THE UNIVERSITY OF
WAIKATO
Te Whare Wānanga o Waikato

Improving the Security and Efficiency of Network Clients Using OpenFlow

Adam Coxhead

This report is submitted in partial fulfillment of the requirements for the degree of Bachelor of Computing and Mathematical Sciences with Honours (BCMS(Hons)) at The University of Waikato.

COMP520-13C (HAM)

© 2013 Adam Coxhead

Abstract

With the complexity and scale of networks these days some of the earlier protocols have started to show their age, presenting vulnerabilities and inefficiencies in the way they run. As a result research and solutions are being put in place to address certain aspects of these protocols, be this extensions to the protocol itself or extensions to the devices within the network. OpenFlow is a open network protocol that is an implementation of a relatively new concept in computer network design, Software Defined Networking (SDN). It allows for the separation of the control plane of a switch and enables it to be moved into an application running in software. This results in far more software oriented approach to network design and control of switches. This project aims to use this new flexibility that OpenFlow provides to explore ways in which the weaknesses apparent with certain protocols in todays networking environment in order to improve their efficiency and security. It focuses on reducing the need for the broadcast of Address Resolution Protocol (ARP) requests as well as looking into an alternative method to the Spanning Tree Protocol (STP).

Acknowledgements

Firstly I would like to thank my supervisor Dr. Richard Nelson for the support and guidance that was given throughout the project. I would also like to thank the following members of the WAND group, Shane Alcock, Brendon Jones and Brad Cowie for answering questions as well as their help in proof reading this report. Finally I would like to thank Christopher Lorier and Joe Stringer for their support when I was learning my way around OpenFlow and the Ryu framework.

Contents

Abstract ii

Acknowledgements iii

1 Introduction 1

2 Background 3

2.1 Software Defined Networking and OpenFlow 3

2.2 Address Resolution Protocol 6

2.3 Spanning Tree Protocol 8

2.4 Implementation Tools 10

2.4.1 Open vSwitch 10

2.4.2 Ryu 10

3 Investigation 11

3.1 Address Resolution Protocol 11

3.1.1 Current Extensions and Implementations 11

3.1.2 OpenFlow Solutions 12

3.2 STP 13

3.2.1 STP Extensions and Successors 13

3.2.2 OpenFlow Solutions 15

4 Implementation 17

4.1 Address Resolution Protocol 17

4.1.1 Controller Redirection 18

4.1.2 ARP Proxy OpenFlow 21

4.2 Loop Prevention and Equal Cost Multipath Forwarding . . . 21

4.2.1 Loop prevention Module 24

4.2.2 Equal Cost Multipath Forwarding 26

4.2.3 Shortest Path Algorithm 27

4.2.4 MAC Learning and Path Choice 29
4.2.5 Flow Distribution 29
4.2.6 Link State Changes 30

5 Evaluation and Discussion 31
5.1 ARP 31
5.1.1 Controller Redirection 31
5.1.2 ARP Proxy Controller 33
5.2 Loop Prevention and ECMF 35
5.2.1 Evaluation 36
5.2.2 Comparisons 40
5.3 Project and OpenFlow Limitations 44
5.3.1 Design Choice Limitations 44
5.3.2 OpenFlow Limitations 45

6 Conclusions 47
6.1 Contributions Made 47
6.2 Future work 48
6.3 Conclusion 49

References 50

A ARP Redirection and Proxy in OpenFlow 53
A.1 ARP response script 53
A.2 ARP Proxy controller 54

B Loop prevention and ECMF 56
B.1 Loop prevention module - Tree calculation 56
B.2 ECMF - shortest paths between pair of switches calculation . 57

List of Figures

2.1 OpenFlow 1.0 Protocol and Switch 5
2.2 Openflow 1.2 Multi-table Switch 6
2.3 Simple STP Configured Network 9

4.1 ARP behaviour comparison 18
4.2 DHCP and ARP responder host make up 20
4.3 Controller composition 23
4.4 Link disabling comparison - STP vs Loop prevention 25
4.5 Packet forwarding comparison - STP vs ECMF 27

5.1 Loop prevention module runtime - Ring topology 37
5.2 Loop prevention module runtime - Partial mesh 38
5.3 ECMF module runtime - Ring topology 39
5.4 ECMF module runtime - Partial mesh 39

List of Tables

5.1 Comparisons between STP and OpenFlow approaches 42
5.2 Comparisons between Projects approach and STP successors . . . 43

List of Acronyms

ARP Address Resolution Protocol

ASIC Application-Specific Integrated Circuit

BPDU Bridge Protocol Data Unit

DAI Dynamic ARP Inspection

DHCP Dynamic Host Configuration Protocol

ECMF Equal Cost Multipath Forwarding

IP Internet Protocol

IPv4 Internet Protocol Version 4

IS-IS Intermediate System to Intermediate System

LLDP Link Layer Discovery Packet

SDN Software Defined Networking

MAC Media Access Control

MSTP Multiple Spanning Tree Protocol

OMAPI Object Management Application Programming Interface

OVS Open vSwitch

RSTP Rapid Spanning Tree Protocol

SPB Shortest Path Bridging

STP Spanning Tree Protocol

TRILL Transparent Interconnection of Lots of Links

TTL Time To Live

VSTP VLAN Spanning Tree Protocol

Chapter 1

Introduction

With the scale and complexity of networks growing rapidly over the last number of years, some of the flaws with existing protocols have become more apparent due to the higher demand placed upon them. As a result solutions improving both the security and efficiency of such protocols in order to tailor them more to the requirements of existing networks are much desired. With the introduction of Software Defined Networking (SDN) [4] network researchers have been presented with a new level of flexibility and control in the design of networks, along with new unique outlook on what switches can be made to do. This is achieved through SDN defining a way in which the control plane of a switch could be decoupled from the data plane, allowing for it to be moved into an application running on a remote host. OpenFlow[22] is one of the most popular realisations of SDN allowing for this separation to be achieved, presenting network researchers a new method in which to explore network design.

The aim of this project is to use the view OpenFlow provides to explore the ways it can potentially be used to to address some of the weaknesses apparent in existing protocols. It focuses on two main protocols used within current networks, these are the Address Resolution Protocol (ARP) and the Spanning Tree Protocol (STP). This project explores ways in which the broadcasting of ARP packets can be reduced or eliminated though the use of an OpenFlow enabled network, resulting in the reduction of processing overhead on the network as a whole, as well as lowering the potential for malicious packets to be sent in response to an ARP request. This project also aims to create an alternative to STP using the logically centralised and controllable view of a network provided by OpenFlow. It aims to introduce both load balancing across the network as well as shortest path forwarding into this alternative,

giving it much more desirable features than STP. The approaches taken by this project are restricted to those that are transparent to the end hosts on the network i.e. the end hosts themselves require no modification to their knowledge of the target protocol in order to continue functioning as normal.

This report first introduces the background and core workings of ARP and STP in Chapter 2. In Chapter 3, some of the current solutions taken by network device vendors and researches alike to address some of the weaknesses that have become apparent in these protocols have been discussed. Chapter 4 presents ways in which these same weaknesses could be addressed in an OpenFlow enabled network by presenting both OpenFlow inspired network designs and the constructed OpenFlow controllers. Finally, Chapter 5 is the evaluation of these approaches by comparing them to the existing techniques, as well as discussing how they could be deployed and what the limitations of their deployment may be.

Chapter 2

Background

There are a number of concepts and protocols important to the development of the controllers for this project, with the idea of SDN and OpenFlow being at the forefront of this. The concept of SDN and its practical application is described in Section 2.1. This concept crucial as it forms the basis from which this project is built. Following this, Section 2.2 addresses the ARP and gives its general background and behaviour. Section 2.3 introduces STP, again giving its background and behaviour. Lastly, Section 2.4 describes the tools used to both implement and test the solutions found.

2.1 Software Defined Networking and OpenFlow

Network switches are comprised of two planes, the data plane, which is responsible for the forwarding of packets and the control plane, which is responsible for making the more complex forwarding decisions. On a traditional switch these two planes are tightly coupled and exist within the same unit. SDN is the concept of abstracting or separating the control plane of a switch from the data plane. This allows for this control plane to be moved into a software application on a remote machine. SDN provides a control plane at a higher level of abstraction then the traditional approach i.e. software can be written to determine the configurations of the switches being controlled. The name given to this remote application is the **controller**. According to [4] SDN enables the design of highly scalable and flexible networks that are readily adaptable to to the needs of the network administrator. SDN enables a new, more software oriented and programmatic approach to the design and building of new networks. As well as a higher level, more open view of the complex forwarding.

This is achieved because the controller is now running in a remote host and in software. As a result it is not bound by the same limitations of hardware. The controllers function can then be modified and extended easily without the need of changing the Application-Specific Integrated Circuit (ASIC)'s or existing hardware of the switch.

OpenFlow is a realisation of SDN, it achieves the seperation of planes outlined by defining a protocol by which the separated control plane (controller) can communicate with the data plane. OpenFlow also provides a switch specification allowing for the configuration of the switch via the protocol. It was originally intended as a way for researchers to run experimental protocols in existing networks by allowing the separation of the experimental traffic from the production network traffic [22]. However as shown by [20], the advantages of OpenFlow may lead to uses beyond research. The main idea behind OpenFlow is to provide an open API for the configuration of a switch regardless of the vendor or underlying hardware. It allows for a controller to be logically centralised in a network, providing a basis on which new approaches can be taken in the configuration of switches to improve upon features of existing networks.

Figure 2.1 presents a visual representation of an OpenFlow switch. The switch consists of three main parts, a flow table, a secure channel and finally the ability to speak the OpenFlow protocol. The flow table is the part of the switch that is used to determine a packet's destination and decides how the switch forwards it on. The flow table is constructed from flow entries, each entry firstly contains a rule (matcher) used to determine whether a packet pertains to a particular flow. In a simple case, a rule may be a match on both the source and destination Media Access Control (MAC) addresses held within an Ethernet header. In this case any packet containing the same source and destination address would belong to the same flow and as such handled in the same manner. Secondly each entry also contains a set of actions to apply when a packet matches a rule. In a basic case this action can be as simple as forwarding the packet out of a particular port of the switch. Lastly each flow entry has a set of statistics, these statistics keep track of information like how many packets have matched a particular rule or how long the entry has existed. In this particular case a "flow" is defined as the set of packets that match a particular rule. Any flow that does not match the currently established rules held within the flow table of the switch can then be sent to the controller,

which in turn decides what to do with it. The OpenFlow protocol allows a controller to insert, modify and remove these flow table entries allowing for the dynamic reconfiguration of switches.

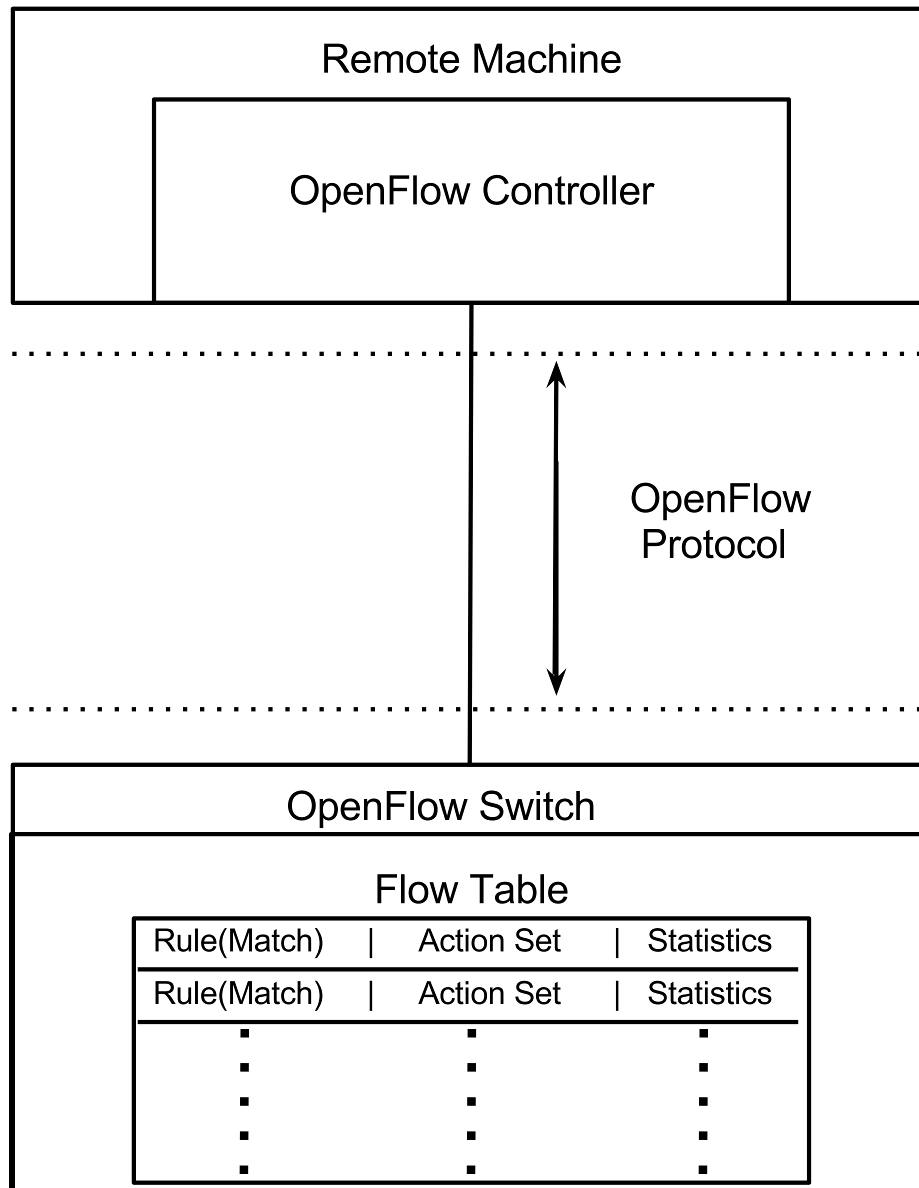


Figure 2.1: OpenFlow 1.0 Protocol and Switch

The specification for OpenFlow has progressed through four versions to this date[9][10][11][12]. OpenFlow specification 1.0 was the initial specification and it brought forward the core aspects of OpenFlow. This specification is still the most common and widely supported version of OpenFlow. From OpenFlow

version 1.1 onwards the biggest change with relation to this project has been the introduction of multiple flow tables. This can be used to reduce the number of rules that need to be added to a switch as well as allowing for more complex forwarding actions. The single table approach described in the initial specification of OpenFlow requires a unique entry for each pair of source and destination MAC addresses. Therefore resulting in a flow entry complexity of O(n^2) with respect to the number of hosts on a network. The multi-tabled approach of OpenFlow version 1.2 allows for a table specific to the set of source MAC addresses and another for the set of destination MAC addresses. Thus reducing the complexity outlined down to O($2n$). OpenFlow version 1.2 is the OpenFlow version that has been used for the controllers in this report.

Another significant change contained within the OpenFlow 1.2 specification, in relation to this project, is the idea of having instructions as opposed to actions. An instruction is a wrapper around an action. An instruction can apply a given set of actions or it can be a pointer to another table with each table defining the own flow behaviour. The interaction between tables is illustrated in Figure 2.2 and shows a Switch under the OpenFlow 1.2 specification.

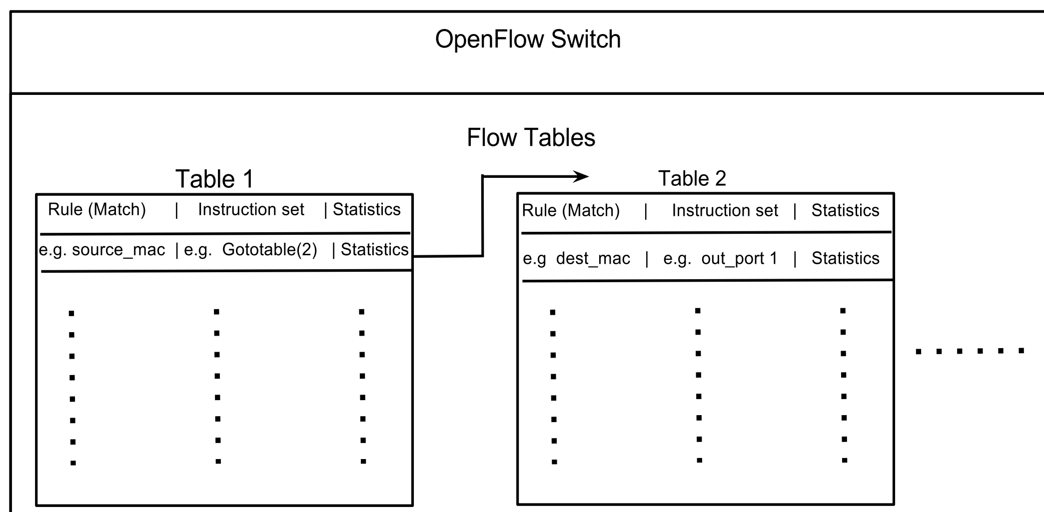


Figure 2.2: Openflow 1.2 Multi-table Switch

2.2 Address Resolution Protocol

ARP[26] is the protocol by which hosts discover each other on a local network. It allows for the mapping of Internet Protocol Version 4 (IPv4) addresses to an Ethernet MAC address. ARP's core functionality consists of two types of

packets, ARP requests and ARP responses.

The ARP request is a packet that is broadcast to the entire local network. It is effectively a “who is” packet, its purpose is to resolve the MAC address for a particular IPv4 address. The Ethernet frame of the ARP request has the MAC address of the requesting host as the source and the special MAC address **ff:ff:ff:ff:ff:ff** as the destination. This destination indicates to all the switches in the network to forward the packet out every port except the port it received it on. Every host in the network will see a ARP request packet and check its payload to see if it refers to them. The ARP response packet is sent from the target host to the original host in response to the request. It lets the original host know the Ethernet MAC address of the target so that they are able to communicate with each other over the local network. Each host on the network keeps a table or cache of the responses it has received and refers to this cache so it does not have to broadcast every time it wants to communicate. Entries in this cache have a timeout associated with them and when they become invalid the request stage is performed again.

The problem with ARP is a combination of the naive way that ARP handles responses and the broadcast of its request. The need for the request to broadcast to be the entire network has some downsides. One such issue is that anyone can see these requests. This presents a problem if there are malicious hosts on the network, as malicious hosts can generate a forged response allowing it to assume the identity of the requested machine. The act of broadcasting also has its own effects on the network such as introducing some processing overhead at both end hosts and intermediary networking devices. For intermediary devices this overhead will be incurred because each one has to forward these broadcast packets out all enabled ports, requiring the device to create duplications of the packet. In the case of the end hosts, this processing overhead is a result of having to process each ARP request in case the host is required to respond.

This project looks to reduce or remove the need for these ARP requests to be broadcast. It also aims to determine the impact on security of the approaches taken.

2.3 Spanning Tree Protocol

Loops in a modern switched network can seriously degrade quality and performance of the entire network. This is due to the fact that broadcast packets get forwarded out every port of a switch. A loop will result in a broadcast packet being infinitely circulated throughout the network, which substantially increases the load on all devices. If more than one loop exists in a network then this issue is magnified resulting in an exponential replication of packets. However, due to the requirements of modern networks, single points of failure are undesired. Thus redundant links are introduced to allow for potential link failures.

STP[23] is a protocol that has been designed to allow for redundant links to be added to networks without incurring the adverse effects that switching loops cause. The core idea behind STP is to take a looped Layer 2 network and find a tree that represents a loop free topology of that network. The ports of a switch not included within this tree are set to a blocking (disabled) state meaning they cannot forward nor receive packets. Each port of a switch starts in a blocked state, from there the following stages are taken by STP to produce a topology similar to that of the one illustrated in Figure 2.3:

- **Stage 1 - Root switch selection**
  STP works by first electing a root switch in the network, this is the switch with the lowest switch ID. All switches start by assuming they are the root switch, then “hello” Bridge Protocol Data Unit (BPDU)s are exchanged between switches. When a switch receives a BPDU it compares its own ID to the ID of the switch that sent it. If the ID within the BPDU is smaller than the ID of itself then that switch selects the sender of the BPDU as its root switch.

- **Stage 2 - Root port assignment**
  Once the root switch has been selected, all other switches select a single one of its ports that is part of the shortest path back to the root. This port is known as the Root Port(RP) of a switch.

- **Stage 3 - Designated port assignment**
  The root switch becomes the designated switch on each of its ports. Each of the other switches advertises itself as a designated switch on each of its ports along with advertising the distance it is from the root switch.

If a switch receives an advertisement on one of its ports where the cost is less the cost of itself back to the root switch, then it will stop advertising itself as the designated switch on that port.

- **Stage 4 - Port state assignment**
  All Root Ports are moved into an forwarding state. At the same time any port for which a switch is the designated switch that port is also brought into the forwarding state, these ports are known as a Designated Port (DP). All other ports are moved into a blocking state.

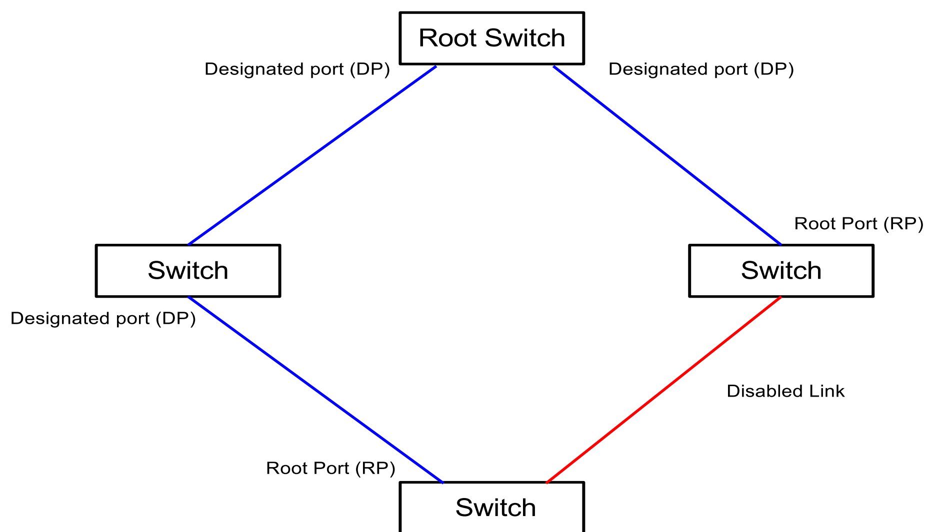


Figure 2.3: Simple STP Configured Network

The possible port states used in STP are:

1. **Forwarding:** Allow the forwarding of packets.
2. **Backup:** Disallow the forwarding of packets.
3. **Pre-Forwarding:** Disallow the forwarding of packets right now. However, if an event does not revert it to a Backup state set it to Forwarding in a short time.
4. **Pre-Backup:** Allow the forwarding of packets right now. However, if an event does not revert it to a Forwarding state set it to Backup in a short time.

The biggest advantage of STP is that its configuration is automatic. In other words if one of the active links fail then STP will notice this and will bring up one of the redundant links without any manual effort of the network ad-

ministrator. However there are three main disadvantages to the original STP approach. One is that the convergence time of STP (the time it takes for a link failure to be detected and all the necessary port modifications to be made) is too long. This convergence time for STP is typically between 30 to 50 seconds. Secondly, the links STP disables are not used for any traffic thus nothing is in place to balance the load across the local network. Lastly, STP does not guarantee using the shortest cost path between any two switches.

2.4 Implementation Tools

2.4.1 Open vSwitch

Open vSwitch (OVS)[25] is an open source virtual switching fabric. It allows for the creation of complete virtual Layer 2 networks on a single machine and has been designed to be a production quality multilayer switch. OVS has native support for OpenFlow version 1.0 and since the release of OVS version 1.9.0, experimental support for the later OpenFlow versions (1.1, 1.2 and 1.3). There is a major limitation around building a real physical network of OpenFlow switches for this project and that is the cost of setting up large enough networks to test the scalability. In light of this limitation OVS has been selected as the method for testing and evaluating the functionality of the OpenFlow controllers designed in this project.

2.4.2 Ryu

There are a number of OpenSource OpenFlow controller development frameworks, some examples of these frameworks include NOX[16], POX[21], Beacon[7], Floodlight[8] and Ryu[28]. Most of these controllers support only the initital version of OpenFlow with the exception of Ryu. Ryu includes support for all of the OpenFlow versions discussed in Section 2.1. It is because of this support that Ryu has been chosen as the framework for this project. Ryu is a component-based SDN framework which provides software components along with a well defined API in order to make it easy for developers to create new network management and control applications. Ryu supports a number of protocols for managing networking devices. In this project Ryu has been used to design and implement the various OpenFlow controllers to achieve the projects goals.

Chapter 3

Investigation

3.1 Address Resolution Protocol

This section is focused on discussing work that has already been done in relation to improving upon the default behaviour of ARP. It addresses some of the proposed extensions to the ARP along with the implementations taken by current networking devices. The extensions and designs described here are limited to those that are relevant to this project, that is approaches that share similar concepts or ones that rely on similar features within the network. Lastly this section also looks at an OpenFlow based approach taken by POX to address the broadcasting of ARP.

3.1.1 Current Extensions and Implementations

A number of traditional approaches to solving ARP's security issues lie in extending the protocol itself. Examples of this are the proposed solutions, Genuine ARP(G-ARP)[6] and Secure ARP(S-ARP)[3]. Both look at extending the ARP protocol to allow for the authentication of ARP traffic at the end hosts, in an effort to prevent cache poisoning. Neither of these approaches is transparent, they both require the end hosts to be running these updated ARP protocols to get the increased security. Both of these proposed solutions are designed to be backwards compatible with the original ARP protocol. This is so that any host that does not understand the updated protocol will still be able to decode the ARP packet, it will however just ignore the extra authentication segment at the end of the packet.

In terms of transparent or on-device approaches, there is Dynamic ARP Inspection

(DAI). DAI is an approach implemented by a network device in order to validate the ARP traffic received on a device. It works by snooping the Dynamic Host Configuration Protocol (DHCP) traffic on a network and saving the Internet Protocol (IP) address to MAC address bindings based off this information to a database. DHCP is the protocol by which hosts on the network are dynamically assigned IP addresses. By watching the messages passed from the DHCP server to the end hosts the devices learn which hosts have been assigned which addresses. From there the ports of the devices are manually assigned to be in one of two states, trusted or untrusted. ARP packets arriving in on a trusted port are handled as usual and processed straight away. On the other hand, packets arriving on an untrusted port have to go through a validation process. This process involves checking the database to validate that the information within the received ARP packet is consistent with the information that was learned from the DHCP snooping process. Only packets that validate are forwarded on, all others are dropped by the device. DHCP snooping only works for the dynamically configured hosts. Statically configured hosts require manual entries to include their information for the validation period. Cisco and Juniper, two of the leading network device vendors, have their own implementations of DAI.

Proxy ARP is an on-device transparent approach aimed at reducing the broadcast of ARP. It is also implemented by both of the previously mentioned network vendors. Proxy ARP is when a centralised network device is tasked with the responsibility of storing a cache of IPv4 to MAC entries. Instead of flooding the ARP request when it arrives, the device responds with a forged response emulating the correct one. This is done only if the device is able to based off the information held within its cache. This reduces the processing overhead at the end hosts by way of reducing the ARP packets broadcasts. Though a trusted device is responsible for the ARP responses, in this case the security is not greatly improved as the device learns where hosts are based on the networks traffic.

3.1.2 OpenFlow Solutions

One of the existing OpenFlow development frameworks POX [21], has explored using OpenFlow to reduce the need to broadcast ARP. Within POX, there is a module called *arp_responder.py* which represents an OpenFlow re-imagining of the proxy ARP approach described in the previous section. The module

works by installing flows onto switches to redirect all ARP traffic up to the controller. It uses this traffic to learn about where hosts are on the network, storing what it learns for later use. When an ARP request arrives at the controller, this module looks up the requested IP in its table and if possible generates the appropriate response forwarding it back to the original host. If the module has yet to learn about the host the request is handled in the usual manner, in which the request is broadcast to entire network. Thus if the controller knows about a host already then it will not need to broadcast the ARP request packet. Instead, by creating the response itself it reduces processing overhead on a network and reduces the amount of processing each host has to do.

3.2 STP

3.2.1 STP Extensions and Successors

Rapid Spanning Tree Protocol (RSTP)[17] was the first extension to the original STP designed to address the weaknesses in STP convergence time after a link failure. It did this by refining the port states as well as introducing several new port roles. The port states of RSTP are:

1. **Discarding:** packets are dropped on a port in this state.
2. **Learning:** packets are not forwarded but the switch is learning which hosts are connected.
3. **Forwarding:** the state in which packets are forwarded.

The port roles of RSTP include the traditional roles within STP along with new Alternative and Backup port roles. An Alternative port is a role assigned to a port that exists on the best alternative path for getting to the Root switch. The Backup port role is assigned to a port to reserve it as a backup path to a network segment in which the switch already has a connection to. The benefit of having these two extra port states is ports in these roles can be moved into a forwarding state quickly with less down time when compared to STP, which moves ports in to an inbetween port state for a set time. The default convergence of RSTP is approximately 6 seconds. This is the time taken for 3 "hello" BPDUs to be sent between switches. This default value can be reduced so typically RSTP takes less than 6 seconds to converge. In 2004 the IEEE

grandfathered STP and replaced it with RSTP.

Following the introduction of RSTP two further extensions, Multiple Spanning Tree Protocol (MSTP) and VLAN Spanning Tree Protocol (VSTP)[18] have been introduced. Both MSTP and VSTP are extensions to RSTP in order to improve its use in a multiple VLAN environment. These protocols however, are not relevant to the aims of this project. STP has recently also received two potential successors Shortest Path Bridging (SPB)[19] and Transparent Interconnection of Lots of Links (TRILL)[24]. SPB and TRILL appear to be designed for use in larger networks as well as offering additional features such as load balancing, shortest path forwarding and a more scalable approach to switch port configuration. These approaches target the network inefficacy and nonoptimal switching behaviour that STP and its current extensions exhibit.

TRILL effectively introduces Layer 3 elements into a Layer 2 network in order to improve upon the weaknesses of STP. This is done by having the switches running a link state protocol known as Intermediate System to Intermediate System (IS-IS). IS-IS is a Layer 3 protocol designed to flood link state information among a network of routers, so that every router can build up knowledge of the network topology. The result of this is that the routers can then select the best or optimum path for IP traffic. In the case of TRILL we have switches instead of routers running this protocol, which can be done because IS-IS does not require any Layer 3 knowledge in order to run. As such the switches themselves build up a knowledge base of the local network. Once this knowledge base is built the switch can then make decisions to allow traffic to take the optimal paths. To calculate these optimal paths IS-IS uses Dijkstra's algorithm to find the shortest paths between switches. This has also been extended to include multi-pathing, allowing a switch to load balance traffic across paths of the same cost. TRILL handles switching loops via a combination of a new Time To Live (TTL) field introduced to a frame which prevents packets from continuing to circulate the network as well as reverse path forwarding. Reverse path forwarding is when the switch determines the reverse path of a broadcast packet based on its source address, any broadcast packets received with the same source as the original not arriving from this path are dropped, this eliminates loops. The main drawback with TRILL is that it requires new ASICs for switches in order for it to run, due to the switch having to preform a new form of lookup for traffic. As TRILL requires new ASICs, to implement in on a network would require the replacement of the networks current Layer

2 switches. However, as it has designed to be backwards compatible not all the switches of a network need to be replaced. The number of TRILL enabled switches determines how effective the load balancing and optimal path switching is in the network.

SPB is a recent STP successor only being approved by the IEEE in May 2012[19]. Like TRILL it works by using the IS-IS protocol to exchange information between switches so that each individual switch has a knowledge base of the network. Again it is capable of multi-path forwarding and forwards down the shortest path based on the IS-IS calculation. Unlike TRILL it is designed to be compatible with existing switch ASICs. SPB handles loops with just the reverse path forwarding calculation.

Both of these successors are designed to allow for better utilisation of a network, as well as allow for better scalability through the use of the IS-IS protocol. IS-IS has been proven to scale to large networks due to its use in layer 3 routing before it was adopted to these two approaches. AVAYA's comparison between SPB and TRILL[1] gives a detailed comparison between the features and design of these two protocols underlining their background as well as the actual implementations.

3.2.2 OpenFlow Solutions

Within the OpenFlow frameworks, POX and NOX have both had modules added to them to allow for their controllers to produces the same loop prevention features of STP. These modules aim to recreate the feature set of STP allowing an OpenFlow controlled network to handle loops. However, no extra work has been done in either of these modules to incorporate any additional features. A detailed overview of how it works in the NOX source code has been given here [14]. It is important to note that even though this NOX module recreates the same feature set as STP it goes about it in an entirely different way due to use of OpenFlow's logically centralised controller. The other thing to take from this approach is that it puts the ports into an OpenFlow version 1.0 defined state known as NO_FLOOD. The benefits of this NO_FLOOD state is that it does not disable the port entirely but rather disables it only for broadcast traffic. The result of this is that the ports can still be used for forwarding non-broadcast traffic.

A shortest path forwarding implementation using OpenFlow[27] has also pre-

sented an alternative to STP along with comparing itself with the two successors TRILL and SPB. This shortest path approach looks at building upon the basic STP module of NOX by introducing a separate traffic forwarding module. In this implementation the controller learns about the topology of the network, then for each new flow that enters the controller the shortest path between its source and destination is calculated. The controller installs the appropriate flow rules along all the switches in this calculated path before the packet is forwarded on. The weakness for this approach are that there is no form of load balancing being down as well as the fact that there is no immediate link failover or path re-configuring when a link goes down.

Chapter 4

Implementation

4.1 Address Resolution Protocol

As previously mentioned in Section 2.2, this project looked at using OpenFlow to explore transparent solutions for reducing or removing the broadcast of Address Resolution Protocol (ARP) packets from Layer 2 networks. The focus on addressing this aspect of ARP was chosen as it provided a good basis in which to form familiarity with both Ryu's environment as well as learning about the OpenFlow protocol and Open vSwitch (OVS). ARP is also a relatively simple protocol to understand conceptually providing a good starting point for this exploration.

This project addresses the previously outlined issues with ARP in two ways:

- **Controller Redirection and DHCP**
  Uses the OpenFlow controller to forward all ARP requests to a particular host running a script to create the ARP response.
- **Proxy ARP Controller**
  Involves using the controller as a proxy for ARP requests, intercepting and responding to ones it is able to, similar to the one used in POX.

These implementations expressed have been designed with the assumption of a fully OpenFlow controlled network. The goal of these implementations is for them to be transparent to the end hosts. This is so the end hosts knowledge of the ARP protocol would not need to be changed in order for it to work. As this was the introductory stage in the project it was decided to focus on a single switch network while a background and familiarity were being developed.

To extend the approach that uses the controller to redirect the ARP packets to a multi-switch network would require minimal work. Using OpenFlow's programmatic nature we could directly program the location of the trusted Dynamic Host Configuration Protocol (DHCP) server into the controller. The controller would need some additional checks in order to preform the dropping of ARP responses not originating from the trusted host.

For the proxy ARP approach, the controller could be extended by including an additional forwarding module similar to the one discussed in Section 4.2.2 to allow for for multiple switches.

4.1.1 Controller Redirection

This approach combines OpenFlow flow control with a separate host running as a DHCP server. This separate host has a script running locally, listening specifically for ARP requests and generating the appropriate responses. Though this approach uses more than just OpenFlow to achieve its goal, it is through the flexibility of OpenFlow's flow control that this is possible in this instance. Figure 4.1 shows the conceptual differences between the traditional approach and the approach taken by this project. Figure (a) demonstrates the traditional behaviour of ARP in the case of Host A attempting to find Host B on the network. The red arrows indicate the ARP request from A, this request is sent to all of the hosts on the network. The blue arrow indicates a response, in this case it is the response sent from B back to A. Figure (b) demonstrates

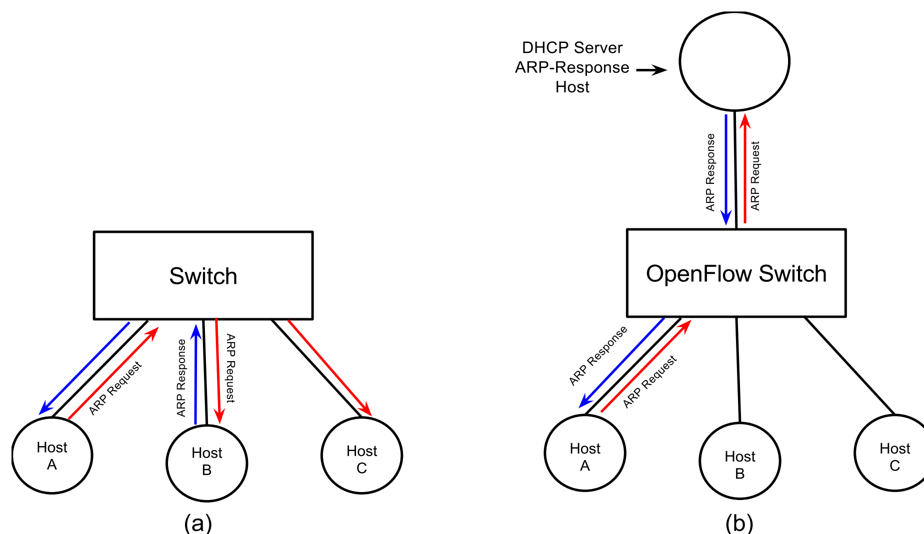


Figure 4.1: ARP behaviour comparison

the controller redirection approach. In this case the ARP request has been forwarded only to the host running the DHCP server and response script. This host generates the response that B would have responded with and sends it back to A.

In this project implementation the DHCP server running on the host was an ISC-DHCP server[5]. The ISC-DHCP server was chosen because it allows for the querying of the server via the Object Management Application Programming Interface (OMAPI) protocol in order to discover the Internet Protocol (IP) addresses it has leased as well as which Media Access Control (MAC) it leased it to. This allows for a script to be running on the host that listens for ARP responses. When it receives one it can query the server in order to construct an appropriate response and send it back to the requesting host. This script is presented in Appendix A.1 and makes use of two python modules in order allow for this querying and ARP response construction. These are Scapy[2] and Pypureomapi[15]. Scapy is a powerful interactive packet manipulation program. It is able to forge or decode packets for a wide number of protocols, send them on the wire, capture them and match requests and replies. Pypureomapi allows a python script to query an ISC-DHCP server for the necessary information to create a replica response. By combining these two module we are able to have the script listen for queries. When one arrives, the script can decode the request and use the data to query the DHCP server via OMAPI. Once it receives the response, the script then uses this information in order to craft the appropriate response packet. Figure 4.2 gives a visual representation of the trusted host and how the DHCP server and script interact.

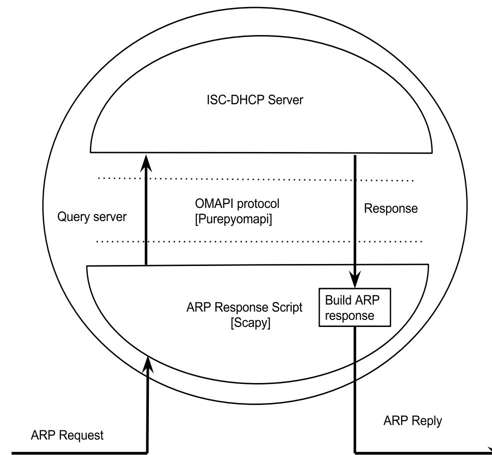


Figure 4.2: DHCP and ARP responder host make up

This python script allows for the designated DHCP server host to be responsible for all ARP traffic handling. The next stage of this approach was to eliminate the broadcast of ARP which was quite simple using OpenFlow. The OpenFlow controller designed for this task was programmed to have a specific port on a switch which it expected the DHCP host to be connected to. Once this port was defined and the server host was plugged in, all of the ARP traffic was redirected through the controller to this DHCP host. This meant that ARP no longer had to be broadcast to the entire network. Instead it was only forwarded to a single host. To help improve security, the controller introduced more flow rules that forwarded all ARP responses not originating from the DHCP server host port to the controller to be dropped. Due to the controller intercepting all requests, no host should need to generate ARP replies. This means all replies not from this trusted port can be considered unsolicited and therefore potentially malicious.

Hosts with statically configured addresses requires some manual entry. The easiest way to get statically configured hosts complying with this ARP response implementation is to use the approach taken by most large current networks that have a high quantity of statically addressed hosts. This is to have DHCP server to reserve and hand out the desired addresses to the hosts, thus the DHCP server still knows where these hosts are.

4.1.2 ARP Proxy OpenFlow

This controller was designed and implemented to be a comparison point for the ARP redirection controller explained above. The implementation of this controller was very similar to the one implemented in POX because it also explored an OpenFlow implementation to the Proxy ARP currently existing in networking equipment. Its job was to learn and build a table or cache of the mapping of host MAC to IP addresses. The mappings were learned through the forwarding of all ARP requests to the controller. The controller was also in charge of responding to all ARP requests when it knew the requested hosts information. The major difference was that this controller was coded with OpenFlow version 1.2 in mind, unlike the POX implementation which is built for OpenFlow version 1.0. Appendix A.2 shows the code responsible for handling the ARP responses. The learning of the mapping between MAC addresses and IP addresses is also slightly different in the fact that it does not just limit it self to the ARP requests. This controller also examines all the other flows that get forwarded onto the controller as well. This is due to the fact that it was designed to be a self contained module i.e. it had this functionality as well as performing the other jobs of a switch. Unlike the POX one which is designed to be run in conjunction with another module in order to get switch behaviour.

This controller served as a good starting point for getting familiar with the Ryu framework as well as learning some of the concepts around OpenFlow specifically version 1.2.

4.2 Loop Prevention and Equal Cost Multipath Forwarding

The alternative to spanning tree presented in this report focuses on using the centralised view that OpenFlow provides to handle loop prevention as well as handling the forwarding of non-broadcast traffic. The most crucial part to using OpenFlow for this task was having an OpenFlow controller that could dynamically discover the network topology as well as detect link and switch failures.

Ryu already has topology discovery module included in its source code. This topology discovery module worked by registering every switch and each port

on that switch when it first communicates with the Ryu controller. Once a switch was registered, the controller would periodically generate Link Layer Discovery Packet (LLDP)s which were then sent out each port. When the Ryu controller received one of these packets it was able to examine the contents to discover from which switch and port the packet originated. As the controller knows which switch and port this packet was received on it is able to discover the links of the network. These LLDP packets are sent every five seconds. If the controller does not receive one of these packets on a given link within 10 seconds that link is assumed to have gone down and the controller will update itself.

One of the initial requirements for being able to design the alternative to Spanning Tree Protocol (STP) was updating the Ryu topology discovery module to OpenFlow version 1.2. This is because the module was originally coded for OpenFlow version 1.0 with plans by the developers to be extended to later versions in the future. Once updated this code was able to discover loops on an OpenFlow version 1.2 enabled network.

The next step was planning the structure of the approach. It was decided that the approach to be taken would be similar to the NOX shortest path forwarding implementation in [27] in terms of splitting the job of loop prevention and packet forwarding into two separate modules. The loop prevention module is responsible calculation of the loop free topology as well as pushing the configurations to the switches. Where as the forwarding module was designed to handle the decisions for forwarding of unicast traffic in the network.

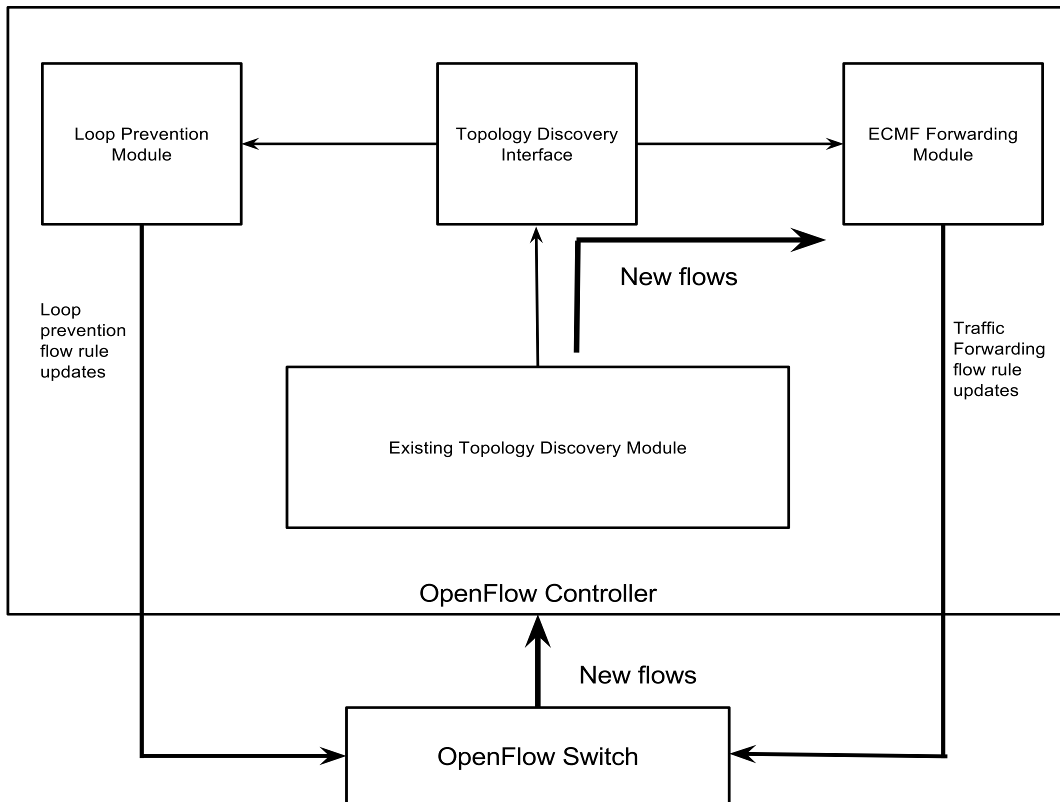


Figure 4.3: Controller composition

Figure 4.3 describes the traffic flow between the switch and the controller and how the modules responded. The normal traffic (non-broadcast) is sent to the controller and is then passed of to the forwarding module. This module makes a decision on how to forward the packet and passes it back to the switch to forward on. This module is also responsible for installing the flow rules to the switch for these packets. On the other hand we have the loop prevention module which when required sends and removes rules from the switches to prevent loops.

Based on the discussion of the shortest path NOX implementation in [27] it was decided that this project's implementation would look into adding load balancing of the traffic between paths in order to build upon the shortcomings of the NOX implementation.

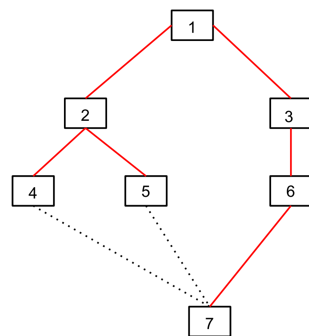
This projects solution was built as an extension to the existing Ryu topology

discovery code. In essence if the packet received by the controller was anything other than a LLDP packet it was passed off to another module for processing.

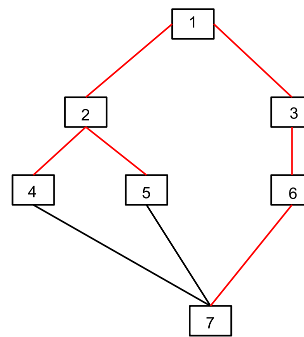
4.2.1 Loop prevention Module

This module aim is to prevent the negative effects of loops, however to do it in such a way that does not disable the actual ports like in STP. This is so that non-broadcast or unicast traffic can still make use of them to better utilise the network. As previously mentioned, OpenFlow version 1.0 introduced its own version of port states. Most notable of these was the NO_FLOOD state which is a port state that prevents broadcast traffic being forwarded out of a port. Non-broadcast traffic however, can still flow freely through ports in this state. Unfortunately this feature was removed from OpenFlow 1.1 onwards. Therefore because this project was using OpenFlow 1.2 this loop prevention approach uses flow rules on the switch to drop broadcast packets on certain ports.

When the controller is initially started each one of the switches that register with it gets a single flow rule added which causes the switch to drop all broadcast traffic on every port. After an initial stand off period, to allow time for all switches to register and the topology discovery code to discover links, the controller makes a call to the loop prevention module. The loop prevention module then looks at the topology learned by the controller to find a loop free tree. Unlike STP which has to pass messages between switches in order to determine this, we have a controller which is central to the network and knows about all the links. To find this loop free topology a simple A* search algorithm is applied over the links in the network.



(a) STP



(b) Loop Prevention and ECMF

Figure 4.4: Link disabling comparison - STP vs Loop prevention

Figure 4.4 show the comparison between the disabled links and tree learned in both the STP and this loop prevention module. This example assumes Switch 1 has been selected as the root. The dotted lines represent disabled links and the red links represent the tree learned for each method.

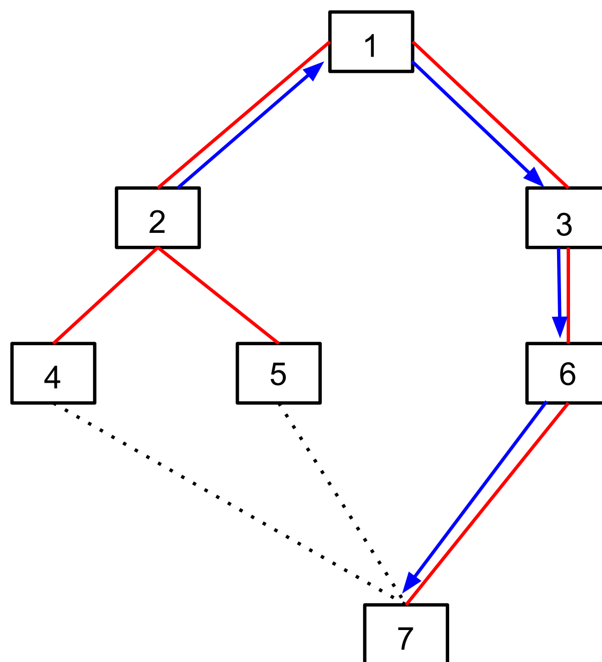
It was intended for this algorithm to expand the lowest cost paths first using the link speed as the cost metric. However, OVS does not associate a link speed with its virtual links. Instead, a simple file was used to define the costs of links. With the controller assigning a default link cost for any link that does not have an entry in this file. The search keeps track of which switches have been visited. The second time a switch is visited the ports associated with the link used to get there are added to the list of ports that cause loops. Once the full network has been explored the controller iterates through all the switches, first removing the original rule that blocks all broadcast traffic, then adding a rule to drop broadcast traffic for each port included in the list of ports that cause loops. This result in broadcast storms being prevented, but keeps the ports in an active state. Appendix B.1 shows the python implementation of this search algorithm run by the controller.

It is worth mentioning that the initial rule added to a switch and the stand off period only ever happens when the controller is first starting. Once the initial topology has been learned then it is not necessary for the controller to stand off again. A link up topology change event results in a rule being added to a switch to drop broadcast traffic only on the new link, similar to STP switch ports coming up in a Blocked state. This is just in case the link results in a switching loop. After this rule has been added the loop prevention module re-calculates the loop free tree and updates the switches accordingly. A link down event just requires a recalculation of a new loop free tree.

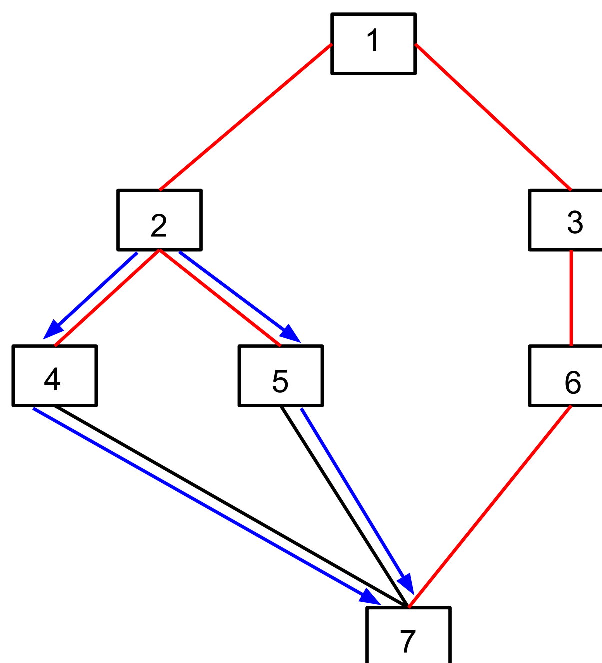
4.2.2 Equal Cost Multipath Forwarding

Once loop prevention has been handled without disabling ports, more interesting things can be done with traffic such as guaranteed shortest path forwarding or load balancing. An OpenFlow controller can be programmed to control traffic as the network administrator sees fit and therein lies the biggest potential of OpenFlow. For this part of the project it was decided to focus on having the controller do shortest path calculations as well as utilising more of the links of a network via a method which will be referred to as Equal Cost Multipath Forwarding (ECMF). ECMF looks for the shortest paths (lowest cost paths) between any two switches. Equal lowest cost paths are stored in the controller and flows are divided up equally among them, allowing for more of the network to be utilised.

4.2.3 Shortest Path Algorithm



(a) STP



(b) Loop Prevention and ECMF

Figure 4.5: Packet forwarding comparison - STP vs ECMF

The Ryu module built for this project works as follows: After the initial stand off period (discussed in Section 4.2.1), the forwarding controller module uses the topology learned by the discovery stage to calculate all the possible lowest cost paths between each pair of switches on the network. The shortest path calculation algorithm used in this module is similar to Dijkstra's algorithm. It works by expanding the lowest cost links first and as soon as the destination is reached the path taken and its cost is recorded. The algorithm continues expanding paths until the cost of getting to the destination exceeds that of the original path. Any and all paths that are of equal cost to this original cost are also saved. Figure 4.5 gives a comparison between the traditional STP behaviour of forwarding packets and the ECMF approach taken for this project. Again Switch 1 is the root switch in both diagrams. The dotted line represents a disabled link and the red tree represents the (broadcast) enabled links in the network. The blue arrow indicates the path that can be taken by flows originating from a host on Switch 2 destined for a host on Switch 7. As can be seen the STP can only traverse the active links of the network resulting in traffic taking a less than optimal path. In the case of the ECMF approach, flows are able to be sent via either of the two shortest paths discovered. Appendix B.2 shows the algorithm currently implemented for finding the shortest path between a pair of switches.

It is important to note that because the centralised controller does all the calculations for each switch, this algorithms complexity is quite high. This is when compared to Shortest Path Bridging (SPB) or Transparent Interconnection of Lots of Links (TRILL) where the shortest path calculations are being done separately at each switch. The shortest path calculation used for this ECMF module is suited for parallelisation however the implementing of such lies outside the project's scope. This shortest path calculation step is done after this initial stand off period as well as after any topology state change.

Initially it was explored as to whether the controller should just do this shortest path calculation only for every new flow entering the controller, limiting the scope of the search to that between the source switch and the switch the destined host is connected to. However, it was decided that it was better to have a larger calculation time at the start in favour of shorter lookup times for path decisions on new flows, as topology state changes are rare. This differs from the NOX shortest path forwarding implementation[27] which calculates a path at the arrival of a new flow to the controller.

4.2.4 MAC Learning and Path Choice

For these paths to be useful the controller needs to learn where hosts are on the network and to which switch they are physically connected. This was achieved by determining which ports of the switch are trunk ports (ports connecting to other switches). The controller will map a hosts MAC to a switch only when the port the traffic was received on does not belong to this trunk port set. By doing this we can map host addresses to switches so when a new flow comes arrives at the controller the destination switch or last switch in the path can be located based off the destination MAC address. Once the destination has been determined the source and destination can be used to look up the possible paths for the flow. The controller selects one of the paths, if multiple exist, using a simple counter which is incremented after every selection thus distributing flows down different paths of the network. To stop these flows being sent to the controller again, the controller installs new flow rules onto the switches along the path so that every new packet that is part of this flow will follow the original.

4.2.5 Flow Distribution

Two methods were implemented for the distribution of flows in the network, each with a slightly different partitioning of flows between equal cost paths. The first method includes a full-path based split between flows, meaning that flows were evenly distributed between the possible paths (set of links) from the source to destination. The other method was a per-switch interface based split, this meant that at each switch the flows were evenly distributed between all the possible ports or interfaces that existed in one of the shortest paths between the source and destination. The difference between them is marginal but worth implementing. In the case of a simple network where there are three possible shortest paths between the source switch and destination switch, labeled P1, P2 and P3 respectively. Assuming there is a switch that has two interfaces I1 and I2, where I1 belongs to the path P1 and I2 belongs to the paths P2 and P3. In the case of full-path split two thirds of the flows will go down I2 and the remaining third down I1. In comparison, the interface based split results in the flows being distributed evenly between I1 and I2.

4.2.6 Link State Changes

When a topology change occurs (such as a new link coming up) shortest paths are recalculated. Alternatively if a link is to go down an extra step is taken before recomputing all the shortest paths. This extra step requires the controller to remove all the flows that used the link that went down from the pair of switches that the link connected. This is so that new packets in the flow will be sent to the controller when they arrive at these switches, then the rules for a new path can be installed. This particular feature was to address the issue with no path recalculation under link failure discussed in the previously mentioned shortest path forwarding approach in NOX.

Chapter 5

Evaluation and Discussion

5.1 ARP

Two factors were chosen to focus on when evaluating the controllers and designs put forward for this project. The first looks at how much of the Address Resolution Protocol (ARP) broadcast traffic is reduced and or eliminated. The second factor is how the approaches taken impact on the security of ARP. This section aims to discuss and evaluate the controllers created in terms of the above factors. In relation the transparency requirement expressed earlier, both of these approaches have been coded to have some level of transparency to the end hosts. However, one of the approaches does require some configuration to one of the network services in order to be used. In the case of the viability of their deployment, all of the approaches discussed in this chapter require a fully OpenFlow enabled network. This is a network in which each of the switches is configurable via the OpenFlow protocol.

5.1.1 Controller Redirection

Evaluation

As previously outlined in section 4.1.1, this approach involved a combination of two parts. A Dynamic Host Configuration Protocol (DHCP) server and ARP response script running on a trusted host and an OpenFlow controller modifying the behaviour of both ARP requests and responses on the network.

Given the assumption that all hosts on the network are handed an address from the DHCP server on the trusted host, including machines with statically assigned addresses, then this approach eliminates the need for all ARP broad-

cast traffic within a local network. This is because all ARP broadcasts are now unicasted to the trusted host instead of being flooded by the switch. As a result the processing overhead on a network which was discussed earlier is reduced. Another result of this is the intermediary OpenFlow switches in the network have a reduced load on them as they no longer need to spend time flooding ARP requests.

In relation to a production network, network administrators would likely want to further extended to include some redundancy such as further trusted DHCP hosts in combination with some DHCP fail over mechanism. This would allow for the load balancing of ARP traffic within the local network as well as provide a backup should links fail. Another extension to this approach would be, rather than having all switches controlled by a single controller, the network could be split up into islands. One OpenFlow controller would be in charge of a single island. This can be done as the OpenFlow controllers only need to know which port the trusted host will be sending packets, they do not require any further knowledge of the network and thus can be run independently.

As discussed in the implementation of the redirection of ARP approach, extending this approach to drop all responses not originating from the trusted host resulted in a positive impact on one of the particular vulnerabilities of ARP. This is because the controller has the notation of trusted hosts, thus any response not from this trusted host has been assumed to be an unsolicited response and therefore potentially malicious. This controller could be extended further to allow for further trusted hosts such as core services if needed. It is the combination of the singular tasking of ARP responses to a trusted host along with the re-defining of ARP behaviour provided by the OpenFlow controller that has allowed for this security improvement of ARP.

There are two main requirements needed for the introduction of the ARP redirection method into a network, other than then just having OpenFlow enabled switches. These are that the host running the DHCP server has to be modified so that it is running the response script discussed, along with the DHCP server(s) having to be one that talks the Object Management Application Programming Interface (OMAPI) protocol. This limits its use to networks running the linux ISC-DHCP server. It is due to these requirements that this approach is not fully transparent. Though the hosts do not need to change their understanding of the ARP protocol, the DHCP servers utilised within the network

do require some modifications in order to gain improvements to the security and efficiency this approach provides.

Comparisons

The closest current method taken with existing network gear at Layer 2 to the controller redirection approach is Dynamic ARP Inspection (DAI). This controller redirection implementation does share certain similarities and ideas with DAI building upon some of the concepts introduced. However, at a much higher level of abstraction thus resulting in a practical implementation that is unique. Unlike DAI the OpenFlow switch itself does not instrument any form of complex validation process instead assigns one port as a trusted port and drops any responses not originating from that port. This removes a lot of the work at the switch level.

Of the two OpenFlow methods explored by this project, this one is certainly the more secure of the two and fully eliminates the need to broadcast ARP traffic within the network. Therefore in terms of the evaluation factors focused on by this report, this approach is certainly the stronger of the two. It does however, have the limitations described above and thus only applicable to certain kinds of networks.

5.1.2 ARP Proxy Controller

Evaluation

Unlike controller redirection implementation, this controller has been designed to be fully self contained i.e. just one system responsible for ARP as opposed to the two factor step taken above. Also unlike the controller discussed above it does not fully eliminate the need to broadcast the ARP request packets as it requires a host to be learned about in advance before responding. If a host has yet to be learned then it must fall back to the traditional behaviour of ARP in order to maintain a connected network. It learns about host locations by observing their ARP requests and storing a mapping of the source information. Once all hosts have been learnt, then the broadcast traffic is eliminated from the local network.

This approach is more focused on reducing the ARP broadcast traffic on the network and offers very little in the way of security, though it does reduce the need for the broadcast of ARP requests so it does marginally reduce the

potential for a malicious ARP response. The reason for this is that this implementation relies on learning the location of hosts based on the traffic on the network which at this stage has no way of checking the validity of the addresses. However certain measures could be put in place to introduce more security into this application. This could be having the controller allow for only a single Internet Protocol (IP) to Media Access Control (MAC) pairing to be learnt on a port. Then making the controller drop any traffic from this port, where the frame information does not match what was learn. Alternatively, the location of the significant hosts could be statically programmed into the controller, making it less likely for a malicious host to assume their identity. As this particular focus of the project was to gain familiarity neither of these approaches were explored but both would make good instances of future work.

ARP proxying is already enabled in current networks, as such this approach could certainly be deployed in a network and requires no further modifications to the underlying network, as the changes are only limited to the switches themselves.

Comparison

It is important to note that this particular approach is very similar to the existing POX code implementation and likewise shares traits with the already existing method implemented in current switches known and Proxy ARP. Relative the POX implementation there are a number of changes, most significant of these is that this implementation has been written under the 1.2 version specification of OpenFlow as opposed to the 1.0 version specification used in POX. The other significant difference was that this was constructed in a entirely different framework, Ryu. This approach was geared more towards learning how Ryu works and how it is put together as the POX implementation offered a basis to learn off. As previously pointed out in another chapter this has also been designed to be a singular switching module. It is in charge of both the ARP response as well as the switching of the network. This being different to the POX module which is designed to run in conjunction with other modules in order to create a switched network.

Due to its reliance on learning where hosts are in the network in order to reduce ARP broadcast, this method does not offer the same kind of security boost as the controller redirection approach. It also does not entirely eliminate the need to broadcast ARP requests, as such the other approach certainly looks like the

more promising. This method is certainly the most transparent between the two outlined thus would require less work to implement.

5.2 Loop Prevention and ECMF

With the limitations imposed by the virtual network, testing the resulting solutions on simulated real network traffic fell outside the scope of this project. Therefore the evaluation of this implementation was focused more on how these approaches would scale to larger networks and the performance of the algorithms used. Three factors of the solution presented were focused on in the evaluation. The first was number of rules having to be introduced to switches. The second was the time it took to calculate the loop free tree for the network. The last was the time it took to calculate all the multiple shortest paths between every switch in the network.

The following discusses how these factors affect the performance and scalability of this implementation:

- **Number of rules**
  With every new rule addition to a switch, the time taken to match a flow in the switch increases. This also means the time taken for the switch to make the desicion to pass a flow up to the controller increases.

- **Tree calculation time**
  This factor impacts more on the scalability of this approach. This is because it contributes to the convergence time, the longer this takes to run the longer the wait time for new flows to be forwarded after a link state change.

- **Shortest path calculations**
  This factor is similar to the above, it contributes to the convergence time of the network. The longer it takes to run the longer wait time on the network for forwarding new flows.

The runtime of the loop prevention algorithm as well as the Equal Cost Multipath Forwarding (ECMF) algorithm were tested on two different network design topologies. These were a ring topology which consists of a ring of switches which only has a single loop created by connecting the last switch up to the first. The second was a partial mesh network created by building a tree

of n switches and then creating a random number of loops between the last two frontiers. The result of constructing a topology in this way is a guaranteed minimum path length between the first and last switch, while also potentially creating multiple equal paths between them. Originally it was discussed to test this on a full mesh (every switch has a unique link to every other switch) network as well, however due to the fact that every switch already has a shortest path of length one to each other switch, it was decided that it was not applicable to what the ECMF module was trying to achieve. A number of tests were run with a varying number of switches and by extension a varying number of links in both the ring and partial mesh topologies in order to discover the scalability of the controller for a given number of switches. It is worth mentioning at this point that it was discovered that Ryu's topology discovery code at the time of building the module can not detect explicit loops. These are loops that occur on a single switch by connecting one port of the switch to the another on the same switch. This is a potential problem, however as an explicit loop is one that offers no benefit to the network in terms of redundancy and is easy to track down then the threat of it becomes somewhat lessened.

All experiments presented in this were carried out a single machine with the following specifications:

- **OS**: Ubuntu Release 12.10 (quantal) 64-bit
- **CPU**: Intel Quad Core i5-3570K CPU 3.40GHz
- **RAM**: 16Gb DDR3
- **SSD**:Samsung 840 Series 120GB SATA3 SSD

5.2.1 Evaluation

Rules

The number of rules having to be introduced to each OpenFlow switch is a comparison factor against other OpenFlow implementations, it is not comparable to the non-OpenFlow methods. As OpenFlow version 1.2 was used in the construction of controllers for this project the total number of rules is kept low relative to the OpenFlow version 1.0 specification. This is due to a point discussed earlier in this report as OpenFlow 1.2 scales at O($2n$) as opposed to OpenFlow 1.0's O(n^2). Because of this increased scalability, the design decision to introduce loop prevention rules to switches still results in a more

scalable alternative in terms of rules introduced to a switch, when compared to a simple forwarding module under the OpenFlow 1.0 specification. The total number of rules having to be introduced per switch can be modeled by the following equation:

$$Rules = 2h + l_i \tag{5.1}$$

were:

- h = total number of hosts on the network
- l_i = number of ports belonging to loops on switch i

Loop Prevention Results

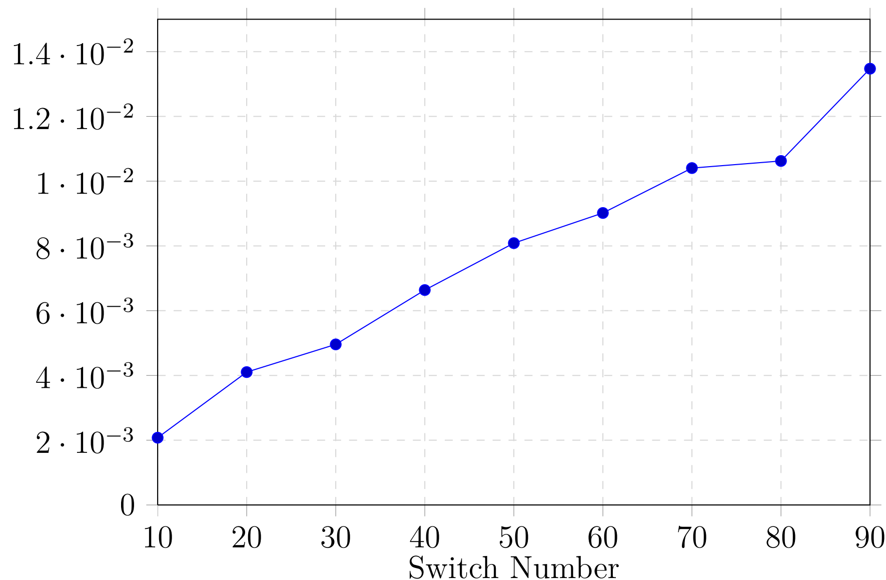


Figure 5.1: Loop prevention module runtime - Ring topology

Figure 5.1 shows the time taken for the loop prevention algorithm to run on the given number of switches laid out in a ring topology. Dijkstra's algorithm has a complexity of O($nlogn$). The algorithm used for this calculation in the loop prevention module is a variation on Dijkstra's, thus the algorithm appears to scale well in this instance easily getting as high as 90 switches while still being under a second in calculation time. The reason 90 switches was chosen as the top was due to links randomly alternating between an up and down state when moving up to and beyond 100 switches. This only occurred in some of the tests, the others completed without incident. Due to the nature of the virtualised environment used when running tests it was to difficult to discern

what caused this link fluctuation. Whether it was a side effect of having so many virtual Open vSwitch (OVS) switches running on a single machine, the Ryu topology code not being able to handle that many switches, or links or if it was in fact an artifact of the overhead introduced when running the loop prevention and ECMF modules.

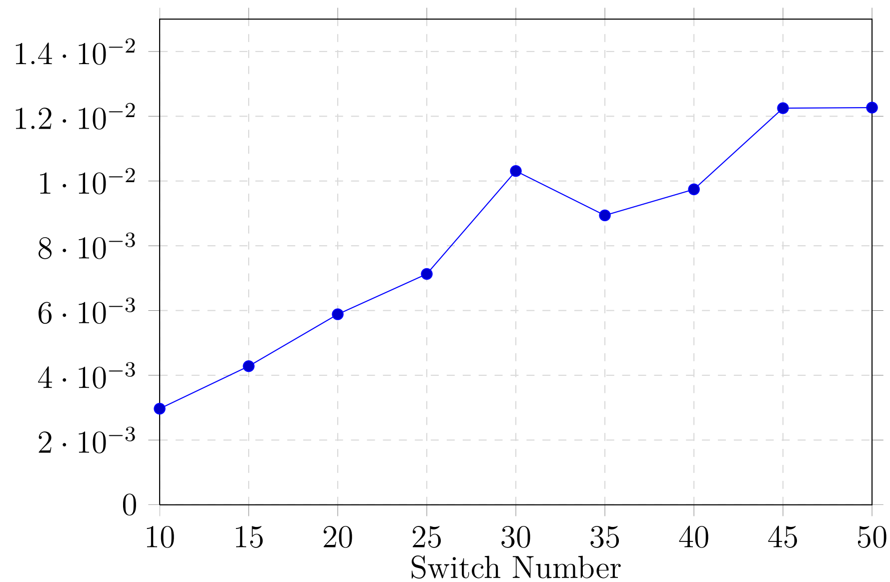


Figure 5.2: Loop prevention module runtime - Partial mesh

Figure 5.2 is the results of the loop prevention algorithm when run on the partial mesh topology. There is a larger amount of links introduced via the introduction of an individual switch to this network as well the existence of multiple loops, as such the link fluctuations were apparent much sooner occurring occasionally at 60 switches.

ECMF Results

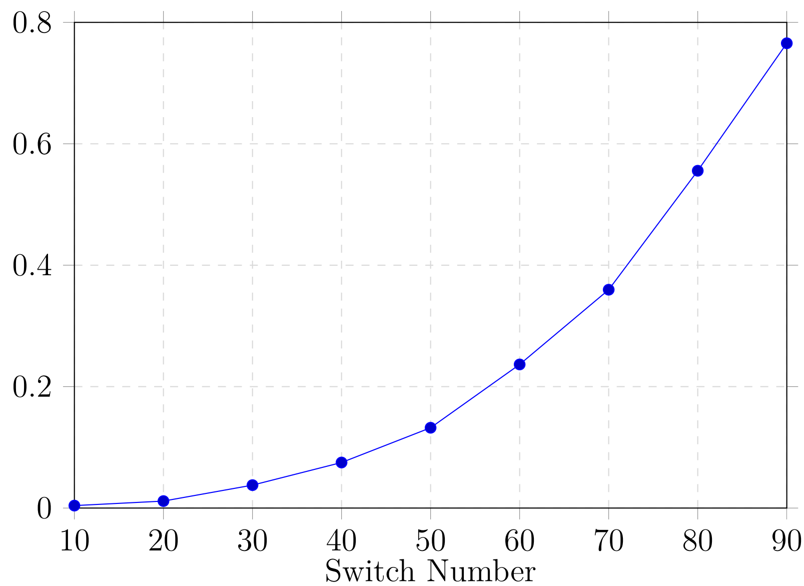


Figure 5.3: ECMF module runtime - Ring topology

Figure 5.3 is the results from finding all the shortest paths between switches in the ring topology. The slope of this graph is what was expected due to fact the controller is singularly responsible for calculating the shortest paths between all pairs of switches. Again this calculation is not run often as it only needs to be run every time there is a link change, so the computation times impact on the network is small.

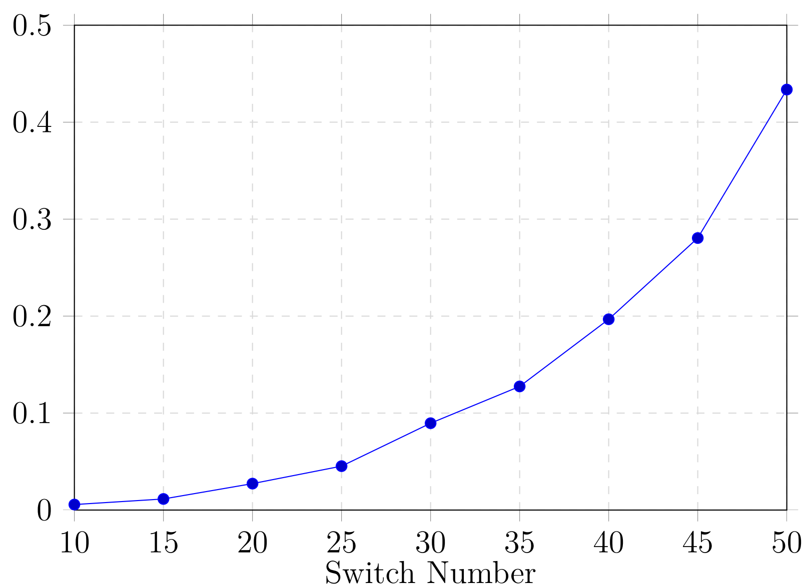


Figure 5.4: ECMF module runtime - Partial mesh

Figure 5.4 follows the same trend as above though with the introduction of more links its complexity becomes more apparent. Again in terms of actual time taken to calculate it is still less than a second in the test cases. Therefore though the complexity is high due to its need to calculate all the shortest paths between all switches within the controller, in practice the calculation does not take long as only simple calculations are needed within this algorithm.

Discussion

It is important to note one main thing about the above results. Due to the modular and programmatic design of the Ryu framework neither the loop prevention module or ECMF module are bound to the algorithms that are currently implemented within them. This property is where one of the other major benefits of an OpenFlow controlled network lies, its ease of extensibility. With the controller currently implemented either of the two algorithms could in fact be swapped out and improved upon. The result of which would be changes only having to be made locally to the controller. In other words to change or extend the functionality of the controller requires no changes of the physical switches running in the network.

5.2.2 Comparisons

Existing OpenFlow Approaches

In comparison to the majority of OpenFlow approaches brought forward in section 3.2.2, the loop prevention and ECMF approach utilises more of the network by load balancing flows across multiple shortest paths. This provides a much more efficient use of the underlying network. This project has aimed to explore what more we could do with an OpenFlow controlled network to address some of the shortcomings of the Spanning Tree Protocol (STP). Therefore it is a vastly different approach to the ones taken in the POX and NOX controllers which aimed only to recreate the features of STP in a OpenFlow enabled network. With respect to the NOX shortest path forwarding approach outlined in [27], this projects approach differs more in the fact that it tries to build upon some of the limitations discussed in the respective report. Within this report there is mention of features within OpenFlow version 1.2 that would improve their own solution. However, due to the incomplete implementation of 1.2 in OVS these are still not possible. As such, other methods have had to be explored by this project to result in an improved implementation. [27] also

states that the flow rules are not removed when a link failure is detected. In this project's implementation this is addressed by the controller removing all flow rules from the pair of switched connected by the failed link. This means that flows following the old path will eventually hit the affected switches and be sent back up to the controller so a new path can be calculated and old rules overwritten. In terms of rules introduced on the switches contained within a network, all previously discussed implementations have used OpenFlow version 1.0. The rule scalability of this project's approach is much improved as well as this project allows for more extensibility due the use of the multiple flow tables.

STP: Successors and Extensions

	STP	**RSTP**	**Shortest Path OpenFlow [27]**	**ECMF OpenFlow**
Convergence	30-50s	Less than 6 s	Not given	Approx. 11s
Shortest path	Not guaranteed	Not guaranteed	Yes	Yes,uses a search algorithm to determine in advance
Loop prevention	Calculate tree, ports not included are disabled	Calculate tree, ports not included disabled	Tree calculated, not included ports put into NO_FLOOD state	Tree calculated, flow rules introduced to drop broadcast traffic on ports not included in tree
Broadcast traffic	Can only travel down enabled paths			
Unicast traffic	Only travels down enabled paths	Only travels down enabled paths	Follow a single shortest path between source and destination	Can travel down any of the shortest paths between source and destination
Load balancing	No	No	No	Yes, flows alternated down equal cost paths
Hardware	No new hardware needed	No new hardware needed	Yes, full OpenFlow enabled network needed	Yes, full OpenFlow enabled network needed

Table 5.1: Comparisons between STP and OpenFlow approaches

Table 5.1 gives a comparison between the ECMF and loop prevention implementation outlined in this project and previously implemented variations of STP as well as a comparison against the shortest path OpenFlow approach discussed in [27]. Note the ll seconds mentioned in the convergence time for the ECMF. This time factors in the 10 seconds it takes for the default Ryu

code to discover a link has gone down. The actual time to recalculate the tree and push configurations out to switches is much shorter than this. This default time can be reduced at the cost of introducing more load to the controller so it is heavily determined by the size of the network. The biggest advantage this approach has compared to the existing ones is the inclusion of load balancing across multiple links.

	ECMF OpenFlow	**TRILL**	**SPB**
Convergence	approx. 11s	Not given	Not given
Shortest path	Yes,uses a search algorithm to determine in advance	Yes, uses IS-IS to calculate	Yes, uses IS-IS to calculate
Loop prevention	Tree calculated, flow rules introduced to drop broadcast traffic on ports not included in tree	Uses a new TTL header as well as reverse path forwarding	Uses a reverse path forwarding check
Broadcast traffic	Can only travel down enabled paths	Multiple trees possible for sending traffic down	Multiple trees possible for broadcast packets to travel down
Unicast traffic	Can travel down any of the equally shortest paths between source and destination , forward and reverse paths symmetric	Travels down the optimal path between source and destination, decisions made on hop by hop basis , forward and reverse paths not symmetric	Traffic assigned to one of the possible equal cost paths between source and destination, forward and reverse paths symmetric
Load balancing	Yes	Yes	Yes
Hardware	Yes, full OpenFlow enabled network needed	Yes switches with new ASIC's needed	No

Table 5.2: Comparisons between Projects approach and STP successors

Table 5.2 compares the successors of STP to the loop prevention and ECMF dicussed. An important fact to note is that at this point the convergence time of Transparent Interconnection of Lots of Links (TRILL) and Shortest Path

Bridging (SPB) is relatively unknown. This is likely attributed to their slow pick up and the fact they have yet to be deployed fully in a production network. At this stage Cisco have their own version of SPB currently in development. Though the convergence times are not yet fully realised, these two successors to STP have been designed with the idea of large, high density datacenters in mind so their convergence time is likely by necessity to be very low.

This loop prevention and ECMF approach is not quite at a level were it could scale as well as the potential successors of STP. This can be attributed to the link failure detection of OpenFlow not being done locally on the switches thus the convergence time is higher as well as reliance on the single controller for controlling the entire network. Having distributed OpenFlow controllers, which is further discussed in section 5.3.2 would certainly improve the viability of the outlines approach. The major advantage of OpenFlow over these two proposed solutions is that it is far more extensible. If there is some desired forwarding behaviour wanting to be introduced to the network, it could be added by including an additional module to the controller allowing for easily customised forwarding. This extension would be significantly easier than modifying the switches in current networks to achieve the same effect, which may not always be possible depending on the desired behaviour.

To conclude the approach covered in this project offers features highly desired in modern networks which are absent from the older, currently used protocol Rapid Spanning Tree Protocol (RSTP) as well as the other OpenFlow approach discussed. However, compared to the currently proposed successors TRILL and SPB, there is still some work to be done in relation to the OpenFlow protocol in order to gain enough footing to provide a potential alternative.

5.3 Project and OpenFlow Limitations

5.3.1 Design Choice Limitations

Most of the design choices of this project stem from the limited OpenFlow version 1.2 support in OVS. Once OVS has implemented more of OpenFlow version 1.2, the ARP redirection controller could be extended to make use of the ability to introduce rules to the switches to drop packets based on the ARP response opcode. By implementing this in a real environment and not with OVS the loop prevention and ECMF module could be extended to using

the link speed and capabilities of a link in order to determine its cost rather than the arbitrary assignment via a static file which is currently implemented. The introduction of these features would allow for much cleaner and improved controllers.

5.3.2 OpenFlow Limitations

Though OpenFlow does have a lot of upsides to its approach and concept there are also certain limitations to its use. One of the most apparent is the introduction of a delay between the switch and controller. Though this delay is also apparent in the traditional tightly coupled switch due to the slower speed of the control plane, the effect here is somewhat larger due to the physical separation of these two planes. For small to medium networks this delay is not such a problem and the extensibility and abstraction that OpenFlow provides in terms of the switch configuration more than cover for this downside.

One of the other major limitations to OpenFlow which impacts on the loop prevention and ECMF approach more significantly is having a single controller controlling the entire network. Though it is great to have a logically centralised view of the network in terms of controllability, in current implementation OpenFlow introduces a single point of failure into the network. As has already been discussed, due to the nature of networks this is somewhat undesired and it would not be recommended running an entire network of a single controller for this reason.

This particular issue has already been realised within the OpenFlow community and work is already being done to address this. In terms of the OpenFlow specifications, OpenFlow 1.2 does introduce the idea of having a master and slave controller setup, allowing for fast fail over should one controller go down and adding potential for load balancing. OVS does partially implement this feature however the implementation only allows for a backup inactive slave to be used rather than two active ones. Due to this, splitting the loop prevention and ECMF code across multiple controllers was not explored as OVS does not yet allow for this. However, this is not the only approach taken, [29] is one example of research being done to allow for a more scalable OpenFlow enabled network. It looks at introducing a distributed system of OpenFlow controllers to allow for the handling of more flows in the network. This also enables further redundancy in terms of the OpenFlow control.

Adapting the loop prevention and ECMF approach to an distributed set of OpenFlow controllers would be largely beneficial for the approach. Not only because it would allow more flows to be handled but also because it allows for the work for both the sending and receiving of the topology discovery Link Layer Discovery Packet (LLDP)s to potentially be distributed between these controllers. Allowing for the topology discovery module to handle more links and switches. This approach would also allow for the reduction of the link expiry timer in the Ryu topology discovery code as well. This link expiry timer is the time taken for two of the LLDP messages to be sent. Currently this is set to 5 seconds to prevent too much load on the controller as it is generating all of the LLDP for each switch. Using a distributed network would allow for this to be smaller and thus ultimately reduce the convergence time. At this stage this loop prevention and ECMF approach would only be suited to smaller networks but given a distributed system as described above its reach would extend to larger networks and would result in a much more scalable solution. The work required to achieve such a system lies outside the scope of this project, but would certainly be a major factor in future work.

Chapter 6

Conclusions

6.1 Contributions Made

This project has explored the ways in which OpenFlow can be used to address some of the weakness of existing protocols that have started to become more visible as the size and complexity of networks has increased. Two designs addressing the overhead and security of Address Resolution Protocol (ARP) have been implemented and evaluted. These are:

- **ARP Redirection and Response**
  The implementation in which the OpenFlow controller has be designed to forward traffic to a trusted host running an ISC-DHCP server in order to generate trusted ARP responses. This allows the controller to drop any other ARP responses as they are unsolicted and therefore potentially malicious.
- **ARP Proxy**
  This implementation is an re-imagining of the approaches already discussed in Section 3.1 for the Ryu framework. It allows the controller to become responsible for generating the ARP responses eliminating much of the overhead caused by the broadcast of ARP requests on the network.

Along with these proposed OpenFlow designs to reducing the negative effects of ARP, this project has also looked into how the same OpenFlow principles can be used design an alternative to Spanning Tree Protocol (STP). Discussing what the advantages and drawbacks of this approach would be. The design put forward by this project was:

- **Loop prevention and Equal Cost Multipath Forwarding (ECMF)**

An implementation that takes the default Ryu topology discovery code and uses the information learned to introduce rules to the OpenFlow switches to prevent loops in order to keep all ports active. It also implements a forwarding module to distribute flows across shortest equal paths between the source and destination, allowing for a load balanced network along with a network that guarantees shortest paths between all switches on the network.

All approaches put forward have been evaluated and compared against both the existing solutions and the more popular proposed successors discussed in Chapter 3.

6.2 Future work

OpenFlow is a rapidly growing, constantly evolving entity. This is due to its flexibility and the increased interest in virtualisation. New features are being introduced to the specifications on an regular basis. As a result it is hard to predict whether features may be added that benefit the approaches outlined in this project or not and how exactly they might benefit it. In the time of writing this report the specification for OpenFlow version 1.4[13] was released and is currently under ratification. This new OpenFlow specification re-introduces port states such as NO_FLOOD that were removed in version 1.1. Once Ryu has been updated to support OpenFlow version 1.4 this would allow the loop prevention code to be updated from its current rule based approach to dropping broadcast traffic, to a port based approach to handle the loops. This would reduce the rules needing to be introduced to the switch as well as simplifying the loop prevention process.

There were a number of design limitations that could be addressed once Open vSwitch (OVS) has implemented more of OpenFlow version 1.1 and up, such as redesigning the controller for the ARP redirection approach to allow for flow entries to match on the ARP operation code held within a packet. This would allow for the switches themselves to drop responses from untrusted ports as opposed to forwarding them up to the controller therefore reducing the load on the controller.

The other major piece of work to be done would be to redesign the loop prevention code to allow it to be easily extended to a distributed OpenFlow controller

environment as discussed in Section 5.3.2. This would be the most valuable extension to the loop prevention and Equal Cost Multipath Forwarding (ECMF) controller presented in this project.

6.3 Conclusion

This project has focused on using the flexibility OpenFlow provides via its centralised software controller along with the dynamic reconfiguration of switches to explore potential solutions to the shortcomings of existing network protocols. It has focused on the design of solutions that do not require a full reconfiguration of a network in order to deploy, but rather ones that require only minimal changes to the underlying network in order to gain improvements. Given the limitations discussed, it would be possible to introduce both of the proposed ARP solutions to an OpenFlow enabled network. However, the loop prevention and ECMF controller, although allowing for certain desirable features, would only be suitable for smaller network environments due to the single point of failure currently introduced by OpenFlow as well as designs not yet fully realised in an OpenFlow environment.

References

[1] AVAYA. Compare and Contrast SPB an TRILL, 2011. http://www.avaya.com/uk/resource/assets/whitepapers/SPB-TRILL_Compare_Contrast-DN4634.pdf (Accessed October 2013).

[2] P. Biondi. Scapy. www.secdev.org/projects/scapy/ (Accessed October 2013).

[3] D. Bruschi, A. Ornaghi, and E. Rosti. S-arp: a secure address resolution protocol. *Computer Security Applications Conference*, pages 66–74, 2003.

[4] O. M. E. Committee. Software-Defined Networking: The New Norm for Networks, 2012. https://www.opennetworking.org/sdn-resources/sdn-library/whitepapers (Accessed October 2013).

[5] Internet Systems Consortium. ISC DHCP: Enterprise grade solution for configuration needs, 2013. https://www.isc.org/downloads/dhcp/ (Accessed October 2013).

[6] S. Dangol, S. Selvakumar, and M. Brindha. Genuine arp (garp): a broadcast based stateful authentication protocol. *ACM SIGSOFT Software Engineering Notes*, 36:1–10, 2011.

[7] David Erickson. The Beacon OpenFlow Controller. In *HotSDN*. ACM, 2013.

[8] Project Floodlight. Floodlight, 2013. https://github.com/floodlight/floodlight (Accessed October 2013).

[9] Open Networking Foundation. OpenFlow Switch Specification 1.0.0, Dec 2009. https://www.opennetworking.org/sdn-resources/onf-specifications/openflow (Accessed October 2013).

[10] Open Networking Foundation. OpenFlow Switch Specification 1.1.0, Feb 2011. https://www.opennetworking.org/sdn-resources/onf-

specifications/openflow (Accessed October 2013).

[11] Open Networking Foundation. OpenFlow Switch Specification 1.2, Dec 2011. https://www.opennetworking.org/sdn-resources/onf-specifications/openflow (Accessed October 2013).

[12] Open Networking Foundation. OpenFlow Switch Specification 1.3.0, Jun 2012. https://www.opennetworking.org/sdn-resources/onf-specifications/openflow (Accessed October 2013).

[13] Open Networking Foundation. OpenFlow Switch Specification 1.4, Aug 2013. https://www.opennetworking.org/sdn-resources/onf-specifications/openflow (Accessed October 2013).

[14] G. Gibb. OpenFlow: Basic Spanning Tree. http://archive.openflow.org/wk (Accessed October 2013).

[15] H. Grohne and T. Szczepanek. Pypureomapi: ISC DHCP OMAPI protocol implementaion in Python. code.google.com/p/pypureomapi (Accessed October 2013).

[16] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. Nox: towards an operating system for networks. *SIGCOMM Comput. Commun. Rev.*, 38(3):105–110, jul 2008.

[17] IEEE. 802.1D Bridging Standard, IEEE Standard for Local and metropolitan area networks: Media Access Control (MAC) Bridges, 2004. incorporates 802.1w (RSTP).

[18] IEEE. 802.1Q VLAN standard, IEEE Standards for Local and metropolitan area networks: Virtual Bridged Local Area Networks, 2006. incorporates 802.1v (VSTP) and 802.1s (MSTP).

[19] IEEE. 802.1aq-2012, IEEE Standard for Local and metropolitan Area Networks: Media Access Control (MAC) Bridges and Virtual Bridged Local Area Networks, 2012.

[20] M. Jarschel, S. Oechsner, D. Schlosser, R. Pries, S. Goll, and P. Tran-Gia. Modeling and performance evaluation of an openflow architecture. In *Proceedings of the 23rd International Teletraffic Congress*, ITC '11, pages 1–7, 2011.

[21] M. McCuley. Pox. https://github.com/noxrepo/pox (Accessed October

2013).

[22] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, mar 2008.

[23] R. Perlman. An Algorithm for Distributed Computation of a Spanning Tree in an Extended LAN. *SIGCOMM '85 Proceedings of the ninth symposium on Data communications*, 15:44–53, 1985.

[24] R. Perlman, D. Eastlake, D. Dutt, S. Gai, and A. Ghanwani. Routing Bridges (RBridges): Base Protocol Specification. *RFC6325*, 2011, July.

[25] B. Pfaff, J. Pettit, T. Koponen, K. Amidon, M. Casado, and S. Shenker. Extending Networking into the Virtualization Layer. *HotNets*, 8:22–23, 2009.

[26] D. Plummer. Ethernet Address Resolution Protocol: Or Converting Network Protocol Addresses to 48.bit Ethernet Address for Transmission on Ethernet Hardware. *Internet RFC 826*, 1982.

[27] J. Soeurt and Hoogendoorn I. Shortest path forwarding using openflow. Master's thesis, University of Amsterdam, 2012.

[28] Nippon Telegraph and Telephone Corporation. Ryu, a component-based software-defined networking framework, 2013. http://osrg.github.io/ryu (Accessed October 2013).

[29] A Tootoonchian and Y Ganjali. HyperFlow: A Distributed Control Plane for OpenFlow, April 2010.

Appendix A

ARP Redirection and Proxy in OpenFlow

A.1 ARP response script

```
1  import threading
2  import time
3  import pypureomapi
4
5  from scapy.all import *
6
7  KEYNAME="defomapi"
8  BASE64_ENCODED_KEY="DaJqBjuYJH1Fb54WdOmA=="
9
10 dhcp_server_ip="0.0.0.0"
11 port = 7911 # Port of the omapi service
12
13 class ArpResponse(threading.Thread):
14         def __init__(self, pkt):
15                 self.pkt = pkt
16                 super(ArpResponse, self).__init__()
17
18         def run(self):
19                 a = ARP()
20                 a.op = 2
21                 a.psrc = self.pkt[ARP].pdst
22                 a.pdst = self.pkt[ARP].psrc
23                 a.hwdst = self.pkt[ARP].hwsrc
24     try:
25       # connect to DHCP server
```

```
26                          o = pypureomapi.Omapi(dhcp_server_ip,port,
                                KEYNAME, BASE64_ENCODED_KEY)
27          # find mac address from a given IP
28          mac = o.lookup_mac(a.psrc)
29                          a.hwsrc = mac
30          # create ARP response
31                          p = Ether(dst=self.pkt[ARP].hwsrc) / a
32                          print "Src: %s, Ip: %s , Dst %s, Ip: %s "
                               % (a.hwsrc, a.psrc, a.hwdst, a.pdst)
33                          sendp(p)
34                  except pypureomapi.OmapiErrorNotFound:
35                          print "%s is currently not assigned" % (a.
                               psrc)
36                  except pypureomapi.OmapiError, err:
37                          print "an error occured: %r" % (err,)
38
39  def arp_request_reply(pkt):
40          if ARP in pkt and pkt[ARP].op == 1:
41                  ArpResponse(pkt).start()
42
43  sniff(prn=arp_request_reply, filter='arp', store=0)
```

A.2 ARP Proxy controller

```
1   def map_get(self, src, dst, ev):
2     pkt = packet.Packet(array.array('B', ev.msg.data))
3     d = None
4     # get arp header so we can use its values
5           for p in pkt:
6       if p.protocol_name == 'arp':
7         d = p
8     try:
9
10      e = ethernet.ethernet(src , self.ip_to_mac[d.dst_ip], ether.
          ETH_TYPE_ARP)
11      a = arp.arp(1, ether.ETH_TYPE_IP, 6, 4, 2, self.ip_to_mac[d.
          dst_ip], d.dst_ip, src, self.mac_to_ip[src])
12      b_pkt = packet.Packet()
13      b_pkt.add_protocol(e)
14      b_pkt.add_protocol(a)
15      b_pkt.serialize()
16      print "from: ", haddr_to_str(self.ip_to_mac[d.dst_ip]), d.
          dst_ip
17      print "to: ",haddr_to_str(src) ,self.mac_to_ip[src]
```

```
18        return b_pkt
19
20      except KeyError:
21        print "not yet learned"
22
23      return None
24
25  #......... Packet_in method Snippet...........#
26
27  if _eth_type == ether.ETH_TYPE_ARP:
28        reply = self.map_get(src,dst,ev)
29        if(reply == None):
30          out_port = ofproto.OFPP_FLOOD
31        else:
32          actions = [datapath.ofproto_parser.OFPActionOutput(in_port,
               max_len=0)]
33          out = datapath.ofproto_parser.OFPPacketOut(
34                datapath=datapath, buffer_id=0xffffffff, in_port=
                     ofproto_v1_2.OFPP_ANY,
35                actions=actions, data=reply.data)
36                datapath.send_msg(out)
37          return
```

Appendix B

Loop prevention and ECMF

B.1 Loop prevention module - Tree calculation

```
1  while len(queue) > 0:
2    # return link/path with lowest cost
3    current_path = queue.pop(self.get_lowest_cost(queue))
4    current_dpid = current_path[1]
5    if current_dpid in visited_switches:
6        continue
7    #otherwise add it to the list of visited switches
8    visited_switches.add(current_dpid)
9    try:
10     all_datapaths.remove(current_dpid)
11   except:
12     print "trying to remove: ", current_dpid
13
14   # finds the lowest cost path to this switch
15   lowest_cost_port = self.find_lowest_cost_port(path_to_switch,
        current_dpid)[1]
16   # ignore the in port of the lowest cost path, so that it is
        included in spanning tree
17   ignore_links[current_dpid].append(lowest_cost_port)
18   # for all ports on switch finds unexplored links
19   for port in self.topology[current_dpid]:
20       if port in ignore_links[current_dpid]:
21     continue
22       dst_dp = self.topology[current_dpid][port][self.DEST_DP]
23       dst_port = self.topology[current_dpid][port][self.DEST_PORT]
24       if not self.up_both_ways(current_dpid, port, dst_dp,
            dst_port):
25      continue
```

```
26        # if the dest switch has been visited already then drop the
              link
27        if dst_dp in visited_switches:
28      drop_links[current_dpid].append(port)
29      drop_links[dst_dp].append(dst_port)
30      continue
31        else:
32      # compute the cost of taking this link and add it to the queue
33      cost_to_here = current_path[0]
34      reverse_cost = self.metrics[dpid_to_str(dst_dp)][
            port_no_to_str(dst_port)]
35      port_cost = self.topology[current_dpid][port][self.LINK_METRIC
            ]
36      total_cost = port_cost + reverse_cost + cost_to_here
37      queue.append([total_cost, dst_dp, dst_port])
38      path_to_switch[dst_dp].append([total_cost, dst_port])
```

B.2 ECMF - shortest paths between pair of switches calculation

```
1  def find_shortest_paths(self, initial_dpid, final_dpid):
2    visited_switches = set([initial_dpid])
3    initial_neighbours = []
4    ports = []
5    # the set of neigbours we can reach from the src datapath
6    for port in self.topology[initial_dpid]:
7        dst_switch = self.topology[initial_dpid][port][self.DEST_DP]
8              if dst_switch not in visited_switches:
9      destination = self.topology[initial_dpid][port]
10     cost = destination[self.LINK_METRIC]
11     initial_neighbours.append([cost, [cost], [initial_dpid, port]+
           destination[:-1] ])
12
13         paths = self.search_for_destination(visited_switches,
               initial_neighbours, final_dpid)
14         return paths
15
16 def search_for_destination(self, visited_switches, neighbours,
       final_dpid):
17   paths_to_take = []
18   least_cost = 0
19   while len(neighbours) != 0:
20       # get the path to neighbouring switch
```

```
21          path_to_neighbour = neighbours.pop(self.select_neighbour(
                neighbours))
22          last_path = len(path_to_neighbour)-1
23          neighbour_dpid = path_to_neighbour[last_path][2]
24          # if we havent hi our target yet then record that we have
                visited this switch
25          if neighbour_dpid != final_dpid:
26              visited_switches.add(neighbour_dpid)
27          if path_to_neighbour[last_path][4] == "UP":
28        if neighbour_dpid == final_dpid:
29            # if this is the first time we had the pairing then we
                  want to capture all equal cost paths
30            if path_to_neighbour[0] == least_cost or least_cost == 0:
31          # caching discovered paths
32          paths_to_take.append(path_to_neighbour)
33          least_cost = path_to_neighbour[0]
34            # else return the path that we are up to
35            else:
36          return paths_to_take
37        # take the path to this neighbour and add an entry to each of
              its neighbours
38        for port in self.topology[neighbour_dpid]:
39            path = self.topology[neighbour_dpid][port]
40            dst_switch = path[self.DEST_DP]
41            if dst_switch not in visited_switches:
42          neighbours.append(self.extend_path(path_to_neighbour, path,
                last_path, port))
43
44    # when only equal cost paths exist between switches this code
          will get hit
45    return paths_to_take
```