A SOFTWARE DEFINED NETWORK IMPLEMENTATION USING MININET AND RYU

A Project

Presented

to the Faculty of

California State University Dominguez Hills

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

in

Computer Science

---

by

Carlos Ontiveros

Summer 2019

PROJECT: A SOFTWARE DEFINED NETWORK IMPLEMENTATION USING MININET AND RYU

AUTHOR: CARLOS ONTIVEROS

APPROVED:

____

Bin Tang, Ph.D
Project Committee Chair

____

Mohsen Beheshti, Ph.D
Committee Member

____

Alireza Izaddoost, Ph.D
Committee Member

ACKNOWLEDGMENTS

I would like to thank California State University and its supportive staff and faculty for mentoring me and providing me with an environment in which I was able to grow both academically and personally. I would like to thank the following professors who have been instrumental in helping me achieve academic success: Dr. Tang, Dr. Tankelevich, Dr. Beheshti, Dr. Han, and Professor McCullough.

PREFACE

This research project is a continuation of an earlier attempt to implement a software defined network (SDN) using physical hosts. The project was ambitious but unsuccessful because there were many barriers such as limited hardware resources and poor documentation. Although there were some setbacks, all hope was not lost and a new experimentation testbed was possible using virtual machines and a server. This research project was successful in implementing an SDN by using tools like Mininet and Ryu. A virtual network was created using Mininet and traffic management was handled using the Ryu controller. The algorithms designed in the research, efficiently assigned virtual machines (VM) pairs to middleboxes (MBs). Knowledge from prior experimentations was crucial in successfully connecting the controller to the switches and configuring them to reroute traffic.

TABLE OF CONTENTS

APPROVAL PAGE ……… ii

ACKNOWLEDGEMENTS ……… iii

PREFACE ……… iv

LIST OF TABLES ……… vii

LIST OF FIGURES ……… viii

ABSTRACT ……… xi

1. INTRODUCTION ……… 1

2. LOAD BALANCED MIDDLEBOX ASSIGNMENT PROBLEM (LB-MAP) ……… 5
   - 2.1. Problem Formulation ……… 5
     - 2.1.1. Network Model ……… 5
     - 2.1.2. Middlebox Model ……… 7
     - 2.1.3. Bump Off The Wire Design ……… 8
     - 2.1.4. Energy Model ……… 9
     - 2.1.5. Problem Formulation of LB-MAP ……… 10
   - 2.2. Minimum Cost Flow (MCF) Optimal Algorithm ……… 11
     - 2.2.1. Transforming a Data Center Network to a Flow Network ……… 13
   - 2.3. Three Heuristic Algorithms ……… 15

3. CONTRIBUTION ……… 18

4. BACKGROUND ……… 19
   - 4.1. Software Defined Networking ……… 19
   - 4.2. OpenFlow A Software Defined Network Protocol ……… 20
   - 4.3. Mininet A Virtual Network Emulator ……… 21
   - 4.4. Ryu A Software Defined Network Controller ……… 23

5. RELATED WORK ……… 24
   - 5.1. Data Center Topology ……… 26

6. METHODOLOGY ……… 28
   - 6.1. Testbed Settings ……… 28
   - 6.2. The Ryu Framework ……… 28
   - 6.3. Ryu Applications ……… 29
   - 6.4. Ryu & Network Traffic ……… 30
   - 6.5. Ryu Algorithm Implementation ……… 34
   - 6.6. Ryu Flow Chart & Structure ……… 43
   - 6.7. Configure Ryu & Mininet Experiment ……… 44

6.8. Start Ryu & Mininet ………..……..……..……..……..……..……..……... 45
6.9. Running Algorithms in Ryu ………..……..……..……..……..……..……... 47
6.10. Issues with Ryu ………..……..……..……..……..……..……..…………... 50
6.11. Traffic Measurement Using Iperf ………..……..……..……..……..………... 50
6.12. Iperf Server ………..……..……..……..……..……..……..……..………... 51
6.13. Iperf Client ………..……..……..……..……..……..……..……..………... 53
6.14. Parsing Output ………..……..……..……..……..……..……..……………. 54
6.15. Plotting Results ………..……..……..……..……..……..……..…………... 56

7. DISCUSSION ………..……..……..……..……..……..……..……………………. 56
7.1. Results ………..……..……..……..……..……..……..……..…………….. 56
7.1.1. Experiment 1 ………..……..……..……..……..……..……………. 57
7.1.2. Experiment 2 ………..……..……..……..……..……..……………... 60

8. CONCLUSION ………..……..……..……..……..……..……..……..…………... 63

9. REFERENCES ………..……..……..……..……..……..……..……..……………. 63

10. APPENDIX ………..……..……..……..……..……..……..……..……..………... 66
10.1. Ryu Python Implementation ………..……..……..……..……………………. 67
10.2. Ryu VM Pairs Data ………..……..……..……..……..……..……..………. 87
10.3. Ryu Superclass ………..……..……..……..……..……..………………….. 92

LIST OF TABLES

Table 1 Notations of Symbols …………………………………………………………………… 6

Table 2. Experiment 1 End-to-end Delay 10VM 3MB ………………………….……………... 57

Table 3. Experiment 1 Packet Loss 10VM 3MB ………………………………...………….. 58

Table 4. Experiment 2 End-to-end Delay 10VM 5MB ………………………..………...….. 61

Table 5. Experiment 2 Packet Loss 10VM 5MB ……………………………………..……... 62

LIST OF FIGURES

Figure 1. Data Center Two VM Pairs and Two Middleboxes……………………..……...…… 5

Figure 2. Fat Tree Network with Capacity *K=1* 10

Figure 3. New Flow Network Transformed From A Data Center... 12

Figure 4. Software Defined Networking Diagram 19

Figure 5. Ryu Architecture 23

Figure 6. Data Center with Fat Tree Topology.. 26

Figure 7. Simple Ryu Application 29

Figure 8. Ryu Python Decorator Listens to OpenFlow Events ……………............................. 29

Figure 9. Ryu Extracts Packet Information 30

Figure 10. Ryu Functions Inspect Packet 30

Figure 11. Ryu Builds PACKET_OUT Message 31

Figure 12. Ryu Match Instruction 31

Figure 13. Ryu Action Instruction 31

Figure 14. Ryu Instruction Object 32

Figure 15. Ryu OFPFlowMod Object 32

Figure 16. Complete Ryu Application.. 32

Figure 17. Ryu Class Attributes.. 34

Figure 18. Ryu Class Functions.. 34

Figure 19. Ryu Algorithm Module.. 35

Figure 20. Ryu Module Handles Flows... 35

Figure 21. Middlebox Initialized........ 36

Figure 22. VM Pairs Initialized........ 36

Figure 23. Ryu Load Variable........ 37

Figure 24. VM Based Algorithm Attributes........ 37

Figure 25. VM Based Algorithm Logic 38

Figure 26. MB Based Algorithm........ 39

Figure 27. MB-Based Algorithm Loop........ 39

Figure 28. VM+MB Based Algorithm........ 40

Figure 29. VM+MB Based Algorithm Nested Loops........ 41

Figure 30. *get_path* Module........ 41

Figure 31. Flow Chart Packet-In........ 42

Figure 32. Flow Chart Controller........ 43

Figure 33. Mininet Loading Network........ 44

Figure 34. Ryu Detects Network........ 44

Figure 35. Mininet Pingall........ 46

Figure 36. Algorithm checks if flow is in VM pairs list........ 46

Figure 37. Ryu Handles Traffic........ 47

Figure 38. MB Based Algorithm Initiated........ 47

Figure 39. MB Based Algorithm Finished........ 48

Figure 40. VM+MB Based Algorithm Initiated........ 48

Figure 41. VM+MB Based Algorithm Finished........ 49

Figure 42. Host Runs Iperf Server........ 51

Figure 43. Host *h13* Runs Iperf Report…………………………………………………………….. 52

Figure 44. Host *h13* Runs Iperf Client…………………………………………………………... 52

Figure 45. Iperf Running in Client Mode………………………………………………………... 53

Figure 46. Parsed Iperf Log data……………………………………………………………….... 54

Figure 47. End-to-end Delay for 10 VMs , 3 MBs , 100Mbps Bandwidth……………………... 56

Figure 48. Packet Loss for 10 VMs, 3 MBs, 100Mbps Bandwidth...58

Figure 49. End-to-end Delay for 10 *VMs*, 5 *MBs*, 100Mbps Bandwidth.................................... 59

Figure 50. End-to-end Delay Closeup.. 60

Figure 51. Packet Loss for 10 VMs, 5 MBs, 100Mbps Bandwidth...61

ABSTRACT

Middleboxes are not only great for providing network services but they also facilitate network management in cloud data centers. Software Defined Networks (SDN) and Network Function Virtualization (NFV) facilitate this process. Virtual machine (VM) communication must traverse middlebox sequences for policy requirements. Data centers are challenged with load-balancing middleboxes and minimizing VM communication costs. This problem is known as the Load-Balanced Middlebox Assignment Problem (LB-MAP). Three algorithms were proposed to solve this problem. They are heuristics and designed to perform near optimal but they have not been implemented in a real network environment. One of them, VM+MB, is a combination of the other two and is expected to be the best. This project is the first to implement all heuristics in an SDN testbed and then analyze and measure the results. From the experiments conducted the VM+MB performed best in terms of end-to-end delay and packet loss.

1. INTRODUCTION

Middleboxes are network appliances which act as intermediary computer networking devices. These network appliances are also called network functions because they serve specific functions for computer networks. Some of these functions are to transform, inspect, filter, or manipulate traffic and forward them to other devices. These middleboxes are widely deployed in enterprise networks such as data centers and play a crucial role in ensuring network security through the use of firewalls, intrusion detection systems (IDS). They also ensure high performance through the use of load balancers, and reduce bandwidth cost through the use of WAN optimizers [1].

According to one recent study, the number of middleboxes have grown to almost the same number of routers in large networks. Traditional middleboxes are physical devices that are usually proprietary, closed and expensive. They are built for specific purposes such as firewalls and are not easily configurable or always compatible with other proprietary devices [1]. Deploying and operating them can be costly because of initial capital investment and the cost of maintaining them. As network resources increase, acquiring more devices is necessary but this will also require more physical space as well as power consumption. When equipment becomes outdated and can no longer be supported, the so-called network ossification problem emerges [12].

In order to provide some relief, network function virtualization (NFV) has been proposed. NFV is a networking virtualization technology that allows operators and administrators to implement middleboxes in software rather than purpose-build hardware. In terms of

management, there are large improvements because now the software-based middleboxes may be instantiated anywhere in the network without having to worry about new equipment purchases nor need any new services from an operator [1].

Software Defined Networking (SDN) further alieves the burden of network management by decoupling the control plane from the forwarding plane. SDN works as a complementary technology to NFV because it moves the management functions from the hardware to a software format that is orchestrated by a centralized controller. The centralized controller utilizes a new open protocol known as OpenFlow to communicate with middleboxes, which are usually implemented inside switches, and dynamically configure them [1].

Cloud data centers are thriving thanks to the facilitation SDN and NFV bring into network management. Hardware resources such as CPU cycles, and memory are divided into smaller isolated computing units known as a virtual machine (VM), which can be rented to users for a fee [1]. These resources are and ideal platform for researchers to implement SDN and NFV experiments. Deployed middleboxes ensure that applications and services hosted in data centers are secure, maintained, and perform optimally. The configuration and synchronization of these middleboxes is possible due to SDN and NFV [12].

Network policies require traffic to traverse specific sequences of middleboxes in order to provide security and performance guarantees for the applications and services being hosted. Different applications have different requirements thus the expectations from cloud service providers and the commitments from cloud users are usually described in the Service Level Agreement (SLA) in order to maintain some sort of accountability. A data center's ability to fully satisfy the SLA has become an important measurement for both efficiency and efficacy.

Data centers who strive to transcend these measurements are known as policy-driven data centers. One example of policy, may be the case where network traffic from a communicating VM must go through an IDS and a load balancer, in order to first filter out malicious traffic and then efficiently route it to avoid congestion [1].

Middlebox management may be facilitated by SDN and NFV but it is a complex operation with a dilemma. It is important to note that there are limits to the hardware resources such as CPU cycles and memory that are available for middleboxes. Additionally, the special memory is used by SDN switches implement middleboxes is very small and expensive. This memory is known as ternary content addressable memory (TCAM) and it is considered perfect for storing MAC addresses and faster than RAM. TCAM consumes a lot of power which increases temperature in data centers and thus also increasing the need for cooling. Since the memory size is still small, only a small amount of forwarding rules may be stored [1]. Whenever middleboxes perform services such as deep packet inspection, complex and extensive processing must be involved. These limitations and resource usage are reasons why middleboxes fail and then lead to packet loss, traffic delays, and waste of power. To prevent this, network administrators must balance the load in middleboxes so that they are not overworked but also not underworked and sit idly wasting energy [12].

NFV and SDN allows administrators to replicate a middlebox as many times as needed in order to achieve load-balancing and fault tolerance. Each clone of a middelbox is known as an *instance*. In this research project the focus lies in policy-driven data centers where only one type of middlebox is considered but multiple instances of that same middlebox can exist. The purpose of a single middlebox will serve a specific need such as load balancing or security. Each VM

communication pair must traverse a middlebox and if that middlebox is full or incurs a heavy cost then another instance will be utilized [1].

The goal of this research project is to consider the situation where a set of VM communication pairs is given as well as a set of middlebox software instances and each VM pair must be assigned to a middlebox instance while considering energy cost and capacity constraints. The total VM communication cost when traversing a middlebox must be kept as low as possible and the maximum load capacity of the middlebox must be balanced. This research will refer to this problem as the Load-Balanced Middlebox Assignment Problem (LB-MAP) [1]. This problem is equal to the well known minimum cost flow problems (MCF) in a transformed flow network. MCF can be solved efficiently and optimally. A suit of efficient heuristic algorithms were designed viz. VM-Based, MB-Based, and VM+MB-Based. Through testing and experimentation it can be seen that all heuristics perform close to the optimal minimum cost flow algorithm but VM+MB-Based performs best from all heuristics. This research is one of the first to implement a testbed in which VM communication cost and load-balancing of middleboxes is addressed in policy-driven data centers [1].

2. LOAD BALANCED MIDDLEBOX ASSIGNMENT PROBLEM (LB-MAP)

2.1. Problem Formulation

2.1.1. Network Model

The data center graph will be modeled as an undirected general graph *G(V,E)*. Set $V = V_p \cup V_s$ includes the set of physical machines V_p (*PMs*) and set of switches V_s (edge, aggregate, and core). The set of edges, E, includes links between switches to switches and links

between switches to PMs. The data center contains l VM pairs expressed in set

$P = \{(v_1, v_1'), (v_2, v_2'), ..., (v_l v_l')\}$ where v_i and v_i' $(1 \leq i \leq l)$, are referred to as the *source* VM and the *destination* VM, respectively. Each VM v is located inside a PM, denoted as $S(v_i)$ and $S(v_i')$ where they are referred to as the *source* PM and *destination* PM of (v_i, v_i'), respectively. A PM is capable of storing both source VM and destination VM simultaneously [1]. Figure 1 shows an example of two communicating VM pairs (v_1, v_1') and (v_2, v_2'). All notations used in this paper are listed in Table 1.

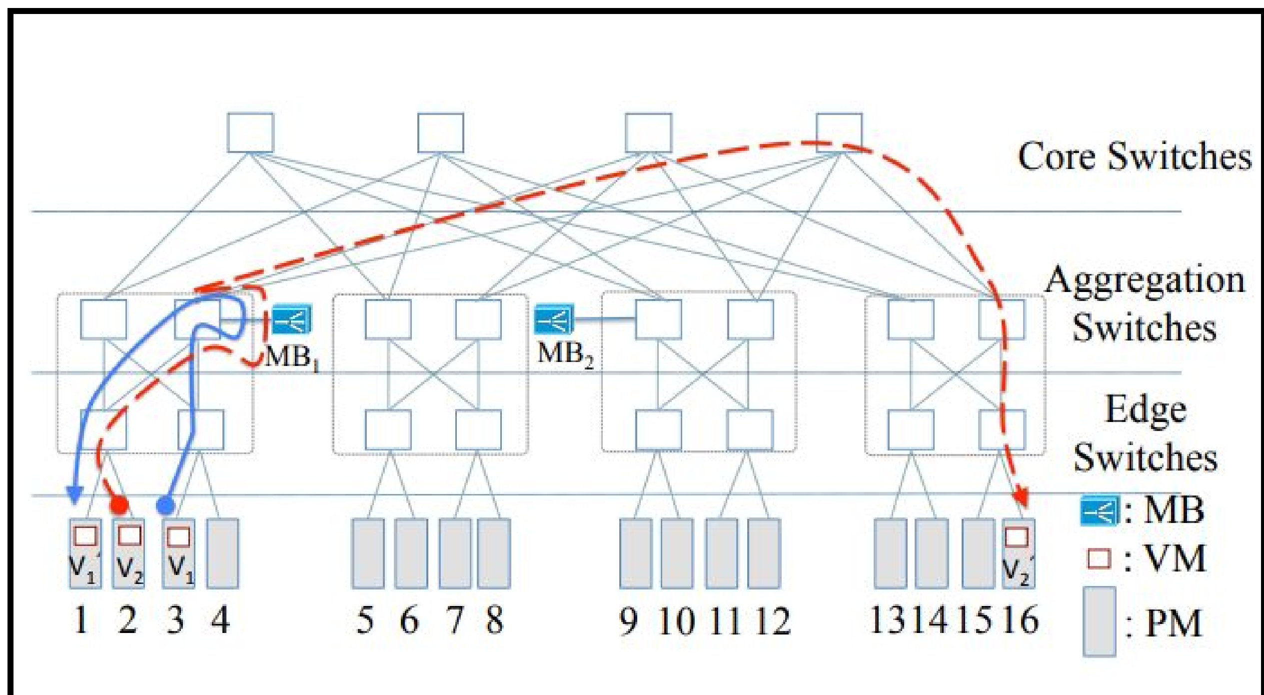

Figure 1. Data Center Two VM Pairs and Two Middleboxes [1]

Table 1. Notations of Symbols

Notation	Explanation
V_p	The set of physical machines (PMs) in the data center
V_s	The set of switches in the data center
P	The set of l VM communication pairs, (v_i, v_i'), $1 \le i \le l$
M	The set of m middlebox instances, mb_j, $1 \le j \le m$
$S(v)$	The PM where VM v is stored
$sw(j)$	The switch where mb_j is located
K	The capacity of each middlebox instance
r_e, r_a, r_c	The energy consumption on edge, aggregate, and core switch
$c(i,j)$	The energy cost between PM (or switch) i and j
$c_{i,j}$	The energy cost when VM pair (v_i, v_i') traverses mb_j
C^p	The total energy cost for an MB assignment function p

2.1.2. Middlebox Model

Load balancers inside data centers are known to have the highest failure probability according to Gill et al [1]. The high failure rate is due to faults arising from configuration errors and bugs in the software. Other errors arise from application-specific integrated circuit (ASIC) and memory. In this research project the general case of only one type of middlebox such as a load balancer will be considered but multiple copies of that same middlebox will be present. Cases where different types of middleboxes with multiple copies exist will not be studied here but discussed in the future work section [1].

During experimentation, m software-based middleboxe instances $M=\{mb_1,mb_2,...,mb_m\}$ were placed inside the data center by installing them inside corresponding switches. Assume that middlebox mb_j $(1 \leq j \leq m)$ is located at switch $sw(j) \in V_s$. Policy specification require each VM pair (v_i,v_i') to traverse a middlebox instance. Each middlebox instance will have a capacity constraint where at most K VM pairs will be able to traverse it. Figure 1 contains an example where there are two load balancers MB_1 and MB_2 and the capacity of each is K=2 [1].

2.1.3. Bump Off The Wire Design

Traditionally middlebox appliances have been deployed using an inline "bump-in-the-wire" design. Cases such as these, involve dedicated middlebox hardware appliances that are physically plugged into the network. There are some drawbacks, such as the fact that all traffic running in the network must pass through these middleboxes whenever they pass through the switch in which they are installed in. In certain situations, this may be unnecessary and a waste of resources since certain application specific traffic may not need to traverse this middlebox and only use up the capacity of the middlebox. Dynamically configuring middleboxes to process traffic was very tedious and time consuming in traditional middleboxes. Sometimes there are multiple copies of the same physical middlebox in the network. In cases like this, resources may be wasted if the traffic only needs to traverse the middlebox once but instead passes multiple times and thus increase processing load and energy consumption. This research project will implement the "bump-off-the-wire" design [1]. This approach was also improved by Zhang [5]. In these designs middleboxes are take off the physical networks and implemented as software modules or *VMs* installed inside a PM which are plugged into each switch. As traffic

traverse switches they will explicitly forward them to an attached middlebox if necessary. Since there are low latency links and minimal performance overhead, data center networks are suitable for this traffic redirection [1].

2.1.4. Energy Model

Power consumption in the data center network will be measured by the total amount of switches that traffic passes through. The notations r_c, r_a, r_e will be used to denote the power consumption on a core switch, aggregate switch, and edge switch, respectively, whenever traffic from a VM communication passes through them. There are two energy consumption models that are currently adopted in cloud data center network research [1].

Uniform Energy Model

This model will measure energy consumption as the minimum number of switches that VM communication traffic passes through [1]. In other words, any switch that forwards traffic will cost the same amount so every core, aggregate, or edge: $r_c = r_a = r_e$. As an example, consider Figure 1, if $r_c = r_a = r_e = 1$ then the power consumption between v_1 and v_1' and between v_2 and v_2' are 3 and 5 respectively.

Skewed Energy Model

This model is based on the simple idea that higher layer switches will consume a greater amount of energy than the lower ones. So, $r_c > r_a > r_e$. For example, in Figure 2, if r_e=1, r_a=5, and r_c=10 the power consumption between v_1 and v_1' and between v_2 and v_2' are 7 and 22, respectively [1].

EXAMPLE 1

In Figure 1 the capacity of each middlebox is $K=2$ and each VM pair must traverse at most one instance. Here (v_1,v_1') traverses MB_1 with a cost of 3 and (v_2,v_2') also traverses MB_1 with a cost of 5 which results in a total cost of 8 which is ideal because it achieves the minimum cost under the uniform energy model [1].

2.1.5. Problem Formulation of LB-MAP

Let $c(i,j)$ denote the minimum energy consumption between PM (or switch) i and j. Let $c_{i,j}$ be the minimum power consumption for VM pair (v_i, v_i') when it is assigned middlebox mb_j. Then,

$$c_{i,j} = c(S(v_i), sw(j)) \ + \ c(sw(j), S(v_i')). \quad (1)$$

The load balanced middlebox assignment function is defined as $p : P -> M$, which means that VM pair $(v_i, v_i') \in P$ is assigned to middlebox instance $p(i) \in M$. Given any middlebox assignment function p, the power consumption for VM pair (v_i, v_i') is then

$$c_{i,p(i)} = c(S(v_i), sw(p(i))) + c(sw(p(i)), S(v_i')). \quad (2)$$

The total energy consumption of l VM pairs with middlebox assignment p as C^p. Then

$$C^p = \sum_{i=1}^{l} c_{i,p(i)} = \sum_{i=1}^{l} (c(S(v_i), sw(p(i))) + c(sw(p(i)), S(v_i'))) \ . \quad (3)$$

Let p_{min} be the assignment that produces the minimum total energy consumption among all the middlebox assignments P, in other words $C^{p_{min}} \leq C^p, \ \forall p \in \mathrm{P}$. It is the objective of the LB-MAP to find such a p_{min} under the constraint that at most K VM pairs can be served by any middlebox instance:

$$|\{1 \leq i \leq l | p(i) = j\}| \leq K, \ \ \forall \ j, 1 \leq j \leq m \ .$$

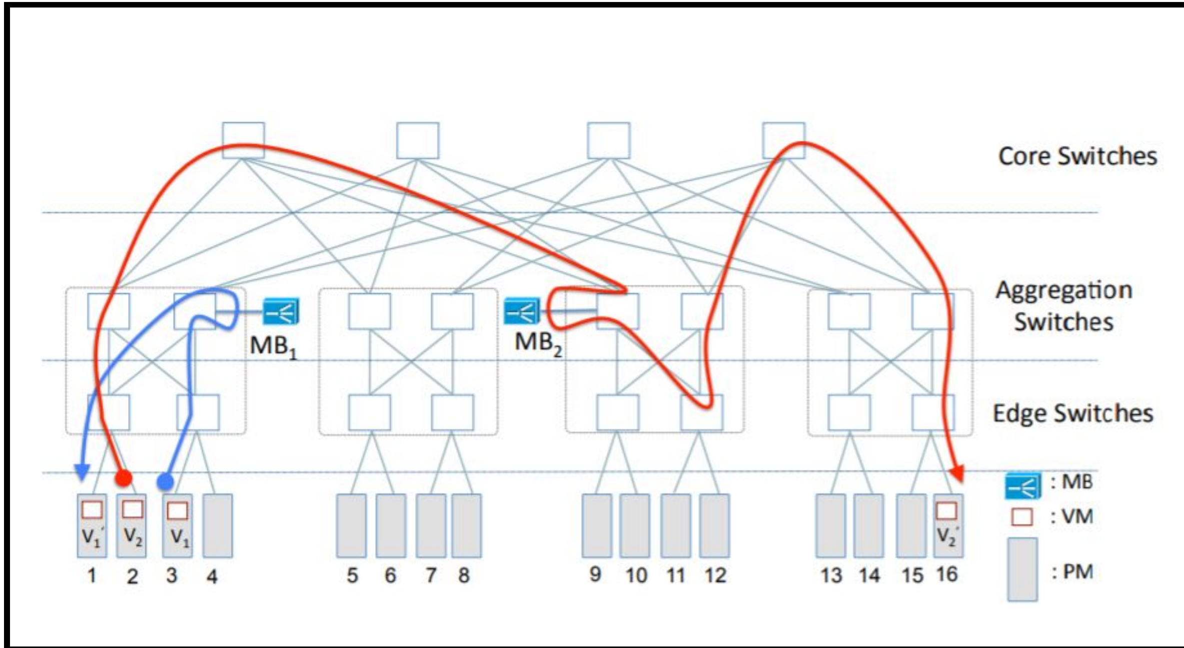

Figure 2. Fat Tree Network with Capacity *K=1* [1]

EXAMPLE 2

Consider Figure 2, which is similar to Example 1, but instead of having K=2, set the capacity to K=1. The VM pairs can no longer traverse the same middlebox simultaneously. VM pair (v_1, v_1') traverses MB_1 (shown in blue color) with a cost of 3 while (v_2, v_2') traverses MB_2 (shown in red color) with a cost of 9 and result in a total cost of 12 under uniform energy model.

2.2. Minimum Cost Flow (MCF) Optimal Algorithm

This section will show that LB-MAP is equivalent to the minimum cost flow problem [1]. The minimum cost flow problem will be presented along with its algorithm and then the data center network will be transformed to the flow network in order to show the equivalency.

<u>Minimum Cost Flow Problem (MCF)</u>

The MCF problem is an optimization problem in which the cheapest possible way of sending a flow from one node to another node is found [1]. The MCF can be represented by a directed graph $G = (V, E)$. Each $(u, v) \in E$ has a capacity *c(u,v)* which is the maximum amount of flow that can pass through that edge. For every single amount of flow that passes through $(u, v) \in E$ there is an associated cost $d(u, v)$. At two ends of the graph there exist a source node $s \in V$ with supply b and a sink node $t \in V$ with demand b. When a flow is present in an edge it is denoted $f(u, v)$ and mapped as $f : E \rightarrow \mathbb{R}^{+}$ and there are two constraints [1]:

1) Capacity constraint: The flow in an edge cannot exceed the capacity:

   $f(u, v) \leq c(u, v), \ \forall (u, v) \in E$.

2) Fow conservation: The same amount of flow that enters a node is same amount that must

   exit. $\sum_{u \in V} f(u, v) = \sum_{u \in V} f(v, u) \forall v \in V \setminus \{s, t\}$. The net flow out of source *s* is

   $\sum_{u \in V} (f(s, u) - f(u, s)) = b$ and the net flow into sink node *t* is $\sum_{u \in V} (f(t, u) - f(u, t)) = b$.

The goal of the MCF is to find a flow function *f* what will yield the smallest cost possible, that is:

$$min \sum_{(u,v) \in E} (d(u, v) \cdot f(u, v)). \quad (4)$$

MCF Algorithms

The MCF can be solved efficiently using combinatorial algorithms. Some of these algorithms include cycle-canceling, successive shortest path, out-of-kilter algorithm, but in this research project it is the scaling push-relabel algorithm by Goldberg that will be adopted [1]. The algorithm's time complexity for any flow network is $O(a^2 \cdot \ b \cdot log(a \cdot c))$ where *a*, *b*, and *c*

represent the number of nodes, number of edges, and the maximum edge capacity in the flow network, respectively [1].

2.2.1. Transforming a Data Center Network to a Flow Network

Transformation of the data center network *G=(V,E)* into a new flow network *G'=(V',E')* will be discussed next. There are five steps to complete:

Step 1. $V' = \{s_0\} \cup \{t_0\} \cup P \cup M$

Step 2. $E' = \{(s_0, (v_i, v_i')) : (v_i, v_i') \in P\} \cup \{((v_i, v_i'), mb_j) : (v_i, v_i') \in P, mb_j \in M\} \cup \{(mb_j, t_0) : mb_j \in M\}$

Step 3. For each edge $(s_0, (v_i, v_i'))$ set capacity to 1, and cost to 0. For each edge (mb_j, t_0) set its capacity to K and cost to 0.

Step 4. For each edge $((v_i, v_i'), mb_j), (v_i, v_i') \in P, mb_j \in M$ set its capacity as 1 and its cost as $c_{i,j}$ which is the cost of VM pair (v_i, v_i') is assigned middlebox mb_j.

Step 5. Set both supply and demand of s_0 and t_0 to l .

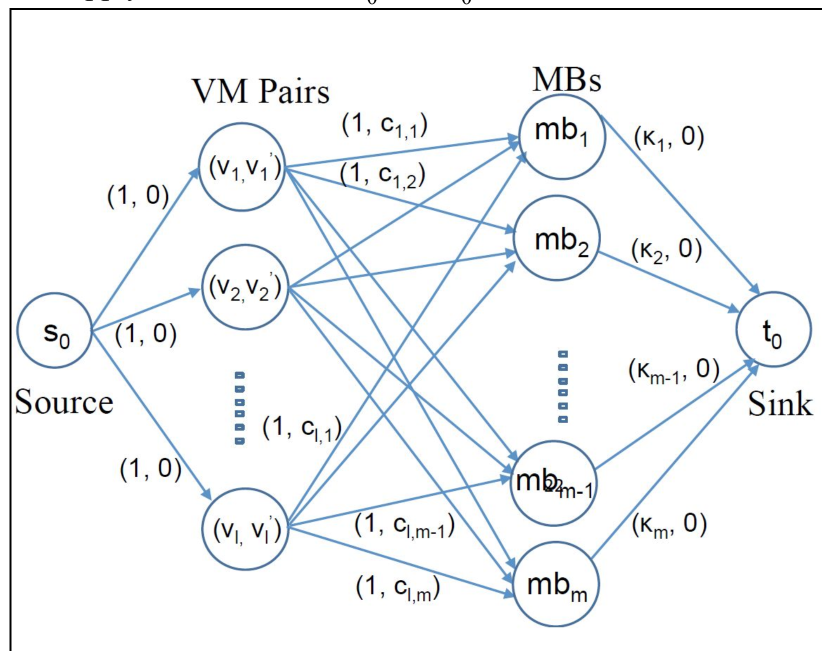

Figure 3. New Flow Network Transformed From A Data Center [1]

The technique to convert the data center to a network flow is similar to [30]. Once the transformation is complete it will be passed to the MCF algorithm discussed earlier, which will then output the load-balanced middlebox assignment which yields the minimum power consumption for l VM pairs. Each VM pair will be assigned a middlebox without violating its capacity constraint [1].

<u>Time Complexity for LB-MAP Algorithm</u>

Time complexity for LB-MAP is made up of two crucial steps: the graph transformation, and MCF algorithm. The graph transformation which consists of calculating costs $c_{i,j}$ and $c(i,j)$ takes $O(|V|^3)$. Creating the edges will take $O(l + m \cdot l + m)$ so total construction complexity is $O(|V|^3 + m \cdot l)$. The MCF algorithm used is the scaling push-relabel algorithm which has a time complexity of $O(a^2 \cdot b \cdot log(a \cdot c))$ where the number of nodes, number of edges, and capacity are represented by a, b, and c, respectively. In the transformation graph we have nodes $|V'| = 2 + l + m$ and $|E'| = l + m \cdot l + m$ and capacity K. Thus the complexity for MCF is $O((l + m)^2 \cdot l \cdot m \cdot log((l + m) \cdot K))$. Putting both parts together give the LB-MAP algorithm a complexity of $O(|V|^3 + (l + m)^2 \cdot l \cdot m \cdot log((l + m) \cdot K))$.

Theorem 1: LB-MAP is equivalent to the minimum cost flow problem.

Proof: Applying the minimum cost flow algorithm to the flow graph will a) assign each of the l VM pairs to exactly one middlebox while b) not violating the middlebox capacity constraints and c) achieve the minimum energy consumption in the graph for all l VM pairs.

The transformed graph will ensure that all l VM pairs will be assigned a middlebox instance. Since the supply at s_0 is l and has an edge with every VM pair, the flow will pass

through every VM pair. Note that the capacity on each edge is one so all flow must go through every edge from s_0 to every VM pair. Due to conservation constraints, each flow must then leave every VM pair and go to a middlebox. The capacity on every edge $((v_i, v_i'), mb_j)$ is one so it will be able to accommodate the flow from any VM pair.

All VM pair assignments to middleboxes respect the capacity constraints. Step 3 says that the capacity for edges (mb_j, t_0) is K so no more than K amount of VM pairs will pass flow through any middlebox.

Total cost of the flow in the transformed network is calculated from edge $((v_i, v_i'),\ mb_j)$. It is the only edge with a cost which is $c_{i,j}$ and it represents the minimum energy consumption when VM pair (v_i, v_i') is assigned to middlebox mb_j. All other edges in the network have a zero cost which indicates that only the VM communication cost is considered in the minimum cost flow. The minimum cost flow algorithm thus sends l amount of flow from s_0 to t_0 and shows that the corresponding VM communication costs are the minimum.∎

2.3. Three Heuristic Algorithms

Three polynomial-time greedy algorithms are proposed for comparison purposes. Each algorithm takes place in rounds and uses different criteria to find an assignment between a VM pair and a middlebox.

VM-Based Algorithm. The VM-Based algorithm assigns VM pairs to the middlebox that yields the lowest power consumption and also satisfies the middlebox capacity constraints. Once a middlebox has been assigned to a VM pair its load capacity will be updated. Finding all minimum energy consumption paths between all pairs of PMs takes $O(|V|^3)$. Assigning each VM

pair to a middlebox takes $O(l \cdot m)$ and so therefore the time complexity for the algorithm is

$O(|V|^3 + l \cdot m)$.

Algorithm 1: VM-Based Algorithm
Input: A data center $G=(V,E)$ with l VM pairs and m MBs
Output: Total energy cost C for all the l VM pairs.
Notations:
i: index for VM pair
j: index for middlebox instances
$load(j) = 0$: current load for mb_j
c^i_{min} : the minimum energy cost for VM pair (v_i, v_i')
j^*: middlebox mb_j assigned to (v_i, v_i')

1. $C=0$;
2. *for (i=1 to l)*
3. c^i_{min} = infinity;
4. *for (j=1 to m)*
5. *if* ($c_{i,j} \le c^i_{min}$ *and* $load(j) < K$)
6. $c^i_{min} = c_{i,j}$;
7. $j^*=j$;
8. **end if;**
9. **end for;**
10. $load(j^*)$++;
11. $C = C + c^i_{min}$;
12. **end for;**
13. **RETURN** C.

MB-Based Algorithm. In the MB-Based algorithm, each MB will be assigned K VM pairs. The assigned VM pairs will be those that produce the minimum energy consumption when traversing that particular middelbox. The running time is $|V|^3 + m \cdot (l + l \cdot lg(l)) + K$ which is

$O(|V|^3 + m \cdot (l \cdot lg(l)))$.

Algorithm 2: MB-Based Algorithm
Input: A data center $G=(V,E)$ with l VM pairs and m MBs
Output: Total energy cost C for all the l VM pairs.

Notations:

i: index for VM pair
j: index for middlebox instances
X_j: the set of VM pairs assigned to mb_j
assigned[i]: set to true if VM pair (v_i, v_i') has been assigned a MB, else it is false

1. $C=0$;
2. *for (i=1 to l)*
3. *assigned[i]=false;*
4. **end for;**
5. *for (j=1 to m)*
6. $X_j = \oslash$
7. *for(i=1 to l)*
8. *if(assigned[i]==false)*
9. $X_j = \{(i, c_{i,j})\} U X_j,$ $assigned[i] = true;$
10. **end if;**
11. **end for;**
12. Sort X_j in non-descending order based on values for $c_{i,j}$;
13. $X_j = \{(x_1, c_{x_1,j}), (x_2, c_{x_2,j}), (x_3, c_{x_3,j}), ...\},$
14. *where* $c_{x_1,j} \le c_{x_2,j} \le c_{x_3,j} \le ...;$
15. *for (k=1 to K)*
16. $C = C + c_{x_k,j};$
17. **end for;**
18. **end for;**
19. **RETURN** C.

VM+MB-Based Algorithm. In each round, a VM pair is assigned to an MB instance such that it produces the minimum energy consumption not only among all MBs with available capacity but all other available VM pairs. The time complexity is $O(|V|^3 + m \cdot l)$.

Algorithm 3: VM+MB-Based Algorithm.
Input: A data center $G=(V,E)$ with l VM pairs and m MBs
Output: Total energy cost C for all the l VM pairs.
Notations:

i: index for VM pair
j: index for middlebox instances
$load(j)=0$: current load for mb_{j*}
i^*, j^*: i^{*th} VM pair is assigned to mb_{j*} in each round,
c_{min} : the minimum energy obtained in each round

```
1.  C=0;
2.  for (i=1 to l)
3.      assigned[i]=false;
4.  end for;
5.  while (unassigned VM pairs exist)
6.      c_min = infinite ;
7.      for (i=1 to l)
8.          if(assigned[i]==false)
9.              for (j=1 to m)
10.                 if (load(j) < K and c_i,j ≤ c_min)
11.                     c_min = c_i,j ;
12.                     j*=j, i*=i;
13.                 end if;
14.             end for;
15.         end if;
16.     end for;
17.     assigned[i*]=true, load(j*)++;
18.     C=C+c_min;
19. end while;
20. RETURN C.
```

3. CONTRIBUTION

This research project is the first to analyze the load balanced middlebox assignment (LB-MAP) problem using an SDN testbed. The testbed has been implemented using a network emulator called Mininet and an SDN controller called Ryu. Three heuristic solutions have been proposed by B. Tang [1] that are near optimal in theory but no quantifiable metrics have been produced. Prior implementations only compiled the algorithms with simple test cases that did not consider real network environments variables such as link bandwidth and packet rate. Each algorithm was implemented in the testbed and then their performances were compared to find the best solution.

Data centers must route traffic to their correct destinations but depending on their service level agreement (SLA), some traffic may be required to traverse one or more middleboxes (MBs) before reaching its destination. Mininet is a powerful network emulator used by academic institutions for SDN research. Mininet was configured to create a data center with a fat tree topology. Ryu is a powerful SDN controller that is compatible with Mininet. Resources are limited and for data centers to be competitive they must eliminate waste. Waste can be eliminated by balancing the load between all middleboxes, and finding the shortest path to travel from source to destination to decrease energy consumption. The heuristics perform traffic management for data center networks.

4. BACKGROUND

4.1. Software Defined Networking

Software Defined Networks (SDN) is a network architecture in which the data plane and the control plane are decoupled or separated. The architecture of SDN involves three primary layers which are the application layer, the controller layer, and the forwarding layer as shown in figure 4. The application layer includes network functions such as firewalls, load balancers, and intrusion detection systems.

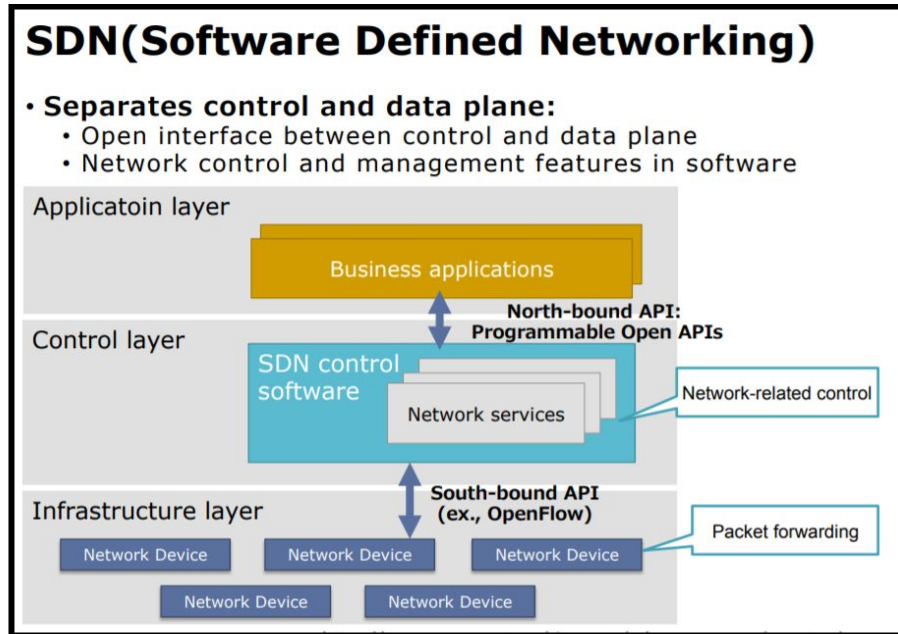

Figure 4. Software Defined Networking Diagram [27]

The application layer communicates with the controller layer via the Northbound interface. The Northbound interface may be implemented using different protocols such as Frenetic, REST, or an API. The controller layer communicates the network functions to the forwarding plane via the Southbound interface. The Southbound interface utilizes the OpenFlow protocol to communicate between the control layer and the forwarding layer. The forwarding layer takes care of routing network packets. The forwarding layer is composed of network appliances such as routers and switches [12].

4.2. OpenFlow A Software Defined Network Protocol

OpenFlow is the protocol that virtual and physical switches use to communicate in a software defined network. SDN controllers manage switches through a secure channel using the OpenFlow protocol. The controllers listens on a specific port and when switches enter the

network they send messages to the port. These messages send the controller information such as the protocol version the switch supports, MAC address, and hardware type. OpenFlow is utilized in the southbound layer which connects the forwarding plane (i.e. physical or software network devices) to the control plane. The forwarding plane is sometimes referred to as the data plane and the control plane is also the management plane [12].

4.3. Mininet A Virtual Network Emulator

Mininet is an open source network emulator that allows researchers to create virtual networks with many switches and hosts in one single machine. Mininet is Python based and runs a collection of end-hosts, switches, routers, and links on a single Linux kernel. Lightweight virtualization is used to make one computer look like a complete system running the same kernel, system, and user code. A host in mininet can run just like a real machine so it can run linux programs and run bash scripts. These hosts have virtual ethernet interfaces which can send and receive packets like a normal ethernet card [21]. Mininet can be installed in a desktop, a server, or in a machine in the cloud. Mininet contains a command line interface (CLI) and an application programming interface (API) so researchers can configure the network anyway they want. It speaks the OpenFlow protocol and is ideal for Software Defined Network (SDN) implementations [20].

Mininet allows hosts to send traffic to another host in the network or a host outside the network. By default Mininet creates a simple network with two hosts, a switch, and uses the default SDN controller. The sample network is small but powerful because it allows users to view interhost communications from different layers. When hosts pings one another, the Mininet

CLI displays packets being sent and received. One can also open up Wireshark and observe communication patterns just as a live physical network. Administrators can view traffic at the regular IPv4 level and also at the TCP layer where OpenFlow operates [20].

Developers can quickly generate any desired network because they can use the CLI or Python scripts to instantiate topologies. Mininet can modify parameters such as the size of the network, the SDN controller, and link bandwidth. Python scripts are a great way to create large and complex networks or implement special labeling so that traffic patterns are easier to identify or manage [20].

The Mininet API gives researchers the power to create any network. The main class to create and manage a network is *Mininet* and the base class for creating topologies is *Topo*. Functions like *addSwitch()*, *addHost()*, and *addLink()* add necessary components to a network. They add switches, hosts, and let developers specify a name. Function *addLink()* creates network links between any two devices like hosts or switches. The network can be started or stopped by using the functions *start()* and *stop()* respectively [20].

There are some limitations to Mininet which impact the speed at which packets may be sent. For example, if mininet runs on a server with a 2.0 Ghz CPU and can handle about 1Gbps of traffic then virtual hosts will have to share these resources and cannot be assigned the entire CPU and bandwidth that is actually available. It is crucial to realize that hosts do not contain their own dedicated switching hardware. When hosts are created they will all share the same memory and can thus run scripts from the local machine. Since Mininet is not a simulator it does not have a strong notion of real time and thus timing measurements are based on real time. For

improved time accuracy, especially under heavy load, it is important to carefully limit the CPU bandwidth of the Mininet hosts [21].

4.4. Ryu A Software Defined Network Controller

Ryu is an open source software defined networking (SDN) framework that give users the ability to control OpenFlow enabled devices [23]. Ryu is the Japanese word for “flow” and its purpose is to keep network traffic flowing smoothly. Ryu is written entirely in Python, open source, and maintained by an active community. Ryu supports other networking protocols such as NETCONF and OF-config. Ryu supports various versions of OpenFlow such as 1.0, 1.2, 1.3, 1.4, and 1.5. Ryu works with OpenFlow switches from various organizations such as Open vSwitch, Centec, Hewlett Packard, IBM, and NEC [28].

Developers can create new network management and control applications with Ryu by using its application programming interfaces (APIs). Ryu can communicate information down to switches and routers using southbound APIs and up to the applications and business logic using northbound APIs. Figure 5 shows the northbound and southbound layers that Ryu interacts with. The northbound is where the logic is found. These are software implementations of standard switches and hubs or other appliances used to load balance the network [29].

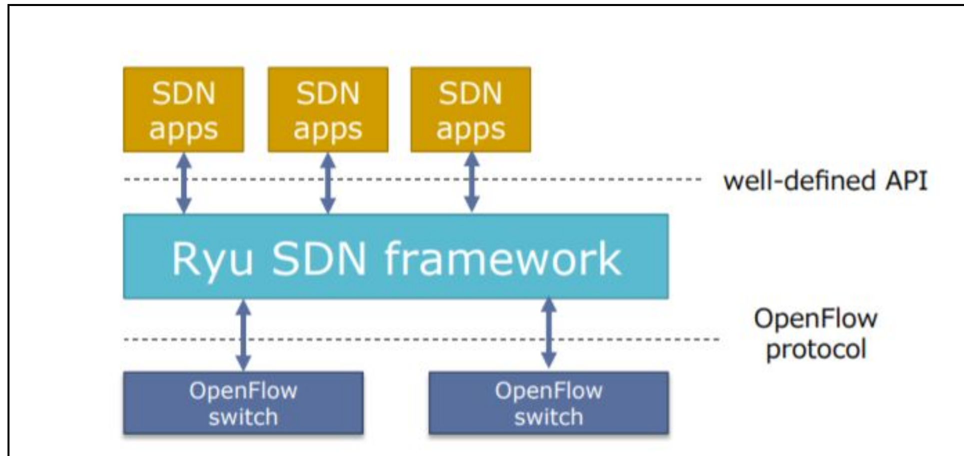

Figure 5. Ryu Architecture [28]

Ryu is powerful because it has a global view of all network devices and can utilizes logic from Python scripts to calculate efficient routes and deliver traffic to its destination. Ryu can act as a traditional switch or developers can programmatically alter network behavior by configuring switches and routers. Thanks to the controller, developers do not have to understand the details of the OpenFlow protocol nor its syntax in order to configuring network devices, instead; they only have to focus on the logic of the flow in network [28].

5. RELATED WORK

One of the first works to discuss middlebox management in data centers was the architectural work of Joseph et al. [1]. He proposed a new layer-2 switching layer that detects different types of traffic and then forwards them through different sequences of middleboxes. It was shown that their technique did traverse middleboxes correctly and efficiently. Further system designs and algorithmic challenges were addressed by Qazi et al. [4] Efficient data play support for policy composition was proposed and they also unified switch and middlebox

resource management. An entirely new and different design for middlebox management was proposed by Sekar in which individual middleboxes and their management were consolidated to multiplex hardware resources and then reused processing modules [1].

Managing middleboxes in enterprise networks is no small feat but an interesting idea has emerged from researcher Sherry, in which it has been proposed to outsource all of the middlebox processes and computations to other cloud service providers as a solution [1]. A software defined middlebox networking framework was realized by Gember et al. [1] in which he represented, manipulated, and controlled the middlebox states. Another framework for SDN-enabled services was presented by Zhang et al. [5] and it dynamically routed traffic through any sequences of middleboxes. Additionally, they proposed an algorithm that would find the optimal location for any middlebox.

In order to create effective solutions to the middlebox problem, researchers have identified deep theoretic roots. Liu has researched the middlebox placement problem by considering network information and policy specifications in order to determine the optimal locations for middleboxes so that end to end delay and bandwidth consumption is optimized [9]. It turns out that this problem is NP hard and so, instead, heuristics were proposed as solutions. Online primal-dual algorithms were designed by Li et al. [1] to deal with the policy-aware cloud application embedding problem. Cuie et al. [1] studied dynamic virtual machine consolidation and dynamic network policy (re)allocation to meet both efficient data center resource management and middlebox traversal requirements.

Among the research covered thus far, the issue optimal load-balancing for middleboxes in data centers was specifically addressed by Qazi et al. [4]. An online integer linear program

(ILP) was formulated to minimize the maximum of the middlebox load across the network but depending on the size of the network this process can become very lengthy. However, the LB-MAP problem discussed in this research has different goals and different solution techniques. Minimizing VM communication costs and satisfying middlebox load capacities are the goals of LB-MAP. The proposed solutions include the minimum cost flow solution and a suite of time-efficient heuristic algorithms [1]. In one recent research experiment, a programmable middlebox was introduced by Tu et al. [1] The middlebox was programmed to distribute data center traffic more evenly and thus enhance bandwidth utilization and reduce traffic delay. The middlebox communicated to an SDN controller that recorded traffic distribution and traffic load and then performed the necessary load balancing. This solution is effective but unlike the LB-MAP problem, it did not consider middlebox capacity constraints nor try to minimize VM communication energy costs [1].

5.1. Data Center Topology

The type of networks that this research will focus on are data centers with fat tree topologies because they are the preferred choice when interconnecting commodity hardware [3]. A fat tree network is a variation of the three-stage Clos networks and it is rearrangeably non-blocking which means that all available bandwidth to the end hosts can be fully saturated. This network also contains a 1:1 oversubscription ratio which means that all hosts may potentially communicate with any other hosts at the full bandwidth of their interface [1].

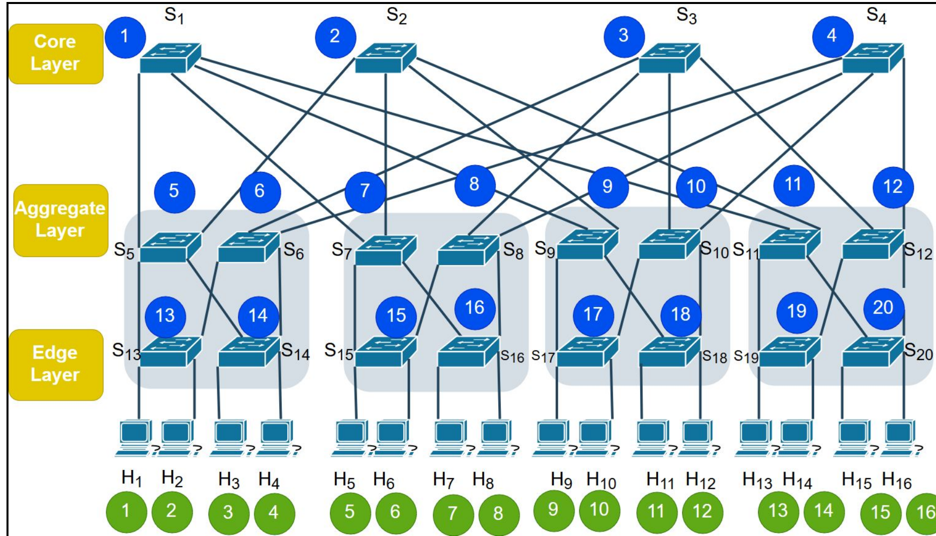

Figure 6. Data Center with Fat Tree Topology

An example of a fat tree network can be seen in Figure 6, which is also called a *k-ar*y fat-tree. The *k* variable stands for the number of ports in each switch and the figure is an example of a *k*=4 fat tree. This topology contains a three-layered hierarchical order of switches. Starting from the top are the edge switches, aggregation switches, and the edge switches. Higher layer switches consume the most energy whereas the bottom layer switches consume the least amount of energy since the top layer switches handle traffic across the network while the bottom layer switches handle traffic form the hosts they are directly connected to. The lower two layers are separated into *k* pods where each pod contains *k/2* aggregation switches and *k/2* edge switches and they form a complete bipartite graph in between [1]. For each edge switch, *k/2* of its ports connect to physical machines (PM), and the other *k/2* ports connect to the higher layer aggregation switch. There are $\frac{k^2}{4}$ k-port core switches and they each connect to k pods. Fat trees contain $\frac{k^3}{4}$ PMs and since the example is a 4-ary fat-tree, it contains 16 PMs [1].

6. METHODOLOGY

6.1. Testbed Settings

The experiment testbed was implemented using a Linux Ubuntu Server system. The experiments ran on an Ubuntu Server 16 (64-bit) with an Intel i7 3.4 Ghz CPU and 16GB of memory. The hypervisor used is Virtualbox 5.1 and it is used to instantiate three virtual machines. Each VM is allocated 1 CPU and 2GB of memory. Each VM contains Mininet and Ryu and each ran and tested one of the algorithms. Python 2.7 is used to write the main logic in Ryu as well as the topology used in Mininet. Communication between the controller and switches occurred on port 6633 and utilized the OpenFlow 1.3 protocol. Each experiment runs for 500 seconds or five minutes.

6.2. The Ryu Framework

Ryu uses scripts and the OpenFlow protocol to communicate and manage the switches. Ryu applications are created using Python scripts. All Ryu applications are subclasses of the *RyuApp* class. Devices from the forwarding plane connect to the Ryu controller by sending messages to the correct port. SDN controllers listen to standard ports 6633 or 6653 and this project uses port 6633. Ryu and the switches will initiate communication between each other by first conducting a handshake in which both agree on a communication protocol to use as well as the corresponding version. Although there are different types of protocols available, this project will use OpenFlow version 1.3 for SDN communication.

The virtual machine image provided by *www.SDNHub.org* was used because it provides a good foundation to implement the testbed. This image was an Ubuntu 14 system with Mininet 2.2, Ryu 3.22, and OpenFlow protocol version 1.3.

The Ryu application is composed of six major components that perform unique functions and are divided by different directories. The *app* directory contains a set of applications that run on the northbound interface of Ryu. The *base* directory contains the base class, *RyuApp*, which is always inherited for all new Ryu applications. The *controller* directory contains files to handle OpenFlow functions. The *lib* directory contains set of packet libraries that are used to parse different protocol headers. The *ofproto* directory contains OpenFlow version specific information so that Ryu can work with different protocol versions. The *topology* directory allows Ryu to discover devices in the network and it uses the LLDP protocol.

6.3. Ryu Applications

Creating Ryu applications is straightforward because only a small amount of code is required but it will not perform any tasks until the logic is implemented. An example of a small and simple Ryu application is shown in Figure 7 but it is useless because it is missing the logic to handle network traffic. Ryu's powerful API can easily be integrated with this code so that it will be able to handle live traffic by either blocking some of the traffic or performing complex packet redirections. Ryu uses handlers and decorators such as *ryu.controller.handler.set_ev_cls* in order to listen and observe events, parse normal packets such as TCP or UDP, and create and send OpenFlow or SDN messages such as PACKET_OUT and FLOW_MOD [27]. Figure 8 shows the decorator that allows Ryu to receive packets sent from a switch to the controller.

```
from ryu.base import app_manager

class L2Forwarding(app_manager.RyuApp):
    def __init__(self, *args, **kwargs):
        super(L2Forwarding, self).__init__(*args, **kwargs)
```

Figure 7. Simple Ryu Application [28]

```
    @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
    def packet_in_handler(self, ev):
```

Figure 8. Ryu Python Decorator Listens to OpenFlow Events [28]

The Python script contains the function *packet_in_handler* that listens to messages from switches. Switches send messages to Ryu via ports. The *set_ev_cls* decorator tells Ryu to call the decorator function whenever a PACKET_IN is detected. A PACKET_IN message occurs when a switch encounters a new flow entry that it does not recognize and this results in a table miss.

6.4. Ryu & Network Traffic

Ryu will use its logic to decide what to do with PACKET_IN messages. Ryu will extract valuable information from packet headers as seen in Figure 9 and 10. Once Ryu determines what is to be done with the PACKET_IN message received from the switch, it builds and sends a PACKET_OUT message. The PACKET_OUT message serves as a quick response as to what a switch should do when encountering a packet or flow of packets that it is not familiar with. The switch first looks at its own flow table to see if it recognizes the incoming flow before deciding to message the controller. Figure 11 shows the OFPPacketOut class that is used to build the PACKET_OUT message. The message will tell the switch what port to send the flow through.

```
    msg = ev.msg               # Object representing a packet_in data structure.
    datapath = msg.datapath    # Switch Datapath ID
    ofproto = datapath.ofproto # OpenFlow Protocol version the entities negotiated. In our
case OF1.3
```

Figure 9. Ryu Extracts Packet Information [28]

```
    pkt = packet.Packet(msg.data)
    eth = pkt.get_protocol(ethernet.ethernet)
```

Figure 10. Ryu Functions Inspect Packet [28]

In order to create a new forwarding rule for a switch, it will build a FLOW_MOD message. This message is sent from Ryu to the switch and it will install a new flow inside the switch's flow table so it will know what to do when similar flows that arrive in the future. This will require a match, and action, and an instruction to be created. Figure 12 shows a match instruction in which a packet must match the *in_port* and *eth_dst* specified.

The match instructions can specify other fields such as the MAC address or the IP address. The actions instructions tell the switch to pass the packet through a specific port or perhaps change its header fields by writing a new destination MAC address. Flow entries may also include a priority number and the higher the number the higher the priority. Note that priority number are only considered when there are wildcard values in the match instruction. For example if a match specifies the source MAC and the destination MAC then that is an exact match. If a match considers any source MAC and a specific destination MAC then it contains a wildcard. When an incoming flow matches multiple flow entries in the flow table and the matches contain wildcard values then the entry with the highest priority will take precedence.

```
    out =
ofp_parser.OFPPacketOut(datapath=dp,in_port=msg.in_port,actions=actions)#Generate the
message
    dp.send_msg(out) #Send the message to the switch
```

Figure 11. Ryu Builds PACKET_OUT Message [28]

```
    msg = ev.msg
    in_port = msg.match['in_port']
    # Get the destination ethernet address
    pkt = packet.Packet(msg.data)
    eth = pkt.get_protocol(ethernet.ethernet)
    dst = eth.dst
    match = parser.OFPMatch(in_port=in_port, eth_dst=dst)
```

Figure 12. Ryu Match Instruction [28]

```
  actions = [ofp_parser.OFPActionOutput(ofp.OFPP_FLOOD)] # Build the required action
```

Figure 13. Ryu Action Instruction [28]

Figure 13 shows an instructions which tells a switch to flood the network. This is useful when the controller does not know how to locate the end host and needs to ask all devices to find out where they are located. An example of this is when traditional switches send ARP requests to identify all the machines or MAC addresses that are connected to its port. All devices connected to the switch receive the ARP request but only the device that is associated with the address in the request will reply using an ARP reply. The switch or controller builds a table matching the MAC address to the port. The action can also be directed towards a specific location. The action will be encapsulated inside the class OFPInstructionActions as seen in Figure 14.

```
inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS, actions)]
```

Figure 14. Ryu Instruction Object [28]

```
mod = parser.OFPFlowMod(datapath=datapath, priority=0, match=match, instructions=inst)
datapath.send_msg(mod)
```

Figure 15. Ryu OFPFlowMod Object [28]

```
from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import MAIN_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.ofproto import ofproto_v1_0

class L2Switch(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_0.OFP_VERSION]

    def __init__(self, *args, **kwargs):
        super(L2Switch, self).__init__(*args, **kwargs)

    @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
    def packet_in_handler(self, ev):
        msg = ev.msg
        dp = msg.datapath
        ofp = dp.ofproto
        ofp_parser = dp.ofproto_parser

        actions = [ofp_parser.OFPActionOutput(ofp.OFPP_FLOOD)]
        out = ofp_parser.OFPPacketOut(
            datapath=dp, buffer_id=msg.buffer_id, in_port=msg.in_port,
            actions=actions)
        dp.send_msg(out)
```

Figure 16. Complete Ryu Application [28]

Once the match, action, and instruction are ready, they can be encapsulated into a FLOW_MOD message using the OFPFlowMod class. Figure 15 shows an OFPFlowMod object which includes the match and instruction to be applied. This will be sent using the Datapath's *send_msg* function. SDN terminology varies slightly from tradition and refers to switches using terms such as *datapaths* but they are also known as *bridges* inside virtual switches [28].

A complete Ryu application is shown in Figure 16. The second argument in the decorator, *MAIN_DISPATCHER*, tells Ryu to only call the function if the handshake is complete. This is useful as it will save time from processing relevant messages. The PACKET_IN data structure is found inside *ev.msg*. The datapath is represented by *dp*. The objects *dp.ofproto* and *dp.ofproto_parser* represent the protocol used between Ryu and the switch. The action set may include one or more action instructions. The class OFPActionOutput is used to specify the switch port that a packet will go out through. Class OFPPacketOut is used to build the PACKET_OUT message that contains the OFPActionOutput object. Datapath classes encapsulate OpenFlow message objects and use the *send_msg* method to instruct switches what to do.

6.5. Ryu Algorithm Implementation

The algorithms for the data center are implemented in a class called ProjectController and it is a subclass of RyuApp. A module called *get_path_module()* contains access to all three algorithms: VM, MB, and VM+MB. As traffic arrives, the module will activate whichever algorithm has been enabled and process the flow.

Global variable *ALGO_CHOICE* is used to enable or disable algorithms in the network. Only one algorithm may run per each session. As traffic is detected by the Ryu controller, it will check it against the master list of VM pairs saved in a variable called *vmpairs*. If the incoming traffic matches a VM pair from the list then it will be assigned to a middlebox. The pairing of a VM pair and a middlebox will depend on the specific algorithm that has been enabled for the session. The path will be calculated using the shortest path algorithm known as Dijkstra’s algorithm. This shortest path will be modified into two parts: 1) path from source VM to

middleboxe and 2) path from middlebox to destination VM. Finally, both paths will be combined and stored in a table. This path will be installed in all switches. Next time traffic from the VM pairs is detected, the switches will reroute the traffic to first traverse the assigned middlebox before it reaches the destination VM.

The class diagram below shows the attributes and functions of the *RyuApp* subclass ProjectController which implements the proposed heuristics. The ProjectController is the only instance since Ryu utilizes the singleton design pattern. Figures 17 and 18 below display the attributes and the functions in the ProjectController class.

ProjectController
+ mac_to_port: Dictionary
+ datapath_list: List
+ SIZE_VM: Integer
+ SIZE_MB: Integer
+ ALGO_CHOICE: Integer
+ switches: List
+ mymac: Dictionary
+ localswitches: List
+ vmpairs: Dictionary
+ load: Dictionary
+ capacity: Integer
+ capacity: Integer
+ costtotal: Integer
+ vm_mb_paths: Dictionary
+ mb_assigned_vmlist: Dictionary
+ eth_to_host: Dictionary
+ host_to_eth: Dictionary
+ GLOBAL_IN_PORT: Integer
+ adjacency: Dictionary

Figure 17. Ryu Class Attributes

+ install_path(self, p, ev, src_mac, dst_mac): None
+ switch_features_handler(self , ev): None
+ get_topology_data(self, ev): None
+ init_reset(): None
+ init_vars(): None
+ run_algo_mb(): None
+ clonevm(): None
+ sort(availablevm, mb): None
+ store_paths_mb(mb_assigned_vmlist): None
+ run_algo_vm_mb_based(): None
+ check_vm_in_set(vm): Boolean
+ get_path(src, dst, first_port, final_port): List
+ minimum_distance(distance, Q): Node
+ get_vm_in_set(vm): VM
+ call_get_path_cost_2(vm,mb): None
+ call_get_path_cost_2_reverse(vm,mb): None
+ get_path_cost_2(src, dst, first_port, final_port, mb): List
+ printheader(): None
+ report_path_results(): None
+ printvm(vmdict): None
+ printpaths(vmmbpaths): None

Figure 18. Ryu Class Functions

In ProjectController the function *_packet_in_handler()* will handle PACKET_IN messages from switches. Ryu will then call module *get_path_module()* which will decide what to do with the incoming flow. Figure 19 shows the module and Figure 20 demonstrates the logic inside the module.

```
    packet_in_cnt=packet_in_cnt+1
    # Pring Packet In #
    print ("\n*")
    print ("[PACKET IN] PACKET_IN (count): %s" %(packet_in_cnt))
    print ("[NEW FLOW]  Switch: %s  Flow: %s(%s) -> %s(%s)   (Not in Flow Table)" %(dpid,eth_
    #              (     src_dp_id,     dst_dp_id,  src_dp_port,  dst_dp_port, src_eth, dst_
    p=get_path_module(mymac[src][0],mymac[dst][0],mymac[src][1],mymac[dst][1],src,dst)
    self.install_path(p, ev, src, dst)
    out_port = p[0][2]
    #print "out_port->",out_por
    #print "ofproto.OFPP_IN_PORT: ",ofproto.OFPP_IN_PORT
else:
    out_port = ofproto.OFPP_FLOOD
```

Figure 19. Ryu Algorithm Module

```
def get_path_module(src,dst,first_port,final_port,srchost,dsthost):
    # Local Vars
    global eth_to_host
    global host_to_eth
    global ALGO_EXECUTED
    global vm_mb_paths
    global ALGO_CHOICE

    # Extract hosts #
    vm=(eth_to_host[srchost],eth_to_host[dsthost])

    # START #
    print("\n\t** Start: M O D U L E **")

    #date#
    ts=time.time()
    st=datetime.datetime.fromtimestamp(ts).strftime('%Y-%m-%d %H:%M
    print ("[DATE] %s" %(st) )

    # Check if algorithm has been executed. #
    if ALGO_EXECUTED is not True:
        print "[CHECK] Next-> START ALGORITHM..."
        if ALGO_CHOICE==1:
            run_algo_vm()          # VM ALGORITHM    #
        elif ALGO_CHOICE==2:
            run_algo_mb()          # MB ALGORITHM    #
        elif ALGO_CHOICE==3:
            run_algo_vm_mb_based() # VM+MB ALGORITHM #
        elif ALGO_CHOICE==4:
```

Figure 20. Ryu Module Handles Flows

Module *get_path_module* will first check if the algorithm has already calculated the path, if not then it will run on of the heuristic algorithms. The heuristic is determined by a global variable called ALGO_CHOICE that stores an integer value pertaining to a specific heuristic function. The boolean variable ALGO_EXECUTED informs Ryu if an algorithm has already been calculated or not. The default value is *False* but is set to *True* as soon as the heuristic finishes. Middleboxes are stored in the *localswitches* variable as a list as shown in Figure 21. VM pairs are stored in a Python Dictionary called *vmpairs* as two-tuple elements as shown in Figure 22. Figure 23 shows the variable *load* which contains the capacity of each switch.

```
########################################
# MB List Initialize                   #
for i in range(SIZE_MB):
    localswitches2.append(mblist[i])
#Endfor                                #

# VM Pairs List  Initialize            #
for i in range(SIZE_VM):
    vmpairs3[ vmlist[i] ]=None
```

Figure 21. Middlebox Initialized

```
# Python File

# List of VMPairs
# Total 240

vmlist=[]
vmlist.append(('h3', 'h1'))
vmlist.append(('h2', 'h16'))
vmlist.append(('h2', 'h8'))
vmlist.append(('h1', 'h7'))
vmlist.append(('h3', 'h13'))
vmlist.append(('h4', 'h14'))
vmlist.append(('h5', 'h15'))
vmlist.append(('h6', 'h16'))
vmlist.append(('h5', 'h8'))
vmlist.append(('h6', 'h9'))
```

Figure 22. VM Pairs Initialized

The VM Based algorithm calculates all paths and stores them in a hash table called *vm_mb_paths* so that they can be retrieved later. VM pairs are represented as two-tuple string values and middleboxes are represented using positive integers. The two-tuple string includes the source and destination host name. A VM pair is used as a key and the value is a list of switches that it is assigned to traverse. The function *call_get_path_cost_2* is called to calculate the cost whenever a VM pair is assigned to a middlebox. The VM pair and middlebox values are passed as parameters to the function.

```
# Reset MB Load to 0 #
for mb in localswitches2:
    load[mb]=0
```

Figure 23. Ryu Load Variable

```
def run_algo_vm():
    # Run Reset()
    init_reset()

    # Globals
    global capacity
    global load
    global vmpairs3
    global localswitches2
    global costtotal
    global vm_mb_paths
    global algo_paths_calculated
    global ALGO_EXECUTED

    print "~~~~~~~~~~~~~~~~~~~~~~~~~~~
    print "~*  V M  B A S E D   ALGOR
    print "~~~~~~~~~~~~~~~~~~~~~~~~~~~
    # Print vmpairs,mb set, capacity #
    printheader()

    for vm in vmpairs3: # Tuples
        costmin= float('Inf')
        cost=0
        path=[]
        vm_i=None
        mb_j=None
        for mb in localswitches2: # In
```

Figure 24. VM Based Algorithm Attributes

```
mb_j=None
for mb in localswitches2: # Integer
    if load[mb]<capacity:
        #
        # cost_path = [ [cost] , [sw & ports] , [sw o
        #
        cost_path = call_get_path_cost_2(vm,mb)
        cost=cost_path[0][0] # Cost
        if(cost<costmin):
            vm_i        = vm # Tuple i.e. ("h1","h7")
            mb_j        = mb # Integer
            costmin     = cost
            path        = cost_path[1] # SW & Ports
            path_sw     = cost_path[2] # Switch only
        #Endif
    #Endif
#Endfor mb
# Update Load #
load[mb_j]=load[mb_j] + 1
# Update Cost #
costtotal=costtotal + costmin
vmpairs3[vm_i]=mb_j # Assignment i->j
vm_mb_paths[vm_i]=path #[Tuple]->Path
#                        #
# Get Reverse Flow->
#   Needed for reply
#   Store flow in table
#   Does not affect the cost.
call_get_path_cost_2_reverse(vm_i,mb_j)
# Update                            #
#print "[NEW ASSIGNMENT] "
#print "VM Pair-> %s Assigned MB-> %s"%(vm_i,mb_j)
#print "Flow (SW Only)     : %s" %(path_sw)
```

Figure 25. VM Based Algorithm Logic

The MB Based algorithm is similar to VM algorithm but it uses a function called *sort* to sort the unassigned list of VM pairs from lowest cost to greatest cost in relation to the specified middlebox. The variable *x_sort* contains the sorted VM pairs list and the function *x_sort.reverse()* reverses the order so that they are ordered in decreasing minimum cost. This places the minimum cost assignment at the top of the stack. The list can be placed inside a for loop and then by applying *x_sort.pop()* function, the first k elements removed will produce the minimum assignment out of all the elements available inside the stack.

```
"""
    MB-BASED ALGORITHM
"""
def run_algo_mb():
    # Run Reset()
    init_reset()

    # Globals #
    global localswitches2
    global mb_assigned_vmlist
    global ALGO_EXECUTED
    global capacity

    # Start #
    print "~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~"
    print "~*  M B  B A S E D   ALGORITHM*~"
    print "~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~"

    # Print #
    printheader()

    availablevm=clonevm()
    for mb in localswitches2: # Iterate Middleboxes
        x_sort=[]
        x_sort=sort(availablevm,mb) # Sort VM Pairs
```

Figure 26. MB Based Algorithm

```
availablevm=clonevm()
for mb in localswitches2: # Iterate Middleboxes
    x_sort=[]
    x_sort=sort(availablevm,mb) # Sort VM Pairs List in
    # Add assgined vms to mb #
    x_sort.reverse() # Reverse list, move small to end,
    # Reset #
    tempsort=[]
    #print "[UPDATE] MB:",mb," Assigning the following V
    for i in range(capacity):
        #check if emtpy
        if len(x_sort)==0: # Empty list
            break
        #Endif
        # Pop next least min cost vmpair #
        element=x_sort.pop()
        # Add vmpair to MB's set #
        tempsort.append(element)
        #print "\t ",element
        # Assign vm pair to current mb #
        vmpairs3[ get_vm_in_set(element) ]=mb
    #Endfor
    mb_assigned_vmlist[mb]=tempsort
    availablevm=x_sort # Store remaining vm-pairs #
#Endfor
```

Figure 27. MB-Based Algorithm Loop

The VM+MB Based algorithm combines techniques from the VM and the MB algorithms. Two for loops are nested inside a while loop until the minimum cost assignment has been achieved from all VM pairs and middleboxes in the sets. In the first round, a VM pair will be assigned to a middlebox that produces the minimum cost. Then, the next VM pair in the list will be assigned a middlebox that produces a minimum cost but only if it is less than all other assignments will the VM pair and middlebox assignment be finalized. Once elements in the *vmlist* have been assigned to a middlebox they are removed from the list. In the next round, the process is similar but now less VM pairs remain in the set. As the VM pairs are assigned they are removed and the list shrinks one by one until it is empty.

```
"""
    VM+MB BASED ALGORITHM
"""
def run_algo_vm_mb_based():
    # Run Reset()
    init_reset()

    # Global #
    global vmpairs3
    global localswitches2
    global load
    global capacity
    global vm_mb_paths
    global costtotal
    global ALGO_EXECUTED

    # Start #
    print "~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~"
    print "~*  V M + M B  B A S E D   ALGORITHM*~"
    print "~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~"
    # Print vmpairs,mb set, capacity #
    printheader()
```

Figure 28. VM+MB Based Algorithm

```
vmlist = clonevm() # Clone list of vm pairs
while len(vmlist)>0:
    costmin=float('Inf') # Set to INF #
    vm_i=None
    mb_j=None
    path_sw_ports=[]
    for vm in vmlist:
        for mb in localswitches2:
            if load[mb]<capacity:                         # Capacity available
                cost_path =call_get_path_cost_2(vm,mb) # 3-Element Array
                cost      =cost_path[0][0]
                if cost<costmin:                      # Check if minimum cost #
                    costmin        =cost             # Update Cost          #
                    path_sw_ports  =cost_path[1] # Path SW, Ports          #
                    path_sw        =cost_path[2] # Path SW Only            #
                    vm_i           =vm
                    mb_j           =mb
                    #print
                #Endif
            #Endif
        #Endfor
    #Endfor
    vmpairs3[get_vm_in_set(vm_i)] = mb_j             # Assigne vm->mb
    vm_mb_paths[vm_i]             = path_sw_ports    # store path for vm->m
    costtotal                     = costtotal+costmin # Total Cost
    load[mb_j]                    = load[mb_j]+1     # Increment Load
    call_get_path_cost_2_reverse(vm_i,mb_j)          # Reverse() for vm res
    vmlist.remove(vm_i)                              # pop()
#Endwhile
# "[RESULTS]"
```

Figure 29. VM+MB Based Algorithm Nested Loops

```
"""
    Calculates path between vm pairs but does not consider
    any middleboxes. A shortest path algorithm is used to
    fin the path.
"""
def get_path (src,dst,first_port,final_port): #Regular, No middleb
                         #label::VM-Based
                         # similar to MB-Based?
    #localswitches = [6,18] #17] #Added by Darshit src to 6, 6 to
    to dest  ---start
    #vmpairs=[101,102,201,202,301,302] # No vm in set so Algo runs
    originalsrc = src
    originaldst = dst
    localpath = []
    for i in range(1): #For MB. Any VM-Pair with source h1,h2,h7,h
    mb0 or j=0
        #print "Shortest Path with No Middleboxes"
        if i==0:
            src=originalsrc
            dst=originaldst
        #Endif
        #Dijkstra's algorithm
        #print ("Called get_path(). src sw: %s, port: %s-> dst sw:
                     #src,first_port,dst,final_port))
        distance = {}
        previous = {}
        for dpid in switches:
            distance[dpid] = float('Inf')
```

Figure 30. *get_path* Module

The function *get_path* will calculate the path for normal flows that do not consider any middleboxes. Once Ryu calculates the paths using one of the algorithms, it will check the new flows against its VM pairs list. If it is in the list, then it will lookup the flow in the flow table. If the flow is not in the VM list, then it will find the shortest path without traversing a middlebox.

6.6. Ryu Flow Chart & Structure

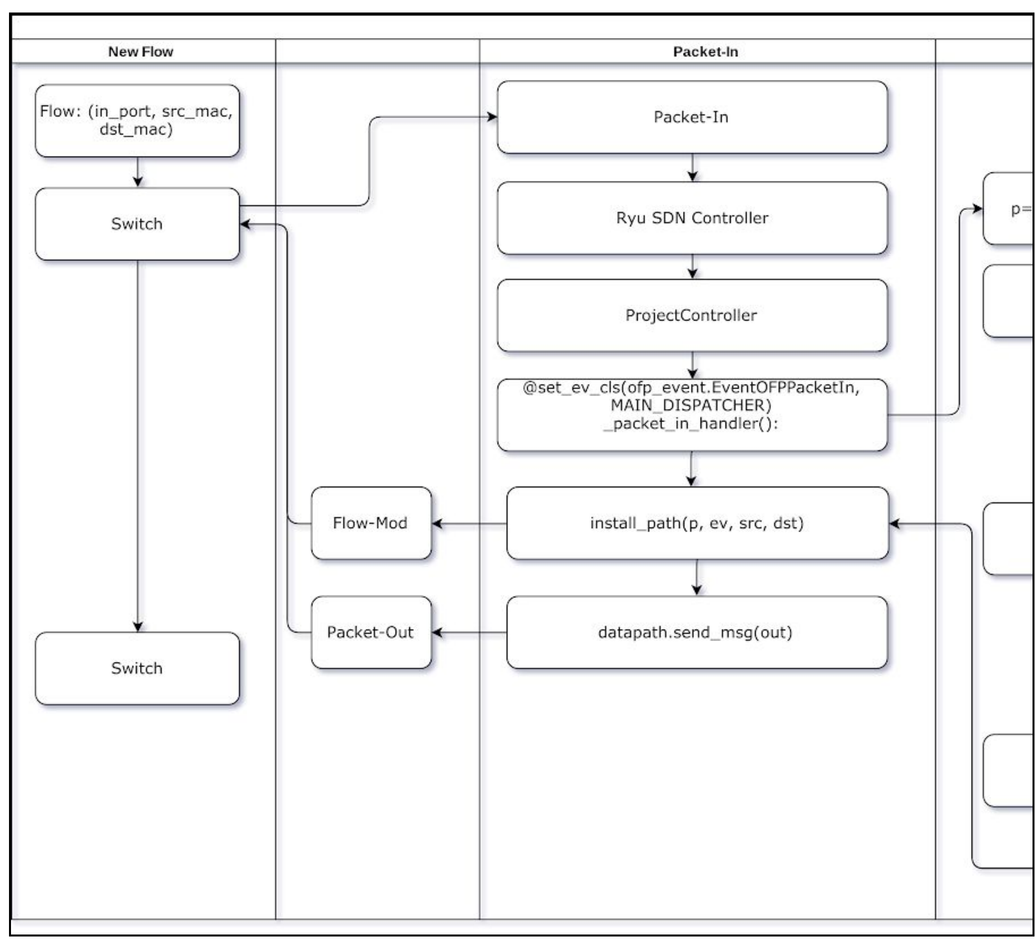

Figure 31. Flow Chart Packet-In

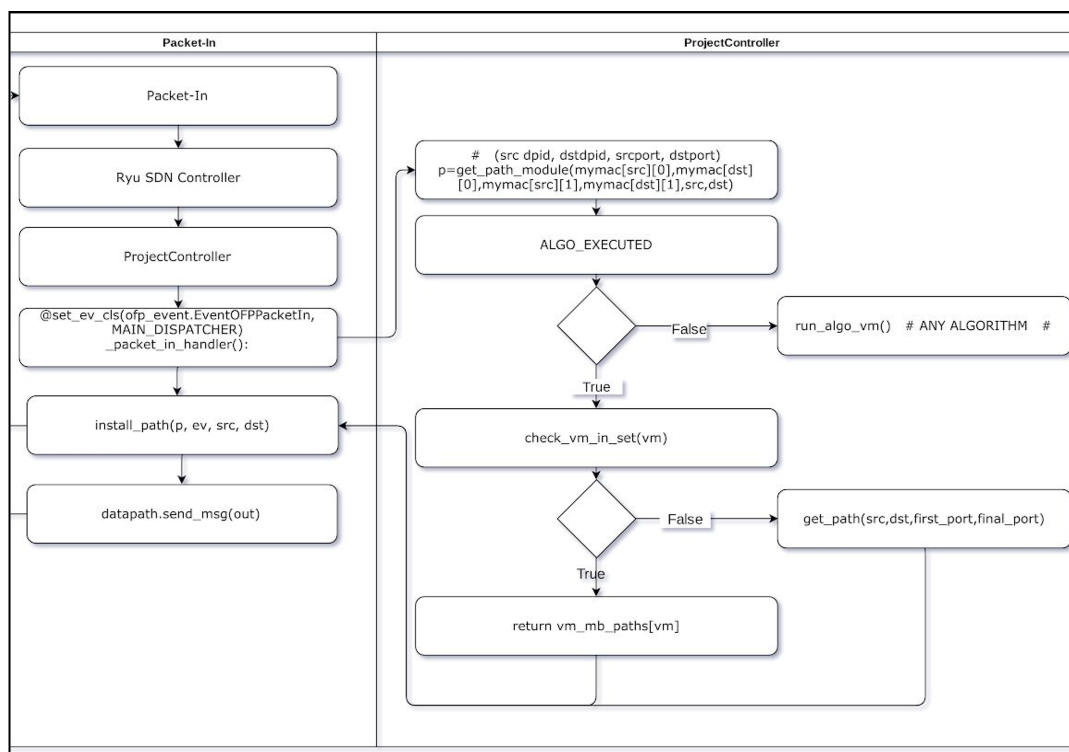

Figure 32. Flow Chart Controller

6.7. Configure Ryu & Mininet Experiment

Ryu and Mininet can only be configured at the start of each session. Developers can adjust the total number of VMs, total number of MBs, and the algorithm that will run in the experiment. Mininet loads a script from a directory to instantiate the data center with a fat tree topology. The Mininet CLI option *mac* will automatically assign mac addresses that match host names as shown in Figure 33. This will increase stability and consistency during the simulation and simplify the logic in Ryu. The Mininet CLI *arp* options allows hosts to cache mac addresses so they know where other hosts are located. The CLI option *ovsk* is used to specified the type of switch. Mininet will utilize the open source virtual switch *Open vSwitch*. All network applicaince

and the controller are configured to use OpenFlow version 1.3 to maintain compatibility. The CLI option *remote* lets switches know where they can find the controller. The controller can be running in the same machine as Mininet as a separate process or it can be running in another machine in the cloud. The SDN controller, Ryu, is a separate application in this experiment and switches connect to localhost IP with port 6633 to send messages to it.

```
ubuntu@sdnhubvm: ~

Last login: Sun Apr  7 19:23:17 2019 from 192.168.56.1
ubuntu@sdnhubvm:~$ sudo mn --custom ~/ryu/ryu/app/fattree4.py
ntroller remote --switch ovsk,stp=1,protocols=OpenFlow13 --mac
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16
*** Adding switches:
s1 s2 s3 s4 s5 s6 s7 s8 s9 s10 s11 s12 s13 s14 s15 s16 s17 s18
*** Adding links:
(s1, s5) (s1, s7) (s1, s9) (s1, s11) (s2, s5) (s2, s7) (s2, s9
6) (s3, s8) (s3, s10) (s3, s12) (s4, s6) (s4, s8) (s4, s10) (s
(s5, s14) (s6, s13) (s6, s14) (s7, s15) (s7, s16) (s8, s15) (s
(s9, s18) (s10, s17) (s10, s18) (s11, s19) (s11, s20) (s12, s1
, h1) (s13, h2) (s14, h3) (s14, h4) (s15, h5) (s15, h6) (s16,
, h9) (s17, h10) (s18, h11) (s18, h12) (s19, h13) (s19, h14) (
6)
*** Configuring hosts
h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16
*** Starting controller
c0
*** Starting 20 switches
s1 s2 s3 s4 s5 s6 s7 s8 s9 s10 s11 s12 s13 s14 s15 s16 s17 s18
```

Figure 33. Mininet Loading Network

```
switches= [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18
]
::Added (count:1):: Host:00:00:00:00:00:09 detected by switch:17 in port
::Added (count:2):: Host:00:00:00:00:00:0a detected by switch:17 in port
::Added (count:3):: Host:00:00:00:00:00:0c detected by switch:18 in port
::Added (count:4):: Host:00:00:00:00:00:0e detected by switch:19 in port
::Added (count:5):: Host:00:00:00:00:00:08 detected by switch:16 in port
```

Figure 34. Ryu Detects Network

6.8. Start Ryu & Mininet

Start by loading Ryu app first and then start Mininet. Give Ryu about one minute to detect the network before interacting with Mininet as seen on Figure 34. Once loaded use

Mininet to start pinging other hosts. Mininet runs pingall so that all switches configure their flow table correctly. The pingall utility may take about 30 minutes or more to complete. Sometimes it may take up to three or more tries to complete successfully. The Ryu app will detect incoming flows. Configure the application to redirect output to a log file and record results. The log can store the paths, list of VM pairs, middleboxes, and total cost.

Bash scripts are used to automate many of the steps and the configurations in the experiment to make the process more efficient. Scripts like *run-global.sh* are used to store global variables such as total amount of VMs, middleboxes, iterations, time, bandwidth, and directories. Scripts are also used to start Iperf for each host in the VM pairs list. Some hosts will run Iperf in server mode and other hosts will run Iperf in client mode. Switches communicate with the Ryu SDN controller using port number 6633 and each switch will be assigned its own unique port number so that the controller can send reply messages back.

Ryu will run each algorithm alone and calculate the costs for all VM pairs. Start Ryu by typing the below command inside the CLI.

```
$~/ryu/bin/ryu-manager ~/ryu/ryu/app/dijkstra_ryu_vm_mb_vmmb.py --observe-links >>
~/ryu/ryu/app/resultslogs/log1.log
```

To start up mininet type the command in a terminal.

```
$sudo mn --custom ~/ryu/ryu/app/fattree4.py --topo mytopo --controller remote --switch
ovsk,stp=1,protocols=OpenFlow13 --arp --mac
```

```
ubuntu@sdnhubvm: ~
h12 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h13 h14 h15 X
h13 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h14 h15 X
h14 -> h1 X h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h15 X
h15 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 X
h16 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 X X h15
*** Results: 7% dropped (222/240 received)
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16
h2 -> h1 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 X h15 h16
h3 -> h1 h2 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16
h4 -> h1 h2 h3 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16
h5 -> h1 h2 h3 h4 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16
h6 -> h1 h2 h3 h4 h5 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16
h7 -> h1 h2 h3 h4 h5 h6 h8 h9 h10 h11 h12 h13 h14 h15 h16
h8 -> h1 h2 h3 h4 h5 h6 h7 h9 h10 h11 h12 h13 h14 h15 h16
h9 -> h1 h2 h3 h4 h5 h6 h7 h8 h10 h11 h12 h13 h14 h15 h16
h10 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h11 h12 h13 h14 h15 h16
h11 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h12 h13 h14 h15 h16
h12 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h13 h14 h15 h16
h13 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h14 h15 h16
h14 -> h1 X h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h15 h16
h15 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h16
h16 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15
*** Results: 0% dropped (238/240 received)
mininet>
```

Figure 35. Mininet Pingall

6.9. Running Algorithms in Ryu

The Ryu algorithms will run in the beginning of the experiment and store results. After the algorithm finishes and stores the calculated paths the boolean value *ALGO_EXECUTED* will be set to *True* and the algorithm will no longer run. In the figure a flow `(h3,h1)` is detected. The module will first check if the flow belongs to a VM pair in the set, and if it does then it will look up the flow table and send a FLOW_MOD message to the switches. Whenever a match is made, the lookup will take place and then display the path found.

```
       ** Start: M O D U L E **
[FLOW]Traffic FROM: h3(00:00:00:00:00:03)   TO: h1(00:00:00:00:00:01)
CHECK IF ALGO_EXECUTED-> TRUE   Next-> CHECK FLOW-TABLES...
*check_vm_in_set(vm)
Checking VM: ('h3', 'h1')
Check:Found in Set.->( h3 , h1 ).
Lookup: Path[vm] in Table...
Found : Path vm_mb_paths[vm]->  [(14, 3, 2), (6, 4, 3), (13, 2, 3)]
Return: Done
('[PACKET_IN] COUNT:', 3, ',SWITCH:', 4, ',SWID#(mymac[src][0]):', 14,
WID#(mymac[dst][0]):', 13)
```

Figure 36. Algorithm checks if flow is in VM pairs list

In Figure 37 the network contains 9 VM pairs, 5 *MBs*, and a load capacity of 2. The output shows that Ryu has detected the VM pairs and middleboxes in the network.

```
        ** Start: M O D U L E **
[FLOW]Traffic FROM: h3(00:00:00:00:00:03)   TO: h1(00:00:00:00:00:01)
CHECK IF ALGO_EXECUTED-> FALSE  Next-> START ALGORITHM...
[ VM Based ALGORITHM ]->
VM Pairs Set:
        ('h3', 'h1')    ('h2', 'h16')
        ('h2', 'h8')    ('h1', 'h7')
        ('h3', 'h13')   ('h4', 'h14')
        ('h5', 'h15')   ('h6', 'h16')
        ('h11', 'h9')
Middlebox Set: [6, 9, 18, 11, 13]
CAPACITY: 2
[ UPDATE: NEW ASSIGNMENT ] ->
```

Figure 37. Ryu Handles Traffic

Figures 38-39 display a network that contains 9 *VMs*, 5 *MBs*, and a capacity of two. In the MB-Based algorithm has finished calculating all the paths and has produced a total minimum cost of 53 as seen on the figure.

```
[ MB Based ALGORITHM ]
VM Pairs Set:
        ('h3', 'h1')    ('h2', 'h16')
        ('h2', 'h8')    ('h1', 'h7')
        ('h3', 'h13')   ('h4', 'h14')
        ('h5', 'h15')   ('h6', 'h16')
        ('h11', 'h9')
Middlebox Set: [6, 9, 18, 11, 13]
CAPACITY: 2
START: Sorting VM Pairs List in relation to-> MB: 6
        [UPDATE] Sort VM->('h3', 'h1')  Cost->3
        [UPDATE] Sort VM->('h2', 'h16')  Cost->5
        [UPDATE] Sort VM->('h2', 'h8')  Cost->5
        [UPDATE] Sort VM->('h1', 'h7')  Cost->5
        [UPDATE] Sort VM->('h3', 'h13')  Cost->5
        [UPDATE] Sort VM->('h4', 'h14')  Cost->5
        [UPDATE] Sort VM->('h5', 'h15')  Cost->7
```

Figure 38. MB Based Algorithm Initiated

```
[ UPDATE: NEW ASSIGNMENT ] ->
VM: ('h3', 'h1') assigned MB: 6
Flow (SW,Ports)    : [(14, 3, 2), (6, 4, 3)
Flow (SW Only)     : [14, 6, 13]
Cost               : 3
[ UPDATE: NEW ASSIGNMENT ] ->
VM: ('h2', 'h16') assigned MB: 6
Flow (SW,Ports)    : [(13, 4, 2), (6, 3, 1)
Flow (SW Only)     : [13, 6, 3, 12, 20]
Cost               : 5
[UPDATE] mb_assigned_vmlist->  {9: [('h11'
)], 11: [('h5', 'h15'), ('h4', 'h14')], 13
costtotal->                    53
SET: ALGO_EXECUTED= True
[ UPDATE: ALGORITHM FINISHED]
        ****
```

Figure 39. MB Based Algorithm Finished

In Figure 40 the VM+MB Based algorithm is running and the network contains 9 VM pairs, 5 *MBs*, and a load capacity of 2. In Figure 41 it has finished and displays the paths for the VM pairs and middlebox pairings. The total cost was 47 which is less than MB Based (53), but more than VM Based (45).

```
[ VM+MB BASED ALGORITHM ]
VM Pairs Set:
        ('h3', 'h1')    ('h2', 'h16')
        ('h2', 'h8')    ('h1', 'h7')
        ('h3', 'h13')   ('h4', 'h14')
        ('h5', 'h15')   ('h6', 'h16')
        ('h11', 'h9')
Middlebox Set: [6, 9, 18, 11, 13]
CAPACITY: 2
[UPDATE] vm: ('h3', 'h1')  assigned-> 6  sw-path-> [
[UPDATE] vm: ('h2', 'h16')  assigned-> 6  sw-path->
[UPDATE] vm: ('h11', 'h9')  assigned-> 9  sw-path->
[UPDATE] vm: ('h2', 'h16')  assigned-> 6  sw-path->
[UPDATE] vm: ('h2', 'h8')  assigned-> 9  sw-path-> [
[UPDATE] vm: ('h2', 'h8')  assigned-> 13  sw-path->
[UPDATE] vm: ('h1', 'h7')  assigned-> 9  sw-path-> [
```

Figure 40. VM+MB Based Algorithm Initiated

```
ubuntu@sdnhubvm: ~/scripts-code
[UPDATE: RESULTS]
TOTAL COST costtotal->                 47
Printout VM Pairs vmpairs3->
        ('h3', 'h1')     ('h2', 'h16')
        ('h2', 'h8')     ('h1', 'h7')
        ('h3', 'h13')    ('h4', 'h14')
        ('h5', 'h15')    ('h6', 'h16')
        ('h11', 'h9')
Assigned paths:vm_mb_paths->
('h3', 'h13')-> [(14, 3, 1), (5, 4, 1), (1, 1, 4), (11, 1, 3), (19, 1, 3)]
('h2', 'h16')-> [(13, 4, 2), (6, 3, 1), (3, 1, 4), (12, 1, 4), (20, 2, 4)]
('h13', 'h3')-> [(19, 3, 1), (11, 3, 1), (1, 4, 1), (5, 1, 4), (14, 1, 3)]
('h5', 'h15')-> [(15, 3, 1), (7, 3, 1), (1, 2, 3), (9, 1, 4294967288L), (1,
, (20, 1, 3)]
('h2', 'h8')-> [(13, 4, 1), (5, 3, 1), (1, 1, 2), (7, 1, 4), (16, 1, 4)]
('h16', 'h6')-> [(20, 4, 1), (11, 4, 1), (1, 4, 3), (9, 1, 4), (18, 1, 4294
), (1, 3, 2), (7, 1, 3), (15, 1, 4)]
('h8', 'h2')-> [(16, 4, 1), (7, 4, 1), (1, 2, 1), (5, 1, 3), (13, 1, 4)]
('h1', 'h3')-> [(13, 3, 2), (6, 3, 4), (14, 2, 3)]
('h11', 'h9')-> [(18, 3, 1), (9, 4, 3), (17, 1, 3)]
('h4', 'h14')-> [(14, 4, 1), (5, 4, 1), (1, 1, 4), (11, 1, 3), (19, 1, 4)]
```

Figure 41. VM+MB Based Algorithm Finished

6.10. Issues with Ryu

If Ryu needs to be restarted then the following steps are helpful: 1) quit Ryu by pressing CTRL+C twice, 2) Exit Mininet and wait until it is done, 3) on the same Ryu terminal clear cache by typing *#mn -c* (root mode), 4) run Ryu again and wait until fully loaded, 5) start Mininet with fat tree script and wait until fully loaded, 6) ping again, and 7) if it fails then reboot the system.

6.11. Traffic Measurement Using Iperf

Iperf is a tool used by engineers in the industry to test network resources, connectivity, and many other things such as troubleshooting slow speeds. It is possible to generate both UDP and TCP traffic with iperf [24]. The goal is to ensure that the algorithms work well with the switches and that traffic is passing through the specified switches in the path. UDP traffic will be

generated through Iperf using one virtual machine as the host and another as the client. Simulation will run for approximately 300 seconds or 5 minutes. Iperf generates constant bit rate (CBR) UDP traffic and at the end of each simulation the averages are displayed. The performance metrics used to compare the algorithms are *End-to-end delay* and the *Packet loss ratio* [25]. Traffic will be generated at rates from 10 megabits per second (Mbps) to 100 Mpbs and will change with strides of 10 Mbps [10].

End-to-end delay is the time that it takes for a packet to travel from a source host *h1* to a destination host *h2*. End-to-end delay is also known as one-way-direction (OWD) and this can be measured through the use of synchronized clocks. The source host will place a timestamp on the packet that it sends and the destination host will note the receiving time and then find the difference [18]. Another method to test end-to-end delay is using the *Minimum-Pairs Protocol* and is generally used with three sets of network nodes. A good quality network link will have a packet loss that is less than 1% [25].

Iperf commands initialize hosts to either run as a server or as a client. Each host will configure Iperf to specify the port number, time interval (in seconds), format, bandwidth, time to listen or transmit traffic, and enable the enhanced report option that calculates end-to-end delay. The client and server will both produce reports of total packet loss and end-to-end delay. The output is saved to log files. The Linux *date* command prints a timestamp to measure how long the process takes to complete. To measure the results, Iperf log files are parsed and the packet loss and end-to-end delays are recorded.

6.12. Iperf Server

When Iperf is in server mode it is configured to listen on port 5566 for this research experiment. This port can be configured to any valid value but for simplicity servers are assigned this specific port. To start the server run the command below inside the terminal:

#iperf -u -s -p 5566 -i 1 -f m -e

This command instructs the host running the Iperf server to listen to UDP traffic on port 5566 and display traffic statistics inside the CLI in intervals of 1 second. The *e* options tells Iperf to produce a summary of the traffic that includes latency such as end-to-end delay which is vital for measuring the performance of each algorithm in this experiment. The Figure below shows host *h3*, a virtual host inside the Mininet network emulator, running Iperf in server mode. There are many options available when using this utility but not all are required or necessary for this experiment. Option *u* specifies that only UDP traffic will be handled, option *p* specifies the port number, and option *f* specifies the format of the results displayed as either bits (b), bytes (B), kilobits (k), kilobytes (K), megabits (m), megabytes (M), or other desired type. Figure 43 shows a log printout every second. At the end of the printout a summary is displayed which shows the total time of the experiment as well as packet delay and loss information.

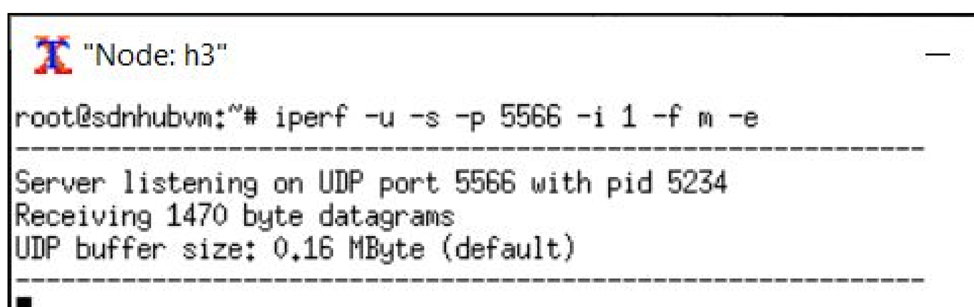

Figure 42. Host Runs Iperf Server

```
[ 78] 281.00-282.00 sec  1.19 MBytes  10.0 Mbits/sec   0.042 ms    0/  851 (0%)  0.082/ 0.016/ 0.677/ 0.094 ms  851 pps  15293.65
[ 78] 282.00-283.00 sec  1.19 MBytes  10.0 Mbits/sec   0.097 ms    0/  850 (0%)  0.076/ 0.016/ 0.542/ 0.089 ms  849 pps  16372.11
[ 78] 283.00-284.00 sec  1.19 MBytes  10.0 Mbits/sec   0.150 ms    0/  850 (0%)  0.092/ 0.016/ 3.318/ 0.148 ms  852 pps  13540.14
[ 78] 284.00-285.00 sec  1.19 MBytes  10.0 Mbits/sec   0.116 ms    0/  850 (0%)  0.092/ 0.016/ 0.484/ 0.096 ms  850 pps  13555.52
[ 78] 285.00-286.00 sec  1.19 MBytes  10.0 Mbits/sec   0.102 ms    0/  851 (0%)  0.092/ 0.016/ 3.612/ 0.154 ms  850 pps  13612.97
[ 78] 286.00-287.00 sec  1.19 MBytes  9.98 Mbits/sec   0.029 ms    0/  849 (0%)  0.086/ 0.016/ 0.623/ 0.096 ms  850 pps  14528.69
[ 78] 287.00-288.00 sec  1.19 MBytes  10.0 Mbits/sec   0.085 ms    0/  851 (0%)  0.079/ 0.016/ 0.560/ 0.089 ms  850 pps  15767.75
[ 78] 288.00-289.00 sec  1.19 MBytes  10.0 Mbits/sec   0.063 ms    0/  851 (0%)  0.081/ 0.016/ 0.426/ 0.089 ms  851 pps  15398.95
[ 78] 289.00-290.00 sec  1.19 MBytes  9.98 Mbits/sec   0.127 ms    0/  849 (0%)  0.090/ 0.016/ 2.468/ 0.131 ms  850 pps  13924.40
[ 78] 290.00-291.00 sec  1.19 MBytes  10.0 Mbits/sec   0.060 ms    0/  851 (0%)  0.080/ 0.016/ 0.427/ 0.088 ms  851 pps  15585.16
[ 78] 291.00-292.00 sec  1.19 MBytes  10.0 Mbits/sec   0.046 ms    0/  851 (0%)  0.083/ 0.016/ 0.612/ 0.094 ms  850 pps  15082.39
[ 78] 292.00-293.00 sec  1.19 MBytes  10.0 Mbits/sec   0.088 ms    0/  850 (0%)  0.081/ 0.016/ 0.390/ 0.091 ms  851 pps  15338.37
[ 78] 293.00-294.00 sec  1.19 MBytes  10.0 Mbits/sec   0.074 ms    0/  850 (0%)  0.083/ 0.016/ 0.465/ 0.092 ms  850 pps  15021.00
[ 78] 294.00-295.00 sec  1.19 MBytes  10.0 Mbits/sec   0.048 ms    0/  851 (0%)  0.082/ 0.016/ 0.576/ 0.091 ms  851 pps  15181.98
[ 78] 295.00-296.00 sec  1.19 MBytes  10.0 Mbits/sec   0.079 ms    0/  850 (0%)  0.088/ 0.016/ 0.619/ 0.095 ms  850 pps  14239.79
[ 78] 296.00-297.00 sec  1.19 MBytes  9.98 Mbits/sec   0.078 ms    0/  849 (0%)  0.089/ 0.015/ 0.712/ 0.095 ms  849 pps  14068.24
[ 78] 297.00-298.00 sec  1.19 MBytes  10.0 Mbits/sec   0.061 ms    0/  852 (0%)  0.091/ 0.016/ 0.388/ 0.095 ms  851 pps  13701.23
[ 78] 298.00-299.00 sec  1.19 MBytes  10.0 Mbits/sec   0.093 ms    0/  850 (0%)  0.087/ 0.016/ 3.256/ 0.144 ms  851 pps  14303.27
[ 78] 0.00-300.00 sec   358 MBytes  10.0 Mbits/sec   0.106 ms   0/255103 (0%)  0.085/ 0.014/16.129/ 0.109 ms  850 pps  14706.11
```

Figure 43. Host *h13* Runs Iperf Report

6.13. Iperf Client

In order for a host to generate UDP traffic in client mode type the command below in a terminal:

#iperf -u -c 10.0.0.3 -p 5566 -t 300 -b10m -e

This tells the host to send traffic to host h3 using port 5566 for 300 seconds at a rate of 10 Mpbs and then generate a report that includes the latency. Figure below shows host h13 running iperf in client mode and its report printout.

```
"Node: h13"
root@sdnhubvm:~# iperf -u -c 10.0.0.3 -p 5566 -t 300 -b10m -e
------------------------------------------------------------
Client connecting to 10.0.0.3, UDP port 5566 with pid 5306
Sending 1470 byte datagrams, IPG target: 1176.00 us (kalman adjust)
UDP buffer size:  160 KByte (default)
------------------------------------------------------------
[ 78] local 10.0.0.13 port 55789 connected with 10.0.0.3 port 5566
[ 78] WARNING: did not receive ack of last datagram after 10 tries.
[ ID] Interval        Transfer     Bandwidth      Write/Err  PPS
[ 78] 0.00-300.00 sec   358 MBytes  10.0 Mbits/sec  255103/0      850 pps
[ 78] Sent 255103 datagrams
root@sdnhubvm:~#
```

Figure 44. Host *h13* Runs Iperf Client

One can extract packet loss ratio, average time delays, minimum time delays, maximum time delays and the amount of packets that were sent during the time period from the log file produced by Iperf. Figure 45 shows Iperf in client mode with log output. The settings were bandwidth of 10 Mbps, 300 seconds, port 5566, IP 10.0.0.1, and enhanced report enabled.

```
"Node: h3"
root@sdnhubvm:~# iperf -u -c 10.0.0.1 -p 5566 -t 300 -b 10m -e
------------------------------------------------------------
Client connecting to 10.0.0.1, UDP port 5566 with pid 8781
Sending 1470 byte datagrams, IPG target: 1176.00 us (kalman adjust)
UDP buffer size:  160 KByte (default)
------------------------------------------------------------
[ 78] local 10.0.0.3 port 46750 connected with 10.0.0.1 port 5566
[ 78] WARNING: did not receive ack of last datagram after 10 tries.
[ ID] Interval        Transfer     Bandwidth      Write/Err  PPS
[ 78] 0.00-300.00 sec   358 MBytes  10.0 Mbits/sec  255102/0      850 pps
[ 78] Sent 255102 datagrams
root@sdnhubvm:~# iperf -u -c 10.0.0.1 -p 5566 -t 300 -b 10m -e
------------------------------------------------------------
Client connecting to 10.0.0.1, UDP port 5566 with pid 8869
Sending 1470 byte datagrams, IPG target: 1176.00 us (kalman adjust)
UDP buffer size:  160 KByte (default)
------------------------------------------------------------
[ 78] local 10.0.0.3 port 45305 connected with 10.0.0.1 port 5566
cccb[ ID] Interval        Transfer     Bandwidth      Write/Err  PPS
[ 78] 0.00-300.00 sec   358 MBytes  10.0 Mbits/sec  255101/0      850 pps
[ 78] Sent 255101 datagrams
[ 78] Server Report:
[ 78] 0.00-300.00 sec   358 MBytes  10.0 Mbits/sec   0.067 ms   0/255101 (0%)  0.052/ 0.000/ 8.380/ 0.074 ms  850 pps  23839.51
root@sdnhubvm:~# cccb
```

Figure 45. Iperf Running in Client Mode

In this example the output format is in Interval, Transfer, Bandwidth, Jitter, Lost/Total, Latency avg/min/max/stdev PPS, and NetPwr. If total data loss is high, it may be because flows are beginning to be calculated and switches are sending PACKET_IN messages. To improve packet loss the algorithm should run first so flows will be installed in all switches and then run the traffic generator.

6.14. Parsing Output

Parsing data from Iperf logs was achieved by using regular expression statements in Linux. Essential tools and utilities were: *concatenate (cat)* , *piping, grep,* and *translate (tr)*. To extract the columns and rows that contained Time, Packet Loss, and Average Delay the *gawk* utility was used.

An example of such command is:

```
$cat outputtext.log | grep sec | tr - " " | tr / " " | awk '{ print " "$4"\t"$15"\t"$16}'
```

After parsing the log file, there are three remaining columns which are time in seconds, packet loss, and average end/end delay.

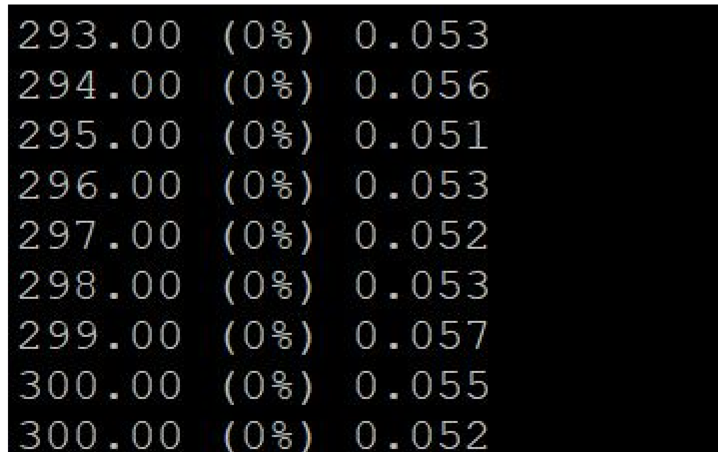

Figure 46. Parsed Iperf log data

6.15. Plotting Results

Results are plotted using a powerful utility known as *Gnuplot*. Start Gnuplot by using the *gnuplot* command. Use *Ghostscript* to view or convert PostScript files to PDF files. To create a simple plot type command below in the terminal.

gnuplot>plot "output-columns.dat" title "End End Delay" with linespoints in

To set the X axis type

gnuplot>set xrange [1:15], and to set X tics use gnuplot>set xtics 1,1,15,

To set the label for X axis type the command below in the terminal.

gnuplot>set xlabel " Time (ms) "

This will be the similar configuration format for the Y axis. Finalizing output by typing

gnuplot>set term postscript

Output the file type

gnuplot>set output "outputpostfile.ps"

Replot the data

gnuplot>replot

Finish plotting

gnuplot>exit

Data and results are stored into three folders labeled vm, mb, and vmmb in reference to the algorithm that they represent. Under each folder there are subfolders that store the number of VM that the experiment uses (i.e. 2,4,6, etc). The main folder will be labeled with total VM's,

total MB’s, and bandwidth. Each subfolder will contain a log of the Iperf printout for each VM pair. The parsed Iperf log data will also be stored.

7. DISCUSSION

7.1. Results

7.1.1. Experiment 1

This experiment contains 10 VM pairs, 3 middleboxes, bandwidth of 100Mbps, and the packet rates of 1.0-10.0 Mbps. The results for the end-to-end delay and packet loss are plotted in the graphs.

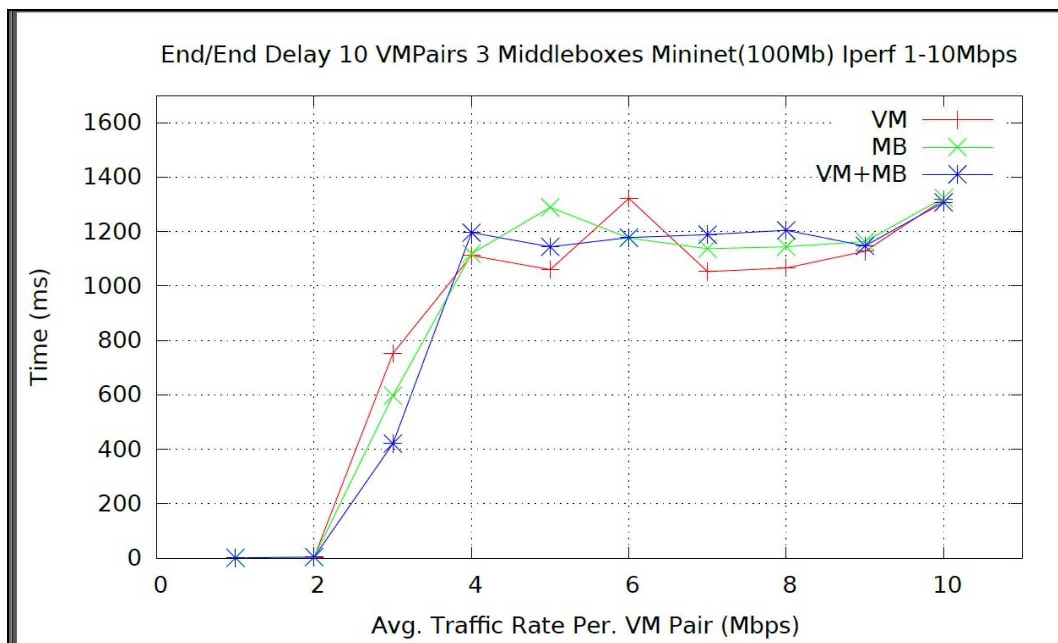

Figure 47. End-to-end Delay for 10 *VMs*, 3 *MBs*, 100Mbps Bandwidth

The three heuristics share a similar trend for the end-to-end delay because time delay increases as the transfer rate increases. VM+MB performs best in about 50% of the time as seen on the Table 2. It produces the smallest time delay in the following rates: 1.0-4.0 Mbps, 5.0-6.0 Mbps, and 9.0-10.0 Mbps. VM performs second best and MB performs last by having the highest end-to-end delay times.

Table 2. Experiment 1 End-to-end Delay 10VM 3MB

Mbps	Smallest Delay	Median Delay	Largest Delay
1-2	Same	--	--
2-3	VMMB	MB	VM
3-4	VMMB	MB	VM
4-5	VM	VMMB	MB
5-6	VMMB	MB	VM
6-7	MB	VMMB	VM
7-8	VM	MB	VMMB
8-9	VM	MB	VMMB
9-10	VMMB	VM	MB

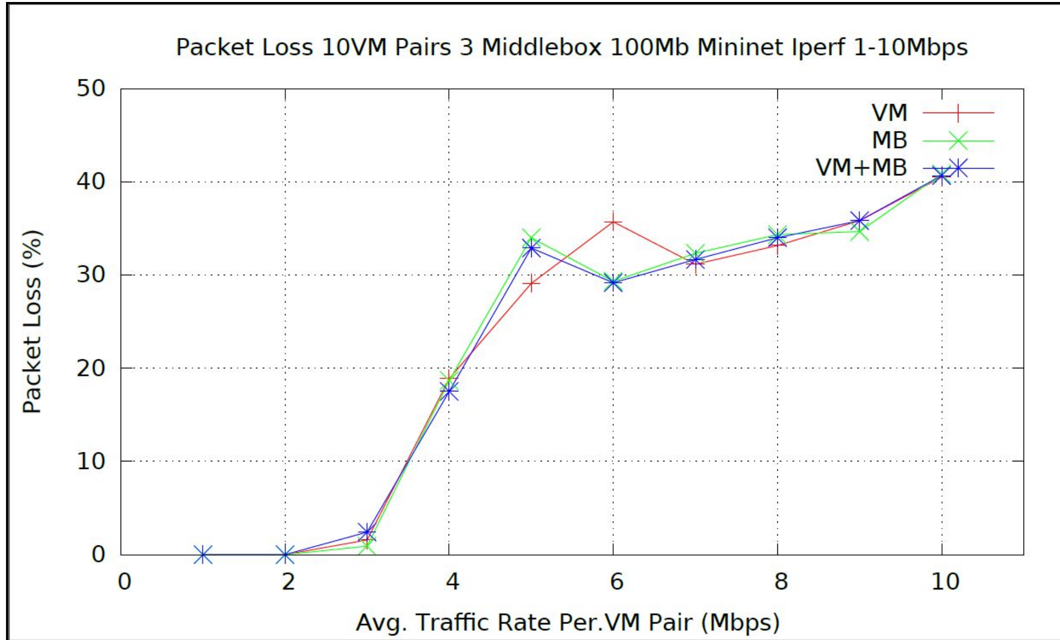

Figure 48. Packet Loss for 10 *VMs*, 3 *MBs*, 100Mbps Bandwidth

The packet loss increase as traffic rate increases for all heuristics. Out of all the heuristics VM+MB and MB perform best as they produced the least packet loss overall and as seen in Table 3. VM produced the largest packet loss overall.

Table 3. Experiment 1 Packet Loss 10VM 3MB

Mbps	Smallest Loss	Median Loss	Largest Loss
1-2	Same	--	--
2-3	Same	--	--
3-4	Same	--	--
4-5	VM		MB/VMMB
5-6	MB/VMMB		VM
6-7	MB/VMMB		VM

7-8	Same	--	--
8-9	Same	--	--
9-10	Same	--	--

7.1.2. Experiment 2

This experiment contains 10 VM pairs, 5 middleboxes, bandwidth of 100Mbps, and the packet rates of 1.0-10.0 Mbps. The results for the end-to-end delay and packet loss are plotted in the graphs.

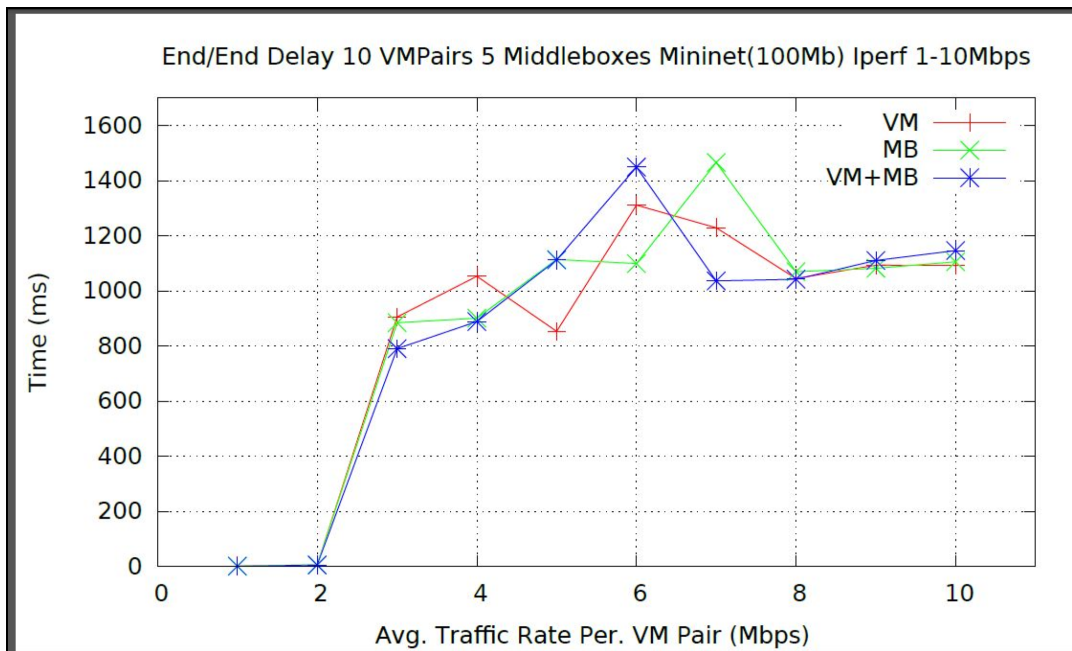

Figure 49. End-to-end Delay for 10 *VMs*, 5 *MBs*, 100Mbps Bandwidth

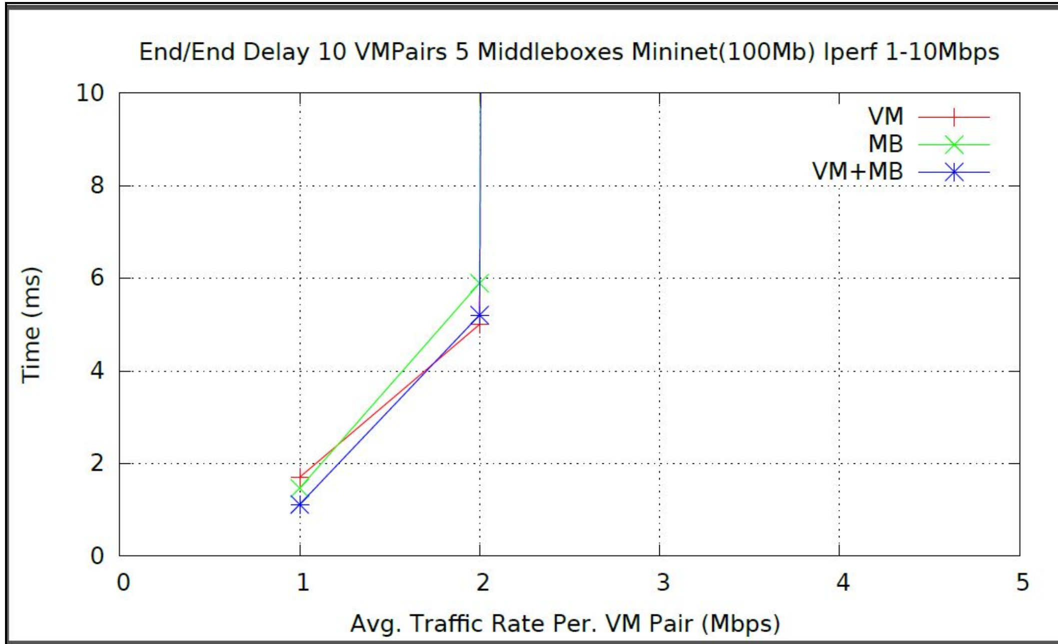

Figure 50. End-to-end Delay Closeup

A close up view can be observed in Figure 50 to better understand the traffic behavior for the 1.0-2.0Mbps interval. These results point out that VM+MB seems to outperform the other algorithms from the beginning of the experiment even as traffic rate is very small.

While End-to-end delay increased for all heuristics when packet rate increased, it was VM+MB that produced the smallest end-to-end delay overall and was the best among all heuristics when it came to 10 VM pairs and 5 MBs. VM+MB performed best during 1.0-4.0Mbps, and 6.0-9.0Mbps as seen in Table 4. VM performed best during 4.0-6.0Mbps interval and MB performed best during 9.0-10.0Mbps interval. Although End-to-end delay peeks at 6.0-7.0Mbps for all heuristics it then drops but this could be due to some random variation and needs more experimentation.

Table 4. Experiment 2 End-to-end Delay 10VM 5MB

Mbps	Smallest Delay	Median Delay	Largest Delay
1-2	VMMB	MB	VM
2-3	VMMB	MB	VM
3-4	VMMB	MB	VM
4-5	VM	VMMB/MB	--
5-6	VM	MB	VMMB
6-7	VMMB	VM	MB
7-8	VMMB	VM	MB
8-9	VMMB	VM	MB
9-10	MB	VM	VMMB

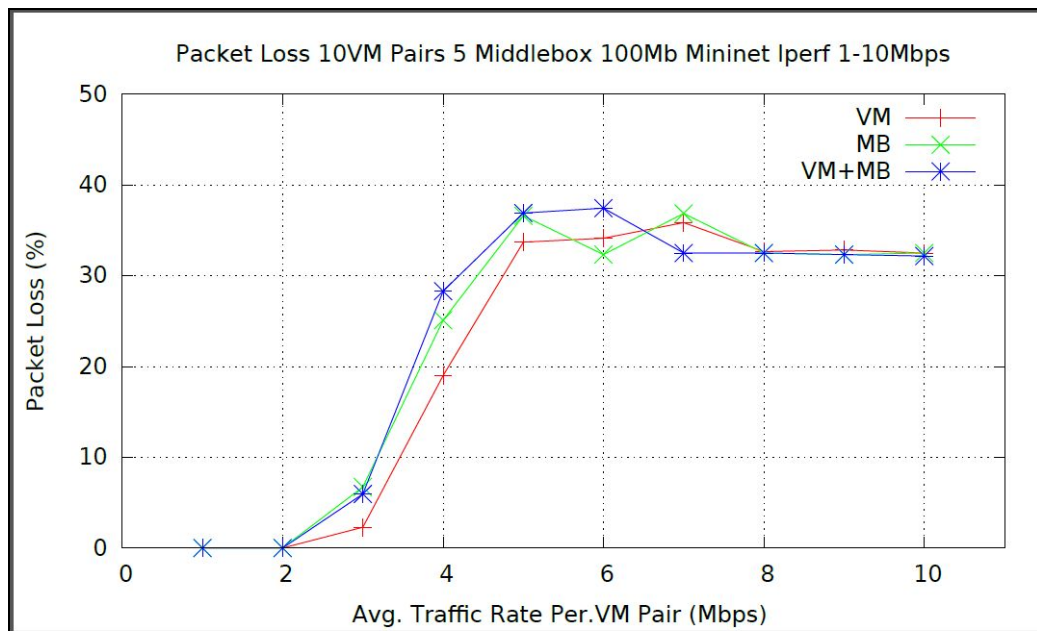

Figure 51. Packet Loss for 10 VMs, 5 MBs, 100Mbps Bandwidth

Packet loss increased in all algorithms as the traffic rate increased. VM and VM+MB performed best overall by having the least packet loss. VM did best in 2.0-5.0Mbps interval and VM+MB did best in 7.0-10.0Mbps interval as seen in Table 5. MB performed the worst in this experiment as it experienced the highest packet loss out of all the heuristic.

Table 5. Experiment 2 Packet Loss 10VM 5MB

Mbps	Smallest Loss	Median Loss	Largest Loss
1-2	Same	--	--
2-3	VM	--	MB/VMMB
3-4	VM	--	MB/VMMB
4-5	VM	--	MB/VMMB
5-6	MB	VM	VMMB
6-7	MB	VM	VMMB
7-8	VMMB	--	VM/MB
8-9	VMMB	--	VM/MB
9-10	VMMB	--	VM/MB

8. CONCLUSION

VM+MB-Based algorithm seems to perform better than the other two algorithms but more testing is needed. Parameters need to be further explored and adjustments must be made

where needed. The amount of middleboxes does affect the outcome of the energy consumption results for every algorithm. When middleboxes are increased, there are clear differences in the final cost for all algorithms. Lower amounts of middleboxes does not seem to have an effect on final cost but there are some variations in end-to-end delay that must be investigated. It is important to note that fixing the bandwidth of Mininet was crucial to producing more consistent and accurate results. Since mininet bandwidth had not been fixed in prior experiments, results had been much more inconsistent. Finding the right Mininet parameters should help improve the results in the testbed that has been implemented.

9. REFERENCES

[1] M. Alqarni, A. Ing, and B. Tang. (2017). LB-MAP: Load-Balanced Middlebox Assignment in Policy-Driven Data Centers. 1-9. 10.1109/ICCCN.2017.8038423.

[2] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar, "Making middleboxes someone else's problem: Network processing as a cloud service," *ACM SIGCOMM Computer Communication Review*, 2012.

[3] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *ACM SIGCOMM*, 2008.

[4] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu, "SIMPLE-fying middlebox policy enforcement using sdn," *SIGCOMM Comput. Commun.* Rev., 2013.

[5] Y. Zhang et al., ''StEERING: A software-defined networking for inline service chaining,'' in *Proc. 21st IEEE ICNP*, Oct. 2013, pp. 1–10.

[6] ETSI NFVISG, "Network functions virtualisation. introductory white paper," 2013.

[7] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: A network programming language," in *ACM SIGPLAN Notices*, vol. 46, no. 9. *ACM*, 2011, pp. 279–291.

[8] D. Erickson, "The Beacon OpenFlow controller," in *Proceedings of the second ACM SIGCOMM* workshop on Hot topics in software defined networking, ser. *HotSDN '13*. New York, NY, USA: ACM, 2013, pp. 13–18.

[9] J. Liu, Y. Li, Y. Zhang, L. Su, D. Jin, "Improve service chaining performance with optimized middlebox placement", *IEEE Trans. Services Comput.*, vol. 10, no. 4, pp. 560-573, Jul./Aug. 2017.

[10] W. Ma, J. Beltran, Z. Pan, D. Pan, and N. Pissinou, “Sdn-based traffic aware placement of nfv middleboxes,” *IEEE TNSM*, vol. 14, no. 3, pp.528–542, 2017.

[11] W. Ma, O. Sandoval, J. Beltran, D. Pan, and N. Pissinou, “Traffic aware placement of interdependent nfv middleboxes,” in *INFOCOM 2017-IEEE Conference on Computer Communications*. IEEE, 2017, pp.1–9.

[12] D. Kreutz et al., "Software-defined networking: A comprehensive survey", *Proc. IEEE*, vol. 103, no. 1, pp. 14-76, Jan. 2015.

[13] R. Kumar, M. Hasan, S. Padhy, K. Evchenko, L. Piramanayagamk, S. Mohan, R. B. Bobba, "End-to-End Network Delay Guarantees for Real-Time Systems using SDN", *IEEE Real-Time Systems Symposium (RTSS)*, 2017.

[14] Y. Ben-Itzhak, K. Barabash, R. Cohen, A. Levin, E. Raichstein, "EnforSDN: Network policies enforcement with SDN", *Proc. IFIP/IEEE Int. Symp. Integr. Netw. Manage.*, pp. 80-88, 2015.

[15] B. Lantz, B. Heller, and N. McKeown, “A network in a laptop: rapid prototyping for software-defined networks,” in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks. ACM*, 2010, p. 19.

[16] Gao, L., Rouskas, G.N.: “Virtual Network Reconfiguration with Load Balancing and Migration Cost Considerations”, *Proc. - IEEE INFOCOM*, 2018, 2018–April, pp. 2303–2311.

[17] Unknown. (1970, January 01). IPERF : Test Network throughput, Delay latency, Jitter, Transefer Speeds , Packet Loss & Raliability. Retrieved from http://linuxthrill.blogspot.com/2016/04/iperf-test-network-throughput-delay.html

[18] *End-to-end delay.* (2019, February 18). Retrieved April 7, 2019, from https://en.wikipedia.org/wiki/End-to-end_delay

[19] *How to Install Ubuntu.* (n.d.). Retrieved May 10, 2018, from https://www.howtoinstall.co/en/ubuntu/trusty/iperf?action=remove

[20] Team, M. (n.d.). *Mininet Overview*. Retrieved from http://mininet.org/overview/

[21] Mininet. (n.d.). *Mininet/mininet*. Retrieved from https://github.com/mininet/mininet/wiki/Introduction-to-Mininet

[22] *About Tics*. (n.d.). Retrieved June 2, 2018, from http://lowrank.net/gnuplot/tics-e.html

[23] *Ryu*. (n.d.). Retrieved January 2, 2018, from https://sourceforge.net/p/ryu/mailman/message/33797329/

[24] *Iperf2*. (n.d.). Retrieved February 7, 2019, from https://sourceforge.net/projects/iperf2/

[25] (n.d.). Retrieved February 7, 2019, from https://fasterdata.es.net/performance-testing/network-troubleshooting-tools/iperf/

[26] *Chapter 27. Bash Arrays*. (n.d.). Retrieved May 2, 2019, from http://tldp.org/LDP/abs/html/arrays.html

[27] *The First Application*. (n.d.). Retrieved January 10, 2019, from https://ryu.readthedocs.io/en/latest/writing_ryu_app.html

[28] Natarajan, S. (n.d.). *RYU Controller Tutorial*. Retrieved January 10, 2019, from http://sdnhub.org/tutorials/ryu

[29] *What is Ryu Controller?* - SDxCentral .com. (n.d.). Retrieved January 10, 2019, from https://www.sdxcentral.com/networking/sdn/definitions/what-is-ryu-controller

[30] P. Khani, B. Tang, J. Han, and M. Beheshti. “Dao-r: Integrating data aggregation and offloading in sensor networks via data replication”. *In Proceedings of IEEE GLOBECOM* 2015.

10. APPENDIX

10.1. Ryu Python Implementation

```
# Copyright (C) 2008 Nippon Telegraph and Telephone Corporation.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#      http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
# implied.
# See the License for the specific language governing permissions and
# limitations under the License.

#
# ** VM-Based Algorithm **
# Input is a data center G(V,E).
# Set all i as assigned=False.//?
# Start iterating with the first i in VM Pairs Set P.
# Iterate through all j in MB Set M.
# Check there is available capacity in j or load(j)<k.
# Calculate Cij.
# Find the minimum cost out of all available j.
# Assign i to j.
# Label assigned[i]=true.//?
# load(j)=load(j) + 1.
# Iterate next i in VM Pair Set M.
#

"""
update april 7 2019

This script will detect flows and apply algorithms to efficently
route them through appropriate middleboxes. The script will first
run the algorithm and use the given set of vm pairs and middleboxes
to calculate the path which yelds the minimum costs. Once the
algorithm finishes, all paths will be stored in a table. When
an incoming flow is detect it will be looked up and verified if
it has been assigned a middlebox. If it has, it will load the
appropiate path, if not, then it will find the shortest path
without including a middlebox.

The flow is installed on all switches that lie in the path. This
is achieved through PACKET_IN and FLOW_MOD messages.
"""
# Libraries #
import math
import sys
import time
```

```
import datetime
from vmpairs_data import vmlist
from collections import OrderedDict
from ryu.base import app_manager
from ryu.controller import mac_to_port
from ryu.controller import ofp_event
from ryu.controller.handler import CONFIG_DISPATCHER, MAIN_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.ofproto import ofproto_v1_3
from ryu.lib.mac import haddr_to_bin
from ryu.lib.packet import packet
from ryu.lib.packet import ethernet
from ryu.lib.packet import ether_types
from ryu.lib import mac
from ryu.topology.api import get_switch, get_link
from ryu.app.wsgi import ControllerBase
from ryu.topology import event, switches
from collections import defaultdict

###############
##  GLOBALS  ##
###############
# Size of VM #
SIZE_VM       =0 # Choose 1:16 #
# Size of MB #
SIZE_MB       =0 # Choose 1:16 #
# ALGO CHOICE         #
# 1: VM-Based         #
# 2: MB-Based         #
# 3: vm+MB Based   #
#                     #
ALGO_CHOICE =0
#switches
switches = []
#mymac[srcmac]->(switch, port)
mymac={}
#adjacency map [sw1][sw2]->port from sw1 to sw2
# Middlebox List      #
# List of middleboxes #
localswitches2       =[]
# VM-Pairs List               #
# old version                 #
#vmpairs2            = {}
# regular dictionary          #
#vmpairs3            ={}
# created an ordered dictionary #
vmpairs3             =OrderedDict()

# Dictionary which contains the current load for all #
# the middleboxes                                  #
load                 ={}

# Capacity for middleboxes  #
# capacity is initialzied as #
# an integer.                #
capacity             =0
```

```
# Set to True when aglorithm finishes #
ALGO_EXECUTED        =False
# Total Cost of network flow #
costtotal            =0
# VM Based Algorithm                #
vm_mb_paths          ={}
# Used for MB-Based Algorithm       #
mb_assigned_vmlist  ={}
# Maps ETH/MAC to Host Name                #
eth_to_host          ={}
# Maps Host Name to ETH/MAC                #
host_to_eth          ={}
# Counst Hosts Discovered during startup #
SW_NOT_DISCOVERED   =True
# Tells Ryu to allow switches to pass packets through #
# same the port. Sets IN_PORT = OUT_PORT           #
GLOBAL_IN_PORT       =4294967288
# PACKET_IN Counter #
packet_in_cnt        =0
# Used to store Edges and their values  #
adjacency=defaultdict(lambda:defaultdict(lambda:None))
# Print path results #
REPORT_PATH_RESULTS_ENABLED = True

###############
##  GLOBALS  ##
###############
"""
       Reset variables
       load
       costtotal
"""
def init_reset(): # Starts whenever algorithm is started
       # Global #
       global load
       global localswitches2
       global costtotal
       # Reset MB Load to 0 #
       for mb in localswitches2:
       load[mb]=0
       # Reset Costtotal to 0 #
       costtotal=0
# Enddef End of init_reset function #

"""
       Initialize variables.
       Only need to run once.
       Set total VM, total MB, load, capacity,
       print reports.
"""
def init_vars():
       print "\t** Initialized Variables **"
       # Globals #
       global vmpairs3
       global load
       global localswitches2
```

```
global host_to_eth
global eth_to_host
global vm_mb_paths
global capacity
global SIZE_VM
global SIZE_MB
global ALGO_CHOICE
global REPORT_PATH_RESULTS_ENABLED
# Enable/Disable REPORT_PATH_RESULTS_ENABLED #
REPORT_PATH_RESULTS_ENABLED=True #True#False

#######################################
# ALGO SETTINGS                      #
#                                    #
SIZE_VM       =200  # Choose 1:16          #
SIZE_MB       =5 # Choose 1:16       #
#                                    #
# _CHOICE_                           #
# 1)VM 2)MB                          #
# 3)VM+MB  4)MCF                     #
# 5)ALL                              #
ALGO_CHOICE =3                       #
#                                    #
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~#

#####################################################
# DATA For Middleboxes and VM Pairs                #
# see outside class/file                           #
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~#
# Middlebox List                                   #
#                                                  #
mblist=[6,9,11, 13,15,17,19,5,7,8,10,12,14,16,18, 20]
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~#
# VM Pairs List  dictionary{}                      #
# Static list of VM Pairs ,16 total                #
#vmlist=[]
#vmlist.append(("h3","h1")) # LB-MAP Graph Examples  #
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~#
# DATA END                                         #
#####################################################

#####################################
# MB List Initialize                #
for i in range(SIZE_MB):
localswitches2.append(mblist[i])
#Endfor                             #
# VM Pairs List  Initialize         #
for i in range(SIZE_VM):
vmpairs3[ vmlist[i] ]=None
#Endfor                             #
#####################################

# Initialize Capacity                                            #
if len(localswitches2) > len(vmpairs3.items()):
capacity=1
# Set capacity using formula ceiling(total vm pairs / total mb). #
else:
```

```
        capacity = math.ceil( len( vmpairs3.items() )*1.0 / len( localswitches2 ) ) #1#3#1 # k,
capacity
        capacity = int(capacity)
        # Initilize load for all MB in list #
        for mb in localswitches2:
        load[mb]=0
        # "\n Initialize Mac/Eth/Host"  #
        #                             #
        # Populate ETHERNET and HOST addresses.  #
        # mymac["00:00:00:00:00:01"][0]=SW_ID
        # mymac[ host_to_eth["h1"] ][0]=SW_ID
        for i in range(1,17):
        if i<16:
                var_host=str("h%d"%(i))
                var_eth =str("00:00:00:00:00:0%s"%(hex(i)[2:]))
        else:
                var_host = str("h%d"%(i))
                var_eth  = str("00:00:00:00:00:%s"%(hex(i)[2:]))
        host_to_eth[var_host] = var_eth
        eth_to_host[var_eth]  = var_host
        # End ETH/MAC                         #
        #                                     #
#Enddef
#End init_vars()

#startbackup
"""
        Checks if vm pair is part of the set.
"""
def check_vm_in_set_BACKUPFUNCTION(vm):
        global vmpairs3
        src=vm[0]
        dst=vm[1]
        #print "*check_vm_in_set(vm)"
        #for vm in vmpairs3:
        #print vm
        #print "Checking VM:",vm
        if (src,dst) in vmpairs3:
        print "[FOUND VM]-> (",src,",",dst,"). "
        return True
        elif (dst,src) in vmpairs3:
        print "[FOUND VM]-> (",dst,",",src,"). "
        return True
        else:
        print "FAIL:VM Pair Not Found->(",src,",",dst,"). "
        return False
#Enddef

"""
        Checks if vm pair is part of the set.
"""
def check_vm_in_set(vm):
        global vmpairs3
        src=vm[0]
        dst=vm[1]
        #for vm in vmpairs3:
        #print vm
```

```
        print ("::New Func Def/New Edit::")
        if (src,dst) in vmpairs3:
        print "(new func def)[FOUND VM]-> (",src,",",dst,"). "
        return True
        #elif (dst,src) in vmpairs3:
        #print "[FOUND VM]-> (",dst,",",src,"). "
        #return True
        else:
        print "(new func def)FAIL:VM Pair Not Found->(",src,",",dst,"). "
        return False
#Enddef
"""
        get if vm pair is part of the set.
"""
def get_vm_in_set(vm):
        global vmpairs3
        src=vm[0]
        dst=vm[1]
        #print "::get_vm_in_set(vm)"
        #for vm in vmpairs3:
        #print vm
        #print "Checking VM:",vm
        if (src,dst) in vmpairs3:
        #rint "Check:Found in Set.->(",src,",",dst,"). "
        return (src,dst)
        elif (dst,src) in vmpairs3:
        #print "Check:Found in Set.->(",dst,",",src,"). "
        return (dst,src)
        else:
        print "FAIL:Not Found->(",src,",",dst,"). "
        return False
#Enddef

"""
# def call_get_path_2(vm,mb):
# This method will prepare vm,mb format and call the
# method in correct format
# call_get_path(vm,mb)
        This method will calculate the costs.
        Should be called by original VM pairs from main list.
"""
def call_get_path_cost_2(vm,mb): # Turns vm 2-tuple into 5-tuple
        global host_to_eth
        global eth_to_host
        path=[]
        src=vm[0]
        dst=vm[1]
        #get_path(src sw id, dst sw id,
        #       src sw port, dst sw port,
        #       mb)
        path=get_path_cost_2(
                mymac[host_to_eth[src]][0],
                mymac[host_to_eth[dst]][0],
                mymac[host_to_eth[src]][1],
                mymac[host_to_eth[dst]][1],
                mb)
```

```
        return path
#End

"""
#
# This method will prepare vm,mb format and call the
# method in correct format
# call_get_path(vm,mb)
        Used this for the returning traffic.
        Designed for ping testing.
        Should not be needed for (UDP traffic)
"""
def call_get_path_cost_2_reverse(vm,mb): # Turns vm 2-tuple into 5-tuple
        global host_to_eth
        global eth_to_host
        global vm_mb_paths
        path=[]
        src=vm[1]
        dst=vm[0]
        #EDIT jul 3 2019
        vm=()
        vm=(src,dst) # Reversed
        #
        ## Installs path in hash table.  #
        ## Includes sw and ports.    #
        path=call_get_path_cost_2(vm,mb)
        vm_mb_paths[vm]=path[1]
# Enddef

"""

        Calculate: Path, Cost
        def get_path_cost_2()
        This function will get_path.
        Input is vm pair set and mb.
        Output is a list of 3 elements.
        Output results->[[cost of path with only switches],
                        [path of only switches],
                        [path of switches and ports]]
"""
def get_path_cost_2(
                src,dst,first_port,final_port,mb):
        # Algorithm #
        originalsrc=src
        originaldst=dst
        localpath=[]
        path=[]
        cost_path=[]
        cost=0
        localswitchesA=mb
        for i in range(2):
        if i==0:
                src=originalsrc
                dst=localswitchesA # mb
        else:
                src=localswitchesA # mb
                dst=originaldst
```

```
#Endif
#print "current src=%s, dst=%s"%(src,dst)
#Dijkstra's Algorithm
distance={}
previous={}
for dpid in switches:
        distance[dpid]=float('Inf')
        previous[dpid]=None
#Endfor
distance[src]=0
Q=set(switches)
while len(Q)>0:
        u=minimum_distance(distance,Q)
        Q.remove(u)
        for p in switches:
        if adjacency[u][p] != None: # Edge
                w=1 # Weight
                new_dist=distance[u]+w
                if new_dist<distance[p]:
                distance[p]=new_dist
                previous[p]=u
                #Endif
        #Endif
        #Endfor
#Endwhile
r=[]
p=dst #goal
#print "goal p=",p
r.append(p)
q=previous[p]
while q is not None:
        if q==src:
        r.append(q)
        break
        p=q
        r.append(p)
        q=previous[p]
#Endwhile
r.reverse()
if src==dst:
        path=[src]
else:
        path=r
#Endif
localpath.append(path)
#print "appended path->",path
#Endfor range(2)
a=localpath[1][1:]
path=localpath[0] + a
#cost_path
path_only_sw=path
cost=len(path)
# Attach Ports #
global GLOBAL_IN_PORT
r=[]
in_port=first_port
for s1,s2 in zip(path[:-1],path[1:]):
```

```
        out_port=adjacency[s1][s2]
        # Ensure in_port,out_port are valid.
        if out_port==in_port :
                #print "[UPDATE]Path Contains: OUT_PORT==IN_PORT"
                #print "out_port:",out_port,"  in_port:",in_port
                out_port=GLOBAL_IN_PORT
                #print " SET-> out_port=GLOBAL_IN_PORT"
        # Endif
        r.append((s1,in_port,out_port))
        in_port=adjacency[s2][s1]
        #Endfor
        r.append((dst,in_port,final_port))
        # Store Data
        cost_path.append([cost]) # Cost (total sw)
        cost_path.append(r)   # Path (with ports)
        cost_path.append(path)   # Path (only sw)
        #print "Calculation:\nCost->%s\nPath (switches/ports)->%s\nPath
(switches)->%s"%(cost_path[0],cost_path[1],cost_path[2])
        #print "Returning cost_path..."
        return cost_path
#Enddef

"""
        Module to start Algorithm
        Receives Packet In.
        Decides which algorithm to run.
        Runs Algorithm.
        Looks up calculated flows from table.
        Return a path.
"""
#def get_path_module_TEST(mymac[src][0],mymac[dst][0],mymac[src][1],mymac[dst][1],src,dst):
def get_path_module(src,dst,first_port,final_port,srchost,dsthost):
        # Local Vars
        global eth_to_host
        global host_to_eth
        global ALGO_EXECUTED
        global vm_mb_paths
        global ALGO_CHOICE

        # Extract hosts #
        vm=(eth_to_host[srchost],eth_to_host[dsthost])

        #Start Module#
        print("\n\t** Start: M O D U L E **")

        #date#
        ts=time.time()
        st=datetime.datetime.fromtimestamp(ts).strftime('%Y-%m-%d %H:%M:%S')
        print ("[DATE] %s" %(st) )

        # Check if algorithm has been executed. #
        if ALGO_EXECUTED is not True:
        print "[CHECK] Next-> START ALGORITHM..."

        if ALGO_CHOICE==1:
                run_algo_vm()          # VM ALGORITHM #
        elif ALGO_CHOICE==2:
```

```
                run_algo_mb()         # MB ALGORITHM #
        elif ALGO_CHOICE==3:
                run_algo_vm_mb_based() # VM+MB ALGORITHM #
        elif ALGO_CHOICE==4:
                print "MCF_ALGO NOT DONE-> EXIT"
                #run_algo_vm_mb_based() # VM+MB ALGORITHM #
        elif ALGO_CHOICE==5:          # ALL ALGOS    #
                run_algo_vm()         #  1           #
                run_algo_mb()         #  2           #
                run_algo_vm_mb_based() #  3                 #
        else:
                print "\nALGO_CHOICE: INVALID!\n***\n"
        else:
        print "CHECK IF ALGO_EXECUTED-> TRUE   Next-> READY."
        #Endif

        # Print incoming flow. #
        print "[FLOW] Traffic FROM: %s(%s)   TO: %s(%s)"%(vm[0],srchost,vm[1],dsthost)

        # Check vmpairs list #
        if check_vm_in_set(vm):
        print "Lookup: VM Pair               ->",vm
        print "Found : Assigned MB           ->",vmpairs3[ get_vm_in_set(vm) ]
        print "Found : Path                  ->",vm_mb_paths[ vm ]
        print "Cost  :                       ->",len(vm_mb_paths[ vm ])
        print "Status:                       -> Done"
        return vm_mb_paths[vm]
        else: # VM is not in set. Get normal path. #
        return get_path(src,dst,first_port,final_port)
        print "Module Done.\nEnd\n\t* * * *."
# End function #

"""
        Module TEST
        Pseudoflow 1
        Check vm in path
        Calculate Algo
        Check Alog Calculated (Tmp Disabled)
        Run Algo if Not Calculated

"""
def get_path_module_TESTER_(src,dst,first_port,final_port,src_host,dst_host):
        global ALGO_EXECUTED
        print("\n\t** Start: M O D U L E **")
        print "[TEST] run_algo_2() " #Check:New
check()->",check_vm_in_set((eth_to_host[src_host],eth_to_host[dst_host]))

        print "Algo Executed: ",ALGO_EXECUTED
        run_algo_mb()
        print "End.\n\n\t* * * *\n"
        print "[Not Found] Run Normal Path. No Middleboxes."
        return get_path(src,dst,first_port,final_port)
        #
        # If Algo vm based has finished
        # the, return paths stored in the table that
        # were calculated at start of the module.
        #
```

```
#Enddef

"""
       V M  B A S E D
       VM Based ALGORITHM
       run_algo_()
       Run algorithms for all paths.
       Calculates all paths for i assigned to j.
       Stores results.
"""
#def run_algo(src,dst,first_port,final_port):
def run_algo_vm():
       # Run Reset()
       init_reset()

       # Globals
       global capacity
       global load
       global vmpairs3
       global localswitches2
       global costtotal
       global vm_mb_paths
       global algo_paths_calculated
       global ALGO_EXECUTED

       print "~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~"
       print "~*  V M  B A S E D    ALGORITHM*~"
       print "~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~"
       # Print vmpairs,mb set, capacity #
       printheader()

       for vm in vmpairs3: # Tuples
       costmin= float('Inf')
       cost=0
       path=[]
       vm_i=None
       mb_j=None
       for mb in localswitches2: # Integer
              if load[mb]<capacity:
              #                                                   #
              # cost_path = [ [cost] , [sw & ports] , [sw only] ] #
              #                                                   #
              cost_path = call_get_path_cost_2(vm,mb)
              cost=cost_path[0][0] # Cost
              if(cost<costmin):
                     vm_i          = vm # Tuple i.e. ("h1","h7")
                     mb_j          = mb # Integer
                     costmin       = cost
                     path          = cost_path[1] # SW & Ports
                     path_sw       = cost_path[2] # Switch only list
              #Endif
       #Endfor mb
       # Update Load #
       load[mb_j]=load[mb_j] + 1
       # Update Cost #
       costtotal=costtotal + costmin
       vmpairs3[vm_i]=mb_j # Assignment i->j
```

```
	vm_mb_paths[vm_i]=path #[Tuple]->Path
	#                     #
	# Get Reverse Flow->
	#   Needed for reply
	#   Store flow in table
	#   Does not affect the cost.
	call_get_path_cost_2_reverse(vm_i,mb_j)
	# Update                            #
	#print "[NEW ASSIGNMENT] "
	#print "VM Pair-> %s Assigned MB-> %s"%(vm_i,mb_j)
	#print "Flow (SW Only)       : %s" %(path_sw)
	#print "Flow (SW,Ports)   : %s" %(vm_mb_paths[vm_i])
	#print "Cost                 : %s" %(costmin)
	#print "Load[%s]             : %s" %(mb_j,load[mb_j])
	#Endfor VM
	# "[RESULTS]" #
	report_path_results()
	print ""
	print "TOTAL COST            : ",costtotal
	# ALGO has been executed. Set to True.
	ALGO_EXECUTED         = True
	#print "SET: ALGO_EXECUTED=",ALGO_EXECUTED
	print "[FINISHED]\n\t****\n"
# End of V M  B A S E D                     #

"""
	MB-BASED auxiliary parts...
	Function:
	Part of MB-BASED, VM+MB BASED
"""
def clonevm(): # Clone original vmpairs set
	global vmpairs3
	vmlist=[]
	for vm in vmpairs3:
	vmlist.append(vm)
	#print"Cloned:",vmlist,"\nEnd."
	return vmlist
#Enddef
"""
	Sort vm list in relation to a mb.
	part of MB BASED
	List of VM Pairs will be sorted and it
	will depend on the minimum cost path
	produced in relation to middlebox j.
"""
def sort(availablevm,mb): # Sort vm pairs in relation to mb
	# Start #
	sortlist=[]
	cost   =None
	#print "START: Sorting VM Pairs List in relation to-> MB:",mb
	availablevm2=availablevm
	#availablevm2=clonevm()
	while (len(availablevm2)>0):
	costmin=float('Inf')
	vm_i=None
	for vm in availablevm2:
		cost_path   =call_get_path_cost_2(vm,mb)
```

```
            cost           =cost_path[0][0]
            if cost<costmin:
            costmin=cost
            vm_i=vm
            #Endif
        #Endfor
        sortlist.append(vm_i)
        availablevm2.remove(vm_i)
        #print "\tVM->%s  Cost->%s"%(vm_i,costmin)
        #Endwhile
        return sortlist
# #
#Enddef
"""
        Function Store Paths
        Part of MB Based
"""
def store_paths_mb(mb_assigned_vmlist):
        # Globals #
        global costtotal
        global vm_mb_paths

        #print "\n\tStoring flows in flow table."
        # Iterate each MB #
        for mb in mb_assigned_vmlist:
        # Reset list of vmpairs #
        vmlist=[]
        # Get list of assigned vmpairs #
        vmlist=mb_assigned_vmlist[mb]
        path_sw_ports=[]
        path_sw=[]
        # Iterate each vm in the list of vmpairs assigned to current MB #
        for vm in vmlist:
                # Get Paths #
                cost_path=call_get_path_cost_2(vm,mb)
                # Ger Reverse() for reply messages #
                call_get_path_cost_2_reverse(vm,mb)
                # Populate Table #
                costmin       =cost_path[0][0] # Cost             #
                path_sw_ports   =cost_path[1]# Path SW and Ports #
                path_sw       =cost_path[2]  # Path SW Only       #
                # Add paths to table #
                # The table Contains paths for VM Pairs in relation to the MB it was assigned.
                vm_mb_paths[vm] =path_sw_ports # Table of (vm_i,mb_j) and assigned paths
                # Update Total Cost
                costtotal=costtotal +costmin
                # Print Stats #
                #print "[UPDATE] New VM for current MB"
                #print "Current MB           :",mb
                #print "VM: %s assigned MB: %s"%(vm,mb)
                #print "Flow (SW Only)       : %s" %(path_sw)
                #print "Flow (SW,Ports)   : %s" %(vm_mb_paths[vm])
                #print "Cost                 : %s" %(costmin)
        #Endfor End MB iteration
#Enddef

"""
```

```
        MB-BASED ALGORITHM
"""
def run_algo_mb():
        # Run Reset()
        init_reset()

        # Globals #
        global localswitches2
        global mb_assigned_vmlist
        global ALGO_EXECUTED
        global capacity

        # Start #
        print "~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~"
        print "~*  M B  B A S E D    ALGORITHM*~"
        print "~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~"

        # Print #
        printheader()

        availablevm=clonevm()
        for mb in localswitches2: # Iterate Middleboxes
        x_sort=[]
        x_sort=sort(availablevm,mb) # Sort VM Pairs List in relation a Middlebox
        # Add assgined vms to mb #
        x_sort.reverse() # Reverse list, move small to end, pop() minimum cost pairs.
        # Reset #
        tempsort=[]
        #print "[UPDATE] MB:",mb," Assigning the following VM Pairs ->  "
        for i in range(capacity):
                #check if emtpy
                if len(x_sort)==0: # Empty list
                break
                #Endif
                # Pop next least min cost vmpair #
                element=x_sort.pop()
                # Add vmpair to MB's set #
                tempsort.append(element)
                #print "\t ",element
                # Assign vm pair to current mb #
                vmpairs3[ get_vm_in_set(element) ]=mb
        #Endfor
        mb_assigned_vmlist[mb]=tempsort
        availablevm=x_sort # Store remaining vm-pairs #
        #Endfor
        # Store the paths and assignments #
        #print "Storing paths..."
        store_paths_mb(mb_assigned_vmlist)
        # "[RESULTS]"
        report_path_results()
        print ""
        print "TOTAL COST            : ",costtotal
        # ALGO has been executed. Set to True.
        ALGO_EXECUTED        = True
        #print "SET: ALGO_EXECUTED=",ALGO_EXECUTED
        print "[FINISHED]\n\t****\n"
#              End            #
```

```
#                             #
#                             #
# Enddef                      #

"""
       VM+MB BASED ALGORITHM
"""
def run_algo_vm_mb_based():
       # Run Reset()
       init_reset()

       # Global #
       global vmpairs3
       global localswitches2
       global load
       global capacity
       global vm_mb_paths
       global costtotal
       global ALGO_EXECUTED

       # Start #
       print "~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~"
       print "~*  V M + M B  B A S E D     ALGORITHM*~"
       print "~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~"
       # Print vmpairs,mb set, capacity #
       printheader()

       vmlist = clonevm() # Clone list of vm pairs
       while len(vmlist)>0:
       costmin=float('Inf') # Set to INF #
       vm_i=None
       mb_j=None
       path_sw_ports=[]
       for vm in vmlist:
              for mb in localswitches2:
              if load[mb]<capacity:                      # Capacity available #
                     cost_path =call_get_path_cost_2(vm,mb) # 3-Element Array  #
                     cost   =cost_path[0][0]
                     if cost<costmin:                    # Check if minimum cost #
                     costmin       =cost          # Update Cost         #
                     path_sw_ports   =cost_path[1] # Path SW, Ports     #
                     path_sw       =cost_path[2] # Path SW Only        #
                     vm_i          =vm
                     mb_j          =mb
                     #print
                     #Endif
              #Endfor
       #Endfor
       vmpairs3[get_vm_in_set(vm_i)] = mb_j              # Assigne vm->mb            #
       vm_mb_paths[vm_i]            = path_sw_ports      # store path for vm->mb      #
       costtotal                    = costtotal+costmin # Total Cost                #
       load[mb_j]                   = load[mb_j]+1       # Increment Load            #
       call_get_path_cost_2_reverse(vm_i,mb_j)           # Reverse() for vm responses #
       vmlist.remove(vm_i)                               # pop()                     #
       #Endwhile
       # "[RESULTS]"
       report_path_results()
```

```
        print ""
        print "TOTAL COST            : ",costtotal
        # ALGO has been executed. Set to True. #
        ALGO_EXECUTED        = True
        #print "SET: ALGO_EXECUTED=",ALGO_EXECUTED
        print "[FINISHED]\n\t****\n"
#Enddef

#def run_algo_4():
        #print "MCF Algorithm"
##Enddef

"""
        Print vm list
"""
def printvm(vmdict):
        # Dict -> List #
        vmlist=[vm for vm in vmdict]
        # Print #
        i=0
        while i<len(vmlist):
        print "\t",vmlist[i],
        if( (i+1)%2==0 ):
                print ""
        i=i+1
#Enddef
"""
        Print Paths: Switches,ports
"""
def printpaths(vmmbpaths):#vm_mb_paths
        #print ("")
        for vm in vmmbpaths:
        print ("%s-> %s"%(vm,vmmbpaths[vm]))
        #Endfor
        print("")
#Enddef
"""
        Print Header
"""
def printheader():

        # Globals #
        global vmpairs3
        global localswitches2
        global capacity

        # Print #
        print "Total VM Pairs :",len(vmpairs3.items())
        print "Total Middleboxes :",len(localswitches2)
        print "CAPACITY      :",capacity
        print ""
        print "Middlebox Set :%s"%(localswitches2)
        print "VM Pairs Set  :"
        printvm(vmpairs3)
        print ""

#Enddef
```

```
#End printheader()
def printresults():
	global vmpairs3
	global vm_mb_paths
	print ""
	for vm in vmpairs3:
	#print "[ASSIGNMENT] "
	print "."
	print "VM Pair        :",(vm)
	print "Assigned MB    : %s" %(vmpairs3[vm])
	print "Flow (SW,Ports)   : %s" %(vm_mb_paths[vm])
	print "Cost           : %s" %(len(vm_mb_paths[vm]))
	#end
	# Update                             #

def report_path_results():
	if REPORT_PATH_RESULTS_ENABLED:
	print "[PATH RESULTS]"
	printresults()
	else:
	print "[REPORT_PATH_RESULTS_ENABLED] IS OFF"

# Minimum Distance for Dijkstra
def minimum_distance(distance, Q):
	min = float('Inf')
	node = 0
	for v in Q:
	if distance[v] < min:
		min = distance[v]
		node = v
	return node

"""
	Calculates path between vm pairs but does not consider
	any middleboxes. A shortest path algorithm is used to
	fin the path.
"""
def get_path (src,dst,first_port,final_port): #Regular, No middlebox
			#label::VM-Based
			# similar to MB-Based?
	#localswitches = [6,18] #17] #Added by Darshit src to 6, 6 to dest; src to 17, 17 to
dest  ---start
	#vmpairs=[101,102,201,202,301,302] # No vm in set so Algo runs with no mb
	originalsrc = src
	originaldst = dst
	localpath = []
	for i in range(1): #For MB. Any VM-Pair with source h1,h2,h7,h8 will be assigned mb0 or
j=0
	#print "Shortest Path with No Middleboxes"
	if i==0:
		src=originalsrc
		dst=originaldst
	#Endif
	#Dijkstra's algorithm
	#print ("Called get_path(). src sw: %s, port: %s-> dst sw: %s, port: %s"%(
			#src,first_port,dst,final_port))
	distance = {}
```

```
        previous = {}
        for dpid in switches:
                distance[dpid] = float('Inf')
                previous[dpid] = None
        #Endfor
        distance[src]=0
        Q=set(switches)
        #print "Q=", Q   # commented by Darshit -- on December 7, 2018
        while len(Q)>0:
                u = minimum_distance(distance, Q)
                Q.remove(u)
                for p in switches:
                if adjacency[u][p]!=None:
                        w = 1
                        if distance[u] + w < distance[p]:
                        distance[p] = distance[u] + w
                        previous[p] = u
                        #Endif
                #Endif
                #Endfor
        #Endwhile
        r=[]
        p=dst
        r.append(p)
        q=previous[p]
        while q is not None:
                if q == src:
                r.append(q)
                break
                #Endif
                p=q
                r.append(p)
                q=previous[p]
        #Endwhile
        r.reverse()
        if src==dst:
                path=[src]
        else:
                path=r
        #Endif
        localpath.append(path)  #Added by Darshit  ---start
        #Endfor in range()
        path = localpath[0] # zip works with list not set for Non MB Version
        #print "path-> ", path       #Added by Carlos
        r = []
        in_port = first_port
        for s1,s2 in zip(path[:-1],path[1:]):
        out_port = adjacency[s1][s2]
        r.append((s1,in_port,out_port))
        in_port = adjacency[s2][s1]
        #Endfor
        r.append((dst,in_port,final_port))
        print "[NORMAL PATH] Shortest Path with No Middleboxes->  ",  r   #Added by Carlos
        return r
#Enddef
class ProjectController(app_manager.RyuApp):
        OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]
```

```
        def __init__(self, *args, **kwargs):
        super(ProjectController, self).__init__(*args, **kwargs)
        self.mac_to_port = {}
        self.topology_api_app = self
        self.datapath_list=[]
        self.cnt=0
        #init_vars() # --Added by Carlos

        # Handy function that lists all attributes in the given object
        def ls(self,obj):
        print("\n".join([x for x in dir(obj) if x[0] != "_"]))
        def add_flow(self, datapath, in_port, dst, actions):
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser
        match = datapath.ofproto_parser.OFPMatch(in_port=in_port, eth_dst=dst)
        inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,actions)]
        mod = datapath.ofproto_parser.OFPFlowMod(datapath=datapath, match=match,
cookie=0,command=ofproto.OFPFC_ADD, idle_timeout=0,
hard_timeout=0,priority=ofproto.OFP_DEFAULT_PRIORITY, instructions=inst)

        datapath.send_msg(mod)

        def install_path(self, p, ev, src_mac, dst_mac):

        #print "install_path is called"

        #print "p=", p, " src_mac=", src_mac, " dst_mac=", dst_mac
        msg = ev.msg
        datapath = msg.datapath
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        for sw, in_port, out_port in p:
               #print src_mac,"->", dst_mac, "via ", sw, " in_port=", in_port, " out_port=",
out_port
               match=parser.OFPMatch(in_port=in_port, eth_src=src_mac, eth_dst=dst_mac)
               actions=[parser.OFPActionOutput(out_port)]
               datapath=self.datapath_list[int(sw)-1]
               inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS , actions)]
               mod = datapath.ofproto_parser.OFPFlowMod(datapath=datapath, match=match,
idle_timeout=0, hard_timeout=0,priority=1, instructions=inst)
               datapath.send_msg(mod)

        @set_ev_cls(ofp_event.EventOFPSwitchFeatures , CONFIG_DISPATCHER)
        def switch_features_handler(self , ev):
        #print "switch_features_handler is called"
        datapath = ev.msg.datapath
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser
        match = parser.OFPMatch()
        actions = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER, ofproto.OFPCML_NO_BUFFER)]
        inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,actions)]
        mod = datapath.ofproto_parser.OFPFlowMod(datapath=datapath, match=match,
cookie=0,command=ofproto.OFPFC_ADD, idle_timeout=0, hard_timeout=0,priority=0,
instructions=inst)
        datapath.send_msg(mod)
```

```
        @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)

        def _packet_in_handler(self, ev):

        # Global #
        global SW_NOT_DISCOVERED
        #print "dir->ev.msg->",dir(ev.msg.buffer_id)#--
ev.msg.buffer_id;;ev.msg.buffer_id#--added carlos
        #print "ev.msg.buffer_id->",(ev.msg.buffer_id)#-- ev.msg.buffer_id;;ev.msg.buffer_id
        msg = ev.msg
        datapath = msg.datapath
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser
        in_port = msg.match['in_port']
        pkt = packet.Packet(msg.data)
        eth = pkt.get_protocol(ethernet.ethernet)
        #print "eth.ethertype=", eth.ethertype
        #avodi broadcast from LLDP

        if eth.ethertype==35020:
                return

        dst = eth.dst
        src = eth.src
        dpid = datapath.id
        self.mac_to_port.setdefault(dpid, {})

        if src not in mymac.keys():
                mymac[src]=( dpid,  in_port)
                self.cnt=self.cnt+1
                print ("::Added (count:%d):: Host:%s detected by switch:%s in port:%s"
%(self.cnt,src,dpid,in_port) )
                #print (". dir(datapath)->",dir(datapath)
                #print("datapaht.address->",datapath.address)
                #print("datapath.xid->",datapath.xid)
                #print ("datapath->",datapath)
                #print ("datapath._get_ports->",datapath._get_ports())
                #print (". Done\n")
                #print "mymac=", mymac

        if dst in mymac.keys():
                #
                # Test host name to eth to swid to port number
                if SW_NOT_DISCOVERED:
                SW_NOT_DISCOVERED=False
                init_vars() # --Added by Carlos
                #printmyname()
                # PACKET IN COUNTER #
                global packet_in_cnt
                packet_in_cnt=packet_in_cnt+1
                # Pring Packet In #
                print ("\n*")
                print ("[PACKET IN] PACKET_IN (count): %s" %(packet_in_cnt))
                print ("[NEW FLOW]  Switch: %s  Flow: %s(%s) -> %s(%s)  (Not in Flow Table)"
%(dpid,eth_to_host[src],src,eth_to_host[dst],dst))
                #print ("[NEW FLOW] Switch:%s Flow: %s->%s (Not in Flow Table) "
%(dpid,mymac[src][0],mymac[dst][0])  )
```

```
                #print ("[NEW FLOW] From: (%s) %s -> To: (%s) %s" 
%(eth_to_host[src],src,eth_to_host[dst],dst))
                #
                #p = get_path(mymac[src][0], mymac[dst][0], mymac[src][1], mymac[dst][1])
                #p = get_path_module(mymac[src][0],mymac[dst][0],mymac[src][1],mymac[dst][1])


p=get_path_module(mymac[src][0],mymac[dst][0],mymac[src][1],mymac[dst][1],src,dst)
#p=get_path_module_TESTER_(mymac[src][0],mymac[dst][0],mymac[src][1],mymac[dst][1],src,dst)
                #print "p received->",p
                self.install_path(p, ev, src, dst)
                out_port = p[0][2]
                #print "out_port->",out_por
                #print "ofproto.OFPP_IN_PORT: ",ofproto.OFPP_IN_PORT
        else:
                out_port = ofproto.OFPP_FLOOD
        actions = [parser.OFPActionOutput(out_port)]
        # install a flow to avoid packet_in next time
        if out_port != ofproto.OFPP_FLOOD:
                match = parser.OFPMatch(in_port=in_port, eth_src=src, eth_dst=dst)
        data=None
        if msg.buffer_id==ofproto.OFP_NO_BUFFER:
        data=msg.data
        #data=None #--added by carlos
        out = parser.OFPPacketOut(datapath=datapath, buffer_id=msg.buffer_id, 
in_port=in_port,actions=actions, data=data)

        #if out_port != ofproto.OFPP_FLOOD:# -- added by Carlos
                #print "of_packet_out->",out," out_port->",out_port
        datapath.send_msg(out)
        @set_ev_cls(event.EventSwitchEnter)

        def get_topology_data(self, ev):
        global switches
        switch_list = get_switch(self.topology_api_app, None)
        #print ("::TEST::from self.topology_api_app->   switch_list")
        #print (switch_list) # Prints Switch Object memory locations
        #print ("::END switch_list")
        switches=[switch.dp.id for switch in switch_list]
        #print ("::TEST from switch.dp.id in switch_list get-> ",switches) # Prints list Switch 
Id Integer number (1,2,3,...) for each switch
        #print (":::TEST::ls(self,obj)->",ls(self,switches) ) # error not Print
        self.datapath_list=[switch.dp for switch in switch_list]
        #print ("::TEST from switch.dp in switch_list get-> ",self.datapath_list) # Prints 
ryu.controller.controller.Datapath Object for each switch
        #print "self.datapath_list=", self.datapath_list
        print ("switches=", switches) # Prints list Switch Id number (1,2,3,...) for each 
switch

        links_list = get_link(self.topology_api_app, None)
        mylinks=[(link.src.dpid,link.dst.dpid,link.src.port_no,link.dst.port_no) for link in 
links_list]

        for s1,s2,port1,port2 in mylinks:
        adjacency[s1][s2]=port1
        adjacency[s2][s1]=port2
        #print s1,s2,port1,port2
```

```
#Endfor
```

10.2. Ryu VM Pairs Data

```
# Python File

# List of VMPairs
# Total 240

vmlist=[]
vmlist.append(('h3', 'h1'))
vmlist.append(('h2', 'h16'))
vmlist.append(('h2', 'h8'))
vmlist.append(('h1', 'h7'))
vmlist.append(('h3', 'h13'))
vmlist.append(('h4', 'h14'))
vmlist.append(('h5', 'h15'))
vmlist.append(('h6', 'h16'))
vmlist.append(('h5', 'h8'))
vmlist.append(('h6', 'h9'))
vmlist.append(('h7', 'h13'))
vmlist.append(('h8', 'h4'))
vmlist.append(('h9', 'h2'))
vmlist.append(('h10', 'h6'))
vmlist.append(('h11', 'h15'))
vmlist.append(('h12', 'h13'))
vmlist.append(('h14', 'h12'))
vmlist.append(('h11', 'h5'))
vmlist.append(('h8', 'h11'))
vmlist.append(('h10', 'h7'))
vmlist.append(('h5', 'h12'))
vmlist.append(('h12', 'h16'))
vmlist.append(('h10', 'h3'))
vmlist.append(('h14', 'h3'))
vmlist.append(('h16', 'h2'))
vmlist.append(('h16', 'h6'))
vmlist.append(('h7', 'h14'))
vmlist.append(('h3', 'h14'))
vmlist.append(('h15', 'h14'))
vmlist.append(('h15', 'h11'))
vmlist.append(('h7', 'h12'))
vmlist.append(('h10', 'h2'))
vmlist.append(('h3', 'h16'))
vmlist.append(('h12', 'h6'))
vmlist.append(('h8', 'h2'))
vmlist.append(('h15', 'h6'))
vmlist.append(('h8', 'h1'))
vmlist.append(('h6', 'h15'))
vmlist.append(('h12', 'h4'))
vmlist.append(('h4', 'h12'))
vmlist.append(('h16', 'h10'))
vmlist.append(('h1', 'h13'))
vmlist.append(('h13', 'h12'))
vmlist.append(('h8', 'h16'))
```

```
vmlist.append(('h1', 'h10'))
vmlist.append(('h11', 'h6'))
vmlist.append(('h2', 'h12'))
vmlist.append(('h1', 'h12'))
vmlist.append(('h11', 'h1'))
vmlist.append(('h13', 'h10'))
vmlist.append(('h7', 'h16'))
vmlist.append(('h13', 'h15'))
vmlist.append(('h7', 'h9'))
vmlist.append(('h8', 'h5'))
vmlist.append(('h15', 'h2'))
vmlist.append(('h8', 'h9'))
vmlist.append(('h12', 'h11'))
vmlist.append(('h3', 'h4'))
vmlist.append(('h9', 'h4'))
vmlist.append(('h1', 'h14'))
vmlist.append(('h12', 'h5'))
vmlist.append(('h10', 'h12'))
vmlist.append(('h16', 'h11'))
vmlist.append(('h15', 'h9'))
vmlist.append(('h15', 'h4'))
vmlist.append(('h9', 'h8'))
vmlist.append(('h12', 'h14'))
vmlist.append(('h16', 'h8'))
vmlist.append(('h2', 'h15'))
vmlist.append(('h10', 'h5'))
vmlist.append(('h9', 'h12'))
vmlist.append(('h16', 'h12'))
vmlist.append(('h2', 'h7'))
vmlist.append(('h6', 'h5'))
vmlist.append(('h1', 'h11'))
vmlist.append(('h10', 'h1'))
vmlist.append(('h2', 'h6'))
vmlist.append(('h5', 'h2'))
vmlist.append(('h8', 'h14'))
vmlist.append(('h2', 'h1'))
vmlist.append(('h1', 'h6'))
vmlist.append(('h6', 'h4'))
vmlist.append(('h9', 'h5'))
vmlist.append(('h13', 'h6'))
vmlist.append(('h9', 'h7'))
vmlist.append(('h14', 'h7'))
vmlist.append(('h10', 'h14'))
vmlist.append(('h14', 'h15'))
vmlist.append(('h13', 'h1'))
vmlist.append(('h12', 'h10'))
vmlist.append(('h5', 'h14'))
vmlist.append(('h7', 'h11'))
vmlist.append(('h3', 'h11'))
vmlist.append(('h13', 'h14'))
vmlist.append(('h4', 'h1'))
vmlist.append(('h11', 'h12'))
vmlist.append(('h13', 'h16'))
vmlist.append(('h10', 'h9'))
vmlist.append(('h12', 'h2'))
vmlist.append(('h12', 'h1'))
vmlist.append(('h1', 'h3'))
```

```
vmlist.append(('h8', 'h6'))
vmlist.append(('h4', 'h5'))
vmlist.append(('h4', 'h8'))
vmlist.append(('h3', 'h12'))
vmlist.append(('h5', 'h7'))
vmlist.append(('h13', 'h7'))
vmlist.append(('h4', 'h10'))
vmlist.append(('h1', 'h5'))
vmlist.append(('h14', 'h16'))
vmlist.append(('h6', 'h7'))
vmlist.append(('h1', 'h16'))
vmlist.append(('h3', 'h10'))
vmlist.append(('h9', 'h16'))
vmlist.append(('h15', 'h1'))
vmlist.append(('h16', 'h7'))
vmlist.append(('h11', 'h7'))
vmlist.append(('h4', 'h9'))
vmlist.append(('h2', 'h9'))
vmlist.append(('h5', 'h3'))
vmlist.append(('h14', 'h6'))
vmlist.append(('h4', 'h13'))
vmlist.append(('h10', 'h8'))
vmlist.append(('h14', 'h10'))
vmlist.append(('h2', 'h3'))
vmlist.append(('h13', 'h2'))
vmlist.append(('h13', 'h11'))
vmlist.append(('h11', 'h3'))
vmlist.append(('h16', 'h13'))
vmlist.append(('h8', 'h12'))
vmlist.append(('h13', 'h9'))
vmlist.append(('h10', 'h13'))
vmlist.append(('h13', 'h3'))
vmlist.append(('h10', 'h4'))
vmlist.append(('h3', 'h6'))
vmlist.append(('h7', 'h4'))
vmlist.append(('h1', 'h9'))
vmlist.append(('h8', 'h15'))
vmlist.append(('h13', 'h8'))
vmlist.append(('h16', 'h1'))
vmlist.append(('h14', 'h1'))
vmlist.append(('h5', 'h16'))
vmlist.append(('h11', 'h16'))
vmlist.append(('h12', 'h9'))
vmlist.append(('h4', 'h2'))
vmlist.append(('h15', 'h7'))
vmlist.append(('h6', 'h12'))
vmlist.append(('h8', 'h10'))
vmlist.append(('h9', 'h13'))
vmlist.append(('h15', 'h13'))
vmlist.append(('h3', 'h15'))
vmlist.append(('h7', 'h2'))
vmlist.append(('h1', 'h15'))
vmlist.append(('h6', 'h3'))
vmlist.append(('h15', 'h10'))
vmlist.append(('h11', 'h13'))
vmlist.append(('h4', 'h7'))
vmlist.append(('h11', 'h4'))
```

```
vmlist.append(('h5', 'h1'))
vmlist.append(('h7', 'h10'))
vmlist.append(('h3', 'h8'))
vmlist.append(('h11', 'h10'))
vmlist.append(('h15', 'h3'))
vmlist.append(('h9', 'h3'))
vmlist.append(('h6', 'h13'))
vmlist.append(('h14', 'h8'))
vmlist.append(('h14', 'h2'))
vmlist.append(('h13', 'h5'))
vmlist.append(('h15', 'h8'))
vmlist.append(('h16', 'h14'))
vmlist.append(('h6', 'h11'))
vmlist.append(('h9', 'h14'))
vmlist.append(('h6', 'h8'))
vmlist.append(('h2', 'h13'))
vmlist.append(('h10', 'h11'))
vmlist.append(('h7', 'h5'))
vmlist.append(('h15', 'h16'))
vmlist.append(('h16', 'h3'))
vmlist.append(('h14', 'h4'))
vmlist.append(('h11', 'h9'))
vmlist.append(('h4', 'h15'))
vmlist.append(('h5', 'h13'))
vmlist.append(('h9', 'h10'))
vmlist.append(('h4', 'h6'))
vmlist.append(('h2', 'h10'))
vmlist.append(('h10', 'h15'))
vmlist.append(('h6', 'h2'))
vmlist.append(('h3', 'h5'))
vmlist.append(('h8', 'h7'))
vmlist.append(('h12', 'h15'))
vmlist.append(('h9', 'h6'))
vmlist.append(('h14', 'h11'))
vmlist.append(('h9', 'h15'))
vmlist.append(('h16', 'h5'))
vmlist.append(('h6', 'h10'))
vmlist.append(('h5', 'h4'))
vmlist.append(('h14', 'h5'))
vmlist.append(('h14', 'h13'))
vmlist.append(('h3', 'h2'))
vmlist.append(('h5', 'h6'))
vmlist.append(('h16', 'h9'))
vmlist.append(('h5', 'h9'))
vmlist.append(('h14', 'h9'))
vmlist.append(('h12', 'h8'))
vmlist.append(('h1', 'h8'))
vmlist.append(('h5', 'h10'))
vmlist.append(('h10', 'h16'))
vmlist.append(('h5', 'h11'))
vmlist.append(('h12', 'h7'))
vmlist.append(('h2', 'h11'))
vmlist.append(('h15', 'h5'))
vmlist.append(('h4', 'h16'))
vmlist.append(('h9', 'h11'))
vmlist.append(('h4', 'h3'))
vmlist.append(('h13', 'h4'))
```

```
vmlist.append(('h7', 'h15'))
vmlist.append(('h11', 'h14'))
vmlist.append(('h6', 'h14'))
vmlist.append(('h1', 'h2'))
vmlist.append(('h7', 'h6'))
vmlist.append(('h16', 'h4'))
vmlist.append(('h7', 'h3'))
vmlist.append(('h11', 'h2'))
vmlist.append(('h2', 'h4'))
vmlist.append(('h15', 'h12'))
vmlist.append(('h4', 'h11'))
vmlist.append(('h3', 'h9'))
vmlist.append(('h16', 'h15'))
vmlist.append(('h6', 'h1'))
vmlist.append(('h7', 'h1'))
vmlist.append(('h12', 'h3'))
vmlist.append(('h7', 'h8'))
vmlist.append(('h1', 'h4'))
vmlist.append(('h8', 'h3'))
vmlist.append(('h3', 'h7'))
vmlist.append(('h2', 'h14'))
vmlist.append(('h8', 'h13'))
vmlist.append(('h11', 'h8'))
vmlist.append(('h2', 'h5'))
vmlist.append(('h9', 'h1'))
```

10.3. Ryu Superclass

```
# Copyright (C) 2011-2014 Nippon Telegraph and Telephone Corporation.
# Copyright (C) 2011 Isaku Yamahata <yamahata at valinux co jp>
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#      http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
# implied.
# See the License for the specific language governing permissions and
# limitations under the License.

"""
The central management of Ryu applications.

- Load Ryu applications
- Provide `contexts` to Ryu applications
- Route messages among Ryu applications

"""

import inspect
```

```
import itertools
import logging
import sys
import os
import gc

from ryu import cfg
from ryu import utils
from ryu.app import wsgi
from ryu.controller.handler import register_instance, get_dependent_services
from ryu.controller.controller import Datapath
from ryu.controller import event
from ryu.controller.event import EventRequestBase, EventReplyBase
from ryu.lib import hub
from ryu.ofproto import ofproto_protocol

LOG = logging.getLogger('ryu.base.app_manager')

SERVICE_BRICKS = {}


def lookup_service_brick(name):
       return SERVICE_BRICKS.get(name)


def _lookup_service_brick_by_ev_cls(ev_cls):
       return _lookup_service_brick_by_mod_name(ev_cls.__module__)


def _lookup_service_brick_by_mod_name(mod_name):
       return lookup_service_brick(mod_name.split('.')[-1])


def register_app(app):
       assert isinstance(app, RyuApp)
       assert app.name not in SERVICE_BRICKS
       SERVICE_BRICKS[app.name] = app
       register_instance(app)


def unregister_app(app):
       SERVICE_BRICKS.pop(app.name)


def require_app(app_name, api_style=False):
       """
       Request the application to be automatically loaded.

       If this is used for "api" style modules, which is imported by a client
       application, set api_style=True.

       If this is used for client application module, set api_style=False.
       """
       if api_style:
       frm = inspect.stack()[2]  # skip a frame for "api" module
       else:
       frm = inspect.stack()[1]
```

```
        m = inspect.getmodule(frm[0])  # client module
        m._REQUIRED_APP = getattr(m, '_REQUIRED_APP', [])
        m._REQUIRED_APP.append(app_name)
        LOG.debug('require_app: %s is required by %s', app_name, m.__name__)


class RyuApp(object):
        """
        The base class for Ryu applications.

        RyuApp subclasses are instantiated after ryu-manager loaded
        all requested Ryu application modules.
        __init__ should call RyuApp.__init__ with the same arguments.
        It's illegal to send any events in __init__.

        The instance attribute 'name' is the name of the class used for
        message routing among Ryu applications.  (Cf. send_event)
        It's set to __class__.__name__ by RyuApp.__init__.
        It's discouraged for subclasses to override this.
        """

        _CONTEXTS = {}
        """
        A dictionary to specify contexts which this Ryu application wants to use.
        Its key is a name of context and its value is an ordinary class
        which implements the context.  The class is instantiated by app_manager
        and the instance is shared among RyuApp subclasses which has _CONTEXTS
        member with the same key.  A RyuApp subclass can obtain a reference to
        the instance via its __init__'s kwargs as the following.

        Example::

        _CONTEXTS = {
               'network': network.Network
        }

        def __init__(self, *args, *kwargs):
               self.network = kwargs['network']
        """

        _EVENTS = []
        """
        A list of event classes which this RyuApp subclass would generate.
        This should be specified if and only if event classes are defined in
        a different python module from the RyuApp subclass is.
        """

        OFP_VERSIONS = None
        """
        A list of supported OpenFlow versions for this RyuApp.
        The default is all versions supported by the framework.

        Examples::

        OFP_VERSIONS = [ofproto_v1_0.OFP_VERSION,
                       ofproto_v1_2.OFP_VERSION]
```

```
If multiple Ryu applications are loaded in the system,
the intersection of their OFP_VERSIONS is used.
"""

@classmethod
def context_iteritems(cls):
"""
Return iterator over the (key, contxt class) of application context
"""
return cls._CONTEXTS.iteritems()

def __init__(self, *_args, **_kwargs):
super(RyuApp, self).__init__()
self.name = self.__class__.__name__
self.event_handlers = {}     # ev_cls -> handlers:list
self.observers = {}   # ev_cls -> observer-name -> states:set
self.threads = []
self.events = hub.Queue(128)
if hasattr(self.__class__, 'LOGGER_NAME'):
       self.logger = logging.getLogger(self.__class__.LOGGER_NAME)
else:
       self.logger = logging.getLogger(self.name)
self.CONF = cfg.CONF

# prevent accidental creation of instances of this class outside RyuApp
class _EventThreadStop(event.EventBase):
       pass
self._event_stop = _EventThreadStop()
self.is_active = True

def start(self):
"""
Hook that is called after startup initialization is done.
"""
self.threads.append(hub.spawn(self._event_loop))

def stop(self):
self.is_active = False
self._send_event(self._event_stop, None)
hub.joinall(self.threads)

def register_handler(self, ev_cls, handler):
assert callable(handler)
self.event_handlers.setdefault(ev_cls, [])
self.event_handlers[ev_cls].append(handler)

def unregister_handler(self, ev_cls, handler):
assert callable(handler)
self.event_handlers[ev_cls].remove(handler)
if not self.event_handlers[ev_cls]:
       del self.event_handlers[ev_cls]

def register_observer(self, ev_cls, name, states=None):
states = states or set()
ev_cls_observers = self.observers.setdefault(ev_cls, {})
ev_cls_observers.setdefault(name, set()).update(states)
```

```
def unregister_observer(self, ev_cls, name):
observers = self.observers.get(ev_cls, {})
observers.pop(name)

def unregister_observer_all_event(self, name):
for observers in self.observers.values():
       observers.pop(name, None)

def observe_event(self, ev_cls, states=None):
brick = _lookup_service_brick_by_ev_cls(ev_cls)
if brick is not None:
       brick.register_observer(ev_cls, self.name, states)

def unobserve_event(self, ev_cls):
brick = _lookup_service_brick_by_ev_cls(ev_cls)
if brick is not None:
       brick.unregister_observer(ev_cls, self.name)

def get_handlers(self, ev, state=None):
"""Returns a list of handlers for the specific event.

:param ev: The event to handle.
:param state: The current state. ("dispatcher")
              If None is given, returns all handlers for the event.
              Otherwise, returns only handlers that are interested
              in the specified state.
              The default is None.
"""
ev_cls = ev.__class__
handlers = self.event_handlers.get(ev_cls, [])
if state is None:
       return handlers

def test(h):
       if not hasattr(h, 'callers') or ev_cls not in h.callers:
       # dynamically registered handlers does not have
       # h.callers element for the event.
       return True
       states = h.callers[ev_cls].dispatchers
       if not states:
       # empty states means all states
       return True
       return state in states

return filter(test, handlers)

def get_observers(self, ev, state):
observers = []
for k, v in self.observers.get(ev.__class__, {}).iteritems():
       if not state or not v or state in v:
       observers.append(k)

return observers

def send_request(self, req):
"""
Make a synchronous request.
```

```
Set req.sync to True, send it to a Ryu application specified by
req.dst, and block until receiving a reply.
Returns the received reply.
The argument should be an instance of EventRequestBase.
"""

assert isinstance(req, EventRequestBase)
req.sync = True
req.reply_q = hub.Queue()
self.send_event(req.dst, req)
# going to sleep for the reply
return req.reply_q.get()

def _event_loop(self):
while self.is_active or not self.events.empty():
       ev, state = self.events.get()
       if ev == self._event_stop:
       continue
       handlers = self.get_handlers(ev, state)
       for handler in handlers:
       handler(ev)

def _send_event(self, ev, state):
self.events.put((ev, state))

def send_event(self, name, ev, state=None):
"""
Send the specified event to the RyuApp instance specified by name.
"""

if name in SERVICE_BRICKS:
       if isinstance(ev, EventRequestBase):
       ev.src = self.name
       LOG.debug("EVENT %s->%s %s",
              self.name, name, ev.__class__.__name__)
       SERVICE_BRICKS[name]._send_event(ev, state)
else:
       LOG.debug("EVENT LOST %s->%s %s",
              self.name, name, ev.__class__.__name__)

def send_event_to_observers(self, ev, state=None):
"""
Send the specified event to all observers of this RyuApp.
"""

for observer in self.get_observers(ev, state):
       self.send_event(observer, ev, state)

def reply_to_request(self, req, rep):
"""
Send a reply for a synchronous request sent by send_request.
The first argument should be an instance of EventRequestBase.
The second argument should be an instance of EventReplyBase.
"""

assert isinstance(req, EventRequestBase)
assert isinstance(rep, EventReplyBase)
```

```
        rep.dst = req.src
        if req.sync:
               req.reply_q.put(rep)
        else:
               self.send_event(rep.dst, rep)

        def close(self):
        """
        teardown method.
        The method name, close, is chosen for python context manager
        """
        pass


class AppManager(object):
        # singletone
        _instance = None

        @staticmethod
        def run_apps(app_lists):
        """Run a set of Ryu applications

        A convenient method to load and instantiate apps.
        This blocks until all relevant apps stop.
        """
        app_mgr = AppManager.get_instance()
        app_mgr.load_apps(app_lists)
        contexts = app_mgr.create_contexts()
        services = app_mgr.instantiate_apps(**contexts)
        webapp = wsgi.start_service(app_mgr)
        if webapp:
               services.append(hub.spawn(webapp))
        try:
               hub.joinall(services)
        finally:
               app_mgr.close()
               for t in services:
               t.kill()
               hub.joinall(services)
               gc.collect()

        @staticmethod
        def get_instance():
        if not AppManager._instance:
               AppManager._instance = AppManager()
        return AppManager._instance

        def __init__(self):
        self.applications_cls = {}
        self.applications = {}
        self.contexts_cls = {}
        self.contexts = {}

        def load_app(self, name):
        mod = utils.import_module(name)
        clses = inspect.getmembers(mod,
                                     lambda cls: (inspect.isclass(cls) and
```

```
                                            issubclass(cls, RyuApp) and
                                            mod.__name__ ==
                                            cls.__module__))
if clses:
       return clses[0][1]
return None

def load_apps(self, app_lists):
app_lists = [app for app
              in itertools.chain.from_iterable(app.split(',')
                                            for app in app_lists)]
while len(app_lists) > 0:
       app_cls_name = app_lists.pop(0)

       context_modules = map(lambda x: x.__module__,
                            self.contexts_cls.values())
       if app_cls_name in context_modules:
       continue

       LOG.info('loading app %s', app_cls_name)

       cls = self.load_app(app_cls_name)
       if cls is None:
       continue

       self.applications_cls[app_cls_name] = cls

       services = []
       for key, context_cls in cls.context_iteritems():
       v = self.contexts_cls.setdefault(key, context_cls)
       assert v == context_cls
       context_modules.append(context_cls.__module__)

       if issubclass(context_cls, RyuApp):
              services.extend(get_dependent_services(context_cls))

       # we can't load an app that will be initiataed for
       # contexts.
       for i in get_dependent_services(cls):
       if i not in context_modules:
              services.append(i)
       if services:
       app_lists.extend([s for s in set(services)
                            if s not in app_lists])

def create_contexts(self):
for key, cls in self.contexts_cls.items():
       if issubclass(cls, RyuApp):
       # hack for dpset
       context = self._instantiate(None, cls)
       else:
       context = cls()
       LOG.info('creating context %s', key)
       assert key not in self.contexts
       self.contexts[key] = context
return self.contexts
```

```
def _update_bricks(self):
for i in SERVICE_BRICKS.values():
       for _k, m in inspect.getmembers(i, inspect.ismethod):
       if not hasattr(m, 'callers'):
              continue
       for ev_cls, c in m.callers.iteritems():
              if not c.ev_source:
              continue

              brick = _lookup_service_brick_by_mod_name(c.ev_source)
              if brick:
              brick.register_observer(ev_cls, i.name,
                                           c.dispatchers)

              # allow RyuApp and Event class are in different module
              for brick in SERVICE_BRICKS.itervalues():
              if ev_cls in brick._EVENTS:
                     brick.register_observer(ev_cls, i.name,
                                           c.dispatchers)

@staticmethod
def _report_brick(name, app):
LOG.debug("BRICK %s", name)
for ev_cls, list_ in app.observers.items():
       LOG.debug("  PROVIDES %s TO %s", ev_cls.__name__, list_)
for ev_cls in app.event_handlers.keys():
       LOG.debug("  CONSUMES %s", ev_cls.__name__)

@staticmethod
def report_bricks():
for brick, i in SERVICE_BRICKS.items():
       AppManager._report_brick(brick, i)

def _instantiate(self, app_name, cls, *args, **kwargs):
# for now, only single instance of a given module
# Do we need to support multiple instances?
# Yes, maybe for slicing.
LOG.info('instantiating app %s of %s', app_name, cls.__name__)

if hasattr(cls, 'OFP_VERSIONS') and cls.OFP_VERSIONS is not None:
       ofproto_protocol.set_app_supported_versions(cls.OFP_VERSIONS)

if app_name is not None:
       assert app_name not in self.applications
app = cls(*args, **kwargs)
register_app(app)
assert app.name not in self.applications
self.applications[app.name] = app
return app

def instantiate(self, cls, *args, **kwargs):
app = self._instantiate(None, cls, *args, **kwargs)
self._update_bricks()
self._report_brick(app.name, app)
return app

def instantiate_apps(self, *args, **kwargs):
```

```
for app_name, cls in self.applications_cls.items():
       self._instantiate(app_name, cls, *args, **kwargs)

self._update_bricks()
self.report_bricks()

threads = []
for app in self.applications.values():
       t = app.start()
       if t is not None:
       threads.append(t)
return threads

@staticmethod
def _close(app):
close_method = getattr(app, 'close', None)
if callable(close_method):
       close_method()

def uninstantiate(self, name):
app = self.applications.pop(name)
unregister_app(app)
for app_ in SERVICE_BRICKS.values():
       app_.unregister_observer_all_event(name)
app.stop()
self._close(app)
events = app.events
if not events.empty():
       app.logger.debug('%s events remians %d', app.name, events.qsize())

def close(self):
def close_all(close_dict):
       for app in close_dict.values():
       self._close(app)
       close_dict.clear()

for app_name in list(self.applications.keys()):
       self.uninstantiate(app_name)
assert not self.applications
close_all(self.contexts)
```