Ishaan Aggarwal

Implementation and Evaluation of ELK, an ARP scalability enhancement

Computer Science Tripos, Part II

Corpus Christi College

May 19, 2011

Proforma

Name:	**Ishaan Aggarwal**
College:	**Corpus Christi College**
Project Title:	**Implementation and Evaluation of ELK, an ARP scalability enhancement**
Examination:	**Computer Science Tripos, Part II, 2010-2011**
Word Count:	**Approximately 11950 words**
Project Originator:	Malcolm Scott
Supervisor:	Dr A. W. Moore

Original Aims of the Project

I aimed to develop a prototype of the ELK system, which would replace the conventional ARP protocol in a local network. Using several metrics, I intended to evaluate the performance of this prototype in comparison with that of ARP.

Work Completed

I successfully completed the development and evaluation of the ELK system. It was an improvement over conventional ARP which has been shown through evidence gathered using several metrics. I have also put forward some areas in ELK for further research and development.

Special Difficulties

None.

Declaration

I, Ishaan Aggarwal of Corpus Christi College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed

Date

Contents

List of Figures

Acknowledgements

For their work on ELK, advice, support and encouragement, I wish to thank Malcolm Scott, and my supervisor Andrew Moore.

For the long hours spent helping me set up and debug the test harness, thanks to Haris Rotsos.

My utmost gratitude is owed to David Greaves for providing me continual support throughout the project and my time at Cambridge.

I would also like to thank the NOX developers who have provided almost instant help with all my queries.

My deepest thanks to my family for providing me moral support and backing me up in my decisions.

For their invaluable feedback on this dissertation, I would like to thank Dhivina Gagoomal, Ayesha Sengupta, and Vaughan Whirtoff.

Last but not least, my friends who have always been there for me. Thank you

Chapter 1

Introduction

Address Resolution Protocol (ARP) [1] is the Internet standard for dynamic address resolution and it has dominated this arena for a couple of decades. However, during its lifetime it has shown several weaknesses, with scalability [2] being one of the major problems besides the various security vulnerabilities. Scalability has been an issue in data centres where the number of ARP broadcasts issued within the network is very large, causing decreased bandwidth and added burden on hosts to process them [3]. An enhanced ARP replacement should produce a reduced number of broadcast storms which would improve performance, while being compatible with the existing hosts. It should be backward-compatible and should keep the hardware and software update required to the bare minimum. A viable solution called Enhanced Lookup (ELK) Directory Service, satisfying all these requirements, was suggested by Scott *et al.* in the MOOSE paper [4]. ELK has never been implemented and hence, there exists no evidence to support its concept. In this paper, I have described the implementation and evaluated the performance of ELK, so that it can be considered as a feasible replacement for the basic ARP.

1.1 Background

1.1.1 MAC and IP Address

Network technologies are often divided into layers based on the OSI model (see Appendix A). The most common network layer protocol used in networks is Ethernet. It is used to deliver data frames to the machines in a network based

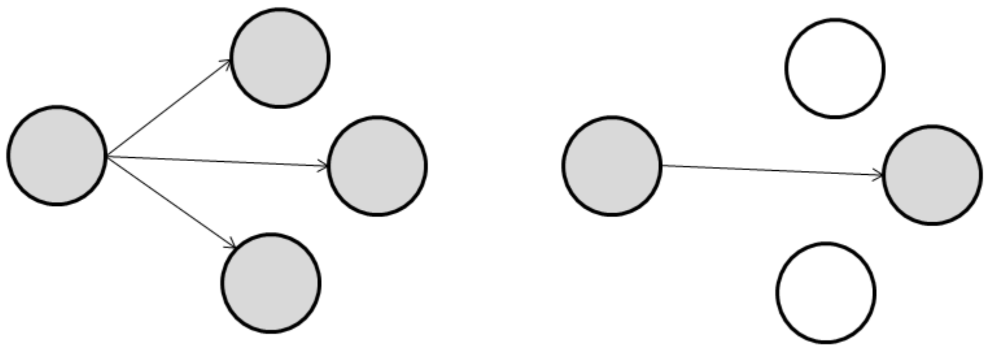


Figure 1.1: Broadcast (left) and Unicast (right)

on their Ethernet addresses, also known as hardware addresses or Media Access Control (MAC) addresses. This essentially controls the data which is accessed from the network by the node. A different address is assigned to every interface of a networking component. For instance, a host with four physical Ethernet interfaces will have four different MAC address, one for each interface. The important property of Ethernet addresses which enables this design is that all the addresses are unique. This ensures that the Ethernet frame is delivered to the correct interface without the need of any other information. An Ethernet Address is 48 bits or 6 bytes long, which is written typically as six pairs of hexadecimal digits, such as `00:06:8C:12:34:56`.

Broadcasting (see Figure 1.1) means transmitting the Ethernet frame to every device on the network. The scope of such messages is limited to a certain broadcast domain. For instance, in Ethernet, the boundary between different domains is demarcated by routers and other higher-layer devices. Broadcast addressing is not supported by all protocols and is mainly confined to local area networks (LANs) such as Ethernet, as it does not affect the performance of the network as much as in the case of wide area networks (WANs). Ethernet frames are broadcast when all the bits of the destination Ethernet address are set to one which gives an address of `ff:ff:ff:ff:ff:ff`. In such a case, by convention, every node in the Ethernet broadcast domain is required to accept such frames. Broadcast is employed when either certain information needs to be transmitted to all nodes, or when the destination hardware address is unknown. This addressing is commonly used in Dynamic Host Configuration Protocol (DHCP) [5] and Address Resolution Protocol (ARP) [1].

Internet Protocol (IP) [6] is the principal data-link protocol which is responsible for routing and relaying packets across network boundaries throughout the internet. This is achieved using logical addresses known as IP addresses, which

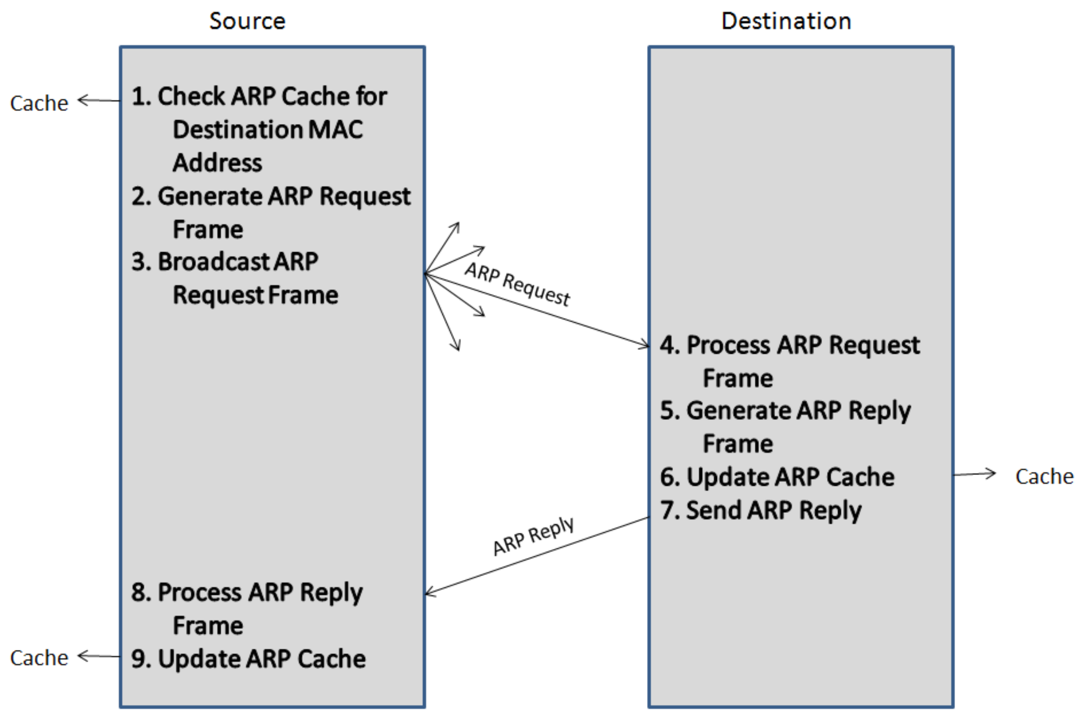
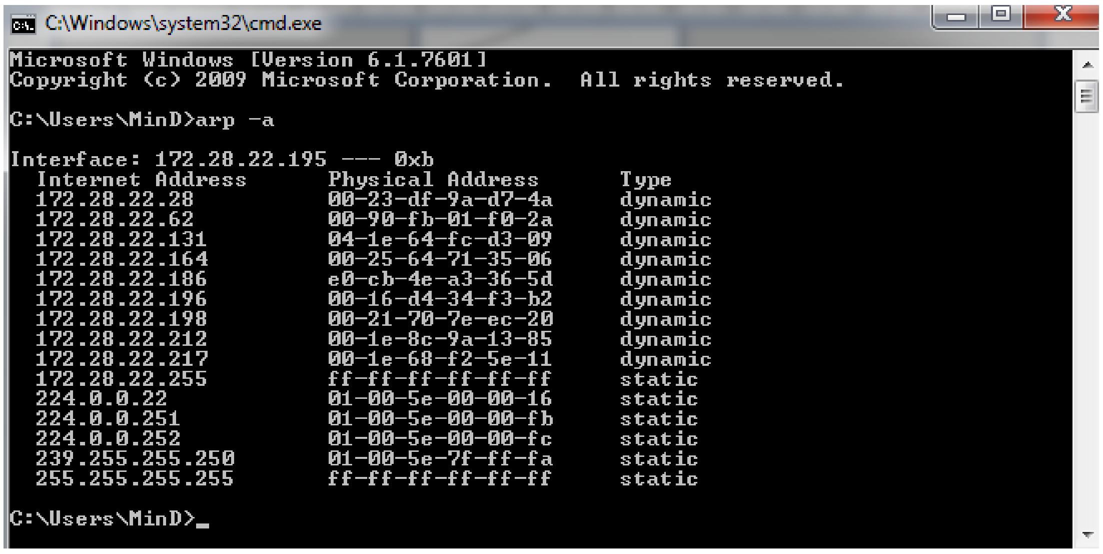

Source
Destination
Cache
1. Check ARP Cache for Destination MAC Address
2. Generate ARP Request Frame
3. Broadcast ARP Request Frame
ARP Request
4. Process ARP Request Frame
5. Generate ARP Reply Frame
6. Update ARP Cache
7. Send ARP Reply
Cache
ARP Reply
8. Process ARP Reply Frame
Cache
9. Update ARP Cache

Figure 1.2: Typical ARP Transaction

are allocated to every end host in the internet. The address allocation can be static (administrator configured) or dynamic (for instance, using DHCP). There are two versions of this protocol which are commonly used - IP version 4 (IPv4) and IP version 6 (IPv6). The latter has not yet been widely accepted and in this dissertation I will restrict my discussion to IPv4 (hereinafter referred to as IP) only. For IPv4, an IP address is 32 bits long, which is divided into 4 octets (8 bits long) with each octet representing a decimal number ranging from 0 to 255, for example, `192.168.1.125`.

1.1.2 ARP

Address Resolution Protocol (ARP) was first suggested in RFC 826 [1] in 1982 and is still in use. It is a networking protocol which is used to dynamically discover the network layer address of a host in a network using its data link address. In a typical network, this mapping would be between the IP (data-link layer) address and the Ethernet (network layer) address. This protocol plays the important role of linking the data-link and network layers together and allowing them to work together. ARP is a control protocol which is not covered by the



```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\MinD>arp -a

Interface: 172.28.22.195 --- 0xb
  Internet Address      Physical Address      Type
  172.28.22.28          00-23-df-9a-d7-4a     dynamic
  172.28.22.62          00-90-fb-01-f0-2a     dynamic
  172.28.22.131         04-1e-64-fc-d3-09     dynamic
  172.28.22.164         00-25-64-71-35-06     dynamic
  172.28.22.186         e0-cb-4e-a3-36-5d     dynamic
  172.28.22.196         00-16-d4-34-f3-b2     dynamic
  172.28.22.198         00-21-70-7e-ec-20     dynamic
  172.28.22.212         00-1e-8c-9a-13-85     dynamic
  172.28.22.217         00-1e-68-f2-5e-11     dynamic
  172.28.22.255         ff-ff-ff-ff-ff-ff     static
  224.0.0.22            01-00-5e-00-00-16     static
  224.0.0.251           01-00-5e-00-00-fb     static
  224.0.0.252           01-00-5e-00-00-fc     static
  239.255.255.250       01-00-5e-7f-ff-fa     static
  255.255.255.255       ff-ff-ff-ff-ff-ff     static

C:\Users\MinD>_
```

Figure 1.3: An example of Windows 7's ARP Table

OSI model of data protocols. The byte-level format details for ARP are provided in Appendix B. The other alternative to this dynamic address resolution is static direct hardware-to-IP mapping which, on its own, is uncommon.

In a network, it is common for hosts to be allocated a new IP Address when they change their location, when their address lease expires, when they restart, or for other reasons. The sender host which is trying to communicate will use ARP to discover the new IP to MAC address mapping for the target host. For instance, host A (sender) with IP address IP-A and Ethernet address MAC-A wants to find host B (target) with IP address IP-B in the same subnet. Host A will broadcast an ARP query packet (see Figure 1.2) with its own IP and MAC address and Host B's IP Address on its NIC. Since it is a broadcast message, every host in the subnet will accept it but they will only send a reply if they own the target IP address. Host B, with IP address IP-B, will send a unicast ARP reply to A with its own MAC address. On receipt, A will record the mapping of B and forward to B the data (for B). It needs to be noted that ARP packets do not have IP headers and hence ARP can resolve addresses only within the same physical network, and the target IP address needs to be from within the same subnet as the sender IP address. Since ARP operates below the network layer, it cannot pass from one network to another via the routers.

In order to reduce repetitive ARP broadcast messages for the same target IP address, all hosts maintain an ARP table (see Figure 1.3) locally which is referred to whenever a mapping is required. An ARP request is only broadcasted if there

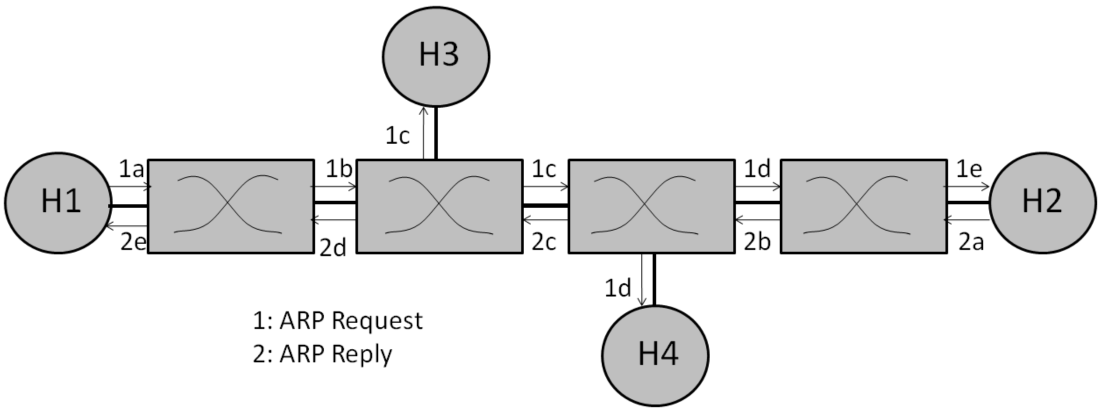


Figure 1.4: Network View of ARP Transaction

is no suitable entry in the table (Step 1, Figure 1.2). A host will use all the information available from the ARP packets received to keeps its table up to date (Step 6 and 9, Figure 1.2). The table essentially works like a cache whereby the entries expire if they have not been used for a certain amount of time. For instance, by default in Microsoft Windows XP and Microsoft Windows Server 2003 Operating Systems, stale entries expire after 4 minutes [7].

ARP, as described above, works well in small network domains as the number of broadcast messages in such a network is limited, and ARP messages use up only a very small proportion of the available bandwidth. However, broadcasting messages frequently (see Figure 1.4) across a whole network of a few ten thousand hosts would not only use more bandwidth, but would also force hosts to unnecessarily process the extra broadcast messages. This problem renders ARP unscalable. A study at Carnegie Mellon University [2] measured the number of ARP queries which arrived at a host in a network of 2456 hosts. During the peak time, the host received 1150 ARPs every second and it averaged 89 ARPs per second which adds up to about 45 kbps of traffic. By extrapolating, for a network of one million nodes, 468240 ARPs per second are expected to arrive at each host during the peak time. This is about 239 Mbps of ARP traffic which can cripple a 100 Mbps LAN connection. ARP also adds significant burden on hosts as every host now needs to handle about half a million ARP packets per second.

1.1.3 ELK

Enhanced Lookup (ELK) is a transparent directory service (see Figure 1.5) which is deployed in a network to reduce the amount of ARP traffic by attempting to

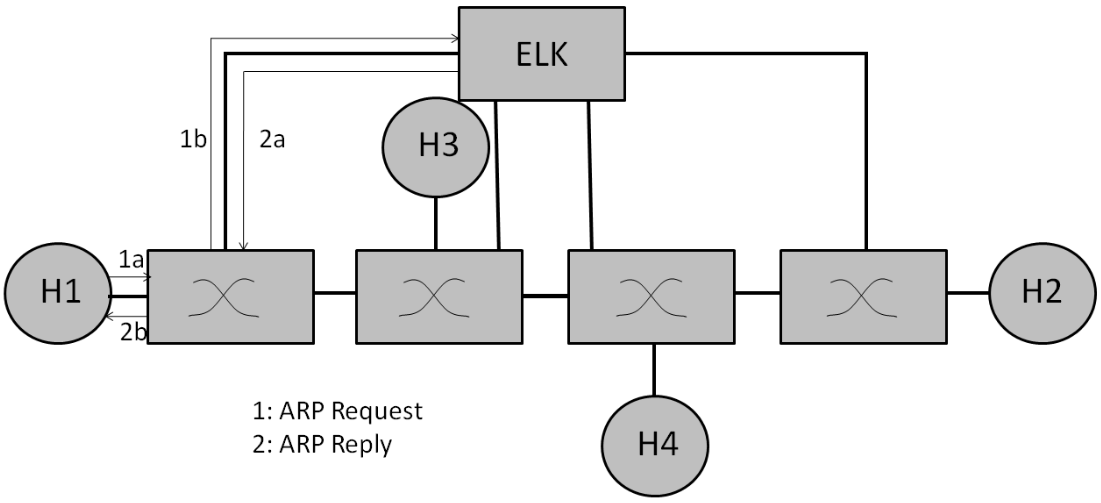


Figure 1.5: Conceptual Network View of ARP Transaction with ELK

convert as many broadcast messages into unicast messages as possible. This system continues to use ARP as its underlying protocol for address resolution and hence it is backward compatible. The basic requirements of the system are a server running ELK, and a way to intercept ARP packets at the switches. This interception of ARP packets could use OpenFlow protocol, as employed in my implementation, or any other available technologies that are capable of doing so. Hosts and any other networking components such as routers, are completely unaware about ELK and do not require any modifications.

ELK maintains an ARP table which has the IP address to MAC address mappings for the network. During the initial learning phase, it learns these mappings using broadcast messages and then regularly updates the table using the information from incoming ARP traffic. The table entries timeout if they have not been refreshed for a certain time. In the network, OpenFlow switches forward all the ARP requests sent by any host to ELK for processing, instead of broadcasting it through the network. They behave like normal switches for all other packets including ARP replies from any source and ARP Requests sent by ELK. This essentially cuts down the amount of unnecessary broadcast traffic and reduces the packet processing burden on hosts. When combined with other optimisations in ELK, such as using pre-loaded table, this setup produces a much more efficient address resolution protocol.

1.2 Related Work

The scalability problem of ARP was soon realised, and there have been papers attempting to overcome this issue. Myers *et al.* [2] suggested the use of a directory service for address resolution to support a million hosts without the need of broadcast messages. However, this approach involved replacing Ethernet entirely with a new architecture, which by no means is practical as Ethernet is a ubiquitous link layer protocol.

SEATTLE [8], a DHT-based replacement for ARP, was suggested by Kim *et al.* Each participating switch maintains a distributed hash table (DHT), which contains the mapping of host MAC addresses to the switch to which it is connected. The switch location information is propagated via the routing protocol. However, the software complexity added by this approach in switches is considerable, and might even undermine the gains from eliminating the broadcast messages.

Scott *et al.* [4] gave a precis of ELK, but no detailed specifications, proof of concept, or implementations of it, have ever been produced. In this dissertation, I present the first ever detailed specifications and implementation for ELK.

1.3 Aims

I aimed to implement ELK in conjunction with OpenFlow controller and switches, and compare its efficiency against basic ARP (without any enhancements). More specifically, I aimed to show that the overall volume of traffic generated for address resolution is lower for ELK compared to that of basic ARP. The main idea behind this was to reduce the number of broadcast messages which are sent by ARP by maintaining a lookup directory. I also plan to evaluate the effect of maintaining an ARP proxy in the network on the latency of the ARP replies.

1.4 Relevant Courses

I found the following modules taught over the course of three years useful for my project: Programming in Java, Programming Methods and Further Java (to design a multi-threaded server in Java), Digital communication I and Principles of Communication (to understand ARP, Ethernet and other networking concepts), Programming in C and C++ (to program in NOX), Concurrent Systems (to understand synchronization between threads), Algorithms I and II (for the knowledge of data structures, algorithm design and implementation), Software Design and Software Engineering (design techniques and project management skills), and Unix Tools.

Beyond the courses taught in the Tripos, I was expected to understand technologies which were required for the project. These included OpenFlow protocol and NOX (OpenFlow controller framework). During this learning process, I also acquired the ability to implement network protocols efficiently and gained the relevant programming skills.

Chapter 2

Preparation

2.1 Technologies Used

Research was done to decide upon the suitable technologies for the project.

2.1.1 Java and Jpcap

The ELK server implementation was aimed to be a prototype, and not a deployable system, so as to prove its advantages, as predicted [4]. Java is considered to be a good prototyping language [9] by researchers as it is a high level language which enables rapid evolutionary prototyping and manages the memory, for example by allocating it and performing garbage collection, for the programmer. Furthermore, it provides good support for multithreading which is important for this project. The platform independent property of Java was crucial during the development and testing phase, as it abstracted away the architecture dependencies of the machines running on different architectures and operating systems. The drawback of using Java is that since it deals in high level concepts, it adds latency and is not fully optimised and hence, it has a performance hit to the project. However, since the project is a prototype development and latency impact is minimal, the advantages outweigh this drawback. Hence, the reason Java was chosen for the server implementation was that it struck a good balance between its performance and the features it provided to aid in the development.

Jpcap is a Java library which provides the ability to capture and send network packets [10], including ARP packets, from the application via a network interface. It is based on libpcap and the Raw Socket API, both of which are not natively available in Java. Libpcap (equivalently winpcap for windows) is a system-independent interface written in C [11], which provides the functionality to capture raw data from the network through the machines adapter independent from hardware. Raw Socket refers to a socket that bypasses the encapsulation process for the packet set by the operating system, and allows applications to interact directly with the network. Jpcap is essentially a wrapper which ports the libpcap functionality from C into Java.

Jpcap is not the only Java library of its type. JNetPcap[12] is another available library based on libpcap, which provides similar functionality. Both of these libraries are suited for this project as they handle ARP equally well. However in this case, Jpcap was the choice of library used in the server to send and receive ARP packets as it is widely accepted, is in active development, and has a simpler API. It also provides support on all the various operating systems used for development and testing, including Windows 7, Ubuntu 10.10 and CentOS 5.6.

2.1.2 OpenFlow and NOX

In order to process ARP requests, the ELK server requires the requests in the network to be redirected and sent to it. OpenFlow [13], a programmable network control plane framework, fits in as the perfect candidate for my requirements, as it allows, among other features, reconfiguration of the behaviour of an OpenFlow-compatible Ethernet switch (hereinafter referred to as the **OpenFlow switch**) and experimentation with network protocols. OpenFlow retains the packet forwarding function in the switch, but moves the high-level routing decisions out of the switch to a separate software controller which typically runs on a server, thereby allowing the programmer to easily manage traffic in the software. The switch communicates with the controller over a secure channel using the OpenFlow protocol (see Figure 2.1).

The OpenFlow protocol maintains a table in the memory of each switch which contains *flow* entries along with the coupled *actions*. A flow entry constitutes some of the fields from the headers of the data-link, the network, and the transport layers (see Figure 2.2), which are used to identify the incoming packets. For instance, a TCP connection or all packets from a particular MAC address would be considered as flows. Defined precisely, a *flow* is a set of packets whose headers give an exact match on any number of these fields in the flow entry, and/or a

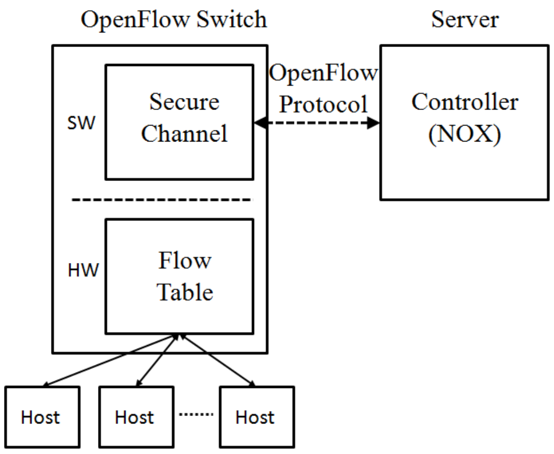


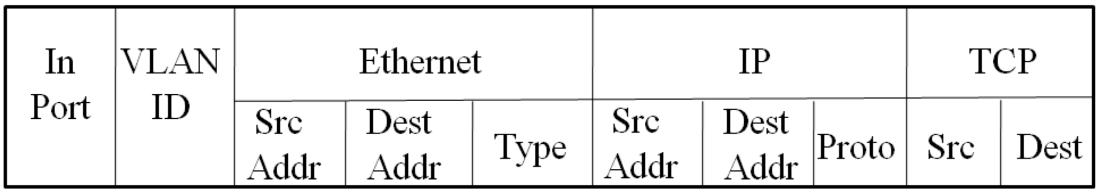
Figure 2.1: OpenFlow switch and controller setup [13]

partial match on IP Source and Destination Addresses [14]. An *action* can consist of any permutation of the following:

- Forward packets through a specified port.
- Encapsulate the packet in an OpenFlow header and send it to the controller.
- Drop packet.

When an OpenFlow switch receives a packet, it examines the header of the packet. If there exists a flow entry corresponding to the packet header, then the switch performs the associated action. In the case of no corresponding flow, the packet is encapsulated and sent to the OpenFlow controller. The controller, after examining the packet header based on the rules it has, determines and performs the suitable action for the packet. It may also send an OpenFlow packet containing this information back to the switch for it to add to its flow entry table for the future packets of the same flow.

Deploying OpenFlow in the network to implement new network protocol is a beneficial strategy. This is because OpenFlow enables the programmer to deal



In Port	VLAN ID	Ethernet			IP			TCP	
		Src Addr	Dest Addr	Type	Src Addr	Dest Addr	Proto	Src	Dest

Figure 2.2: The header fields matched in an OpenFlow switch [13]

with high level concepts of protocol implementation instead of having him try to manage tedious and error prone low-level hardware implementation when not required. OpenFlow also comes with elaborate testing frameworks which can be exploited for debugging the protocol and monitoring the network. OpenFlow protocol is widely supported and hence covers a broader range of deployable hardware. This means that it has the upper hand if compared to other platform specific technologies such as Verilog. OpenFlow is used extensively by the network research community. It is easy to collaborate with them as most researchers are comfortable dealing with the languages used for OpenFlow development (C++ and Python). Moreover, OpenFlow abstracts away unnecessary details and helps maintain the focus on the important features of the design.

A drop in performance is expected when using OpenFlow because of the additional overhead of processing the packet in software. However, this drop exists only for the first packet of the flow, which needs to be forwarded to the controller in order to determine the associated action for the remaining flow packets. All the future packets from the same flow would be handled directly by the switchs hardware lookup table. This hit would affect the performance of our system when it comes to measuring the query latency, the time taken before a querying host gets a reply. However, this project predicts that the added latency will be insignificant in large networks because it would be overshadowed by the gains achieved by cutting down broadcast traffic in the network.

Major vendors such as Hewlett-Packard and NEC, are part of the OpenFlow project and produce switches and routers which are capable of running OpenFlow protocol. Linux machines with multiple interfaces can be configured to run as soft switches which support OpenFlow using applications such as Open vSwitch [15]. NOX is a programmable OpenFlow controller which is used to control the behaviour of OpenFlow switches and routers by adding flow entries in their hardware tables for new flows. It is written in C++ and Python and can be extended using either of the languages. Though there are alternative controllers available, such as Maestro [16], NOX is the most stable and commonly used OpenFlow controller with good support available. For these reasons, NOX was used as the software controller in the project.

2.2 Modifications to Project Proposal

I changed the ELK design as described in the project proposal, to remove unnecessary complexity which adds latency, and to discard non-implementable design features.

2.2.1 Change in design

The initial ELK design involved minimalistic use of Jpcap, and instead required the OpenFlow switch to connect to the ELK server directly via a TCP connection and forward ARP packets to it. This design was constructed around misguided information that the OpenFlow switch can create TCP connections with any machine. However, the switch can only form a TCP connection with the controller and does not have the capability to do the same with any other machine.

During the development phase, the ELK server was the first to be designed and it was coded whilst keeping TCP connections between the server and the switches in the design. The design was thoroughly tested using mock data before commencing with the development of the controller and the switches. It was during the controller design stage when I realised that the implementation was not feasible due to the flaw discussed above, and I therefore restructured my server design to incorporate Jpcap, and remove the dependency between the switches and the server. The new design also does not involve sending the ARP reply to the host via the triggering host as initially proposed, but instead directly by the ELK server as a unicast message, as the switches automatically find the shortest path to the host. In my code submission, I have included my initial server code but henceforth I will only discuss the final implementation.

2.3 Learning

Prior to the development stage, it was necessary for me to understand the new systems that I would be dealing with during the course of the project. It mainly entailed the setting up of a suitable development environment to produce and test the required code.

2.3.1 Compiling OpenFlow and NOX

OpenFlow and NOX were compiled on the development and the test machines. Compiling OpenFlow on both types of machines was relatively straightforward due to its extensive support. However, NOX was not fully compatible with operating systems on either of the machines due to dependency issues. It was therefore necessary to tweak the compilation process and create patches when needed, to ensure that NOX compiled on Ubuntu 10.10 and CentOS 5.6. During this process, I also learnt how to work with the repositories using *git*, the distributed version control system.

There is a lack of detailed C++ API documentation for NOX [17] as it is still under active development. I became comfortable with the source of NOX along with the Boost C++ libraries which were used in the source, through examples and experimentation.

2.4 Strategies for success

I laid down development plans to ensure the success of the project.

2.4.1 Test-Driven Development

I worked on my implementation strategies based around a test-driven development plan which ensured that the code was behaving correctly. In order to define correctness, detailed specifications were laid down which dealt with class interactions and method definitions. This behaviour was then moulded into unit tests which were used to test the code throughout the development phase. The design ensured that the Java classes were decoupled, so it was easier to change the implementation and test it. The code was thoroughly commented on and documented so as to aid debugging. The testing phase ran in parallel to the development phase so that the possible sources of the emerging bugs were kept to a minimum to facilitate easier debugging.

2.4.2 Source Control and Backup

All of my code and documentation were version-controlled using SVN on Assembla, an online source control service. I also maintained a version-controlled backup using a professional automatic offsite cloud-based backup service called Spideroak. These services enabled me to easily revert back any unwanted changes during the development phase and test different implementation strategies.

Chapter 3

Implementation

3.1 ELK Specification

ELK was described briefly by Scott *et al.*[4] as an idea which would improve the efficiency of ARP. However, no further specifications were drawn for it. Hence, in order to implement this concept, I needed to decide on the behaviour of the system and how it would interface with the network to provide its directory services.

3.1.1 NOX ELK Controller

The NOX ELK controller, along with NOX core and OpenFlow switches, is used for the purpose of converting broadcast ARP requests into unicast messages, which are sent to the ELK Server over a TCP Connection. Essentially, these components combined, would cut down the unnecessary broadcast traffic in the network which would later be processed by the server instead of the hosts.

The OpenFlow switches in the network form a TCP connection with the NOX core, on which they send every packet that they do not have a flow entry for. The controller sends flow entry packets to the switches for every new flow encountered, indicating that they should behave the way an ordinary switch is expected to behave in all cases, except when the packet is an ARP request from a host. In this special case, the switches should always forward these ARP requests to the controller, which then pushes them down a TCP pipe to the ELK server that was created during the start-up. The only type of ARP message that is allowed to broadcast is a request sent by the ELK server when it does not have an answer to a request sent to it by a host. ARP reply messages sent by hosts and the server are allowed to pass as per usual, as they are unicast messages which are heading to a particular destination in response to a query.

It is crucial to note that NOX controller will forward a copy of all the request packets sent by all the hosts throughout the network to the ELK Server, assuming that there is at least one OpenFlow switch. This is true, because during the broadcast phase when the packet is flooded on all ports, it will try to reach every host in the network. In order to do so, the packet will have to go through every switch in its path, and hence it is guaranteed to hit at least one OpenFlow switch, which will then not further broadcast the packet itself, and forward it to the controller.

3.1.2 ELK Server

When an ARP packet reaches the ELK server, it undergoes processing and in certain cases, produces a resultant packet which is sent out in the network. Below I have summarised the behavioural specifications for my implementation of the server. It is important to note that this implementation assumes that all the hosts in the network are well behaved, and hence the server does not incorporate any security measures. For instance, ELK assumes that the ARP packets are sent by the host as specified in the packet, and not by any malicious host that is faking its identity in an attempt to poison the network.

ARP Request

The server receives ARP requests from two sources — from the NOX controller over a TCP connection, and from the raw interface that the server is listening to in order to capture ARP packets sent to it directly. In either case, ELK extracts the senders information and updates its ARP table as it is free information and can be used later to reply to other requests.

Any ARP requests received from the raw interface are ignored after updating the table. The reason for this, as explained earlier (see Section 3.1.1), is that the NOX controller would send the complete set of ARP requests in the network to the server. Ignoring these packets thereby reduces the extra overhead of processing them, and also prevents the server from spamming the network with unnecessary ARP messages.

The server will service all the requests sent to it by the controller which were captured by the OpenFlow switches. There are three possible outcomes of the processing stage, depending on the input request. These are summarised below.

The first scenario is that the packet is a Gratuitous ARP request [18], which means it is a broadcast packet with the same source and destination IP address. The purpose of such a broadcast request is to help detect IP conflicts when a new host joins the network. Ordinarily, the sending host would not expect a reply to this. However, a reply to this query would imply that there is another host in the network with the same IP address, and hence the sender would need to change its IP address. Gratuitous ARP request is also used to preload ARP tables of other network hosts when a new host goes online, or to update the tables when the host migrates to another MAC address due to reasons such as change of network interface. On detecting a gratuitous request, the server would simply broadcast it back into the network, as it is crucial for all the network hosts to receive it for the reasons stated above.

The second case is when the server receives an ARP request, for which it has the required IP to MAC address mapping. Here, the server simply creates an ARP reply packet with all the required fields sends it to the querying host as a unicast message through the network.

In the last case, the server does not have the necessary mapping for the request, and so it has to buffer the request until it finds a suitable reply for it, or until the request times out. In order to find the correct mapping, it rewrites the ARP request packet by naming itself as the source in the query, and then sends this broadcast message into the network.

ARP requests are sent by the server out of the raw interface under four circumstances, two of which, as stated above, are to find a suitable mapping for an unsatisfied request and to broadcast a gratuitous request. In the third case, before an unsatisfied buffered request times out, the server reattempts to find the mapping by sending three copies of the same broadcast message. This is done to overcome the problem of packet loss. Finally, in the fourth case, the server sends a unicast request message to a host when its mapping in the server is about to expire. The purpose of this message is to verify the entry before evicting it, so that it can, if possible, be retained for future use.

ARP Reply

The only ARP reply ingress for the ELK server is via the raw network interface that ELK is listening to. As discussed earlier, the NOX controller does not forward any ARP replies to the server (see Section 3.1.1). The replies received by the server would be responses to the ARP requests that the server had sent from the same interface.

The server extracts the senders MAC and IP address mapping from the packet and stores it with the intention of using this information to answer ARP requests. It then checks for any buffered requests that require this newly learnt mapping as a reply. If there are any such requests, the server generates unicast ARP reply packets for all these querying hosts and sends them into the network.

The only other scenario, in which the server sends replies to the hosts, is when it receives a request for which it already has a valid mapping required for answering the query. In this case, the request is not buffered and a reply is sent out immediately.

3.2 Approach to Implementation

This section outlines the methodology employed for implementing the system.

3.2.1 Modularity

In the implementation, a modular approach was employed by using the dependency injection design pattern. This involved subdividing the system into smaller independent modules which were decoupled from each other which made debugging and testing easier.

Dependency Injection

The system was designed using dependency injection design pattern. In this, the classes or the *consumers* are decoupled from each other by not allowing any object construction in class definition and the required objects are passed to the class constructors through parameters. All the dependent objects are constructed, initialised and passed to the required classes by one class which acts as the *injector*. Since all the dependencies are now *injected* or passed to the classes, they are insensitive to the underlying implementation of the dependencies. This makes it easy to test each module of the code individually and to reuse code using different implementations.

The role of the injector is played by `ServerMain` in the server code which constructs and initialises the required data-structures, processes, and threads, and passes them their dependencies. This design enables the server to easily allocate multiple threads for a demanding task by initialising and starting new threads for the task in this class. For instance, it could accommodate multiple `ARPProcessor` threads to service the load quickly, or have multiple `ARPListener` threads to handle high volume of incoming ARP traffic. Another possibility is to have multiple `ClientHandler` threads to connect to the various NOX controllers across a large network. The other advantage is that this system supports incremental design approach. This means that new functionality could be added to the system by simply creating new components which then can be constructed by the ServerMain.

Separation of Concerns

The ELK server and the NOX ELK controller were kept as two entirely independent modules which interfaced via a TCP pipe that carried only one-way ARP traffic. The server constituted the main processing unit of the system and the controller formed the re-directional unit. Furthermore, the OpenFlow protocol deployment in the switches was completely disjoint from the controller and the switch communicated with the controller using OpenFlow messages. This modularity allowed the switches to use a standard implementation of the OpenFlow switches, which in this case was Open vSwitch. These switches only required to be configured to have a database to save flow entries, and to have the location of the controller in order to establish a connection with it. No other implementation was necessary.

All the functions of the server, such as receiving ARP packets and sending them, were modelled to run in independent threads so as to exploit parallelism without any conflicts arising from dependencies. The only common link stringing the threads together were the thread-safe data structures, as detailed in Section 3.3.1, which were used to communicate between different threads. In the server, the table holding the information of the unsatisfied requests (from here on referred as the **RequestTable**) and table holding the IP to MAC address mapping (referred as the **ARPTable**) were both implemented in their own independent classes. These tables along with all their operations such as caching and entry renewing could thus be developed and tested independent of the core modules.

System modularity allowed the development of various components to run in parallel without interfering or hindering the process. This feature allowed the modules to be developed and tested individually without the need of an actual working network at all times.

3.2.2 Platforms of Development

Due to complexity of OpenFlow and NOX coupled with the lack of detailed API documentation, it was necessary to learn the technologies using examples without being bogged down by irrelevant details. For this purpose, a virtual OpenFlow network was simulated on my development machine using the OpenFlow Virtual Machine Simulation package, OpenFlowVMS. This package created a fully functional OpenFlow network with virtual hosts running on QEMU connected to each other via OpenFlow switch modules. NOX was deployed alongside this package to act as the OpenFlow controller. The resulting network was first used to better understand OpenFlow and NOX, and later employed to observe the behaviour and debug the ELK controller during the initial development phase. This virtual network was later replaced by a physical network to serve as the development and testing environment after I was familiar with the technologies.

3.3 Main areas of code

I will describe how the code for this project was put together in this section. Figure 3.1 shows the class diagram for the server.

3.3.1 Data Structures

I incorporated some useful data structures in my implementation.

Safe Message Queue This is an implementation of an unbounded queue which allows objects to be added or removed in constant time. It is based on a linked list which does not allow random access as the incoming packets should be serviced in order of arrival (FIFO). It is a concurrency-safe

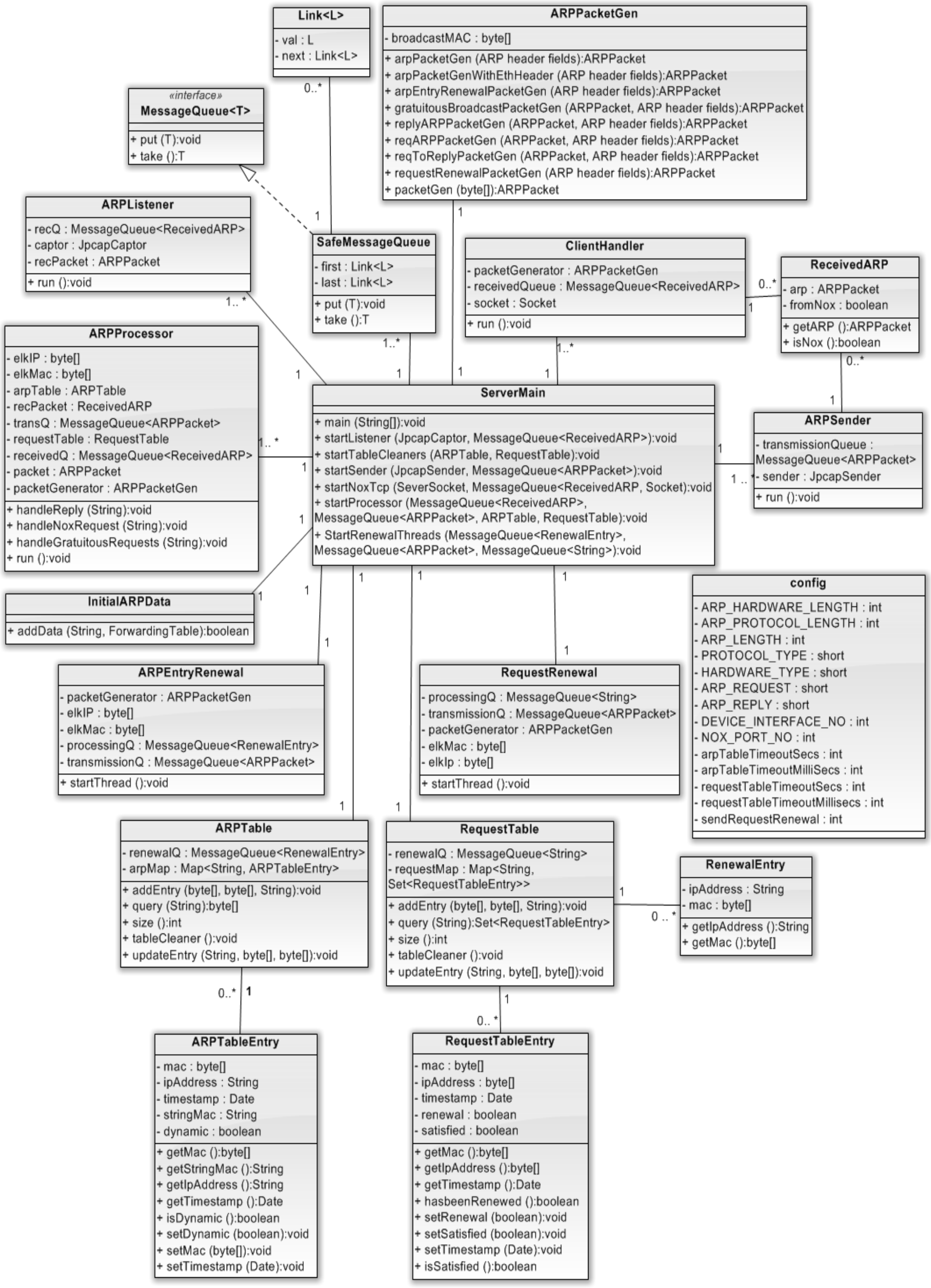

Link<L>
- val : L
- next : Link<L>
ARPPacketGen
- broadcastMAC : byte[]
+ arpPacketGen (ARP header fields):ARPPacket
+ arpPacketGenWithEthHeader (ARP header fields):ARPPacket
+ arpEntryRenewalPacketGen (ARP header fields):ARPPacket
+ gratuitousBroadcastPacketGen (ARPPacket, ARP header fields):ARPPacket
+ replyARPPacketGen (ARPPacket, ARP header fields):ARPPacket
+ reqARPPacketGen (ARPPacket, ARP header fields):ARPPacket
+ reqToReplyPacketGen (ARPPacket, ARP header fields):ARPPacket
+ requestRenewalPacketGen (ARP header fields):ARPPacket
+ packetGen (byte[]):ARPPacket
«interface»
MessageQueue<T>
+ put (T):void
+ take ():T
ARPListener
- recQ : MessageQueue<ReceivedARP>
- captor : JpcapCaptor
- recPacket : ARPPacket
+ run ():void
SafeMessageQueue
- first : Link<L>
- last : Link<L>
+ put (T):void
+ take ():T
ClientHandler
- packetGenerator : ARPPacketGen
- receivedQueue : MessageQueue<ReceivedARP>
- socket : Socket
+ run ():void
ReceivedARP
- arp : ARPPacket
- fromNox : boolean
+ getARP ():ARPPacket
+ isNox ():boolean
ARPProcessor
- elkIP : byte[]
- elkMac : byte[]
- arpTable : ARPTable
- recPacket : ReceivedARP
- transQ : MessageQueue<ARPPacket>
- requestTable : RequestTable
- receivedQ : MessageQueue<ReceivedARP>
- packet : ARPPacket
- packetGenerator : ARPPacketGen
+ handleReply (String):void
+ handleNoxRequest (String):void
+ handleGratuitousRequests (String):void
+ run ():void
ServerMain
+ main (String[]):void
+ startListener (JpcapCaptor, MessageQueue<ReceivedARP>):void
+ startTableCleaners (ARPTable, RequestTable):void
+ startSender (JpcapSender, MessageQueue<ARPPacket>):void
+ startNoxTcp (SeverSocket, MessageQueue<ReceivedARP, Socket):void
+ startProcessor (MessageQueue<ReceivedARP>, MessageQueue<ARPPacket>, ARPTable, RequestTable):void
+ StartRenewalThreads (MessageQueue<RenewalEntry>, MessageQueue<ARPPacket>, MessageQueue<String>):void
ARPSender
- transmissionQueue : MessageQueue<ARPPacket>
- sender : JpcapSender
+ run ():void
InitialARPData
+ addData (String, ForwardingTable):boolean
config
- ARP_HARDWARE_LENGTH : int
- ARP_PROTOCOL_LENGTH : int
- ARP_LENGTH : int
- PROTOCOL_TYPE : short
- HARDWARE_TYPE : short
- ARP_REQUEST : short
- ARP_REPLY : short
- DEVICE_INTERFACE_NO : int
- NOX_PORT_NO : int
- arpTableTimeoutSecs : int
- arpTableTimeoutMilliSecs : int
- requestTableTimeoutSecs : int
- requestTableTimeoutMillisecs : int
- sendRequestRenewal : int
ARPEntryRenewal
- packetGenerator : ARPPacketGen
- elkIP : byte[]
- elkMac : byte[]
- processingQ : MessageQueue<RenewalEntry>
- transmissionQ : MessageQueue<ARPPacket>
+ startThread ():void
RequestRenewal
- processingQ : MessageQueue<String>
- transmissionQ : MessageQueue<ARPPacket>
- packetGenerator : ARPPacketGen
- elkMac : byte[]
- elkIp : byte[]
+ startThread ():void
ARPTable
- renewalQ : MessageQueue<RenewalEntry>
- arpMap : Map<String, ARPTableEntry>
+ addEntry (byte[], byte[], String):void
+ query (String):byte[]
+ size ():int
+ tableCleaner ():void
+ updateEntry (String, byte[], byte[]):void
RequestTable
- renewalQ : MessageQueue<String>
- requestMap : Map<String, Set<RequestTableEntry>>
+ addEntry (byte[], byte[], String):void
+ query (String):Set<RequestTableEntry>
+ size ():int
+ tableCleaner ():void
+ updateEntry (String, byte[], byte[]):void
RenewalEntry
- ipAddress : String
- mac : byte[]
+ getIpAddress ():String
+ getMac ():byte[]
ARPTableEntry
- mac : byte[]
- ipAddress : String
- timestamp : Date
- stringMac : String
- dynamic : boolean
+ getMac ():byte[]
+ getStringMac ():String
+ getIpAddress ():String
+ getTimestamp ():Date
+ isDynamic ():boolean
+ setDynamic (boolean):void
+ setMac (byte[]):void
+ setTimestamp (Date):void
RequestTableEntry
- mac : byte[]
- ipAddress : byte[]
- timestamp : Date
- renewal : boolean
- satisfied : boolean
+ getMac ():byte[]
+ getIpAddress ():byte[]
+ getTimestamp ():Date
+ hasbeenRenewed ():boolean
+ setRenewal (boolean):void
+ setSatisfied (boolean):void
+ setTimestamp (Date):void
+ isSatisfied ():boolean

Figure 3.1: Class diagram outlining pseudo code of the ELK server. Some features omitted for clarity and brevity.

data structure which prevents illegal structural modification as each thread wanting to modify the queue requires mutually exclusive access to it. The wait-notify paradigm is incorporated in this queue structure which improves its efficiency as packets are serviced as quickly as possible and it does not actively wait for packets to arrive thereby saving processing time.

The main purpose of this queue in the multi-threaded ELK system design is to transmit data safely across various threads without any loss. There are three main queues employed in the code:

- `ReceivedQueue`: Contains all the ARP packets received by the server from all sources.
- `TransmissionQueue`: Placeholder for all packets that need to be transmitted via the network interface.
- `RenewalQueue`: Holds the information about the ARP proxy and the request entries that require renewal.

The ARP packets in the system are treated as being objects of class ARP-Packet from the Jpcap library. This ensures that the network and host byte order are conserved.

HashMap This data structure was used to store the key-value pairs as it provides constant-time performance for the basic operations, get and put [19], which is critical to reduce the query latency while processing packets. The structure also provides random access to the data which is required for the lookups. The map was externally synchronized using the Java collections framework in order to allow safe concurrent access by multiple threads. The `ARPTable` and the `RequestTable` are both instances of synchronized hash maps.

ARPTable The map uses IP address as the key and holds the IP to MAC address mappings in a `ARPTableEntry` object as the value. Each entry also has a timestamp indicating when the entry was last updated. An entry is deemed expired if it is has not last updated for more than the permitted cache time. Expired entries are never returned as query results.

The table uses all the possible information from incoming packets to keep itself up-to-date. When updating an existing entry for an IP address, the mapped MAC address is updated along with the last updated timestamp. This update ensures that the mapping is correct, as it might have changed for various reasons such as expiration of IP address, or change of location by the host.

Manually added entries (see `InitialARPData`, Section 3.3.3) in this table can either be dynamic or static. Static entries never expire and hence are never removed from the table. All mappings learnt from the network are defaulted to dynamic. This feature helps reduce broadcast traffic for machines with fixed IP addresses.

When an entry is about to expire, the table issues a unicast ARP request, which is directed to the mapping owner to verify if the mapping is still correct. The entry is marked stale until the table hears a reply from the host or the stale entry times out. Any requests for the stale mapping would be added to the `RequestTable`. This is done to avoid additional broadcast requests for the entry which is being verified. If there is a reply, the stale entry is fully restored back into the table and all outstanding requests are satisfied. If the stale entry times out, then it is very likely that that the mapping no longer exists and the entry should be evicted from the table. The eviction is done by a table-cleaning thread which checks the timestamp for dynamic entries only. This thread runs once every half cache time so as to ensure that the entries are removed from the table within the next half cache time after expiration.

RequestTable This table is very similar to the `ARPTable` in terms of caching entries and table cleaning. This map holds the IP address for which the MAC address was requested as the key and the set of the `RequestTableEntry` objects as the value. A `RequestTableEntry` object encapsulates the required information of the requesting machine in order to reply to it when the server finds the appropriate mapping. A set is maintained in each entry to accommodate for multiple hosts requesting for the same mapping. When the required mapping is found, the reply is sent to all the members of the set.

During the table cleaning phase, any entry which has not yet expired but has already been satisfied with an appropriate response, is expunged from the table. Before evicting an expired entry, the cleaner sends out multiple ARP requests for the required mapping (to overcome packet loss). These requests are only sent out once for the whole set and none of the current members of the set are allowed to send ARP requests again. This is because a lack of response after multiple requests implies that it is highly unlikely that the required IP address exists anymore. Hence, sending ARP requests only once per set helps reduce unnecessary broadcast traffic. Due to this process, an expired request does not get evicted for an extra half cache time until the cleaning thread iterates over the table again.

3.3.2 Life of an ARP packet

In this section, I will discuss how the system interacts internally with the ARP packets. Figure 3.2 shows a flow diagram for it.

Receiving

NOX Controller Switches forward the first packet of the flows with no associated actions in switches flow table to the controller. The NOX controller learns the MAC address to source port mapping for every packet sent to it. This is done to learn the location of the various hosts in the network in order to make packet forwarding decisions (which port should a packet be sent out from so that it reaches its destination via the shortest path available).

If the controller receives an ARP packet from a switch, it deduces if it is an ARP request from a host by examining the Ethernet header. ARP packets with Ethernet destination address as broadcast (replies never have broadcast destination) and Ethernet source address not equal to ELKs hardware address (ELK requests are required to be broadcasted as normal) are classified as host ARP requests and are sent to ELK server over TCP for processing. This logic also prevents looping of packets because if the controller starts sending ELK requests back to the server, it will cause these packets to oscillate in the network. For all other packets, the controller sends an OpenFlow packet that contains the fields (see Figure 2.2) used to define the new flow, along with the required associated actions, to the switch. This logic implies that the controller adds flow entries for all the ARP reply packets and the ELK ARP request packets, and these packets are sent directly to their destination and not to ELK.

ClientHandler All the packets sent by the controller are received at the server by a thread which runs continuously. It removes incoming packets from TCP socket and parses them using the `ARPPacketGenerator` (discussed later) into `ARPPacket` instances by removing all the other additional headers. The thread then adds these parsed objects to the `ReceivedQueue`.

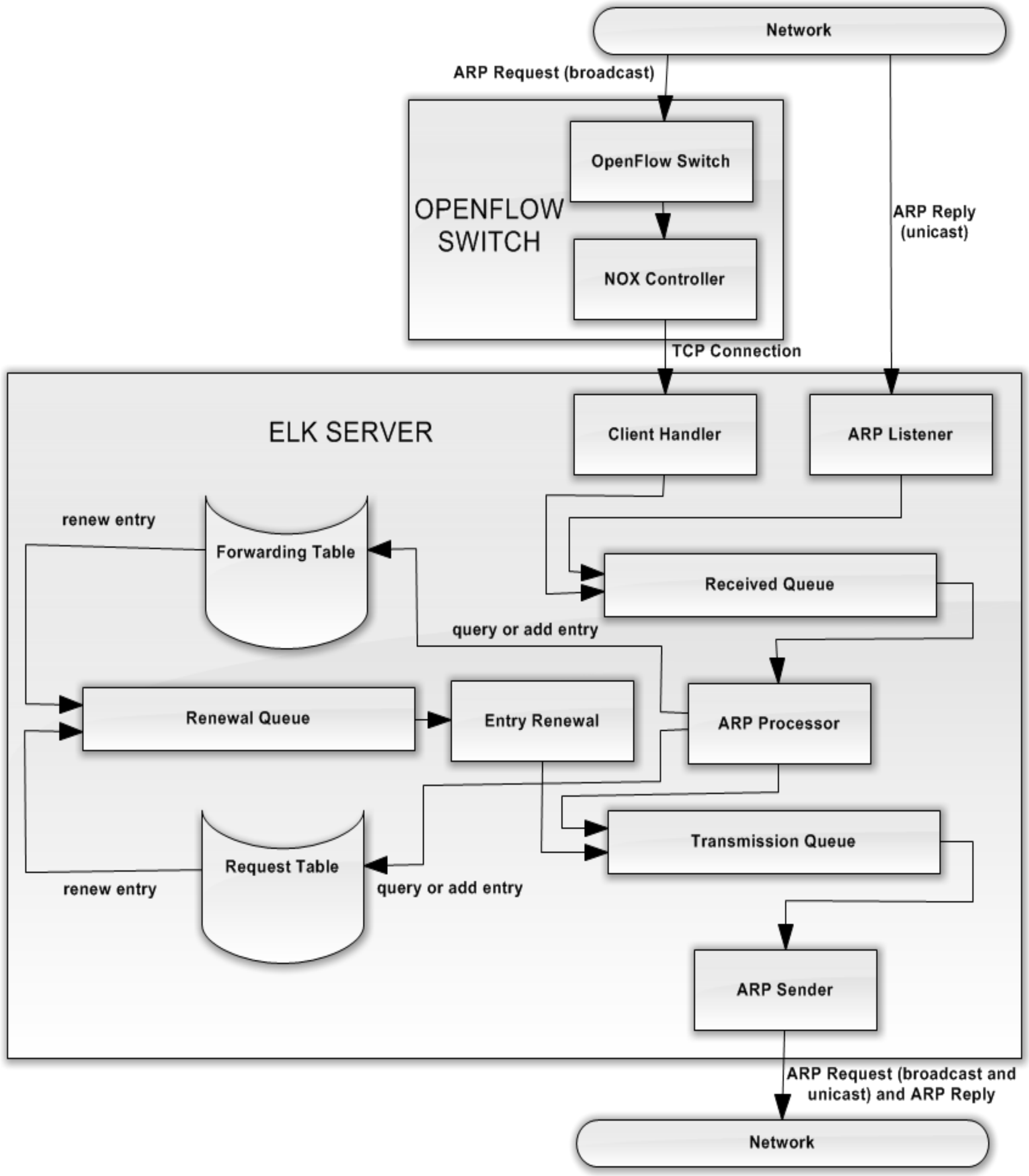


Figure 3.2: Flow chart diagram showing the processing path taken by the incoming packets.

ARPListener This thread actively captures all the packets from the raw interface in promiscuous mode and filters out the ARP packets. Similar to the `ClientHandler`, the thread parses the packet correctly and augments it to the `ReceivedQueue`.

Processing

ARPPacketGenerator The class is solely responsible for parsing the incoming packets and generating `ARPPacket` instances based on specified parameters for transmission.

ARPProcessor The packet processing logic of the server is based entirely on the specification mentioned in Section 3.1.2 and is encapsulated in this class, which runs on an independent thread. The `ReceivedQueue` serves as the classs ingress for packets to be processed and the `TransmissionQueue` forms the queue for the packet to be released in the network.

The processor uses an idea similar to that used in ARP Poisoning [20], an ARP security vulnerability, to answer queries. In ARP Poisoning, the attacker sends fake ARP replies by pretending to be the owner of the destination IP address of the request. The fake reply can provide either its own MAC address or a non-existent MAC address depending on the intentions of the attacker. In ELK, when the processor sends an ARP reply, it sends the correct data (unlike an attacker) to the hosts but it fakes its identity by pretending to be the actual owner of the IP address. This is done because the querying host will extract the senders information from the reply and use it as the answer for its request. ELK server cannot use its own identity as it will then only ever be able to advertise its own IP to MAC address mapping. However, the source address field in the Ethernet header is filled in correctly using the servers address. This is because the intermediate switches use the Ethernet header to learn the location of the hosts, and using a fake address will misdirect the traffic in the network. This security flaw is exploited based on the assumption as stated earlier that all the hosts are non-malicious.

Sending

ARPSender This thread actively transmits packets from the `Transmission-Queue` (if any) through the network interface. These packets constitute unicast and broadcast ARP requests, ARP replies, and gratuitous ARP requests.

3.3.3 Maintenance of the Tables

These classes attempt to automatically renew the expiring entries in the tables and also provide the administrator the ability to manually add entries in the ARP table.

EntryRenewal This class runs on a separate thread and takes its input from the `RenewalQueue` which contains the information of entries to be renewed from the `ARPTable` and the `RequestTable`. It generates the required packets as per the information provided and adds them in the `TransmissionQueue`. The number of renewal packets sent per entry (to prevent packet loss — see Section 3.1.2) is defined by the value in the `Config` file (see Section 3.3.4).

InitialARPData The ELK system is a learning algorithm which means it builds up its database by gathering information about new mappings while processing the packets. This provides the server with a slow start in terms of cutting down broadcast traffic. This class allows the administrator to insert entries in the `ARPTable` prior to the start up of the server which will help reduce the initial broadcast traffic and boost the systems performance.

3.3.4 Glue

I wrote some classes which constructed the required objects, started the threads, resolved unnecessary dependencies and tied all the classes together.

ServerMain This class is used to encapsulate the state of the entire server. Essentially, it forms the injector of the dependency injection design pattern. This class initialises all the queues used for data transmission and all the threads (under the current implementation there are eight threads) including the table cleaning and entry renewal threads. It also sets up the TCP connection with the controller and creates an instance of the interface to communicate with the network.

Config This holds all the values of the globally-used parameters in the code. These values are kept in one file so that it is easy to modify the behaviour of the system as required. These constants include the port at which NOX controller connects, and the cache time for the `ARPTable` and the `RequestTable`.

3.4 Testing

Testing was a critical phase of the development process so as to ensure that the code worked reliably and showed expected behaviour. Testing frameworks were deployed to perform unit testing, black box testing, and integration testing.

3.4.1 Testing Framework

It was important to choose a testing framework which fits in the requirements of the development environment. Out of the various options available to test the server written in Java, the decision went in favour of the JUnit testing framework[1] in conjunction with the debugging environment provided by the Eclipse, an Integrated Development Environment for Java. This regression testing framework helps create a one-to-one relationship between the units of code being developed and the tests for them. It is easy to write tests in this framework and they can be automated in Eclipse. Any modification in the code which deviates from the specifications and breaks these tests would be flagged to the programmer. Eclipse plays an excellent supporting role as its debugger allows full control over the execution of the code which allows the programmer to trace the data and the control flows. JUnit provides an elaborate API which handles assertions for a wide range of data types well. The test results, including full details of failed tests, are displayed using a very well laid out graphical user interface which makes debugging easier.

3.4.2 Testing Strategy

Unit tests were written for the the class methods and the logic detailed in the specifications for the server (see Table 3.1 for example cases). These tests were used to prove the correctness of the code behaviour. Integration tests were then devised around classes such as the ServerMain class which invoked methods from other dependent classes. The NOX ELK controller was heavily dependent on the NOX core which made it hard to test. Since there was a distinct boundary

[1]http://www.junit.org/

TestRequestEntryEviction	Checks if the expired entries are evicted from the request table after their failed attempts to renew.
TestBadPacketGen	Tests that the ARP generator flags attempts to generate bad ARP packets.
TestNonRequestOrReplyPacket	Tests if ELK behaves predictably when it encounters ARP packets such as RARP reply, which are neither ARP reply nor ARP request.
TestUpdateEntry	Checks that the entry in the ARP table has been updated correctly and no duplicate entries for it are created.

Table 3.1: Example test cases

between the controller and the server, it was possible to leave the controller out of the unit testing and perform only some simple testing and sanity checks. It was then combined with the server in the systems testing so that the controllers behaviour could be scrutinized in more detail.

To test the complete ELK system, I performed black box testing it by creating a simple test network topology consisting of ELK and triggering ARP commands using Jpcap along with real ICMP traffic from the ping command. The behaviour of the traffic, such as correct redirection at switch level and accurate server response, was examined using the Wireshark network protocol analyser. Wireshark allowed me to trace the exact path of each packet including OpenFlow traffic, in the network by monitoring the flow of packets at each network interface. I used the *arp* command to check the ARP tables of the hosts and the *ovs-dpctl*, an Open vSwitch utility, to study the flow entries in the switches.

Chapter 4

Evaluation

4.1 Experiments and Results

The aim of the evaluation was to examine the behaviour of ELK and compare its performance against that of conventional ARP. The experiments conducted measured the effectiveness of ELK in decreasing broadcast messages in the network and hence reducing the unnecessary broadcast packet processing at the hosts. Although, ARP has known security issues such as ARP spoofing [20], the aim of this project was not to fix these and so ELK's security was not evaluated.

The machines used for the experiments were running CentOS 5.6 and had six Ethernet interfaces each. Four of the interfaces on each machine were used for creating the necessary topology. One of the remaining interfaces on each machine was connected to a switch which formed a separate network and was used for out-of-band control by the NOX controller. The four interfaces of the machines which acted as switches were not allocated an IP address. They simply make forwarding decisions based on the MAC address of the incoming Ethernet frame. Open vSwitch was installed on these machines which made them behave as OpenFlow switches.

Due to a limited number of available test machines, only two machines formed the hosts in the topology. In order to increase the number of hosts and thus the ARP traffic, all of the interfaces of the host machines were allocated a different IP address (within the same subnet) and were connected to the switches. However, due to the implementation of the Linux network stack in the hosts kernel, all of the interfaces in the machine replied to an ARP request that was only directed to one of these interfaces, resulting in duplicate ARP replies. This problem was resolved by using only one physical interface per host but creating multiple (eight in this case) virtual interfaces on that physical interface. All of these virtual interfaces were allocated a different IP address from the same subnet and they had the same MAC address. Since now there was only one physical interface, Linux network stack generated only one ARP reply per query which was sent out from the virtual interface with the correct IP address.

The experiments were aimed at evaluating ARP traffic with ELK. The ARP caches of all the hosts were thus cleared to remove any stale entries which could affect the correctness of the tests.

4.1.1 Reduction in ARP Traffic

The primary purpose of ELK is to cut down unwanted ARP traffic and reduce the amount of broadcast traffic in the network. This eventually helps to reduce unnecessary ARP processing at the hosts. The amount of benefit that the network can derive out of ELK is proportional to the size of the network.

Let H be the number of hosts and S be the number of switches in the network. Thus, there will be $O(S+H)$ connections in common network topologies, such as the tree and the linear topology. The total amount of broadcast traffic generated throughout the network by ARP and ELK (after the learning phase) when every host is trying to resolve the address for every other host in the network, would be $O(H^2(S+H))$ and $O(H^2)$, respectively. This is because in ELK, broadcasts are converted into unicast messages which are redirected to the controller by the first OpenFlow switch that the broadcast encounters. In the network using ARP, each host will have to process $O(H^2)$ broadcast packets. However, hosts in the network with a fully learnt ELK server will process zero broadcast packets because they are being handled by the server.

The test setup for this evaluation used the topology as shown in Figure 4.1. Each host had eight virtual interfaces. One of the hosts artificially sent eight ARP queries, one for each of the eight virtual interfaces of the second host, every 5 seconds. Wireshark was used to monitor the flow of these ARP packets in the network. This test was run four times under different conditions in the network — with no ELK server, with ELK server, with ELK server with some static entries inserted in the table at setup, and with the server with the ARP table having static entries for all the mappings in the network.

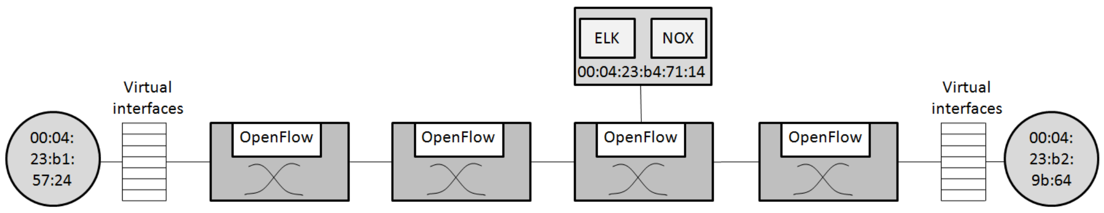


Figure 4.1: Evaluation topology used for the experiments. A rectangle denotes an OpenFlow switch and a circle represents an end host.

The conventional ARP with no enhancements formed one of the extremes (worst case) in the test cases. As expected, in this scenario, all eight broadcast queries traversed the entire network and were processed by all the hosts in the network. The target host answered all of the request packets and sent back eight replies. In the best-case scenario, the ELK server had complete knowledge of the IP address to MAC address mapping throughout the network. ARP requests were redirected by the OpenFlow switch to the server which responded to every request. No ARP messages were broadcast in the network and hence the hosts were completely oblivious to the ARP traffic in the network. A network with an ELK server that had no, or some, static entries formed the common-case scenario. The server, in this case, learnt the mappings for all missing entries using broadcast, and renewed the expiring entries by using unicast request messages.

The results from these tests are summarised in Figure 4.2 and Figure 4.3. On the primary axis, the first graph depicts the number of ARP packets sent or received by the target host on log scale over time. The packets include the ARP broadcast and unicast request messages sent by ELK, and the ARP replies being sent by the target host. The secondary axis shows the cumulative total of the number of ARP packets exchanged by the host. This graph shows the significant difference between the number of ARP messages exchanged under different ARP mechanisms. However, in the case of ELK with an incomplete table (no or some static entries), the behaviour of the ELK system is not ideal. The spikes depicting the ARP traffic are not as well defined as in the case of conventional ARP. They are split into multiple shorter and wider peaks. The ARP packets sent by the server exhibit a significant delay before they arrive at the target host. This is because of the added latency due to additional processing of the packet in the server. This latency in discussed in more detail in Section 4.1.2. Figure 4.3 shows the number of broadcast ARP messages received by the host. It can be seen that beyond the initial learning set of broadcast messages, the host does not receive any broadcast packets, unless of course, the entry expires and fails to renew,

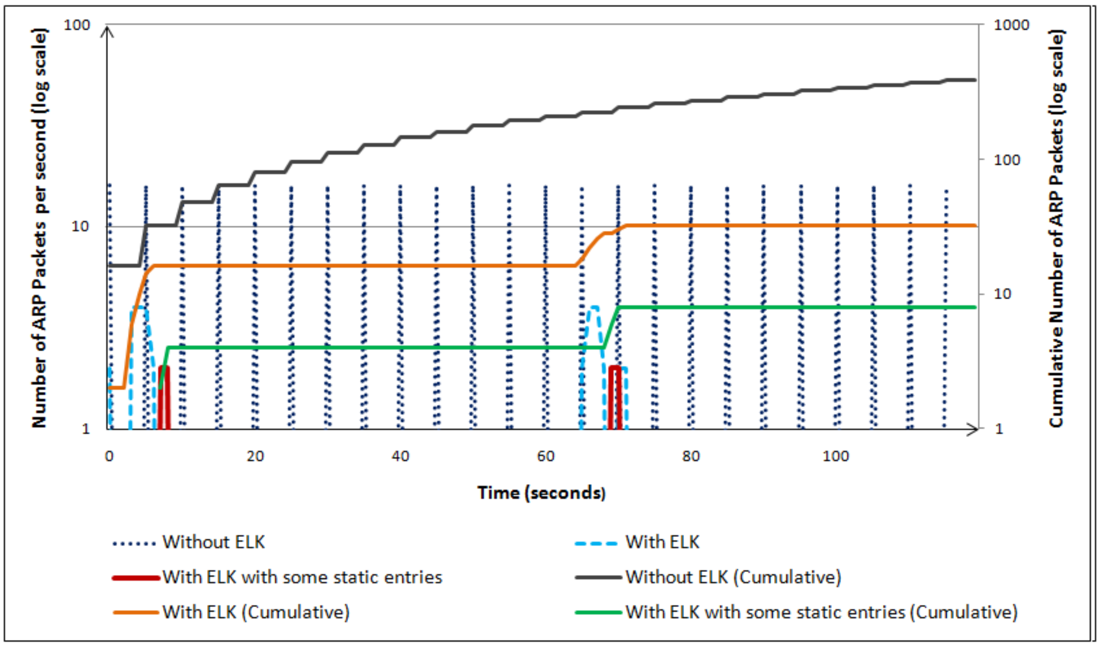


Figure 4.2: Graph showing the rate and the cumulative total of ARP messages exchanged by a host using different ARP mechanisms

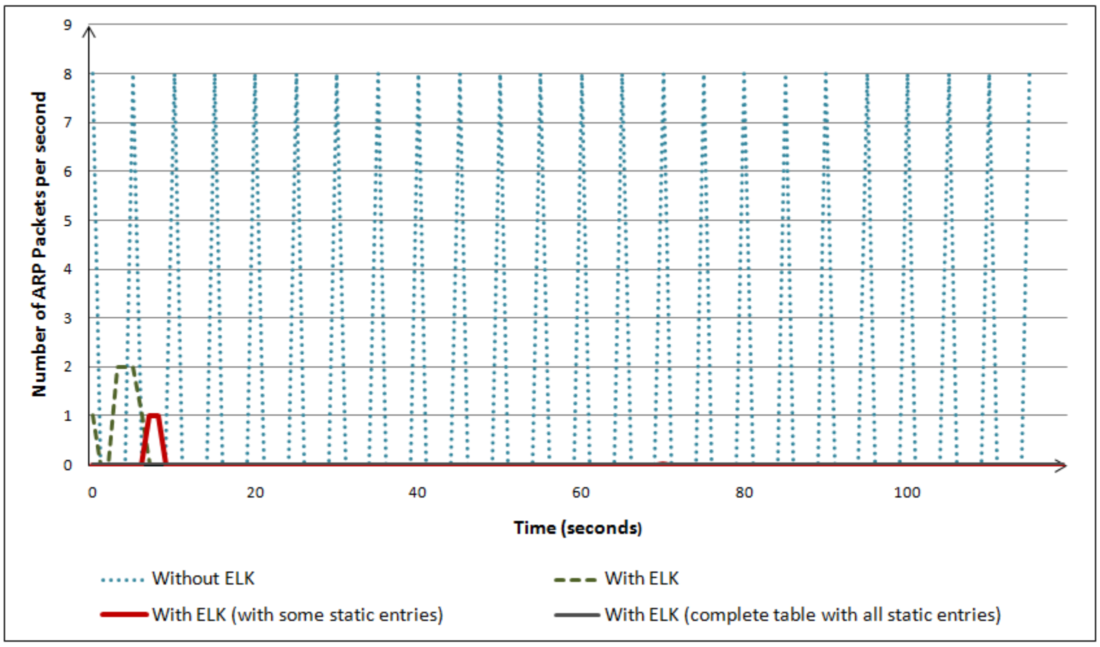


Figure 4.3: Graph showing the rate of broadcast ARP messages received by a host using different ARP mechanisms

resulting in its eviction from the server.

Screenshots of these Wireshark captures are included in Appendix C for illustration.

Although there was a clear reduction in the amount of ARP traffic, this does not depict a complete picture of the network. ELK relies heavily on OpenFlow switches which generate their own traffic to communicate with the controller and vice versa. The OpenFlow traffic consists of messages such as Flow-Mod (instructs switch to add a particular flow to the flow table), Packet-In (sent to controller when a packet for unknown flow is detected) and Flow-Expired (flow timeout due to inactivity). These messages, however, are usually transmitted when a new flow is detected or when a flow expires and therefore, in the average case, do not count towards the network traffic. The traffic also comprises Echo Request and Reply packets which are sent to keep the connection between the controller and the switch alive. Echo requests containing arbitrary data are sent only once every 15 seconds (the value is stored in the openflow class of NOX in Openflow_connection::probe_interval) which the switch simply needs to echo back. These packets are unicast messages that are processed by the switches and not the hosts, and are therefore do not burden the network. The deployed NOX controller and OpenFlow switches can also be used to optimise filtering, security or other network protocols and this sharing of OpenFlow traffic would make the costs insignificant.

4.1.2 Query Latency

Query latency, in this case, is defined as the time taken by the network to reply to an ARP request. It is measured as the time from the end of packet transmission from the sender to the end of the reply packet reception by the sender. Low query latency is ideal in networks.

When the host sends an ARP request, it expects the reply to be returned as soon as possible. Ideally, adding ELK in the network should not increase this latency, as this will cause a slowdown in the network. To test this hypothesis, a host was made to send a new ARP request (to a different virtual interface) after it received the reply for the previous query using the network topology in Figure 4.1. This is an artificial situation created for the experiment and would not occur in real networks. The reason for not measuring latency using real network data is that the frequency of ARP messages in the small test network would be too low to deduce meaningful results. The number of ARP messages exchanged and the

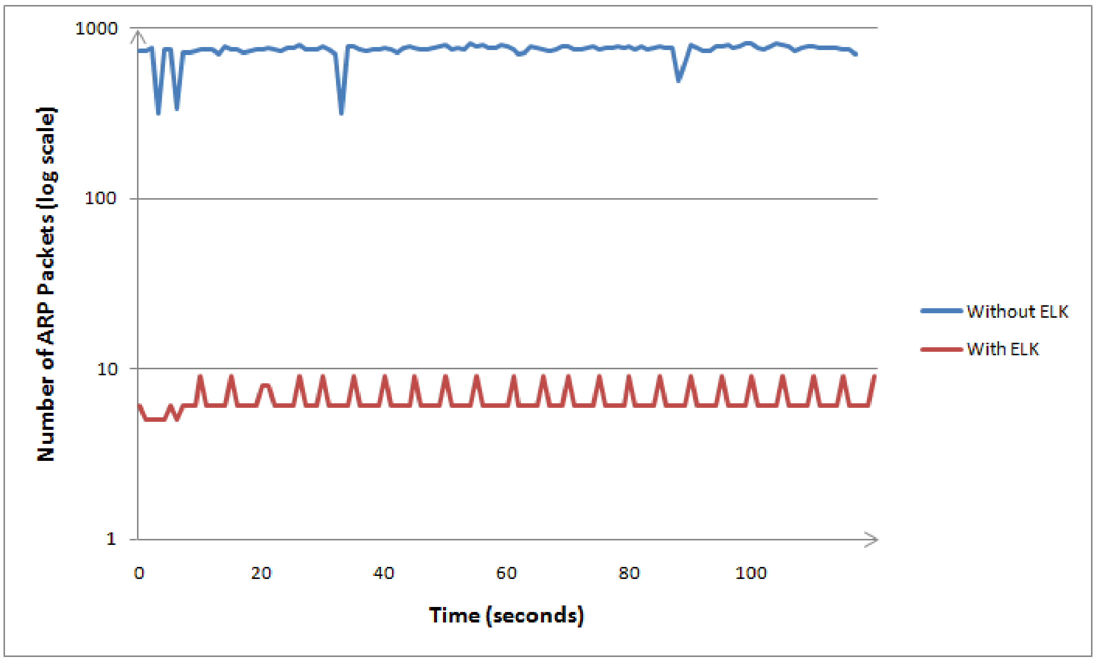


Figure 4.4: Graph depicting the number of ARP packets that can be exchanged between two hosts per second.

timestamps for these messages were recorded and analysed using Wireshark. The host used a Java client to send and receive ARP packets and the timing values observed included the Jpcap and hardware interaction overhead. Hence, all the observed values should be used in relative rather than absolute sense.

The graph in Figure 4.4 shows the number of ARP messages exchanged between two hosts every second on a log scale with only one request in-flight at any point in time. It implies that the query latency in ARP is significantly lower than that of ELK, which enables ARP to resolve about 100 times more queries than ELK each second. For conventional ARP, on average the host observed the reply after 3ms of sending the query. ELK on the other hand took on average of 615 ms during its initial phase, and 307 ms after the learning phase. The initial phase takes almost double the time taken after the learning phase because it has to send out a broadcast ARP request to find the correct mapping and then generate a reply after discovering this mapping.

The reason for latency in ELK is because every ARP request has to pass through two TCP connections (from switch to controller and from controller to server) and has to undergo processing at the server (which includes querying hash-maps, storing values and regenerating packets) before the ELK can send an appropriate reply. There is also a large overhead of using Java Native Interface (JNI) based Jpcap library to interact with the hardware [21]. The ELK server is designed to be a prototype and hence does not account for efficiency in great detail. Another reason for the delay is that the OpenFlow switches were, in reality, soft switches running on Linux machines with the Open vSwitch module; soft switches are not as efficient as normal switches [22].

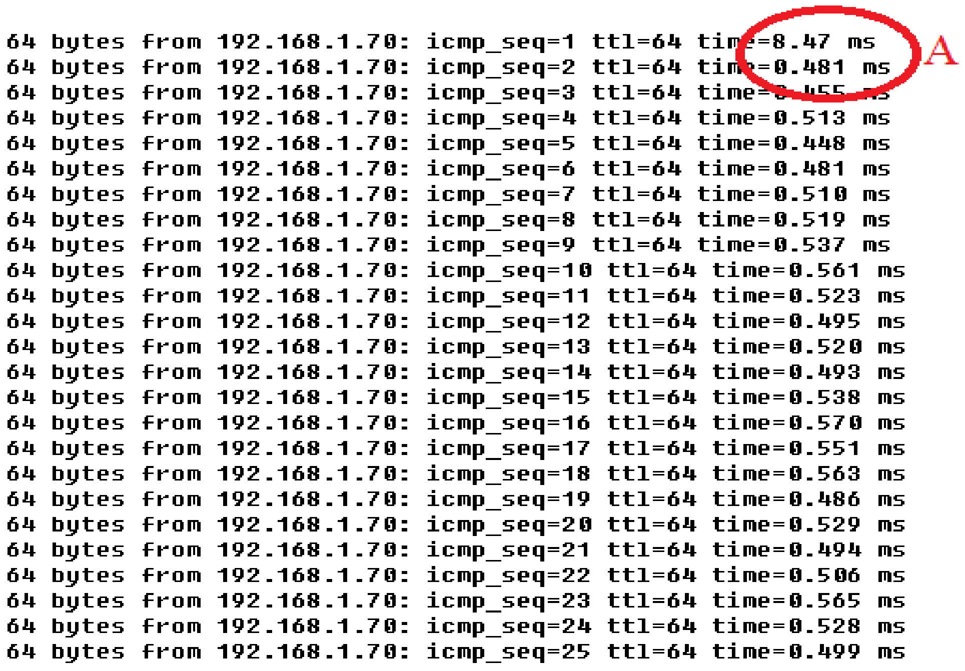
This issue of latency can be circumvented by using a better-optimised implementation of the ELK server, which uses more efficient data structures and utilizes threading to its full potential. Another possibility is to replace the TCP connection between the controller and the server with a UDP connection. Under certain circumstances, such as new connection setup and traffic congestion, UDP is faster than TCP. The system does not have to worry about the packet loss in UDP as the host will simply send a new ARP request when it does not receive a reply.

A realistic comparison of the query latencies between the two systems can only be performed in a larger test network. In such a case, ELK may be able to bypass the route across the network (depending on the position of the server and the querying host) and be able to query the server directly, whereas ARP will have to traverse the whole network to find the mapping. This test, however, could not be performed on a large network due to lack of test machines.

4.1.3 Interference with non-ARP traffic

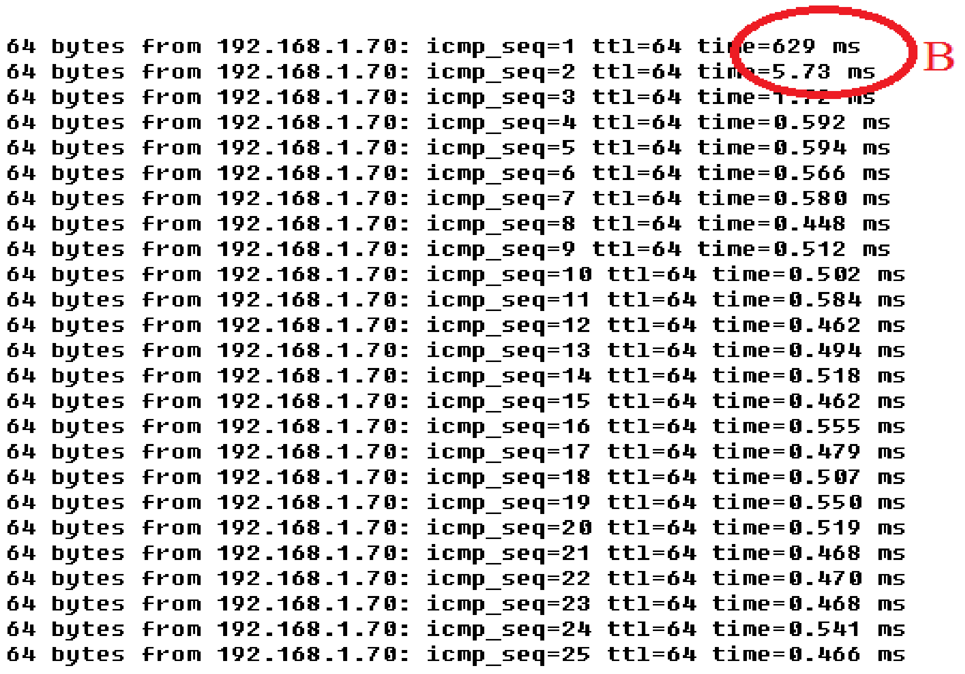
Under normal network conditions, ELK should only aid address resolution aspects of the network, and not interfere with communication between systems. This means that non-ARP traffic should not experience any disruptions due to the additional system in the network.

In this experiment, the network topology described in Figure 4.1 was used. Non-ARP traffic was generated using the *ping* command which generated Internet Control Message Protocol (ICMP) echo request packets. The target host was expected to reply back to the sender using ICMP echo response packets for a successful ping. Both hosts were made to ping each other. Before ping could send out the first ICMP request, the hosts had to resolve the IP address of the target. The first test case employed conventional ARP for address resolution, and in the second case, ELK was deployed in the network to perform this resolution.



```
64 bytes from 192.168.1.70: icmp_seq=1 ttl=64 time=8.47 ms
64 bytes from 192.168.1.70: icmp_seq=2 ttl=64 time=0.481 ms
64 bytes from 192.168.1.70: icmp_seq=3 ttl=64 time=0.455 ms
64 bytes from 192.168.1.70: icmp_seq=4 ttl=64 time=0.513 ms
64 bytes from 192.168.1.70: icmp_seq=5 ttl=64 time=0.448 ms
64 bytes from 192.168.1.70: icmp_seq=6 ttl=64 time=0.481 ms
64 bytes from 192.168.1.70: icmp_seq=7 ttl=64 time=0.510 ms
64 bytes from 192.168.1.70: icmp_seq=8 ttl=64 time=0.519 ms
64 bytes from 192.168.1.70: icmp_seq=9 ttl=64 time=0.537 ms
64 bytes from 192.168.1.70: icmp_seq=10 ttl=64 time=0.561 ms
64 bytes from 192.168.1.70: icmp_seq=11 ttl=64 time=0.523 ms
64 bytes from 192.168.1.70: icmp_seq=12 ttl=64 time=0.495 ms
64 bytes from 192.168.1.70: icmp_seq=13 ttl=64 time=0.520 ms
64 bytes from 192.168.1.70: icmp_seq=14 ttl=64 time=0.493 ms
64 bytes from 192.168.1.70: icmp_seq=15 ttl=64 time=0.538 ms
64 bytes from 192.168.1.70: icmp_seq=16 ttl=64 time=0.570 ms
64 bytes from 192.168.1.70: icmp_seq=17 ttl=64 time=0.551 ms
64 bytes from 192.168.1.70: icmp_seq=18 ttl=64 time=0.563 ms
64 bytes from 192.168.1.70: icmp_seq=19 ttl=64 time=0.486 ms
64 bytes from 192.168.1.70: icmp_seq=20 ttl=64 time=0.529 ms
64 bytes from 192.168.1.70: icmp_seq=21 ttl=64 time=0.494 ms
64 bytes from 192.168.1.70: icmp_seq=22 ttl=64 time=0.506 ms
64 bytes from 192.168.1.70: icmp_seq=23 ttl=64 time=0.565 ms
64 bytes from 192.168.1.70: icmp_seq=24 ttl=64 time=0.528 ms
64 bytes from 192.168.1.70: icmp_seq=25 ttl=64 time=0.499 ms
```

Figure 4.5: Ping result in a network without ELK



```
64 bytes from 192.168.1.70: icmp_seq=1 ttl=64 time=629 ms
64 bytes from 192.168.1.70: icmp_seq=2 ttl=64 time=5.73 ms
64 bytes from 192.168.1.70: icmp_seq=3 ttl=64 time=1.72 ms
64 bytes from 192.168.1.70: icmp_seq=4 ttl=64 time=0.592 ms
64 bytes from 192.168.1.70: icmp_seq=5 ttl=64 time=0.594 ms
64 bytes from 192.168.1.70: icmp_seq=6 ttl=64 time=0.566 ms
64 bytes from 192.168.1.70: icmp_seq=7 ttl=64 time=0.580 ms
64 bytes from 192.168.1.70: icmp_seq=8 ttl=64 time=0.448 ms
64 bytes from 192.168.1.70: icmp_seq=9 ttl=64 time=0.512 ms
64 bytes from 192.168.1.70: icmp_seq=10 ttl=64 time=0.502 ms
64 bytes from 192.168.1.70: icmp_seq=11 ttl=64 time=0.584 ms
64 bytes from 192.168.1.70: icmp_seq=12 ttl=64 time=0.462 ms
64 bytes from 192.168.1.70: icmp_seq=13 ttl=64 time=0.494 ms
64 bytes from 192.168.1.70: icmp_seq=14 ttl=64 time=0.518 ms
64 bytes from 192.168.1.70: icmp_seq=15 ttl=64 time=0.462 ms
64 bytes from 192.168.1.70: icmp_seq=16 ttl=64 time=0.555 ms
64 bytes from 192.168.1.70: icmp_seq=17 ttl=64 time=0.479 ms
64 bytes from 192.168.1.70: icmp_seq=18 ttl=64 time=0.507 ms
64 bytes from 192.168.1.70: icmp_seq=19 ttl=64 time=0.550 ms
64 bytes from 192.168.1.70: icmp_seq=20 ttl=64 time=0.519 ms
64 bytes from 192.168.1.70: icmp_seq=21 ttl=64 time=0.468 ms
64 bytes from 192.168.1.70: icmp_seq=22 ttl=64 time=0.470 ms
64 bytes from 192.168.1.70: icmp_seq=23 ttl=64 time=0.468 ms
64 bytes from 192.168.1.70: icmp_seq=24 ttl=64 time=0.541 ms
64 bytes from 192.168.1.70: icmp_seq=25 ttl=64 time=0.466 ms
```

Figure 4.6: Ping result in a network with ELK

As seen from Figures 4.5 and 4.6, in the cases with and without ELK, after the initial few packets, the round trip times (RTT, which is the time taken by the packet to reach the target and the reply to return to the sender) for the packets were similar. The average RTTs, omitting the first two packets, were 0.519ms without ELK and 0.507ms with ELK, which is almost equivalent. The standard deviations were 0.041ms without ELK and 0.048ms with ELK. The minor divergence between the two averages is insignificant as it is possibly due to varying processing delays at the soft switches.

The major difference between the two scenarios lies in the initial packets. In the ELK network, these packets experienced much higher RTT.

As observed at **A** and **B** in Figure 4.5 and 4.6, ELK and ARP suffered a higher delay for the first packet in the test network. This is due to the use of OpenFlow protocol. The ICMP packet flow is a new flow, and none of the switches have an entry for it in their flow tables, the first ICMP request packet was sent to the controller by every OpenFlow switch it encountered. The controller interpreted this packet, generated an OpenFlow packet instructing the switch about the new flow, and then sent the ICMP and the OpenFlow packet back to the switch. The whole procedure was repeated for the first ICMP response. This forms one of the limitations of using the OpenFlow protocol. However the delay, as detailed above, applied only to the first packet.

The processing delay at the ELK server, as discussed in Section 4.1.2 was the major cause of the extra added latency as observed at **B** in Figure 4.6. The sender requested the IP to MAC mapping for the target using an ARP broadcast request before it could start sending out ICMP packets. This request, which it received from the controller, was eventually serviced by the ELK server. The server had to find the correct mapping in the network for this new request, before it could reply to the host.

Thus, although adding ELK into the network adds a delay to the first few packets of a new flow, it would not disrupt the network in a normal case.

4.1.4 Cache time

The ELK server maintained both the ARP table and the request table as caches, whereby the entries that were not renewed were evicted when they expired. It was crucial to maintain a caching time which was not too small, as otherwise there would be too many renewals, and not too large, as there would be stale entries being sent out in the network. Microsoft Windows XP [7] uses 4 minutes as the cache time for the ARP table and Linux kernel 2.6.35 has 1 minute as its default value for it.

Experiments with different ARP table cache times ranging from 30 seconds to 4 minutes were conducted on the network topology in Figure 4.1. There was no significant difference in the systems performance. The number of renewal unicast messages sent per unit time increased with decreasing cache time as the entries started to expire earlier. However, there is a risk of replying with stale values if the cache time is kept too high in order to avoid unicast messages. As a compromise, 1 minute was decided as the cache time for the ARP table.

When running the experiments on the request table cache time using the same setup, it was found that the cache time must strictly be greater than latency inherent in ELK but not too high. This was because, if the cache time was lower, the query would either expire or send out additional broadcasts even before the system finished processing the reply for it. This introduced unnecessary broadcasts in the network. On the other hand, having a high cache time proved to be useless, as either the querying application had aborted by the time a late reply came in or the application had already sent a repeat query. Thus, the smallest cache time with the least number of re-broadcasts by ELK was chosen. This value was set to 4 seconds.

4.1.5 Summary of Results

Through the implementation and evaluation of ELK, I have successfully gathered positive evidence showing that further research into ELK is a worthwhile endeavour. The evaluation showed significant reduction in the number of ARP broadcast messages across the network. This reduction decreases the ARP processing at the host level, which improves the efficiency of the hosts. Both of these factors, when combined, improved the overall performance of the network.

As part of the aims, the system implemented was backward compatible, as it did not disrupt any non-ARP traffic in the network and no hosts were aware of its existence. This system runs on top of the basic ARP and can be deployed in any network that uses ARP for address resolution. ELK also supports incremental updates of the hardware involved. The initial cost for it includes a server to run ELK, the OpenFlow controller, and at least one OpenFlow switch to channel the ARP requests to the server. The benefit of having more than one OpenFlow switch is that although the system can cope with one OpenFlow switch, more of these switches in the network will help convert the ARP broadcast into a unicast message sooner than later. The OpenFlow switches can initially be introduced as aggregation switches near large broadcast sources and later moved towards the edges. This ensures that the core switches are not subjected to the extra load.

4.2 My Project

My project has been successful and I achieved the goals set for it. I wrote detailed specifications for ELK, and implemented a fully functional server and controller based on Java and OpenFlow technologies, which help reduce ARP traffic in the network. I deployed this system in test networks, and ran experiments to test its effectiveness and compare its performance with conventional ARP. It showed significant improvement as predicted, but there are still areas, such as query latency, which require more research. As part of the preliminary background research for the project, I also taught myself new technologies, including OpenFlow and NOX.

4.3 Limitations

The evaluation stage was limited by several factors which reduced the effectiveness of the tests.

4.3.1 Number of test machines available

The number of machines with multiple Ethernet network interface adapters was limited which restricted the size and topologies of the test networks. This problem was most evident when evaluating the query latency, as a large network with long linear topology would have given a more realistic latency measure comparison. I made attempts to design more advanced topologies by using all the interfaces of each available machine, but as discussed in Section 4.1, this failed due to internal routing in the hosts kernel. The use of a modular approach, however, makes it easier to create topologies and revaluate the performance when more machines are available.

4.3.2 Limitations of OpenFlow

As outlined in Section 4.1.3, the first packet of any non-ARP traffic in the network suffered delays as none of the switches had flow entries for it in their flow tables. Furthermore, all ARP requests from the host had to be redirected to the controller over a TCP connection before they could be sent to the server over another TCP connection. This was due to the limited functionality in OpenFlow switches. The additional double overhead of encapsulating ARP packets as a TCP packet added up to the total processing delay experienced by ELK. This delay cannot be eradicated by the OpenFlow implementation and would require a more native implementation.

4.3.3 Use of Java

Java was used to design the server as it is a high level language which abstracts away unnecessary detail and helps in rapid evolutionary prototyping. It did, however, contribute to the additional processing time taken by the network to reply to a query. The performance of the real time implementation was lowered by synchronization in the threads and also by the enforced garbage collection [23]. Jpcap library used JNI to make libpcap library calls and these calls contribute to the overhead in the system processing [21].

If time had permitted it, I would have implemented the server using C++ (the language used to develop the NOX controller) and integrated it within the controller. This is because, now that I understand ELK and OpenFlow more thoroughly and the detailed specifications of ELK have been laid out, I could focus on increasing the efficiency of the system. This development would be directed at improving the real-time performance of the system by handling the data at a lower level. The controller and server integration would also overcome the need for an additional TCP connection between them which was an extra computational overhead in the current implementation. The C++ implementation might reduce the processing delay in the server as it will be able to access the hardware natively. This would thus, help model the query latency more accurately in comparison with ARP.

4.4 ELK

ELK, an ARP enhancement, has displayed significant performance improvements over basic ARP and has satisfied all the improvements predicted in [4]. Once initialised by the administrator, it is a self sustaining system which tries to improve the performance of the network. Although the protocol displayed high query latency, this problem can be mitigated with further research. This protocol has shown great promise and should be regarded by the research community as a suitable replacement for broadcast ARP. ELK can be further developed to incorporate more features and improve the quality of network traffic, some of which have been listed in Section 5.1.

Chapter 5

Conclusion

5.1 Future Work

I have included some possible extensions for my implementation for future research and development.

5.1.1 DHCP in ELK

In the current implementation, the IP addresses are either statically assigned by the administrator or dynamically allocated by a separate DHCP server. ELK server learns the IP address to MAC address mapping by listening to incoming ARP messages and by sending ARP broadcast requests. This implies that there is DHCP and ARP broadcast traffic in the network which takes up the bandwidth and is processed by all the hosts. By combining the functionalities of the DHCP and the ELK servers into one [4], the ELK server can learn the mapping directly from the DHCP server as soon as they are allocated. This will not only reduce broadcast traffic but also improve IP conflict detection.

5.1.2 Multiple ELK servers

Hosts in a large network with only one ELK server might experience huge delays during peak traffic. In order to overcome this, the master-slave paradigm [4] can be incorporated in the design. This design will have one ELK master which will have the most up to date copy of entries, and will also handle DHCP traffic.

Multiple ELK slaves will be distributed across the network which synchronise their ARP tables with the master server. The ARP broadcast queries will then be forwarded to the closest ELK server for service. This will improve scalability of the system by reducing query latency. Other beneficial side effects will include overcoming the problem of single point of failure in the system, and increasing stability against a Denial of Service attack.

5.1.3 Implementation for MOOSE

ELK was initially intended for use with MOOSE[4], a replacement protocol for Ethernet. Among its many benefits, MOOSE rewrites MAC addresses and ARP messages to reduce the size of the routing table, to better utilize the physical links and to reduce the end-to-end latency of transmission. The efficiency of the network would improve significantly if ELK were deployed in conjunction with MOOSE. There is the added benefit that the two protocols would share overhead costs, since the current implementation of MOOSE also uses OpenFlow.

5.2 Summary

Through this project, I successfully wrote the detailed specifications for, and completed what is the first implementation and proof-of-concept of ELK. This included making crucial design decisions during the development stages. The system supported incremental deployment in the network, an admin-friendly feature which was not mentioned in the original paper. I met my goals set in the project proposal which included proving the worthiness of this concept based on supporting evidence. For this, I designed evaluation test schemes which explicitly compared the performance of ELK against that of ARP. ELK, an unimplemented concept before this project, showed significant gains over ARP, a protocol that has dominated address resolution space for decades. Evaluation results showed substantial reduction in the broadcast traffic and hence considerably lower levels of ARP processing at the hosts without affecting the non-ARP traffic in the network. However, query latency in ELK was much higher than expected which was an artefact of the implementation. Several changes to the implementation have been suggested which might mitigate the problem. I have also included some possible future extensions to the ELK. I successfully achieved the aims set out for this project, and proved that with appropriate research, ELK can be developed to replace ARP.

Bibliography

[1] D. Plummer, “Ethernet Address Resolution Protocol: Or Converting Network Protocol Addresses to 48.bit Ethernet Address for Transmission on Ethernet Hardware.” RFC 826 (Standard), Nov. 1982. Updated by RFCs 5227, 5494.

[2] A. Myers, E. Ng, and H. Zhang, “Rethinking the service model: Scaling Ethernet to a million nodes,” in *Proc. HotNets*, Citeseer, 2004.

[3] H. Shah, A. Ghanwani, and N. Bitar, “ARP Broadcast Reduction for Large Data Centers (draft),” Oct. 2010. Expires May 2011.

[4] M. Scott, A. Moore, and J. Crowcroft, “Addressing the Scalability of Ethernet with MOOSE,” in *Proc. DC CAVES Workshop*, Sept. 2009.

[5] R. Droms, “Dynamic Host Configuration Protocol.” RFC 2131 (Draft Standard), Mar. 1997. Updated by RFCs 3396, 4361, 5494.

[6] J. Postel, “Internet Protocol.” RFC 791 (Standard), Sept. 1981. Updated by RFC 1349.

[7] MSDN, “MS-V4OF: IPv4 Over IEEE 1394 Protocol Extensions,” *Microsoft Corporation*, Mar. 2011.

[8] C. Kim, M. Caesar, and J. Rexford, “Floodless in SEATTLE: a scalable ethernet architecture for large enterprises,” in *Proceedings of the ACM SIGCOMM 2008 conference on Data communication*, pp. 3–14, ACM, 2008.

[9] J. Gosling and H. McGilton, “The Java Language Environment: A White Paper,” *Sun Microsystems: Mountain View, CA*, 2005.

[10] K. Fujii, “Jpcap–a Java library for capturing and sending network packets,” *Consultado na Internet em*, vol. 2, no. 09, 2008.

[11] V. Jacobson, C. Leres, and S. McCanne, “libpcap.” `http://www.tcpdump.org/`, June 1994.

[12] Sly Technologies, Inc., "jNetPcap OpenSource — A Libpcap/WinPcap Wrapper." `http://jnetpcap.com/`, 2009.

[13] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[14] D. Wagner-Hall, "NetFPGA Implementation of MOOSE," *Computer Science Part II Dissertation, Cambridge University*, May 2010.

[15] B. Pfaff, J. Pettit, T. Koponen, K. Amidon, M. Casado, and S. Shenker, "Extending networking into the virtualization layer," *Proc. HotNets*, Oct. 2009.

[16] Z. Cai, A. L. Cox, and T. S. E. Ng, "Maestro: A System for Scalable OpenFlow Control," tech. rep., Rice University, Dec. 2010.

[17] The NOX Team, "Developing in NOX." `noxrepo.org/noxwiki/index.php/Developing_in_NOX`. Retrieved 2011-03-30.

[18] W. Stevens and G. Wright, *TCP/IP Illustrated: the protocols*, vol. 1. Addison-Wesley, 1994.

[19] Oracle, "Java 2 platform standard edition 6.0 API specification," *Oracle Web site. download.oracle.com/javase/6/docs/api/java/util/HashMap.html*, 2011.

[20] S. Whalen, "An introduction to ARP spoofing." `http://www.rootsecure.net/content/downloads/pdf/arp_spoofing_intro.pdf`, Apr. 2001.

[21] S. Wilson and J. Kesselman, *Java Platform Performance: Strategies and Tactics*. Prentice Hall, 2000.

[22] C. Rostos, N. Sarrar, S. Uhlig, R. Sherwood, and A. Moore, "Deep diving into performance and scalability of OpenFlow implementations." 2011.

[23] R. Martin, "Java and C++ A critical comparison," *Java Gems: Jewels from Java Report*, pp. 51–68, 1998.

[24] ISO/IEC 7498-1, "Information technology - Open Systems Interconnection - Basic Reference Model," Nov. 1994.

[25] A. Moore, L. James, A. Wonfor, I. White, R. Penty, M. Glick, and D. McAuley, "Chasing errors through the network stack: a testbed for in-

vestigating errors in real traffic on optical networks," *Communications Magazine, IEEE*, vol. 43, pp. s34 – s39, aug. 2005.

Appendix A

The OSI Model

The Open Systems Interconnection (OSI) model often divides communication systems into layers[24]. Each layer has certain characteristics that define it, and it has various protocols normally associated with it. They are responsible for performing a particular type of task, as well as for interacting with the layers above and below it. Lower layers perform more concrete functions, such as hardware modulation and low-level communication. They provide services to the upper layers. The upper layers in turn use these services to implement more abstract functions, such as implementing user applications.

The model defines the layers as follows:

1. **Physical layer**: It deals with the signalling and encoding of the data, and physical transmission of it over the network.
2. **Data Link layer**: This is responsible for data framing and basic addressing. It also controls access to the network medium. A prime example of the protocol used in this layer is Ethernet.
3. **Network layer**: The third layer manages logical addressing and high level routing in the network. Internet Protocol (IP) is the most commonly used network layer protocol.
4. **Transport layer**: It enables communication between software application processes on different computers. TCP and UDP are the most notable examples.
5. **Session layer, Presentation layer and Application layer**: These layers together form the upper layers which are responsible for session manage-

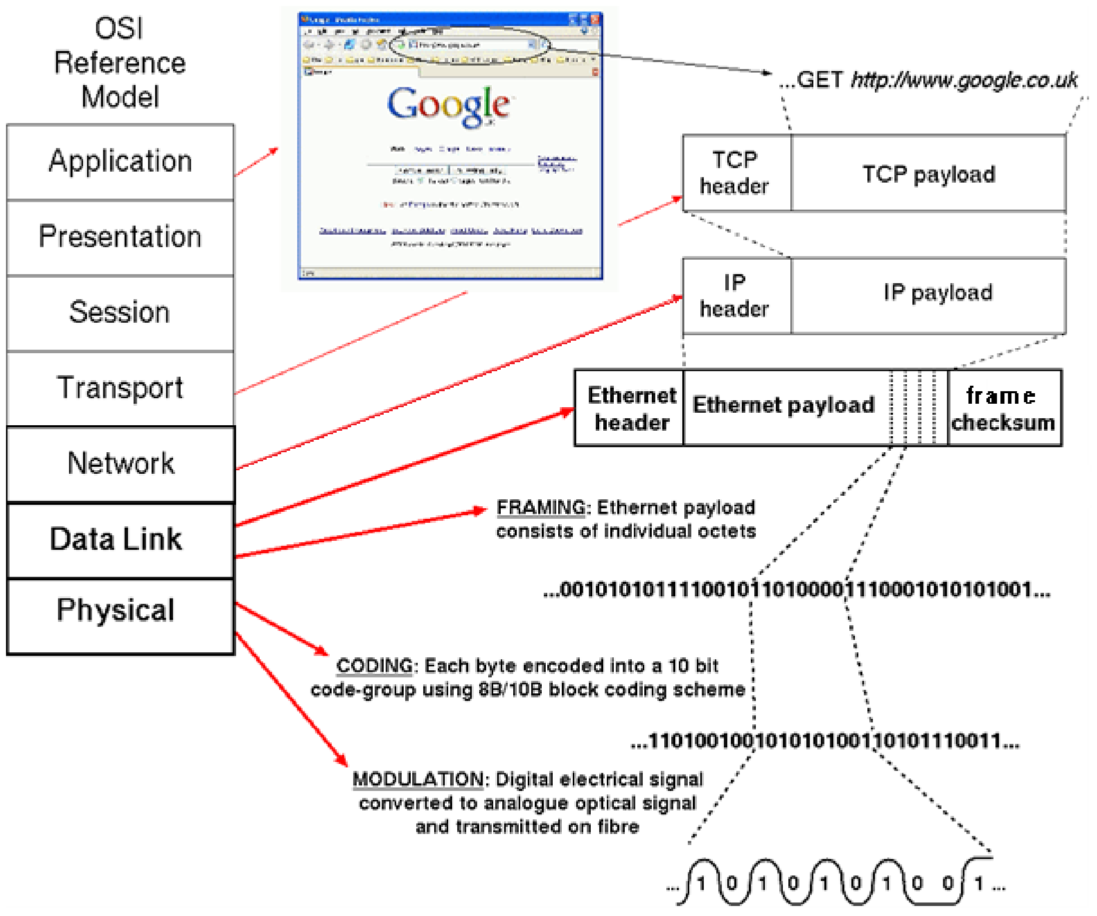


Figure A.1: OSI model of network protocol layers. Adapted from [25, Figure 1.1]

ment, compression and encryption, and user application services. HTTP is an example of the application layer protocol.

Appendix B

Address Resolution Protocol Packet Format

The Address Resolution Protocol (ARP) structure employs a simple message format. The standard format [1] for an ARP packet is as follows:

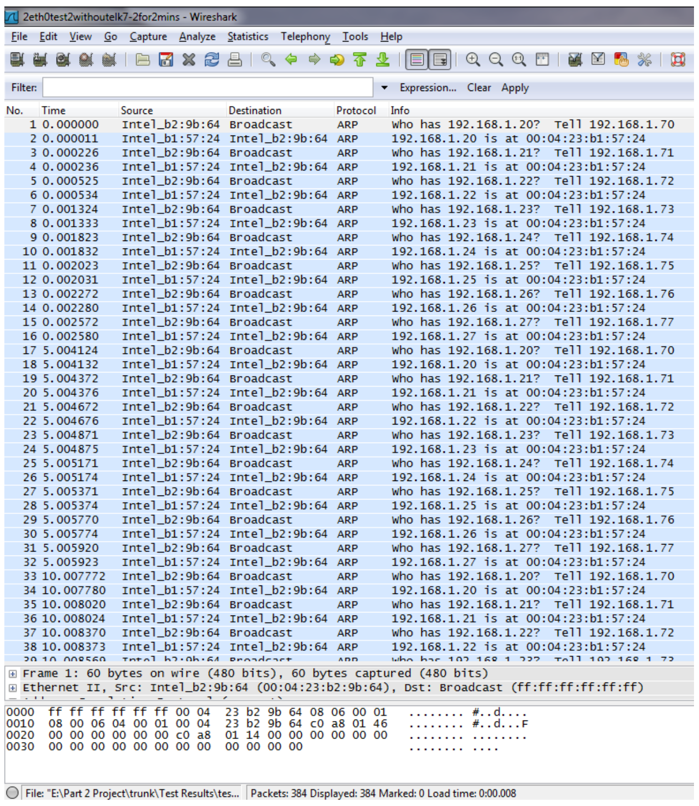
```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Hardware type                  |Protocol type                  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Hardware Length|Protocol Length|Operation                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Sender Hardware address                                     ...|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|...                            |Sender protocol address     ...|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|...                            |Target hardware address     ...|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Target protocol address                                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

A packet with Hardware Type set to 1 and Protocol Type set to 0x0800 indicates that it resolving Ethernet addresses for IP addresses. The operation defines the action intended by the sender, such as request or reply. Full details are specified in the RFC826[1].

Appendix C

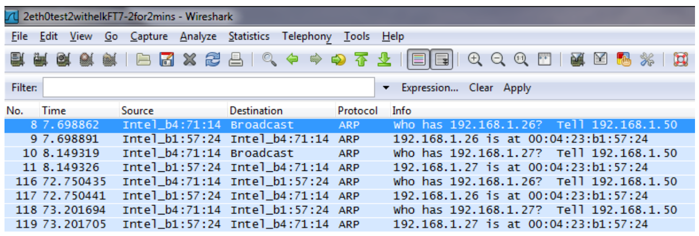
Figures of Results

Reduction in ARP Traffic Test - Wireshark screenshots

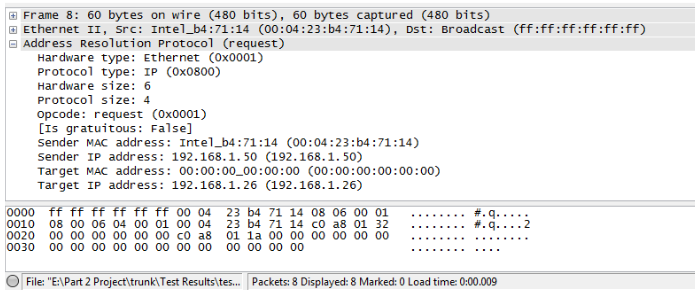

2eth0test2withoutelk7-2for2mins - Wireshark
File Edit View Go Capture Analyze Statistics Telephony Tools Help
Filter:
Expression... Clear Apply

```
No.  Time        Source          Destination     Protocol Info
  1 0.000000    Intel_b2:9b:64  Broadcast       ARP      Who has 192.168.1.20?  Tell 192.168.1.70
  2 0.000011    Intel_b1:57:24  Intel_b2:9b:64  ARP      192.168.1.20 is at 00:04:23:b1:57:24
  3 0.000226    Intel_b2:9b:64  Broadcast       ARP      Who has 192.168.1.21?  Tell 192.168.1.71
  4 0.000236    Intel_b1:57:24  Intel_b2:9b:64  ARP      192.168.1.21 is at 00:04:23:b1:57:24
  5 0.000525    Intel_b2:9b:64  Broadcast       ARP      Who has 192.168.1.22?  Tell 192.168.1.72
  6 0.000534    Intel_b1:57:24  Intel_b2:9b:64  ARP      192.168.1.22 is at 00:04:23:b1:57:24
  7 0.001324    Intel_b2:9b:64  Broadcast       ARP      Who has 192.168.1.23?  Tell 192.168.1.73
  8 0.001333    Intel_b1:57:24  Intel_b2:9b:64  ARP      192.168.1.23 is at 00:04:23:b1:57:24
  9 0.001823    Intel_b2:9b:64  Broadcast       ARP      Who has 192.168.1.24?  Tell 192.168.1.74
 10 0.001832    Intel_b1:57:24  Intel_b2:9b:64  ARP      192.168.1.24 is at 00:04:23:b1:57:24
 11 0.002023    Intel_b2:9b:64  Broadcast       ARP      Who has 192.168.1.25?  Tell 192.168.1.75
 12 0.002031    Intel_b1:57:24  Intel_b2:9b:64  ARP      192.168.1.25 is at 00:04:23:b1:57:24
 13 0.002272    Intel_b2:9b:64  Broadcast       ARP      Who has 192.168.1.26?  Tell 192.168.1.76
 14 0.002280    Intel_b1:57:24  Intel_b2:9b:64  ARP      192.168.1.26 is at 00:04:23:b1:57:24
 15 0.002572    Intel_b2:9b:64  Broadcast       ARP      Who has 192.168.1.27?  Tell 192.168.1.77
 16 0.002580    Intel_b1:57:24  Intel_b2:9b:64  ARP      192.168.1.27 is at 00:04:23:b1:57:24
 17 5.004124    Intel_b2:9b:64  Broadcast       ARP      Who has 192.168.1.20?  Tell 192.168.1.70
 18 5.004132    Intel_b1:57:24  Intel_b2:9b:64  ARP      192.168.1.20 is at 00:04:23:b1:57:24
 19 5.004372    Intel_b2:9b:64  Broadcast       ARP      Who has 192.168.1.21?  Tell 192.168.1.71
 20 5.004376    Intel_b1:57:24  Intel_b2:9b:64  ARP      192.168.1.21 is at 00:04:23:b1:57:24
 21 5.004672    Intel_b2:9b:64  Broadcast       ARP      Who has 192.168.1.22?  Tell 192.168.1.72
 22 5.004676    Intel_b1:57:24  Intel_b2:9b:64  ARP      192.168.1.22 is at 00:04:23:b1:57:24
 23 5.004871    Intel_b2:9b:64  Broadcast       ARP      Who has 192.168.1.23?  Tell 192.168.1.73
 24 5.004875    Intel_b1:57:24  Intel_b2:9b:64  ARP      192.168.1.23 is at 00:04:23:b1:57:24
 25 5.005171    Intel_b2:9b:64  Broadcast       ARP      Who has 192.168.1.24?  Tell 192.168.1.74
 26 5.005174    Intel_b1:57:24  Intel_b2:9b:64  ARP      192.168.1.24 is at 00:04:23:b1:57:24
 27 5.005371    Intel_b2:9b:64  Broadcast       ARP      Who has 192.168.1.25?  Tell 192.168.1.75
 28 5.005374    Intel_b1:57:24  Intel_b2:9b:64  ARP      192.168.1.25 is at 00:04:23:b1:57:24
 29 5.005770    Intel_b2:9b:64  Broadcast       ARP      Who has 192.168.1.26?  Tell 192.168.1.76
 30 5.005774    Intel_b1:57:24  Intel_b2:9b:64  ARP      192.168.1.26 is at 00:04:23:b1:57:24
 31 5.005920    Intel_b2:9b:64  Broadcast       ARP      Who has 192.168.1.27?  Tell 192.168.1.77
 32 5.005923    Intel_b1:57:24  Intel_b2:9b:64  ARP      192.168.1.27 is at 00:04:23:b1:57:24
 33 10.007772   Intel_b2:9b:64  Broadcast       ARP      Who has 192.168.1.20?  Tell 192.168.1.70
 34 10.007780   Intel_b1:57:24  Intel_b2:9b:64  ARP      192.168.1.20 is at 00:04:23:b1:57:24
 35 10.008020   Intel_b2:9b:64  Broadcast       ARP      Who has 192.168.1.21?  Tell 192.168.1.71
 36 10.008024   Intel_b1:57:24  Intel_b2:9b:64  ARP      192.168.1.21 is at 00:04:23:b1:57:24
 37 10.008370   Intel_b2:9b:64  Broadcast       ARP      Who has 192.168.1.22?  Tell 192.168.1.72
 38 10.008373   Intel_b1:57:24  Intel_b2:9b:64  ARP      192.168.1.22 is at 00:04:23:b1:57:24
 39 10.008569   Intel_b2:9b:64  Broadcast       ARP      Who has 192.168.1.23?  Tell 192.168.1.73

⊞ Frame 1: 60 bytes on wire (480 bits), 60 bytes captured (480 bits)
⊞ Ethernet II, Src: Intel_b2:9b:64 (00:04:23:b2:9b:64), Dst: Broadcast (ff:ff:ff:ff:ff:ff)

0000  ff ff ff ff ff ff 00 04  23 b2 9b 64 08 06 00 01   ........ #..d....
0010  08 00 06 04 00 01 00 04  23 b2 9b 64 c0 a8 01 46   ........ #..d...F
0020  00 00 00 00 00 00 c0 a8  01 14 00 00 00 00 00 00   ........ ........
0030  00 00 00 00 00 00 00 00  00 00 00 00               ........ ....

File: "E:\Part 2 Project\trunk\Test Results\tes...   Packets: 384 Displayed: 384 Marked: 0 Load time: 0:00.008
```

Figure C.1: Screenshot of Wireshark packet capture using conventional ARP


2eth0test2withelkFT7-2for2mins - Wireshark
File Edit View Go Capture Analyze Statistics Telephony Tools Help
Filter:
Expression... Clear Apply

```
No.  Time       Source          Destination     Protocol  Info
  8 7.698862   Intel_b4:71:14  Broadcast       ARP       Who has 192.168.1.26?  Tell 192.168.1.50
  9 7.698891   Intel_b1:57:24  Intel_b4:71:14  ARP       192.168.1.26 is at 00:04:23:b1:57:24
 10 8.149319   Intel_b4:71:14  Broadcast       ARP       Who has 192.168.1.27?  Tell 192.168.1.50
 11 8.149326   Intel_b1:57:24  Intel_b4:71:14  ARP       192.168.1.27 is at 00:04:23:b1:57:24
116 72.750435  Intel_b4:71:14  Intel_b1:57:24  ARP       Who has 192.168.1.26?  Tell 192.168.1.50
117 72.750441  Intel_b1:57:24  Intel_b4:71:14  ARP       192.168.1.26 is at 00:04:23:b1:57:24
118 73.201694  Intel_b4:71:14  Intel_b1:57:24  ARP       Who has 192.168.1.27?  Tell 192.168.1.50
119 73.201705  Intel_b1:57:24  Intel_b4:71:14  ARP       192.168.1.27 is at 00:04:23:b1:57:24
```



```
⊞ Frame 8: 60 bytes on wire (480 bits), 60 bytes captured (480 bits)
⊞ Ethernet II, Src: Intel_b4:71:14 (00:04:23:b4:71:14), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
⊟ Address Resolution Protocol (request)
    Hardware type: Ethernet (0x0001)
    Protocol type: IP (0x0800)
    Hardware size: 6
    Protocol size: 4
    Opcode: request (0x0001)
    [Is gratuitous: False]
    Sender MAC address: Intel_b4:71:14 (00:04:23:b4:71:14)
    Sender IP address: 192.168.1.50 (192.168.1.50)
    Target MAC address: 00:00:00_00:00:00 (00:00:00:00:00:00)
    Target IP address: 192.168.1.26 (192.168.1.26)

0000  ff ff ff ff ff ff 00 04  23 b4 71 14 08 06 00 01   ........ #.q.....
0010  08 00 06 04 00 01 00 04  23 b4 71 14 c0 a8 01 32   ........ #.q....2
0020  00 00 00 00 00 00 c0 a8  01 1a 00 00 00 00 00 00   ........ ........
0030  00 00 00 00 00 00 00 00  00 00 00 00               ........ ....

File: "E:\Part 2 Project\trunk\Test Results\tes...   Packets: 8 Displayed: 8 Marked: 0 Load time: 0:00.009
```

Figure C.2: Screenshot of Wireshark packet capture using ELK with 6 static entries and 2 dynamic entries

Appendix D

Project Proposal

Network Protocol Implementation and Evaluation

Ishaan Aggarwal
Corpus Christi College
ia264@cam.ac.uk

Project Originator: Malcolm Scott

Project Supervisor: Andrew Moore

Director of Studies: David Greaves

Project Overseers: Frank Stajano & Robert Mullins

Special Resources Required

- Access to SRG/NetOS machines.
- Physical access to SE18 and SW02 at all times during the project
- Account in CL (with at least 1GB space)
- NetFPGA machines - at least 3 initially and as many as possible later for more comprehensive testing
- 25 GB of scratch space

Introduction

At Data Link layer, the incoming network frame contains the destination Internet Protocol (IP) Address and this frame might be missing the destination physical address which is known as Media Access Control (MAC) address. The destination MAC address is necessary for the frame to be delivered to the correct hosts Network Interface Controller (NIC).

Currently IP or Ethernet protocol uses the Address Resolution Protocol (ARP) [1] whereby if the sending host does not have the destination MAC address in its ARP cache, it will broadcast an ARP query containing the destination IP address. One of the relevant hosts will send an ARP reply which will map the IP address to MAC address which is cached by the querying host. The switches connecting the hosts together learn the MAC address to their port mapping in order to switch MAC addresses.

When this concept is applied in networks with a large number of hosts, the hosts are not able to hold all entries in their ARP cache and start to drop old entries. There is a large number of broadcast ARP queries as the hosts are trying to find the MAC address for a given IP address This adds up to the network traffic and causes unnecessary congestion. [2]

The idea of this project is to implement Enhanced Lookup (ELK) directory service. It was suggested by Scott [3] in his paper on MOOSE which is a proposed replacement to Ethernet as layer-2 protocol by using hierarchical MAC address to ensure scalability. ELK is implementable on both MOOSE and Ethernet.

ELK will hold the IP to MAC mapping table. An ELK enabled switch will unicast their query to the ELK server instead of forwarding the broadcast query to all its port. ELK will check its own table to answer the query and send the reply

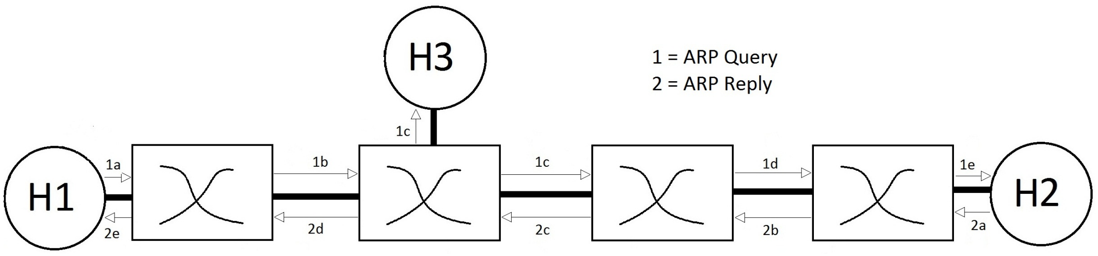


Figure D.1: ARP in simple line topology - 4 switches, 3 hosts, 6 ARP queries and 5 ARP replies

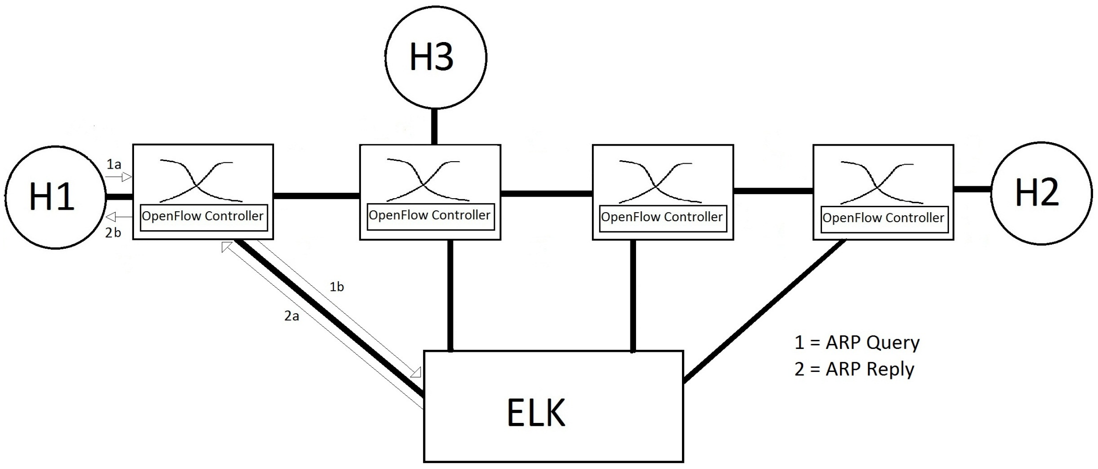


Figure D.2: ARP with ELK in line topology - 4 switches, 3 hosts, 2 ARP queries and 2 ARP replies

to the switch. If ELK doesnt have the entry, it will broadcast the query and add the reply into its table. Since ELK is an auto learning service, the first query from each host which hasnt communicated in the network will be broadcasted so that it can build up its table.

ELK is backward compatible because if a switch is not ELK enabled, it will continue to behave as normal. Only when a frame with missing destination MAC address reaches a ELK enabled switch is the query forwarded to the ELK.

Work to do

All the programming in this project will be done in software, mostly in C/C++, Python and Java.

- Write out the specifications for the protocol to encapsulate ARP queries and replies.
- Implement the ELK directory service which can do the following:
  - Extract and store the IP address to MAC address of the querying host from the incoming ARP query.
  - Send a broadcast ARP query if it does not have an IP to MAC mapping
  - Send an encapsulated ARP reply back to the querying switch with the MAC address in question.

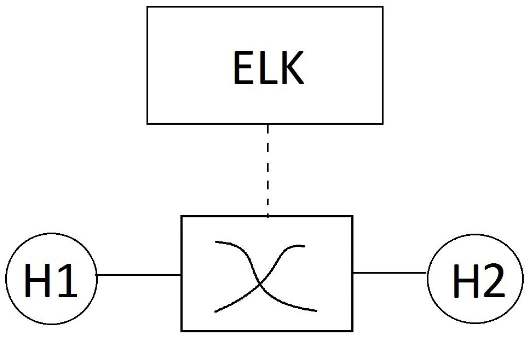


Figure D.3: Trivial one switch topology

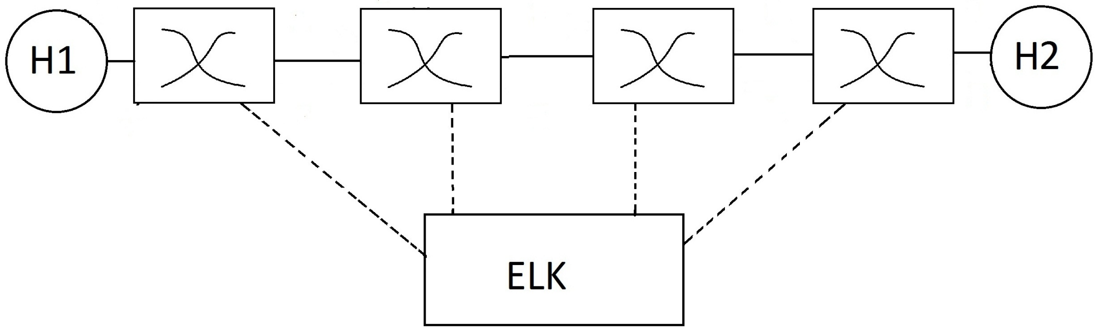


Figure D.4: Long Line Topology

- Maintain connections with the switches in the network. When it is first introduced into the network or recovers from failure, it should establish connection with the switches and activate the OpenFlow module in them so as to redirect ARP traffic.

- Write a new OpenFlow module for network switch so that it can do the following:
  - Detect all ARP queries in the traffic passing through it, then encapsulate them and send it as a unicast to the ELK.
  - Check if an ELK server exists and only then perform a unicast. Otherwise, it should carry on with the broadcast of ARP query.
  - On receiving the encapsulated ARP reply, it should extract the reply and send it back to the querying host.
- Perform comprehensive testing on the whole system as follows:

- The OpenFlow switch can be tested modularly by forming a trivial network whereby a local machine sends variety of packets to the switch from one of its interface and after switch processing, the machine receives only encapsulated ARP queries at another interface.
- Again, ELK can be modularly tested by forming a trivial network using a soft switch which is implemented on a local machine with multiple interfaces. The test can be performed by sending encapsulated ARP queries from switch to ELK and then monitoring the replies sent back by ELK. ELK can be connected to other switches and hosts in order to verify that the broadcast messages are sent out correctly when the mapped entries are missing.
- Perform regressing testing on networks with and without ELK using different topologies such as trivial one switch and long line topologies and then using the data to evaluate the performance of both.

Initial Learning Tasks

In order to work on the project, the following learning tasks will have to be undertaken:

- learning about OpenFlow Switch Programming and NOX;
- learning about ARP and switches in more detail;
- devising suitable test suite topologies and regression tests in order to evaluate the success of ELK.

Starting Point

I have a good understanding of the IP, protocols involved and switches from the Digital Communication I course from Part IB. Knowledge gained from the course Programming in C and C++ in Part IB will also be relevant here.

I will be using OpenFlow, an open standard which helps gain control of the forwarding tables in the network switch, to implement ELK protocol on the switch. To develop the OpenFlow controller, I will be using NOX, a library (available in both C++ and python) which allows the developer to determine the flows in the networks and the paths taken by the various flows.

Resources

SRG/NetOS machines will be used to form different network topologies to perform regression testing on networks with and without ELK.

In order to backup my code and documentation and also maintain a version control, I plan to use the Subversion (SVN) service provided on the Cambridge PWF. I also intend to use professional automatic offsite cloud-based backup services such as Spider Oak to keep an encrypted and version controlled copy of my work in case of a mishap.

Assessment Criteria

As a minimum requirement, ELK should not be any worse than when no directory service is used. However, this condition will not be satisfied in trivial configurations of less than one switch in the network. It should be able to work with IPv4.

A comparison of ELK against standard Ethernet or IP without any ARP proxy should show that ELK reduces ambient traffic by cutting down the number of broadcast messages in the network beyond a certain size (more than 2 switches). It should also reduce latency and hence enable faster application.

Possible extensions

1. One of the possible extensions is that ELK will support DHCP. This has the advantage that ELK would have the IP to MAC address mapping for all the hosts it allocates IP address to and also know the lease time for the all the hosts. This will reduce the initial broadcast messages as ELK will not need to learn about the host addresses themselves.
2. Since all the ARP traffic will be now routed via the ELK, an anti-ARP spoofing technique can be implemented in ELK.

3. Another extension could be having multiple ELKs in the network in order to facilitate load balancing and removing the danger of single point of failure.

Work Plan

	Date	Objective
1	2010-10-25	Start of Work
2	2010-11-15	Finish setting up the development environment and understanding the working of OpenFlow and NOX.
3	2010-12-15	Implement ELK directory service
4	2011-01-10	**Milestone:** Finish ELK system testing
5	2011-01-20	Implement the ELK protocol in the OpenFlow switch
6	2011-02-04	**Deliverable:** Progress Report Deadline
7	2011-02-15	Continue and finish the OpenFlow switch implementation
8	2011-03-01	**Milestone:** Finish testing of OpenFlow Switch
9	2011-03-20	Finish collecting all the test data from complete system regression testing
10	2011-04-10	Complete Dissertation Draft
11	2011-05-20	**Deliverable:** Dissertation Deadline

References

[1] D. Plummer, An Ethernet Address Resolution Protocol, RFC 826, November 1982, updated by RFC 5227, 5494. [Online]. Available: http://tools.ietf.org/html/rfc826

[2] A. Myers, E. Ng, and H. Zhang, Rethinking the Service Model: Scaling Ethernet to a Million Nodes, in ACM SIGCOMM Workshop on Hot Topics in Networking, Nov. 2004

[3] M. Scott, A. Moore, and J. Crowcroft, Addressing the Scalability of Ethernet with MOOSE in Proc. DC CAVES Workshop, Sept. 2009