Visvesvaraya Technological University

A Project Report
on

Application Development in SDN - Reduction of Broadcast Traffic in Data Centers

submitted in partial fulfillment of the requirements for the award of the degree of
Bachelor of Engineering
in
Computer Science & Engineering

by

G. Vijay Teja - 1PI10CS038
Karishma Sureka - 1PI10CS043
Sai Gopal - 1PI10CS075

under the guidance of
Dr. Ram P. Rustagi

January 2014 - May 2014

Department of Computer Science & Engineering
PES Institute of Technology
(an autonomous institute under VTU)
100 FEET RING ROAD, BANASHANKARI III STAGE
BANGALORE - 560085

PES Institute of Technology

(an autonomous institute under VTU)

100 FEET RING ROAD, BANASHANKARI III STAGE
BANGALORE - 560085

Department of Computer Science & Engineering

<u>CERTIFICATE</u>

Certified that the project work entitled **"Application Development in SDN - Reduction of Broadcast Traffic in Data Centers"** is bonafide work carried out by

G. Vijay Teja, 1PI10CS038
Karishma Sureka, 1PI10CS043
Sai Gopal, 1PI10CS075

in partial fulfillment for the award of degree of Bachelor of Engineering in Computer Science & Engineering at PES Institute of Technology (an autonomous institute under VTU, Belgaum) during the academic semester January 2014 - May 2014. It is certified that all corrections and suggestions indicated for internal assessment have been incorporated in the report deposited in the departmental library. The project report has been approved as it satisfies the academic requirements in respect of the project work prescribed for the said Bachelor of Engineering degree.

Signature of the Guide	Signature of the HOD	Signature of the Principal
Dr. Ram P. Rustagi	Prof. Nitin V Pujari	Dr. K N B Murthy

External Viva

Name of the examiners	Signature with date
1.	....
2.	....

Acknowledgement

We would like to acknowledge and commend the people who have contributed in the different aspects of this project for their effort, cooperation and collaboration in spite of the challenges and predicaments that we were all faced with.

We are deeply grateful to Prof. M. R. Doreswamy, PESIT founder, Prof. D. Jawahar, our CEO and Dr. K. N. Balasubramanya Murthy, our Principal, for providing us with a convivial atmosphere for carrying out the project. We owe a debt of gratitude to Prof. Nitin V. Pujari, Head of the Department, Computer Science, PESIT, for believing in this project.

We thank our panel members for their valuable suggestions and feedback.

We are extremely grateful to Dr. Ram P. Rustagi, Head of the Networking Domain, Dept. of Information Science, PESIT, our project guide, for entrusting us with the project and for his unflagging effort. His constant aid and foresight helped us realise this project. His unwavering support and persistence kept us motivated throughout the project.

We are also indebted to Ms. Veena, Associate Professor, Dept. of MCA, PESIT, for her input during the weekly project discussions.

Warm thanks to our parents who always kept us motivated and to our friends for their valuable assistance.

G. Vijay Teja
Karishma Sureka
Sai Gopal

Contents

List of Figures

Abstract

It is common for user applications to span thousands of servers, where a single user search request might access an inverted index spread over 1k+ servers. Analytics involves constant querying of the data stored in these servers. In addition, Data Centers are now converging and becoming multitenanted. These use cases indicate a high amount of traffic within Data Centers - on a scale comparable to the total Internet communication. With such a large scale, the efficiency of the communication within data centers, becomes very important. Data centers have new needs due to emerging trends in computing like cloud services, Big Data etc. The existing data center networks are not able to meet these new requirements elegantly.

The data center designer primarily has to make a choice between either using Layer 2 (L2) network for the data center or Layer 3 (L3) network. Both of them have their advantages and disadvantages. Using L2 network would give more performance in terms of lower latency and more throughput in terms of pushing packets through the network, have zero configuration overheads and also allow full mobility of virtual machines within the network without disrupting existing connections. On the other hand L2 network would cause the broadcast traffic due to unknown unicast to increase exponentially with the size of the network and also would require running of spanning tree protocol to prevent broadcast storms when there are loops in the network for redundancy, which would cause the redundant links to be disabled and wasted. L3 networks do not suffer from the drawbacks of L2 networks - there is no broadcast traffic since routes are aggregated and routing supports load balancing between redundant links in the form of extensions like ECMP in OSPF. But they are slower, have configuration overheads and physically limit the mobility of virtual machines to the subnet they belong to.

Parallely, recognizing the changing requirements of networks and the inability of traditional network architectures to support these requirements, a new architecture for network was developed - Software Defined Networking has emerged as an exciting area of research which is challenging the current networking paradigm of designing and managing computer networks. SDN has the potential to revolutionize not only the way networks are built but can lead to emergence of new applications that were not hitherto possible.

This project aims to achieve the best of both L2 and L3 networks by using an L2 network and removing its drawbacks, namely, broadcast traffic and running of spanning tree protocol by using this new architecture. Due to this architecture, one entity(the controller) has the entire knowledge of the network. We leverage this knowledge and partition the MAC address space hierarchically and assign hierarchical pseudo MAC addresses to hosts based on their position within the network thereby solving the unknown unicast broadcast problem. This central knowledge is also used to define broadcast semantics which facilitate efficient broadcasts even with loops in the network and avoid running of spanning tree protocol.

1. Introduction

1.1 Introduction to the domain

This project comes under the realm of Software Defined Networking (SDN) in the Networking Domain. This is a relatively new field that has gained traction in the last couple of years and is still in development.

"Software-Defined Networking (SDN) is an emerging architecture that is dynamic, manageable, cost-effective, and adaptable, making it ideal for the high-bandwidth, dynamic nature of today's applications. This architecture decouples the network control and forwarding functions enabling the network control to become directly programmable and the underlying infrastructure to be abstracted for applications and network services."

Key computing trends such as the Changing traffic patterns, the "consumerization of IT", the rise of cloud services and "Big data" require more bandwidth and led to the evolution of SDN. For an enterprise, SDN can facilitate the hybrid cloud paradigm, allowing businesses to use public cloud services to manage peak enterprise workloads. In addition, with the advent of big data, network data mining that collects information about real-time behavior of applications running on the network can be an SDN application of particular interest.

This project involves application development in SDN for Data Centers to enhance their performance by the reduction of Layer 2 Broadcast traffic. It also supports Data Center requirements like redundancy and load balancing within the internal network as well as emerging requirements like Virtual IP and Virtual Machine Migration.

1.2 Challenges and Opportunities

OpenFlow-based SDN is currently being rolled out in a variety of networking devices and software, delivering substantial benefits to both enterprises and carriers, including:

1. Centralized management and control of networking devices from multiple vendors
2. Improved automation and management by using common APIs to abstract the underlying networking details from the orchestration and provisioning systems and applications
3. Rapid innovation through the ability to deliver new network capabilities and services without the need to configure individual devices or wait for vendor releases

4. Programmability by operators, enterprises, independent software vendors, and users (not just equipment manufacturers) using common programming environments, which gives all parties new opportunities to drive revenue and differentiation
5. Increased network reliability and security as a result of centralized and automated management of network devices, uniform policy enforcement, and fewer configuration errors
6. More granular network control with the ability to apply comprehensive and wide-ranging policies at the session, user, device, and application levels
7. Better end-user experience as applications exploit centralized network state information to seamlessly adapt network behavior to user needs.

Therefore, SDN holds a tremendous amount of promise to change the status quo in networking. However, this is an emerging field and analytical support for the same level of visibility as in traditional networks still needs attention which is crucial for troubleshooting and visualization.

1.3 Identify Problems

It is common for user applications to span thousands of servers, where a single user search request might access an inverted index spread over 1k+ servers. Analytics involves constant querying of the data stored in these servers. In addition, Data Centers are now converging and becoming multitenanted. These use cases indicate a high amount of internet traffic within Data Centers - a fair percentage of the total Internet Communication. Hence, any kind of mechanism that can reduce the internal traffic like broadcast traffic, can lead to a lot of performance enhancement for Data Centers and improve the throughput for user applications.

Virtual Machine Migration and Virtual IP are becoming increasingly important for supporting business needs in a responsive manner, consequently new Data Center designs need to support the same. For VM Migration, migration within the same Layer 2 network is flexible as the same IP can be retained for communication. Also, Layer 2 networks have no configuration overheads like Layer 3, making them less prone to errors. These are some of the reasons for building Data Centers with large Layer 2 networks. However, the downside is the generation of a lot of broadcast traffic in the network which poses scalability issues.

Data Centers also have built-in redundancy for link failures as well as higher transmission bandwidth. Supporting this in Layer 2 networks becomes challenging since switches rely on the spanning tree protocol to prevent broadcast floods and storms. Switch fail-overs and security are also major concerns.

Application development in SDN can be used to synthesize a new design for the control in Data Centers to enhance the performance and address all the above requirements efficiently.

1.4 Problems to which we are providing solution

An SDN Application that can cater to the following requirements in Data Centers -

1. Reduction of the internal broadcast traffic : both switch-level broadcast and subnet-level broadcast, for improvement in performance.

2. The new design should continue to support VM migration and Virtual IP addressing.

3. Efficient utilization of the Data Center's built-in redundancy in its topology for addressing link failures and for supporting high bandwidth transmissions. This is not possible if spanning tree is used.

4. In the absence of a spanning tree algorithm in a Layer 2 network, broadcast semantics have to be defined to ensure loop free forwarding.

Above all, the application should run without effecting the user applications as well as the conventional networking protocols being used for communication in the network and without changing the physical topology of Data Centers.

2. Problem Definition

Most Data Centers are based on a multi-rooted 3-tier topology with built-in redundancy. It is preferable to leverage large Layer 2 (L2) networks inside Data Centers, over Layer 3 (L3), due to a number of reasons based on the requirements in Data Centers. To understand this, it is necessary to evaluate the benefits of Layer 2 networks in Data Centers against those of Layer 3:

The L2 network switches have no configuration overheads - the switch is self learning, plug and play. VM Migration within the same L2 network will ensure that the established connection can be maintained and the host can continue using the same IP address even after migration. For greater flexibility in VM Migration, it is desirable to have large L2 networks within Data Centers. On the contrary, unlike L3, L2 networks generate a lot of broadcast traffic. Also, the topology of Data Centers is designed to ensure easy fail-over and support load distribution among the links by introducing redundant links. The presence of such physical loops can cause problems of flooding/storms in case of a broadcast in L2 networks and switches would need to run a spanning tree protocol to counter it. L3 networks use IP addressing, a hierarchical approach and can leverage the built-in redundancy in Data Centers due the the TTL field in the packet header. There is no broadcast traffic like L2, but there are configuration overheads:

1. New hosts to be configured with the address of the Gateway Router for the subnet it belongs to.
2. The addition of each new Router involves setting its subnet information.
3. DHCP servers have to be synchronized to distribute IP addresses based on the host's subnet.

These tasks are overseen by network administrators leading to a lot of difficulty in management. More importantly, VM Migration across L3 networks is not desirable.

Also, L2 networks have better packet processing efficiency than L3 networks, for several reasons :

1. Switches don't have to modify packets - only lookup and forward. But routers have to modify L2 frames and update it with its own MAC address.
2. Routers have to decapsulate one more extra layer - L3 and also modify it, mainly the TTL field, which the switches don't have to. This also adds an overhead computation because the IP checksum has to be recomputed due to the modification of the packet.

This kind of switch efficiency is desirable in Data Centers as low latency is of very high priority, but broadcast traffic remains as one of the biggest downsides of using L2 network. Hence, an ideal system would involve the benefits of both - structured Layer 3 addressing which doesn't use broadcast as well as the simple Layer 2 forwarding mechanism with no configuration overheads and easy VM migration.

The project's main deliverable is to design and implement a system for the reduction of L2 broadcast traffic in Data Centers based on SDN, leading to an improved performance. This will also ensure better scalability on leveraging large L2 networks in Data Centers. The project also makes use of the in-built redundancy in Data Centers, with support for multi-path loadbalancing as well as supports VM Migration, Virtual IP, ARP Probe and Gratuitous ARP packets sent from the hosts.

PROJECT SCOPE: This system will be based on Data Centers which make use of Software Defined Networking to separate out the control and data planes from forwarding devices (switch, hub, router, etc.) and communicate in compliance with the OpenFlow protocol. This system will only apply to Data Centers incorporating a multi-rooted, 3-tier topology and using a single controller based software defined network. We restrict ourselves to only IPV4 networks.

3. Literature Survey

3.1 Mininet

(*A Network Emulator*)

Mininet [7] is a tool for rapid prototyping of SDN. It creates a realistic virtual network with real working components, but runs on a single machine for ease of testing. It provides an ability to creates hosts, switches and controller via - Command line, Interactive user interface and python application.

A Mininet host behaves just like a real machine; you can ssh into it (if you start up sshd and bridge the network to your host) and run arbitrary programs (including anything that is installed on the underlying Linux system.) The programs you run can send packets through what seems like a real Ethernet interface, with a given link speed and delay.

Network Namespaces - Nearly every operating system virtualizes computing resources using a process abstraction. Mininet uses **process-based virtualization** to run many hosts and switches on a single OS kernel. Since version 2.2.26, Linux has supported network namespaces, a lightweight virtualization feature that provides individual processes with separate network interfaces, routing tables, and ARP tables. Mininet can create kernel or user-space OpenFlow switches, controllers to control the switches, and hosts to communicate over the simulated network. Mininet connects switches and hosts using **virtual ethernet (veth) pairs** . While Mininet currently depends on the Linux kernel, in the future it may support other operating systems with process-based virtualization, such Solaris containers or !FreeBSD jails.

Mininet includes a command-line interface (CLI) which may be invoked on a network, and provides a variety of useful commands, as well as the ability to display xterm windows and to run commands on individual nodes in the network.

Mininet's API, including classes such as Topo, Mininet, Host, Switch, Link and their subclasses. It is convenient to divide these classes into levels (or layers), since in general the high-level APIs are built using the lower-level APIs.
Mininet's API is built at three primary levels:

1. Low-level API: The low-level API consists of the base node and link classes (such as Host, Switch, and Link and their subclasses) which can actually be instantiated individually and

used to create a network, but it is a bit unwieldy.

2. Mid-level API: The mid-level API adds the Mininet object which serves as a container for nodes and links. It provides a number of methods (such as addHost(), addSwitch(), and addLink()) for adding nodes and links to a network, as well as network configuration, startup and shutdown (notably start() and stop().)

3. High-level API: The high-level API adds a topology template abstraction, the Topo class, which provides the ability to create reusable, parameterized topology templates. These templates can be passed to the mn command (via the –custom option) and used from the command line.

Limitations -

1. Mininet-based networks cannot (currently) exceed the CPU or bandwidth available on a single server.

2. Mininet cannot (currently) run non-Linux-compatible OpenFlow switches or applications.

3.2 OpenFlow

(*An enabler of SDN*)

OpenFlow [4] is the first standard communications interface defined between the control and forwarding layers of an SDN architecture. OpenFlow allows direct access to and manipulation of the forwarding plane of network devices such as switches and routers, both physical and virtual (hypervisor-based).

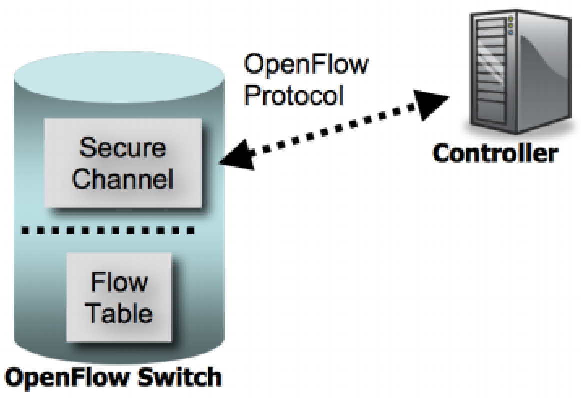


Figure 3.1: A OpenFlow switch communicates with the Controller over a secure TLS connection using the OpenFlow protocol.

The OpenFlow protocol is implemented on both sides of the interface between network infrastructure devices and the SDN control software. OpenFlow uses the concept of flows to identify network traffic based on pre-defined match rules that can be statically or dynamically programmed by the SDN control software. It also allows IT to define how traffic should flow through network devices based on parameters such as usage patterns, applications, and cloud resources. Since OpenFlow allows the network to be programmed on a per-flow basis, an OpenFlow-based SDN architecture provides extremely granular control, enabling the network to respond to real-time changes at the application, user, and session levels. Current IP-based routing does not provide this level of control, as all flows between two endpoints must follow the same path through the network, regardless of their different requirements.

3.2.1 Overview

Controller-to-Switch

Controller/switch messages are initiated by the controller and may or may not require a response from the switch.
Features: Upon Transport Layer Security (TLS) session establishment, the controller sends a features request message to the switch. The switch must reply with a features reply that species the capabilities supported by the switch.
Configuration: The controller is able to set and query configuration parameters in the switch. The switch only responds to a query from the controller.
Modify-State: Modify-State messages are sent by the controller to manage state on the switches. Their primary purpose is to add/delete and modify rows in the flow tables and to set switch port properties.
Read-State: Read-State messages are used by the controller to collect statistics from the switches, flow-tables, ports and the individual flow entries. Send-Packet: These are used by the controller to send packets out of a specified port on the switch.
Barrier: Barrier request/reply messages are used by the controller to ensure message dependencies have been met or to receive notifications for completed operations.

Asynchronous Messages

Asynchronous messages are sent without the controller soliciting them from a switch. Switches send asynchronous messages to the controller to denote a packet arrival, switch state change, or error. The four main asynchronous message types are - **Packet-in, Flow-Removed, Port-Status, Error**

Symmetric Messages

Symmetric messages are sent without solicitation, in either direction. **Hello, Echo, Vendor** are some examples of Symmetric messages that can be changed between the Controller and the switches.

3.2.2 Packet Processing in Flow Tables

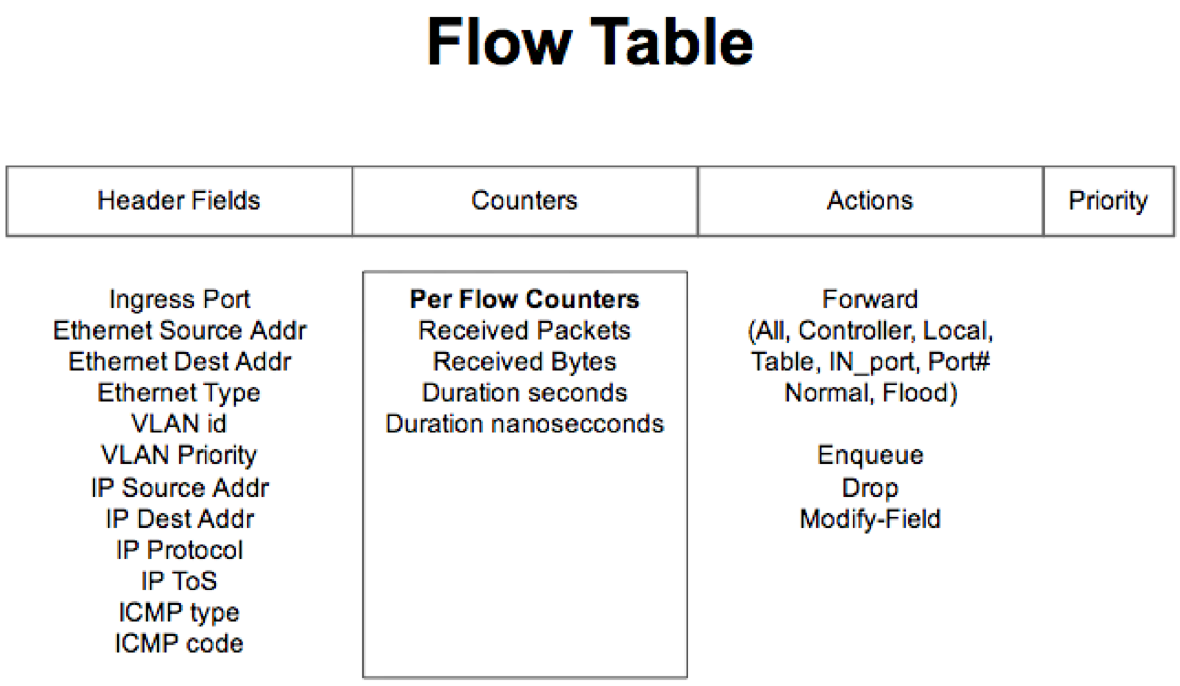


Figure 3.2: OpenFlow Instruction Set

Each flow table entry contains - (Refer figure 3.2)

1. **Header fields** - to match against packets

2. **Counters** - to update for matching packet. Counters are maintained per-table, per-flow, per-port and per-queue. Duration refers to the time the flow has been installed in the switch. The Receive errors filed includes all explicitly specified errors.

3. **Actions** - to apply to matching packets. Each flow entry is associated with zero or more actions that dictate how the switch handles matching packets. If no forward actions are present, then the packet is dropped. Action lists for inserted flow entries must be processed in the order specified. However, there is no packet output ordering guaranteed within a port.

4. **Priority** - The priority field is only relevant for flow entries with wildcard fields. The priority field indicates table priority, where higher numbers are higher priorities; the switch must keep the highest-priority wildcard entries in the lowest-numbered (fastest) wildcard table, to ensure correctness. It is the responsibility of each switch implementer to ensure that exact entries always match before wildcards entries, regardless of the table configuration.

When a packet arrives at the switch, the packet's headers are matched against the Header fields in the flow table entry. There could be multiple matches in the flow table, like in the case of wildcards. In that case, the entry with the highest priority is matched. If a matching entry is found, any actions for that entry are performed on the packet (e.g., the action might be to forward a packet out a specified port). If multiple actions are present, then they are executed in order. If no match is found, the packet is forwarded to the controller over the secure channel. The controller is responsible for determining how to handle packets without valid flow entries, and it manages the switch flow table by adding and removing flow entries.

3.2.3 Topology Discovery in SDN

Discovery in Software-defined Networking includes [9] :

1. **Discovery of Switches** :
   An OpenVSwitch is initially configured with a master controller IP address. Upon Transport Layer Security (TLS) session establishment, the Controller sends a feature request message to the switch and waits for a response. When the reply reaches to the controller, controller gets informed about the features provided by the switch, for instance, the datapath ID (i.e., DPID), list of ports, etc.

2. **Discovery of Links** :
   When a switch connects to a controller, controller periodically (e.g., every 5 seconds) commands the switch to flood LLDP (Link Layer Discovery Protocol) and BDDP (Broadcast Domain Discovery Protocol) packets through all of its ports. A discovery protocol packet typically contains the DPID of the sender along with the port of the switch that the message originates from. The reserved set of destination MAC addresses and ethertypes used by the discovery protocol packets lets the controller to differentiate them with the other data packets. LLDP is used to discover direct links between switches and BDDP is used to discover the switches in the same broadcast domain. LLDP and BDDP differ from each other by the ethertypes they use.

3. **Discovery of Hosts** :
   As new hosts join the network and send packets, the lack of corresponding flow table entries in the adjoining switches causes the packet to be forwarded to the controller. The Controller then discovers the host and the port of the switch that the host connects to. If the Controller doesn't have a mapping of the destination host as well, it might cause switches to flood the packets or any other custom action as defined in the Controller.

3.2.4 Nicira / Open vSwitch Extensions

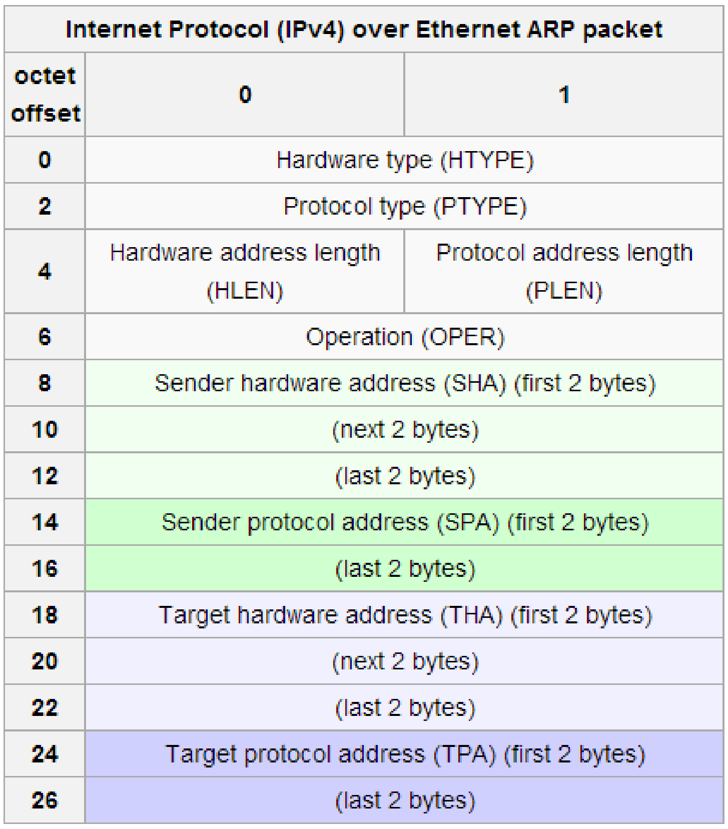
Open vSwitch supports a number of extensions to OpenFlow 1.0, and POX (Python-based Controller used in this project) has growing support for these through the *openflow.nicira* module. Some of the additional features provided by the nicira extensions include-

1. **Extended PacketIn Messages** - OVS has an extended version of the packet-in message which contains the reason for the packet-in (e.g., whether it was because of a send-to-controller action or a table miss) and in the former case, the match of the relevant table entry. This extended version is encapsulated inside an OpenFlow vendor message, and can be read via the generic vendor message hook mechanism or by handling the vendor event.

2. **Multiple Table Support** - While OpenFlow 1.0 only supports a single table, the Nicira extensions add support for multiple tables.

3.3 Address Resolution Protocol (ARP)

The Address Resolution Protocol is a request and reply protocol that runs encapsulated by the line protocol. It is communicated within the boundaries of a single network, never routed across internetwork nodes. This property places ARP into the Link Layer of the Internet Protocol Suite.



Internet Protocol (IPv4) over Ethernet ARP packet		
octet offset	**0**	**1**
0	Hardware type (HTYPE)	
2	Protocol type (PTYPE)	
4	Hardware address length (HLEN)	Protocol address length (PLEN)
6	Operation (OPER)	
8	Sender hardware address (SHA) (first 2 bytes)	
10	(next 2 bytes)	
12	(last 2 bytes)	
14	Sender protocol address (SPA) (first 2 bytes)	
16	(last 2 bytes)	
18	Target hardware address (THA) (first 2 bytes)	
20	(next 2 bytes)	
22	(last 2 bytes)	
24	Target protocol address (TPA) (first 2 bytes)	
26	(last 2 bytes)	

Figure 3.3: ARP Packet Structure

The Address Resolution Protocol uses a simple message format that contains one address resolution request or response. The size of the ARP message depends on the upper layer and lower layer address sizes, which are given by the type of networking protocol (usually IPv4) in use and the type of hardware or virtual link layer that the upper layer protocol is running on. The message header specifies these types, as well as the size of addresses of each. The message header is completed with the operation code for request (1) and reply (2). The payload of the packet consists of four addresses, the hardware and protocol address of the sender and receiver hosts.
ARP Probe - Before beginning to use an IPV4 address, a host implementing ARP Probe tests to see if this address is already in use by sending an ARP Probe. An ARP Probe is an ARP request with the target IP as the IP it wants to use and an all-zero sender IP address. The term is used in the IPv4 Address Conflict Detection specification (RFC 5227).

Gratuitous ARP - This is a case of ARP being used as an announcement protocol. A Gratuitous ARP message is usually broadcast as an ARP request containing the sender's protocol address (SPA) in the target field (TPA=SPA), with the target hardware address (THA) set to zero. An alternative is to broadcast an ARP reply with the sender's hardware and protocol addresses (SHA and SPA) duplicated in the target fields (TPA=SPA, THA=SHA). An ARP announcement is not intended to solicit a reply; instead it updates any cached entries in the ARP tables of other hosts that receive the packet. Many operating systems perform gratuitous ARP during startup. That helps to resolve problems which would otherwise occur if, for example, a network card was recently changed (changing the IP-address-to-MAC-address mapping) and other hosts still have the old mapping in their ARP caches.

3.4 PortLand

(*A scalable, fault tolerant, layer-2 data center network fabric.*)

PortLand [1] is a system that was designed and proposed in the Sigcomm'09 conference for the reduction of layer-2 broadcast traffic in Data Centers using k-ary Fat Tree topology (3-tier k-pod Fat Tree topology with k-port switches). The PortLand design caters to the requirements of the Data Centers by redesigning the Layer 2 network -

1. It assigns a *pseudo MAC* address to all the hosts in the network. This pseudo MAC, as opposed to the actual MAC is assigned hierarchically based on the host's position in the Data Center network - by the switch that it is directly connected to(edge switch).

2. The format of the 48-bit pseudo MAC is | *pod-num-16* | *switch-pos-8* | *port-8* | *vmid-16* |

3. The maintenance of this assignment mapping and translation of pseudo MAC to actual MAC and vice versa is done at the edge switches. Within the network, the actual MACs are completely abstracted using this translation and the whole system is transparent.

4. It the design addresses both the kinds of broadcast:
   (a) This hierarchical addressing is leveraged to eliminate the switch level broadcast since the switches always know where to forward the packet just by looking at the structure of pseudo MAC.
   (b) It also reduces the ARP broadcast traffic by the introduction of a logically central unit called Fabric Manager. The edge switches intercept any ARP packets and send it to the Fabric Manager which inturn updates its ARP table and processes the ARP packet. If its an ARP request and the manager has the *IP->pseudo MAC* mapping, the manager sends the pseudo MAC to the edge switch and the edge switch crafts an ARP reply and sends it back to the requesting host, thus avoiding a broadcast. If the *IP->pseudo MAC* is not present, then it resorts to a broadcast; when the other host replies to the ARP request, the Fabric Manager also gets a copy of the ARP reply and can update its ARP table.
5. Since this whole fabric is a Layer 2 network, VM migration requirement is met - moving the machine anywhere within the whole network wouldn't break any TCP connection as the whole thing is a part of the same subnet and IP can be retained.
6. Also, Layer 2 networks are plug and play with no configuration overheads like in Layer 3 networks. There are no forwarding loops since the forwarding is hierarchical in fashion. Layer 3 networks do not generate switch level broadcast traffic like in Layer 2 as the IP addressing is hierarchical; Routes are aggregated and in case a destination IP does not match any route, the packet is typically sent to a default route or is dropped unlike in switches where packet is flooded out of all interfaces.

PortLand mimics this benefit of Layer 3 networks by bringing in the concept of hierarchical PMACs. (pseudo MAC). We are implementing the PortLand concept using Openflow to explore its benefits, limitations and how it can be enhanced.

3.5 Reduction of ARP Broadcast in TOR switches

(*Proxy ARP*)

There is an internet draft which was submitted by *Shah, et al. in Oct.'11* on reducing ARP broadcast in Data Centers. The solution proposed is to have the first hop switch, typically the TOR switch in a traditional data center, to maintain a local ARP table by examining the ARP packets flowing through it and proxying ARP responses. When the switch receives an ARP request and the switch has a mapping for the requested destination IP in its ARP table, it creates a unicast ARP reply with that MAC and sends it to the requested host. This way, some amount of subnet level ARP broadcast is reduced. But it does not help in reducing the switch level broadcast or any other kind of subnet level broadcast and does not meet other requirements like VM migration.

3.6 Definitions

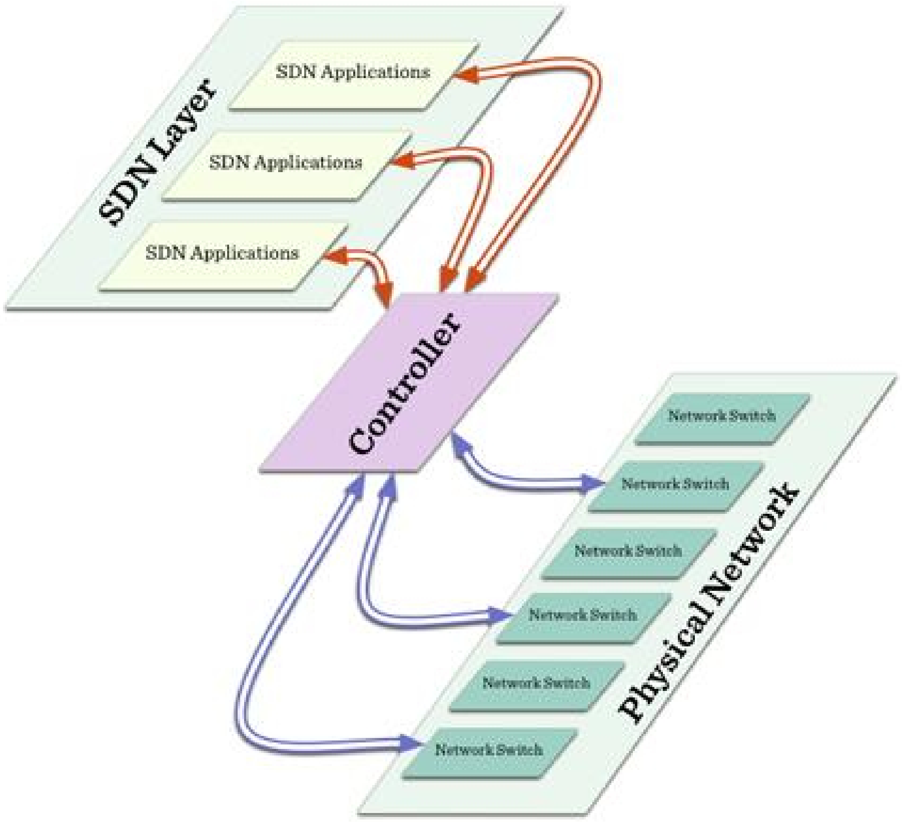


Figure 3.4: Software-Defined Network Architecture

1. **Control Plane -** The control plane of the switch/router is a part of the architecture that is concerned with drawing a networking map that defines the actions to take on the incoming data packets. Eg : population of routing table.

2. **Data Plane -** It is the part of the forwarding device that decides what to do with packets arriving on an inbound interface, typically by looking up in a table defined by the control plane. Eg : routing table.

3. **Software Defined Networking (SDN)-** A networking system where the control and the data planes are separated out from forwarding devices - The control resides outside the device in a specialized component called the *controller*, which is responsible for the

forwarding (data plane) performed by the switches. [3]

4. **Openflow -** It is a communications protocol used by a controller to access the data plane of a forwarding device. It is an enabler of SDN.

5. **Flow Table -** This is the abstraction of the data plane provided by an Openflow switch. Each flow table entry contains a set of packet fields to match, and a corresponding action(s) to take once a packet matches the fields.

6. **Controller -** An OpenFlow controller is an application that manages flow control in a software-defined networking (SDN) environment It is responsible for adding, modifying and deleting the entries in a flow table. Eg : POX (Python based controller)

7. **Broadcast -** There are two kinds of broadcast :
   - *Switch level broadcast* : This is the broadcast done by the switch when it does not have a mapping for the destination MAC address in its MAC table. The packet is output on all ports except the input port. This is known as unknown unicast.
   - *Subnet level broadcast* : This kind of broadcast is one where the packet has to reach all the hosts within the same IP subnet. Eg : ARP broadcast

8. **Virtual IP -** It is an IP address assigned to multiple applications residing on a single server, multiple domain names, or multiple servers, rather than being assigned to a specific single server or network interface card (NIC). Incoming data packets are sent to the VIP address which are routed to actual network interfaces. In the context of Data Centers, it is used for connection redundancy by providing alternative fail-over options on one machine; a VIP address may still be available if a computer or NIC fails, because an alternative computer or NIC replies to connections

9. **VM Migration -** Virtual Machine migration refers to the process of moving a running virtual machine or application between different physical machines without disconnecting the client or application. Memory, storage, and network connectivity of the virtual machine are transferred from the original host machine to the destination.

4. Problems Requirements Definition

4.1 Project Perspective

This product is an SDN application that runs as a module on top of a POX controller targeting networks in Data Centers to reduce broadcast traffic in Layer 2 networks. It is applicable to SDN based Data Centers, running a single-controller. The custom SDN application for POX can be run on any machine in the Data Center or even at a remote location as far as it is connected to the switches in the Data Center Network.

4.2 Project Functions

This project aims to reduce the Layer 2 level broadcast in SDN based Data centers. It eliminates switch-level broadcast - by defining a hierarchical addressing scheme for the internal MAC addresses and reduces the subnet-level ARP broadcast as well. The Controller intercepts the ARP packets in the network to help in the latter.
In addition to achieving a performance enhancement by the reduction of internal broadcast traffic, the custom module for POX also addresses Data Center requirements of VM Migration and Virtual IP. The module leverages the in-built redundancy of multi-rooted 3-tier Data Center Networks by defining custom broadcast semantics which support multi-path loadbalancing by hashing. This eliminates the need of running a Spanning Tree protocol within the Layer 2 network.

4.3 User Classes and Characteristics

The consumer of this POX module are network administrators in Data Centers. There isn't any specialised user characteristic involved as the module is plug and play, once set up and running, there is no intervention required.

4.4 Operating Environment

POX, the controller upon which our module runs on is written in python. Therefore, the primary operating environment is python interpreter. The operating environment should have the capability to run and support python interpreter. Apart from the support of python interpreter, it should have hardware which is proportional to the performance expected.
Mininet, the emulator used in this project requires an operating environment with a python interpreter like POX to support the custom data center topologies being emulated which are written in Python. It has to be run in a Linux environment since it leverages the linux namespaces to achieve a process based virtualization. Mininet also needs OpenvSwitch support which is essential for switches in the network to be able to communicate with POX using the OpenFlow protocol.

4.5 Assumptions and dependencies

The following are the assumptions made in this project -

1. During topology discovery, it is assumed that in Edge Switches, the proportion of the ports connected to physical hosts is greater than those connected to the Aggregate Switches.
2. This project is targeted towards Data Centers which are based on a multi-rooted 3-tier topology only.
3. It is assumed that a Virtual machine sends a Gratuitous ARP upon migration.
4. It is assumed that hosts send Gratuitous ARP upon virtual IP takeover.
5. Data Centers incorporating this application run a POX controller.
6. The network uses a single controller for software-defined networking.
7. The control plane is out of band.
8. Other applications running in parallel shouldn't use the host's actual MAC address for communication.
9. The VM software shouldn't run at the application level.
10. Hashing (for loadbalncing) also assumes that for each switch, the number of uplinks is less than or equal to the number of downlinks.

5. System Requirements Specification

5.1 Interface Requirements

5.1.1 User Interface

SDN application : - The interface for our software is provided through commandline. It is started as another module when starting POX and accepts the following parameters :

1. the max percentage of ports in an edge switch that are connected to other switches
2. the amount of time to wait in seconds before topology identification starts and
3. the maximum ARP cache timeout of any host.

It has no other means of interaction as it is meant to be started and left.

5.1.2 Software Interface

For our POX module:

- Operating system : any OS that supports Python interpreter
- Language : Python
- POX controller

For mininet:

- Operating system : any OS that runs on Linux
- Language : Python
- OpenVSwitch

5.2 Functional Requirements

1. **Network functionality** - The network has to be fully functional. All the network functions should work as they do on a traditional network.

2. **Transparency** - No changes should be required to either the hosts or the physical topology of the network. The behaviour of the hosts should not be altered and they should not be aware of any modifications being done by the software.

3. **Reduce broadcast traffic** - The software should completely eliminate unknown unicast and minimize ARP broadcast.

4. **Utilize redundant links** - Multiple redundant links should be fully utilized for load balancing without running spanning tree and disabling the links.

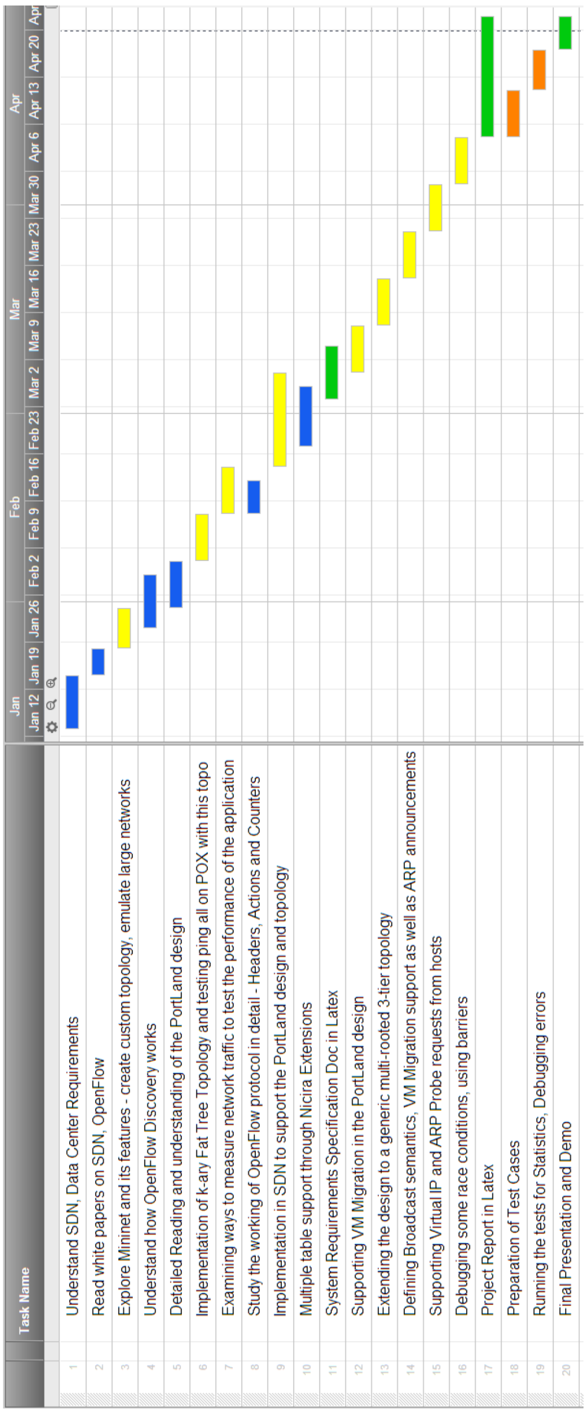
5. **Support VM migration and minimize VM downtime** - VM migration should be immediately identified and the network connectivity to the migrated position should be restored immediately.

5.3 Non-Functional Requirements

1. **Portability** - The controller module should be able to be easily ported onto a real network.

2. **Zero configuration** - Switches in the network should not have to be configured because of the software.

6. Gantt Chart



	Task Name
1	Understand SDN, Data Center Requirements
2	Read white papers on SDN, OpenFlow
3	Explore Mininet and its features - create custom topology, emulate large networks
4	Understand how OpenFlow Discovery works
5	Detailed Reading and understanding of the PortLand design
6	Implementation of k-ary Fat Tree Topology and testing ping all on POX with this topo
7	Examining ways to measure network traffic to test the performance of the application
8	Study the working of OpenFlow protocol in detail - Headers, Actions and Counters
9	Implementation in SDN to support the PortLand design and topology
10	Multiple table support through Nicira Extensions
11	System Requirements Specification Doc in Latex
12	Supporting VM Migration in the PortLand design
13	Extending the design to a generic multi-rooted 3-tier topology
14	Defining Broadcast semantics, VM Migration support as well as ARP announcements
15	Supporting Virtual IP and ARP Probe requests from hosts
16	Debugging some race conditions, using barriers
17	Project Report in Latex
18	Preparation of Test Cases
19	Running the tests for Statistics, Debugging errors
20	Final Presentation and Demo

Figure 6.1: Block Diagram

7. System Design

7.1 Block Diagram

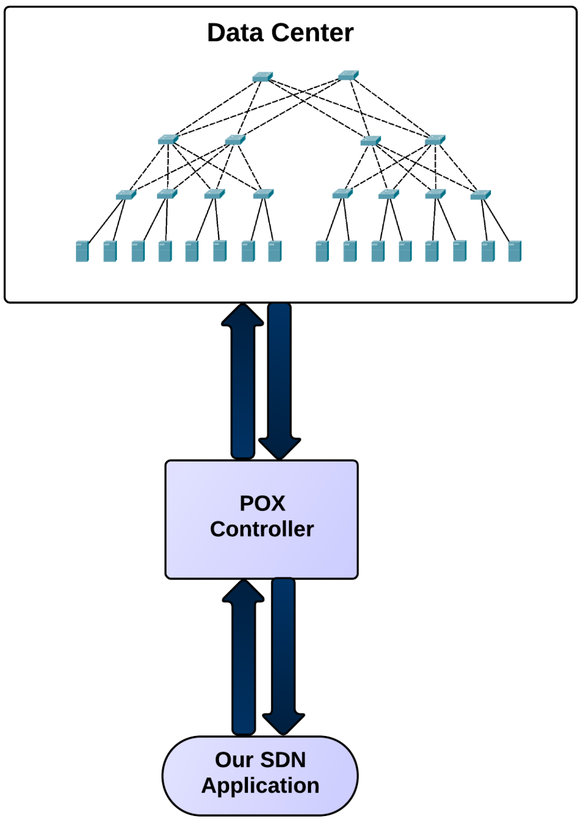


Figure 7.1: Block Diagram

The Data Center Network is a generic multi rooted 3-tier topology which is OpenFlow enabled. In our project, we have emulated the network by using Mininet, for development and testing purposes. The network switches communicate with the POX controller using OpenFlow protocol. Our application runs on top of the controller and is responsible for the normal functioning of the network which includes forwarding job, while reducing broadcast traffic and supporting other requirements like VM migration as mentioned before.

7.2 Architecture

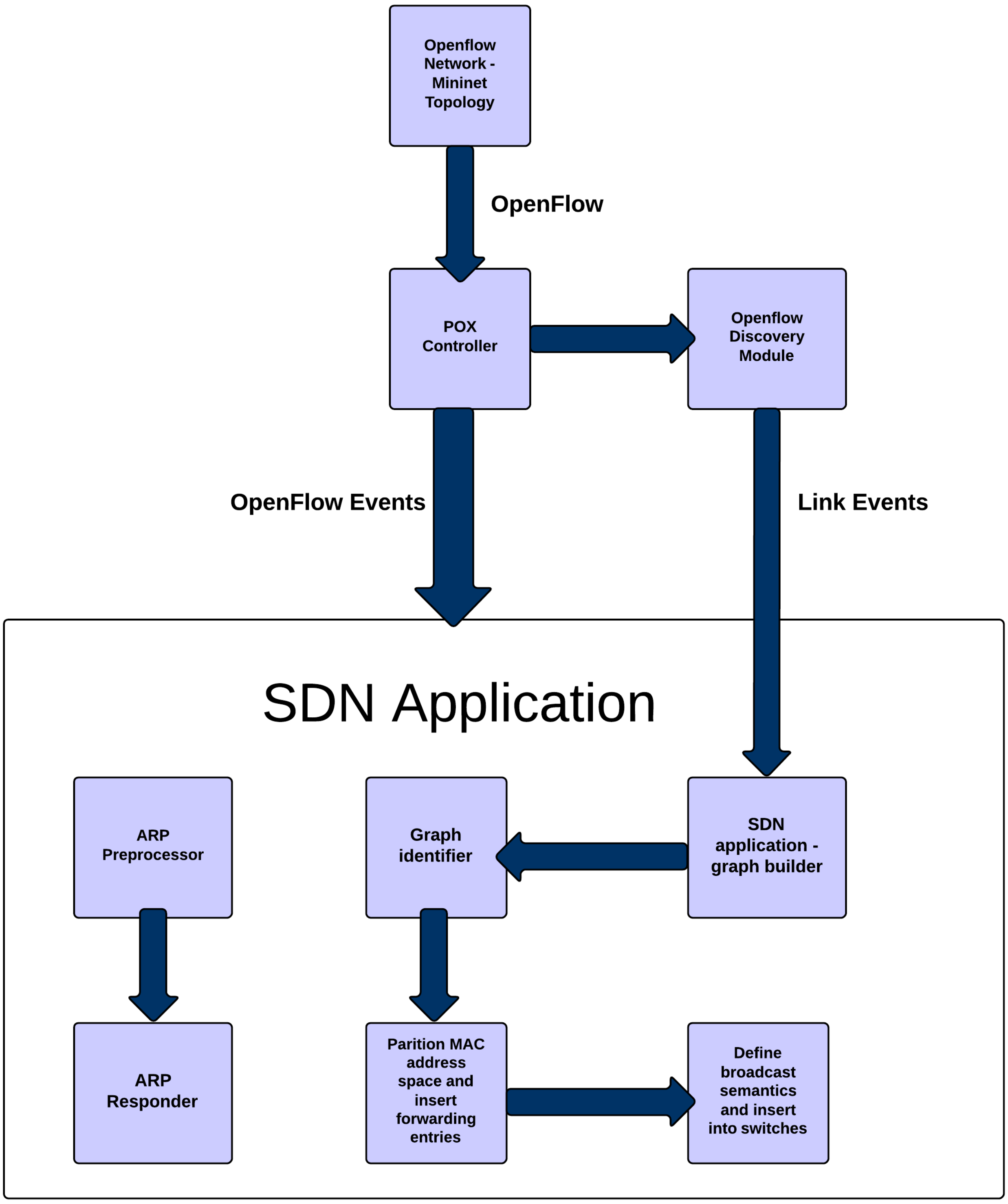

Openflow Network - Mininet Topology
OpenFlow
POX Controller
Openflow Discovery Module
OpenFlow Events
Link Events
SDN Application
ARP Preprocessor
Graph identifier
SDN application - graph builder
ARP Responder
Parition MAC address space and insert forwarding entries
Define broadcast semantics and insert into switches

Figure 7.2: Architecture Diagram

OpenFlow Network - Mininet Topology -
A custom multi-rooted 3-tier topology of Data Centers is emulated using Mininet.

POX Controller -
Once the OpenFlow compliant network is up and running, the POX Controller is run with the custom SDN application module. This is the component that is responsible for all the communication(through OpenFlow) with the network switches. It abstracts out the physical network and all communication that the application wants to do with the network is done through this component.

OpenFlow Discovery module -
This module detects the topology as well as changes in the topology and provides this info to other modules, like our application, in the form of Link events - link up and link down events.

ARP Preprocessor -
This is the part of the application which is responsible for first receiveing all the intercepted ARP packets and preprocessing them. Preprocessing involves detecting new hosts, VM migration, virtual IP takeover, inserting translation tables and maintaining internal state tables.

ARP Responder -
The module takes care of the ARP functioning of the SDN app. It takes care of requests, replies, gratituous ARP and ARP probes.

SDN Application Graph builder -
This part of the SDN application is responsible for building the whole network map of the datacenter network using the link events generated by the openflow.discovery module and connectionUp event generated by POX core module.

Graph Identifier -
This module interprets the graph that is built - it identifies the graph nodes as Edge, Aggregate and Core switches, classifies sets of switches into pods and assigning positions to switches.

Partition MAC address space and inserting forwarding tables -
Based on the analysis of the topology from the Graph Identifier, this module partitions the pseudo MAC address space between the different pods and different switches within the pod in a hierarchical manner. It also inserts forwarding entries into the flow tables of all the switches based on the hierarchy it just defined. These entries are static and they forward packets based on the destination PMAC. This type of forwarding eliminates switch-level broadcast.

Define Broadcast semantics and insert into switches -

The network has a lot of redundancy in-built. This module defines the broadcast semantics - what ports are connected upwards, what ports are connected downwards and uses this info to insert flow table entries, which can efficiently perform a broadcast even with loops in the network.

7.3 Use Case Diagrams and Scenarios Description

Here we describe some of the scenarios that can take place during communications within the network like broadcast, migration etc and how they are handled. We use **ping** as the communication method.
Known host in the context of this project is when the host has already communicated with other hosts in the past and hence, its identity is known to the Controller.
Here is the **base topology** that all our scenarios are based upon (a 3-tier Multi-rooted tree topology):

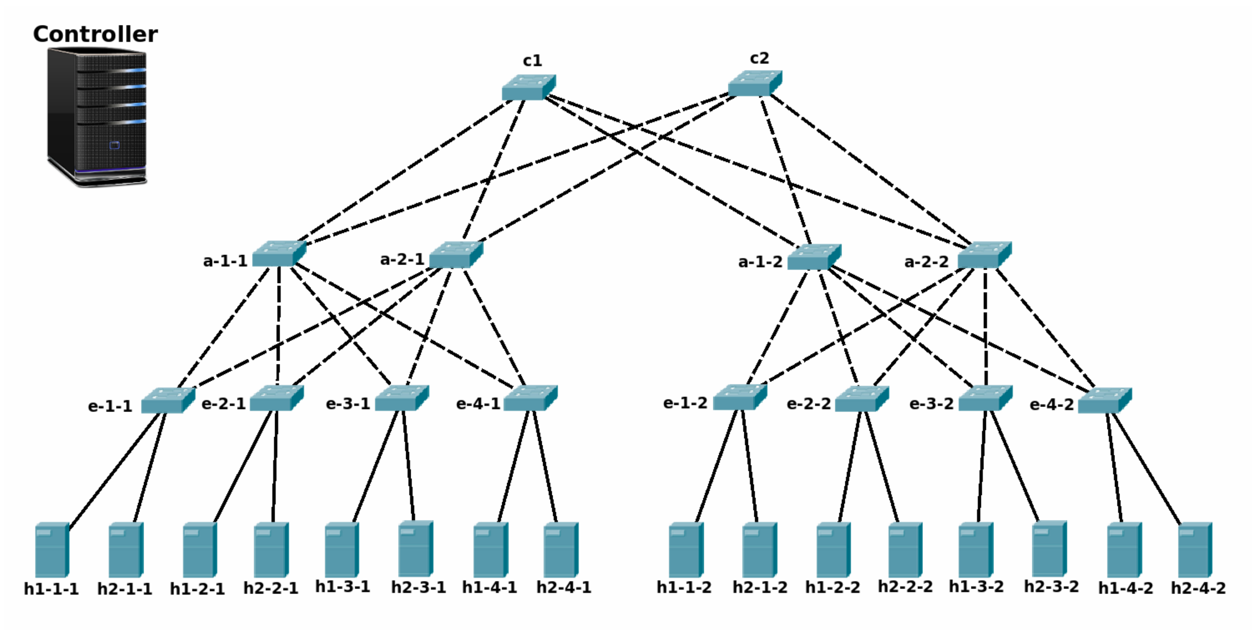


Figure 7.3: Network Topology

Here are the some of the flow tables of switches that we will be referring to in the later part. These were taken after the base topology above was setup and the forwarding entries have been inserted into the switch flow tables by the SDN app. A ping between host h1-1-1 and host h1-1-2 was performed before taking a snapshot of the tables. The flow table fields are :

- cookie : A cookie is an opaque identifier to handle a set of flows. The default value is 0. Since we are not using cookies, all the entries in the flow table are the default value zero.
- duration : This is the time that has elapsed, in seconds, from the time the flow table entry was added till now.
- table : this is the table number. It starts from 0. All the packet matches first start from table 0. We make use of multiple tables in edge switches and have 3 tables in all edge switches. Table 0 for special processing like sending ARP packets to controller and for translating actual src MAC to the assigned PMAC. Table 1 is used for translating dst PMAC to actual MAC and table 2 is used for forwarding.

- n_packets : This is a counter that counts the number of packets that have matched this flow table since its insertion.
- n_bytes : The sum in bytes, of size of each packet that has matched this entry so far.
- idle_age : The time elapsed in seconds since the last packet match to this entry.
- priority : in case of multiple matches in the flow table, like in wild card matches, the entry with highest priority field is matched. Higher number is higher priority.
- in_port : a part of the matching process - specifies the input port the packet should come from.
- dl_src : a part of the matching process - specifies the src MAC address the incoming packet should have. Can have bitmask wildcard to match set of MAC addresses
- dl_type : a part of the matching process - specifies the payload protocol that the incoming packet should match. Eg : 0x0806 for ARP
- dlldst : a part of the matching process - specifies the dst MAC address the incoming packet should have. Can have bitmask wildcard to match set of MAC addresses.
  Eg : dl_dst=00:01:00:00:00:00/ff:ff:00:00:00:00 would match all packets whose destinaiton MAC address have 1st 2 bytes as 0x0001. This works in a manner similar to netmasks in IP addresses.
- actions : a list of actions to perform once a packet is matched.
  We used the following actions:
  - output action sends the packet to the specified port(s).
  - mod_dl_src is used to rewrite the src MAC address with the value specified.
  - mod_dl_dst is used to rewrite the dst MAC address with the value specified.

Edge switch e-1-1's flow tables :

```
cookie=0x0, duration=2671.279s, table=0, n_packets=2, n_bytes=196, idle_age=14,
     priority=10 actions=resubmit(,1)
cookie=0x0, duration=14.455s, table=0, n_packets=2, n_bytes=196, idle_age=14,
    priority=5000,in_port=3,dl_src=46:d9:a2:6d:d5:94 actions=mod_dl_src
    :00:00:01:03:00:01,resubmit(,1)
cookie=0x0, duration=2691.132s, table=0, n_packets=330, n_bytes=13530, idle_age
    =1, priority=65000,dl_dst=01:23:20:00:00:01,dl_type=0x88cc actions=
    CONTROLLER:65535
cookie=0x0, duration=2671.279s, table=0, n_packets=1, n_bytes=42, idle_age=14,
    priority=9000,arp,in_port=3 actions=CONTROLLER:65535
cookie=0x0, duration=2671.279s, table=0, n_packets=0, n_bytes=0, idle_age=2671,
     priority=9000,arp,in_port=4 actions=CONTROLLER:65535
```

```
cookie=0x0, duration=2671.279s, table=1, n_packets=2, n_bytes=196, idle_age=14,
    priority=10 actions=resubmit(,2)
cookie=0x0, duration=14.454s, table=1, n_packets=2, n_bytes=196, idle_age=14,
   priority=5000,dl_dst=00:00:01:03:00:01 actions=mod_dl_dst:46:d9:a2:6d:d5:94,
   output:3
cookie=0x0, duration=2671.304s, table=2, n_packets=0, n_bytes=0, idle_age=2671,
    priority=2000,dl_dst=00:00:00:00:00:00/00:00:00:01:00:00 actions=output:1
cookie=0x0, duration=2671.304s, table=2, n_packets=2, n_bytes=196, idle_age=14,
    priority=2000,dl_dst=00:00:00:01:00:00/00:00:00:01:00:00 actions=output:2
cookie=0x0, duration=2671.255s, table=2, n_packets=0, n_bytes=0, idle_age=2671,
    priority=8500,in_port=4,dl_dst=ff:ff:ff:ff:ff:ff actions=output:3,output:2
cookie=0x0, duration=2671.255s, table=2, n_packets=0, n_bytes=0, idle_age=2671,
    priority=8500,in_port=3,dl_dst=ff:ff:ff:ff:ff:ff actions=output:4,output:1
cookie=0x0, duration=2671.255s, table=2, n_packets=0, n_bytes=0, idle_age=2671,
    priority=8500,in_port=2,dl_dst=ff:ff:ff:ff:ff:ff actions=output:3,output:4
cookie=0x0, duration=2671.255s, table=2, n_packets=0, n_bytes=0, idle_age=2671,
    priority=8500,in_port=1,dl_dst=ff:ff:ff:ff:ff:ff actions=output:3,output:4
cookie=0x0, duration=2671.304s, table=2, n_packets=0, n_bytes=0, idle_age=2671,
    priority=3000,dl_dst=00:00:01:03:00:00/ff:ff:ff:ff:00:00 actions=output:3
cookie=0x0, duration=2671.304s, table=2, n_packets=0, n_bytes=0, idle_age=2671,
    priority=3000,dl_dst=00:00:01:04:00:00/ff:ff:ff:ff:00:00 actions=output:4
```

Aggregate switch a-2-1's flow tables :

```
cookie=0x0, duration=2671.201s, table=0, n_packets=0, n_bytes=0, idle_age=2671,
    priority=2000,dl_dst=00:00:00:00:00:00/00:00:00:01:00:00 actions=output:1
cookie=0x0, duration=2671.201s, table=0, n_packets=2, n_bytes=196, idle_age=14,
    priority=2000,dl_dst=00:00:00:01:00:00/00:00:00:01:00:00 actions=output:2
cookie=0x0, duration=2671.153s, table=0, n_packets=0, n_bytes=0, idle_age=2671,
    priority=8500,in_port=5,dl_dst=ff:ff:ff:ff:ff:ff actions=output:3,output:4,
   output:6,output:1
cookie=0x0, duration=2671.153s, table=0, n_packets=0, n_bytes=0, idle_age=2671,
    priority=8500,in_port=6,dl_dst=ff:ff:ff:ff:ff:ff actions=output:3,output:4,
   output:5,output:2
cookie=0x0, duration=2671.153s, table=0, n_packets=0, n_bytes=0, idle_age=2671,
    priority=8500,in_port=4,dl_dst=ff:ff:ff:ff:ff:ff actions=output:3,output:5,
   output:6,output:2
cookie=0x0, duration=2671.153s, table=0, n_packets=0, n_bytes=0, idle_age=2671,
    priority=8500,in_port=3,dl_dst=ff:ff:ff:ff:ff:ff actions=output:4,output:5,
   output:6,output:1
cookie=0x0, duration=2671.153s, table=0, n_packets=0, n_bytes=0, idle_age=2671,
    priority=8500,in_port=2,dl_dst=ff:ff:ff:ff:ff:ff actions=output:3,output:4,
   output:5,output:6
cookie=0x0, duration=2671.153s, table=0, n_packets=0, n_bytes=0, idle_age=2671,
    priority=8500,in_port=1,dl_dst=ff:ff:ff:ff:ff:ff actions=output:3,output:4,
   output:5,output:6
cookie=0x0, duration=2671.201s, table=0, n_packets=0, n_bytes=0, idle_age=2671,
    priority=3000,dl_dst=00:00:03:00:00:00/ff:ff:ff:00:00:00 actions=output:5
cookie=0x0, duration=2671.201s, table=0, n_packets=2, n_bytes=196, idle_age=14,
    priority=3000,dl_dst=00:00:01:00:00:00/ff:ff:ff:00:00:00 actions=output:3
cookie=0x0, duration=2671.201s, table=0, n_packets=0, n_bytes=0, idle_age=2671,
    priority=3000,dl_dst=00:00:02:00:00:00/ff:ff:ff:00:00:00 actions=output:4
cookie=0x0, duration=2671.201s, table=0, n_packets=0, n_bytes=0, idle_age=2671,
    priority=3000,dl_dst=00:00:04:00:00:00/ff:ff:ff:00:00:00 actions=output:6
cookie=0x0, duration=2693.355s, table=0, n_packets=986, n_bytes=41086, idle_age
   =1, priority=65000,dl_dst=01:23:20:00:00:01,dl_type=0x88cc actions=
   CONTROLLER:65535
```

Core switch c2's flow tables :

```
cookie=0x0, duration=161.104s, table=0, n_packets=0, n_bytes=0, idle_age=161,
    priority=5000,in_port=4,dl_src=00:00:00:01:00:00/00:00:00:01:00:00,dl_dst=
    ff:ff:ff:ff:ff:ff actions=output:2
cookie=0x0, duration=161.104s, table=0, n_packets=0, n_bytes=0, idle_age=161,
    priority=5000,in_port=1,dl_src=00:00:00:01:00:00/00:00:00:01:00:00,dl_dst=
    ff:ff:ff:ff:ff:ff actions=output:4
cookie=0x0, duration=161.105s, table=0, n_packets=0, n_bytes=0, idle_age=161,
    priority=5000,in_port=1,dl_src=00:00:00:00:00:00/00:00:00:01:00:00,dl_dst=
    ff:ff:ff:ff:ff:ff actions=output:3
cookie=0x0, duration=161.104s, table=0, n_packets=0, n_bytes=0, idle_age=161,
    priority=5000,in_port=2,dl_src=00:00:00:00:00:00/00:00:00:01:00:00,dl_dst=
    ff:ff:ff:ff:ff:ff actions=output:3
cookie=0x0, duration=161.104s, table=0, n_packets=0, n_bytes=0, idle_age=161,
    priority=5000,in_port=2,dl_src=00:00:00:01:00:00/00:00:00:01:00:00,dl_dst=
    ff:ff:ff:ff:ff:ff actions=output:4
cookie=0x0, duration=161.104s, table=0, n_packets=0, n_bytes=0, idle_age=161,
    priority=5000,in_port=3,dl_src=00:00:00:00:00:00/00:00:00:01:00:00,dl_dst=
    ff:ff:ff:ff:ff:ff actions=output:1
cookie=0x0, duration=161.104s, table=0, n_packets=0, n_bytes=0, idle_age=161,
    priority=5000,in_port=3,dl_src=00:00:00:01:00:00/00:00:00:01:00:00,dl_dst=
    ff:ff:ff:ff:ff:ff actions=output:2
cookie=0x0, duration=161.104s, table=0, n_packets=0, n_bytes=0, idle_age=161,
    priority=5000,in_port=4,dl_src=00:00:00:00:00:00/00:00:00:01:00:00,dl_dst=
    ff:ff:ff:ff:ff:ff actions=output:1
cookie=0x0, duration=161.156s, table=0, n_packets=0, n_bytes=0, idle_age=161,
    priority=2000,dl_dst=00:00:00:00:00:00/ff:ff:00:01:00:00 actions=output:1
cookie=0x0, duration=161.156s, table=0, n_packets=0, n_bytes=0, idle_age=161,
    priority=2000,dl_dst=00:01:00:00:00:00/ff:ff:00:01:00:00 actions=output:3
```

```
cookie=0x0, duration=161.151s, table=0, n_packets=2, n_bytes=196, idle_age=106, priority=2000,dl_dst=00:01:00:01:00:00/ff:ff:00:01:00:00 actions=output:4
cookie=0x0, duration=161.156s, table=0, n_packets=2, n_bytes=196, idle_age=106, priority=2000,dl_dst=00:00:00:01:00:00/ff:ff:00:01:00:00 actions=output:2
cookie=0x0, duration=178.531s, table=0, n_packets=41, n_bytes=1681, idle_age=0, priority=65000,dl_dst=01:23:20:00:00:01,dl_type=0x88cc actions=CONTROLLER:65535
```

Aggregate switch a-2-2's flow tables :

```
cookie=0x0, duration=4567.827s, table=0, n_packets=0, n_bytes=0, idle_age=4567, priority=2000,dl_dst=00:00:00:00:00:00/00:00:00:01:00:00 actions=output:1
cookie=0x0, duration=4567.827s, table=0, n_packets=2, n_bytes=196, idle_age=4513, priority=2000,dl_dst=00:00:00:01:00:00/00:00:00:01:00:00 actions=output:2
cookie=0x0, duration=4567.795s, table=0, n_packets=0, n_bytes=0, idle_age=4567, priority=8500,in_port=5,dl_dst=ff:ff:ff:ff:ff:ff actions=output:3,output:4,output:6,output:1
cookie=0x0, duration=4567.795s, table=0, n_packets=0, n_bytes=0, idle_age=4567, priority=8500,in_port=6,dl_dst=ff:ff:ff:ff:ff:ff actions=output:3,output:4,output:5,output:2
cookie=0x0, duration=4567.796s, table=0, n_packets=0, n_bytes=0, idle_age=4567, priority=8500,in_port=4,dl_dst=ff:ff:ff:ff:ff:ff actions=output:3,output:5,output:6,output:2
cookie=0x0, duration=4567.796s, table=0, n_packets=0, n_bytes=0, idle_age=4567, priority=8500,in_port=3,dl_dst=ff:ff:ff:ff:ff:ff actions=output:4,output:5,output:6,output:1
cookie=0x0, duration=4567.796s, table=0, n_packets=0, n_bytes=0, idle_age=4567, priority=8500,in_port=2,dl_dst=ff:ff:ff:ff:ff:ff actions=output:3,output:4,output:5,output:6
cookie=0x0, duration=4567.796s, table=0, n_packets=0, n_bytes=0, idle_age=4567, priority=8500,in_port=1,dl_dst=ff:ff:ff:ff:ff:ff actions=output:3,output:4,output:5,output:6
cookie=0x0, duration=4567.827s, table=0, n_packets=0, n_bytes=0, idle_age=4567, priority=3000,dl_dst=00:01:03:00:00:00/ff:ff:ff:00:00:00 actions=output:5
cookie=0x0, duration=4567.827s, table=0, n_packets=2, n_bytes=196, idle_age=4513, priority=3000,dl_dst=00:01:01:00:00:00/ff:ff:ff:00:00:00 actions=output:3
cookie=0x0, duration=4567.827s, table=0, n_packets=0, n_bytes=0, idle_age=4567, priority=3000,dl_dst=00:01:02:00:00:00/ff:ff:ff:00:00:00 actions=output:4
cookie=0x0, duration=4567.827s, table=0, n_packets=0, n_bytes=0, idle_age=4567, priority=3000,dl_dst=00:01:04:00:00:00/ff:ff:ff:00:00:00 actions=output:6
cookie=0x0, duration=4585.215s, table=0, n_packets=1586, n_bytes=65026, idle_age=1, priority=65000,dl_dst=01:23:20:00:00:01,dl_type=0x88cc actions=
    CONTROLLER:65535
```

Edge switch e-1-2's flow tables :

```
cookie=0x0, duration=24.956s, table=0, n_packets=2, n_bytes=196, idle_age=9,
   priority=10 actions=resubmit(,1)
 cookie=0x0, duration=10.068s, table=0, n_packets=2, n_bytes=196, idle_age=9,
    priority=5000,in_port=3,dl_src=52:c2:0d:68:be:fd actions=mod_dl_src
    :00:01:01:03:00:01,resubmit(,1)
 cookie=0x0, duration=43.279s, table=0, n_packets=5, n_bytes=205, idle_age=8,
    priority=65000,dl_dst=01:23:20:00:00:01,dl_type=0x88cc actions=CONTROLLER
    :65535
 cookie=0x0, duration=24.956s, table=0, n_packets=2, n_bytes=84, idle_age=4,
    priority=9000,arp,in_port=3 actions=CONTROLLER:65535
 cookie=0x0, duration=24.956s, table=0, n_packets=0, n_bytes=0, idle_age=24,
    priority=9000,arp,in_port=4 actions=CONTROLLER:65535
 cookie=0x0, duration=24.948s, table=1, n_packets=2, n_bytes=196, idle_age=9,
    priority=10 actions=resubmit(,2)
 cookie=0x0, duration=10.033s, table=1, n_packets=2, n_bytes=196, idle_age=9,
    priority=5000,dl_dst=00:01:01:03:00:01 actions=mod_dl_dst:52:c2:0d:68:be:fd,
    output:3
 cookie=0x0, duration=24.978s, table=2, n_packets=0, n_bytes=0, idle_age=24,
    priority=2000,dl_dst=00:00:00:00:00:00/00:00:00:01:00:00 actions=output:1
 cookie=0x0, duration=24.978s, table=2, n_packets=2, n_bytes=196, idle_age=9,
    priority=2000,dl_dst=00:00:00:01:00:00/00:00:00:01:00:00 actions=output:2
 cookie=0x0, duration=24.908s, table=2, n_packets=0, n_bytes=0, idle_age=24,
    priority=8500,in_port=4,dl_dst=ff:ff:ff:ff:ff:ff actions=output:3,output:2
 cookie=0x0, duration=24.908s, table=2, n_packets=0, n_bytes=0, idle_age=24,
    priority=8500,in_port=3,dl_dst=ff:ff:ff:ff:ff:ff actions=output:4,output:1
 cookie=0x0, duration=24.908s, table=2, n_packets=0, n_bytes=0, idle_age=24,
    priority=8500,in_port=2,dl_dst=ff:ff:ff:ff:ff:ff actions=output:3,output:4
 cookie=0x0, duration=24.908s, table=2, n_packets=0, n_bytes=0, idle_age=24,
    priority=8500,in_port=1,dl_dst=ff:ff:ff:ff:ff:ff actions=output:3,output:4
 cookie=0x0, duration=24.978s, table=2, n_packets=0, n_bytes=0, idle_age=24,
    priority=3000,dl_dst=00:01:01:03:00:00/ff:ff:ff:ff:00:00 actions=output:3
 cookie=0x0, duration=24.978s, table=2, n_packets=0, n_bytes=0, idle_age=24,
    priority=3000,dl_dst=00:01:01:04:00:00/ff:ff:ff:ff:00:00 actions=output:4
```

7.3.1 Communication between two known hosts

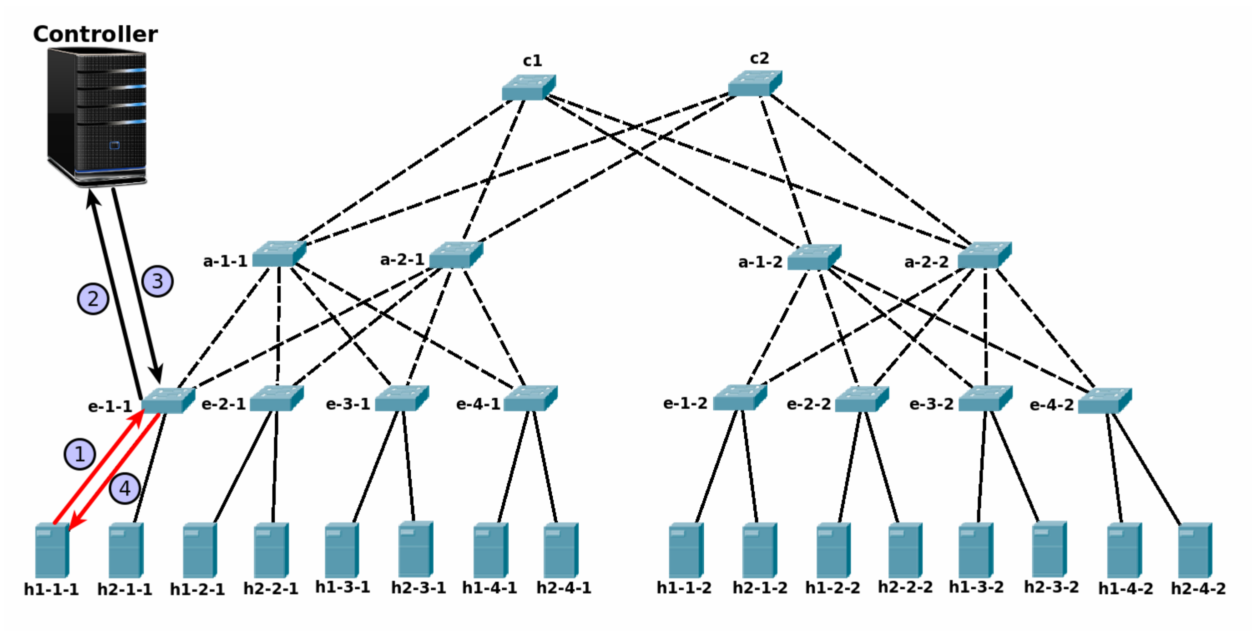


Figure 7.4: Known host h1-1-1 sends an ARP Request for known host h1-1-2 and receives an ARP Response from the Controller

Known host h1-1-1 pings known host h1-1-2 :
Let the IP of h1-1-1 be 10.0.0.1, original MAC be 46:d9:a2:6d:d5:94, and the pesudo MAC(PMAC) assigned by the controller based on host's postion be 00:00:01:03:00:01. Similarly, let the IP of h1-1-2 be 10.0.0.2, original MAC be 3a:57:3d:b2:78:be and the pseudo MAC(PMAC) be 00:01:01:03:00:01

1. h1-1-1 sends an ARP request with h1-1-2's IP address to query for h1-1-2's MAC address (the MAC address of the destination host).

2. This request reaches the Edge switch that h1-1-1 is connected to, namely, e-1-1. All Edge Switches have flow table entries that are configured to send all the ARP packets received from any of its hosts to the controller. Flow table entries 4 and 5 are responsible for this.

3. The controller receives the packet in from the edge switch, makes sure that it knows the src IP, and then checks for VM migration or virtual ip by checking if the previous position and current position of the src IP are still the same. Now, it sees that it is an ARP Request and looks for the queried IP address in its ARP table which is mapping of IP to the PMAC. Since the destination is a known host, the controller has the requested mapping. It crafts an ARP reply as if it originated from h1-1-2, with the src MAC in ARP reply and ethernet packet as the PMAC looked up in its ARP table. The controller then sends this reply as a packet out to the edge switch instructing it to output the packet back to the port connected to h1-1-1.

4. The edge switch e-1-1 receives the packet out from controller and outputs it to the port connected to h1-1-1.

5. h1-1-1 receives the ARP reply and sends the data packet(ICMP echo in this case) to the destination host h1-1-2 with destination MAC as the PMAC it just received in the reply.

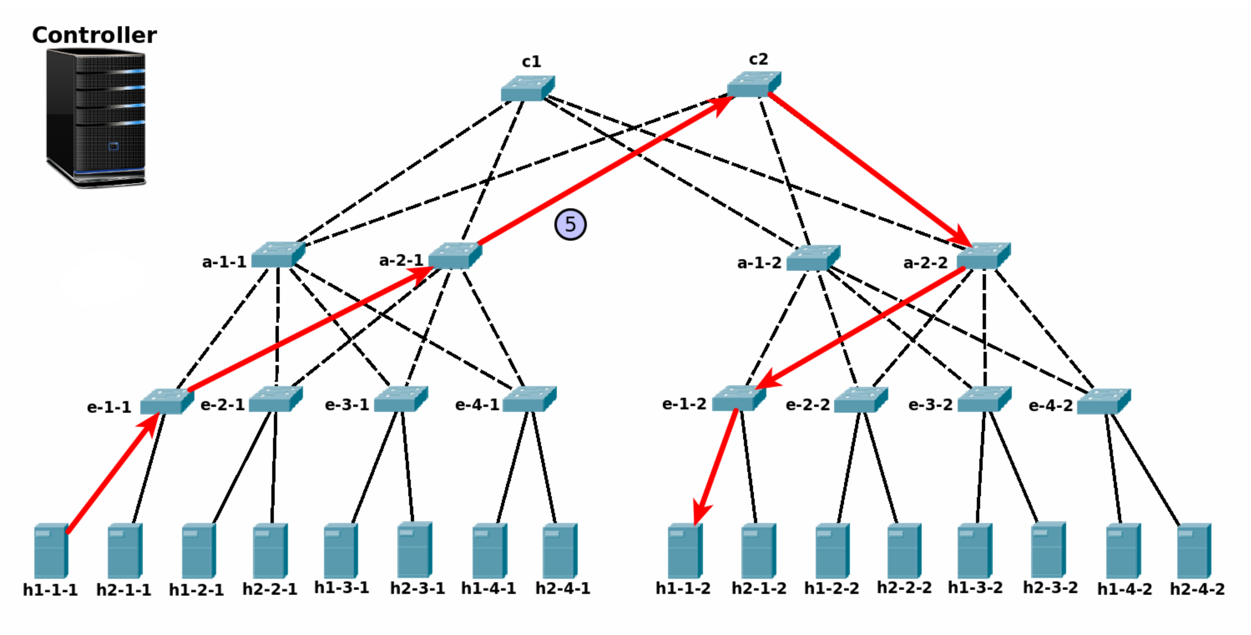


Figure 7.5: h1-1-1 sends ICMP echo request to h1-1-2 through the network

6. Switch e-1-1 receives the packet. Since h1-1-1 is a known host, the switch would have translation entries to change h1-1-1's actual MAC to PMAC and vice versa in its flow tables. Flow table entry 2 corresponds to translating original MAC to PMAC and flow table entry 7 is responsible for translating PMAC to actual MAC. The packet would match two flow table entries : 1 and 3 of table 0. But since entry 3 has a higher priority, that is the one that is matched. The actual mac is replace with the PMAC and the packet is resubmitted to table 1 for further processing. The purpose of table 1 is to translate PMAC to actual MAC. Since that is not required in our case, none of the translation entries match and entry number 6 is matched, whose purpose is to forward the packet to table 2, table responsible for forwarding. Flow tables entries 8 and 9 are responsible for forwarding packets not to one of its hosts or to broadcast address. There are 2 entries to achieve hashing, to make use of the 2 up links. This hashing is based on the destination MAC's port bits(byte 4 of PMAC). Since there are only 2 links, only one bit is required in the mask to choose one of them. Hence the mask is "00:00:00:01:00:00". This mask is applied on the destination MAC and the result compared to MAC in the table. If both are the same, this entry is matched. Here, the destination mac is 00:01:01:03:00:01. The 32nd bit is 1, hence it matches entry 9 and the packet is output to port 2 which is connected to aggregate switch a-2-1.

7. The Aggregate Switch a-2-1 receives the packet after the src MAC was modified. Since the dest is in another pod, it has to send the packet to one of the core switches it is attached to. Flow table entries one and two are responsible for doing this. As explained before, bit mask is applied and matching is done and flow table entry 1 is matched. The packet is output to port 2 which is connected to core switch c2.

8. The Core switch, c2 receives the packet from a-2-1 and figures out which pod the destination MAC is in by looking at the first two bytes of MAC address. The pod number of the destination MAC is 1 in our case(pod numbers are 0 indexed in the MAC address). It has two links to that pod. Flow table entries 10 and 11 are responsible for forwarding to that mod and load balancing between the two links. The loadbalancing is done as described before based on the destination MAC's 4th byte. In our case entry 11 matches and the packet is output to port 4 which is connected to aggregate switch a-2-2.

9. The aggregate switch a-2-2 receives the packet from C2 and should forward the packet to the corresponding edge switch. It does this by looking at the 3rd byte of the destination MAC to find out which edge switch that destination MAC belongs to. Flow table entries 9, 10, 11 and 12 are responsible for forwarding packets to its own pod. In this case, flow table entry 9 matches. So this packet is output to port 3 which is connected to edge switch e-1-2, the switch host h1-1-2 belongs to.

10. Switch e-1-1 receives the packet. The switch would have translation entries to change h1-1-2's actual MAC to PMAC and vice versa in its flow tables as h1-1-2 is a known host. No flows match in table 0 apart from the default flow 1 which resubmits to table 1. Table 1 which is for translating PMAC to actual MAC has an entry(7) to translate the PMAC to the actual MAC of h1-1-2. This will also output the modified packet to the port connected to h1-1-2. This way, the data packet(ICMP echo request in our case) from h1-1-1 has reached its destination h1-1-2.

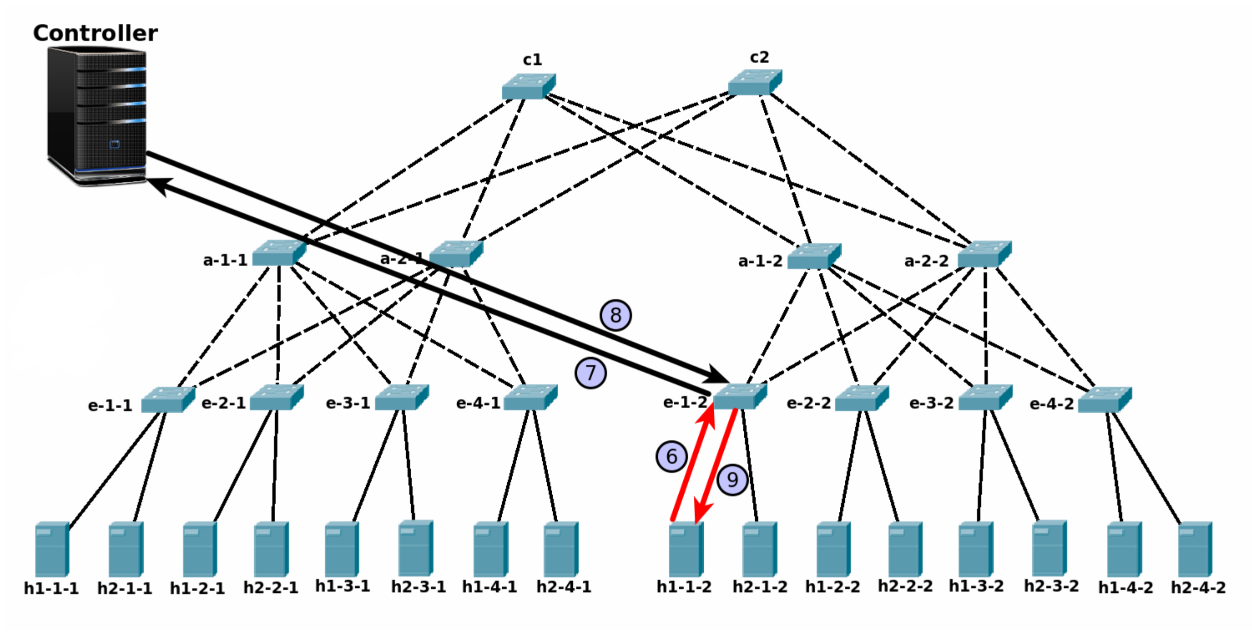


Figure 7.6: h1-1-2 sends an ARP Request for h1-1-1's MAC address and receives an ARP Response from the Controller

11. Now h1-1-2 has to reply to the ICMP echo request. But it would not have the ARP entry in its table as h1-1-1's ARP request was intercepted by the controller. Now since h1-1-2 does not have the MAC, it makes an ARP request. As explained before, the packet would go to edge switch(step 6) which is then sent to the controller(step 7) and controller crafts an ARP reply with h1-1-1's PMAC and sends it back to edge switch(step 8). The edge switch receives the packet out and outputs to the port connecting to h1-1-2(step 9). Now h1-1-2 has the PMAC of h1-1-1 and can reply back. The steps to reach h1-1-1 is similar to the steps taken to reach h1-1-2 - forwarding and load balancing based on the destination MAC(step 10). In a traditional network, the ARP request would have been broadcast. And also, along the path to the destination, there could have been unknown unicasts if the MAC was not known by the switches in the path. With our system, all these have been avoided, redundant links are being utilized for load balancing and also the ARP reply could be faster since the ARP query does not have to go over the network to the destination and back again.

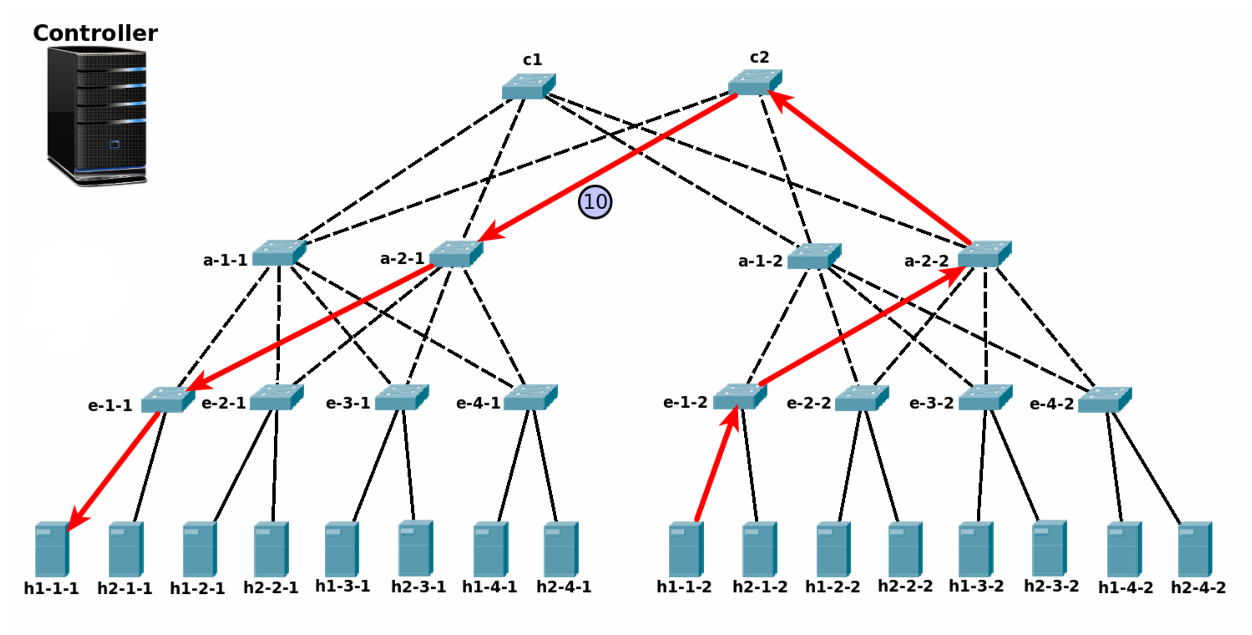


Figure 7.7: h1-1-2 sends ping response back to h1-1-1

7.3.2 Communication between a new source host and a known destination host

New host h1-1-1 pings known host h1-1-2 :

Let the IP of h1-1-1 be 10.0.0.1 and original MAC be 46:d9:a2:6d:d5:94.
Similarly, let the IP of h1-1-2 be 10.0.0.2, original MAC be 3a:57:3d:b2:78:be and the pseudo MAC(PMAC) assigned by the controller based on host's position be 00:01:01:03:00:01

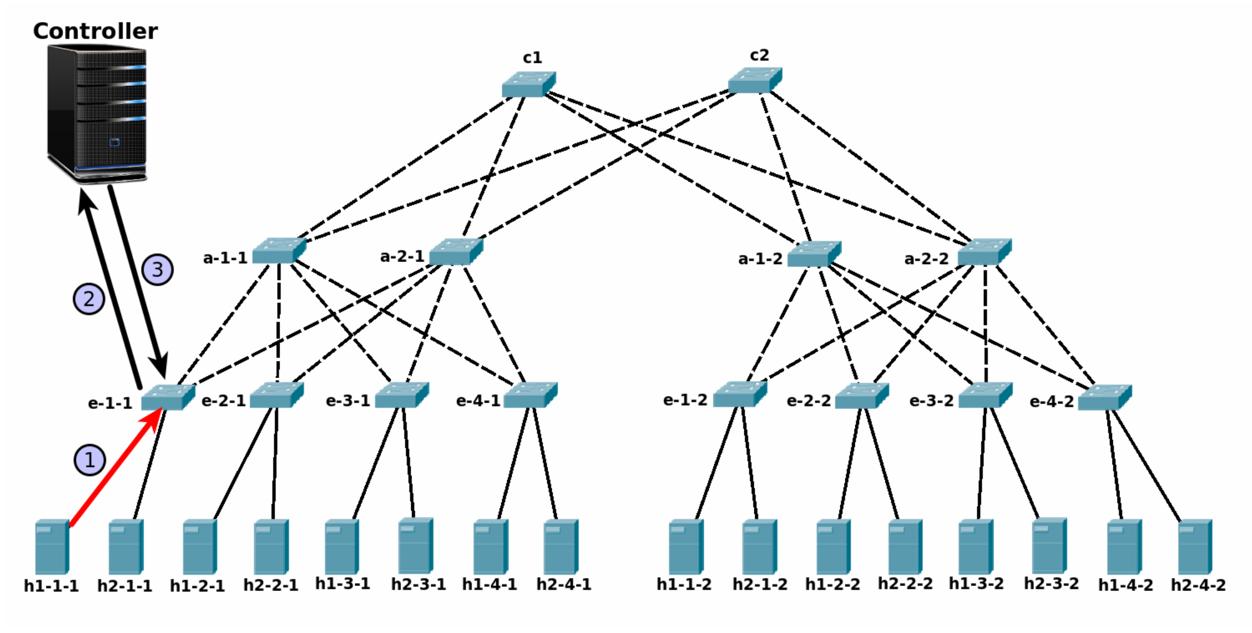


Figure 7.8: h1-1-1 sends an ARP Request for h1-1-2 and controller inserts flow table entries

1. h1-1-1 sends an ARP request with h1-1-2's IP address to query for h1-1-2's MAC address

(the MAC address of the destination host).

2. This request reaches the Edge switch that h1-1-1 is connected to, namely, e-1-1. All Edge Switches have flow table entries that are configured to send all the ARP packets received from any of its hosts to the controller. Flow table entries 4 and 5 are responsible for this.

3. The controller receives the packet in from the edge switch and finds out that the src IP is new. It detects that the src is a new host and assigns a PMAC for this IP based on the position. The position here is pod 1(but pod numbers are zero indexed), belongs to edge switch 1, and port 3. Assuming this is the first host from that port, the vmid will be 1. So the PMAC that will be assigned is 00:00:01:03:00:01. It then crafts translation entries to be put in the edge switch for translating src MAC from the actual MAC to PMAC and dst MAC from PMAC to actual MAC in tables 0 and 1 respectively. IN table 1, after modifying dst MAC, we also output to the port of h1-1-1 as that is the destination. The controller then sends these flow mod messages to the edge switch along with a barrier request. A barrier request tells the switch to process all the flow mods it has received and reply back with barrier reply after it has processed them. We do this to make sure that packets can then be sent and received from the assigned PMAC. Only after the switch replies back for the barrier can we be safe to use the newly assigned PMAC in the network. Before sending the barrier, the controller adds a listener for the barrier reply on this switch and also stores the some state associated to that barrier like the event object to be processed later on barrier reply, the IP and the PMAC just assigned to update its tables later.

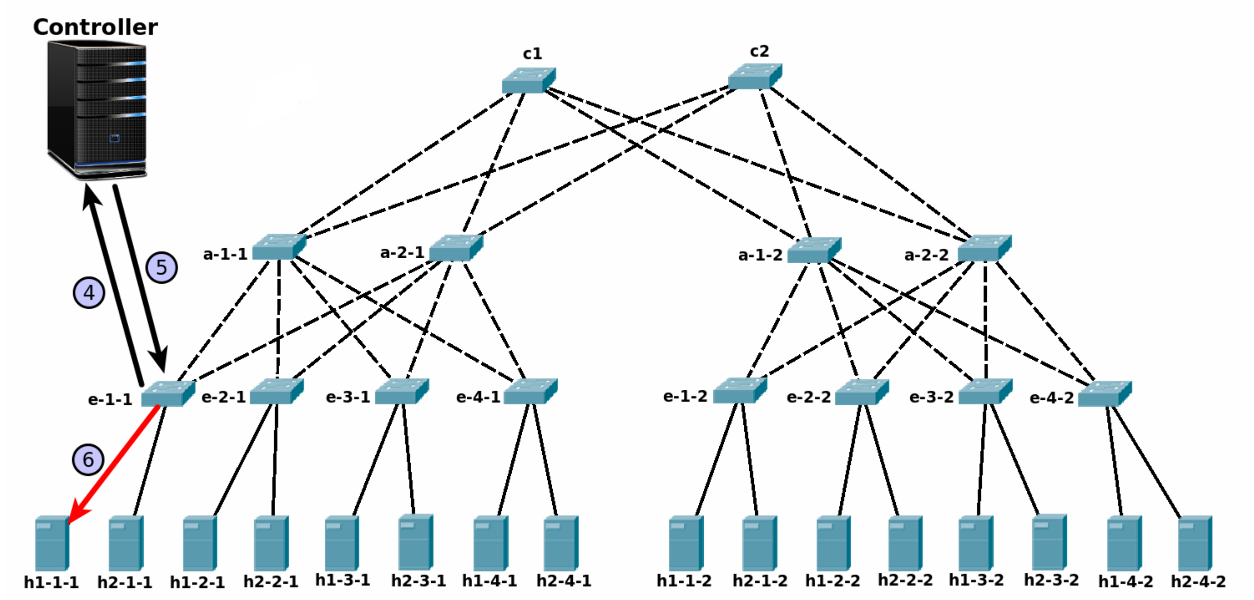


Figure 7.9: e-1-1 replies to the barrier and controller sends ARP reply to h1-1-1

4. The edge Switch e-1-1 receives the flow mods and the barrier. Since it has received the barrier, it will process all the flow mods it has received which will result in the creation of these two entries in its flow tables.

```
cookie=0x0, duration=14.455s, table=0, n_packets=2, n_bytes=196, idle_age
    =14, priority=5000,in_port=3,dl_src=46:d9:a2:6d:d5:94 actions=
    mod_dl_src:00:00:01:03:00:01,resubmit(,1)
cookie=0x0, duration=14.454s, table=1, n_packets=2, n_bytes=196, idle_age
    =14, priority=5000,dl_dst=00:00:01:03:00:01 actions=mod_dl_dst:46:d9:
    a2:6d:d5:94,output:3
```

The edge switch then sends back a barrier response(step 5) to the controller indicating it has finished processing the mods.

5. The controller receives the barrier reply. It will then retrieve the state info it has associated for this barrier, stop listening for barrier replies from this switch, update its ARP table with the IP -> PMAC mapping, actual MAC -> PMAC mapping and vice-versa and then further processes the ARP request. Since the destination is a known host, its IP is there in the controllers ARP table. On finding the IP -> PMAC mapping, controller crafts an ARP reply as if it originated from the h1-1-2 by replacing the src MAC in ethernet and ARP packet with the PMAC it obtained from the look up. It then sends this reply as a packet out to the edge switch instructing it to output the packet to the port connected to h1-1-1. The edge switch e-1-1 receives the packet out from controller and outputs it to the port connected to h1-1-1.

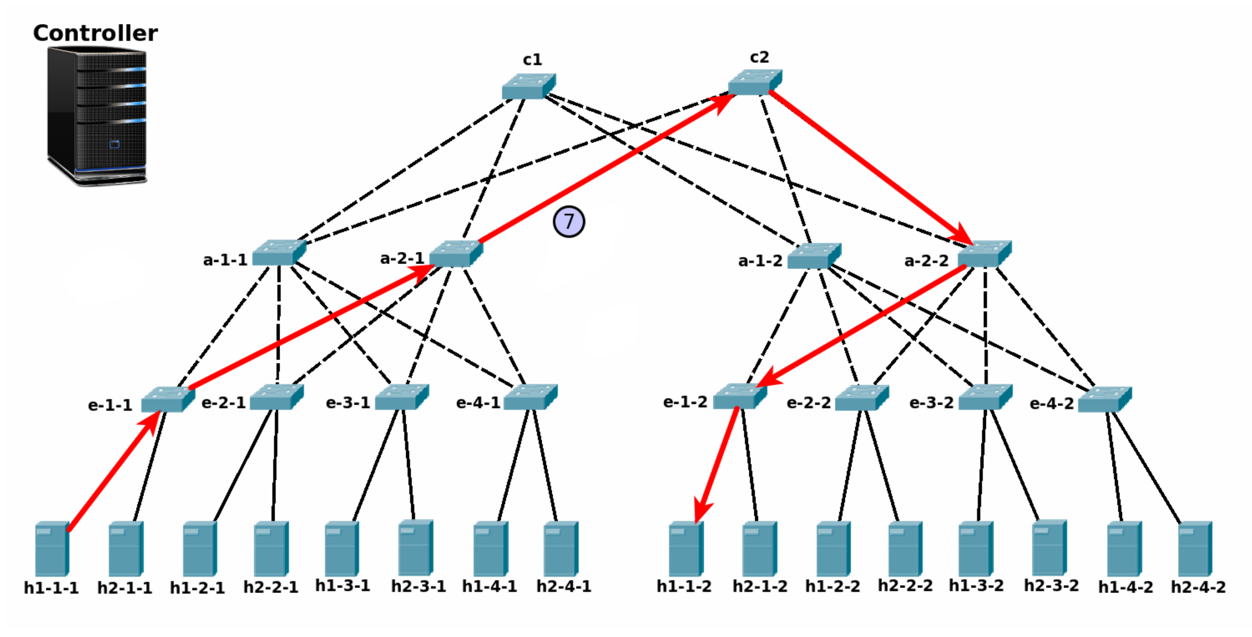


Figure 7.10: h1-1-1 sends ICMP echo request to h1-1-2 through the network

6. h1-1-1 receives the ARP reply and sends the data packet(ICMP echo in this case) to the destination host h1-1-2 with destination MAC as the PMAC it just received in the reply. The rest of the process is as described in scenario 1.

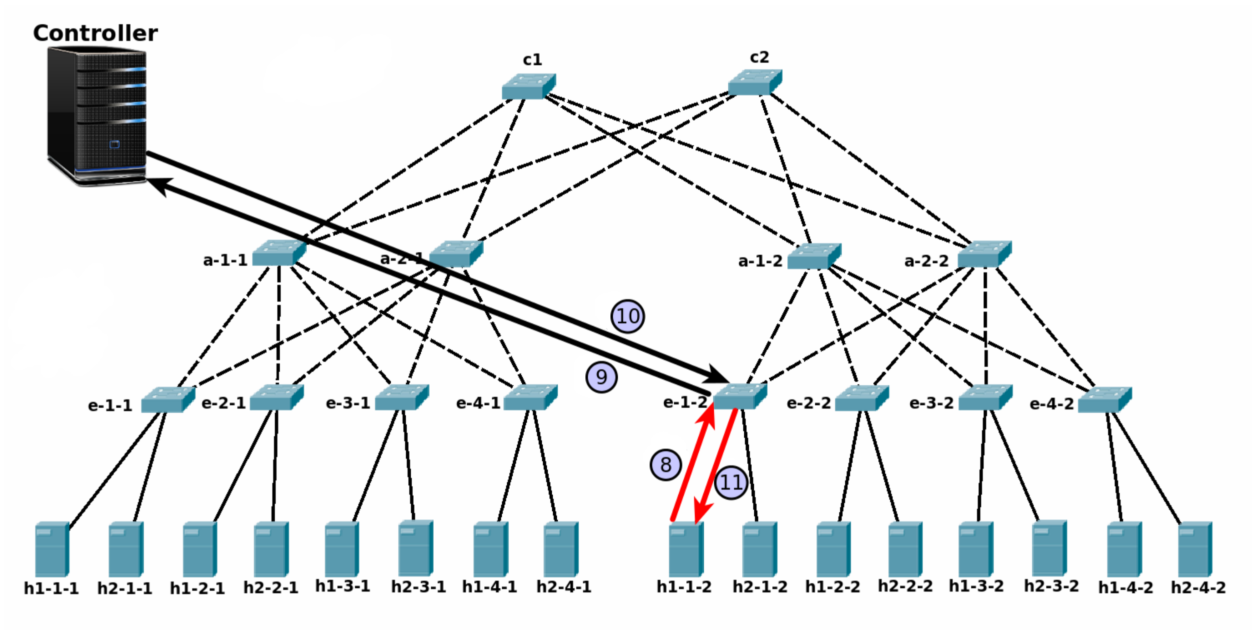


Figure 7.11: h1-1-2 sends an ARP Request for the h1-1-1's MAC address and receives an ARP Response from the Controller

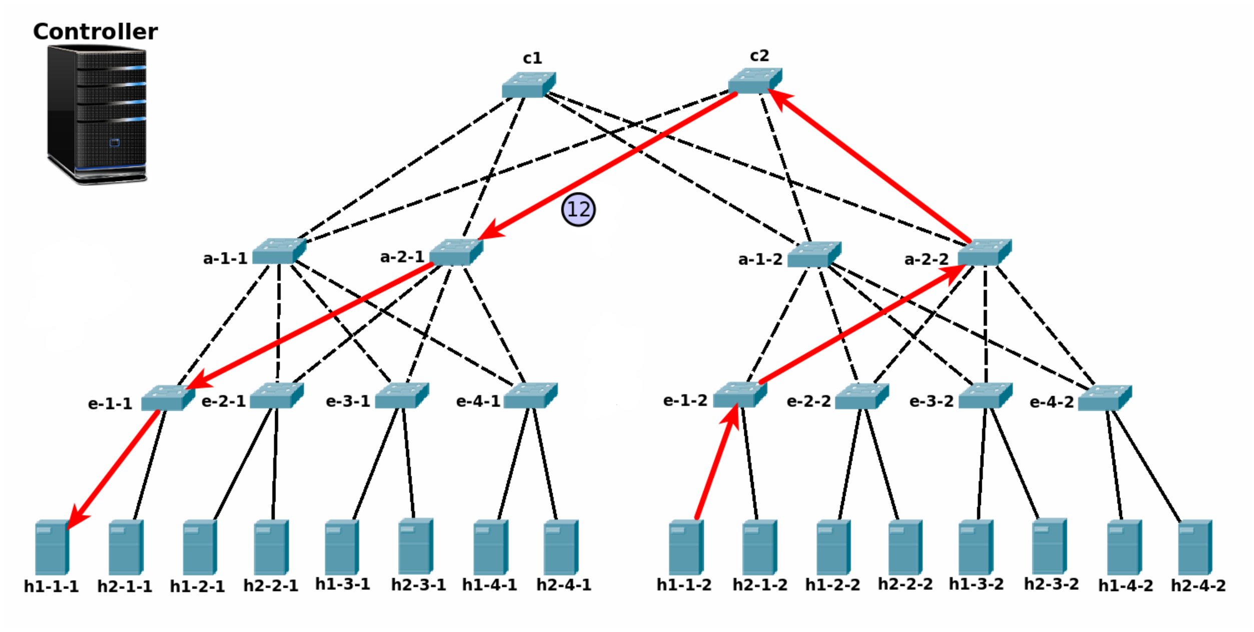


Figure 7.12: h1-1-2 sends ping response back to h1-1-1

7.3.3 Communication between a known source host and a new destination host

Known host h1-1-1 pings new host h1-1-2 :

Let the IP of h1-1-1 be 10.0.0.1 and original MAC be 46:d9:a2:6d:d5:94 and the pseudo MAC(PMAC) assigned by the controller based on host's postion be 00:00:01:03:00:01. Similarly, let the IP of h1-1-2 be 10.0.0.2 and original MAC be 3a:57:3d:b2:78:be.

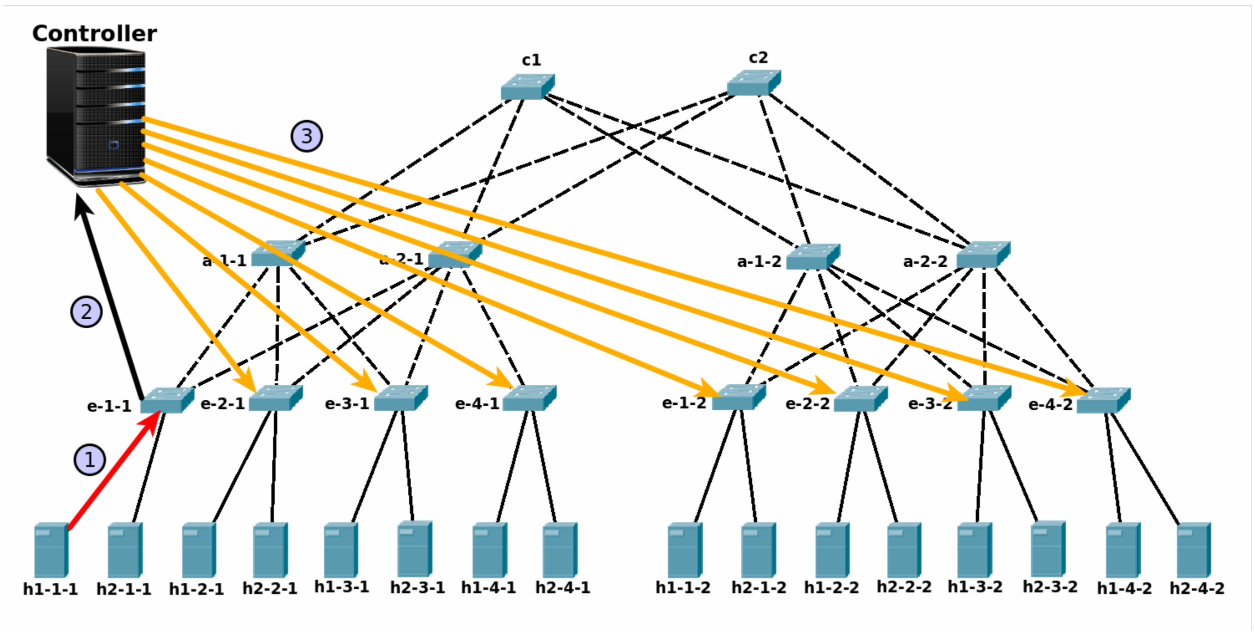


Figure 7.13: h1-1-1 sends ARP request for h1-1-2 and controller initiates an edge broadcast

1. h1-1-1 sends an ARP request with h1-1-2's IP address to query for h1-1-2's MAC address (the MAC address of the destination host).

2. This request reaches the Edge switch that h1-1-1 is connected to, namely, e-1-1. All Edge Switches have flow table entries that are configured to send all the ARP packets received from any of its hosts to the controller. Flow table entries 4 and 5 are responsible for this.

3. The controller receives the packet in from the edge switch, makes sure that it knows the src IP, and then checks for VM migration or virtual ip by checking if the previous position and current position of the src IP are still the same. Now, it sees that it is an ARP Request and looks for the queried IP address in its ARP table. Since the destination is a new host, the controller does not have the requested mapping. It now needs to broadcast this request to the whole network. But before broadcasting, the controller replaces the src MAC in the ARP request and ethernet packet with the PMAC of h1-1-1. It gets the PMAC by looking up the actual MAC in its actual MAC -> PMAC table. It now performs an edge broadcast(step 3 and 4) - send this modified ARP request as a packet out to each edge switch and instruct it to output the packet to all the ports that are connected to hosts. While doing this, it does not include the port connected to h1-1-1 in the output ports for the edge switch of h1-1-1. This is done because h1-1-1 would then receive an ARP packet which says that 10.0.0.1 is belonging to some other MAC(the PMAC of h1-1-1 itself). And in cases like ARP probe, it will interpret this as someone else already owning its IP and stop using the IP.

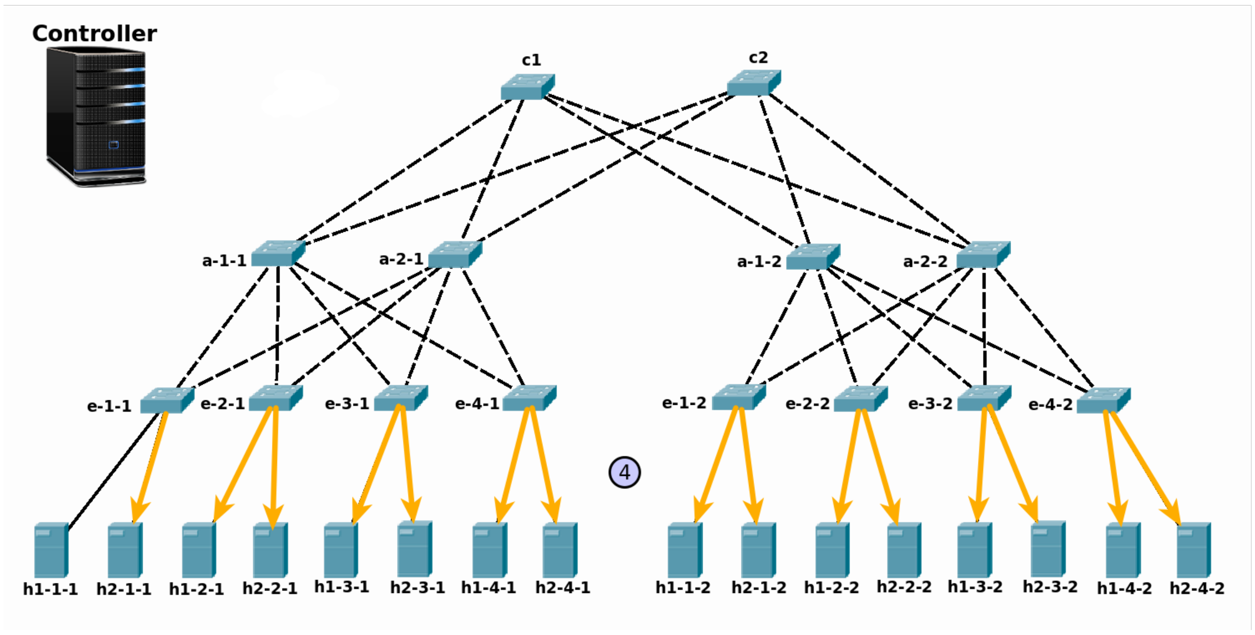


Figure 7.14: Edge broadcast in progress

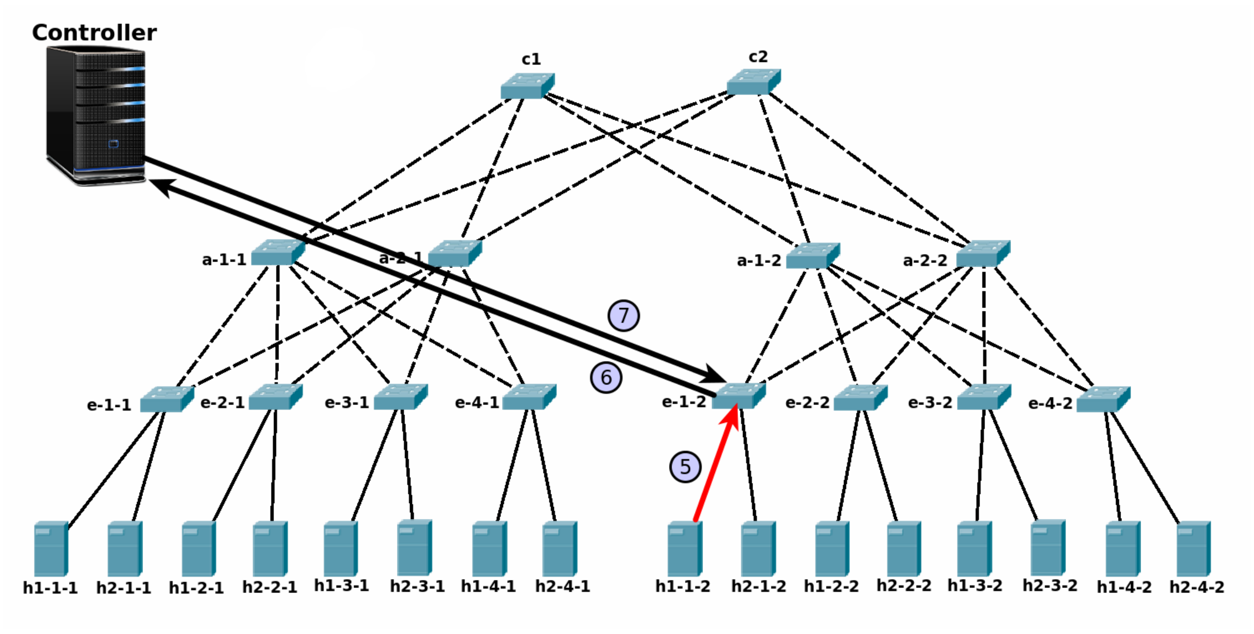


Figure 7.15: h1-1-2 responds and controller intercepts it and inserts flow table entries in e-1-2

4. h1-1-2 receives the ARP request, just like all other hosts, as the result of edge broadcast. Since it is the owner of 10.0.0.2, it sends an ARP reply with its own MAC address.

5. The edge switch e-1-2 receives the ARP reply, and on matching that it is an ARP, sends the packet to the controller.

6. The controller receives the packet in from the edge switch and finds out that the src IP is new. It detects that the src is a new host and assigns a PMAC for this IP based on the position. The position here is pod 2(but pod numbers are zero indexed), belongs to

edge switch 1 in that pod, and port 3. Assuming this is the first host from that port, the vmid will be 1. So the PMAC that will be assigned is 01:00:01:03:00:01. It then crafts translation entries to be put in the edge switch for translating src MAC from the actual MAC to PMAC and dst MAC from PMAC to actual MAC in tables 0 and 1 respectively. In table 1, after modifying dst MAC, we also output to the port of h1-1-2(port 3) as that is the destination. The controller then sends these flow mod messages to the edge switch e-1-2 along with a barrier request for reasons explained before.

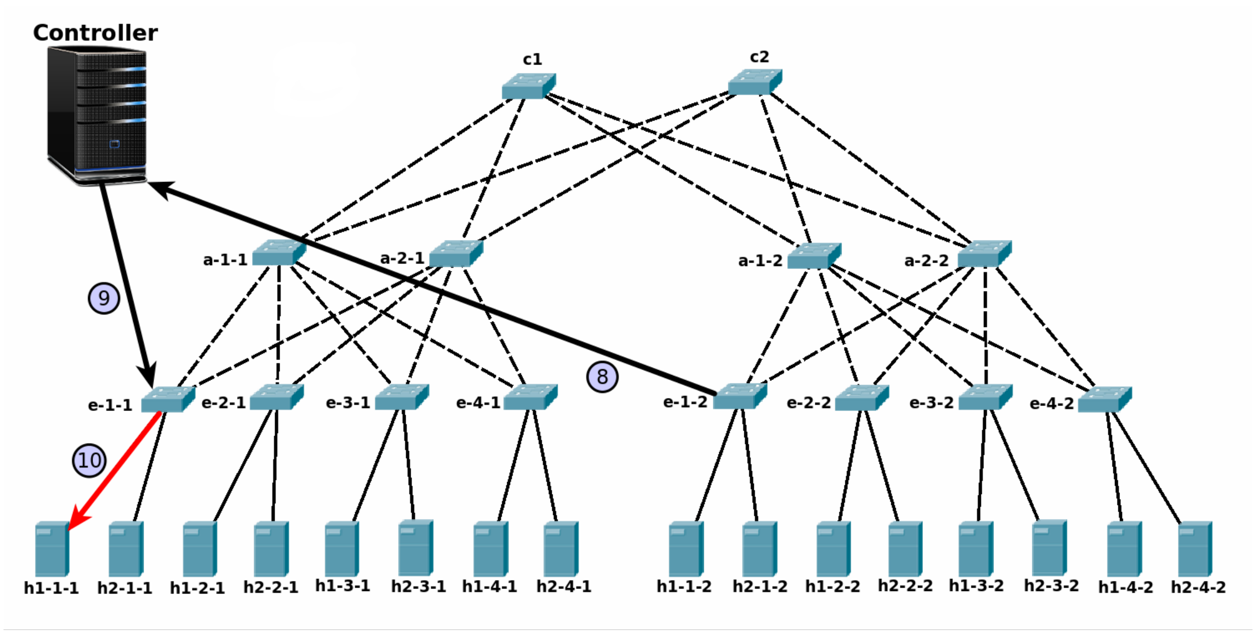


Figure 7.16: e-1-2 responds to barrier and controller sends a modified ARP reply of h1-1-2 as packet out to e-1-1

7. The edge Switch e-1-2 receives the flow mods and the barrier. Since it has received the barrier, it will process all the flow mods it has received which will result in the creation of these two entries in its flow tables.

```
cookie=0x0, duration=14.455s, table=0, n_packets=2, n_bytes=196, idle_age
    =14, priority=5000,in_port=3,dl_src=46:d9:a2:6d:d5:94 actions=
    mod_dl_src:00:01:01:03:00:01,resubmit(,1)
cookie=0x0, duration=14.454s, table=1, n_packets=2, n_bytes=196, idle_age
    =14, priority=5000,dl_dst=00:01:01:03:00:01 actions=mod_dl_dst:46:d9:
    a2:6d:d5:94,output:3
```

The edge switch then sends back a barrier response to the controller indicating it has finished processing the mods.

8. The controller receives the barrier reply. It will then retrieve the state info it has associated for this barrier, stop listening for barrier replies from this switch, update its ARP table with the IP -> PMAC mapping, actual MAC -> PMAC mapping and vice-versa and then further processes the ARP packet. It sees that it is an ARP reply, it replaces the src MAC

in the reply an the ethernet packet with the PMAC of h1-1-2. Also it replaces the dst MAC in the ARP reply and ethernet packet with the actual MAC of h1-1-1. From the pseudo MAC of h1-1-1, it figures out the switch and port h1-1-1 is connected to by looking up in its switch position tables and sends this modified reply as a packet out to that switch instructing it to send it to the port connected to h1-1-1. The edge switch e-1-1 receives the packet out from controller and outputs it to the port connected to h1-1-1.

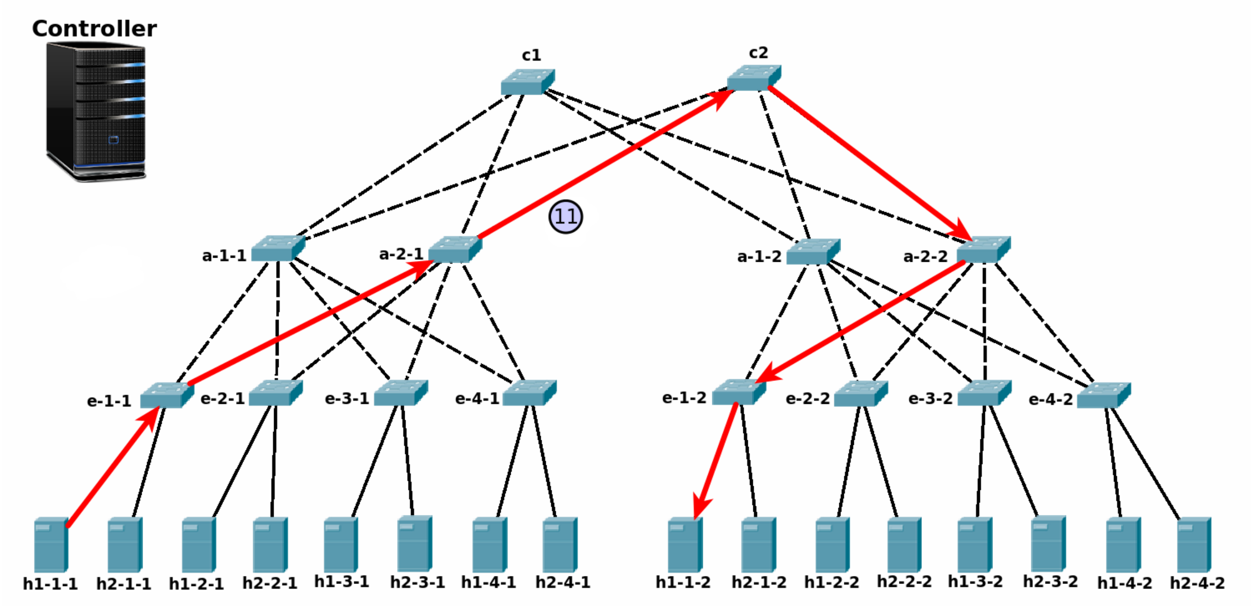


Figure 7.17: h1-1-1 sends an ICMP echo request to h1-1-2

9. h1-1-1 receives the ARP reply and sends the data packet(ICMP echo in this case) to the destination host h1-1-2 with destination MAC as the PMAC it just received in the reply. The rest of the process is as described in scenario 1. But in this case, h1-1-2 will most likely have the ARP mapping for h1-1-1. So there will not be an additional ARP request like in scenario 1. The pictures shown assume that the ARP mapping in h1-1-2 for h1-1-1 has not stalled.

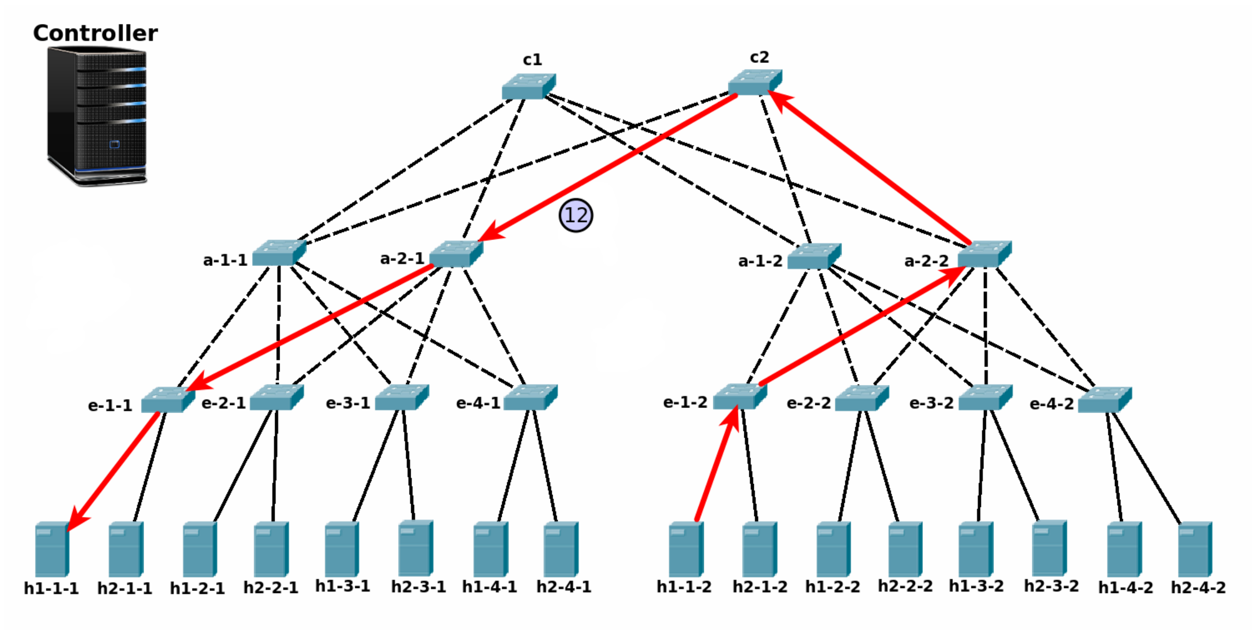


Figure 7.18: h1-1-2 sends ping response back to h1-1-1

7.3.4 Communication between two new hosts

New host h1-1-1 pings new host h1-1-2 :

Let the IP of h1-1-1 be 10.0.0.1 and original MAC be 46:d9:a2:6d:d5:94. Similarly, let the IP of h1-1-2 be 10.0.0.2 and original MAC be 3a:57:3d:b2:78:be.

This scenario is a combination of scenarios mentioned in sections 6.4.2 and 6.4.3.

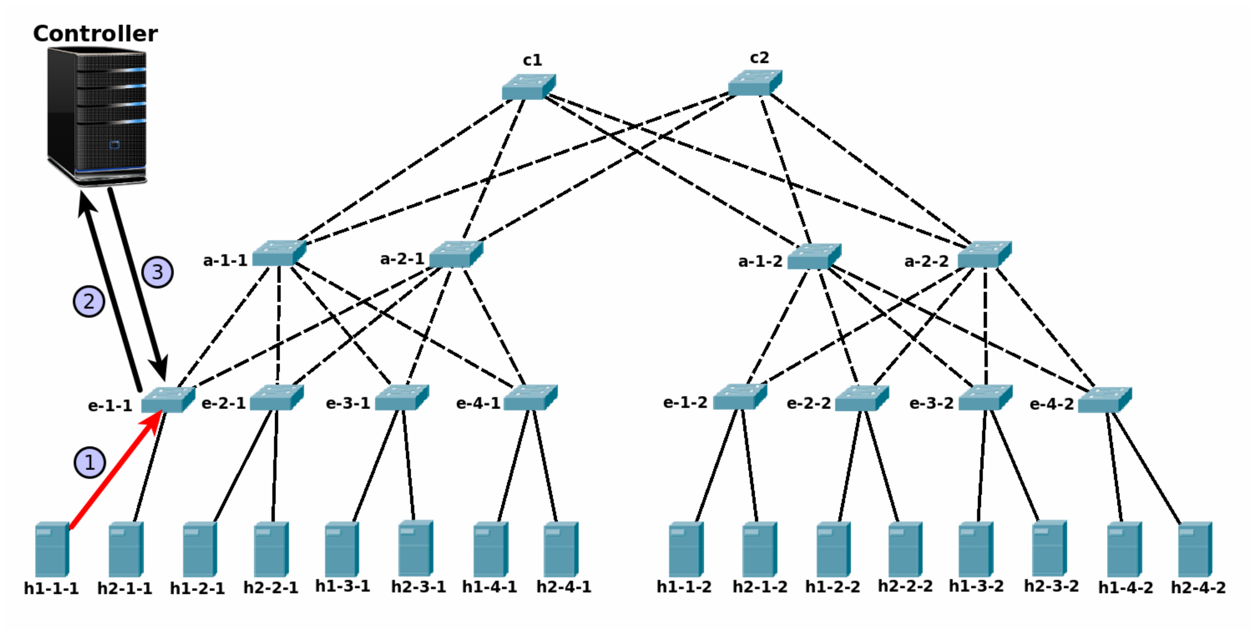


Figure 7.19: h1-1-1 sends ARP request for h1-1-2's MAC and controller inserts flow table entries in e-1-1 for h1-1-1

1. The steps 1-4 are identical to what has been described in scenario 6.4.2.

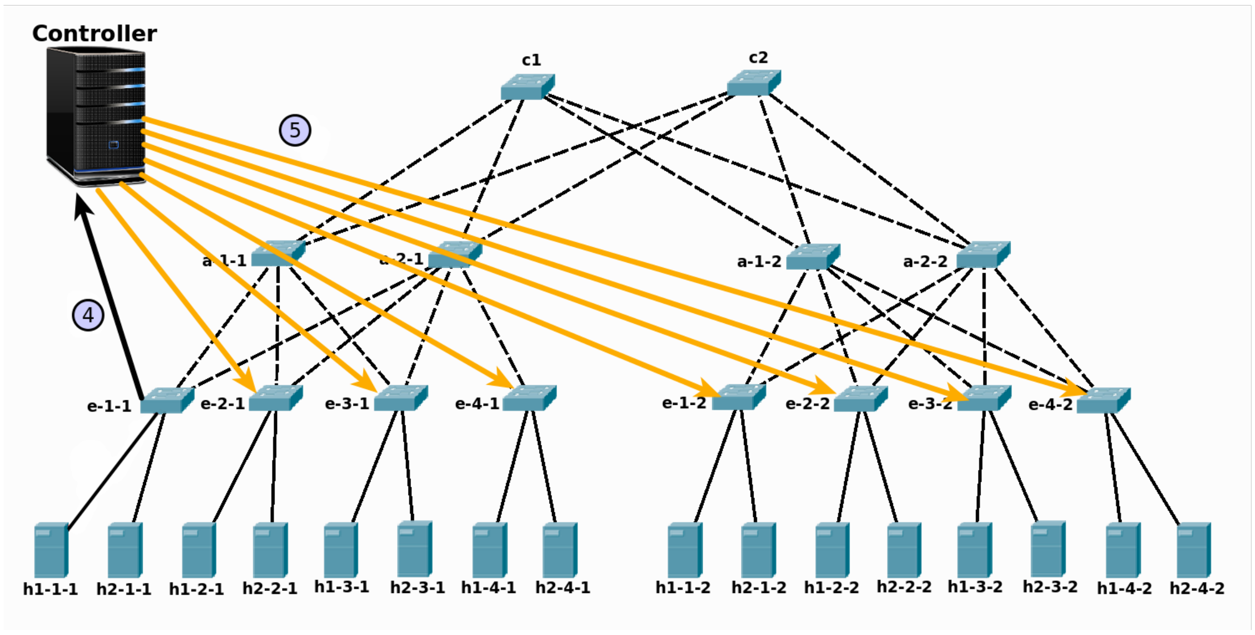


Figure 7.20: e-1-1 sends a barrier response and controller initiates edge broadcast for h1-1-2

2. After the edge switch e-1-1 has responded with a barrier indicating that it has finished processing the flow mods, the controller then initiates an edge broadcast since host h1-1-2 is not known.

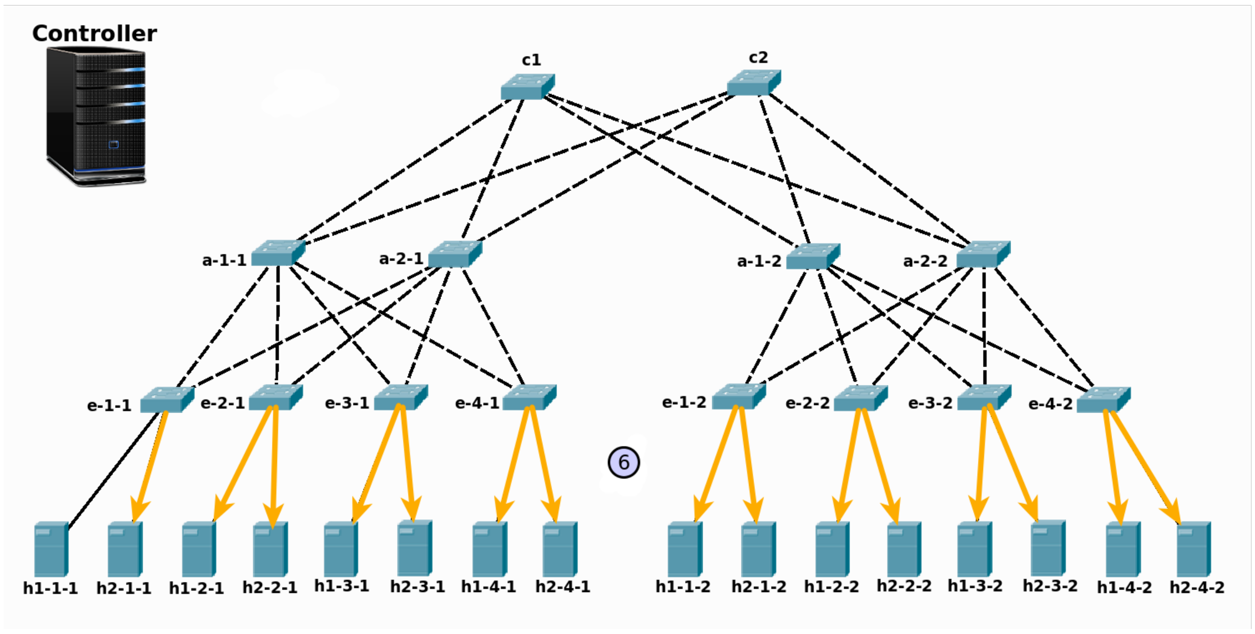


Figure 7.21: edge broadcast for h1-1-2 in progress

3. The rest of the process is as described in section 6.4.3(steps 5-12).

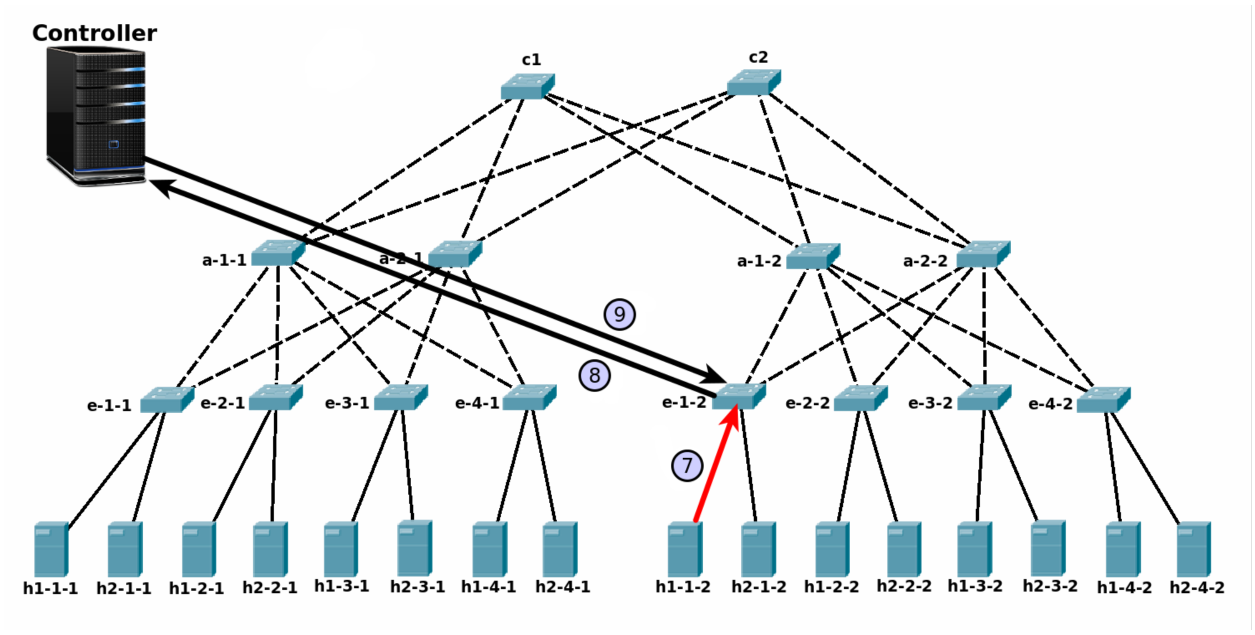


Figure 7.22: h1-1-2 responds and controller intercepts it and inserts flow table entries in e-1-2

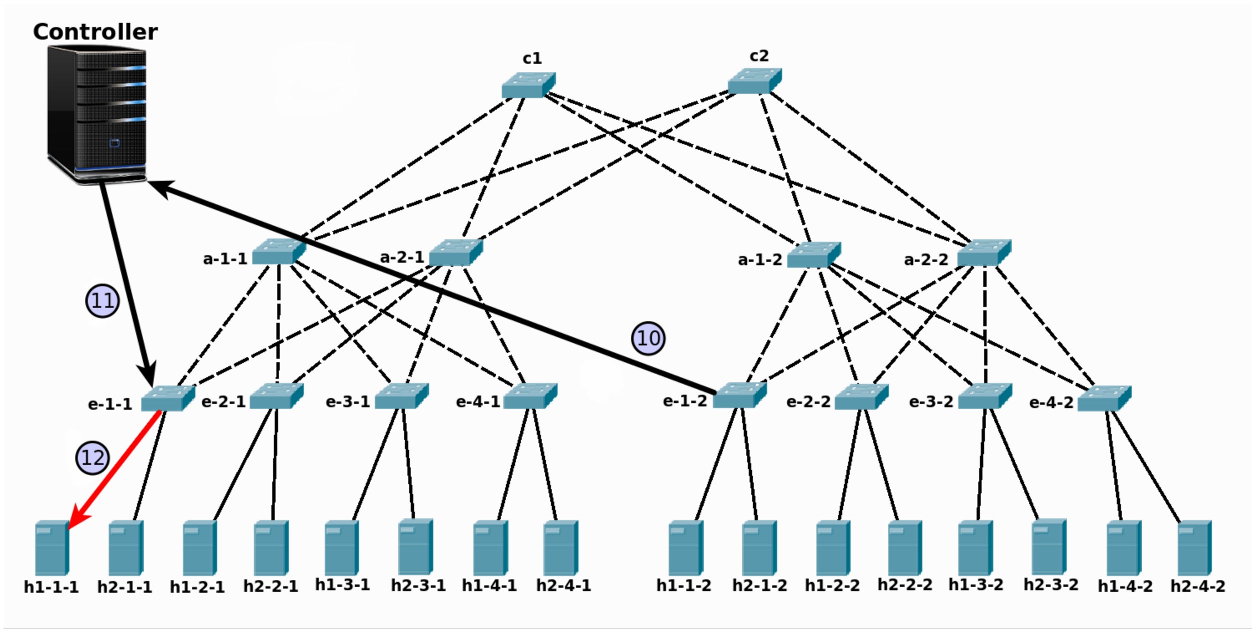


Figure 7.23: e-1-2 responds to barrier and controller sends a modified ARP reply of h1-1-2 as packet out to e-1-1

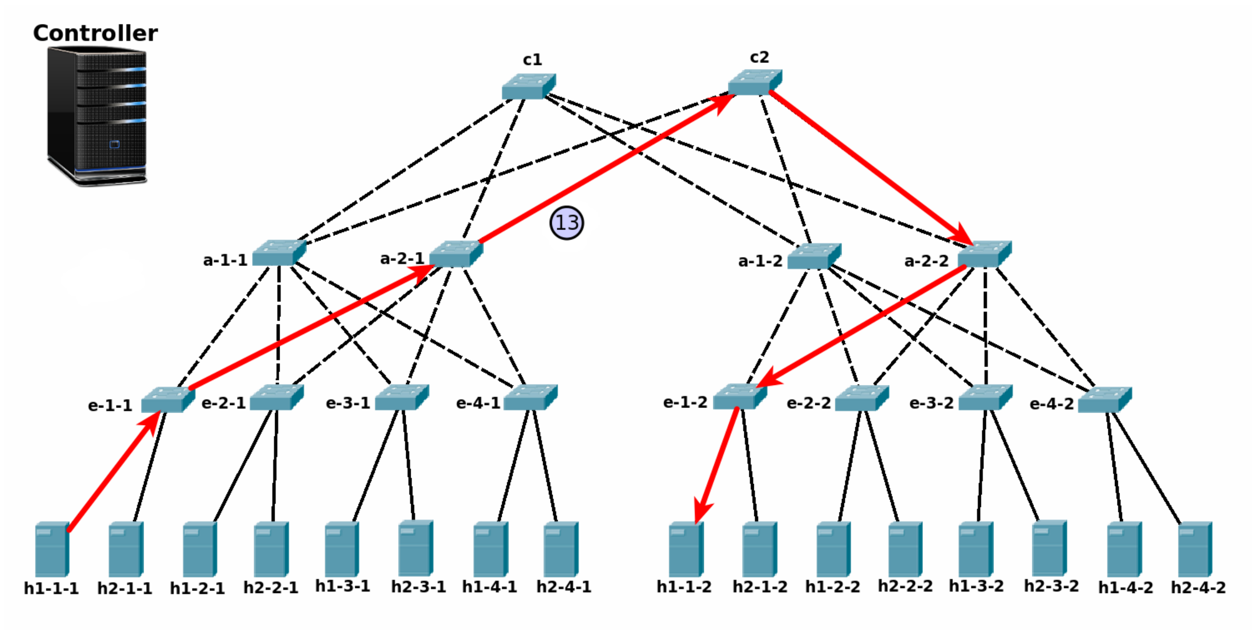


Figure 7.24: h1-1-1 sends an ICMP echo request to h1-1-2

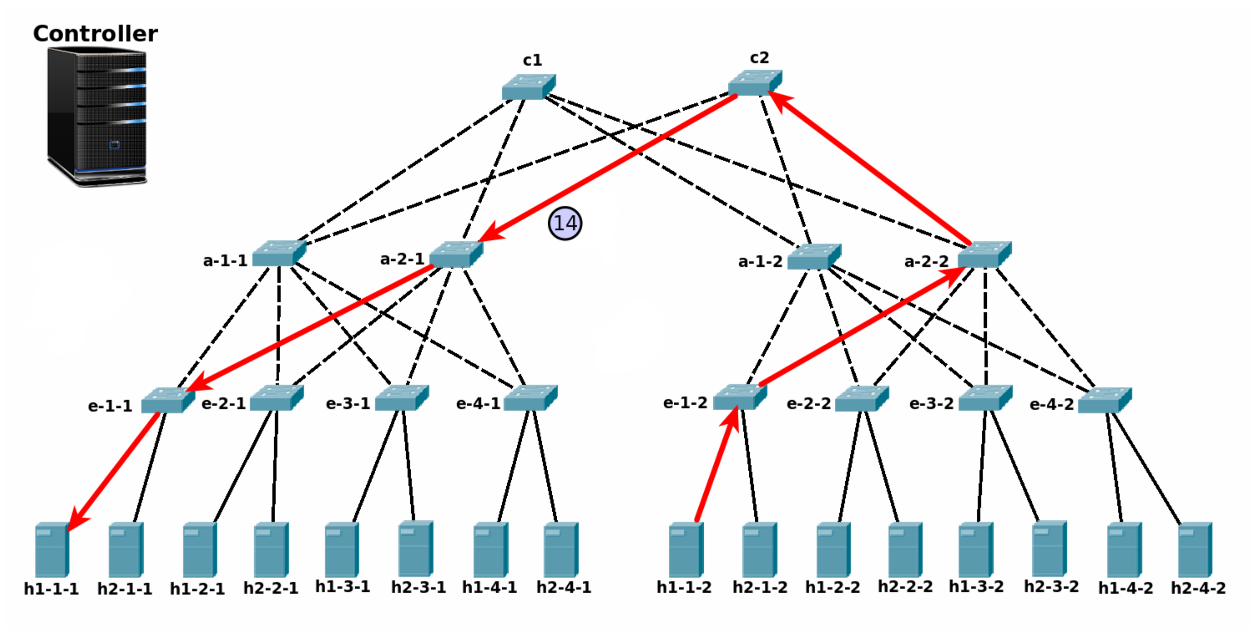


Figure 7.25: h1-1-2 sends ping response back to h1-1-1

8. Detailed Design

8.1 Modules from Architecture

8.1.1 OpenFlow Network - Mininet Topology

This part of the architecture is the OpenFlow enabled, 3-tier multi rooted tree topology data center. For emulating this, mininet was used. A custom topology module was developed using mininet API to create this network.

8.1.2 ARP Preprocessor

It is responsible for -

1. Detecting and handling new hosts by assigning it a PMAC, inserting translation entries into the corresponding edge switch and updating internals tables with these mappings.
2. Detection of VM migration by monitoring the change of VM's position within the network and supporting it by immediately assigning it a new PMAC, updating ARP entries and diverting all the flows going towards the old VM position, to the new VM position.
3. Detecting a Virtual IP takeover by observing that IP is now being used by another host, assigning it a new PMAC, updating ARP entries and diverting all the flows going towards the previous owner of VIP, to the new owner.
4. Receiving gratuitous ARP and updating its ARP table, detecting an ARP probe and maintaining position table for potential hosts(host which do not yet have an IP) to be used later by ARP responder.

8.1.3 ARP Responder

ARP responder is responsible for responding to the ARP packets after ARP preprocessor has done its job and updated tables if any. If the ARP packet is a request and it has the mapping in its ARP table, it responds with the mapping and if it does not have the mapping, it initiates an edge broadcast. If it is an ARP reply, it modifies the reply with PMAC and sends it to the destination.

8.1.4 SDN Application Graph builder

This module listens for connectionUP events(new switches) from the POX controller and link events from the openflow discovery module to construct a graph of the whole network topology which will then further be used by other modules like Graph Identifier.

8.1.5 Graph Identifier

This module interprets the graph that is built and identifies the edge switches based on the percentage of its ports connected to other switches and comparing it with the max value given by the user. All the switches whose percentage is less than or equal to that max value get classified as edge switches. All switches connected to the edge switches are classified as aggregate and the remaining switches as core switches. Once this classification is done, it identifies the pods. It does this by removing all the aggregate to core links, marking all the aggregate and edge switches as unvisited, and doing a breadth first traversal on the unvisited aggregate switches. Each such traversal will yield a pod. These pods are assigned unique pod numbers and each switch within a pod is assigned a unique position. And all this info is stored in tables to be used later.

8.1.6 Partition MAC address space and insert forwarding tables

The whole MAC address space is partitioned based on this structure | *pod-num-16* | *switch-pos-8* | *port-8* | *vmid-16* |. The information from that graph identifier is used to build the forwarding entries for the switches. When there are multiple ways to reach a destination, simple bit mask hashing is done to utilize redundant links and load balance the flows between them. The size of the whole forwarding table is very compact as they are based on the hierarchical PMAC and unknown unicast is completely avoided as there are entries to match any address within the whole MAC address space. The detailed algorithm is explained later in the pseudo code section.

8.1.7 Define Broadcast semantics and insert into switches

The network typically has a lot of redundancy in-built, which means there are loops present. In L2 networks, spanning tree protocol has to be run if the network has loops to avoid broadcast storms. The purpose of this module is to facilitate efficient broadcasts without resorting to spanning tree protocol which would disable the redundant links. This is achieved by defining the broadcast semantics(up ports and down ports) and inserting flow tables entries into the switches to perform broadcast in a bubble and burst fashion.

9. Implementation

9.1 Implementation choices

To create network topology, mininet API needs to be used and that API is available only in Python. Hence our topology modules were developed in Python. Similarly, even genTraffic module, the module to create parallel Ping traffic, needed mininet API and was developed in Python.
We had to choose a controller upon which to build our SDN application. The choices were :

1. NOX
2. POX
3. Floodlight
4. Ryu
5. Pyretic

9.2 Reasons for choosing a particular implementation

Pyretic is a controller mainly for Northbound API. Hence, we would not have that as much granular control at the flow table entries as we would with southbound controllers like the other 4. Hence, pyretic was discarded. All of remaining support OpenFlow 1.0, but apart from features of OpenFlow 1.0, we need multiple table support and MAC address wild card match support which are provided by OpenFlow version 1.1 and above or Nicira extensions; Floodlight does not support either of these and was hence discarded. Among the remaining, POX [6] was the controller of choice as the learning curve for it is low as it was designed for rapid prototyping and experimentation. It is written in Python and hence easy to read and write code. Since POX was chosen as controller, both the controller modules - our SDN app and global_switch_stats were also implemented using Python as they need to interact with POX which provides interface in Python.

9.3 Data structures used

- Graph : An associative array with key(node) as the dpid of the switch and value(links) as another associative array with key(src port) and value a list of 2 times - dpid of the connected switch and the dst port on that switch
- ARP Table : An associative array with IP address as the key and the corresponding PMAC as value
- All switches : stores as a set of the dpids of the switches
- AMAC -> PMAC : associative array with key the AMAC and value as PMAC. Same structure is used for PMAC -> AMAC table
- Assigned PMAC list : associative array with key as the dpid and value as an associative array with key as port on the switch and value as a string of 1s and 0s.

9.4 Communication mechanisms used and their structure

- Communication between the OpenFlow switches(our Mininet topology) and the POX controller is achieved using OpenFlow protocol. The types of messages that are exchanged and explanation about OpenFlow is defined in section 3.2.
- Openflow.discovery module provides the SDN app with topology information in the form of Link events. The link event has the following structure :
  - link : (src switch dpid, src switch port, dst switch dpid, dst switch port)
  - added : a boolean value indicating if this link was added
  - removed : a boolean value indicating if this link was removed(not added)
- The SDN app and POX controller communicate with each other using various OpenFlow events. Some of them are :
  - PacketIn - when a switch sends a packet to the controller
  - ConnectionUp - when a new switch has connected to the controller
  - PacketOut - sending a packet to a switch with some specified actions
  - Flowmod - a meesage to the switch to insert, modify or delete flow table entries.
- Internally, the communication between most of the components in the architecture is through global data that is resident inside the SDN app module namespace

9.5 Pseudocode

9.5.1 Main module

```
graph builder()
graph identifier()
insert forwarding()
define bcast semantics()
listen for arp and call arp preprocessor on receiving each arp packet in
```

9.5.2 Graph builder

```
add listener for connectionup event and on connectionup
    create node in the graph G
    save all the port info for this node
    add this switch to all switches set

add listener for link event and on link
    create a link between dpid1, port1 and dpid2 and port2 in G
```

9.5.3 Graph identifier

```
for each switch
    if the percentage of ports of this switch connected to other switches is
        less than user configured percentage
        add switch to edge switch
        host ports of this switch <- all its ports - all ports connected to
            switches
        for every port in this switch, initalize the default assigned pmac
            string

for each edge switch
    for each switch it is connected to
        add to aggregate switches
        mark this port as down port of this agg switch
core switches = all switches - edge switches - aggregate switches

for each aggregate switch
    up ports = all ports - down ports

```

```
logically remove all aggregate to core links in the graph G
mark all aggregate switches as unvisited
pod num <- 0
for each unvisited aggregate switch
    bfs(agg switch, pod num)
    incr pod num by 1

logically readd aggregate to core links in the graph G

bfs(agg switch, pod num)
    initialize queue q with agg switch
    pos <- 0
    while q not empty
        node <- q.pop()
        if not not visited
            mark it as visited
            switch pos of this node = (pod num, pos)
            incr pos by 1
            for each neighbour of node
                if neighbour not visited add to q
```

9.5.4 Insert forwarding

```
for each core switch
    for each pod
        if there are n(> 1) links to this pod
            add flow tables matching this pod num in the 1st 2 bytes and hash on
                the port num(4th byte) of the destination pmac and load balance
        else add a flow table entry to match this pod and forward to that one
            port as action

for each aggregate switch
    if there are multiple up links
        add flow tables entries to hash the uplinks based on the dest mac's port
             number
        else add flow table entry to output to that port

        for each down port
        add a flow table entry to forward to that port if the 3rd byte matches
            the pos of the switch this port is pointing to
        this flow is added with a greater priority than up link flow

```

```
for each aggregate switch
    if there are multiple up links
        add flow tables entries to hash the uplinks based on the dest mac's port
             number
        else add flow table entry to output to that port

    for each host port
        add a flow table entry to forward to that port if the 4th byte matches
            this host port
        this flow is added with a greater priority than up link flow
        add a flow table entry to forward to controller if an ARP message comes
            from this port
```

9.5.5 Define broadcast semantics

```
for each aggregate switch
    for each up port
        add flow table entry to match this port and output the packet to all
            down ports if dst mac is bcast mac

     for each down port
         add flow table entry to match this port and output the packet to one of
              the up ports if dst mac is bcast mac

for each edge switch
    for each up port
        add flow table entry to match this port and output the packet to all
            host ports if dst mac is bcast mac

     for each host port
         add flow table entry to match this port and output the packet to one of
              the up ports if dst mac is bcast mac

for each core switch
    if there is only one link to each pod, add flow table entry to flood if dst
        mac is bcast mac
    else, for each src port
        add flow table entry to not output to any of the ports of that are
            connected to the same pod as this src port and flow hash based on dst
             mac port number if there are multiple links to a pod
```

9.5.6 ARP Preprocessor

```
arp preprocessor(event)
    if src ip is known
        check and handle migration(event)
        arp responder()
    else if src ip is zero
        if dst ip is known
            arp responder()
        else
            latent host pos <- (event.dpid, event.port)
            edge broadcast(event.data, event.dpid, event.port)
    else
        pmac <- handle new host(actual mac, ip, switch dpid, port)
        listen for barrier reply on this switch and on barrier received, call
            handle barrier()
        barrier <- new barrier req
        data[barrier] = (event, pmac, ip)
        send barrier to switch

handle barrier(barrier)
    event, pmac, ip <- data[barrier]
    stop listening for barrier on this switch
    arp table[ip] <- pmac
    actual to pmac[actual mac] <- pmac
    pmac to actual[pmac] <- actual mac
    arp responder()

handle new host(actual mac, ip, dpid, port)
    pmac <- assign pmac(dpid, port)
    send translation entries to switch dpid to translate actual mac to pmac in
        src and pmac to actual mac in destination

assign pmac(dpid, port)
    get pod and pos of switch for table
    get assigned pmac string of this switch and port
    pmac <- a free pmac not yet assigned
    mark this pmac as assigned
    return pmac

check and handle migration(event)
    get src ip and actual mac from event
    pmac <- actual to pmac[actual mac]
    from pmac identify switch dpid and port it belongs to
    check if it is the same as the switch and port this event came from
    if it is, no migration
    else
        new pmac <- handle new host(actual mac, src ip, event.dpid, event.port )
        send barrier to this switch

        add flow table entry in old switch to rewrite the dst mac to new pmc if
            its receives a packet with dst mac as old pmac
        remove old translation entries from old switch
        mark the old pmac as unassigned after arp cache timeout which is config
            by user
        remove old pmac from host mac from tables
        arp table[src ip] <- new pmac
        actual to pmac[actual mac] <- new pmac
        pmac to actual[new pmac] <- actual pmac

edge broadcast(data, dpid, port)
    for each edge switch
        if edge switch dpid is equal to dpid
            send a packetout with the data given to output to all host ports
                except port
        else
            send a packetout with the data given to output to all host ports
```

9.5.7 ARP Responder

```
arp responder(event)
    fromt the event, extract src ip, src mac, dst ip, dst mac
    if src ip is same as dst ip
        return
    if arp pkt is a request and dst ip is in arp table
        pmac <- arp table[dst ip]
        construct an arp reply with src mac as pmac
        sent this packet as packet out to event.dpid and output to event.port
    else if dst ip not in arp table
        pmac <- actual to pmac[src mac]
        change arp request src mac to pmac
        edge broadcast(arp request, event.dpid, event.port)
    else if its a reply
        pmac <- actual to pmac[src mac]
        change src mac in reply pkt to pmac
        if dst ip is zero
            dpid, port <- latent host pos[dst mac]
            send the changed reply to the switch dpid and output to port
        else
            dpid, port <- pos(dst mac)
            send the changed reply to the switch dpid and output to port
```

10. Testing

10.1 Functional Testing

Test Cases	**Funtional Requirement being tested**	**Expected Output**	**Actual Output**	**Pass / Fail**
For a newly setup network, two hosts are able to ping each other	1, 2	Ping succeeds	As expected	PASS
New host pinging existing host	1, 2	Ping succeeds	As expected	PASS
Existing pinging new host	1, 2	Ping succeeds	As expected	PASS
Host sends a broadcast packet	4	Bubble and burst broadcast semantics are followed, all hosts receive packet and no flooding	As expected	PASS
One host taking IP of another(Virtual IP)	5	All the data packets should to that IP should reach new host	As expected	PASS

We were able to test and meet all our functional requirements, except for testing Virtual Machine(VM) Migration. We use Virtualbox and Vmware Player to run our virtual machines. Both of them, run at application level and as a result, the host that the VM is running on replaces the VM's MAC address with its own MAC address in the packets going from the VM into the network. Due to this, the packets appear as if they had originated from the physical host. This would not happen in real data centers as in the real data centers hypervisors are used to run the VMs. These hypervisors run at the kernel level and do not modify the MAC addresses of packets going to and from the network. Our code is designed for Virtual Machines running on hypervisors ad we did not have hypervisors to test VM migration. But, our implementation handles VM migration and Virtual IP in the same manner and Virtual IP is tested and working.

10.2 Performance Testing

Comparative testing was done to see performance difference between using our SDN app for forwarding and using the traditional layer 2 learning switch component for forwarding with spanning tree running.
POX provides a module called 'forwarding.l2_learning' which makes the OpenFlow switches behave as layer 2 learning switches.
Since the topology has redundant links and loops, spanning tree protocol has to be run along with the l2 learning component to prevent broadcast storms. POX provides a module 'openflow.spanning_tree' to run a centralized spanning tree.
Both our SDN app and the spanning tree component require the openflow.discovry module to build a map of the network.
To measure the traffic within the network, a module was developed called 'global_switch_stats' that queries all the switches for the traffic that flowed through them and aggregates the stats.
The two different setups are :

1. POX running with openflow.discovery and our SDN app module
2. POX running with openflow.discovery, openflow.spanning_tree, and forwarding.l2_learning component

For generating traffic, a module 'genTraffic' was developed which uses the Mininet API to generate lot of ping traffic parallely and collect performance metrics like RTT, packets sent , received and dropped from the result of the generated traffic.
Mininet was used to emulate the network. Two topology modules were developed in mininet to create custom data center topologies. One is for creating a fat tree topology and the other for creating a generic 3 tier multi - rooted topology. The tests were run on these topologies with the two different POX setups explained above.

The interfaces for the modules mentioned above are :

- **gstats()** - the pox module to aggregate switch statistics does not take any parameters.
- **genTraffic** - the module to generate traffic take these parameters :
    1. interval - interval between two ping commands
    2. duration - for how long to send the ping commands
    3. count - the number of packets to send in each ping command
    4. output file - the file to write the generated output into
- **genTreeTopo** - the custom topology module to create a generic multi rooted 3 tier tree takes these parameters :
    1. num of pods

2. number of core switches
3. number of aggregate switches per pod
4. number of edge switches per pod
5. number of hosts per edge switch

10.2.1 Effect on Network Traffic

Gentraffic module was instructed to send the same number of ping requests on both the setups and amount of traffic that results within the network due to these pings was measured using the global_switch_stats POX module that was developed. This was repeated multiple times with varying number of hosts within the network.
As can be seen in fig 10.1, there is a huge gap in the number of packets received by the switches with SDN app setup and without. This huge gap is because the broadcast traffic is drastically reduced. Unknown unicast is completely eliminated as the flow tables have entries to forward any MAC address in the entire MAC address space due to the hierarchical pseudo MAC structure. ARP broadcast is reduced because there is is only one ARP broadcast per one new IP and all ARP request are handled by the SDN app. On the other hand in traditional L2 learning setup, there is an ARP broadcast everytime a host does not have an IP in its ARP table. If you observe fig 10.2, you can see that the transmitted packets increase exponentially with number of hosts. This is again due to broadcast traffic. When a packet is received with broadcast MAC address, then the switch floods that packet out. Consider a switch with n ports which are connected to n other switches. When this switch receives a broadcast packet, it will send the packet out of n-1 ports and those n-1 switches further repeat this. This is exponential and hence the reason for why the transmitted packets grow exponential with size of hosts as increase in the size of network implies increase in the number of ARP broadcasts and unknown unicasts.
Hence the functional requirement 3, of reducing broadcast traffic, is met.

Number of Hosts	**Rx Packets (With SDN App)**	**Rx Packets (Without SDN App)**
32	139854	361484
160	136139	283540
320	132374	289968
480	132859	320929
640	129020	324767
800	127355	441912

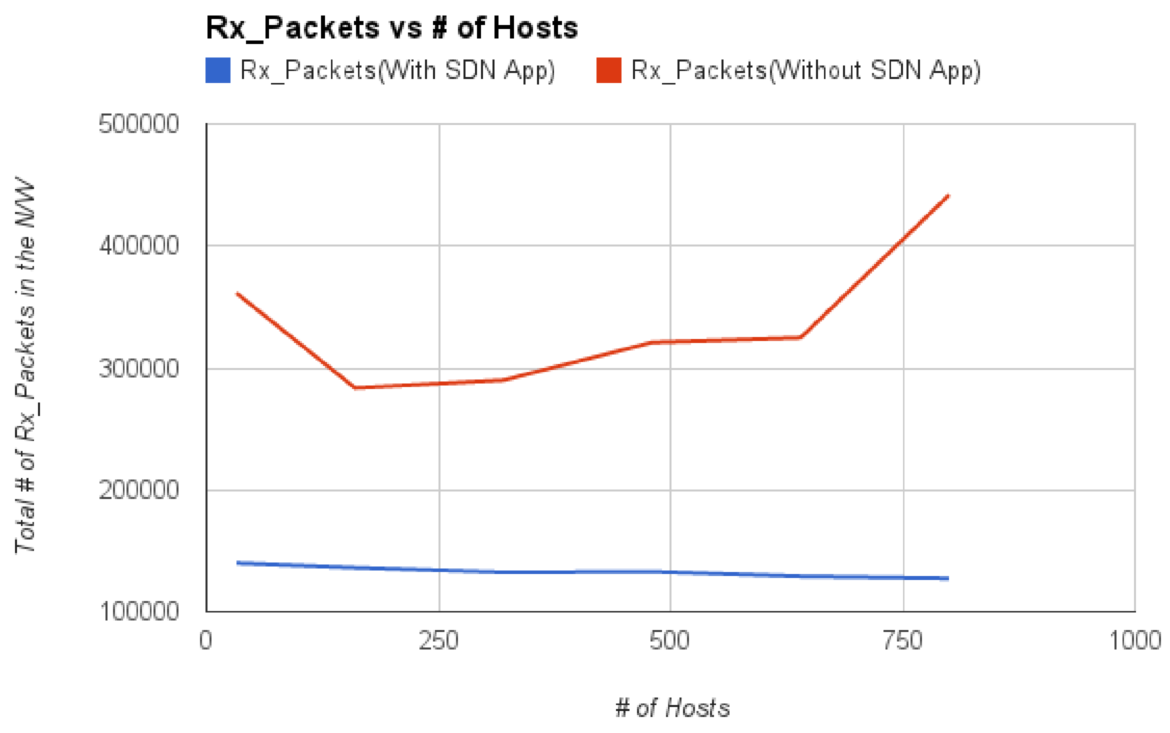


Figure 10.1: Test Results of the variation of Number of Packets Received on ping - With and without our application

Number of Hosts	**Tx Packets (With SDN App)**	**Tx Packets (Without SDN App)**
32	141096	418280
160	150056	988038
320	182966	1841724
480	251008	3163085
640	330640	4383145
800	449645	8188230

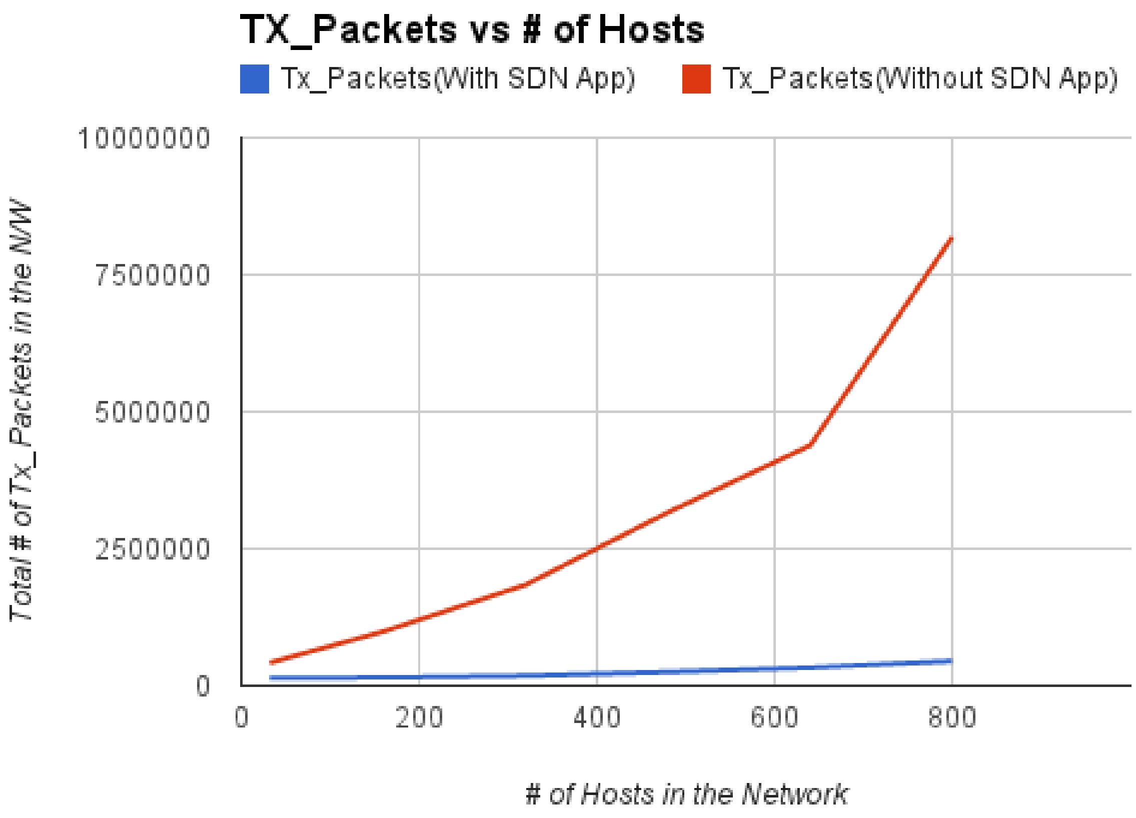


Figure 10.2: Test Results of the variation of Number of Packets Transmitted on ping - With and without our application

10.2.2 Impact on the speed by measuring Average RTT

Gentraffic module was instructed to send the same number of ping requests on both the setups and the average RTT for the ping response was measured. This was repeated multiple times with varying number of hosts within the network. As can be seen from the graph and table, the average RTT with our setup is much faster and grows much slower with increase in number of hosts. The reason why our setup is faster is because:

1. **Highly reduced broadcast traffic** : As can be seen in 10.2.1, the amount of traffic within the network is reduced drastically. Because of this, the links are not choked by unnecessary packets and ping packets can flow through freely.

2. **Faster ARP responses** : In traditional L2 network, the ARP request has to reach the target host and back again. These are multiple hops. With our app, the ARPs are intercepted at edge switch itself and reply sent back to edge switch. This is fast if the control network is fast.

3. **Compact flow table size** : The number of entries in the switch flow table is very compact as the forwarding entries are hierarchical and the size depends on the number the network topology. Due to this, the lookup is very fast. On the other hand, the switches in traditional L2 have to maintain a flow table entry for each host. Here the size of the table is dependent on the number of hosts in the network and grows with it. Hence, the look up is slow. Also, they entries have to be inserted by the controller. This insertion takes time. And this has to be done repeatedly as there is a timeout associated for each entry. This overhead is not present with our app as the forwarding tables are computed initially based on the hierarchy and inserted only once.

Number of Hosts	**Average RTT (With SDN App)**	**Average RTT (Without SDN App)**
32	4.4608298	51.4675272
160	24.065292	146.6146174
320	21.511374	317.302709
480	34.4747476	722.4307914
640	65.3359544	972.3884374
800	94.5180354	1729.643691

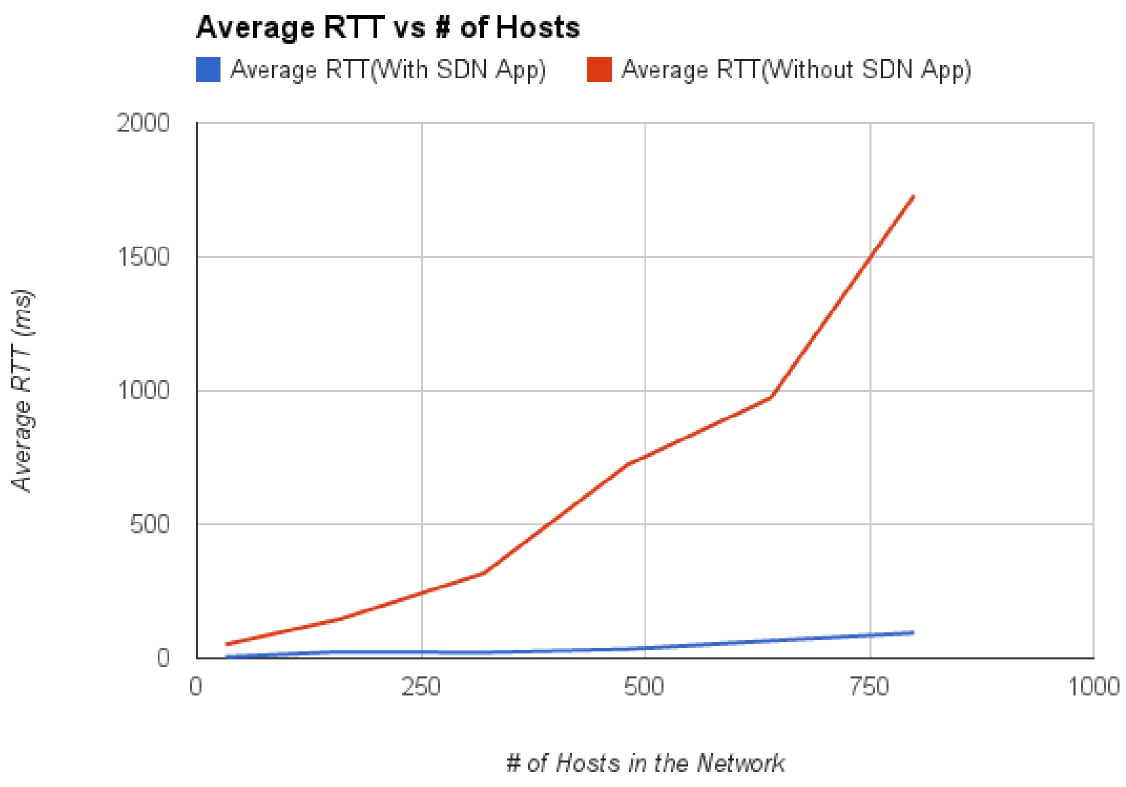


Figure 10.3: Test Results of the variation of Average RTT on ping - With and without our application

11. Conclusion

We were successfully able to make an SDN Application which reduces broadcast traffic in Data Centers. This application has shown promising results so far, with significant improvements in the overall network traffic in Data Centers as well the response time, improving the network performance and efficiency. We were also able to leverage the redundancy in the network topology inspite of a Layer 2 network by designing custom broadcast semantics, removing the need to run a spanning tree protocol at the Layer 2 level. This project supports emerging Data Center requirements of Virtual Machine Migration and Virtual IP successfully.

We also plan to **publish the work** done in this project along with the successful results that were obtained.

12. Future Scope

Software Defined Networking is an emerging paradigm in computer networking. With frequent enhancements in OpenFlow as well as the development tools, there is a constant scope for improvement as well as providing more granular control over the networks.

1. Future work on this project can be extended by making the network more robust by supporting Switch and Link Failures.

2. This project's scope was limited to SDN networks with a single controller, future work can be carried out to extend this to a system with multiple controllers.

3. Native multicast can be implemented as well, with reduced broadcast traffic compared to traditional networks.

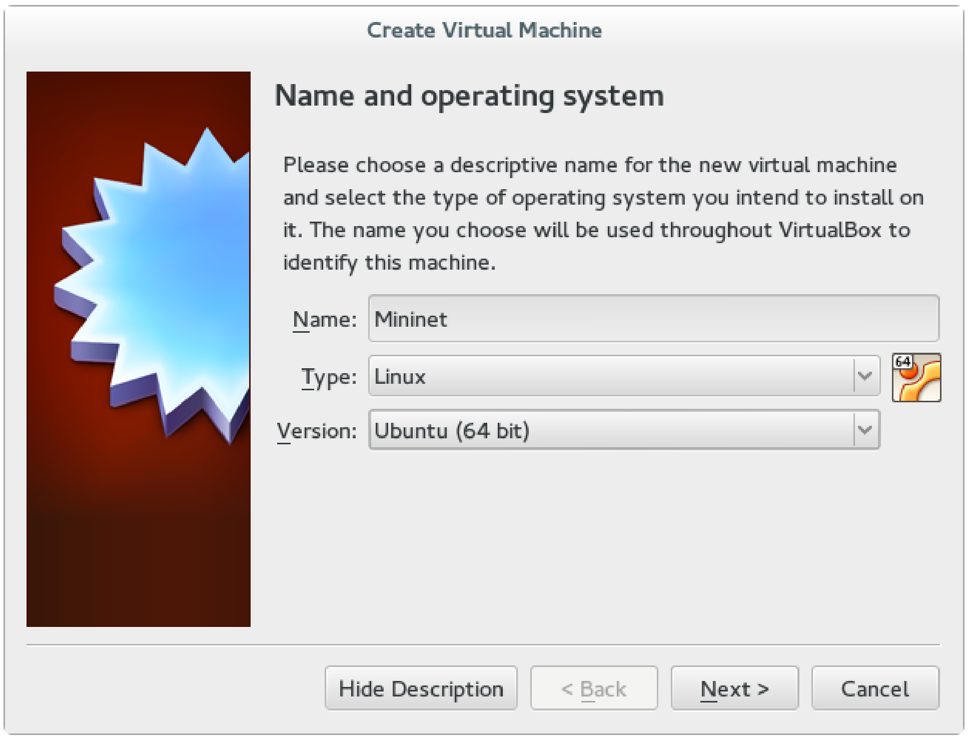
4. POX, the Controller used, is ideal only for development purposes, and fails to scale to support the traffic requirements of Data Centers. This design and implementation can hence be ported to a controller that is designed to work in production, like Flood Light, etc.

5. The hierarchical addressing can be modified to accommodate a scenario where multiple Data Centers might work in conjunction with each other.

6. Also, the design gives equal addressing space to all pods in the network, this can be made dynamic, so that the address space distribution for Pseudo MAC addresses is done with respect to the size of the pods.

User Manual

1. Download latest Mininet VM Image from <http://mininet.org/download/>
2. Extract the image.
3. Download Virtual Box for your system OS from <https://www.virtualbox.org/wiki/Downloads>
4. Install Virtualbox on your system.
5. Open Virtualbox And Choose New :



Create Virtual Machine

Name and operating system

Please choose a descriptive name for the new virtual machine and select the type of operating system you intend to install on it. The name you choose will be used throughout VirtualBox to identify this machine.

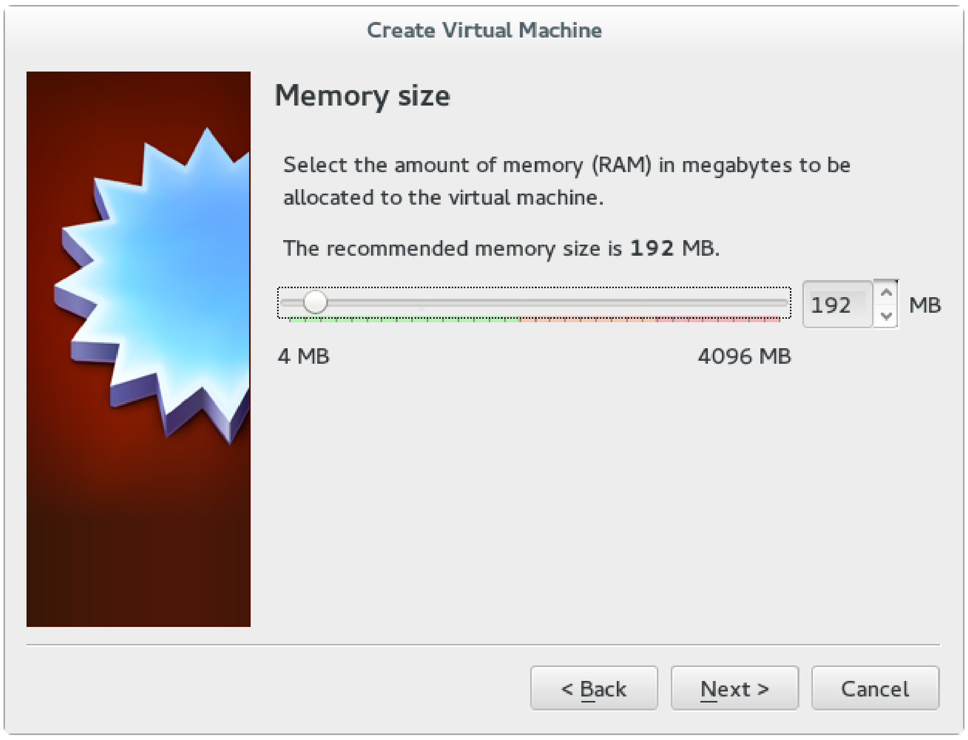
Name: Mininet

Type: Linux

Version: Ubuntu (64 bit)

Figure 12.1: Virtual Machine Setup

6. Give an appropriate name and the Type is Linux And Version is Ubuntu 64bit
7. Next, Choose an Appropriate Amount Of Ram (Half your system RAM) :



Memory size

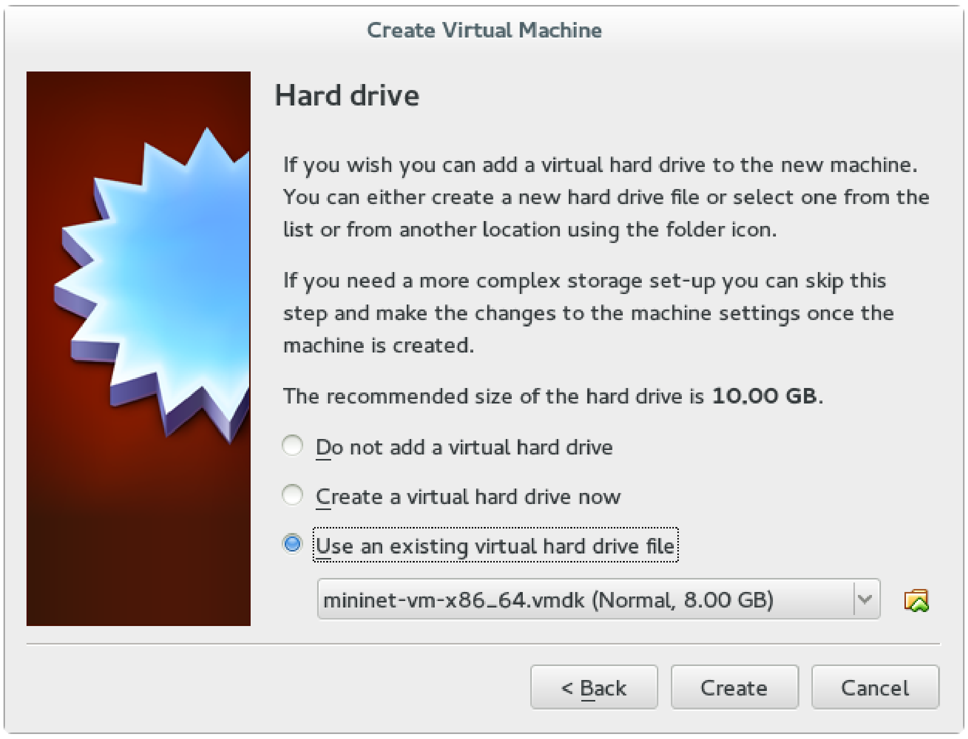
Select the amount of memory (RAM) in megabytes to be allocated to the virtual machine.

The recommended memory size is **192** MB.

192 MB

4 MB — 4096 MB

8. Next, Choose Use an existing virtual hard disk file :



Hard drive

If you wish you can add a virtual hard drive to the new machine. You can either create a new hard drive file or select one from the list or from another location using the folder icon.

If you need a more complex storage set-up you can skip this step and make the changes to the machine settings once the machine is created.

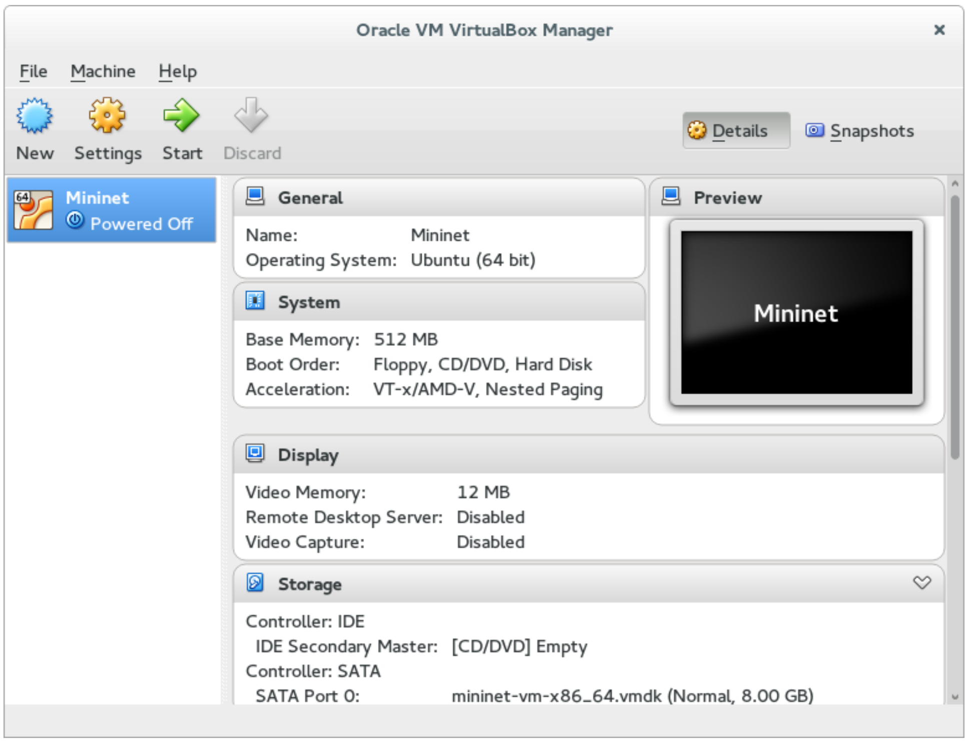
The recommended size of the hard drive is **10.00 GB**.

Do not add a virtual hard drive [ ]
Create a virtual hard drive now [ ]
[✓ Use an existing virtual hard drive file]

mininet-vm-x86_64.vmdk (Normal, 8.00 GB)

9. Browse to folder where you extracted mininet and choose the vmdk file

10. The VM is now ready!



General
Name: Mininet
Operating System: Ubuntu (64 bit)

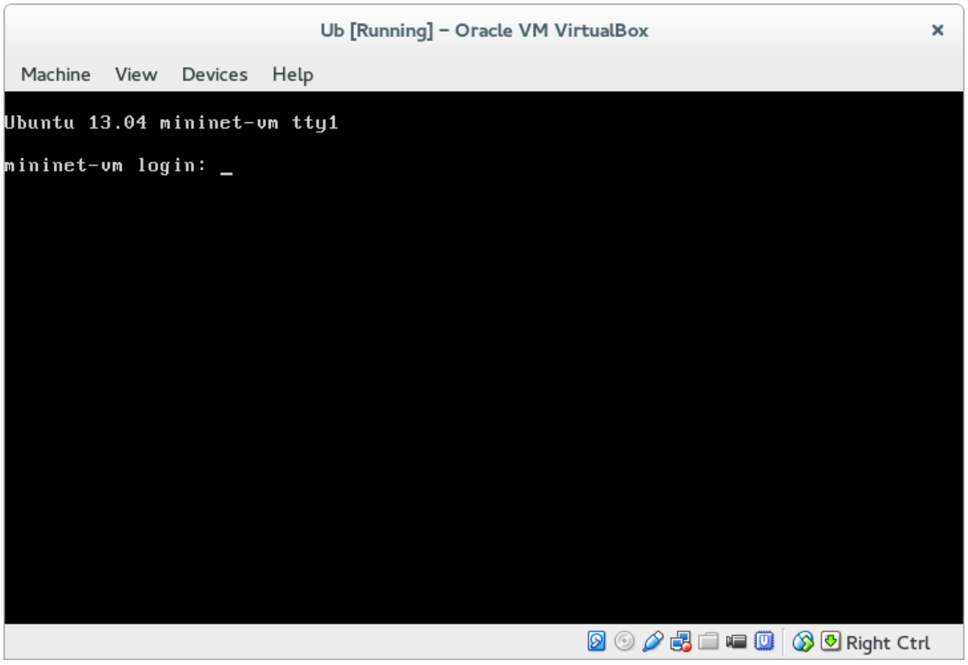
System
Base Memory: 512 MB
Boot Order: Floppy, CD/DVD, Hard Disk
Acceleration: VT-x/AMD-V, Nested Paging

Display
Video Memory: 12 MB
Remote Desktop Server: Disabled
Video Capture: Disabled

Storage
Controller: IDE
IDE Secondary Master: [CD/DVD] Empty
Controller: SATA
SATA Port 0: mininet-vm-x86_64.vmdk (Normal, 8.00 GB)

Check this link <`http://mininet.org/vm-setup-notes/`> for advanced configurations.

11. Click the Start button to start the VM, you will see this screen :

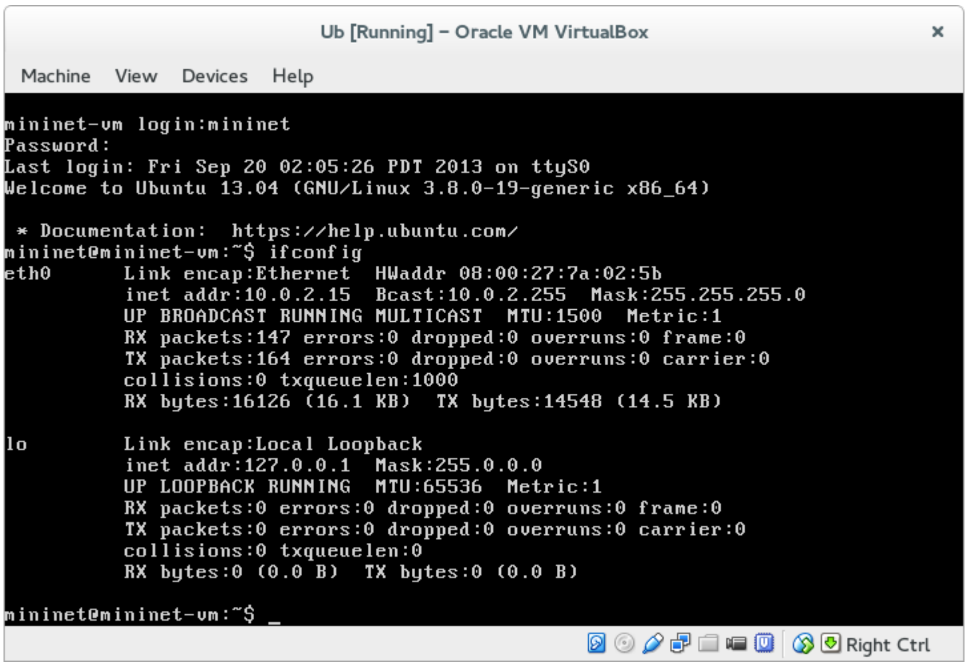

Ub [Running] – Oracle VM VirtualBox
Machine View Devices Help
Right Ctrl

```
Ubuntu 13.04 mininet-vm tty1

mininet-vm login: _
```

12. Login to the system using user and password as mininet

13. Run *ifconfig* to get the ip of the VM :



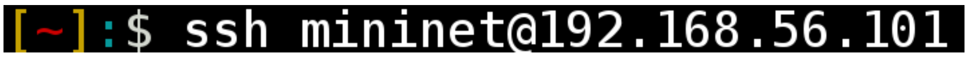
```
mininet-vm login:mininet
Password:
Last login: Fri Sep 20 02:05:26 PDT 2013 on ttyS0
Welcome to Ubuntu 13.04 (GNU/Linux 3.8.0-19-generic x86_64)

 * Documentation:  https://help.ubuntu.com/
mininet@mininet-vm:~$ ifconfig
eth0      Link encap:Ethernet  HWaddr 08:00:27:7a:02:5b
          inet addr:10.0.2.15  Bcast:10.0.2.255  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:147 errors:0 dropped:0 overruns:0 frame:0
          TX packets:164 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:16126 (16.1 KB)  TX bytes:14548 (14.5 KB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

mininet@mininet-vm:~$ _
```

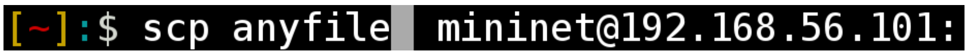
14. Open terminal on the host machine connect to the VM by ssh using the command:
    – ssh mininet@<ip from ifconfig vm>
    – type password when prompted



```
[~]:$ ssh mininet@192.168.56.101
```

15. In a similar manner, you can open multiple terminals and connect to the vm.

16. Copy the custom modules to the vm using the scp tool :
    – scp path-to-py-file mininet@<ip from ifconfig vm>
    – type password when prompted



```
[~]:$ scp anyfile  mininet@192.168.56.101:
```

17. Move the codes to the appropriate folders and recompile mininet by running this command:
    – /mininet/util/install.sh -n

    Path for files :

    Mininet core files :

    /mininet/mininet/node.py

    /mininet/mininet/cli.py

    Traffic Generator :

    /mininet/mininet/genTraffic.py

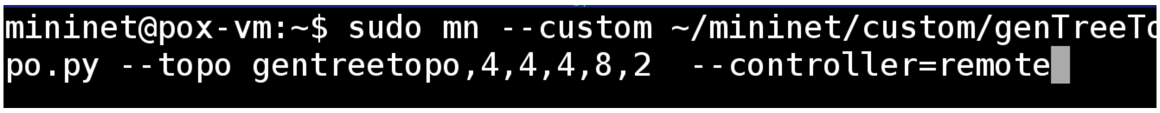
    Data Center Topology files :

    /mininet/custom/fatTreeTopo.py

    /mininet/custom/genTreeTopo.py

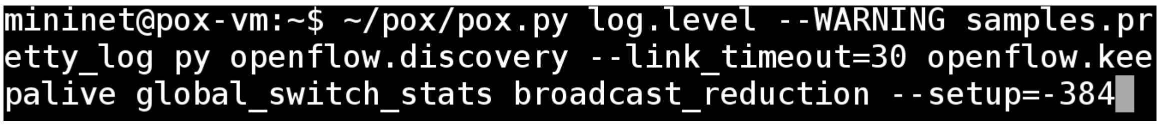
Custom Controller module :
/pox/ext/broadcast_reduction.py
POX module for measuring switch traffic :
/pox/ext/global_switch_stats.py

18. In one terminal start Mininet with a generic multi-rooted 3-tier topology, using the command:



```
mininet@pox-vm:~$ sudo mn --custom ~/mininet/custom/genTreeTo
po.py --topo gentreetopo,4,4,4,8,2  --controller=remote
```

19. In another terminal, start the pox controller with our SDN application (module):



```
mininet@pox-vm:~$ ~/pox/pox.py log.level --WARNING samples.pr
etty_log py openflow.discovery --link_timeout=30 openflow.kee
palive global_switch_stats broadcast_reduction --setup=-384
```

20. Run *pingall* in the mininet cli to test host reachability.

Bibliography

[1] ***"PortLand:A Scalable Fault-Tolerant Layer 2 Data Center Network Fabric"*** SIGCOMM'09, August 17-21, 2009, Barcelona, Spain.

[2] ***"A Scalable, Commodity Data Center Network Architecture"*** SIGCOMM'08, August 17-21, 2008, Seattle, Washington, USA.

[3] ***"Software Defined Networking: The New Norm for Networks"*** ONF, White Paper, April 13, 2012.

[4] ***"OpenFlow Switch Specification"*** Version 1.0.0 (Wire Protocol 0x01), ONF, December 31, 2009.

[5] [RFC6820], ***"Address Resolution Problems in Large Data Center Networks"*** RFC 6820, January 2013.

[6] POX Wiki: `https://openflow.stanford.edu/display/ONL/POX+Wiki`

[7] Mininet Walkthrough: `http://mininet.org/walkthrough/`

[8] Linux Namespaces: http://blog.scottlowe.org/2013/09/04/introducing-linux-network-namespaces

[9] Topology Discovery in SDN: http://vlkan.com/blog/post/2013/08/06/sdn-discovery