SOFTWARE DEFINED LOAD BALANCING OVER AN OPENFLOW-ENABLED NETWORK

by

DEEPAK VERMA

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN

COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT ARLINGTON

May 2017

Acknowledgements

I would like to express my sincere gratitude to my advisor, Prof. David Levine, for giving me the opportunity to work with him. I have learned a lot from you over the two years through various classes and research. Your support and guidance has been instrumental in completing this work. Thank you for pushing me when I needed it and challenging me intellectually.

I would also like to thank the other members of my research committee, Dr. Hao Che and Dr. Gautam Das, for their time spent reviewing my work. Dr. Hao Che provided me with guidance Dynamic Load balancing.

Also, I thank my parents. They have always been there to support me in life and have made me the person I am today. Thank you for always encouraging me to work harder and achieve more in life.

Throughout my academic career, I have had the support of numerous friends. I would like to thank all of you for your support and friendship over the years.

April 21, 2017

Abstract

SOFTWARE DEFINED LOAD BALANCING OVER AN

OPENFLOW-ENABLED NETWORK

Deepak Verma, MS

The University of Texas at Arlington, 2017

Supervising Professor: David Levine

In this modern age of the Internet, the amount of data flowing through networking channels has exploded exponentially. The network services and routing mechanism affect the scalability and performance of such networks. Software Defined Networks (SDN) is an upcoming network model which overcomes many challenges faced by traditional approaches. The basic principle of SDN is to separate the control plane and the data plane in network devices such as router and switches. This separation of concern allows a central controller to make the logical decisions by having an overall map of the network. SDN makes the network programmable and agile as the network application is abstracted from the lower level details of data forwarding. OpenFlow allows us to communicate with the data plane directly, gather traffic statistics from network devices and dynamically adjust the rules in OpenFlow enabled switches. Currently load balancers are implemented as specific hardware devices which are expensive, rigid and lead to a single point of failures in the whole network. I propose a software defined load balancing mechanism that increases efficiency by modifying the flow table rules via OpenFlow. This mechanism dynamically distributes the upcoming network traffic flows without disrupting existing connections.

Table of Contents

List of Illustrations

List of Tables

Chapter 1

Introduction

Computers communicate with each other via packets over a network. There are a lot of devices in the middle, which route the data from its source computer to its destination computer. As the computers are exponentially growing so is the networking backbone to handle the huge amounts of data generated by them. Traditional networking is rigid and error prone. A newer architecture called software defined networking has gained ground in the academic and business world, bringing a fresh set of ideas, overcoming a lot of the problems faced in the traditional networks.

The main goal of this thesis is to create a load balancer, which is written in the software and deployed over a SDN [1] compatible network. An algorithm is developed which uses resources from different components in a network to decide which server should next handle the upcoming packet. The algorithm is then tested against traditional load balancing [2] techniques on a virtualized testbed consisting of emulated servers, hosts and switches.

Traditional Architecture

The devices which take of routing packets like routers and switches at the lower level are all currently hardware based with the control logic being embedded in them. These are hardware appliances which are generally proprietary, vendor locked and thus costly. They use custom ASICs and FPGAs [3]. Each device needs to interact with its surrounding device to gain valuable information about decision making and packet forwarding.

Each device has two planes, one control and one data plane. The control plane is a software hardcoded on the hardware of the router or switch. The control planes use networking information to generate the forwarding table which is used by data plane to route packets accordingly. In the traditional way, the two planes are combined in one device itself. Thus, networks can’t be changed on the fly and the whole systems becomes rigid and static.

Software Defined Networking

SDN proposes a new architecture, one where the device only forwards the packet according to an action in its table, but the decision to whom to forward to is made at a central server. This separation is a departure from the traditional architecture. Due to this separation, we can abstract the complex tasks from the repetitive forwarding task. The complex task can be automated and managed separately in a centralized SDN controller [4] or Network Operating System [5]. These tasks can now even be programmed leading to a programmable network, which is flexible and agile. It gives the admin a centralized image of the whole network, allowing on the fly changes in the network without much manual interruption.

Motivation

Majority of the web companies on the internet use load balancer in their networks to handle incoming traffic. These are currently hardware based devices with the logic loaded on the firmware for eternity. These commercial devices use standard load balancing techniques like round robin [6] or least connections [7] leading to an inflexible solution which is costly to scale. SDN proposes software based devices which are virtualized and run on commodity x86 servers. It is cheaper to scale such a network and we can also try different approaches to load balancing over the same hardware. Motivated by the new opportunities which SDN provides, a dynamic load balancing [7] technique was designed as a software application over a SDN enabled network. The software based solution being cheaper than the hardware based is loaded in a server that also handles the controller.

Proposed Work

This thesis proposes a software defined dynamic load balancer which selects servers in real time depending on multiple attributes. The load balancer takes into consideration the CPU utilization of each individual server and periodically asks the switch for port statistics, which are used to calculate the average load on each port, the port speed and packet rate loss. Predefined weights are given to the servers simulating real life conditions where one server might be able to handle more connections than others. Different controllers were tested and tried but one Ryu [8] was chosen for building the load balancer on. Mininet [9] was used to emulate a network with all components being virtualized. Results were derived from experiments by running the dynamic technique against two other load balancers random and round balancer. The data prove the usability and better performance of dynamic load balancer over standard methods.

Organization of Thesis

Chapter 1 – Starts with the introduction and goals of the thesis.

Chapter 2 – The background concepts and underlying technologies are presented here which explain SDN operations and features. Different load balancing techniques and previous work relevant to it are discussed.

Chapter 3 – This chapter describes the implementation of the load balancing technique proposed for the thesis. It goes into detail about the architecture and used to setup and run the load balancer.

Chapter 4 – The setup used to test the algorithm is described in this chapter. The experiments and their results are analyzed and discussed upon.

Chapter 5 – It consists of a summary with the conclusion and future work

Chapter 2

Technical Background

SDN proposes a new architecture, which separates the network control and forwarding functions, this separation is a departure from the traditional architecture. Due to this separation, we can abstract the complex tasks from the repetitive forwarding task. The complex task can be automated and managed separately in a centralized SDN controller or Network Operating System. In this chapter, we introduce key aspects of OpenFlow [10] protocol and present the SDN architecture.

Software Defined Architecture

SDN abstracts [11] the forwarding functionality from the control plane. The controller uses OpenFlow [12] protocol to communicate with the actual physical or virtual switch. The data path uses flow entries present in the flow table inserted by controller for routing data. There are three ways of separating data planes and control planes: strictly centralized, logically-centralized and fully distributed. The switching devices have only one simple functionality of forwarding packets with no controlling power in the first case which can lead to single point of failures. In the second, devices have partial functionality embedded in them e.g. MAC learning [13] and the centralized controller handles higher functions [14]. Lastly, we have the traditional way of having nuclear devices which have both planes in them and need to cooperate with each other to route packets across the network.

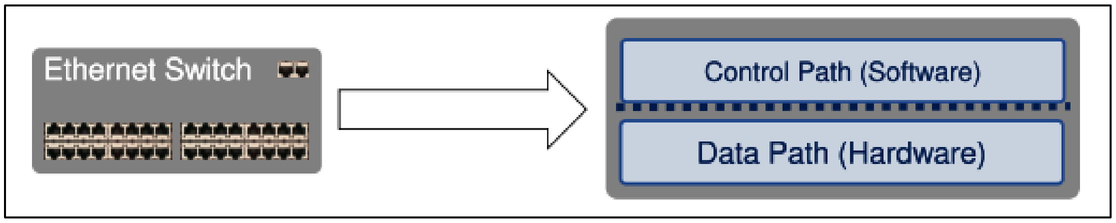


Figure 2-1 Switch Data and Control Plane

The location of the control plane or how wide can the separation be between control and data plane or can all control plane functions be relocated and how many instances would be needed to provide high-availability are still hot research topics.

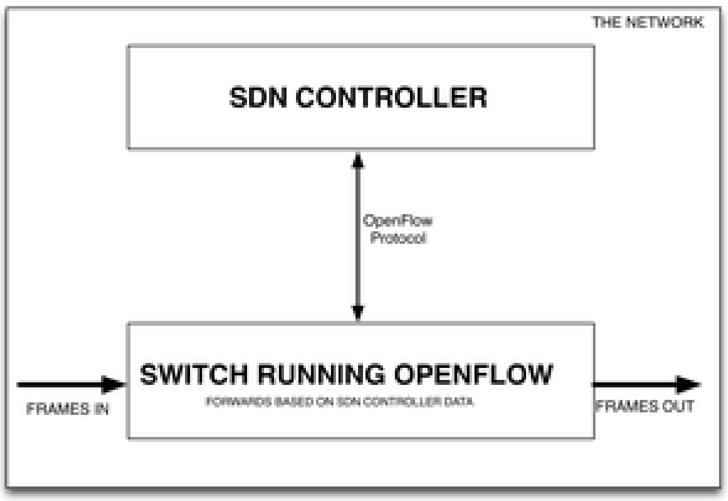


Figure 2-2 SDN conceptual design

The control plane is the manager which decides where each packet should go where. It communicates with the data plane to exchange management messages, update the routing table or talk to other control planes for network updates. The data plane has only one job of routing packets and is manufactured using different technologies e.g. field programmable gate arrays or application specific integrated circuits. If a packet’s destination address has no entry in the table, then the data plane forwards it to control plane for further processing.

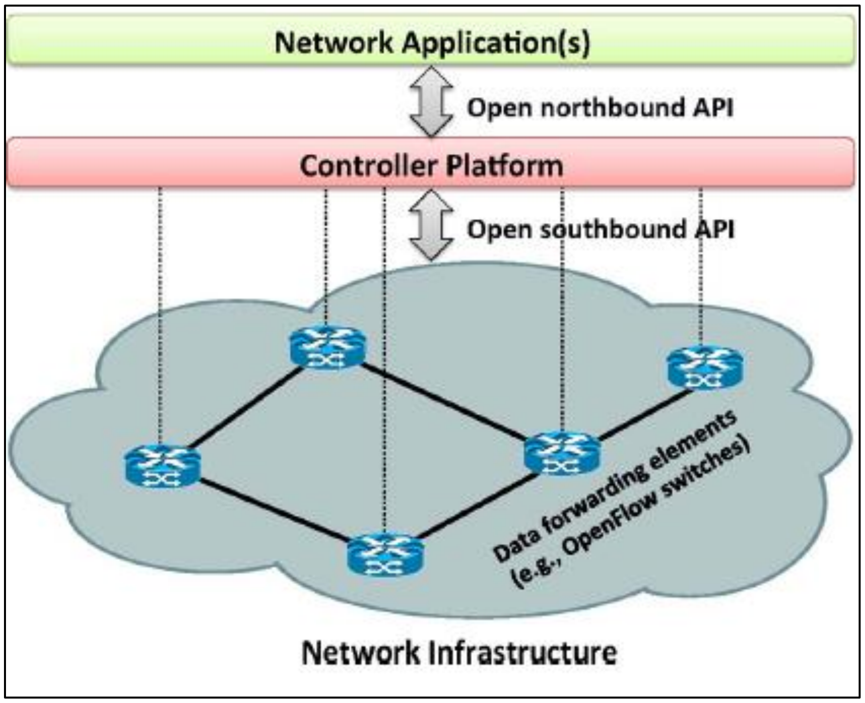


Figure 2-3 SDN infrastructure level view

Control plane:

The network brain which contains the basic logic and programs the forwarding plane using Southbound APIs. It allows interfacing with northbound API via REST calls.

Forwarding plane:

This plane is at the lowest level in the network infrastructure as it's the physical layer that forwards packet at line rate. It consists of the ASIC or FPGA with TCAM memory. The router does a fast lookup in its TCAM and forwards the packet from one port to port to another.

Northbound API

The network operating system offers APIs to programmers to probe the lower level working of a switch like packet information. This bidirectional communication between applications and control planes enable support for switching, routing, firewall, etc. as it abstracts the forwarding device.

Southbound API

The southbound API handles the communication between control plane and dumb router in a standard protocol such as OpenFlow. This south facing or low level interaction between control planes and forwarding planes is defined by Southbound APIs.

Management plane:

This plane contains the actual software applications that use functions in the Northbound API and provides variety of network functionalities e.g. load balancers, monitoring, routing.

Separation of concern

The idea of separating the functionalities has come into the minds of Network Engineers from time and again. There have been implementations by few companies, in the form of service card, but they were costly, not scalable and vendor specific.

Companies can provide a variety of services by changing just the code, independent of the underlying hardware. There is a boost to network automation as we can control multiple devices from a centralized plane thus reducing complexity and run templates or scripts to standardize the process across the network. Due to this abstraction of control and resulting network automation we can easily develop and deploy common services like load balancing, firewalls, multicasting, etc.

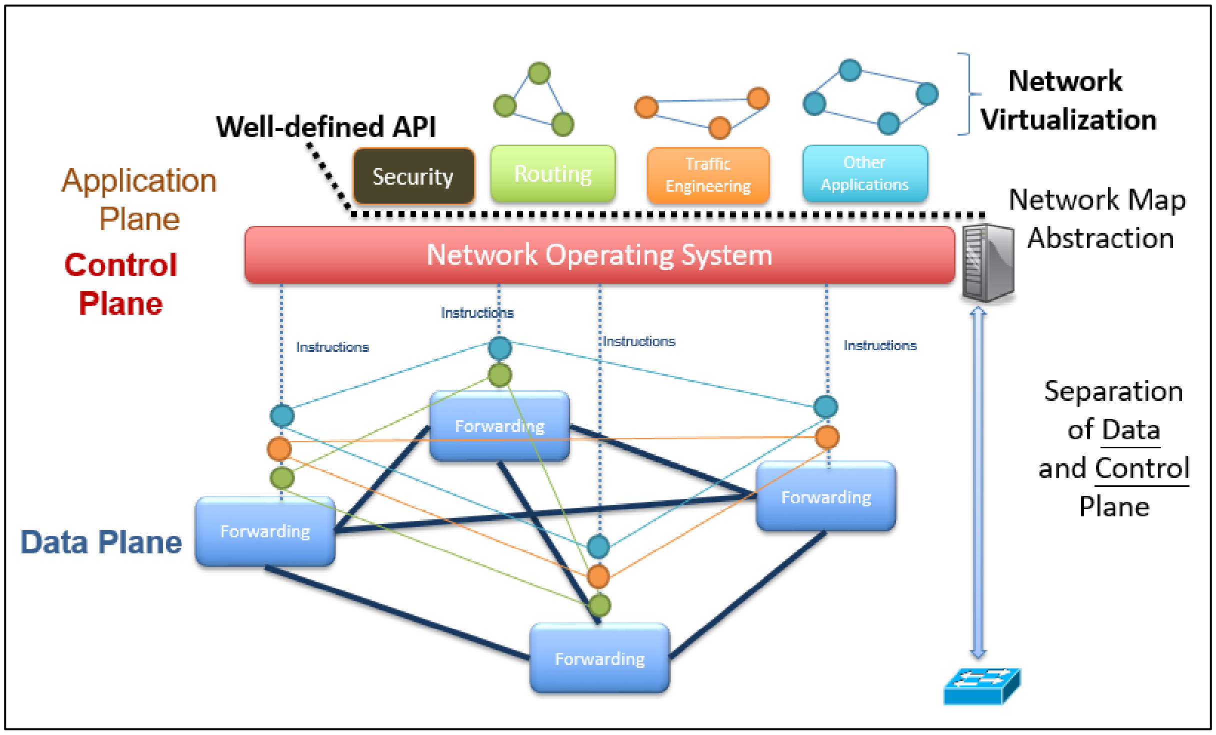


Figure 2-4 SDN Architecture

Networking companies can use commodity hardware e.g. x86 servers to build such data planes which would lower cost of manufacturing. By having an open network configuration protocol, we can bypass vendor lock in for routing software and firmware as we don't need to know different network commands for a variety of devices. As business requirements shift, the whole network can be updated easily without much interference from Admins. The controller can be replicated over networks which can share a global view point of the whole landscape.

SDN Protocol: OpenFlow

OpenFlow [12] has become the most famous standard for SDN forwarding abstraction and controller protocol. It was developed by engineers from Stanford university, which enabled them to run experiments over college networks without disturbing the current configurations. An OpenFlow enabled switch creates VLANs (Virtual Local Area Networks) in the existing production network, separating the experimental traffic. It is a standard specification [15] which describes how to interface centralized controllers and forwarding plane present in the devices. An OpenFlow enabled switch starts by contacting a remote controller over a TCP connection. Traffic decisions which are handled flow based are either added or removed in the device in the form of flow tables to control packet routes.

Flow Table

Flow tables are made up of prioritized rules. These rules are made up of a match condition and an action. A rule that includes values for all fields is called an exact rule and that places no condition on one or more fields is called a wildcard rule. The match condition decides which rule applies to a packet and then action defines how the packet should be handled. As packets arrive, the device matches the rules to the packets and which ever has the highest priority that action is chosen and the ones that match are processed at line rate. If no rules get matched than the packet is sent to the controller in the form an OpenFlow message. This notification, called a packet-in event, includes a prefix of the arriving packet contents (enough to include the common layer 2,3, and 4 packet headers) along with some extra information, such as the physical port on which the packet arrived. Having received this notification, the controller may decide to add a new rule to the flow table to handle this type of packet.

OpenFlow Flow entry

There are three basic parts to an OpenFlow flow entry. The rule is a field used to match the packet header with. An incoming packet is first matched from the in-port number, VLAN ID till TCP destination port. The user can define the flow entries via the controller which are than inserted in the TCAM. The packet encapsulates Ethernet, IP and TCP, etc. protocols. Each implementation of the controller for example Ryu or FloodLight [16] handle these encapsulations differently according to the language they are programmed in.

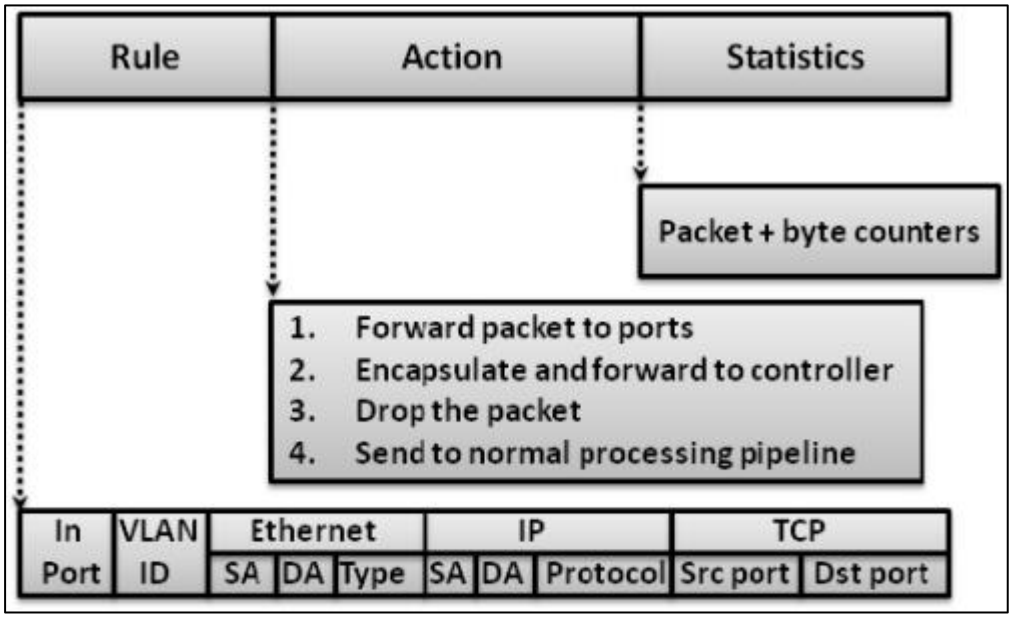


Figure 2-5 OpenFlow Flow table entry

The actions are defined as instructions by the SDN controller that must be followed by the switch when a packet matches the rule. OpenFlow also maintain counters for statistics. These are incremented by the compatible switch and keep a count of the number and size of packets passing through the network.

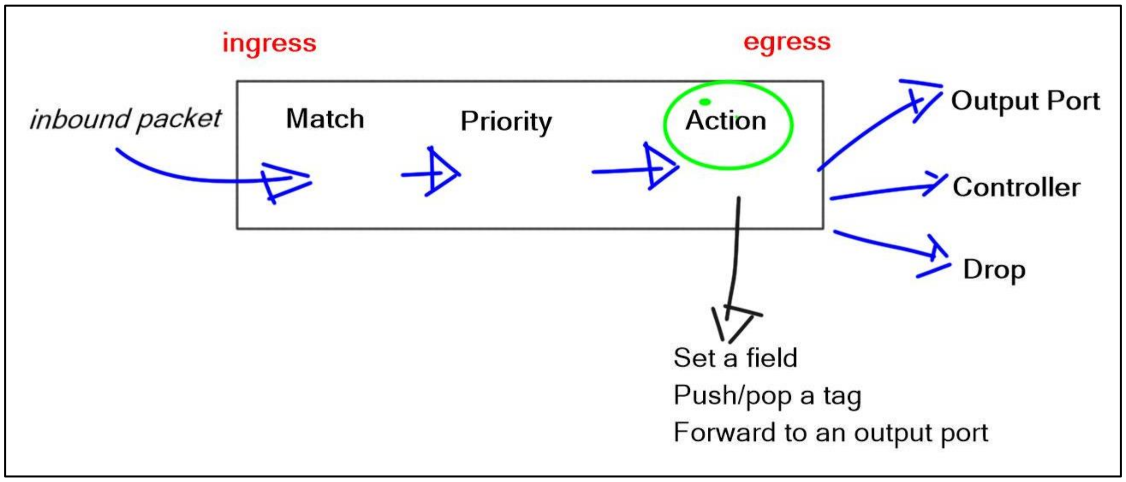


Figure 2-6 OpenFlow packet processing in switch

As a packet enters any port of the switch it goes thorough certain steps depending on the actions set in the flow table. The inbound packet is first matched to a field in the TCAM and then according to the highest priority matching a corresponding action is taken by the switch logic. The actions might be to set a field in the packet or push/pop a tag in the header. The switch might drop or forward the packet to another port as well. If the packet is not defined in the flow tables than it can be passed to the controller for further processing.

OpenFlow Message Types

The Openflow 1.3 specification [15] basically defines three types of messages: symmetric, asynchronous and controller-to-switch. The symmetric messages are of the following types:

1. Hello: exchanged at connection startup
2. Echo: determine latency and bandwidth of controller and switch connection
3. Experimenter: provide a standard way to offer additional functionalities or as a base for future revisions.

At specific events, asynchronous messages are sent by the switches. They are of the following types:

1. Packet-in: this message is used when a packet that has arrived does not match any entry defined in flow table and if switch supports buffering than it can be buffered.
2. Flow-removed: these are used by controllers to remove route flows from switches
3. Port-status: used during port configuration settings
4. Error: controller is notified by switches for error

The controller can directly send messages to the switch to manage its state but may not require a response from switch, such messages are called controller-to-switch and few are summarized below:

1. Features: the controller understands the capabilities of the switch and the switch responds with a feature reply.
2. Configuration: this type is used by the controller to change the configuration parameters in the switch
3. Packet-out: Its used to send packets out of a port on the switch. The message must contain a list of actions that should be applied in a defined order.

OpenFlow Connections

An OpenFlow controller manages many OpenFlow channels simultaneously to multiple switches. It can also have multiple channels to a single switch as well for reliability purpose. The switch connects to the controller at a fixed IP and TCP port 6633 using a standard TCP or TLS connection. There is an exchange of certificates between the two devices at the start. The switch differentiates between the messages present in the OpenFlow channel and the regular internet traffic.

An OFPT_HELLO message containing OpenFlow version is exchanged between the controller and switch when a connection is established. If the version doesn’t match than lower of the two is negotiated as the primary version for all future conversations, else the connection is terminated if no common version is supported by either device.

In case there is an interruption in the connection between the switch and controller, the switch must immediately enter one of the following modes. It can drop packets meant for the controller or act as a legacy switch.

Multiple Controllers

A single switch may establish connection with multiple controllers as it increases reliability during failures. The controllers manage the hand-overs between them and provide controller load balancing. The controller can act in different roles with respect to a switch. All the controllers can be in Equal state or at most one controller can be in a Master state and rest be in Slave state. In slave state the controller has read access only while Master state is same as Equal state with full write access to the switch. The switch should remember all the states of the controllers as controllers can change role anytime.

OpenFlow Versions

OpenFlow has an active development lifecycle and is continuously upgraded with newer extensions. Features such as MPLS, Flow monitoring and Egress table have been introduced slowly in the master branch.

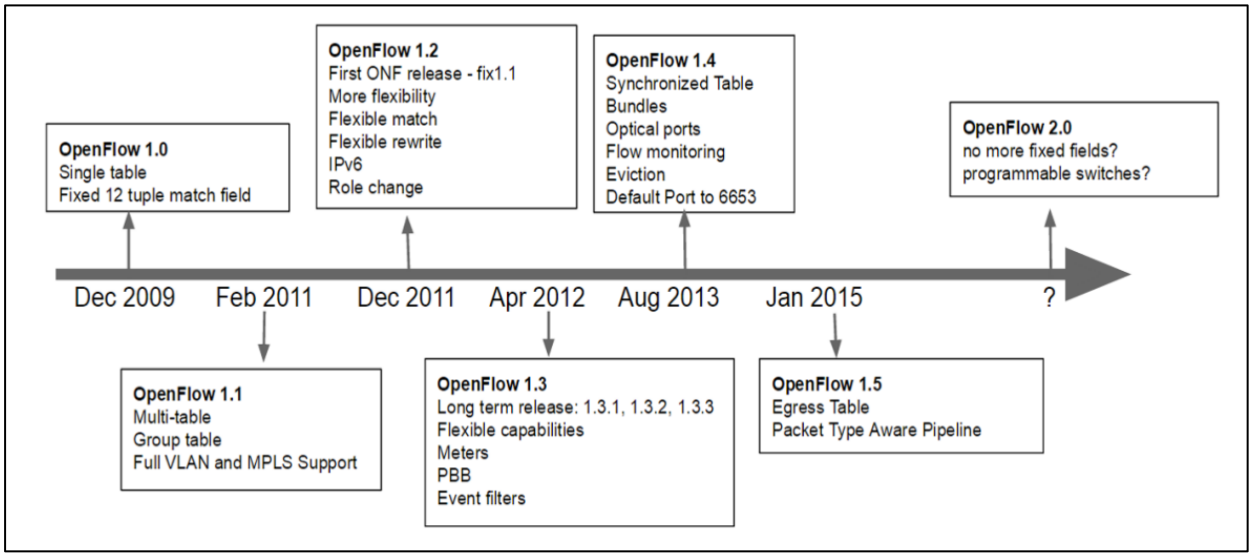


Figure 2-7 OpenFlow version history

SDN Switch

An OpenFlow switch also maintains various traffic statistics, including per-port byte and packet counters and per-flow table entry byte and packet counters. In addition, each flow table entry can be configured with both hard and soft timeouts. Because of limited TCAM capacity, it is important for network controllers to use flow table capacity efficiently. If the network controller requires more rules for its forwarding policy than the available capacity, it will need to handle packets for some flows at the controller.

The first OpenFlow enabled switch is Open vSwitch(OVS). It's purely a software based implementation which can use OpenFlow protocol to transmit flow table data from controller to the switching hardware. It allows programmatic control and vendor independent management interfaces. It can be used to manage networking for VMs on same physical host or forward traffic to VMs on different physical hosts. It runs on a variety of Unix based virtualization platform like KVM, Virtualbox and Xen.

Open vSwitch

The developers of OpenFlow were involved in creating OVS [17] and thus resulted in an open source, multi-platform virtual switch. It's an alternative to Linux native bridges and VLAN interfaces. It supports mostly all the features that a hardware switch has such as bridge, VXLANS, QoS, SPAN, tunneling, ACL, etc. It is tailored to work in virtualized environments. Open vSwitch supports Rapid Spanning Tree protocol to prevent loops in Ethernet LANs. It implements flow caching to reduce the consumption of hypervisor resources.

There are two components inside Open vSwitch, one acts as the management layer called ovs-vswitchd which is on top and second is the OVS kernel module which is in the bottom and acts as the forwarding plane. Ovs-vswitchd is in the user space and interacts with the user and gathers packet information and flow routes while kernel module talks to the NIC or VM's virtual NIC for collecting packets and is in the kernel space.

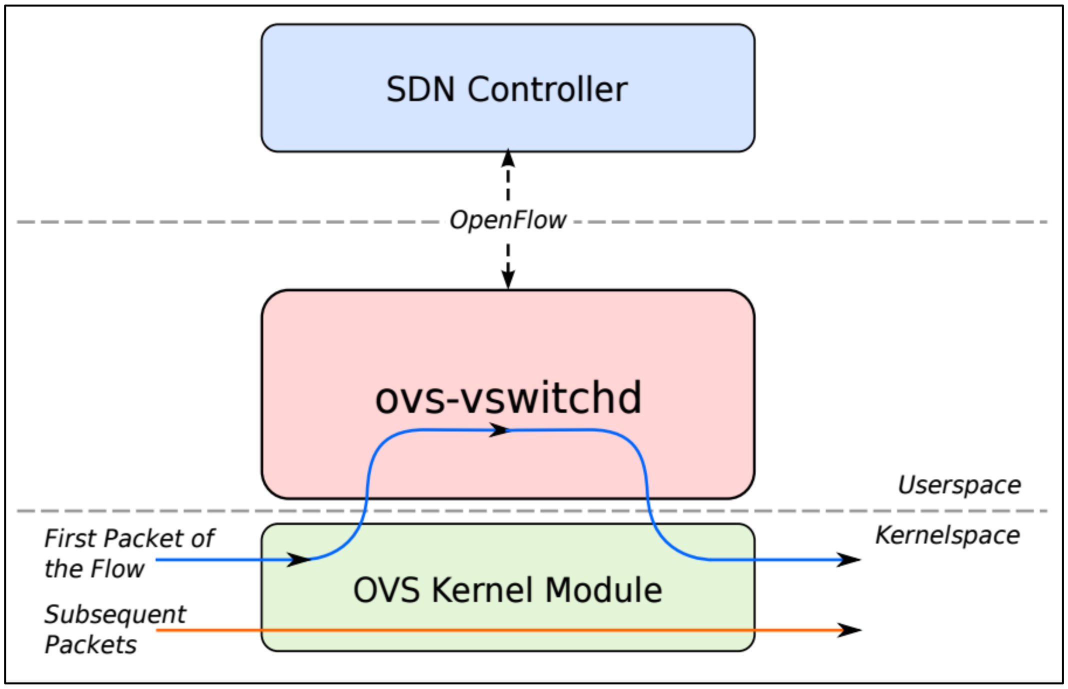


Figure 2-8 Open vSwitch design

The user daemon is a common application in all operating environments while the kernel module is written specific to an operating system for speed purposes. When a packet arrives, it is passed from the NIC to the kernel module for processing. The kernel module will simply follow the rules setup by ovs-vswitchd in the upper layer. If it has been instructed to be dropped than the packet would be simply dropped else some action defined by the datapath is taken. Actions are the instructions passed on by the datapath to the kernel module. If there is no action defined or if it’s a new packet, then it is pushed up the layers to the ovs-vswitchd from the kernel module. Actions with corresponding ports are defined by ovs-vswitchd which is used by the kernel module to transfer packets on. Actions can vary, such as packet should be dropped, modified or sampled. The kernel module caches the actions for handling similar packets or routes in the future. The SDN controller transfers the flow tables to the upper layer of OVS via OpenFlow protocol and then the ovs-vswitchd would instruct the lower layer kernel module on how to process incoming packets. Thus, the kernel space component is abstracted from the user level controller and application as the ovs-vswitchd acts as the interface in-between. The kernel module is unaware of the controller and OpenFlow protocol while the SDN controller view is hidden from lower level implementations of packet forwarding. The flow table is programmed in the form of a hash table, details of which are beyond the scope of this thesis. Open vSwitch has become the most popular virtual switch in the industry for software defined networking in data centers due to its minimalistic CPU utilization, easy deployment and advanced flow classification features. It utilizes multi core environments with multithreading enabled and there are forthcoming ports for interfacing it directly with the lower hardware bypassing the kernel, thus giving greater speed gains. It cannot reach the throughputs provided by ASICS and FPGAs but can provide greater flexibility with more functionalities at a cheaper price, thus can be combined with legacy hardware devices for real world environments for greater gains.

SDN Controllers

The controller or network operating systems is the central manager which uses OpenFlow protocol to interact with the forwarding plane handled by switches. it can manage, control and administrate the flow tables. These controllers are written in software thus provide flexibility and dynamism but lesser speed compared to controllers in hardware. The flow tables can be inserted with rules beforehand (pro-active) or after a packet arrives (react-active).

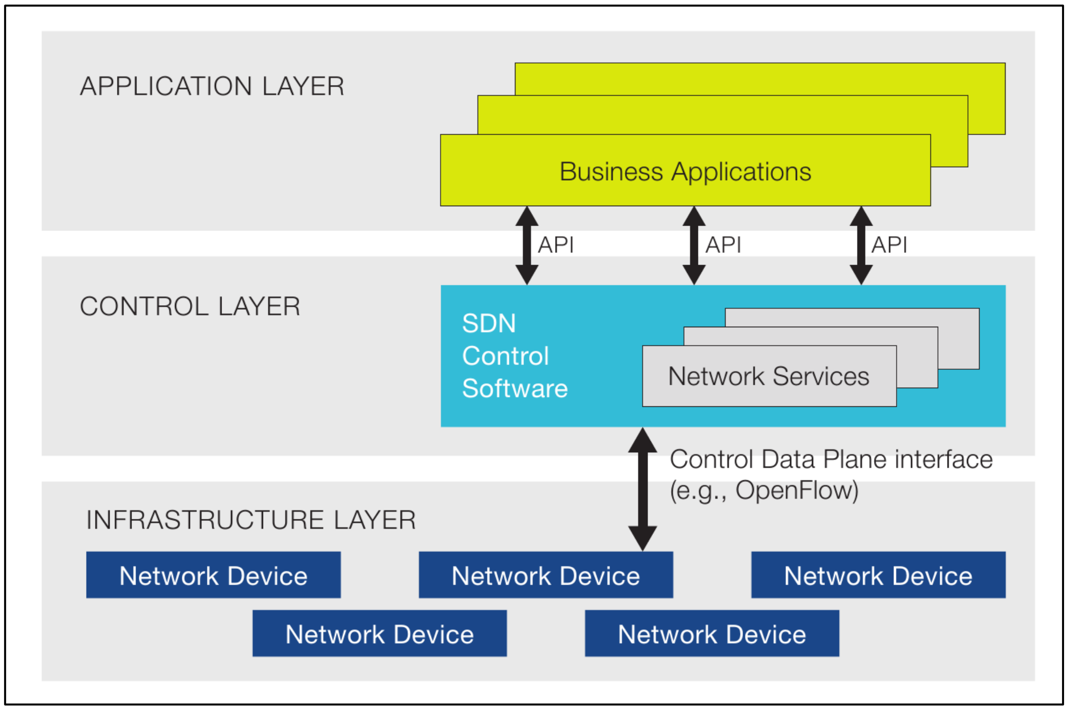


Figure 2-9 SDN high level design

There has been development of a few robust controllers which are basically software’s that command and control the OpenFlow enabled switch. Few popular open source controllers are Pox [18], Nox [19] [20], OpenDayLight [21], FloodLight, Ryu and Cisco APIC [22], HP VAN [23], VMware NSX [24] are some popular commercial ones. All controllers come with a GUI and REST API interfaces for connection with the user space and provide commands for enabling and setting routes via OpenFlow protocol. 5 different open source controllers were looked into to decide the best fit for this project. Certain controllers have gained popularity with industry backing and few have lost support from the community and become obsolete. There is either reduced or no active development going on with Pox and Nox controllers. These two controllers also don’t support the latest OpenFlow protocol versions. The controllers are bare bone appliances with a small number of basic functionalities implemented in them for interfacing with user level applications. They are majorly used for academic purpose to introduce students to SDN technologies due to their reduced feature set and low learning curve.

Table 2-1 SDN Controller Comparison

Name	Language	Developer	Learning Curve	Industrial Usage
OpenDayLight	Java	ONF	High	Yes
FloodLight	Java	Big Switch Network	Medium	Yes
Ryu	Python	Ryu Community/NTT	Medium	Yes
NOX	C++	Nicira	Low	No
POX	Python	Nicira	Low	No

On the other spectrum are FloodLight, OpenDaylight and Ryu SDN controllers. OpenDayLight is the most popular out of these three controllers and the most battle tested. It is multi-protocol and is built using the Java OSGi framework giving it a modular framework. This allows it to be updated during run time unlike Ryu controller which must be shut down and restarted to load a new or an updated REST application. OpenDayLight SDN controller currently has the biggest feature set of all open source controllers resulting in a steep learning curve and a high startup time.

During the thesis, smaller applications were developed for each of the 5 types of controllers to get familiar with the feature set and friendliness of the programming environment. Nox and Pox were rejected due to their reduced community support, documentation. A lot of boilerplate code would be required to develop the load balancer application, shifting the focus from the thesis. OpenDayLight controller isn’t taken into consideration due to its steep learning curve and complex feature sets. Though both controllers, FloodLight and Ryu, provide nearly the same features but Ryu is selected as its programmed in Python while the former in Java which reduces time in prototyping. Ryu has a medium learning curve and good availability of documentation with an active community keeping it compatible with latest OpenFlow protocol versions.

Ryu

Ryu is one of the most popular controllers in the industry with NTT labs support and an active community with in depth documentation. Ryu is entirely written in Python and uses green threads to implement multithreading. It provides software components that allow software developers to extend network management and control applications to utilize SDN controller features. it currently supports variety of southbound protocols such as OpenFlow, OF-Config, NETCONF, etc.

It supports a lot of third party libraries that include Open vSwitch Python bindings, help parse VLAN, MPLS, etc packets of various protocols. It supports sFlow and Netflow protocols which are specific to network traffic management. Ryu supports up to OpenFlow version 1.4. It uses an encoder and decoder library for OpenFlow packets and contains the OpenFlow controller at

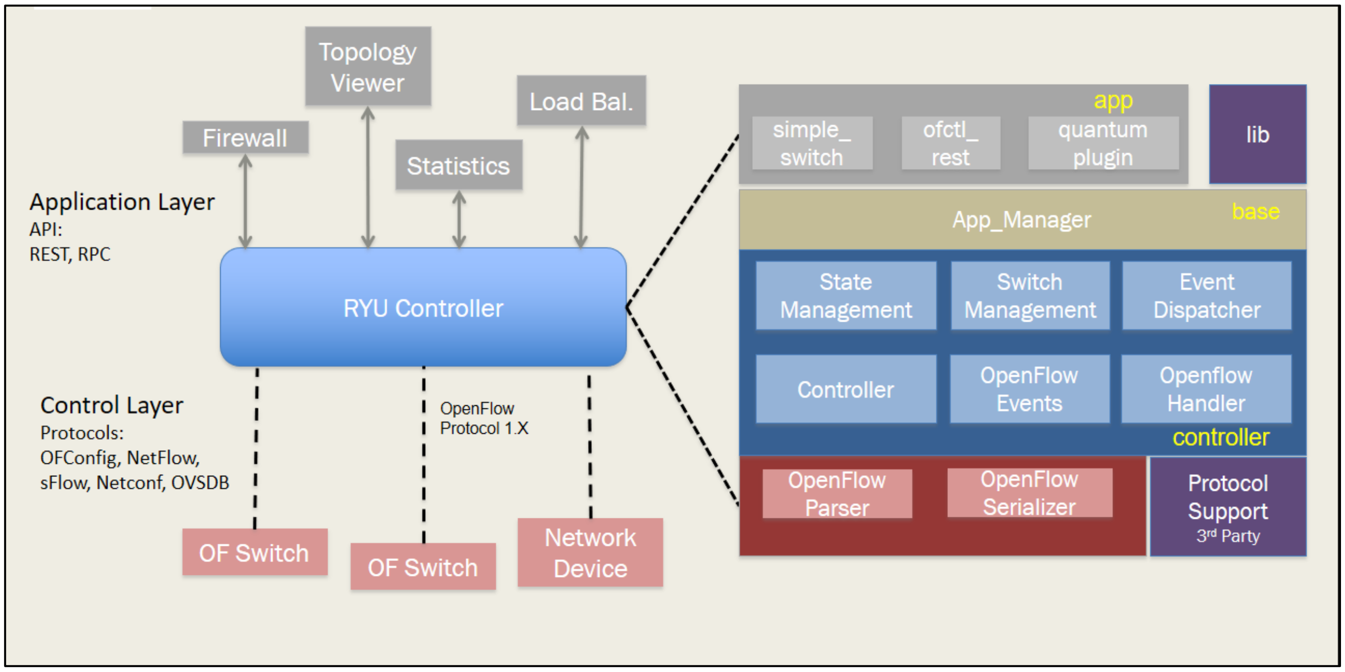


Figure 2-10 Ryu Architecture

its heart. It consists of a main executable called Ryu manager where it listens to a specific IP address and on port 6633, the standard OpenFlow port. The core components in the Ryu architecture include messaging service, event management, etc. Ryu supports Openstack [25] Neutron plugins and exposes REST API for its lower level OpenFlow operations.

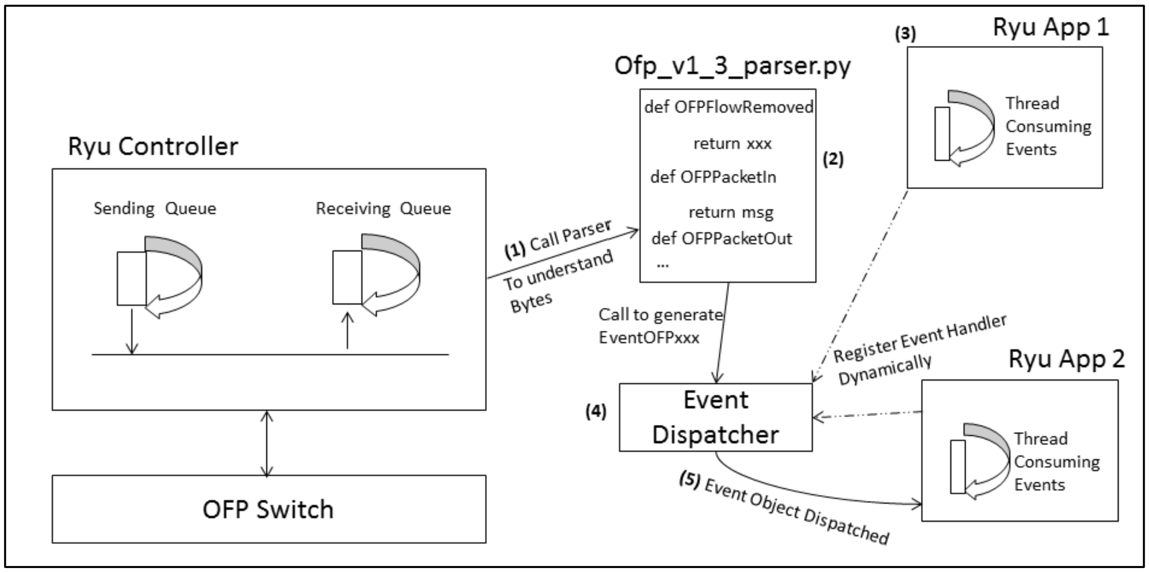


Figure 2-11 Ryu Event Handling

Ryu utilizes thread pools to take care of incoming packets. Asynchronous events are sent by Ryu applications which use a receive queue for these events. FIFO is used to maintain the order of incoming events and threads are used to handle the events. In the main loop of the thread, an event is first pops out of the queue and call the corresponding event handler. As the event handler is called by the thread it comes within its context or local block, causing it to work in a blocking fashion. Due to this behavior that event handler won’t be able to handle a new packet as its blocking the thread.

Load Balancing Techniques

The primary aim of a load balancer is to spread incoming traffic such that no single server take all the requests and starts slowing down or hog's memory. There are different parameters according to which load balancing techniques can be classified. One of the popular one is content awareness. If the load balancer algorithm takes into consideration what type of data is flowing through its channels than it is considered as being content aware else, it's a content blind.

Content Aware Load Balancing techniques

The load balancer can be developed to specifically cater to certain types of data stream such as banking, gaming, website loading.

IP Hash

The IP address of the client and server are combined in a specific way to generate a unique hash key. The key generated is used to allocate the incoming request a server. During a connection breakup, the key can be regenerated and so the client will be directed to the same server.

DNS

There are multiple ways of DNS load balancing. One method is called DNS delegation. We delegate a domain as a sub-domain to zones that are served by a group of servers serving the same web site. Even if the servers are geographically spread on the internet the web site will be able to handle loss of servers and distribute incoming requests better.

Persistence

This method of load balancing is particularly used in banking and security related companies as the HTTP connections shouldn't get terminated. The session should be maintained between the server and client as there might be banking operation going on and endpoints can get exposed to an attacker or can be spoofed. A specific type of traffic will always to passed through the same channel persistently till the session ends using the source and destination IP addresses.

Content Blind Load Balancing techniques

If the data passing through the network is generalized that content blind techniques can be effective and inexpensive. It might not give the level of persistence as required in some instances but might provide more speed or react faster to a new request.

Latency/ Weighted Response Time

This type of load balancer is popular in the gaming industries. The gaming server which has the fastest response time will be given the next connection. In this manner during online gaming the application won't show a lag due to connection setup.

Round Robin / Weighted Round Robin

A simple algorithm in which the server IP addresses are used in a rotating sequential manner. This can be modified to include weights where server with higher ratings will get more requests sent to them.

Least Connection / Weighted Least Connection

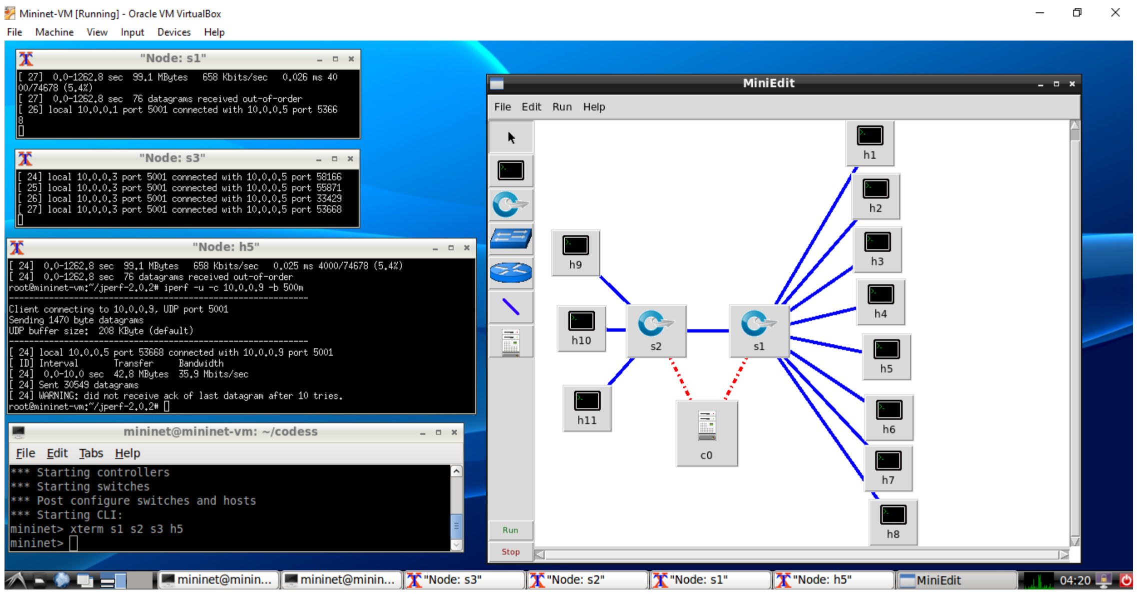
As the above-mentioned load balancers, don't take server load into account, we use the least connection method. An incoming request is directed to the server that has the least number of active connection at that time. It can be built upon by specifying a weight to each server so that if two servers have the same number of connections than the one with ore rating would handle the next request.

Adaptive/ Dynamic

This technique involves the most innovation as each server will report its current systems stats to the load balancer via an agent that is installed in it. Thus, the load balancer can either use historic data or current values and in conjunction with the previous discussed algorithms or some other innovative method like machine learning come up with the IP address of the server that should be next in handling an incoming request.

SDN Emulator: Mininet

Mininet is a network emulator written in Python and C which create virtual hosts, switches and links. It supports multiple software based OpenFlow switches, custom topologies and provides a Python API for programmability of the network. The virtual components utilize the actual Linux



"Node: s1"

```
[ 27]  0.0-1262.8 sec  99.1 MBytes   658 Kbits/sec   0.026 ms 40
00/74678 (5.4%)
[ 27]  0.0-1262.8 sec  76 datagrams received out-of-order
[ 26] local 10.0.0.1 port 5001 connected with 10.0.0.5 port 5366
8
```

"Node: s3"

```
[ 24] local 10.0.0.3 port 5001 connected with 10.0.0.5 port 58166
[ 25] local 10.0.0.3 port 5001 connected with 10.0.0.5 port 55871
[ 26] local 10.0.0.3 port 5001 connected with 10.0.0.5 port 33429
[ 27] local 10.0.0.3 port 5001 connected with 10.0.0.5 port 53668
```

"Node: h5"

```
[ 24]  0.0-1262.8 sec  99.1 MBytes   658 Kbits/sec   0.025 ms 4000/74678 (5.4%)
[ 24]  0.0-1262.8 sec  76 datagrams received out-of-order
root@mininet-vm:~/jperf-2.0.2# iperf -u -c 10.0.0.9 -b 500m
------------------------------------------------------------
Client connecting to 10.0.0.9, UDP port 5001
Sending 1470 byte datagrams
UDP buffer size:  208 KByte (default)
------------------------------------------------------------
[ 24] local 10.0.0.5 port 53668 connected with 10.0.0.9 port 5001
[ ID] Interval       Transfer     Bandwidth
[ 24]  0.0-10.0 sec  42.8 MBytes  35.9 Mbits/sec
[ 24] Sent 30549 datagrams
[ 24] WARNING: did not receive ack of last datagram after 10 tries.
root@mininet-vm:~/jperf-2.0.2#
```

mininet@mininet-vm: ~/codess

File Edit Tabs Help

```
*** Starting controllers
*** Starting switches
*** Post configure switches and hosts
*** Starting CLI:
mininet> xterm s1 s2 s3 h5
mininet>
```

Figure 2-12 Mininet VM

network applications including the kernel and network stack for emulating devices. This allows the topology to be easily moved to an actual system for real world experimentation.

Mininet uses process based virtualization to run switches and hosts inside the operating system. It requires Linux kernel support for network namespaces which provides each process with its own separate routing and ARP table, network interfaces. Hosts are emulated as bash processes on which any code be run like in a normal Linux shell. A virtual Ethernet (veth) is used to connect to all the networking device. Mininet as an application can be either installed on an existing Linux system or it can be downloaded in a prepackaged VM which runs on VirtualBox [26].

Network Benchmarking Tool

Iperf

Iperf [27] widely-used network testing tool, it creates TCP and UDP data streams and measures the throughput, bandwidth and the quality of a network link. Both client and server functionalities are available and can be used to measure the throughput between the two ends in both unidirectional and bidirectional connections. It supports tuning of different parameter related to buffers, protocols and timing. It measures packet loss, delay jitter, etc. and support multiple simultaneous connections.

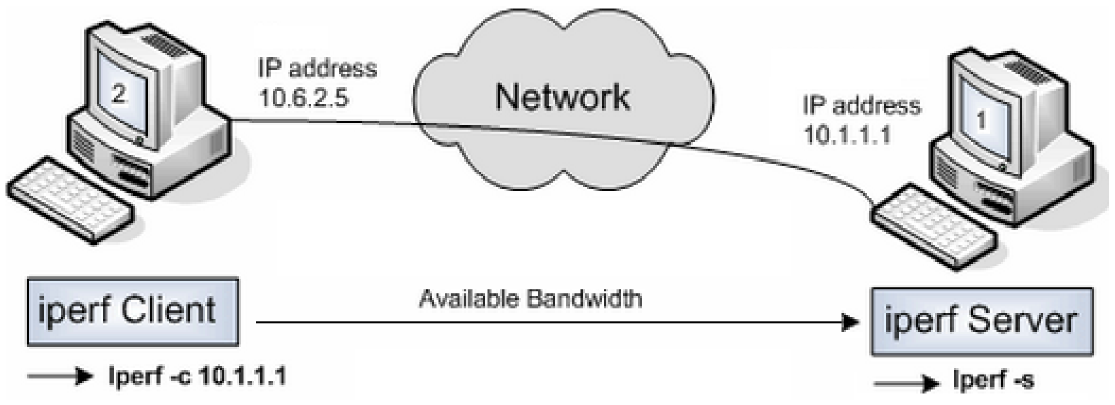


Figure 2-13 Iperf Flow

ApacheBench

ApacheBench or ab [28] is a command line utility used to for performance testing of HTTP web servers. It is like Apache JMeter with a reduced feature set. Ab supports concurrent and serial request creations and has a small memory footprint.

Related Work

OpenFlow has been used by researchers in a lot of different scenarios from implementing network functions to network debuggers. OpenFlow allows the developer to use a variety of software methods in traditional networks which isn't possible earlier due to hardware restrictions. Few novel ideas include the following:

1. VM migration [29] made easy across subnets [30]
2. Deep Packet Inspection [31] with Machine Learning
3. Load balancing applications specific to services like email, webservers [32]
4. Security honeypots [33], Firewalls
5. Path Optimization in real time [34]
6. Energy Efficiency [35] by sleeping nodes, links

7. Use switch diversity with SDN and legacy nodes [36]

8. Fabric layer hides vendor diversity

For the purpose of this thesis, the research domain is restricted to innovations in load balancer techniques enabled due to OpenFlow and SDN.

Handigol et al. [37] proposed a novel use of OpenFlow in implementation of a load balancer [6]. Plug-n-server minimizes the response time by managing the load on the network paths and servers using customized flow routing. The architecture contains the host manager which gets feedback from the servers and a net manager that gathers the network statistics from different components in the network. The method suggested is a reactive solution having some scalability issues.

Richard Wang et.al in [38] proposed an innovative technique to proactively map certain fixed blocks of source IP address to load servers using OpenFlow wildcard rules. In the method clients request, would then be automatically without much delay be forwarded to certain server via the SDN switch. The author made assumptions that the flow through the network would be uniform and in the range of IP address defined which cannot be accurately represent real world traffic.

Qingwei Du et.al [39] in have used a third-party tool called sFlow [40] to measure certain network parameters that avoided the use of OpenFlow flow statistics counter. The architecture uses a layer approach. The load balancing logic is on top of the network management layer and floodlight SDN controller in the bottom. The sFLow server is a separate instance which connects with the load balancer application via REST calls and there are sFlow agents in each client machines. These agent use packet sampling methods to aggregate statistics. The agents can export physical and virtual server performance, memory, disk and network IO performance. The agent can also be installed in the switch to gather data in real time about the packets flowing irrespective of the flows and routes saved in the switch. The data collected is then directed towards the server using sFlow protocol and then passed to the controller for load balancing use. The experimental results in the paper show better CPU utilization per server but a decrease in response time for dynamic load balancing as the algorithm reacts slowly compared to random and round robin algorithms.

Chapter 3

Design and Implementations

Setup

The experiment is performed in a virtual environment with all the devices being emulated. The actual hardware used to perform the experiment consisted of Dell Inspiron laptop with I7 5500 CPU, 8 GB RAM. Virtual Box is used to virtualize the environment for loading a Mininet 2 image. A 64bit image is used to create a VM in Virtual Box with 2GB RAM and 20GB HDD. The Mininet image is a modified Ubuntu platform with a variety of network tools installed and preconfigured including Mininet, Open vSwitch, Pox, Wireshark [41].

Ryu is installed as part of the SDN controllers to manage the OpenFlow compatible switch. Open vSwitch comes with utilities to configure it via command line. The ovs-ofctl program is used to directly manage the flow tables and entries inserted into the switch. It is used for monitoring and administration of the OpenFlow switch. The flow table can be deleted, viewed and inserted into using the command line utility.

Deployment

Once all the applications have been setup we must deploy the load balancer application in the Mininet VM. Before running Ryu, we must deploy the network and create the virtual devices which will emulate the servers and hosts. Mininet provides Python libraries and command line tools to initiate the network topology. For this thesis, a single switch is used with 3 servers and 8 hosts. The OpenFlow version 1.3 is used for this experiment as the versions above it were still in the starting phase and applications based on OpenFlow which included Mininet and Ryu haven’t implemented all the features introduced in it. After Mininet created the topology few properties have to be configured. First the ARP functionality has to be enabled in Mininet so that MAC addresses get populated in each host table in order of their creation. Enabling the ARP functionality would help reduce ARP packets flooding the network. Second property is the protocol version in Open vSwitch which is configured using ovs-vsctl command. Third property is the flow table of Open vSwitch that has to be deleted after each experiment to clear and reset the switch memory.

In the topology, the switch sits in between the hosts and servers and is connected to the Ryu controller using OpenFlow protocol over port 6633 by default. The Ryu controller is started with a specific load balancer application for each new experiment.

Design

The load balancer doesn't just distribute the incoming requests but it also acts as a wall in between the server pool and clients. The clients should not know the identity of the server that is servicing its request or that any server is down or the number of servers' present.

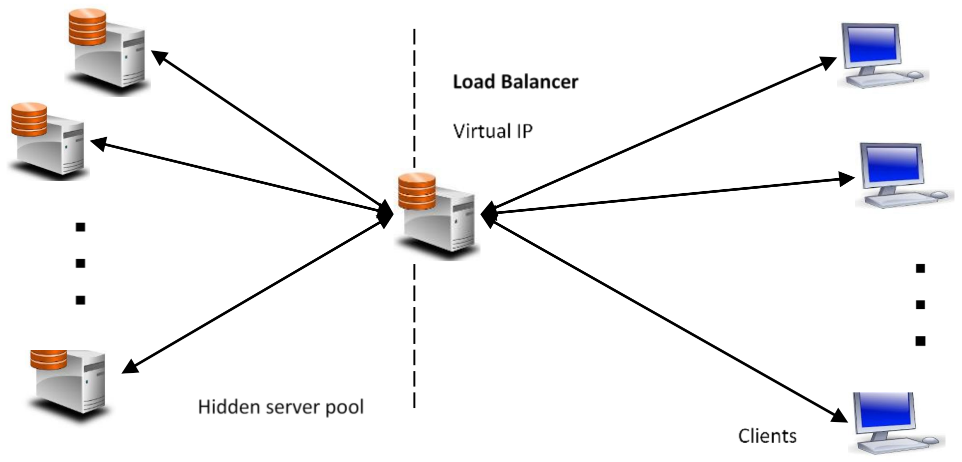


Figure 3-1 Load Balancer as a Mirror

Thus, it's the load balancer's job to act as the middle man that hides the servers behind it and spoofs the identity of the servers.

A virtual IP and MAC are specified as the address of the load balancer and is not used in the network for any other device. In the network the clients will connect to this IP and MAC, believing it's the website's server address that is answering the requests. The servers are assigned IPs addresses that are not exposed to the clients and there might be more than one load balancer behind a virtual IP.

The interaction flow of a client and server through the load balancer device can be understood in the following steps:

For a first-time TCP/UDP connection:

1. The client creates a TCP request and uses the virtual IP and MAC as the destination address and the source address as its own IP and MAC.
2. The switch present at the load balancer site will receive it and because the packet doesn’t match any flow entry it is forwarded to the SDN controller. The controller will parse the packet and extract the source IP and MAC and the in port. According to the algorithm used for load balancing, one server will be chosen to handle the connection.
3. For that flow a forward action (client to server) will be inserted in the switch that modifies the packet header and changes the destination IP and MAC to that of the selected server. Than it is simply given to the switch to transmit it to that server. Also for the flow in reverse, another (reverse) action (server to client) will be inserted that changes the source IP and MAC to that of the load balancer (virtual IP and MAC).
4. The selected server will receive the packet and take some action than generate a reply/response packet with the destination IP and MAC as the source IP and MAC of the packet that was received. This makes sure that the packet goes to the client from where the request originated.
5. The reply packet would flow through the switch and the in port would match that of the server while the out port would match that of the client machine so a reverse action would be created by the switch. The packet header will be edited and the source IP and MAC will be changed to that of the load balancer and put back in the network.

6. To the client which started the connection it would look like the same server (load balancer) has replied, though the actual server addresses has been spoofed.

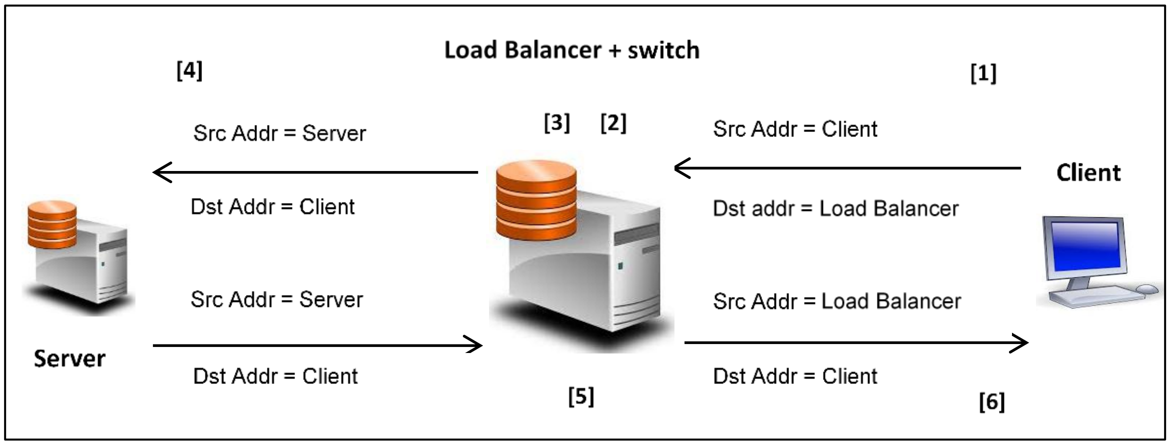


Figure 3-2 TCP Packet Flow via Load Balancer

For successive TCP/UDP connections:

After the first encounter the SDN controller would have inserted the entries into the switch flow table. From now on the switch would match the packet with a flow and priority and take the corresponding action at line rate speed just like a legacy switch.

For a first-time ARP connection:

1. The client creates an ARP packet and uses the virtual MAC as the destination address and the source address as its own MAC.
2. The switch present at the load balancer site will receive it and because the packet doesn't match any flow entry it is forwarded to the SDN controller. The controller will parse the packet and extract the source MAC and the in port.
3. An ARP reply is generated in the controller code with the source address as load balancers and destination as client address and passed to the switch which transmits it back to the client. For the future, a flow is created (load balancer to client) and inserted in the switch that modifies the packet header and changes the destination MAC to that

of the client and source MAC to load balancers. So, further ARP requests from the same client on that in port will automatically be replied to by the switch at line rate speed.

4. To the client which started the connection it would look like the server (load balancer) has replied with its MAC.

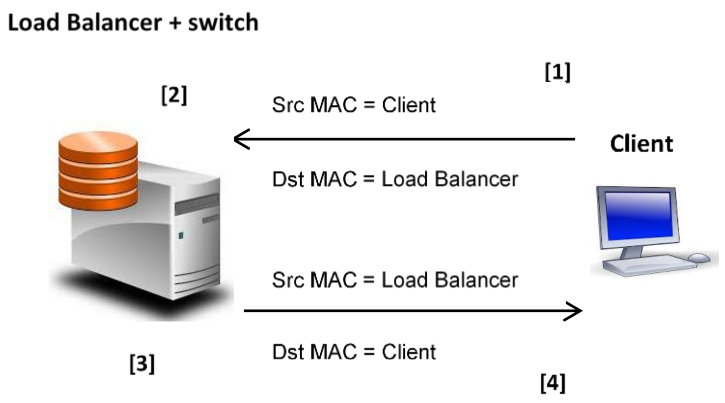


Figure 3-3 ARP packet flow via a load balancer

For successive ARP connections:

After the first encounter, the controller inserts the flow table entries into the switch so the switch would match the packet with a flow and according to priority take the corresponding action at line rate speed just like a legacy switch.

Implementation

The load balancer application is a Python code which imports libraries and methods provided by Ryu controller. The three load balancing applications utilize few common libraries from the Ryu application manager. These libraries enable an event such as packet arrival to be handled and pass control to its respective handler function.

At the lower level once the switch receives a packet if there is no flow entry which matches the packet than it is pushed to the controller else it will follow the action described in the flow action field. In the user level where the Ryu load balancing application is defined the packet is parsed or peeled off to reveal its data. If a packet is received in the SDN controller than either it’s a new connection or something that couldn’t be handled by the switch.

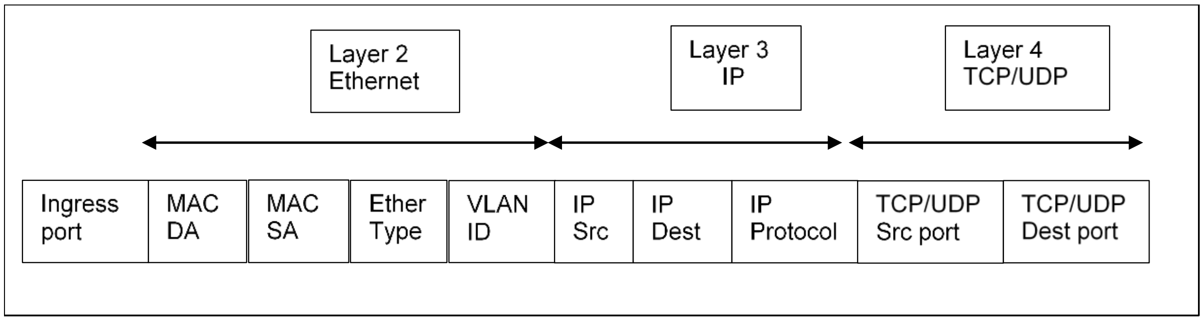


Figure 3-4 TCP/UDP simplified packet header

An incoming packet needs to be decoded and validated before the controller takes any decision and so a packet handler function is common to all load balancing applications.

The EventOFPPacketIn class describes the reception of OpenFlow messages from the connected switches. This event class contains the decoding functionality of an OpenFlow packet. The ofp_parser class is predefined in the Ryu codebase and provides functions that parse individual protocol layers and represent data in a key value pair format. The first data which is easily deciphered is the in port and then Ethernet protocol details are extracted. The in port will be used later in the algorithm. The Ethertype property in the Ethernet protocol helps determine payload, if the packet is of ICMP, IP, etc type and once it’s determined only ARP and IP packets are processed further and the rest are dropped. Thus, the Ryu application would return the packet to the switch to with drop action. If the packet is an ARP packet than the ARP handler function is called and packet is passed to it.

The ARP handler function will parse the layer 2 and layer 3 packages present in the packet as it’s an ARP protocol. The source mac address, source IP address, Ethernet type, ingress port/

in port will be used to create the ARP reply or response message according to the algorithm used.

If the packet is of type IP than its further stripped to see if it contains a UDP or TCP package. Depending on the package a function call will be made to the corresponding package handler function. For UDP package a UDP request handler is called and the packet is passed as a parameter to it but if it’s a TCP package than its passed to a TCP request handler.

The TCP handler function will receive the packet and look at the layer 2 header format. The destination and source IP addresses are retrieved from this section and TCP header is than parsed to get the TCP source port and destination port. Similarly, in the UDP handler function first the IP address is read than the UDP package is parsed to get the UDP source and destination ports.

ofp_parser.OFPMatch class is used to create the match field in the flow entry. It consumes the in port and source and destination IP and MAC addresses. To create the action field ofp_parser.OFPActionSetField is used. It defines what change has to be made to the packet header for example editing the destination IP or MAC. The ofp_parser.OFPFlowMod function inserts the flow entry in the flow entry in the flow table present in the connected switch.

The function that returns the selected server address is different for each load balancing algorithm and might use extra space to store the past and current statistics of a server.

Random Load Balancing

One of the simplest algorithms and still very effective is the random load balancing. The switch is connected to the controller over a secured connection using SSL and uses port 6633 on the controller to exchange OpenFlow packets.

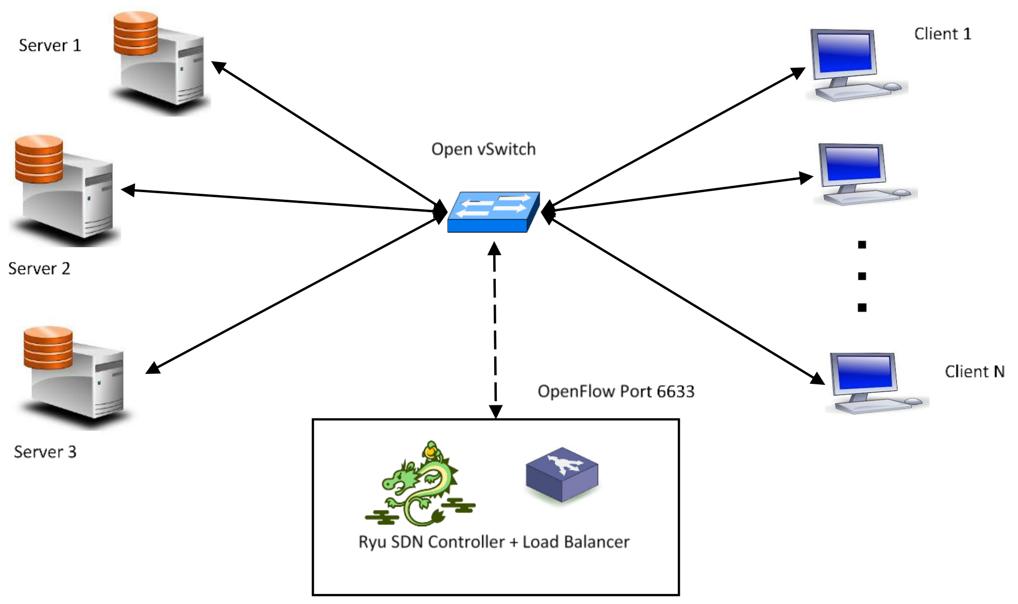
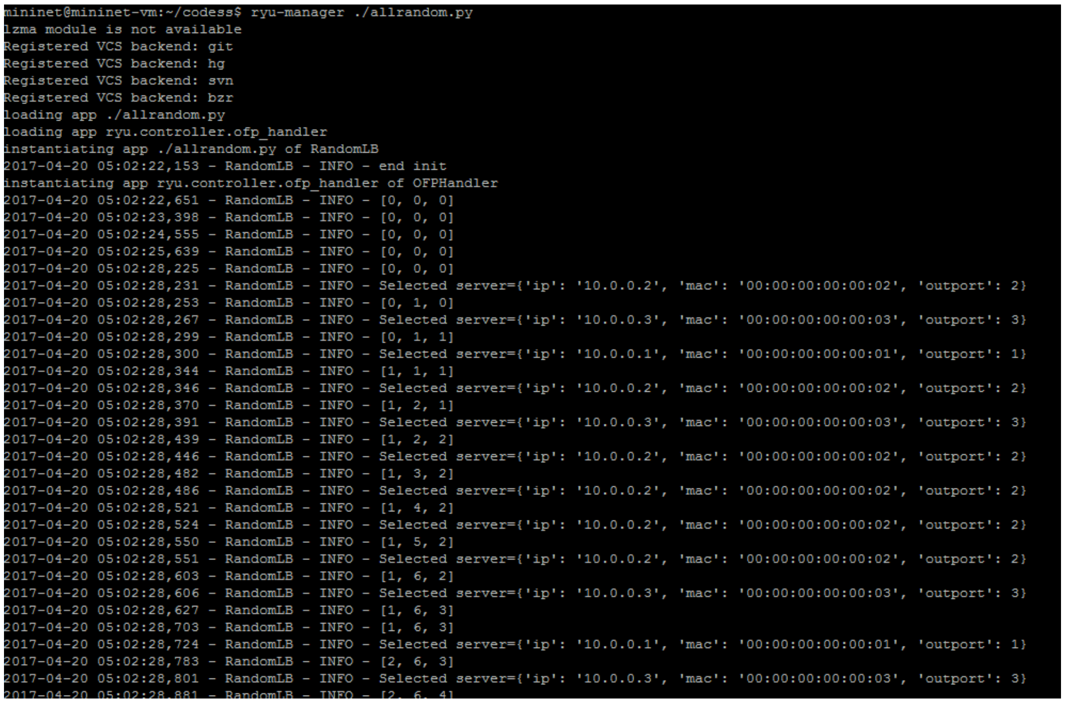


Figure 4-1 Architecture for Random Load Balancing

It can be seen in the logs generated during the running of the algorithm that first the ryu manager gets initialized and it loads the load balancing application. Once the packets start flowing into the controller via the switch the algorithm starts assigning the servers IP and Ethernet address to each connection. It uses a random function which returns a value that is in the range of the number of servers online. The number generated corresponds to the index value of the server. The selected servers IP and MAC address are used to generate a flow for handling future requests from that client. The algorithm doesn’t consider any metric and or has any past knowledge of the server’s present.



```
mininet@mininet-vm:~/codess$ ryu-manager ./allrandom.py
lzma module is not available
Registered VCS backend: git
Registered VCS backend: hg
Registered VCS backend: svn
Registered VCS backend: bzr
loading app ./allrandom.py
loading app ryu.controller.ofp_handler
instantiating app ./allrandom.py of RandomLB
2017-04-20 05:02:22,153 - RandomLB - INFO - end init
instantiating app ryu.controller.ofp_handler of OFPHandler
2017-04-20 05:02:22,651 - RandomLB - INFO - [0, 0, 0]
2017-04-20 05:02:23,398 - RandomLB - INFO - [0, 0, 0]
2017-04-20 05:02:24,555 - RandomLB - INFO - [0, 0, 0]
2017-04-20 05:02:25,639 - RandomLB - INFO - [0, 0, 0]
2017-04-20 05:02:28,225 - RandomLB - INFO - [0, 0, 0]
2017-04-20 05:02:28,231 - RandomLB - INFO - Selected server={'ip': '10.0.0.2', 'mac': '00:00:00:00:00:02', 'outport': 2}
2017-04-20 05:02:28,253 - RandomLB - INFO - [0, 1, 0]
2017-04-20 05:02:28,267 - RandomLB - INFO - Selected server={'ip': '10.0.0.3', 'mac': '00:00:00:00:00:03', 'outport': 3}
2017-04-20 05:02:28,299 - RandomLB - INFO - [0, 1, 1]
2017-04-20 05:02:28,300 - RandomLB - INFO - Selected server={'ip': '10.0.0.1', 'mac': '00:00:00:00:00:01', 'outport': 1}
2017-04-20 05:02:28,344 - RandomLB - INFO - [1, 1, 1]
2017-04-20 05:02:28,346 - RandomLB - INFO - Selected server={'ip': '10.0.0.2', 'mac': '00:00:00:00:00:02', 'outport': 2}
2017-04-20 05:02:28,370 - RandomLB - INFO - [1, 2, 1]
2017-04-20 05:02:28,391 - RandomLB - INFO - Selected server={'ip': '10.0.0.3', 'mac': '00:00:00:00:00:03', 'outport': 3}
2017-04-20 05:02:28,439 - RandomLB - INFO - [1, 2, 2]
2017-04-20 05:02:28,446 - RandomLB - INFO - Selected server={'ip': '10.0.0.2', 'mac': '00:00:00:00:00:02', 'outport': 2}
2017-04-20 05:02:28,482 - RandomLB - INFO - [1, 3, 2]
2017-04-20 05:02:28,486 - RandomLB - INFO - Selected server={'ip': '10.0.0.2', 'mac': '00:00:00:00:00:02', 'outport': 2}
2017-04-20 05:02:28,521 - RandomLB - INFO - [1, 4, 2]
2017-04-20 05:02:28,524 - RandomLB - INFO - Selected server={'ip': '10.0.0.2', 'mac': '00:00:00:00:00:02', 'outport': 2}
2017-04-20 05:02:28,550 - RandomLB - INFO - [1, 5, 2]
2017-04-20 05:02:28,551 - RandomLB - INFO - Selected server={'ip': '10.0.0.2', 'mac': '00:00:00:00:00:02', 'outport': 2}
2017-04-20 05:02:28,603 - RandomLB - INFO - [1, 6, 2]
2017-04-20 05:02:28,606 - RandomLB - INFO - Selected server={'ip': '10.0.0.3', 'mac': '00:00:00:00:00:03', 'outport': 3}
2017-04-20 05:02:28,627 - RandomLB - INFO - [1, 6, 3]
2017-04-20 05:02:28,703 - RandomLB - INFO - [1, 6, 3]
2017-04-20 05:02:28,724 - RandomLB - INFO - Selected server={'ip': '10.0.0.1', 'mac': '00:00:00:00:00:01', 'outport': 1}
2017-04-20 05:02:28,783 - RandomLB - INFO - [2, 6, 3]
2017-04-20 05:02:28,801 - RandomLB - INFO - Selected server={'ip': '10.0.0.3', 'mac': '00:00:00:00:00:03', 'outport': 3}
2017-04-20 05:02:28,881 - RandomLB - INFO - [2, 6, 4]
```

Figure 3-5 Log for Random Load Balancing

Round Robin Load Balancing

Another popular algorithm that is used in a lot of different scenarios is round robin. The clients on one side of the switch will connect to the load balancer's global IP. The algorithm uses a counter to decide the server. The counter is incremented each time a new connection is made and reset once the count increases to more than the total number of servers present in the pool. Thus, the counter acts as the index of the next server. The algorithm is fast and simple with space complexity of 1.

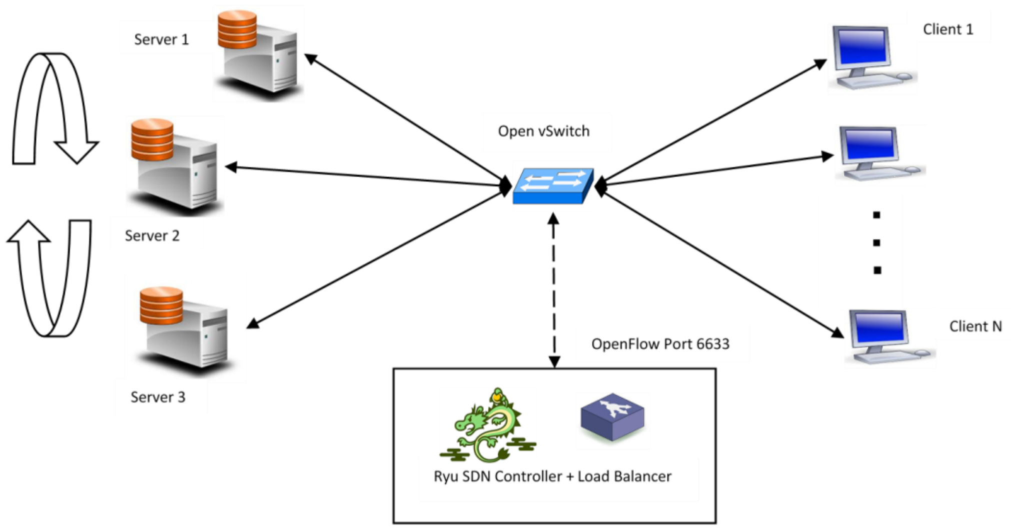
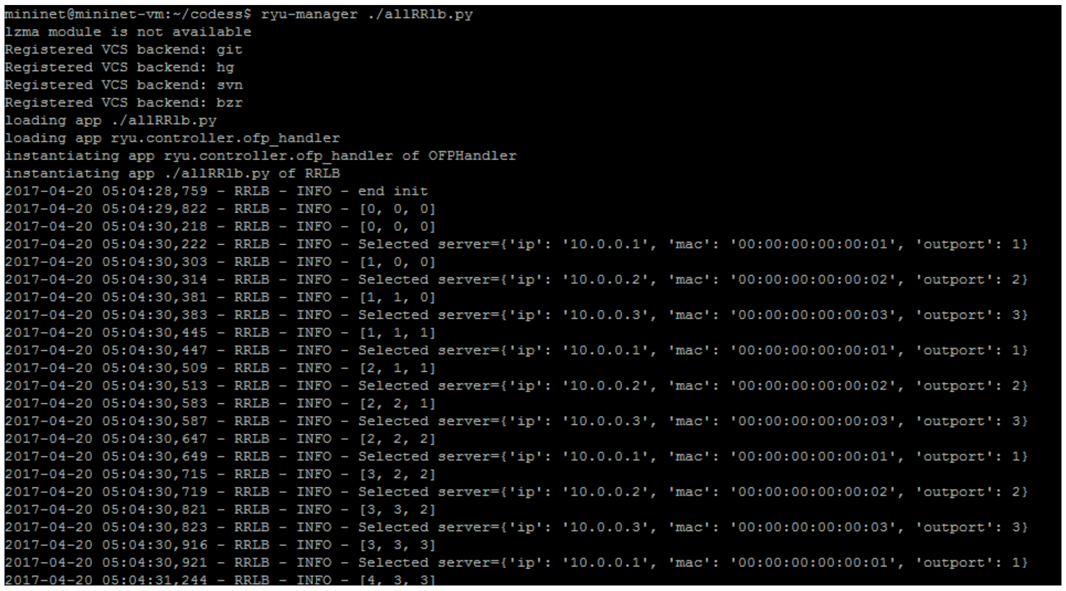


Figure 3-6 Architecture for Round Robin Load Balancing

The round robin algorithm is clearly visible in the logs as each new request is served by the next server in the order of their index. The IP, MAC and output port of the selected server is returned by the selection algorithm.



```
mininet@mininet-vm:~/codess$ ryu-manager ./allRRlb.py
lzma module is not available
Registered VCS backend: git
Registered VCS backend: hg
Registered VCS backend: svn
Registered VCS backend: bzr
loading app ./allRRlb.py
loading app ryu.controller.ofp_handler
instantiating app ryu.controller.ofp_handler of OFPHandler
instantiating app ./allRRlb.py of RRLB
2017-04-20 05:04:28,759 - RRLB - INFO - end init
2017-04-20 05:04:29,822 - RRLB - INFO - [0, 0, 0]
2017-04-20 05:04:30,218 - RRLB - INFO - [0, 0, 0]
2017-04-20 05:04:30,222 - RRLB - INFO - Selected server={'ip': '10.0.0.1', 'mac': '00:00:00:00:00:01', 'outport': 1}
2017-04-20 05:04:30,303 - RRLB - INFO - [1, 0, 0]
2017-04-20 05:04:30,314 - RRLB - INFO - Selected server={'ip': '10.0.0.2', 'mac': '00:00:00:00:00:02', 'outport': 2}
2017-04-20 05:04:30,381 - RRLB - INFO - [1, 1, 0]
2017-04-20 05:04:30,383 - RRLB - INFO - Selected server={'ip': '10.0.0.3', 'mac': '00:00:00:00:00:03', 'outport': 3}
2017-04-20 05:04:30,445 - RRLB - INFO - [1, 1, 1]
2017-04-20 05:04:30,447 - RRLB - INFO - Selected server={'ip': '10.0.0.1', 'mac': '00:00:00:00:00:01', 'outport': 1}
2017-04-20 05:04:30,509 - RRLB - INFO - [2, 1, 1]
2017-04-20 05:04:30,513 - RRLB - INFO - Selected server={'ip': '10.0.0.2', 'mac': '00:00:00:00:00:02', 'outport': 2}
2017-04-20 05:04:30,583 - RRLB - INFO - [2, 2, 1]
2017-04-20 05:04:30,587 - RRLB - INFO - Selected server={'ip': '10.0.0.3', 'mac': '00:00:00:00:00:03', 'outport': 3}
2017-04-20 05:04:30,647 - RRLB - INFO - [2, 2, 2]
2017-04-20 05:04:30,649 - RRLB - INFO - Selected server={'ip': '10.0.0.1', 'mac': '00:00:00:00:00:01', 'outport': 1}
2017-04-20 05:04:30,715 - RRLB - INFO - [3, 2, 2]
2017-04-20 05:04:30,719 - RRLB - INFO - Selected server={'ip': '10.0.0.2', 'mac': '00:00:00:00:00:02', 'outport': 2}
2017-04-20 05:04:30,821 - RRLB - INFO - [3, 3, 2]
2017-04-20 05:04:30,823 - RRLB - INFO - Selected server={'ip': '10.0.0.3', 'mac': '00:00:00:00:00:03', 'outport': 3}
2017-04-20 05:04:30,916 - RRLB - INFO - [3, 3, 3]
2017-04-20 05:04:30,921 - RRLB - INFO - Selected server={'ip': '10.0.0.1', 'mac': '00:00:00:00:00:01', 'outport': 1}
2017-04-20 05:04:31,244 - RRLB - INFO - [4, 3, 3]
```

Figure 3-7 Log for Round Robin Load Balancing

Dynamic Load Balancing

The dynamic algorithm is different from the above implemented static algorithms as it considers the state of the server at current time. There are two ways of getting the server statistics either internally from the server itself or by using an external software to measure and collect the stats. To get data from the server itself, we can install an agent in it which sends periodic updates to the load balancer or a third-party server that than sends it to the load balancer. For simulation purpose, we have implemented the statistics collection in the HTTP webserver code itself. Therefore, once the Python HTTP server is started on the emulated server, the application will periodically, every 5 seconds, send heartbeats to the load balancer. The data sent by the server to the load balancer should not be seen by the client or outside network and should not face time delay. Due to these conditions, UDP protocol was decided as the carrier for time sensitive information.

Multiple parameters can be used in load balancing of a website. For the scope of this thesis the following parameters were considered:

Rx= received bytes, Tx= transmitted bytes

CPU utilization

The server will utilize its RAM, HDD, CPU to handle the incoming packets and process them. This parameter takes in to consideration the CPU utilization at a given moment and which if lower shows that the server can handle ore connections. A high-performance system might show lesser CPU utilization even though it might be handling high number of requests.

Average load

The average number of packets that a system received between two requests. It can found out using the switch flow statistics. The number of packets received between the current request and the previous request divide by the time space between the two request will give us the average load that the port was handling.

Average load = (current Rx - previous Rx) / delta time

Port speed

The maximum amount of data that can pass through the port is called port speed.

Port speed = (current Rx + current Tx) - (previous Rx + previous Tx) / delta time

Packet loss rate

Due to heavier loads on a port and multiple connections passing many a times packets would be lost due to congestion and buffer overflow at the entry. Thus, the transmitted bytes would be less than the received bytes at the same port.

Packet loss rate = |current Tx – current Rx| / current Tx

Predefined weights

To emulate the difference between the performance of the servers and have a fall back to a weighted round robin algorithm incase the server stops sending heartbeats, predefined weights are used.

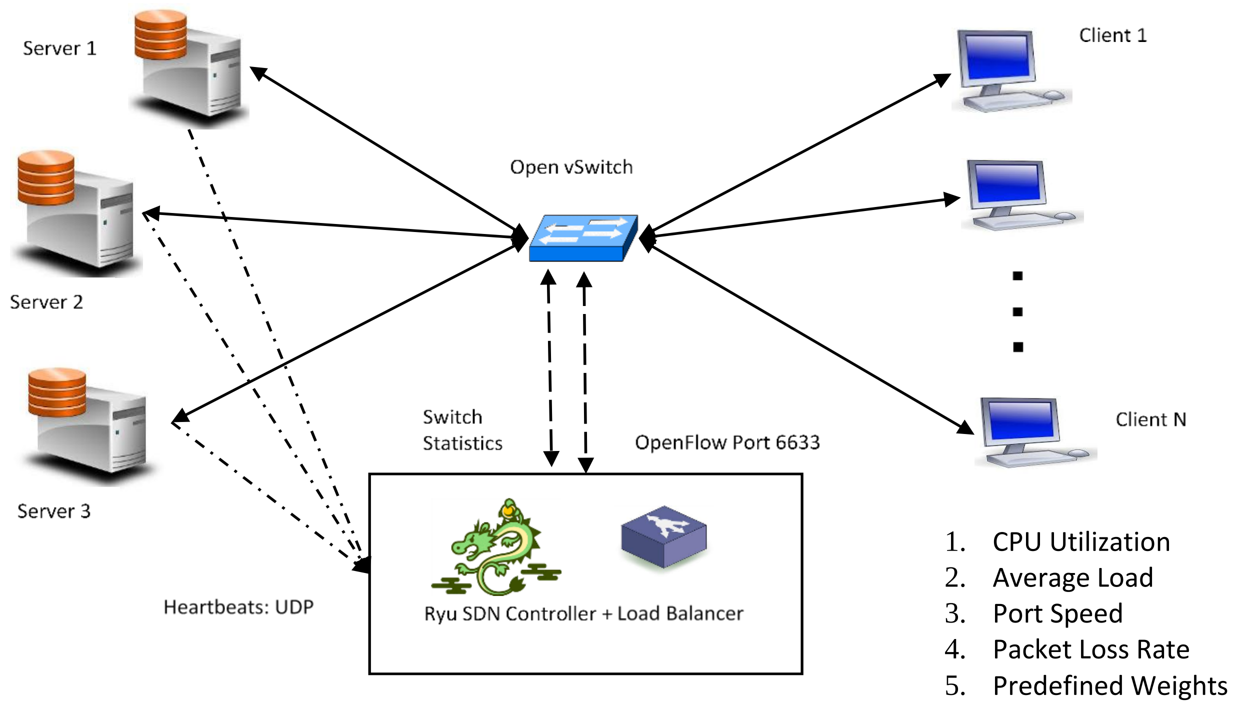


Figure 3-8 Architecture for Dynamic Load Balancing

The architecture has the same topology as the one used in the previous experiments but has two 2 additional connections in it. The servers send heartbeats in the form of UDP packets to the load balancer at a user defined port 8188. The load balancer needs to differentiate between the client UDP traffic generated and a server's heartbeat.

To gather port statistics the load balancer has to periodically ping the switch with a OFPFlowStatsRequest and wait for the reply. A separate thread is started by the dynamic load balancing application that takes care of sending and receiving request messages. This exchange is done in the secure channel between the controller and the switch utilizing the OpenFlow protocol. These messages are not of TCP/UDP types but are defined under OpenFlow specification. The main thread in the application waits for incoming packets from the switch and starts a parallel thread to maintain a parallel connection with the connected switch. Once a packet arrives it is checked for a TCP Ethertype parameter. If packet contains a TCP payload than the TCP event handler function handles it. If the packet contains a UDP payload than its segregated based on certain conditions.

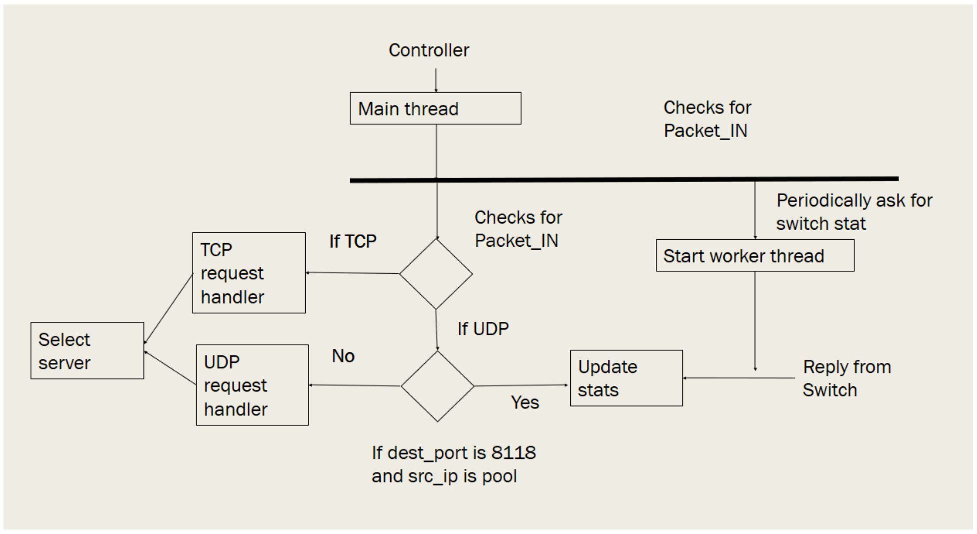
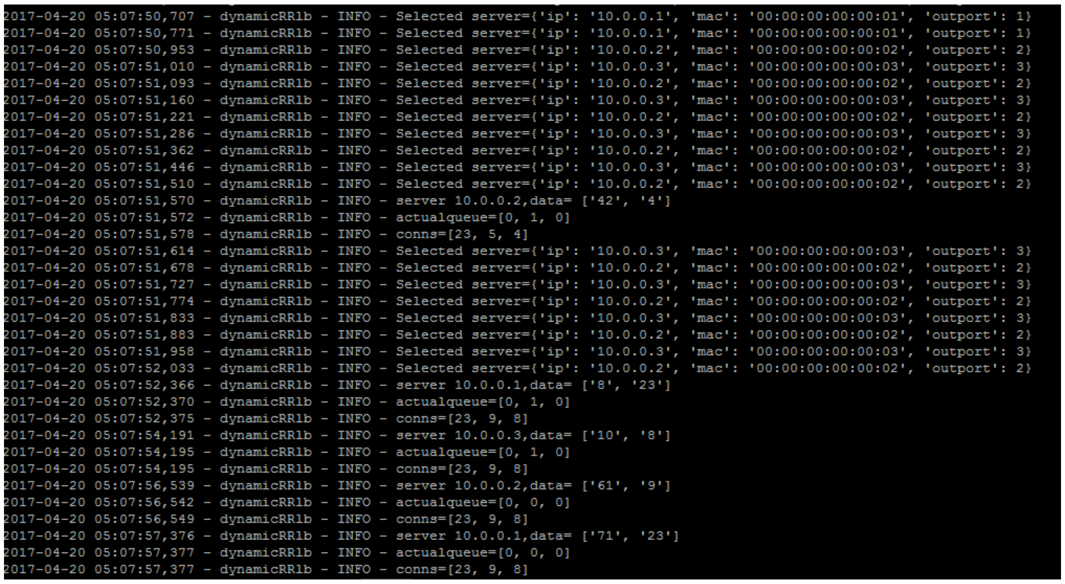


Figure 3-9 Process Flow for Dynamic Load Balancing

If the UDP packet has the destination address of the load balancer with the destination port as 8188 than its meant for the load balancer as a heartbeat and so the packet would be passed to the update stat function. In this function, different parameter like port speed are calculated and the global data structure is updated with the current values.

If the packet fails, the condition than its meant for a client machine and it would be handled by the UDP handler function.



```
2017-04-20 05:07:50,707 - dynamicRRlb - INFO - Selected server={'ip': '10.0.0.1', 'mac': '00:00:00:00:00:01', 'outport': 1}
2017-04-20 05:07:50,771 - dynamicRRlb - INFO - Selected server={'ip': '10.0.0.1', 'mac': '00:00:00:00:00:01', 'outport': 1}
2017-04-20 05:07:50,953 - dynamicRRlb - INFO - Selected server={'ip': '10.0.0.2', 'mac': '00:00:00:00:00:02', 'outport': 2}
2017-04-20 05:07:51,010 - dynamicRRlb - INFO - Selected server={'ip': '10.0.0.3', 'mac': '00:00:00:00:00:03', 'outport': 3}
2017-04-20 05:07:51,093 - dynamicRRlb - INFO - Selected server={'ip': '10.0.0.2', 'mac': '00:00:00:00:00:02', 'outport': 2}
2017-04-20 05:07:51,160 - dynamicRRlb - INFO - Selected server={'ip': '10.0.0.3', 'mac': '00:00:00:00:00:03', 'outport': 3}
2017-04-20 05:07:51,221 - dynamicRRlb - INFO - Selected server={'ip': '10.0.0.2', 'mac': '00:00:00:00:00:02', 'outport': 2}
2017-04-20 05:07:51,286 - dynamicRRlb - INFO - Selected server={'ip': '10.0.0.3', 'mac': '00:00:00:00:00:03', 'outport': 3}
2017-04-20 05:07:51,362 - dynamicRRlb - INFO - Selected server={'ip': '10.0.0.2', 'mac': '00:00:00:00:00:02', 'outport': 2}
2017-04-20 05:07:51,446 - dynamicRRlb - INFO - Selected server={'ip': '10.0.0.3', 'mac': '00:00:00:00:00:03', 'outport': 3}
2017-04-20 05:07:51,510 - dynamicRRlb - INFO - Selected server={'ip': '10.0.0.2', 'mac': '00:00:00:00:00:02', 'outport': 2}
2017-04-20 05:07:51,570 - dynamicRRlb - INFO - server 10.0.0.2,data= ['42', '4']
2017-04-20 05:07:51,572 - dynamicRRlb - INFO - actualqueue=[0, 1, 0]
2017-04-20 05:07:51,578 - dynamicRRlb - INFO - conns=[23, 5, 4]
2017-04-20 05:07:51,614 - dynamicRRlb - INFO - Selected server={'ip': '10.0.0.3', 'mac': '00:00:00:00:00:03', 'outport': 3}
2017-04-20 05:07:51,678 - dynamicRRlb - INFO - Selected server={'ip': '10.0.0.2', 'mac': '00:00:00:00:00:02', 'outport': 2}
2017-04-20 05:07:51,727 - dynamicRRlb - INFO - Selected server={'ip': '10.0.0.3', 'mac': '00:00:00:00:00:03', 'outport': 3}
2017-04-20 05:07:51,774 - dynamicRRlb - INFO - Selected server={'ip': '10.0.0.2', 'mac': '00:00:00:00:00:02', 'outport': 2}
2017-04-20 05:07:51,833 - dynamicRRlb - INFO - Selected server={'ip': '10.0.0.3', 'mac': '00:00:00:00:00:03', 'outport': 3}
2017-04-20 05:07:51,883 - dynamicRRlb - INFO - Selected server={'ip': '10.0.0.2', 'mac': '00:00:00:00:00:02', 'outport': 2}
2017-04-20 05:07:51,958 - dynamicRRlb - INFO - Selected server={'ip': '10.0.0.3', 'mac': '00:00:00:00:00:03', 'outport': 3}
2017-04-20 05:07:52,033 - dynamicRRlb - INFO - Selected server={'ip': '10.0.0.2', 'mac': '00:00:00:00:00:02', 'outport': 2}
2017-04-20 05:07:52,366 - dynamicRRlb - INFO - server 10.0.0.1,data= ['8', '23']
2017-04-20 05:07:52,370 - dynamicRRlb - INFO - actualqueue=[0, 1, 0]
2017-04-20 05:07:52,375 - dynamicRRlb - INFO - conns=[23, 9, 8]
2017-04-20 05:07:54,191 - dynamicRRlb - INFO - server 10.0.0.3,data= ['10', '8']
2017-04-20 05:07:54,195 - dynamicRRlb - INFO - actualqueue=[0, 1, 0]
2017-04-20 05:07:54,195 - dynamicRRlb - INFO - conns=[23, 9, 8]
2017-04-20 05:07:56,539 - dynamicRRlb - INFO - server 10.0.0.2,data= ['61', '9']
2017-04-20 05:07:56,542 - dynamicRRlb - INFO - actualqueue=[0, 0, 0]
2017-04-20 05:07:56,549 - dynamicRRlb - INFO - conns=[23, 9, 8]
2017-04-20 05:07:57,376 - dynamicRRlb - INFO - server 10.0.0.1,data= ['71', '23']
2017-04-20 05:07:57,377 - dynamicRRlb - INFO - actualqueue=[0, 0, 0]
2017-04-20 05:07:57,377 - dynamicRRlb - INFO - conns=[23, 9, 8]
```

Figure 3-10 Log for Dynamic Load Balancing

During the running of this algorithm the values received in the heartbeat from the server is logged and visible. The CPU utilization and a random number is displayed. The output of the algorithm is random looking as the selection varies from time to time due to real time calculations. If the request arrives before there is an update in the server statistics than historical values are used to predict the next server.

Chapter 4

Evaluation

Evaluation of the performance and efficiency of the dynamic load balancer is the main objective in this chapter. It uses multiple attributes to decide which server should next service an incoming request. It's compared against random and round robin load balancer. HTTP and UDP connections are used to test different properties of the software appliance.

Topology and Testbed Setup

A standard topology is created using Mininet consisting of servers and host on the opposite sides of a switch.

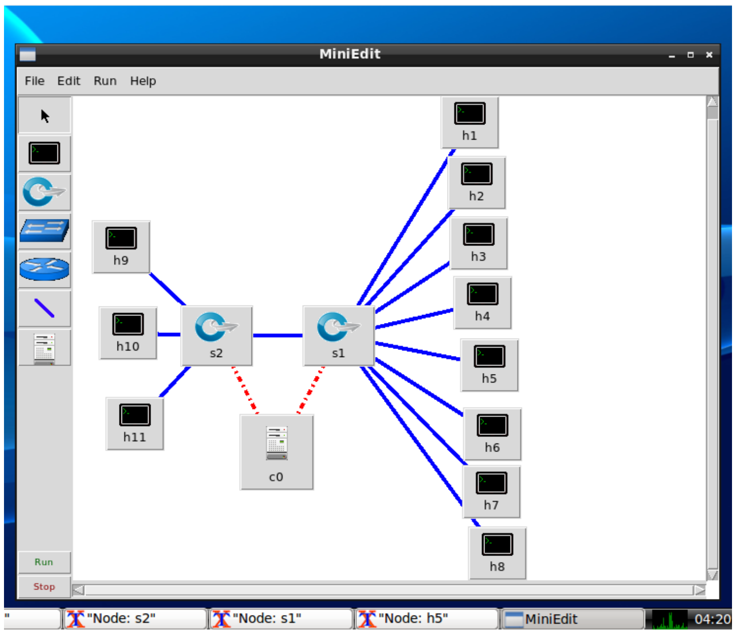


Figure 4-1 Example topology in Mininet

All the components in this setup are virtualized including the links, paths, switches, clients and servers. The network conditions can be simulated across routes including latency, CPU utilization and jitters. Firstly, HTTP requests are used as these connections are faced by load balancers the most. Secondly, we used UDP packets to check the jitter, rate of packet loss and throughput values for data transmission. Apache Benchmark tool tests HTTP connections. There are lot of different scenarios that can be tested using AB tool. We can specify if the connection needs to be serial or parallel and with variable or constant load. A second tool called Iperf is used to test the UDP bandwidth. A python script is used to simulate an HTTP server to which the AB tool gets data from. The HTTP server also uses a server weight with some random value to emulate the current CPU utilization which is sent as a heartbeat to the load balancer. The servers are assigned weights in decreasing order and its assumed that the higher weight the more connections it can handle than others. Server 3 is given the most weight and then server 2 and the least powerful is assumed to be server 1.

Evaluation Results

HTTP Serial Requests

In this experiment, each type of load balancer is subject to HTTP request ranging from 10 to 100k connections. All the requests were sent from the clients to the load balancer IP for servicing. The load balancer application handles the packet in the controller and then passed to the switch to be forwarded to any one IP from the server pool. Each server upon receiving the request would return an HTML of defined number of bytes. The Apache Bench tool is used to produce the client requests. The data generated by the tool is than analyzed to produce the following graphs.

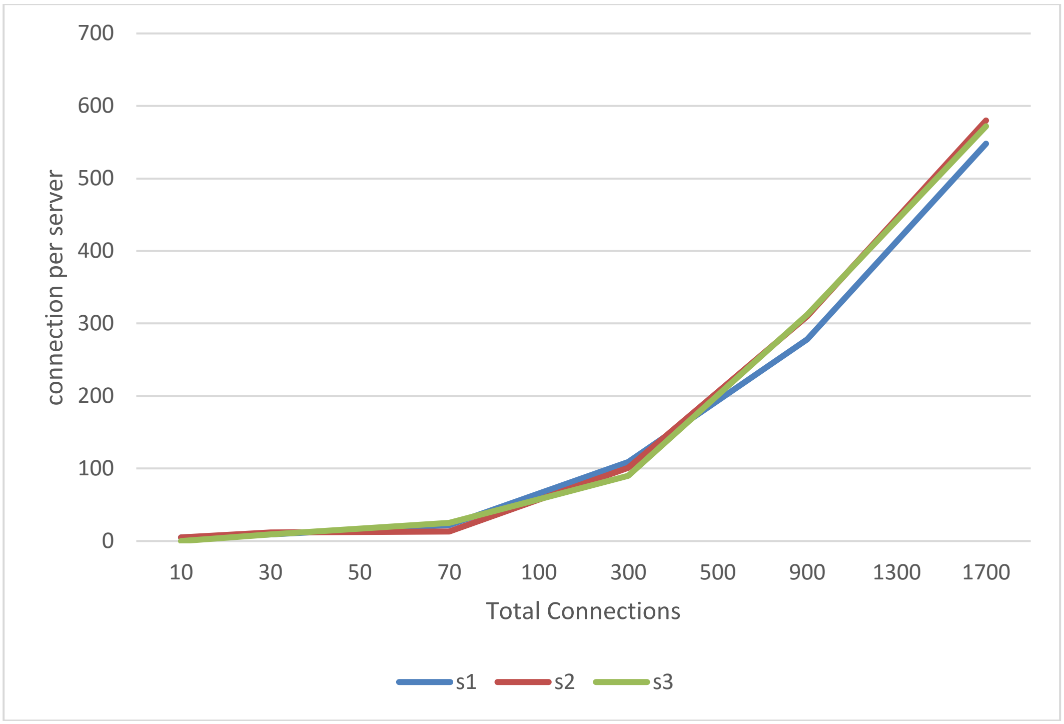


Figure 4-2 Load Distribution for Random Load Balancing

The first algorithm to be benchmarked is the Random Load balancing algorithm as this forms a minimum foundation for comparison as it is the most basic type of load balancing. The balancer clearly doesn’t take into consideration the weights assigned to each server. Neither the real capacity and processing power of each machine is not considered nor the least connected or loaded. The graphs for each server randomly cross path and server 1 tends to receive the least number of connections though it was assigned the biggest weight.

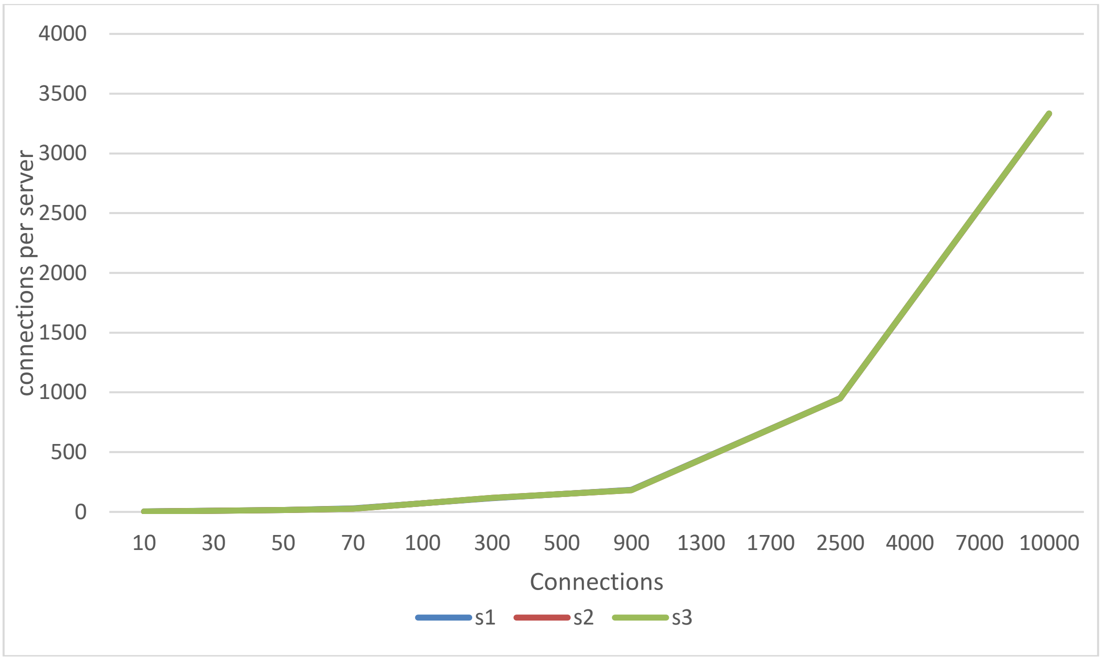


Figure 4-3 Load Distribution for Round Robin Load Balancing

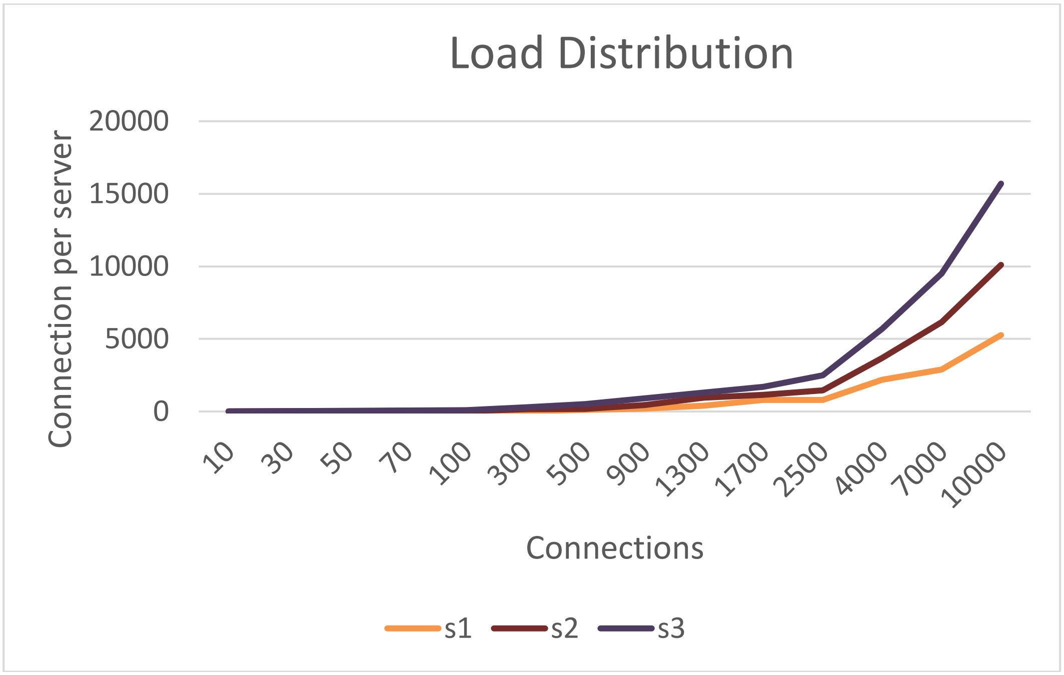


Figure 4-4 Load Distribution for Dynamic Load Balancing

Round robin load balancer is next to be experimented with and the graphs clearly show that each server received equal number of connections as the total connections were equally divided among the servers in the pool. Round robin algorithm doesn’t feature in the weights and least connections that any serer would be handling at a point of time. It can lead to increased utilization of one machine and thus loss of bandwidth for future connections.

The number of connections distributed by the dynamic load balancer considers the current CPU utilization and server load capability. Values are received from the switch and port speed, etc. parameters are also used in selected the next server. Thus server 1 is least loaded in totality while server 3 is heavily loaded with more connections.

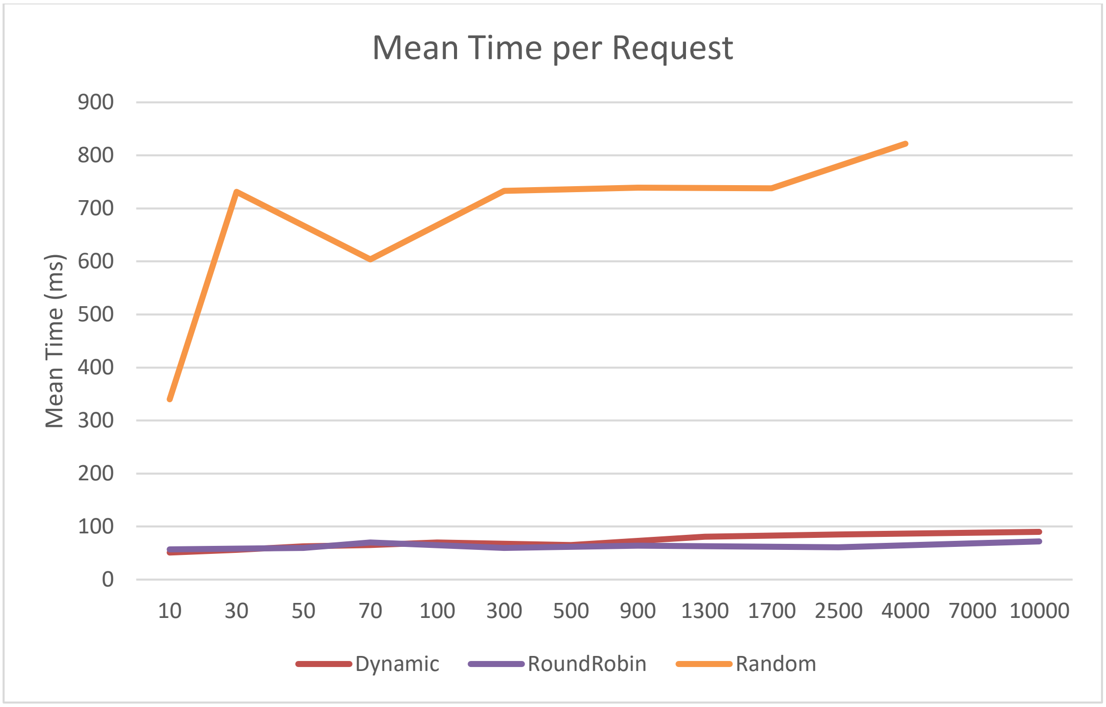


Figure 4-5 Comparing the average Request completion time

If the same server is handling all the incoming requests than it can start slowing and might take a lot of time in servicing each request or simply reject request next in the queue. Random load balancer doesn’t take in to count the number of connection being served by a server at any given point time and can keep giving the same server many requests too service. Thus, by not distributing the requests it might overload one server which will reduce the average time to complete a request. Round robin algorithm will equally distribute the requests in a circular way and thus will not suffer this problem with the same intensity. Though it might happen that one less powerful server can start getting loaded with multiple requests even though it has finished completing the previous request. The curve for round robin and dynamic algorithm are overlapping till 1200 connections after which round robin algorithm maintains its slope while dynamic algorithm shows an increase in the mean time to complete. This behavior might be due to the extra processing time required by the dynamic algorithm to select the next server. This calculation takes over the request processing time leading to a slight increase in average time to finish processing a request.

HTTP Concurrent Requests

A big performance test for any HTTP server is to handle multiple concurrent requests. To handle multiple requests the server has to be able to create threads to receive and server a request.

Table 4-1 HTTP Concurrent Requests

Concurrency	Connections	Random	Round Robin	Dynamic
1	100	Y	Y	Y
5	50	Y	Y	Y
10	100	Y	Y	Y
30	600	Y	Y	N(244)
90	2700	N(1138)	N(173)	N

The server here has been kept constant and it’s only the load balancing logic which changes. random load balancer is the fastest in assigning the requests and thus can cater to huge amounts of concurrency. Round robin load balancer comes second to random load balancer in serving multiple concurrent requests, completing at the most 173 connections when 90 connections parallel HTTP streams were connected to it by ApacheBench. Dynamic load balancer starts giving connection reset errors early on as the algorithm doesn’t react fast enough to so many parallel requests. It nearly simulated a denial of service attack as the server couldn’t handle all incoming connections and so were being dropped.

UDP Throughput

Iperf tool is used to benchmark the UDP bandwidth and throughput. The tool can be used to start a UDP server on the server machines and multiple Iperf clients can be started on the client machines. This tool is used to create multiple UDP connections to the load balancer IP which are than assigned to each available server IP where the corresponding Iperf server handles the incoming UDP packet.

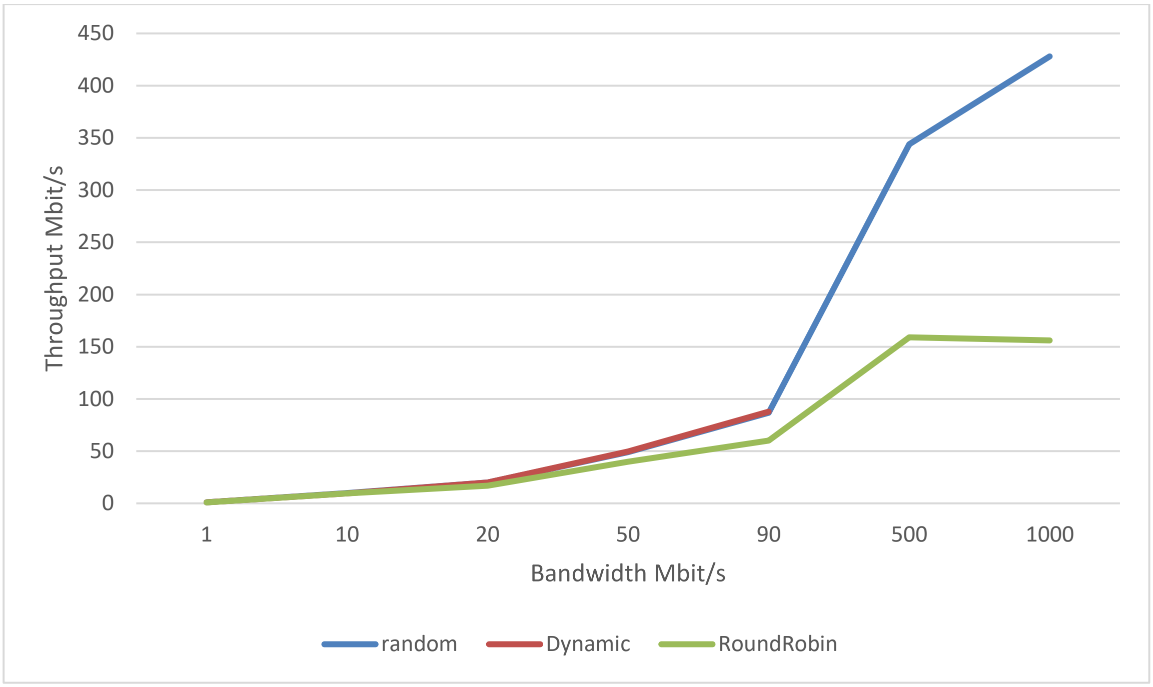


Figure 4-6 Comparing the UDP throughput

All three load balancing techniques show an increase in bandwidth and throughput to a certain level and then stagnates due to network link and higher packet loss. Initially dynamic load balancer is performing better than Round Robin technique but as the bandwidth of the link rises, the Iperf client starts giving connection errors.

High amount of jitter and increased fragmentation in packets limits the throughput of dynamic load balancer. Random load balancer though performs exceptionally well in this scenario where UDP floods the network with datagrams. The run time of the random load balancing algorithm being small doesn’t add much delay for processing of each new fragment.

Chapter 5

Summary

Conclusion

Traditional networking has relied upon distributed control logic which limits its agility. The routers and switches need to keep itself updated by periodically refreshing its flow table and network map by communicating with surrounding devices. This decentralized control doesn’t allow rapid changes to the network. Experiments cannot be easily implemented in such networks and scalability takes a hit as manual intervention is required a lot. The complexity and cost of the networks increase due to components being vendor specific and hardware based. Software defined networking allows the use of programming language to come up with network devices and topology. The building blocks of a network router and switch can now be virtualized and used as a software based component. This reduces vendor lock in and makes network admins job of maintenance easier.

There are different ways to load balance an application or a website. We have dealt with one such type and have successfully implemented a dynamic load balancer. It uses a variety of characteristics obtained from different sources to determine the next server.

The load balancers were tested in several scenarios from continuous data stream such as HTTP to more connection less protocols such as UDP. A basic topology is developed in Mininet and a couple of benchmarking tools were used to create connections to the virtualized servers. From the results, we can conclude that dynamic load balancer distributes the load better than its competitors. It does give a poor performance in certain other tests due to a small overhead in simultaneously coming up with the next server as well as gathering the updates from different parts of the network. The role of SDN is emphasized here as it allows us to experiment with network and virtual functions easily without harming current networks.

Future Work

A lot of techniques which weren’t applicable before are now being made possible due to Software Defined networks. Having a centralized view of the whole network allows the admin to easily come up with updated and faster routes with lesser resources. Many domains are still left untouched in this thesis.

Many other protocols can be allowed to pass through the switch and thus the logic of the load balancer would change.

Currently MAC values were already loaded from the beginning in each machine to reduce the number of the ARP requests flooding the network. Though this is feature is used in data centers currently but it can be looked deeper into.

A more rigorous performance testing of the HTTP and UDP can be performed. There are open source tools other than ApacheBench and Iperf which can be utilized for testing the network parameters such as bandwidth, throughput, packet loss.

Multiple topologies can be used to test the load balancer appliance with multiple switches being placed in between the servers and clients. The fat tree topology isn’t utilized in the experiments, which is another important type of topology widely used in data centers. Path management is another vertical that can be applied as a next feature to the core logic. Finding the fastest route between the server and hosts in a data center is a real-world problem. Dijkstra algorithm can be modified to find the best path in the network using features extracted from the OpenFlow protocol and third party network monitoring applications.

The number of parameters that were currently used in the algorithm to decide the next server can be varied to gives us more control of the flow and decision speed.

Instead of an emulator we can experiment with real computers with one machine acting as the virtual switch and having the controller in it. This could give us more accurate measurements with actual network latencies and jitters present in the network. A machine learning module can be attached to the load balancer to extend the prediction capability of the algorithm resulting in better resource utilization.

This thesis proves that dynamic load balancing takes multiple factors into consideration which are not used in static techniques.

Reference

[1] "Software-Defined Networking (SDN) Definition," [Online]. Available: https://www.opennetworking.org/sdn-resources/sdn-definition. [Accessed 3 September 2016].

[2] "Load balancing (computing)," [Online]. Available: https://en.wikipedia.org/wiki/Load_balancing_(computing). [Accessed August 2016].

[3] "NetFPGA: Programmable Networking Hardware.," [Online]. Available: http://netfpga.org.. [Accessed December 2016].

[4] "What are SDN Controllers (or SDN Controllers Platforms)?," sdxcentral, [Online]. Available: https://www.sdxcentral.com/sdn/definitions/sdn-controllers/. [Accessed October 2016].

[5] "Network operating system," wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Network_operating_system. [Accessed October 2016].

[6] "Round Robin Load Balancing," kemptechnologies, [Online]. Available: https://kemptechnologies.com/load-balancing/round-robin-load-balancing/. [Accessed October 2016].

[7] "Load Balancing Algorithms," kemptechnologies, [Online]. Available: https://kemptechnologies.com/load-balancer/load-balancing-algorithms-techniques/. [Accessed October 2016].

[8] "Ryu Controller," NTT, [Online]. Available: https://osrg.github.io/ryu/. [Accessed October 2016].

[9] "Mininet," [Online]. Available: http://mininet.org/. [Accessed October 2016].

[10] M. J. F. J. P. J. L. N. M. S. S. Martin Casado, "Ethane: taking control of the enterprise," in *ACM SIGCOMM '07*, Kyoto, Japan, 2007.

[11] N. F. A. G. Martin Casado, "Abstractions for Software-Defined Networks," *Communications of the ACM,* pp. 86-95, September 2014.

[12] T. A. H. B. G. P. L. P. N. McKeown, "Openflow: Enabling innovation in campus networks," in *ACM SIGCOMM Computer Communications Review*, New York, 2008.

[13] P. Simoneau, "How do Switches Work?," globalknowledge, [Online]. Available: https://www.globalknowledge.com.eg/about-us/Knowledge-Center/Article/How-do-Switches-Work/. [Accessed October 2016].

[14] M. J. F. J. P. J. L. N. G. N. M. S. S. Martin Casado, "Rethinking Enterprise Network Control," *IEEE/ACM Transactions on Networking,* vol. 17, no. 4, 2009.

[15] "OpenFlow Switch Specification," [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.1.pdf.

[16] "FloodLight," [Online]. Available: http://www.projectfloodlight.org/floodlight/.

[17] J. P. T. K. E. J. J. A. Z. J. R. J. G. A. W. J. S. P. S. K. A. M. C. B. Pfaff, "The Design and Implementation of Open vSwitch," in *USENIX/ACM Symposium on Networked Systems Design and Implementation*, 2015.

[18] "The POX Controller," [Online]. Available: https://github.com/noxrepo/pox. [Accessed October 2016].

[19] "NOX Network Control Platform," [Online]. Available: https://github.com/noxrepo/nox.

[20] T. K. J. P. B. P. M. C. N. M. S. S. Natasha Gude, "NOX: towards an operating system for networks," *ACM SIGCOMM Computer Communication Review,* vol. 38, no. 3, pp. 105-110, 2008.

[21] "OpenDaylight: Open Source SDN Platform," [Online]. Available: https://www.opendaylight.org/. [Accessed 2016 November].

[22] "Cisco Application Policy Infrastructure Controller (APIC)," Cisco, [Online]. Available: http://www.cisco.com/c/en/us/products/cloud-systems-management/application-policy-infrastructure-controller-apic/index.html.

[23] "Wide Area Network," HPE, [Online]. Available: https://www.hpe.com/us/en/networking/wan.html.

[24] "VMware NSX Network Virtualization and Security Platform," VMware, [Online]. Available: http://www.vmware.com/products/nsx.html.

[25] "OpenStack," [Online]. Available: https://www.openstack.org/.

[26] "Welcome to VirtualBox.org!," [Online]. Available: https://www.virtualbox.org/.

[27] "iPerf - The ultimate speed test tool for TCP, UDP and SCTP," [Online]. Available: https://iperf.fr/.

[28] "ab - Apache HTTP server benchmarking tool," [Online]. Available: https://httpd.apache.org/docs/2.4/programs/ab.html.

[29] S. S. Hellen Maziku, "Network Aware VM Migration in Cloud Data Centers," in *Third GENI Research and Educational Experiment Workshop*, atlanta, 2014.

[30] A. V. K. K. Vijay Mann, "CrossRoads: Seamless VM mobility across data centers through software defined networking," in *IEEE Network Operations and Management Symposium (NOMS)*, Maui, 2012.

[31] M. D. O. W. L. Y. Gaolei Li, "Deep Packet Inspection Based Application-Aware Traffic Control for Software Defined Networks," in *IEEE Global Communications Conference (GLOBECOM)*, Washington, 2016.

[32] F. W. H. Z. T.-G. Michael Jarschel, "SDN-Based Application-Aware Networking on the Example of YouTube Video Streaming," in *EWSDN '13 Proceedings of the 2013 Second European Workshop on Software Defined Networks*, 2013.

[33] C. W.-J. L.-W. L.-N. K. Nen-Fu Huang, "An OpenFlow-based collaborative intrusion prevention system for cloud networking," in *IEEE International Conference on Communication Software and Networks (ICCSN)*, Chengdu, 2015.

[34] A. B. J. R. J. Widhi Yahya, "The Extended Dijkstra's-based Load Balancing for OpenFlow Network," *International Journal of Electrical and Computer Engineering,* vol. 5, no. 2, 2015.

[35] S. S. M. Y. S. B. M. Brandon Heller, "ElasticTree: saving energy in data center networks," in *7th USENIX conference on Networked systems design and implementation*, San Jose, 2010.

[36] J. B. Tao Feng, "OpenRouteFlow: Enable Legacy Router as a Software-Defined Routing Service for Hybrid SDN," in *24th International Conference on Computer Communication and Networks (ICCCN)*, Las Vegas, 2015.

[37] S. S. M. F. G. A. N. M. R. J. Nikhil Handigol, "Plug-n-Serve: Load-Balancing Web Traffic using OpenFlow," in *ACM SIGCOMM 2009 Demo*, Barcelona, 2009.

[38] D. B. J. R. Richard Wang, "OpenFlow-based server load balancing gone wild," in *11th USENIX conference on Hot topics in management of internet, cloud, and enterprise networks and services*, 2011.

[39] H. Z. Qingwei Du, "OpenFlow-Based Dynamic Server Cluster Load Balancing with Measurement Support," *Journal of Communications,* vol. 10, no. 8, 2015.

[40] "Traffic Monitoring using sFlow," [Online]. Available: http://www.sflow.org/sFlowOverview.pdf.

[41] "WIRESHARK," [Online]. Available: https://www.wireshark.org/.

Biographical Information

The author has been in the field of computer science for more than 8 years at the time of this writing. He received Master of Science in Computer Science degree at The University of Texas at Arlington in May 2017. Before graduation, the author worked in a multinational IT company, Persistent Systems, India for 3 years. The work was in the domain of cloud computing and included working on IBM cloud stack, OpenStack, etc. This was preceded by a Bachelor of Engineering Degree in Information Technology from University of Pune, India in June 2012. The authors' research interests include cloud computing, systems engineering and software engineering.