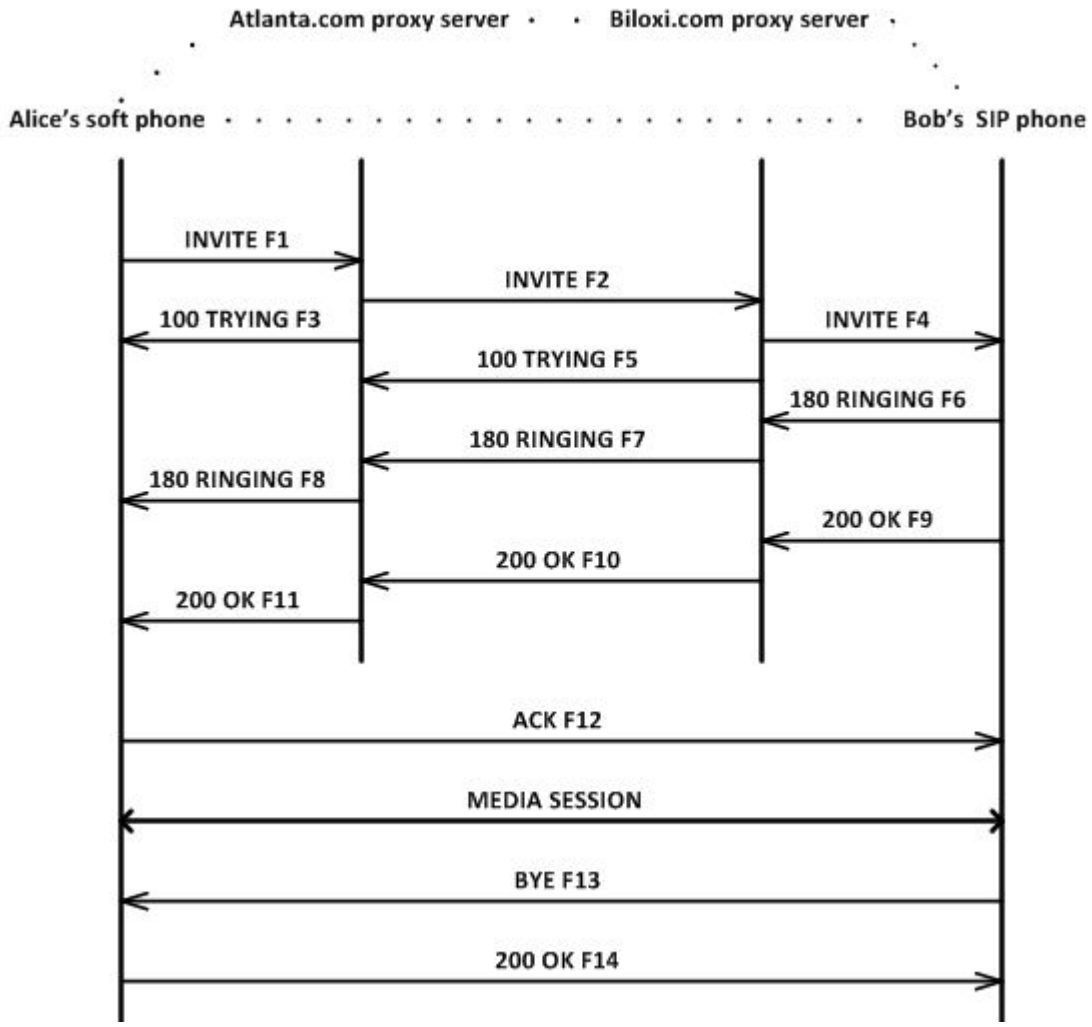

Fig. 5.3. SIP basic call flow

Alice uses her SIP identity for calling Bob. A SIP identity is nothing but a Uniform Resource Identifier (URI) which is also called a SIP URI. SIP URI is similar to an email address. Refer Fig 5.3 SIP:bob@biloxi.com is SIP URI of Bob, wherein biloxi.com is the domain of SIP service provider. Similarly the SIP URI for Alice is alice@atlanta.com [17].

Like HTTP, SIP is also a text-based protocol. Messages contain headers and the body depends on the type of message. Session Description Protocol (SDP) message which is wrapped inside a SIP message is used to negotiate session parameters (like codec and media type) between endpoints using an offer/answer model. Once the end-hosts agree to the session characteristics, the Real-time Transport Protocol (RTP) message are generally used to carry media traffic [19].

SIP message starts with the method name (INVITE in this example) followed by the header fields. Below is the set of the mandatory header fields:

1. Via: It is the address where the requester or sender of the request is expecting to receive the responses. In this example, pc33.atlanta.com is the via address at which requester (Alice) is expecting to receive replies to the requests. Via also contains a branch parameter that identifies this transaction.

2. To: This header field is used to display the name of destination end-point (Bob), for e.g. SIP:bob@biloxi.com is the SIP URI towards which the request is actually directed.

3. From: This header field is used to display name of the source end-point (Alice), for e.g. SIP:alice@atlanta.com is the SIP URI of the sender of the request. From field also contains a tag which is a randomized string that is added to the URI by the softphone or the calling device itself. It serves the identification purpose.

4. Call-ID: Each call has its own unique ID which is in the form of a string. This string is generated by the combination of a random string and the softphone's host name or IP address. The above defined four fields: Via, To, From, Call-ID together defines an end to end dialog between the sender (Alice) and receiver (Bob).

5. Command Sequence or Cseq: This header field contains method name and an integer. CSeq number is incremented whenever a new request is received within a particular dialog.

6. Contact: This header field contains a SIP URI that specifies a direct route to contact the requester (Alice). Contact is written in a username and fully qualified domain name (FQDN) format. FQDN is preferred over IP addresses. The difference between the Via field and the Contact field is that the former informs where to send the replies, whereas the latter informs where to send future requests.

7. Max-Forwards: This header field defines the maximum number of hops a request prior to reaching destination. Each Proxy that handles the message would decrement this number by one (similar to the time to live field in certain protocols). If the message is received by the proxy and it has the max-forwards set to 0, then proxy is supposed to return a 483 (Too many hops) response. This way endless looping of the message can be prevented.

8. Content-Type: This header field contains a description of the message body.

9. Content Length: This header field contains length of the message body in bytes. SIP message body is present after the header fields. SIP message body contains session description parameters defined by SDP protocol [20].The format of SDP messages is: code=value. The field code is always a single alphabet. Following is the minimum required set of fields in a SDP message:

i) v: This field contains the value of protocol version.

ii) o: This field contains the session owner and session identifier. Format of this field is: username, session id, version, network type, address type.

iii) s: This field contains the session name.

iv) t: This field contains the time for which the session is active.

v) m: This field contains the media type, format, and transport address.

```
INVITE SIP:bob@biloxi.com SIP/2.0
    Via: SIP/2.0/UDP pc33.atlanta.com; branch=z9hG4bK776asdhds
    Max-Forwards: 70
    To: Bob <SIP:bob@biloxi.com>
    From: Alice <SIP:alice@atlanta.com>;tag=1928301774
    Call-ID: a84b4c76e66710@pc33.atlanta.com
    CSeq: 314159 INVITE
    Contact: <SIP:alice@pc33.atlanta.com>
    Content-Type: application/SDP
    Content-Length: 142

v=0
o=alice 2890844526 2890844526 IN IP4 pc33.atlanta.example.com
s=session
t=0 0
m=audio 49170 RTP/AVP 0 8 97
```

Fig. 5.4. SIP invite body and SDP message enclosed

5.2 Insight to current scenario – PSTN-SIP interworking architecture

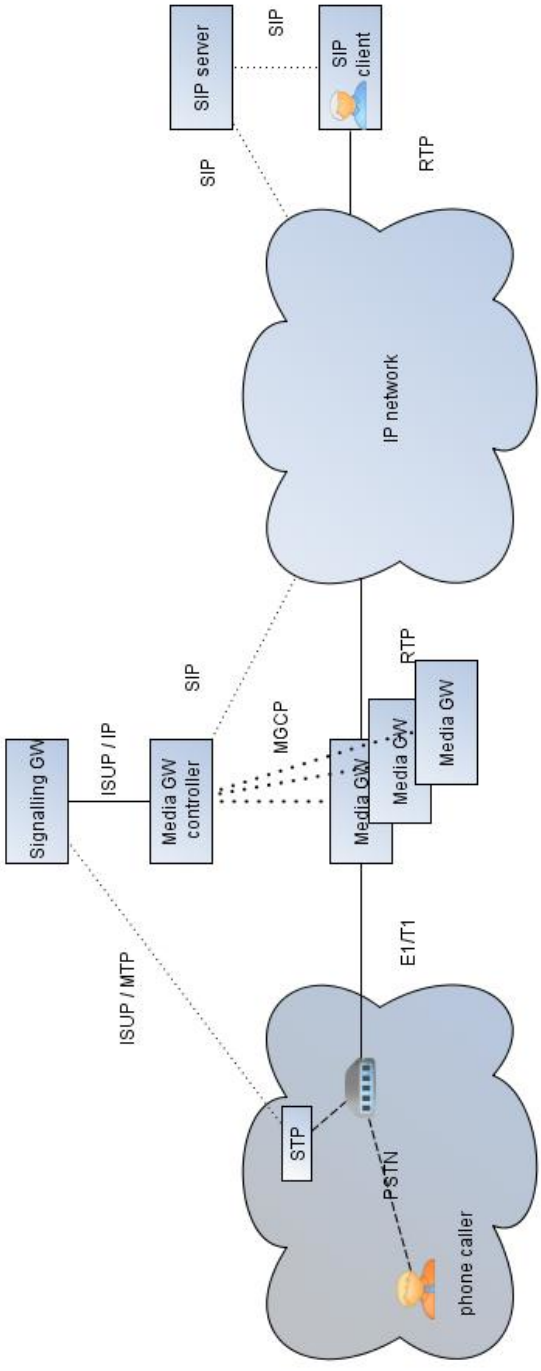

Fig. 5.5. A PSTN and SIP network interworking

Voice over IP offers phenomenal advantages, such as low network establishment investment cost and allows easy union of data and voice applications. But, it is practically impossible to replace all the existing circuit-switched telephony on the fly. It may require decades for VoIP to completely take over PSTN networks. VoIP networks and traditional circuit-switched networks (PSTN) and will have to work in parallel for a long time, making their interworking or compatibility an inevitable issue. Generally, the following network configurations are implemented: PSTN to SIP termination, PSTN to PSTN transition via SIP networks and SIP termination to PSTN. Wherein a SIP gateway (GW) has to be present for connecting the PSTN with the Internet, refer Fig 5.5.

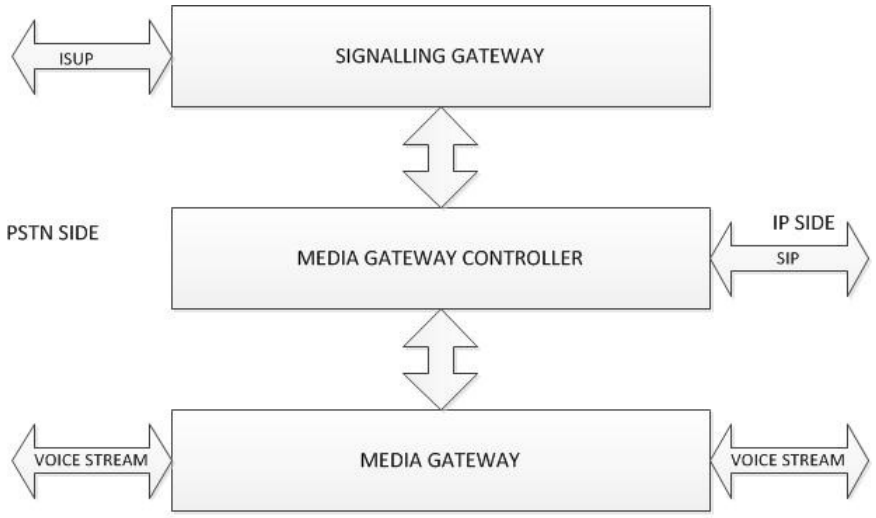

Fig. 5.6. Functional description of SIP-PSTN gateway

A network gateway comprises of three functional units 5.6: media gateway controller (MGC), a signaling gateway (SG) and media gateway(MG). Signaling gateway performs routing of all ISUP (ISDN user part) messages for the media gateway. The Message Transfer Part (MTP) (which is the lower layer of SS7) is replaced by IP so that IP network entities understand it. ISDN User Part (ISUP) which forms the

upper layer of MTP is encapsulated into TCP/IP headers and sent to a signaling gateway. It is the role of the signaling gateway to perform a conversion of the dialed number into an IP address. After this conversion, the call is routed over an IP network.

Media gateway converts Pulse Coded Modulated information (PCM) to packetized information (used in VoIP) and vice versa. Media gateway controller handles management, registration and control functionality of resources in the media gateways. The media gateway controller accepts signaling from the PSTN in native format. It would then convert it to the format which an IP network would understand. It also controls multiple media gateways by introducing Megaco/MGCP and performs 3A functions (authentication, authorization and accounting). One fundamental problem of load balancing in SIP-PSTN architecture can be solved by selection of an appropriate media gateway for calls originating from the VoIP networks. When the SIP proxy server receives an INVITE request, it takes help of the location server to locate the possible media gateway controller to terminate this call. While signaling conversion is taken care by signaling gateway, media gateway controller takes of the resource allocation, deciding the call parameters (like codec selection). After this, the corresponding voice stream goes directly through the selected media gateway. Sometimes media gateway and media gateway controller are in one physical device. In heavily loaded networks we can have multiple media gateways being controlled by one media gateway controller. A primitive gateway selection algorithm will route incoming calls solely on the basis of the destination address; however an intelligent approach can lead to better services.

5.3 PSTN-SIP architecture integrated with OpenFlow

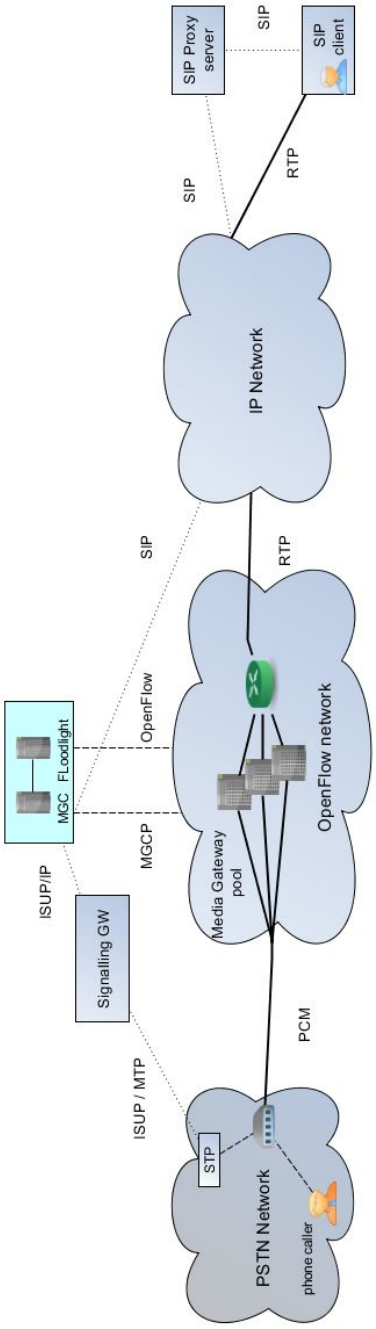

Fig. 5.7. SIP-PSTN architecture integrated with OpenFlow load balancing

Refer Fig 5.9 which proposes an integration of OpenFlow Protocol to current existing architecture for appropriate selection of media gateway. The OpenFlow controller OFC will run a discovery process for the selection of media gateway to handle the incoming voice load. This discovery process can be based on any algorithm depending on the needs of the network. OFC needs to inform media gateway controller about the selected media gateway. Once media gateway controller and signaling gateway will handle the entire SIP signaling conversion then the voice load can directly go through the selected media gateway. For this to happen, media gateway controller instructs the gateway about the coding characteristics expected by the line-side of the caller. It also decides other call-parameters for the media gateway. Media gateway controller then sends create connection command to both the endpoints (media gateway and PSTN switch in our example). After this the caller and callee can communicate. Once the call is disconnected, media gateway notifies media gateway controller, which would then take care of accounting or billing of the caller. This whole process involves a lot of signaling at intermediate steps: SIP client and SIP proxy server, SIP proxy server and media gateway controller signaling gateway, signaling gateway and PSTN switch.

The scope of this paper is limited to media gateway selection and load balancing using OpenFlow protocol. Signaling between intermediate devices, though being a very important part of the complete analysis, is out of scope of this paper. Following OpenFlow load balancers are proposed and are based on the algorithms which are discussed in Chapter 3:

1. Round Robin Load Balancer: In this approach, the discovery process running on OpenFlow controller will be based on a simplistic round robin approach. Once the media gateway is selected for an incoming invite from a caller, the proceeding voice load from that connection will go through the same media-gateway.

Any other request following this request will be assigned another media-gateway from the media gateway cluster. Health or load on the gateways is not taken into consideration.

2. Load Based or Dynamic Redirection Load-balancer: In this approach, the discovery process running on OpenFlow controller will be a more complex approach. All the media gateways will send health beats at regular intervals to the OFC (load balancer). These health beats will contain various statistics of the media gateways e.g. received bytes, CPU usage, free memory etc. A trade-off between all these parameters will lead to designating a least loaded media gateway. OFC shares this information with the media gateway controller. Since, media gateway controller knows which media gateway is least loaded; the voice load will be traversed through that gateway. As discussed before, in real world the duration of calls is drastically variable; some calls can be too long whereas some calls are too short. This can make us use the resources unevenly because some media gateways are free whereas some are being used extensively. Using load based load balancer we could potentially resolve such an issue as well. If other media gateways are free, a long ongoing conversation from a heavy loaded media gateway could be redirected to the free media gateway. This should be done without letting the caller know that the intermediate path was changed.

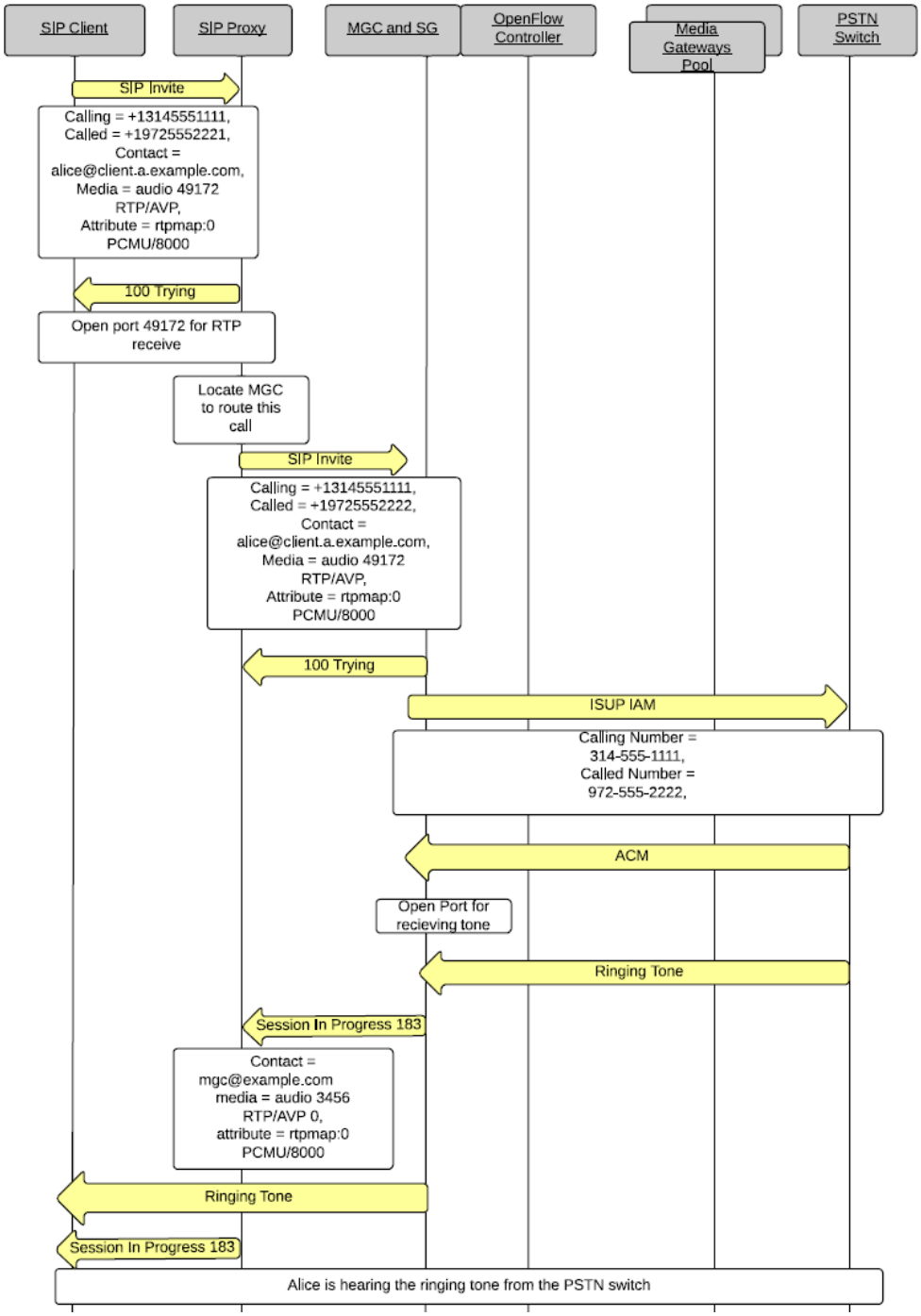

Fig. 5.8. Event diagram for SIP-PSTN architecture integrated with OpenFlow load balancing [Contd..]

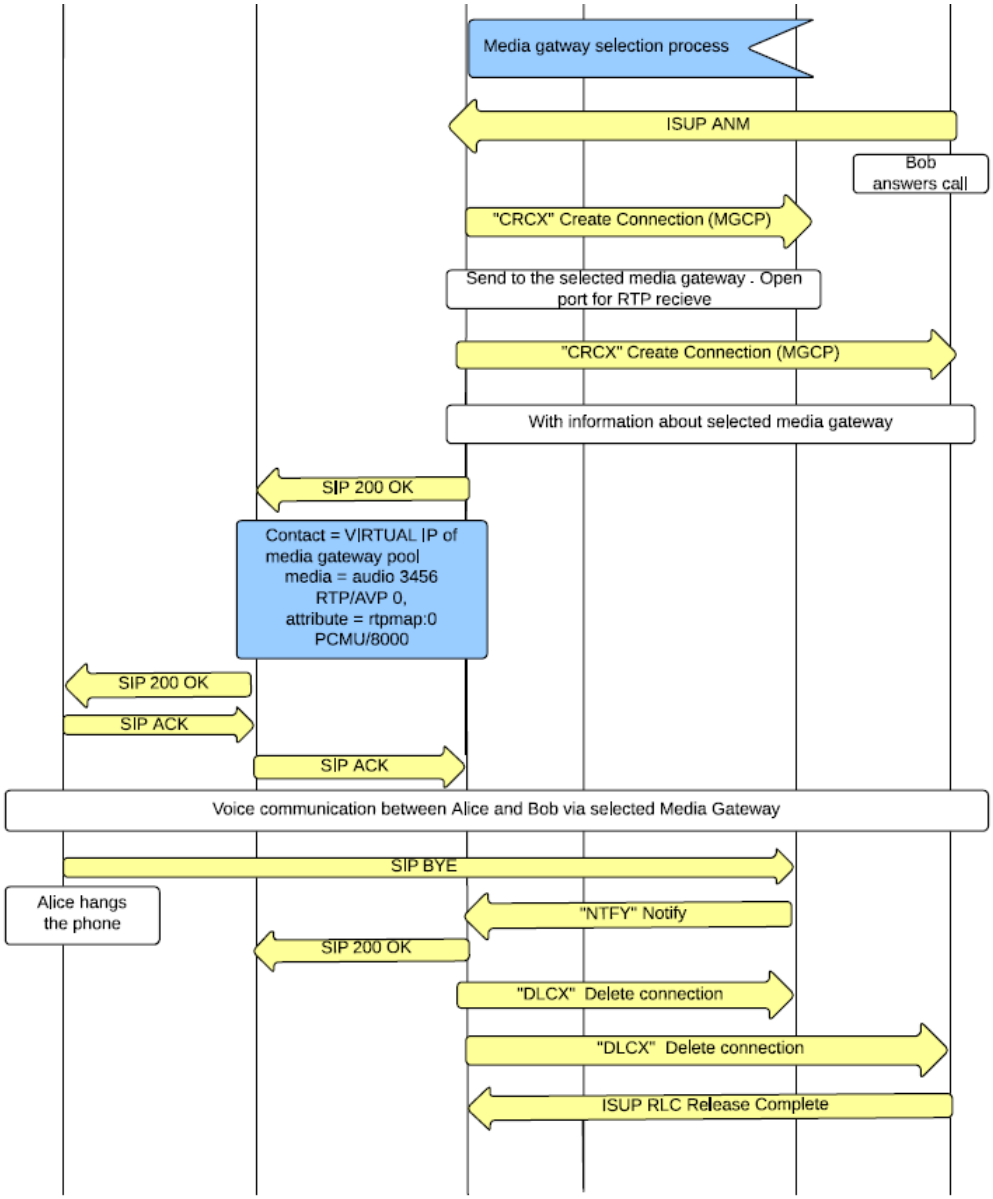

Fig. 5.9. Event diagram for SIP-PSTN architecture integrated with OpenFlow load balancing

Fig 5.9 shows a series of events that would take place in this proposed architecture. SIP client dials the globalized number to reach the callee. SIP proxy informs the SIP

client about the call establishment. Client prepares itself to receive data by opening a port. SIP proxy uses a location server to locate the gateway for routing this call. A media gateway controller (MGC) address is returned by the location server as it is a PSTN call. The SIP INVITE message is sent to MGC. MGC informs proxy that it is trying to set up the call and also sends (IAM) ISUP Initial Address Message to the PSTN. This message contains the caller and callee number information. In response to the IAM, the switch replies back with a ISUP Address Complete message (ACM) which ensures that all the digits were included in the ISUP IAM. Hence, switch has received all the digits and is now processing the call.

Next step is connecting a one way voice path between switch and SIP client for sending the voice path. Once the ACM message is received from the switch, SIP session in progress message is generated by the MGC. The session in progress message contains RTP media information for this call.

Callee answers the call. ISUP Answer message is sent to MGC by the switch. A bi-directional path is established between the end-points. A SIP OK message is used by MGC to indicate that the call has been answered. Media gateway selection is done based on the load balancing algorithm used. MGC sends CRCX message to both the endpoints (PSTN switch and Media Gateway) to establish the voice path. MGC also sends SIP OK message to SIP client via SIP proxy. This OK message does not contain the address of MGC itself, but it contains the virtual IP address of the media gateway pool. This would ensure that SIP client can directly send voice traffic via media gateway now. The SIP client acknowledges the receipt of SIP OK message to proxy.

At this point, media gateway bridges the bidirectional RTP path between the caller and the PSTN Switch. When the SIP client has to hang the phone, it sends a BYE message to proxy server which further sends this BYE indication to the media

gateway. Media gateway notifies the media gateway controller that the end party has hung up using NTFY message. MGC sends DLCX connection to media gateway and PSTN switch to terminate this existing connection and takes care of the billing and accounting for this call.

When a long ongoing conversation is going on and the server is heavily loaded we can redirect this call to another server without letting the caller know. This is possible because caller never knows the physical address of the media gateway; instead it sends the voice stream to the virtual ip of the media gateway pool. It becomes the responsibility of OpenFlow controller to route the call to the selected media gateway. Refer Fig 5.10, which shows the series of steps we can follow and redirect the call from a heavily loaded gateway to a least loaded gateway without letting the caller know about the intermediate redirection.

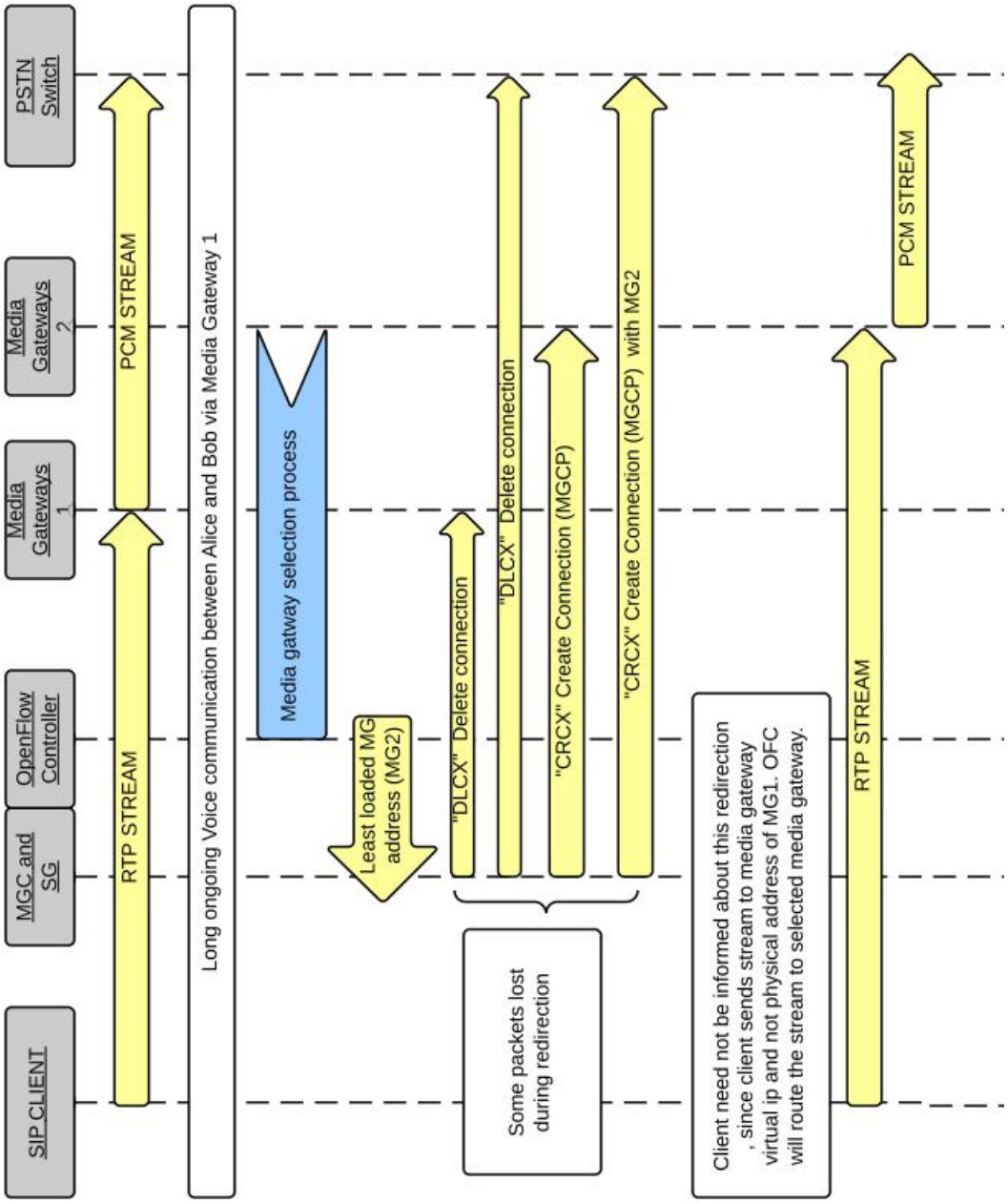

Fig. 5.10. Redirection of call from a heavily loaded media gateway to least loaded gateway

6. FUTURE SCOPE AND CONCLUSION

This paper shows that it is possible to get similar functionality of a commercial load balancer using only commodity hardware OpenFlow switches. Depending on the needs of the network we can define our own load balancing algorithms and balance the network without using expensive devices. OpenFlow gives control of network to the user and makes the network programmable as per our requirement. In this paper, two basic algorithms: Round Robin and Load Based are implemented using OpenFlow and their comparative study is done. Load Based has shown better results than Round Robin algorithm. This paper has also extended a theory of using OpenFlow load balancing in current SIP-PSTN architecture to improve VoIP performance. The feasibility of such a theory needs to be verified with a complete working model and a detailed analysis needs to be done before we can conclude its deployability.

With the growth in Software defined Networking, OpenFlow protocol holds a great potential to improve current Internet services but improvement in the OpenFlow device hardware is a must. Due to limitations in device hardware, all the flows are not processed in switch hardware but are processed in software, which is a slower path. This can add more delay to the service. Number of flows handled by an OpenFlow device and the flow processing time are important factors that will determine the overall performance of using OpenFlow in any Internet service.

LIST OF REFERENCES

LIST OF REFERENCES

[1] N. Handigol, M. Flajslik, S. Seetharaman, R. Johari, and N. McKeown, "Aster*x: Load-balancing as a network primitive," in *9th GENI Engineering Conference (Plenary)*, pp. 1–2, November 2010.

[2] G. Kambourakis, D. Geneiatakis, T. Dagiuklas, C. Lambrinoudakis, and S. Gritzalis, "Towards eective sip load balancing," in *3rd Annual VoIP Security Workshop*, pp. 1–17, ACM Press, June 2006.

[3] "Software-defined networking: The new norm for networks," pp. 1–12, Open Networking Foundation, April 2012.

[4] N. McKeown, T. Anderson, and H. Balakrishnan, "Openflow: Enabling innovation in campus networks," in *ACM SIGCOMM Computer Communication Review*, pp. 69–74, April 2008.

[5] R. Sherwood, G. Gibb, K. K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar, "Flowvisor: A network virtualization layer," pp. 1–14, Deutsche Telekom Inc. RD Lab, Stanford University, Nicira Networks, October 2009.

[6] B. Heller, "Openflow switch specification version 1.0.0 (wire protocol 0x01)," pp. 1–42, December 2009.

[7] Y. Hong, J. H. No, and S. Y. Kim, "Dns based load balancing in distributed web-server systems," in *Software Technologies for Future Embedded and Ubiquitous Systems, 2006 and Second International Workshop on Collaborative Computing, Integration, and Assurance. SEUS 2006/WCCIA 2006.*, pp. 1–4, 2006.

[8] IBM, "Dns-based load balancing." http://publib.boulder.ibm.com/infocenter /iseries/v5r3/index.jsp?topic=%2Frzajw%2Frzajwdnsrr.htm. Date last accessed: July 19, 2013.

[9] J.-S. Leu, H.-C. Hsieh, Y.-C. Chen, and Y.-P. Chi, "Design and implementation of a low cost dns-based load balancing solution for the sip-based voip service," in *Asia-Pacific Services Computing Conference, 2008. APSCC '08. IEEE*, pp. 310–314, 2008.

[10] J. Hongbo, A. Iyengar, E. Nahum, W. Segmuller, A. Tantawi, and C. Wright, "Design, implementation, and performance of a load balancer for sip server clusters," in *IEEE/ACM Transactions, vol.20, no.4*, pp. 1190–1202, August 2012.

[11] N. Handigo, S. Seetharaman, N. McKeown, and R. Johari, "Plug-n-serve: Load-balancing web traffic using openflow," pp. 1–2, 2009.

[12] H. Uppal and D. Brandon, "Openflow based load balancing," (University of Washington), pp. 1–7, 2012.

[13] M. Koerner and O. Kao, "Multiple service load-balancing with openflow," in *High Performance Switching and Routing (HPSR), 2012 IEEE 13th International Conference*, pp. 210–214, 2012.

[14] BigSwitch, "Floodlight: Load balancer." http://docs.projectfloodlight.org /display/floodlightcontroller/Load+Balancer. Date last accessed: October 21, 2013.

[15] B. Lantz and B. Heller, "Mininet." https://github.com/mininet/mininet /wiki/Introduction-to-Mininetwiki-what. Date last accessed: October 21, 2013.

[16] "Iperf tutorial." http://openmaniak.com/iperf.php. Date last accessed: May 3, 2013.

[17] Rosenberg and H. Schulzrinne, "Sip: Session initiation protocol," in *Internet Engineering Task Force, RFC 3261*, June 2002.

[18] B. Sterman, "Real-time billing in sip." http://www.recursosvoip.com/docs/english /realtimebilling.pdf. Date last accessed: August 1, 2013.

[19] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson, "Rtp: Real time transport protocol," in *Internet Engineering Task Force, RFC 3550*, July 2003.

[20] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson, "An offer/answer model with session description protocol (sdp)," in *Internet Engineering Task Force, RFC 3550*, July 2003.

APPENDIX

APPENDIX

Script 1

```
#!/usr/bin/python

"""
Script demonstrates basic mininet topology
This code links two switches S1 ----- S2 and then hooks it to a
reference controller in mininet.
Also, starts a wireshark trace and stops it

"""

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 switches
        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 &")

#Start mininet network
net.start()
s3, s4 = net.get('s3', 's4')
CLI(net)

#Stops mininet network
net.stop()

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

#!/bin/sh
```

Script 2

```
"""
This script is used to set up the load balancer vips, pools,
and members in any linux console.

After starting floodlight controller, start mininet, configure pools and
load balancing will be performed
"""

curl -X POST -d '{"id":"1","name":"vip1","protocol":"icmp",
"address":"10.0.0.100","port":"8"}'
http://localhost:8080/quantum/v1.0/vips/
curl -X POST -d '{"id":"1","name":"pool1","protocol":"icmp",
"vip_id":"1"}' http://localhost:8080/quantum/v1.0/pools/
curl -X POST -d '{"id":"1","address":"10.0.0.1","port":"8",
"pool_id":"1"}' http://localhost:8080/quantum/v1.0/members/
curl -X POST -d '{"id":"2","address":"10.0.0.2","port":"8",
"pool_id":"1"}' http://localhost:8080/quantum/v1.0/members/
curl -X POST -d '{"id":"3","address":"10.0.0.3","port":"8",
"pool_id":"1"}' http://localhost:8080/quantum/v1.0/members/

curl -X POST -d '{"id":"2","name":"vip2","protocol":"udp",
"address":"10.0.0.100","port":"200"}'
 http://localhost:127.0.0.1:8080/quantum/v1.0/vips/
curl -X POST -d '{"id":"2","name":"pool2","protocol":"udp",
"vip_id":"2"}' http://localhost:8080/quantum/v1.0/pools/
curl -X POST -d '{"id":"4","address":"10.0.0.1","port":"200",
```

```
"pool_id":"2"}' http://localhost:8080/quantum/v1.0/members/
curl -X POST -d '{"id":"5","address":"10.0.0.2","port":"200",
"pool_id":"2"}' http://localhost:8080/quantum/v1.0/members/
curl -X POST -d '{"id":"6","address":"10.0.0.3","port":"200",
"pool_id":"2"}' http://localhost:8080/quantum/v1.0/members/
```

Script 3

```
#!/usr/bin/python

"""
This script creates a test network for load balancing. Hosts can talk to
Internet through NAT.
To test loadbalacing iperf is used to generate load from the clients .
Server performance reports get generated and is redirected to
an output file for analysis.

"""

from mininet.cli import CLI
from mininet.log import lg, info
from mininet.node import Node
from mininet.topolib import TreeNet
from mininet.util import quietRun
from mininet.net import Mininet
from mininet.node import RemoteController
from mininet.topo import *
from mininet.link import TCLink
import sys
from time import sleep
```

```
##################################
class MyTopo(Topo):
    "Single switch connected to n hosts."
    def __init__(self, n=5, **opts):
        Topo.__init__(self, **opts)
        switch = self.addSwitch('s1')
        for h in range(n):
            # Each host gets 50%/n of system CPU
            host = self.addHost('h%s' % (h + 1))
            # 10 Mbps, 5ms delay, 10% loss, 1000 packet queue
            self.addLink(host, switch,
           bw=10)


def startNAT( root, inetIntf='eth0', subnet='10.0/8' ):
    """Start NAT/forwarding between Mininet and external network
    root: node to access iptables from
    inetIntf: interface for internet access
    subnet: Mininet subnet (default 10.0/8)="""

    # Identify the interface connecting to the mininet network
    localIntf =  root.defaultIntf()

    # Flush any currently active rules
    root.cmd( 'iptables -F' )
    root.cmd( 'iptables -t nat -F' )
```

```
    # Create default entries for unmatched traffic
    root.cmd( 'iptables -P INPUT ACCEPT' )
    root.cmd( 'iptables -P OUTPUT ACCEPT' )
    root.cmd( 'iptables -P FORWARD DROP' )

    # Configure NAT
    root.cmd( 'iptables -I FORWARD -i', localIntf, '-d', subnet,
'-j DROP' )
    root.cmd( 'iptables -A FORWARD -i', localIntf, '-s', subnet,
'-j ACCEPT' )
    root.cmd( 'iptables -A FORWARD -i', inetIntf, '-d', subnet,
'-j ACCEPT' )
    root.cmd( 'iptables -t nat -A POSTROUTING -o ', inetIntf,
'-j MASQUERADE' )

    # Instruct the kernel to perform forwarding
    root.cmd( 'sysctl net.ipv4.ip_forward=1' )

def stopNAT( root ):
    """Stop NAT/forwarding between Mininet and external network"""
    # Flush any currently active rules
    root.cmd( 'iptables -F' )
    root.cmd( 'iptables -t nat -F' )

    # Instruct the kernel to stop forwarding
    root.cmd( 'sysctl net.ipv4.ip_forward=0' )

def fixNetworkManager( root, intf ):
    """Prevent network-manager from messing with our interface,
```

```
       by specifying manual configuration in /etc/network/interfaces
       root: a node in the root namespace (for running commands)
       intf: interface name"""
    cfile = '/etc/network/interfaces'
    line = '\niface %s inet manual\n' % intf
    config = open( cfile ).read()
    if ( line ) not in config:
        print '*** Adding', line.strip(), 'to', cfile
        with open( cfile, 'a' ) as f:
            f.write( line )
    # Probably need to restart network-manager to be safe -
    # hopefully this won't disconnect you
    root.cmd( 'service network-manager restart' )

def connectToInternet( network, switch='s1', rootip='10.254',
subnet='10.0/8'):
    """Connect the network to the internet
       switch: switch to connect to root namespace
       rootip: address for interface in root namespace
       subnet: Mininet subnet"""
    switch = network.get( switch )
    prefixLen = subnet.split( '/' )[ 1 ]
    routes = [ subnet ]  # host networks to route to

    # Create a node in root namespace
    root = Node( 'root', inNamespace=False )

    # Prevent network-manager from interfering with
 our interface fixNetworkManager( root, 'root-eth0' )
```

```
    # Create link between root NS and switch
    link = network.addLink( root, switch )
    link.intf1.setIP( rootip, prefixLen )

    # Start network that now includes link to root namespace
    network.start()

    # Start NAT and establish forwarding
    startNAT( root )

    # Establish routes from end hosts
    for host in network.hosts:
        host.cmd( 'ip route flush root 0/0' )
        host.cmd( 'route add -net', subnet, 'dev',
s host.defaultIntf() )
        host.cmd( 'route add default gw', rootip )

    return root

if __name__ == '__main__':
    lg.setLogLevel( 'info')

# Configure and start NATted connectivity

    topo = MyTopo(int(sys.argv[1]))
    net = Mininet(topo=topo,link=TCLink,controller=lambda/
    name: RemoteController(name, ip='127.0.0.1'))
    h1,h2,h3,h4,h5,s1 = net.get('h1', 'h2', 'h3', 'h4' , 'h5', 's1')
```

```
    rootnode = connectToInternet( net )

    h6 = net.get('h6')


    s1.cmd('./config_pools.sh &')
   # h1.cmd('ping -c 2 10.0.0.100 &')
   # h2.cmd('ping -c 2 10.0.0.100 &')
   # h3.cmd('ping -c 2 10.0.0.100 &')
   # h4.cmd('ping -c 2 10.0.0.100 &')
   # h5.cmd('ping -c 2 10.0.0.100 &')

    h1.cmd('iperf -s -u >> log1 &')
    h2.cmd('iperf -s -u >> log2 &')
    h3.cmd('iperf -s -u >> log3 &')
    h4.cmd('iperf -c 10.0.0.100 -b 1m -t 100 -i 5 -u >> log4 &')
    h5.cmd('iperf -c 10.0.0.100 -b 0.2m -t 100 -i 5 -u >> log5 &')
    h6.cmd('iperf -c 10.0.0.100 -b 5m -t 100 -i 5 -u >> log6 &')
    h7.cmd('iperf -c 10.0.0.100 -b 3m -t 100 -i 5 -u >> log7 &')


    CLI( net )
    # Shut down NAT
    stopNAT( rootnode )
    net.stop()
```

Script 4

```
"""
This script runs on the servers
```

```
and is used to create health beats using SIGAR API.
The health beats are sent at regular intervals of 5 seconds.

"""
package com.shreya.loadmonitor;

import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;
import java.net.UnknownHostException;
import java.util.Timer;
import java.util.TimerTask;

import org.hyperic.sigar.CpuPerc;
import org.hyperic.sigar.Mem;
import org.hyperic.sigar.NetInterfaceStat;
import org.hyperic.sigar.Sigar;
import org.hyperic.sigar.SigarException;

public class LoadMonitor {
public static void main(String[] args) throws
 IOException, SigarException {
final Sigar sigar = new Sigar();

final DatagramSocket clientSocket =
 new DatagramSocket();

try {
```

```
final InetAddress controllerIpAddr = InetAddress
.getByName("192.168.144.1");

System.out.println
(InetAddress.getLocalHost().getHostAddress());

Timer timer = new Timer();

timer.scheduleAtFixedRate(new TimerTask() {
@Override
public void run() {
try {
StringBuilder data = new
StringBuilder("health=>");
data.append
("ip:"+sigar.getNetInterfaceConfig().getAddress());
data.append(",");
CpuPerc[] cpus = sigar.getCpuPercList();
double totalCpuUsage = 0.0;
for (CpuPerc cpu : cpus) {
totalCpuUsage += cpu.getSys();
}
data.append
("cpuUsed:" + (totalCpuUsage / cpus.length));
data.append(",");

Mem mem = sigar.getMem();
data.append("memFree:"
```

```
+ (mem.getActualFree() / 1024 / 1024));
data.append(",");



double loadAvgMin1 =
sigar.getLoadAverage()[0];
data.append("loadAvg:" + loadAvgMin1);
data.append(",");

// DiskUsage disk =
sigar.getDiskUsage(name);
// data.append("disk:"+disk);
// data.append(",");

String[] netInterfaces =
sigar.getNetInterfaceList();

long totalRxBytes = 0;
NetInterfaceStat netInterfaceStat = null;
for (String netInterface : netInterfaces) {
netInterfaceStat = sigar
.getNetInterfaceStat(netInterface);
totalRxBytes +=
 netInterfaceStat.getRxBytes();
}
data.append("rxBytes:" + totalRxBytes);
```

```
data.append("|"); // end
System.out.println
("sending... " + data.toString());

byte[] dataToSend =
data.toString().getBytes();
DatagramPacket packetToSend =
 new DatagramPacket(dataToSend, data.length(),
controllerIpAddr, 8111);
clientSocket.send(packetToSend);
} catch (SigarException | IOException ex) {
ex.printStackTrace();
}
}
}, 0, 5 * 1000);
} catch (UnknownHostException e) {
clientSocket.close();
System.err.println("Unknown host...");
System.exit(1);
}
}
}
```