Daniel Wagner-Hall

NetFPGA Implementation of MOOSE

Computer Science Tripos, Part II

2009-2010

Daniel Wagner-Hall, Homerton College

Proforma

Name:	**Daniel Wagner-Hall**
College:	**Homerton College**
Project Title:	**NetFPGA implementation of MOOSE**
Examination:	**Computer Science Tripos, Part II, 2009-2010**
Word Count:	**Approximately 11,950 words**
Project Originator:	Malcolm Scott
Supervisor:	Dr A. W. Moore

Original Aims of the Project

I set out to prototype network switch implementing MOOSE, a contemporary research network protocol to replace Ethernet, using a specialist FPGA. With this prototype, I aimed to gather several metrics comparing MOOSE to Ethernet, and to evaluate MOOSE in terms of its proposed improvements.

Work Completed

I completed all of the work which I set out to do, including the extensions I had set to make my switch more flexible and add extra features. I gathered several metrics showing MOOSE to be an improvement over Ethernet, and outlined several areas for future research. My work is being put forward for publication and has fed directly into active research.

Special Difficulties

None.

Declaration

I, Daniel Wagner-Hall of Homerton 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 MOOSE (indeed giving me a dissertation to do), advice, support and encouragement, I wish to thank Jon Crowcroft, Malcolm Scott, and especially my supervisor Andrew Moore.

For countless hours debugging horrible awkwardness with NetFPGAs, occasionally my fault, occasionally entirely random, thanks to David Miller.

My utmost gratitude is owed to Matthew Hodgson for mentoring me through my first programming job, you're amazing.

I am also extremely grateful to Simon Stewart — TDD changed my life, and that was all you.

Tilda: You and Steve Holt have kept me sane.

Emma, Otis, you've seen me through the good, the bad, and the ridiculously shifted sleeping schedules. Thank you.

Chapter 1

Introduction

Ethernet, a dominant network protocol, has been used in networks for several decades and is beginning to show its age. No obvious general-purpose replacement has been found for Ethernet. Scalability has long limited Ethernet deployments — Heathrow Terminal 5 could not use Ethernet for its networking needs [1] and this motivated the Intelligent Airport project [2]. Several properties are desirable in a replacement: compatibility with existing systems, the ability to incrementally introduce replacement hardware to networks, minimal equipment to be obsoleted or to require update, and significant performance improvements. At the forefront of research, the Intelligent Airport project has recently proposed Multi-level Origin-Organised Scalable Ethernet (MOOSE) [3] as a replacement for Ethernet which meets all of these criteria. No complete implementation of MOOSE exists, so no evidence has been given that these criteria are met beyond conjecture. In this paper, I set out to create an implementation of MOOSE to provide this evidence, so that MOOSE can be reasonably judged by the research community and compared with incumbent Ethernet standards.

1.1 Background

Network architecture can be conceptually divided into layers of protocols, each forming a separate level of abstraction. Layers interact with the layers directly above and below themselves. The bottom layer is the **physical layer** — the physical means of connecting computers to each other and encoding bits onto a medium, such as 10BASE-T. Above this lies the **data-link layer**, providing basic addressing, and controlling access to the physical layer. The ubiquitous data-link-layer protocol is Ethernet. Above the data-link layer lies the **network layer**, providing for better large-scale routing, and more control over network traversal. The principal network-layer protocol is IP. Above the network layer lies the **transport layer**, providing multiplexing of network channels for applications between hosts. Examples are TCP and UDP. The three layers above these can be considered merged into a single layer of application data for the purposes

of addressing and routing.

Each layer may add headers to the front of the data payload being transmitted, which may be used by routers when deciding where to send packets, by firewalls when deciding to drop packets, and by hosts when deciding which application(s) to notify when packets are received. Figure 1.1 illustrates this layering.

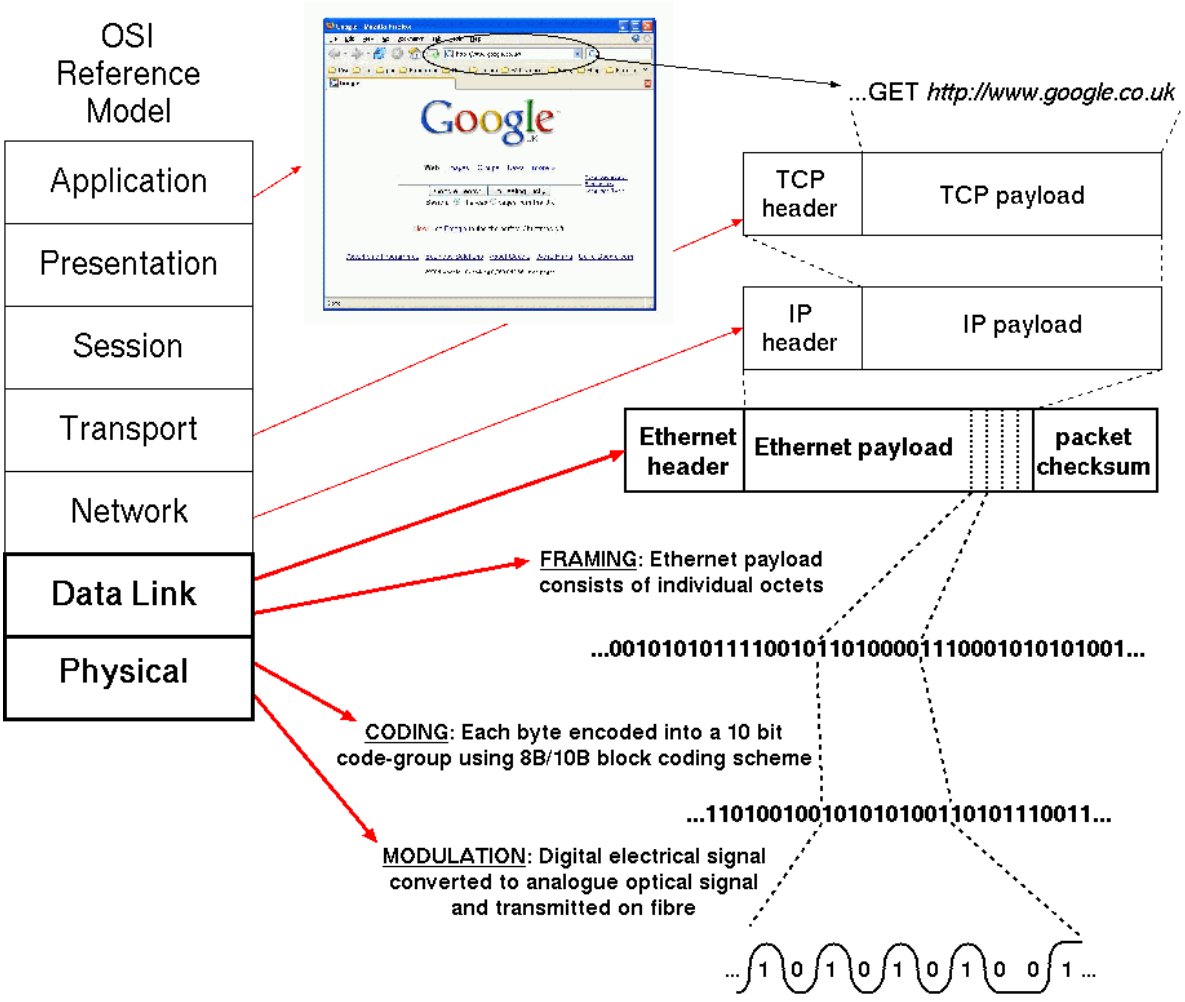
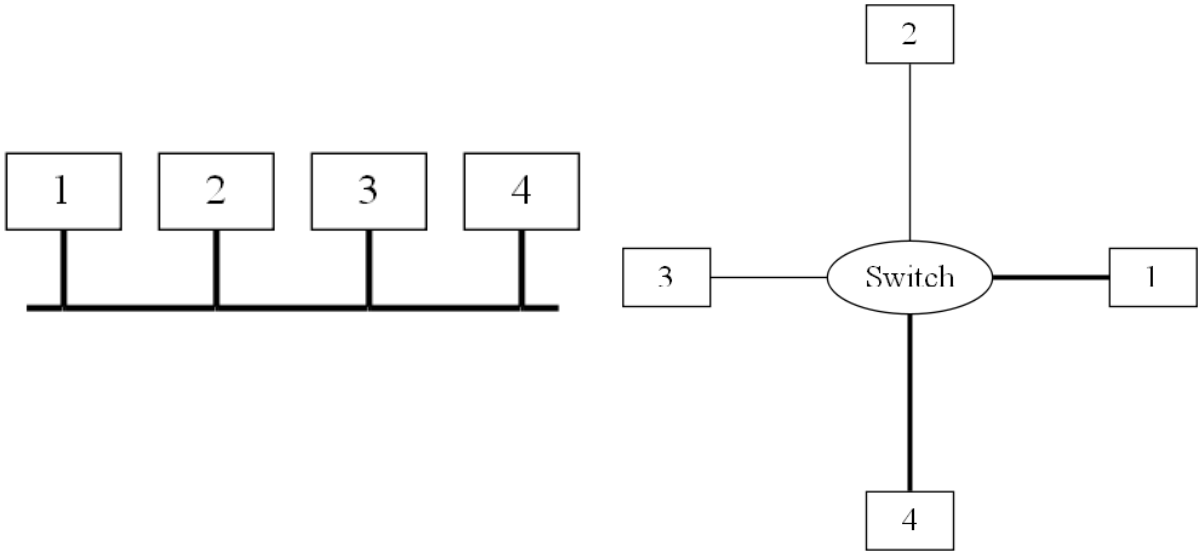


Figure 1.1: The OSI model of network protocol layers. Amended from [4, Figure 1]

1.1.1 Ethernet

The principal data-link-layer protocol used in networks today is Ethernet. Ethernet was first proposed in 1976 with the objective "to design a communication system which can grow smoothly to accommodate several buildings full of personal computers" [5]. Originally, Ethernets were shared medium networks, using a single coaxial cable between all computers. All computers on a network would receive every packet that was sent, and a computer's Ethernet controller would determine whether to notify the operating

system of them based on their Ethernet destination address. The important property of Ethernet addresses was uniqueness. No information was required from an Ethernet address other than that it uniquely identify a destination (or source). The format of Ethernet addresses, which are 6 bytes long, is that the first three bytes of the address are unique to the manufacturer which created the device (as allocated by the IEEE), and the last three bytes of the address are guaranteed by that manufacturer to be unique. Ethernet addresses are typically written as six hex. pairs, e.g. `00:01:02:03:04:05`. In this example `00:01:02` indicates the manufacturer (here 3Com), and the whole address `00:01:02:03:04:05` is guaranteed by 3Com to be unique.



(a) A shared medium network (b) A switched network

Figure 1.2: Comparison of shared-medium and star network topologies
Bold links are those used (thus not free for other use) when 1 and 4 communicate

As scalability became an issue, and because coaxial cable was much more expensive than twisted pair cable, Ethernet was updated in 1990 [6] to support switching; using wires connected only to one host or switch at each end. Switches are used to join multiple hosts, controlling where in the network packets need to be sent, as opposed to having a long wire connecting all hosts. Whereas IP addresses (scalable, network-layer addresses) are allocated in blocks of nearby nodes within a network, so that routing decisions can be based on the addresses' prefixes, Ethernet addresses (data-link-layer addresses) give no such routing information. In Ethernet, therefore, a switch's forwarding table is built up by remembering the source address and physical port of every packet which is seen. No aggregation of nearby nodes can take place because Ethernet addresses have no hierarchy. These forwarding tables must be kept in fast, expensive memory, particularly as physical layer speeds increase. Typically they are stored in Content Addressable Memory (CAM), a specialist form of hardware which performs lookups in parallel and gives $O(1)$ rather than $O(n)$ retrieval on unsorted data. CAM is expensive, requires lots of power (a particular problem in datacentres),

and does not efficiently scale to many entries [7], giving most switches a practical limit of storing approximately 16,000 entries in this table [8]. This fundamentally limits the size of networks. When the forwarding table becomes full, entries are discarded. Packets with previously unseen or discarded destination addresses are flooded to all ports, resulting in excessive traffic. This is particularly problematic for low-capacity edge links, which can become congested. All of these problems arise because no useful routing information is embedded within an Ethernet address.

One bit of Ethernet addresses indicates whether the address has been assigned by a manufacturer to a physical device, and is thus universally unique, or has been manually allocated by a network administrator. This feature is rarely used, and in practice addresses almost always have this bit set to manufacturer-assigned.

Ethernet, a flat-address based protocol, has no routing protocol associated with it, because its addresses give no scope for one. Issues naturally arise from networks containing loops, because there is no way to detect loops, or select paths for packets to avoid loops. Ethernet's mechanism to cope with loops is to use the Rapid Spanning Tree Protocol (RSTP) [9, §17] to form a spanning tree of the network before any traffic is allowed on it, and disable redundant loop-causing links. This constrains packets to suboptimal routes, reducing capacity. In datacentres, this is a particular problem where redundancy is often planned to increase bandwidth and throughput, but is disabled.

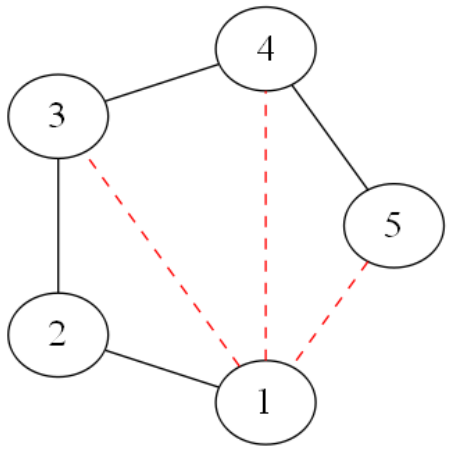


Figure 1.3: Diagram showing issues with using RSTP

Figure 1.3 shows the problems with using RSTP — the dashed links having been disabled by RSTP may not be used, so that if 1 wishes to communicate with 5, rather than using the link directly between 1 and 5, the packets would need to travel via switches 2, 3 and 4, unnecessarily congesting those switches and links, and adding latency.

1.1.2 MOOSE, a Proposed Improvement

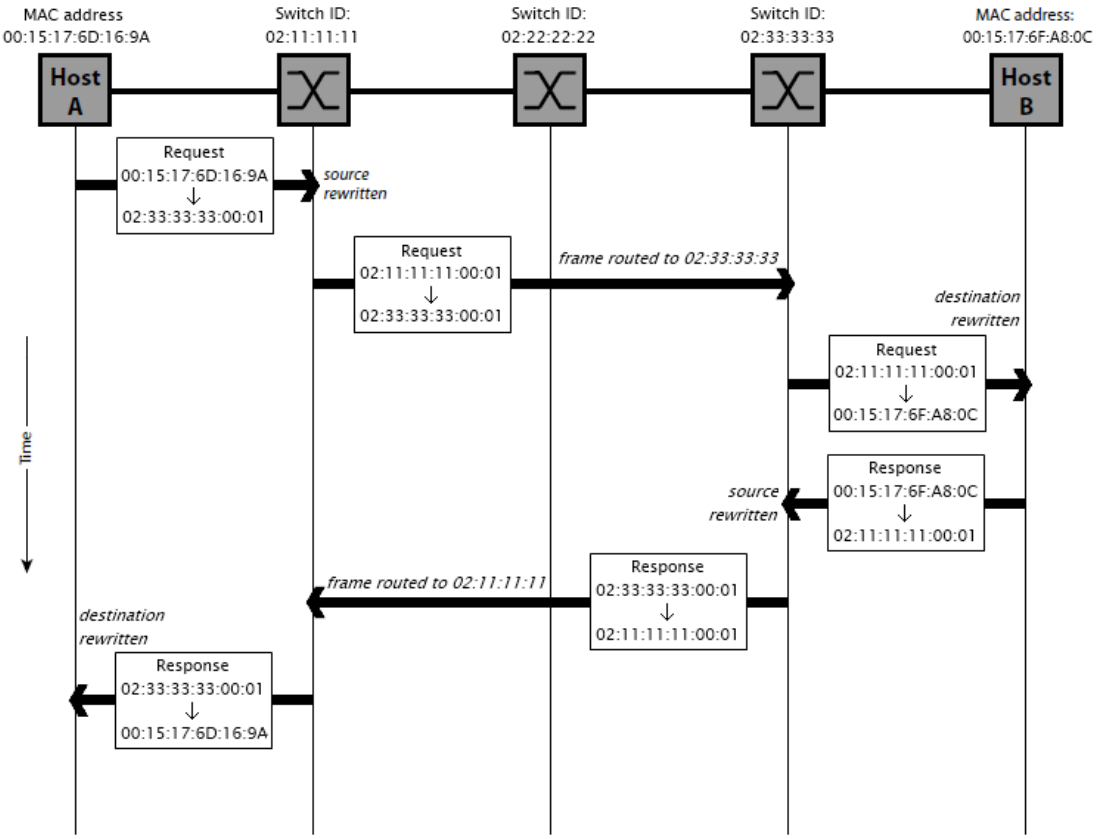


Figure 1.4: Sequence diagram of request and response using MOOSE.
Amended from [3, Figure 2]

Multi-level Origin-Organised Scalable Ethernet (MOOSE) [3] alleviates the issues of over-filling forwarding tables, as well as issues caused by the required use of RSTP, by introducing hierarchy into the Ethernet address space. MOOSE preserves backwards compatibility with Ethernet, and requires no host reconfiguration or modification. It only requires that switches be replaced (or ideally, have software updates installed), and this can be done incrementally through the network. Each switch is given an address of length less than six bytes. Every host in a MOOSE network necessarily has a nearest MOOSE switch. When this switch receives a packet from one of the hosts for which it is the nearest switch, it rewrites the source address of the packet to that host's MOOSE address. A host's MOOSE address is formed by concatenating the switch's address and some identifier for the host, decided by the switch. As an example, if our switch's address is `02:22:22:22`, it may assign an attached host the identifier of `AA:BB`, and so that host's MOOSE address would be `02:22:22:22:AA:BB`. The host is not aware of its MOOSE address. The switch rewrites the source address on packets sent from the host's Ethernet address to its MOOSE address. It also rewrites the destination address

of packets sent *to* the host's MOOSE address to its true Ethernet address so that the host's network interface knows to receive the packet rather than discard it. When a packet from a host enters the MOOSE network, only that host's MOOSE address is known to the rest of the network. The network does not ever see its Ethernet address, but the host itself is unaware of this.

Unlike standard Ethernet addresses, MOOSE addresses contain routing information, and so can be used in forwarding decisions. Switches need only to keep track of one entry in their forwarding table per switch in the network, rather than per host in the network, a massive reduction. Furthermore, a routing protocol such as OSPF [10] can be used to provide shortest-path routing between switches without RSTP disabling redundant links.

MOOSE addresses are distinguished from Ethernet addresses because MOOSE addresses have their administrator-assigned (i.e. not manufacturer-assigned) bit set, whereas it is never set in natural Ethernet addresses. Any Ethernet device ignores this distinction and the MOOSE address appears as any other Ethernet address, but any MOOSE-aware device can differentiate between them.

1.2 Context of Work

The ideas behind MOOSE are summarised by Scott *et al.* in [3], and a pure-software Python implementation of some parts of MOOSE has been created as a proof of concept. This prototype was not designed to operate in realistic conditions due to its software nature — though it could switch at 100Mb/s — it simply showed that the ideas behind MOOSE were sound, and its addressing system could be used transparently without modifying hosts. No detailed specification, reference implementation, or even realistic partial implementation exists.

1.3 Aims

I aimed to implement a MOOSE switch using an FPGA to give realistic operating conditions and constraints, and evaluate MOOSE's proposed improvements. Specifically, I aimed to show that MOOSE reduces the size of switches' forwarding tables, and permits shortest path routing without disabling any links. I also aimed to verify that resolution of conflicts of MOOSE addresses could be handled smoothly, and that hosts could move transparently from one switch to another.

I used NetFPGA [11] as the platform to do this. A NetFPGA is a PCI card with a single Xilinx Virtex-II Pro 50 FPGA, four 1Gb/s Ethernet ports, and several banks of fast memory [12], as well as DMA access to the host computer's main memory. The NetFPGA platform was created to enable rapid prototyping of network protocols at

gigabit speeds. Before its inception, no similar platform existed, and researchers had to either rely on slow software prototypes, or create expensive custom hardware. The NetFPGA provided a natural platform for prototyping MOOSE.

1.4 Relevant Courses

I used knowledge gained from the following courses: Advanced System Topics and Digital Communication 1 and 2 (for network concepts), Algorithms I and II (for algorithmic design and implementation principles), Programming in C and C++, Software Design and Software Engineering (for programming design and implementation techniques), ECAD and Computer Design, (for hardware and FPGA ideas), and Unix Tools. The MPhil ACS Network Architecture course was also helpful.

The tripos does not, however, offer any instruction in the practicalities of implementing network protocols. The NetFPGA platform and MOOSE, a research protocol, are also entirely outside the tripos. I learnt about each of these areas of my own accord. I taught myself not only technologies but also language skills, and gained expertise in implementing efficient network applications.

Chapter 2

Preparation

2.1 Research

Research into the development technologies available on NetFPGA was required.

2.1.1 NetFPGA and OpenFlow

Some time was initially spent investigating project development, deployment and testing on NetFPGAs [11]. Projects on NetFPGAs are programmed using Verilog. Though a huge improvement on pure hardware implementation, and much more performant than a software implementation, development is a slow, intricate and error-prone process, relying heavily on simulation for low-level testing, and Perl scripts to generate sample packets for higher-level unit testing.

OpenFlow [13] has been conceived as a way for researchers to experiment with new network protocols. It is a specification which switches can support to provide a simple mechanism for experimentation with network protocols. OpenFlow allows protocol logic to be implemented in software, but run on native hardware.

OpenFlow operates by keeping a table in memory of *flows*, with associated *actions*. A flow is defined in terms of some fields from data-link-layer, network-layer and transport-layer headers (see Figure 1.1). These fields are listed in Table 2.1.

A *flow* is defined as a set of exact matches on any number of these fields, and/or partial matches on IP Source and Destination Addresses. An *action* is defined as any combination of:

- Send out packet on a specific physical port.
- Rewrite any header fields.

- Drop packet.

When an OpenFlow switch receives a packet, its headers are inspected. If the headers match an existing flow, the associated action is taken. If the packet's headers don't match any flow, the packet is sent to a software controller. The software controller then determines the proper action to be taken for the packet, and may add this action to the hardware flow-table to be used for this packet, for future matching packets, or for both.

Some switches made by HP, Cisco, Juniper and NEC support OpenFlow, and any Linux computer with multiple network interfaces can support OpenFlow. Hardware fulfilling the OpenFlow specification has been written in Verilog and can be loaded onto NetFPGAs, so these too can be used as OpenFlow-compliant switches.

There are many benefits to using OpenFlow on NetFPGA, rather than synthesising a custom Verilog switch. OpenFlow allows for rapid prototyping without having to worry about tedious and irrelevant low-level hardware details, such as managing buffers of incoming packets while others are being processed. Much better testing frameworks and debugging tools exist for writing software than Verilog, and statistics can be much more easily recorded, as they can be saved in files on the controller computer rather than having to manually synthesise a data-collection interface on the hardware switch. The benefits of many people working on the existing Verilog implementation of OpenFlow on NetFPGA also make the core hardware-interface code more reliable. NetFPGAs only have four network ports, few compared with the standard 24- or 48-port switches that are ubiquitous in commercial deployment. Using OpenFlow allows a much larger range of hardware to be used in experiments than writing very platform-specific Verilog, overcoming some of NetFPGA's limitations.

OpenFlow also allows for much easier sharing and collaboration with the research community. Researchers are often more comfortable with the languages supporting OpenFlow controller development (C++ and Python), than Verilog, which permits little

Layer	**Protocol**	**Field**	**Size**
Physical	Physical	Physical port	16 bits (with special values)
Data Link	Ethernet	Source Address	48 bits
Data Link	Ethernet	Destination Address	48 bits
Data Link	Ethernet	Type/Length	16 bits
Data Link	Ethernet	VLAN	16 bits
Network	IP	Source Address	32 bits
Network	IP	Destination Address	32 bits
Network	IP	Protocol	8 bits
Transport	TCP/UDP	Source Port	16 bits
Transport	TCP/UDP	Destination Port	16 bits

Table 2.1: Headers used for OpenFlow matches

abstraction over irrelevant concerns and makes it hard to focus on the important features being demonstrated by a prototype. Also, any PC can run a supplied OpenFlow prototype, whereas only 150 institutions worldwide actually have NetFPGA cards to run a Verilog implementation, and using them is a rare specialism.

Using OpenFlow adds only a small amount of latency to packet processing. This performance hit (from sending the packet to the software controller) only affects the first packet of each flow, as subsequent packets are handled with the fast hardware lookup table. For the metrics being considered: size of forwarding table, optimality of routing, ease of host migration, and functionality of conflict resolution, the impact of this small one-off latency is minimal and insignificant. The benefits of using OpenFlow far outweigh the advantage of writing the whole switch in Verilog (a small decrease in latency on a few packets).

As outlined in Section 1.4, MOOSE, hardware implementation and network practicalities are not included in the tripos. NetFPGA and OpenFlow are also topics outside of the tripos which I had to learn about entirely on my own.

2.2 Modifications to Project Proposal

I decided to use OpenFlow to create my MOOSE switch on NetFPGA, rather than using Verilog. This combined a hardware and software implementation, taking the advantages of both, with the only limitation being a small amount of latency.

2.2.1 Decision to Use OpenFlow

Using OpenFlow required modification to my project proposal. Originally I was going to gather information relating to end-to-end latency of transmission. This measurement, however, would no longer give fair comparisons because the software implementation would be slower than custom hardware. As well as this, OpenFlow only supports exact matches on whole Ethernet addresses. OpenFlow rules must therefore be generated between pairs of hosts, rather than simply keeping a look-up table of swiches and ports as would be expected in a purely hardware switch. Using OpenFlow, all of the other metrics can still be gathered and the added benefits outlined above are also introduced.

I also decided early on that rather than simply sending lots of arbitrary IP traffic through large networks in arbitrary topologies, as originally envisaged, gathering specifically targetted benchmarks, and comparing these with standard Ethernet performance would be more insightful. These benchmarks are described in Section 4.1.

2.3 Learning

Before programming could begin, I had to learn more about the systems I was going to be working with. This involved setting up a working development platform and learning a new programming language, Python.

2.3.1 Compiling NetFPGA, OpenFlow and NOX

Very few people use OpenFlow with NetFPGAs because both are very specialised tools, and not many NetFPGAs exist. Unfortunately, this means that developments in one are occasionally not entirely compatible with the other. Several weeks were spent identifying a version of OpenFlow compatible with the NetFPGA hardware, tracking down undocumented dependencies of OpenFlow, and creating patches to compile it on CentOS, the only operating system which NetFPGAs support. These patches were contributed back to the community. I also learnt how to use the distributed version control system *git*, to allow me to work with the OpenFlow repository.

NOX is a library for creating OpenFlow controllers in C++ or Python in order to handle packets for which no flow exists in hardware flow-tables. It is by far the most commonly used and stable project for this purpose, and was a natural choice for this project. Unfortunately, a suitable version of NOX then had to be identified which would interoperate with the identified version of OpenFlow. Again, undocumented dependencies were tracked down and patches created to enable compilation.

NOX's website claims that "Unfortunately there currently isn't extensive API documentation so you should expect to get comfortable with the source." [14]. I became comfortable with the source, including extensive use of Boost C++ paradigms (e.g. `boost::function` and `boost::bind`) with which I was not familiar, through lots of investigation and many experiments.

2.3.2 Learning and Evaluating Python

The NOX development team recommend:

> NOX components can be written in either C++ or Python (or both). At the time of this writing, the Python API to the network is more mature and therefore more friendly for new NOX developers. We recommend that unless the component being developed has serious performance requirements, that developers start with Python. [14]

Because the only traffic affected by the chosen language is the first packet from each host, the work done there is minimal, and the only significant effects of using Python

should be the number of flows that can be processed concurrently, I followed the advice of the NOX development team and used Python. I spent some time learning this new language, very different in nature from any programming language I had used before with its dynamic typing, and exploring NOX's Python APIs from NOX's source code. I created a MOOSE switch in Python which could perform all of my core project requirements (see Appendix C § *Work to do—Have attached hosts*).

I ran some performance tests on this Python controller to verify that my approach was suitable and found that rates of around 100Mb/s were achieved, far below the expected 1Gb/s. Further investigation showed that the sample NOX Ethernet switch written in Python also operated at about 100Mb/s, whereas the C++ version operated at about 1Gb/s. The approach of using Python to write the switch should certainly not have had such a significant performance impact, and the slowness was caused by a bug in NOX encountered when crossing language boundaries between Python and C++. Rather than track down the cause of this inherent slowness across language boundaries in the NOX library, I opted to re-write my MOOSE controller in C++. This required learning a new API (as using NOX in C++ and Python are somewhat different), and rewriting some work in C++ that I had already written in Python, but seemed the best approach. I re-wrote an equivilent controller in C++ and found that it was switching at 1Gb/s as expected. This speed was comparable to the rate achievable by the NetFPGA hardware running with an Ethernet controller.

I give no further details of the Python implementation, though the state of the code when I abandoned this approach is included in my source code submission for reference. All work referred to henceforth is in relation to the C++ controller.

2.4 Strategies for success

I made decisions about how I would implement my project to ensure its success.

2.4.1 Test-Driven Development

I decided to adopt a test-driven development paradigm to ensure that all code worked reliably. Unit tests would be written for every method before writing the method's implementation. This means that time has to be spent considering the desired effects of code and distinguishing between the separate classes of input to the methods and how they should be handled. It also forces the independence of modules, and encourages modules to be as state-free as possible, as this makes testing easier. Occasionally, when some aspects of code are refactored, or the implementation behind their interface is changed (as described in Section 3.2.1), this means that there is a good suite of tests written in advance which would show whether the refactoring was correct and complete.

This also means that an exhaustive test suite is built up as code is written, rather than allowing testing to be put off until the end of the project (where memory of the code is hazy), and possibly even neglecting it due to time pressure. Having an exhaustive test suite makes debugging much easier, because the debugging is done as the code is written rather than when trying to explain obscure emergent properties of the whole complex system.

2.4.2 Source Control

I frequently committed my code to a Subversion repository held on the PWF, of which I had a backup in London. This ensured that I could look through historical changes to my code, and revert changes if needed.

Chapter 3

Implementation

3.1 Extending MOOSE

To implement a fully functioning switch, I needed to better formalise some details of the MOOSE protocol that had only been outlined in the existing paper [3]. I developed these myself, and describe them in this section. I have also discussed them with the MOOSE team, leading to their inclusion in an as-yet unpublished paper [15].

3.1.1 Status And Management Interface

I specified Status and Management Interface (SAMI), the format of inter-switch messages, and intended behaviours upon receiving them, for MOOSE. I briefly describe these messages below, but leave byte-level format details for Appendix A. My MOOSE switch implementation supports all of these messages.

Conflict Resolution

So that switches don't have to be manually allocated addresses by a network administrator, and to prevent requiring universally unique switch addresses (artificially restricting the number of switches which could be made, and any future additional hierarchies in MOOSE addresses), it is desirable for switches to choose their own addresses when joining a network. Switches could choose the same addresses, introducing the potential for conflicting switch addresses which must be able to be autonomously resolved. I devised a scheme, fully detailed in [15, §4.2], whereby all switches listen out for packets which appear from known switches on ports other than where they have been seen before. If they find such a packet, they check whether the previously known switch is online. If it is online, the switch instructs the newly found switch to change its address. If the

previously known switch is not online, the switch updates its view of the network to reflect where the discovered switch is now, and notifies its neighbours of this. Controls are also in place to prevent denial of service attacks.

This protocol requires the ability to send echo requests to switches to see whether they are online, to respond to such requests, and to instruct switches to change their address. I specified SAMI messages for these purposes.

Host Mobility

In basic operation, when a host moves from one switch to another and the host is allocated a new MOOSE address, any traffic sent to the host's old MOOSE address will be lost. The old address will be used for a short time by those who had in the past communicated with the host, as it will be present in their ARP caches. Scott *et al.* [3, §IV.E] propose that the new switch could notify other switches of the Ethernet addresses of hosts as they connect to it, by multicast, so that the old switch can forward traffic destined for the host's old MOOSE address to its new MOOSE address. Again, I specified a SAMI message to notify switches of new hosts' attachment.

3.1.2 Variable Length Prefixes

Scott *et al.* [3] define MOOSE switch identifiers as exactly three bytes long in their initial research paper (which is not a full specification). This unnecessarily limits the number of switches in any network, and increases the likelihood of conflicting identifiers in a network. This may pose constraints on practical deployment. I devised an addressing scheme which allows for variable length switch identifiers, significantly raising this limit.

The scheme operates similarly to class-based IP addressing [16, §3.2]. The six byte MOOSE address is split up into a switch identifier and a host identifier. The bitwise negation of the most significant two bits of the address indicate how many of the bytes form the switch identifier (the remaining bytes being the host identifier). This gives four classes of switch identifier — 1, 2, 3 and 4 bytes, giving approximately a 32-fold increase in available switch identifiers. This also restricts switch prefixes to 32-bits, which is a common word-size for content addressable memory as used for IP addresses, potentially allowing existing IP routers to be re-purposed as MOOSE switches with a simple firmware upgrade, and allowing for existing manufacturing of CAM to be exploited. This address format is well suited for hardware because the length of the switch identifier is directly encoded in a predictable way in the address.

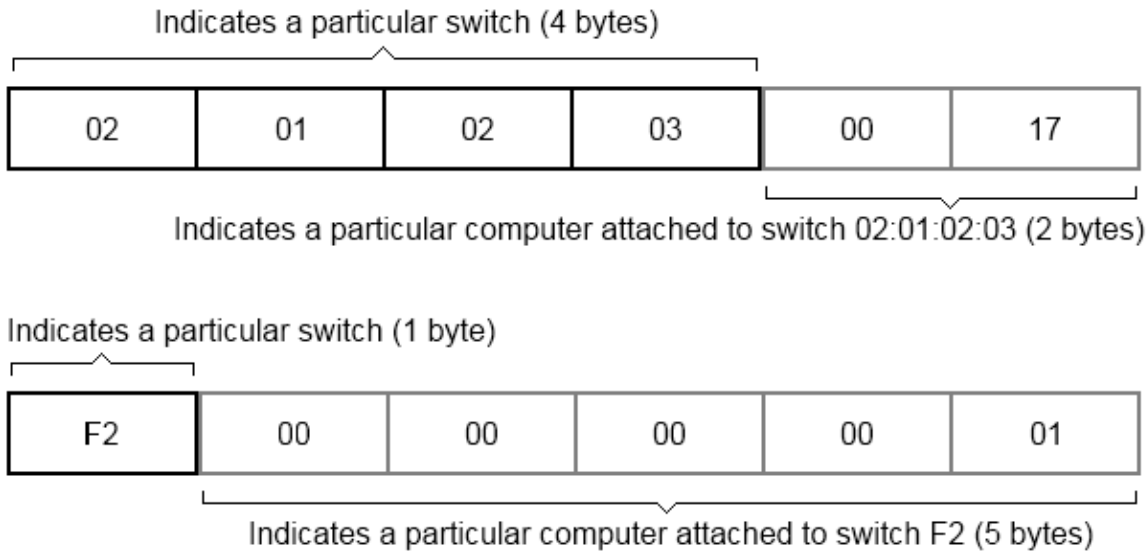


Figure 3.1: Examples of address classes, maximum length prefix (above) and minimum length prefix (below)

3.1.3 Routing Protocol

The expected number of switches in a MOOSE network is expected to be small relative to the number of hosts, and of the scale not to require much manual management or routing policy in the network. Accordingly, I choose to use a link-state routing protocol with MOOSE. I adapted Open Shortest Path First [10], the Internet link state protocol, to use MOOSE addresses, and removed unrequired complexities of the protocol. I call this protocol Open Shortest Path First for MOOSE (OSPFM).

3.2 Approach to Implementation

I detail the guiding principles of my implementation, and give an outline of my code.

3.2.1 Modularity

A modular design form was used so that code could be reused where possible. This also allowed unit-testing without having to search for high-level properties of the actions of switches when exposed to actual network traffic in order to verify behaviour.

Separation of Concerns

The interface with OpenFlow/NOX was kept separate from the MOOSE-specific logic by keeping all OpenFlow-related code in its own class, `NoxMooseSwitch`. `NoxMooseSwitch` has a single `MooseSwitchImplManager` member, which contains all of the MOOSE logic. When a packet is received, NOX passes an OpenFlow event to the `NoxMooseSwitch` containing a copy of the packet. The `NoxMooseSwitch` works out the correct action to perform on the packet by consulting the `MooseSwitchImplManager`. The `NoxMooseSwitch` then sends the proper OpenFlow messages back to the hardware which performs that action.

This separation of concerns meant that development could be done without needing the NetFPGA machines until late in the process. When the machines were unavailable due other commitments, work could still be done because most of the logic being performed is independent of the actual NetFPGA-OpenFlow interface. These tests also provided demonstrations of the correctness of the building blocks of the project, giving a firm foundation for the correctness of the whole project.

The OSPFM routing protocol behaviour was also kept separate from the other components. Dijkstra's algorithm [17] and OSPFM were implemented in their own classes, and so could be tested entirely separately from the rest of MOOSE.

Key Modules

The key abstractions for the MOOSE-related logic are a `MooseSwitchImpl` class which manages the locally attached hosts for a switch (with a single switch-identifier prefix), and a `MooseSwitchImplManager` class which deals with inter-switch communication. A `MooseSwitchImplManager` has one or more `MooseSwitchImpls` — possibly more than one if a switch is in the process of migrating from one prefix to another, e.g. if it had exhausted its host address space. It also has an `OSPFMModule`, which keeps track of other switches and provides a forwarding table. It delegates to a particular `MooseSwitchImpl` where relevant, i.e. when a packet is sent to or from a directly attached host. This provides for good separation of concerns, avoiding entangling different address rewrites and port lookups in the same functions. It also means that a particular `MooseSwitchImpl` needn't worry about changing its prefix, or keeping any state with respect to more than just its own prefix. This additionally meant that much of a `MooseSwitchImpl`, for instance its prefix, could be kept immutable. This immutability provides compile-time guarantees that programmer assumptions about unchanging variables are upheld, giving more confidence in the program, and allowing for some optimisations which would not otherwise be possible.

Interface was separated from implementation for all major components of the project. This gave obvious interfaces to test, and ensured that modules could not interfere with the internal state of other modules (for instance, data fields were kept `private` within

classes, and access was only allowed through a controlling method). This also enabled easy replacement of implementation while keeping interfaces unchanged, and refactoring of code.

Advantages of this Approach

An example of useful separation of interface from implementation is the `AttachedHosts` class. This class keeps a mapping between locally attached Ethernet addresses and their associated MOOSE addresses. MOOSE addresses need to be looked up from Ethernet addresses when a packet is received from that host, but a lookup in the opposite direction is needed when a packet destined for that host is received. I used a single bidirectional map to store this mapping because it was memory efficient, and ensured that changes in one direction were consistently enforced in the other direction. Originally, to get something working and usable, I wrote an implementation which kept two `std::maps`, one from MOOSE addresses to Ethernet addresses and the other from Ethernet addresses to MOOSE addresses. Adding, removing, or modifying entries to these maps was closely controlled in the methods which other classes could call, and the only actions supported were to atomically add, remove, update, or look up entries, which affected both maps. I then replaced this simple implementation with an implementation which used a single bidirectional map. This allowed me to work out the operations required and logic behind them, and then learn the more complex bidirectional map interface when this was established. I also already had an exhaustive test suite for `AttachedHosts` and so had confidence that when I replaced the two maps implementation with the one bidirectional map implementation, my changes would work consistently.

3.2.2 Platforms of Development

To learn about the operation of NOX and OpenFlow without getting confused by new platforms, a VirtualBox image of a Debian virtual machine was used to experiment with NOX. This image contained within it a full development environment to use NOX and several QEMU virtual machines to act as hosts connected to the virtual NOX switch. This virtual machine image was distributed after use in a hands-on tutorial at SIGMETRICS 2009 [18]. First, an Ethernet switch was built. This was expanded to a standalone MOOSE switch which could have several hosts attached, but not interoperate with other switches as this environment was not set up to support multiple switches communicating with each other.

Then, having some experience with NOX, a development environment was set up on the actual NetFPGAs to be used, and the basic MOOSE switch was deployed and tested.

Using virtual machines to learn about NOX gave a very easy debugging environment and allowed changes to be quickly tried, enabling a rapid development cycle. Irrelevant

issues, such as concerns over how hardware worked, could be ignored, and concentration given to learning about NOX. This was a very helpful stage.

3.3 Main Areas of Code

The code written falls under four categories, which I shall describe separately.

3.3.1 Novel algorithms, Data Structures and Types

I wrote the code to implement my novel conflict resolution algorithm, as well as some useful data structures and types:

Conflict When it is detected that two switches have the same address, this conflict must be resolved using the conflict resolution algorithm which I devised (Section 3.1.1). This class autonomously holds all of the state (addresses, ports, etc.) relating to a conflict, and sends packets when the algorithm mandates that it should.

ConflictListener As part of the conflict resolution algorithm, a switch must keep state about when it was last notified that it was in conflict. This class stores that state.

OSPFM Several constants and classes were defined to help with the OSPFM routing protocol. The `OSPFMNeighbour` class keeps state about particular neighbouring switches of the current switch, `OSPFMPortList` keeps a list of which neighbouring switches of the current switch are connected to which ports, and `OSPFMLinkStateForSingleSwitch` keeps state about the neighbouring switches of other switches.

AttachedHost This simple type embodies all information about each locally attached host, initially its Ethernet address and attachment port, but potentially extendable to include things like last received packet time.

I also wrote code to rewrite ARP queries as they were received. This involved getting a good knowledge of the types of ARP requests and replies used, and carefully implementing address rewriting based on the nature of the message.

In storing some custom data-types, it was incredibly important that network and host byte order were used correctly.

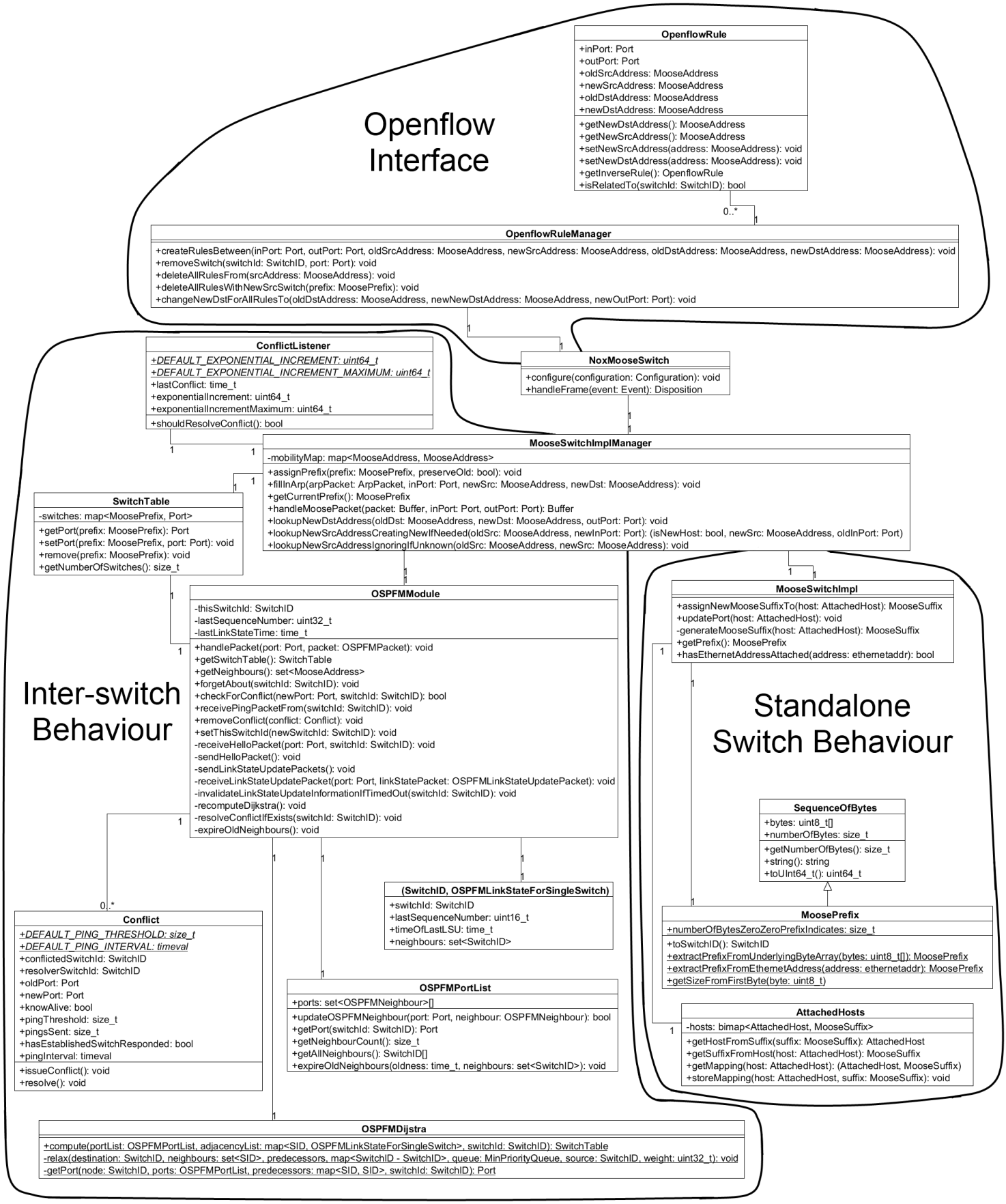


Figure 3.2: Class diagram outlining code. Some features omitted for clarity and brevity.

3.3.2 Implementations of Existing Algorithms and Data Structures

I implemented several existing algorithms and data structures as part of the routing protocol for MOOSE:

OSPFMDijkstra Here I implemented Dijkstra's algorithm [17] for efficiently finding the shortest paths to other switches and storing them in the form of a `SwitchTable`.

PriorityQueue Dijkstra's algorithm requires a priority queue [19] for storage of nodes, and relies on being able to decrease the distances to nodes after adding them to the queue. Unfortunately, the C++ STL implementation of `std::priority_queue` does not allow decreasing of distances. Accordingly, I implemented my own `PriorityQueue` class using a heap.

3.3.3 Wrappers and Abstractions

I wrote several classes to provide useful interfaces to existing data structures, and to allow communication with hardware using the OpenFlow protocol:

OpenflowRule To keep track of what *rules* and *actions* have been put in the hardware tables of the NetFPGA, so that they can be modified or deleted in the future, a convenient abstraction for these rules was created. Rules were stripped down to the minimum relevant for my application — source and destination address to match to incoming packets, source and destination address to put on outgoing packets, and relevant ports. This helped overcome some of the limitations of OpenFlow described in Section 4.3.

OpenflowRuleManager This wrapper around a `std::set` keeps track of the `OpenflowRules`, taking care of generating, modifying and deleting rules based on meaningful events such as a new host being attached, or another switch moving from one port to another. This wrapper class enforced consistency of rules, e.g. if a switch moved from one port to another, all rules associated with that switch would atomically be updated.

OpenflowRuleBridge To send OpenFlow messages to the NetFPGA so that it would modify its hardware tables, instructions to add, modify or remove `OpenflowRules` need to be translated into OpenFlow packets. Utility functions were written to create these packets.

SwitchTable The `SwitchTable` interface facilitates the lookup and storage of which port other switches are connected to. The wrapper keeps the implementation loosely coupled from the interface, and allows the return of a special *no port*

value if no port is stored for a switch, rather than relying on the caller checking for the case of a missing entry and having to know the correct value to substitute in. This wrapper around a `std::map` achieved this.

AttachedHosts To store and look up details about locally attached hosts (Ethernet address, MOOSE address, port), a similar interface to that of `SwitchTable` was used. This is a wrapper around a `boost::bimap`, and its development was described in Section 3.2.1.

3.3.4 Glue

I wrote some classes which tied together the other written code, constructed and kept the required objects, and enforced relationships between objects:

OSPFMModule This class keeps state relating to a switch as outlined in the `OSPFM` data structures. It has a public interface to handle `OSPFMPackets`, to (if required compute using `OSPFMDijkstra`, and) return a `SwitchTable`, and to return a list of neighbouring switches.

MooseSwitchImpl This class is used to encapsulate the state of a single prefix of a MOOSE switch. It keeps track of attached hosts using an `AttachedHosts` member, allocates new suffixes to newly attached hosts, and gives an interface to look up information about locally attached hosts.

MooseSwitchImplManager This class is used to encapsulate the state of an entire MOOSE switch, which may have multiple prefixes. It wires together an `OSPFMModule` (and accordingly, a `SwitchTable`), a list of `Conflicts` managed by the switch, a `ConflictListener` for the switch, mobility rules for formerly attached hosts, and some number of `MooseSwitchImpls`. It provides an interface for translating between MOOSE and Ethernet addresses, looking up ports of switches and hosts, filling in ARP packets, and handling inter-switch packets.

NoxMooseSwitch This class acts as a bridge between OpenFlow and a `MooseSwitchImplManager`. It receives packets, gets relevant information from its `MooseSwitchImplManager`, keeps and updates an `OpenflowRuleManager`, and issues OpenFlow commands to the NetFPGA.

3.4 Testing

Testing was important to ensure my implementation worked as specified. I detail the forms of testing used, and frameworks exploited to get to high confidence in the correctness of the written code, and thus the results gathered with it.

3.4.1 Testing Framework

Having adopted a test-driven development strategy, choosing a testing framework was important. The available options, none of which I had used before, were surveyed. CxxTest[1] was chosen because its simple interface enabled rapid prototyping. C++ has no introspection capabilities, which means that most C++ testing frameworks require manual registration of test cases, and use of ugly macros in defining tests. CxxTest uses Python to parse tests, rather than itself being written in C++, and so has none of these restrictions inherent in C++. This removes complexity and adds flexibility to the testing process. The cruft required to write a test suite is minimal — a class is simply defined which extends the `CxxTest::TestSuite` class, and any methods within whose names start with the word `test` are automatically registered as tests. The framework is distributed as a set of header files, so no other compilation is required. Assertions are handled very well — again, the parsing stage means that if an equality assertion fails, both the original assertion *and* the values present when checking the assertion can be outputted, adding convenience and aiding debugging.

3.4.2 Testing Strategy

Other than for the `NoxMooseSwitch` class, unit tests were written for all methods before writing the method itself. The different classes of input to the method were determined, and tests written to exercise the methods with each of those classes of input, verifying the expected behaviour. More integration tests were then written for the `MooseSwitchImplManager` class, which exercised all of the relevant subroutines required to process packets.

The `NoxMooseSwitch` class was very hard to test in an automated fashion because it relied on being run in the full NOX event framework, which would be complicated to interact with in a simple enough manner to write reliable and useful tests. My modular design meant that explicit unit testing of this class could be avoided. Periodically, I set up test networks of NetFPGAs and generated real traffic using `ping` and `wget`. I verified that packets were being sent properly, with the correct addresses being rewritten, and to the correct hosts using the packet capture software Wireshark. I manually inspected the sent and received packets using Wireshark, and verified (and, where suitable, filled in) the contents of hosts' ARP tables with the Unix `ifconfig` and `arp` commands. I also inspected OpenFlow control messages sent to the NetFPGA cards, again using Wireshark.

[1] http://cxxtest.tigris.org

3.4.3 Mocking

I wrote my own mocking system because the amount of mocking required was small and restricted only to methods which were passed callback functions (a small amount of conflict resolution and OSPFM code). Writing my own system took less time than would have been needed to identify a suitable existing framework and learn how to use it. This code records calls that are made to callbacks, and the arguments passed, and makes it easy to verify that the correct calls to these callbacks were made, and to fail tests if any incorrect calls were made.

3.4.4 Valgrind

When tests were run, they were run in a `valgrind` memory checker environment. `Valgrind` keeps track of every memory read and write, ensuring unassigned memory is never read from or written to, and that all memory is correctly freed after it is allocated. This again helped to verify the code, and ensured that it would not encounter unforseen errors at runtime, because my tests exercised all of the non-trivial code.

Chapter 4

Evaluation

4.1 Experiments and Results

Data was gathered to compare the behaviour of MOOSE with that of Ethernet, and to evaluate the effectiveness of host mobility and conflict resolution in MOOSE.

Unless otherwise noted, all experiments used computers with NetFPGA cards as switches. These computers have two Ethernet interfaces. The two interfaces were assigned IP addresses on separate subnets, and one computer was never asked to communicate with its other interface, to ensure that packets were always being sent through the network.

The switches used were statically assigned the (hexadecimal) prefixes: `02:01:01:01`, `02:02:02:02`, ..., `02:0D:0D:0D`. In the early stages of testing, the hardware in the switch assigned `02:03:03:03` was found to be faulty, so was removed from the experiments before data was gathered. 12 switches were therefore used.

Before each experiment, the ARP caches of all hosts were cleared, so that no stale addresses could be used.

In all topology diagrams, ellipses indicate switches and rectangles indicate hosts.

4.1.1 Forwarding Table Size

In the common case, MOOSE offers to reduce the number of entries in switches' forwarding tables, allowing more entries in the same amount of memory. Also, the entries being stored are smaller, so again more entries can be kept.

Let H be the number of hosts present on the network. This will vary between around 1 and 16,000 on existing networks, though could be much larger. Multiple interfaces in the

same device would each appear as an individual host to the network. A recent trend has been towards devices having many interfaces, for instance most modern mobile phones now have 4 or more modems (GPRS, 3G, HSDPA, etc.). Virtualisation techniques also add extra hosts to the network. One of the objectives of the design of MOOSE was to deal with this case of dense networks, with many more hosts than switches.

Let A be the average number of hosts attached to a single switch. Switches tend to have between 4 and 96 ports, most commonly 24 or 48.

Let S be the number of switches present on the network. This will again vary, but values between $\frac{1}{96}H$ and $\frac{1}{8}H$ are expected.

Ethernet switches require $O(H)$ 6-byte entries in their forwarding table, because they keep track of every Ethernet address they see.

MOOSE switches are claimed by Scott *et al.* [3, §V.E] to require on average $O(A)$ entries of at most 12 bytes[1], and $O(S)$ entries of at most 4 bytes[2], aggregating to $O(A + S)$ entries.

Though the expected number of entries is different, $O(H)$ is normally expected to be greater than $O(A + S)$. An experiment was performed with two constrasting configurations, one where MOOSE would be expected to *decrease* the size of the forwarding table, and one where MOOSE would be expected to *increase* the size of the forwarding table. My MOOSE switch periodically recorded the contents of its forwarding table. I adapted the standard OpenFlow Ethernet switch (included in the OpenFlow source) so that it would record the contents of its forwarding table whenever it changed.

Best Case for MOOSE

The best case for MOOSE should be where the number of hosts on the network is large in comparison to the number of switches, as is expected in commercial deployment. To test this case, a line of switches was set up, each connected to the previous and next switch in the line. Each switch had exactly two hosts attached — a restriction due to NetFPGAs only having 4 ports, the true best case for MOOSE would have many hosts per switch. Every host sent packets to every other host in the network[3]. Every switch therefore saw at least one packet from every host, and accordingly at least one packet via each other switch on the network.

[1] 6 bytes for the Ethernet address of the attached host, and at most 6 bytes for the MOOSE address of the attached host, though only the suffix needs storing, so this could be reduced to between 8 and 10 bytes per host

[2] For simplicity and consistency in memory, each entry would likely be padded to a full 4-byte entry, though some prefixes are shorter than four bytes. In memory-constrained environments, this could be reduced.

[3] The command `ping -c 5 <destination IP>` was used

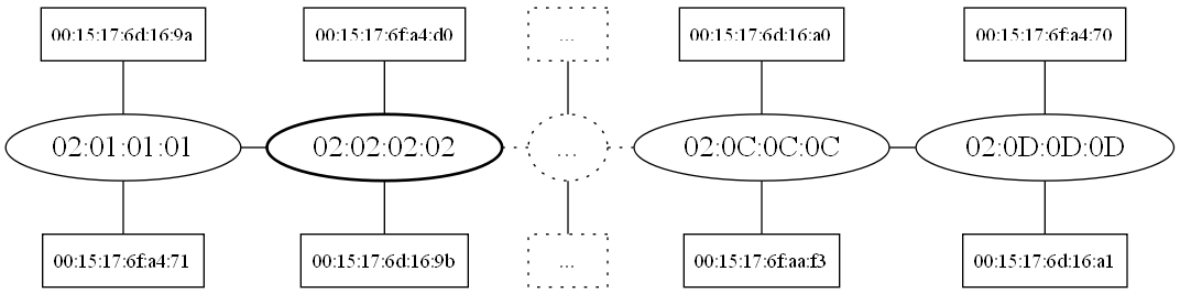


Figure 4.1: Network topology of first MOOSE forwarding table size experiment, 12 switches in total. Ellipses indicate switches, boxes indicate hosts

This first configuration was run using the MOOSE switches. All of the hosts were then reset, and the same configuration was run using the Ethernet switches. The comparison of the switch table size of switch `02:02:02:02` is illustrated below.

```
Dumping table:
00:15:17:6f:a4:71 - 2
00:15:17:6f:a4:70 - 3
00:15:17:6f:aa:f3 - 3
00:15:17:6f:aa:f2 - 3
00:15:17:6f:a8:0d - 3
00:15:17:6f:a8:0c - 3
00:15:17:6f:ae:fd - 3
00:15:17:6f:ae:fc - 3
00:15:17:6f:a8:13 - 3
00:15:17:6f:a8:12 - 3
00:15:17:6f:9f:64 - 3
00:15:17:6f:9f:65 - 3
00:15:17:6f:a4:d1 - 3
00:15:17:6d:16:9b - 1
00:15:17:6f:a4:d0 - 4
00:15:17:6d:16:9a - 2
00:15:17:6f:ae:43 - 3
00:15:17:6f:ae:42 - 3
00:15:17:6d:16:a1 - 3
00:15:17:6d:16:a0 - 3
00:15:17:6f:aa:31 - 3
00:15:17:6f:aa:30 - 3
00:15:17:6f:a6:53 - 3
00:15:17:6f:a6:52 - 3
Total: 24 hosts
```

(a) Ethernet

```
Dumping tables:
Switches:
02:01:01:01 - 2
02:04:04:04 - 3
02:05:05:05 - 3
02:06:06:06 - 3
02:07:07:07 - 3
02:08:08:08 - 3
02:09:09:09 - 3
02:0A:0A:0A - 3
02:0B:0B:0B - 3
02:0C:0C:0C - 3
02:0D:0D:0D - 3
Hosts:
00:15:17:6d:16:9b - 1
00:15:17:6f:a4:d0 - 4
Total: 11 switches, 2 hosts
```

(b) MOOSE

Figure 4.2: Forwarding tables of Ethernet and MOOSE switches in first forwarding table size experiment

The experiment was run several times, and each time every Ethernet switch's forwarding

table contained 24 6-byte entries, one for each host on the network, agreeing with the expected H entries. Every MOOSE switch's forwarding table contained 11 4-byte entries, one for each other switch, and two Ethernet-address-to-MOOSE-address entries, one for each attached host. This agreed with the expected $O(A + S)$ entries. The predicted improvement was observed — per-hosts entries were aggregated into per-switch entries in forwarding tables. Had my prototype switches had more (say, 24) ports, the total amount of memory used for forwarding tables in the network would be a useful comparison, unfortunately this could not be tested using NetFPGAs.

Worst Case for MOOSE

The worst case for MOOSE should be where the number of hosts on the network is small in comparison to the number of switches — a very sparse network with many switches, which is rare in deployment. To test this case, the same line of switches was set up, but with only two hosts attached to switch `02:01:01:01`, and no other hosts attached to any switch. The switch table of the switch with connected hosts, `02:01:01:01`, was monitored, as well as that of one switch with no connected hosts, `02:0D:0D:0D`.

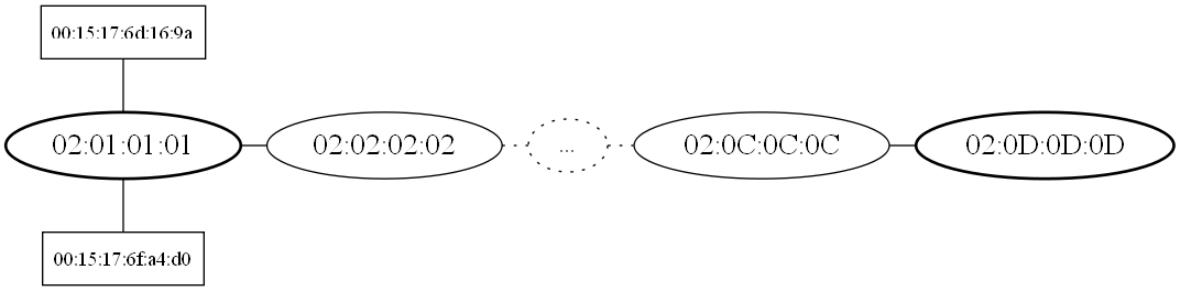


Figure 4.3: Network topology of second MOOSE forwarding table size experiment, 12 switches in total. Ellipses indicate switches, boxes indicate hosts

```
Dumping table:
00:15:17:6d:16:9a - 4
00:15:17:6f:a4:d0 - 1
Total: 2 hosts
```

(a) Switch with attached hosts

```
Dumping table:
00:15:17:6d:16:9a - 2
Total: 1 host
```

(b) Switch without attached hosts

Figure 4.4: Forwarding tables of Ethernet switches

```
Dumping tables:
Switches:
02:02:02:02 - 3
02:04:04:04 - 3
02:05:05:05 - 3
02:06:06:06 - 3
02:07:07:07 - 3
02:08:08:08 - 3
02:09:09:09 - 3
02:0A:0A:0A - 3
02:0B:0B:0B - 3
02:0C:0C:0C - 3
02:0D:0D:0D - 3
Hosts:
00:15:17:6d:16:9a - 4
00:15:17:6f:a4:d0 - 1
Total: 11 switches, 2 hosts
```

(a) Switch with attached hosts

```
Dumping tables:
Switches
02:01:01:01 - 2
02:02:02:02 - 2
02:04:04:04 - 2
02:05:05:05 - 2
02:06:06:06 - 2
02:07:07:07 - 2
02:08:08:08 - 2
02:09:09:09 - 2
02:0A:0A:0A - 2
02:0B:0B:0B - 2
02:0C:0C:0C - 2
Hosts:
Total: 11 switches, 0 hosts
```

(b) Switch without attached hosts

Figure 4.5: Forwarding tables of MOOSE switches in second forwarding table size experiment

After running this configuration, the forwarding table of the MOOSE switch with attached hosts contained 11 4-byte entries, one for each other switch, and two Ethernet-address-to-MOOSE-address entries, one for each attached host. The other switches' forwarding tables each contained 11 4-byte entries, one for each other switch, and no other entries, again $O(A + S)$ entries.

The Ethernet switches had much smaller forwarding tables present. The forwarding table of the Ethernet switch with attached hosts contained two 6-byte entries, one for each attached host, and the other switches' forwarding tables each contained a single 6-byte entry (from the broadcast ARP packet sent as part of the `ping` command). MOOSE brought a significant increase, as expected, to the number of entries in the forwarding table. This kind of topology is expected to be uncommon in practice. MOOSE, however, has been shown to be deterministic, and provide improvement in the expected common case.

4.1.2 Shortest Path Routing

No routing protocol can be used with Ethernet because Ethernet addresses have no hierarchy and no concept of identifiers of switches which could act as way-points when routing. The only way an Ethernet switch has of selecting which port to output a packet to for a given destination is by remembering the last port from which each address was seen as a source address. Because no routing protocol is used, no switch has knowledge of the network topology, and so switches cannot know how to avoid loops. Broadcast

packets which go around a loop will keep being sent around the loop endlessly, clogging up the network, and also generating lots of traffic to all hosts on the loop. This means that either a spanning tree protocol must be used, disabling some links and reducing efficiency, or no spanning tree protocol is used, and any broadcast traffic (of which there is plenty, as its use is required and indeed encouraged in Ethernet) will go around the loop *in perpetuum*. As well as this, packets are received from hosts on the incorrect port, putting corrupt data into switches' forwarding tables and preventing any useful throughput.

Loops

To show that broadcast traffic can traverse a MOOSE network containing loops, a small network of three switches, each with one host attached, was set up. A broadcast packet was sent from one of the hosts, as part of an ARP query from the utility `ping`, and the packets received by the hosts not involved with the `ping` command were captured using Wireshark. These switches were not operating with the inefficient Rapid Spanning Tree Protocol. Though all Ethernet networks today use RSTP to avoid the issues which this experiment shows, MOOSE requires no such protocol. This experiment was somewhat artificial, as it did not compare MOOSE with a realistic Ethernet network deployment, but it clearly illustrates one of the problems of Ethernet that MOOSE solves.

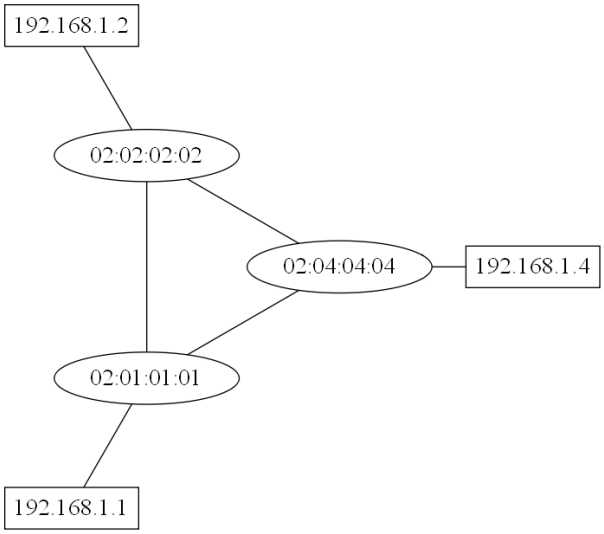


Figure 4.6: Network topology of loop experiment

In the Ethernet network, 10 seconds after `192.168.1.1` sent a broadcast message, over 120,000 copies of the same broadcast packet had been received by each host. Though only one packet was sent by `192.168.1.1`, switch `02:01:01:01` forwarded this packet to switches `02:02:02:02` and `02:04:04:04`. As this was a broadcast packet,

switch `02:02:02:02` sent this packet to `02:04:04:04`, who in turn sent it to *back* to `02:01:01:01`. Ethernet packets have no sequence number or unique identifier to show which switch they originated at, so this switch sent it again to `02:02:02:02`, and it would continue to be sent around the loop *ad infinitum*. This congested the network hugely, eating up all available bandwidth. When switches sent the packets to the other switches, they also sent them to all attached hosts, which caused enough network interrupts in those hosts to cause them to hang.

In the MOOSE network, no such problems were experienced. MOOSE's multicast mechanism between switches ensured that each switch only received each packet once, discarding any further copies before passing them on. Switches distributed copies of the packet directly to each host, rather than blindly flooding them to everyone.

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

Optimal Routes

To show that no links between switches were disabled in a MOOSE loop, where they were in the equivalent Ethernet loop, 12 NetFPGA switches were set up in a loop, with one host connected to one particular switch (`02:01:01:01`). Another host was plugged in to each other switch in turn, and the average round trip time of `ping` packets between the two hosts was calculated[4].

This experiment was then repeated with the same configuration of Cisco Catalyst 3500-48 switches, with the Rapid Spanning Tree Protocol (RSTP) enabled.

The raw data rates are not comparable because the Cisco Ethernet switches only operate at 100Mb/s, slower than the 1Gb/s at which the MOOSE switches on NetFPGAs operate. Unfortunately, 1Gb/s Ethernet switches were unavailable to me. Regardless, the effect of the automatically disabled link in the Ethernet network is very much observable.

[4]Specifically, the command `ping -c 10000 -s 1400 -f <destination IP>` was used

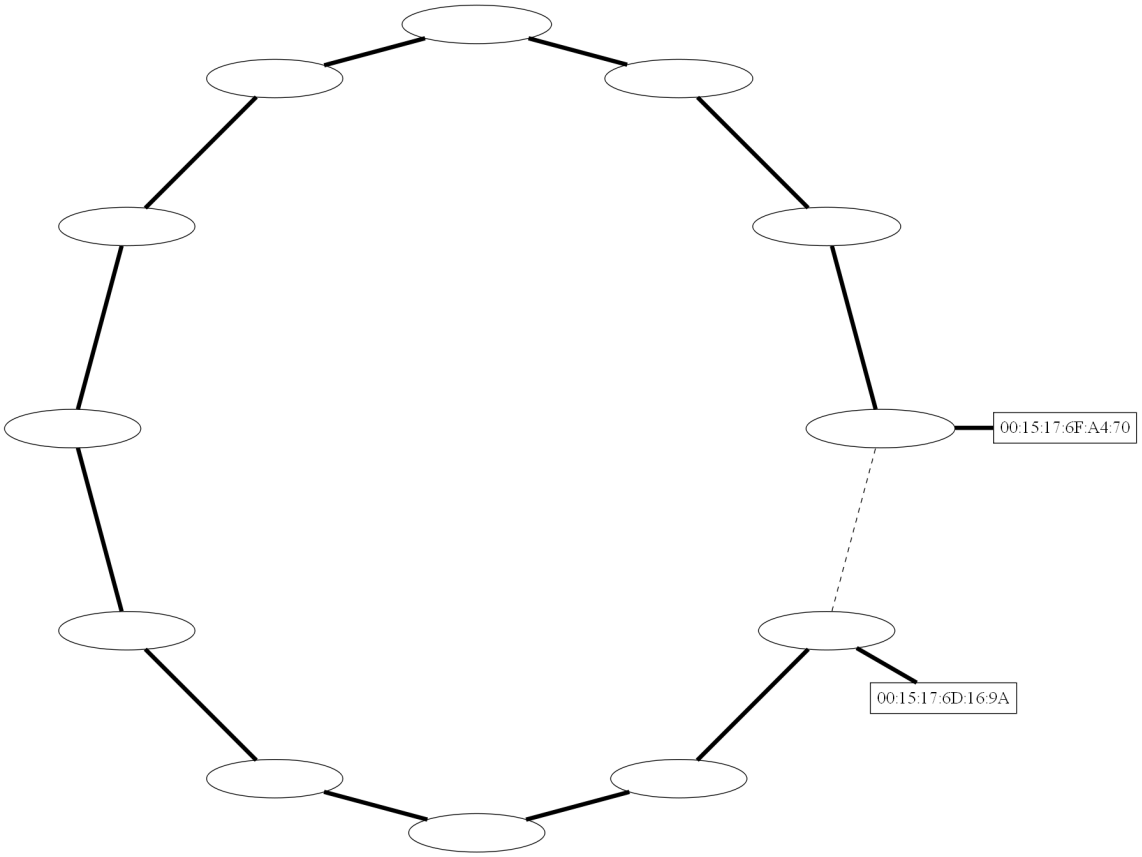


Figure 4.7: Ethernet network topology of optimal routes experiment with RSTP — Bold links show route taken, dashed links were disabled by RSTP

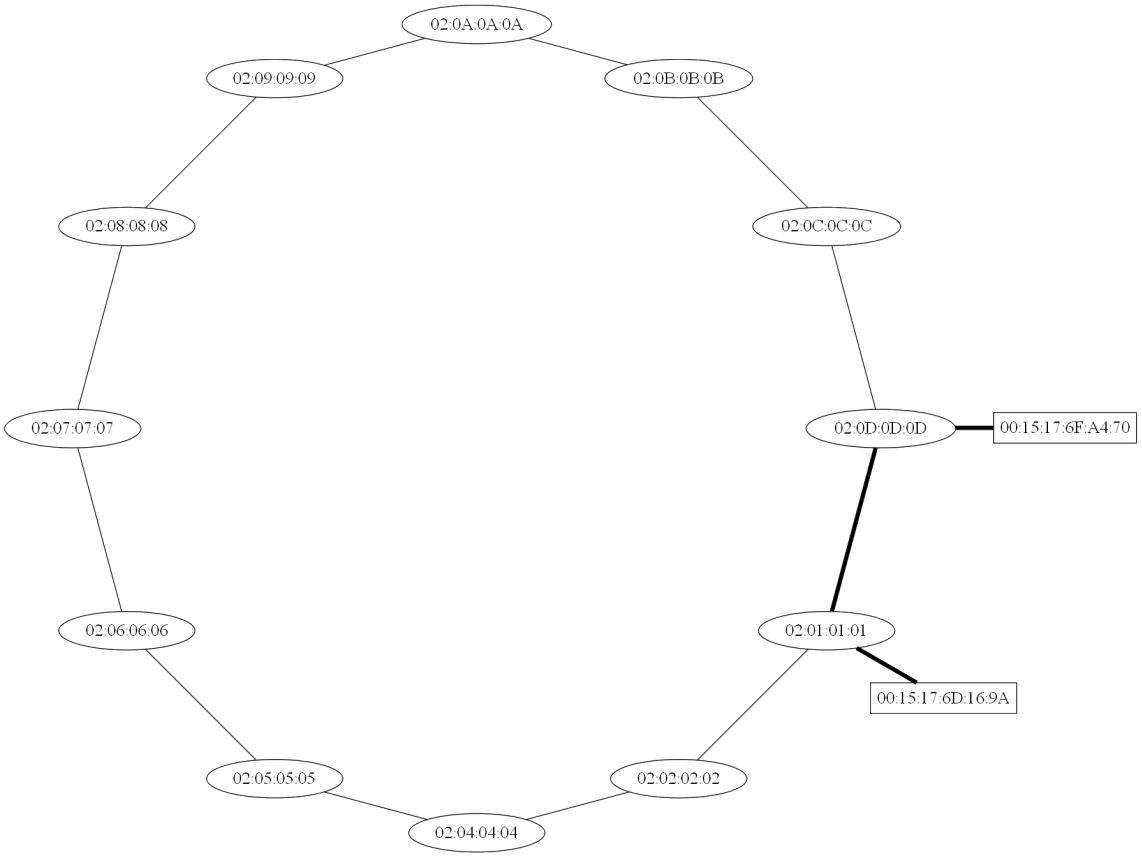
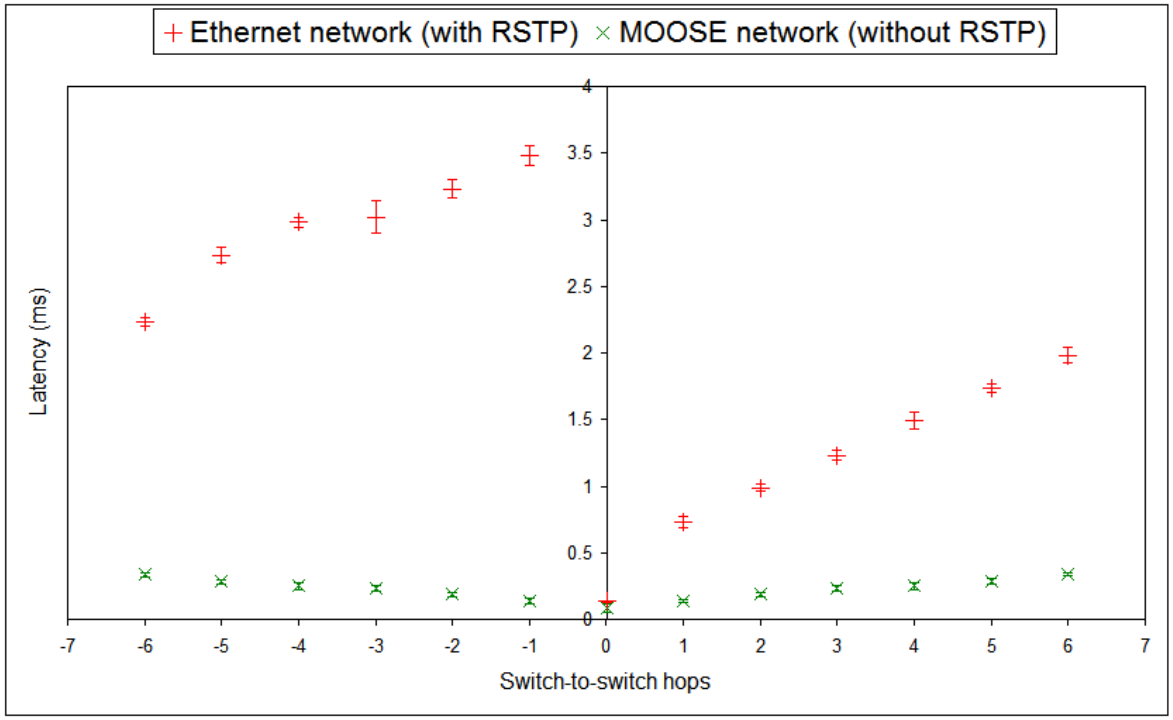
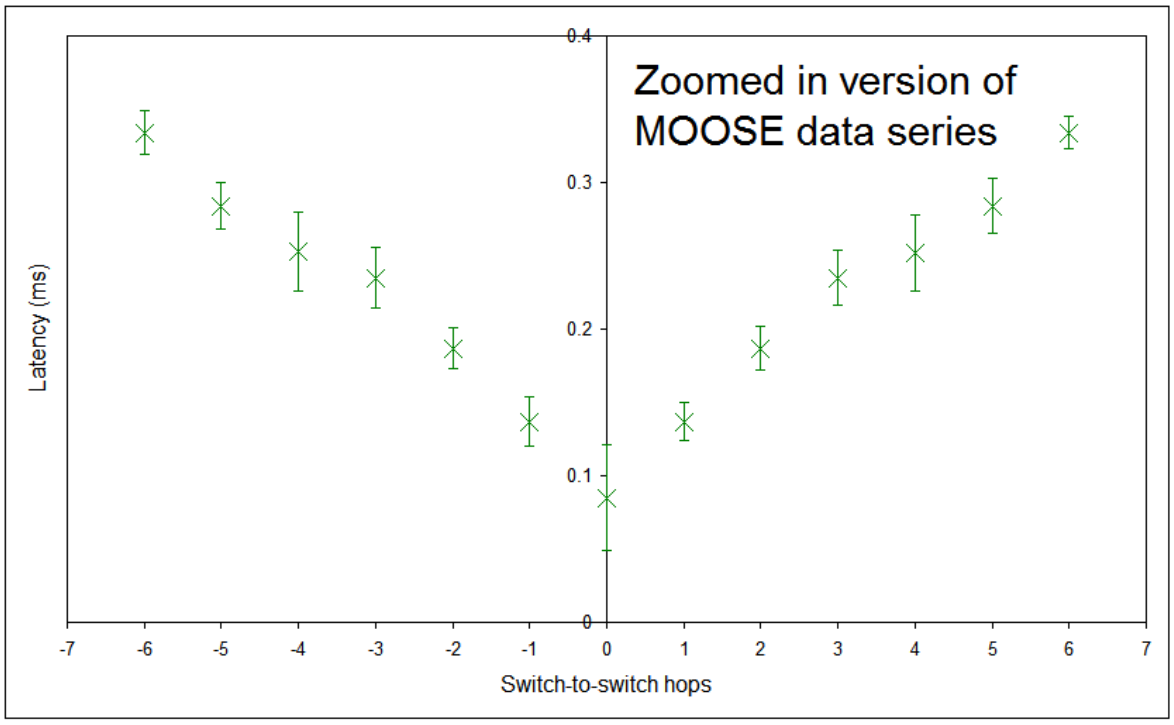


Figure 4.8: MOOSE network topology of optimal routes experiment — Bold links show route taken



(a) Ethernet network (with RSTP) and MOOSE network (without RSTP)



(b) Enlarged version of just MOOSE network (without RSTP)

Figure 4.9: Graphs of ping times around a loop in both Ethernet network with RSTP and MOOSE network without RSTP (above), and enlarged version of graph with just MOOSE network (below). Error bars indicate ±1 standard deviation.

The unit of the horizontal axis is the smallest number of switch-to-switch hops between the hosts. The value is positive if those hops are in a clockwise direction and negative if they are in an anticlockwise direction, in the network topology shown in figures 4.7 and 4.8.

The error bars provide insight into the variation in marshalling delays as packets traverse switches. These marshalling delays are caused by the buffering of packets in one or more consecutive switches due to contention on the outgoing port. Recall that the ping messages (from which these measurements are derived) share the network with other data packets such as RSTP messages and background messages between hosts.

From the symmetry of the MOOSE graph it is clear that the same number of hops were being traversed between hosts which are the same distance apart, regardless of direction. From the asymmetry of the Ethernet graph, it is clear that longer paths were being traversed where the shortest path would require crossing the link between switches `02:01:01:01` and `02:0D:0D:0D` than where this link was not on the shortest path. The improvement in efficiency of use of links, and optimality of routing in the MOOSE network has been very clearly shown.

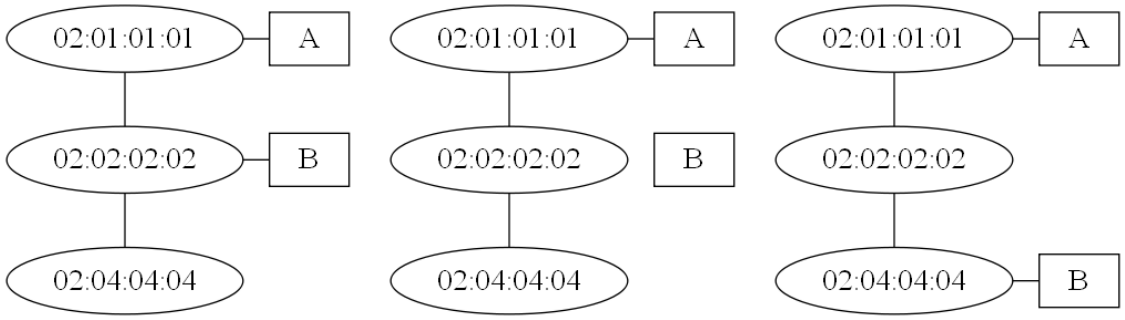
4.1.3 Host Mobility

In an Ethernet network, when a host moves from one switch to another, connectivity with any part of the network with which it was formerly connected is blocked for several minutes as switches flush caches about the host's location. Eventually, when the host has not been seen by other hosts for some time, other hosts' ARP caches will clear, and to communicate with the host, they will need to send a new ARP request by broadcast, finding its new location.

To demonstrate that moving a host from one switch to another did not have this delay in a MOOSE network, three NetFPGA switches were set up in a line, each connected to the next in the line. A host (***A***) was connected to the switch at one end of the line, and another host (***B***) to the middle switch. A long TCP transfer, typical of an FTP or web download[5], was initiated between hosts ***A*** and ***B***.

Static entries were put in the ARP tables of each host indicating the MOOSE address of the other host. In this way it was ensured that when ***B*** was moved to another switch and allocated a new MOOSE address by that switch, its old MOOSE address was being used, with MOOSE mobility between switches, rather than an ARP request being sent to get its new address, which happens in Ethernet as described above (except returning the old address, as no new address is allocated in Ethernet).

[5]This transfer was large enough to ensure that it would not complete before the end of the experiment



(a) Before moving the host (b) While moving the host (c) After moving the host

Figure 4.10: Topology of network for host mobility experiment

Approximately two seconds after the TCP flow was initiated, the cable connecting host ***B*** to switch `02:02:02:02` was unplugged, and was connected instead to switch `02:04:04:04`.

This experiment was repeated multiple times and it was clear that results were consistent in each experiment. When using MOOSE switches, the uninterrupted transfer time was found to be $6.4s \pm 0.5s$. The interrupted transfer time was found to be $10.2s \pm 0.9s$. In the interrupted transfer, a pause of approximately three seconds was observed between the disconnection of the cable and the resumption of the transfer. This was due to a combination of the physical time that the cable was disconnected (around one second), the time taken for switch `02:04:04:04` to send a multicast packet notifying switch `02:02:02:02` of the newly connected host, the time taken for switch `02:02:02:02` to modify its OpenFlow rules to forward packets, and the time taken for TCP Slow Start [20, §3.1] to resume the TCP flow.

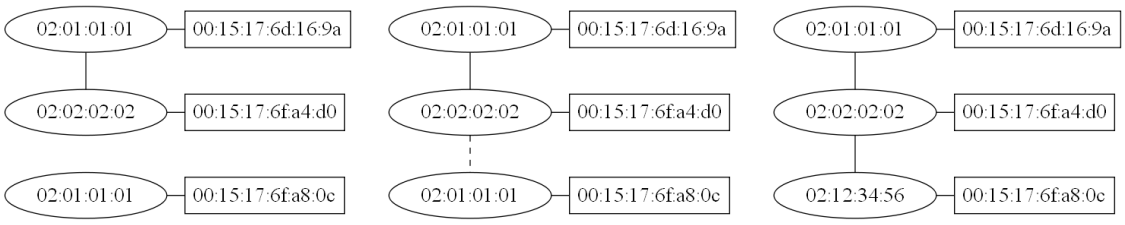
When using Ethernet switches, the uninterrupted transfer time was comparable to the MOOSE transfer time. The interrupted transfer, however, did not resume in over five minutes. This is because when multiple retransmissions of TCP data are not successfully delivered for five minutes (a global timeout set out in the TCP standard [21, §3.8]), the TCP connection is closed. Though there was physical connectivity between the hosts participating in the TCP session, by default Ethernet switches are specified to cache the mapping between ports and addresses for 300 seconds [9, §7.9.2], thus any host which is moved between switches will be disconnected from the network for at least 300 seconds; the same amount of time as TCP's retransmission timeout. These effects combine pathologically so that the Ethernet switch blocks the TCP traffic for long enough to cause TCP to close the session, and so abort the file transfer. MOOSE encounters no such problems for host mobility as the network only interferes with TCP retransmissions for two seconds.

To confirm that this was the cause of Ethernet's failing, I set up a second Ethernet network with a cache timeout of five seconds, rather than the specified 300. This resulted in a total delay of about 10 seconds (including physical cable relocation time), followed by a reliable resumption of TCP traffic. From this, it can be concluded that a

cache timeout significantly lower than that specified (and too low for efficient network functionality) would be required to permit mobility, yet MOOSE mobility just works.

4.1.4 Conflict Resolution

Because MOOSE addresses are guaranteed in uniqueness only if switches allocate themselves unique addresses on the network, I had devised a system to resolve conflicting switch addresses (described in Section 3.1.1). To test this, I set up a network of two switches, `02:01:01:01` and `02:02:02:02`, each with one host attached. The hosts were set to continually `ping` each other. A third switch was attached to `02:02:02:02`, also with the address `02:01:01:01`. This new switch's host attempted to `ping` the host attached to `02:02:02:02`. The conflict resolution protocol was immediately observed (using Wireshark) to segregate the new switch from the rest of the network, and send the correct packets to each switch. This, in turn lead to the new switch assigning itself a new address, and its host having connectivity to the network.



(a) Before conflicting switch was connected (b) During conflict (c) After conflict was resolved

Figure 4.11: Topology of network before conflicting switch was connected (left), during conflict (middle), and after conflict was resolved (right)

The test was run again, but instead of attaching a new switch to the network, switch `02:01:01:01` was moved to be connected to a different port on switch `02:02:02:02`. Again, the conflict resolution protocol properly segregated the switch from the network, and when `02:01:01:01` was found not to be online on its old port, switch `02:02:02:02` updated its forwarding table to reflect the new topology, and pings started to be successful once again.

(a) Before moving switch (b) After moving switch

Figure 4.12: Topology of network before (left) and after (right) switch `02:01:01:01` was moved

The devised conflict resolution protocol was demonstrated to work as expected in these cases.

4.1.5 Summary of Results

I successfully gathered conclusive, unequivocal results showing the research into MOOSE to be worthwhile. The forwarding table size evaluation demonstrated that the forwarding tables grew exactly as predicted. The shortest path routing evaluation demonstrated incredible improvements in the behaviour of MOOSE over Ethernet. The host mobility evaluation showed again a clear improvement in the behaviour of MOOSE over Ethernet.

I was implementing a preliminary protocol specification. As an unintended but desirable outcome of my project, I have tested how implementable the specification is and found some areas which could be improved, which I describe in section 5.1. These have been put to the MOOSE research team, and are currently being used to modify the MOOSE proposal.

I also encountered several limitations in the OpenFlow research project. I have put some potential solutions to the OpenFlow developement team and they are being considered for inclusion in future specifications of OpenFlow. These will be useful to network researchers in the future, and I describe them in section 4.3.3.

4.2 My Project

Subject to the modifications made to my proposal to accommodate using OpenFlow as my prototyping platform (meaning that absolute latency would no longer be considered as an evaluation metric, and that specific targetted metrics rather than general network traffic would be used), my project succeeded in every goal that I set, as well as every extension suggested. I produced a fully functioning MOOSE switch on the NetFPGA platform, and deployed it in test networks to compare MOOSE's performance to that of Ethernet. A great deal more work than originally expected was done, such as learning a new programming language (Python) and prototyping platform (OpenFlow), both outside the scope of the tripos, and writing two versions of the switch.

The improvements of MOOSE as predicted in [3] were all shown to be correct, and MOOSE was shown to be a protocol worth further investigation. Some areas for future work have been suggested, and these are summarised in Section 5.1.

4.3 Limitations

Several factors limited which areas could be evaluated, and how effective evaluation of some areas could be.

4.3.1 Number of NetFPGA Ports

NetFPGAs have only four network ports, limiting the size and topologies of networks which can be modelled with them. This was particularly noticeable when evaluating the size of the forwarding table in Section 4.1.1, where having 24- or 48-port switches would give much more useful and realistic results. I had hoped to be able to obtain some commercial switches with OpenFlow support for this testing, but this did not eventuate. The choice to use OpenFlow, however, makes this an easy evaluation to perform as such hardware becomes available.

4.3.2 Use of OpenFlow

Using OpenFlow, rather than a Verilog implementation, meant that the latency of MOOSE could not be fairly compared with that of Ethernet, as outlined in Section 2.2.1. Areas for further investigation include the latency introduced by the address re-writes on the first and last MOOSE switches that a packet encounters and the latency introduced by the check for whether the incoming packet is destined for a locally attached host. These could not be feasibly tested using OpenFlow, and would require a more native implementation.

4.3.3 Limitations of OpenFlow

OpenFlow only supports exact matches for Ethernet addresses in the hardware flow-table, so many more rules needed to be created than were strictly necessary: between every pair of hosts and pair of switches, rather than simply between pairs of switches, and between locally attached hosts and switches. This artificially restricted the size of test networks. Support for partial matches for Ethernet addresses, as is present for IP addresses, would mitigate this.

Better still would be the facility to pipeline hardware action lookups. If actions could be performed in stages, the *rewrite source address if necessary*, *rewrite destination address if necessary*, and *lookup output port* operations could have taken part separately, simplifying the programming model, and reducing the number of flow-table entries.

These suggestions for improvements have been received warmly by the OpenFlow development team.

4.4 Changes with Hindsight

My project was very successful. The decision to use OpenFlow was, I believe, the right one, as having a useful and clear reference implementation with the easy ability to make small changes and gather data was incredibly useful, and will continue to be in the future, especially as more hardware becomes available.

Had I had more time, I would have made some modifications to OpenFlow to implement some of the suggestions in Section 4.3.3. Unfortunately, these modifications were well beyond the scope of this project.

Toward the end of the implemention procedure, when finishing off and tidying up my code, I found it to be messier and more complicated than it needed to be. First I wrote a standalone switch. I then modified it to learn about other switches' locations as Ethernet does for hosts, to test that my implementation was working on the hardware. Thereafter, I modified this switch so that it would interact with other switches and participate in a routing protocol. This meant that I could concentrate on MOOSE during implementation, and provided a useful way to learn about OpenFlow, but meant that the routing protocol was not a built-in part of my code, and had to be bolted on. I believe that this is why issues with conflict resolution were not found until the experiments were being performed. Now that I understand OpenFlow and MOOSE more fully, if I were to start again, I would write the routing protocol and inter-switch communication first, because this is the primary network function. I would then add in the host-specific functionality, which is more special-case behaviour.

4.5 MOOSE

MOOSE has been shown to be an improvement over Ethernet in many important areas. Clearly more research must be done into areas such as latency of implementation, but the protocol warrants this further research, and has been shown to be a viable and worthwhile replacement for Ethernet. Specific areas for further research have been summarised in Section 5.1.

Chapter 5

Conclusions

5.1 Future Work

I have identified several areas for future research:

5.1.1 Multicast Traffic

There was a significantly higher amount of background traffic in the MOOSE networks than the Ethernet networks because OSPFM is a protocol with a relatively high use of multicast traffic. OSPF is known to run into issues in networks of over approximately 1000 routers [22], so a similar order of magnitude limit could be reasonably expected to be placed on OSPFM. Unfortunately, this restricts MOOSE's scalability. Using a Designated Router in OSPFM, another link state routing protocol with less reliance on multicast traffic (such as IS-IS [23]), or a distance-vector routing protocol could remove (or at least significantly raise) this limit.

5.1.2 Discovery of Hosts

To be allocated a MOOSE address at all, a host needs to send a packet. In this way, the switch will become aware of the existence of the host. Until a host sends a packet, it will not be able to receive packets. I forced packets to be sent by having each host which was being used in any particular experiment ping some other host before the experiment began. In real deployment, some DHCP or ARP packet is almost certain to be sent by any host within seconds of connection to the network, getting around this problem, but some thought should go towards solving this problem, perhaps by use of the Link Layer Discovery Protocol [24].

5.1.3 MOOSE-Ethernet Interaction

Networks containing both MOOSE and Ethernet switches were not considered in this work. Such networks could cause issues. My prototype switch can be used for research into these networks in the future.

5.1.4 Conflict Resolution and Loops

In the network-loop test, it was noted that as soon as OSPFM packets started going around the network, conflicts started being detected. Some investigation showed that the multicast link-state update packets, sent as part of the OSPFM protocol to notify switches of the network topology, were being sent both ways around the loop, and so conflict was detected. This conflict could not be resolved as when a switch was informed to change its address, a new conflict would be detected for its new address in the same fashion. Accordingly, for that set of tests, I disabled conflict detection. I offer several potential solutions to this problem below.

Only detect conflict in HELLO packets
If the only switch to attempt to resolve conflict were to be one with a direct connection to the switch, then the port to which the switch is connected could be known for certain, and conflict detection would become more reliable.

Globally unique switch identifiers
Routing information and identification needn't be so closely tied together. Each switch could be allocated a unique identifier on manufacture, which it would include alongside its MOOSE prefix in OSPFM messages, so that switches could better distinguish between conflict and the presence of loops. This would increase the memory requirement of switches, but that memory needn't be particularly fast, as it is only accessed as part of the control function of the switch, rather than in the path of packets.

A DHCP-like protocol
When hosts join a network, they either allocate themselves an IP address of their choice, or request one from the network. This is typically done using the Dynamic Host Configuration Protocol [25], whereby some computer (or router) acts as a DHCP server for a certain subnet, responsible for allocating all IP addresses on the network. A similar system could be introduced for MOOSE switches.

5.2 Summary

For this project, I taught myself many technologies outside of the tripos (MOOSE, NetFPGA, OpenFlow, Python, protocol implementation and hardware implementation) to support knowledge from several courses (mostly networking and systems based).

I successfully implemented a complete and fully-functioning MOOSE switch that achieved all of the goals, as well as each of the ambitious extensions, in my proposal. I designed experiments to show that MOOSE, a previously unimplemented research network protocol, serves as an improvement over Ethernet, the dominant data-link layer network protocol. These experiments showed significant measurable improvements in the size of forwarding tables, utilisation of physical links in the network and migration of hosts between switches, as well as that shortest path routing was possible with MOOSE. I also extended the current research agenda of MOOSE myself, introducing a practical and useful scheme of using variable length prefixes and offering a way of detecting conflicts of switch addresses. The insight gained from making and deploying MOOSE switches and networks has enabled me to suggest several areas for future research into MOOSE, which are being actively pursued by the MOOSE team. Alongside specific contributions, I also put forward some areas of potential improvement in the OpenFlow project which are being considered for inclusion in a future specification. My project was successful in all of its aims, and showed MOOSE to be a worthwhile protocol.

Bibliography

[1] Alan Newbold of Ove Arup, quoted by Peter Judge of TechWorld. Case Study: Getting to grips with UK's biggest WLAN. Online, http://howto.techworld.com/mobile-wireless/4029/case-study-getting-to-grips-with-uks-biggest-wlan, March 2008. Retrieved 2010-05-06.

[2] Ian White, Richard Penty, Jon Crowcroft, Jaafar Elmirghani, Marc Clement, Alwyn Seeds, and Paul Brennan. The INtelligent Airport (TINA): A Self-Organising, Wired/Wireless Converged Machine. Presentation to EPSRC, March 2006.

[3] Malcolm Scott, Andrew Moore, and Jon Crowcroft. Addressing the scalability of Ethernet with MOOSE. In *ITC 21 First Workshop on Data Center – Converged and Virtual Ethernet Switching (DC CAVES)*, September 2009.

[4] Andrew W. Moore, Laura B. James, Madeleine Glick, Adrian Wonfor, Ian H. White, Derek McAuley, and Richard V. Penty. Chasing errors through the network stack: a testbed for investigating errors in real traffic on optical networks. *Communications Magazine, IEEE*, 43(8):s34 – s39, August 2005.

[5] Robert M. Metcalfe and David R. Boggs. Ethernet: Distributed Packet Switching for Local Computer Networks. *Communications of the ACM*, 19(7):395–404, July 1976.

[6] IEEE. 802.3i-1990 IEEE Supplement to Carrier Sense Multiple Access with Collision Detection CSMA/CD Access Method and Physical Layer Specifications: System Considerations for Multisegment 10 Mb/s Baseband Networks (Section 13) and Twisted-Pair Medium Attachment Unit (MAU) and Baseband Medium, Type 10BASE-T (Section 14), 1990.

[7] Kostas Pagiamtzis and Ali Sheikholeslami. Content-Addressable Memory (CAM) Circuits and Architectures: A Tutorial and Survey. *Solid-State Circuits, IEEE Journal of*, 41(3):712 – 727, March 2006.

[8] 3Com Corporation. Switch 5500G 10/100/1000 family data sheet. Online, http://www.3com.com/other/pdfs/products/en US/400908.pdf. Retrieved 2010-03-14.

[9] IEEE. 802.1D Standard for Local and metropolitan area networks: Media Access Control (MAC) Bridges, June 2004.

[10] IETF. RFC2328 OSPF Version 2, April 1998.

[11] Jad Naous, Glen Gibb, Sara Bolouki, and Nick McKeown. NetFPGA: reusable router architecture for experimental research. In *PRESTO '08: Proceedings of the ACM workshop on Programmable routers for extensible services of tomorrow*, pages 1–7, New York, NY, USA, August 2008.

[12] Jad Naous, David Erickson, G. Adam Covington, Guido Appenzeller, and Nick McKeown. Implementing an OpenFlow switch on the NetFPGA platform. In *ANCS '08: Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, pages 1–9, New York, NY, USA, November 2008.

[13] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Computer Communication Review*, 38(2):69–74, April 2008.

[14] The NOX Team. Developing within NOX webpage. Available from http://noxrepo.org/manual/app.html — Retrieved 2010-03-25.

[15] Malcolm Scott, Daniel Wagner-Hall, Andrew Moore, and Jon Crowcroft. Addressing the Scalability of Ethernet with MOOSE. Available from http://www.cl.cam.ac.uk/∼mas90/MOOSE/MOOSE.pdf, 2010.

[16] IETF. RFC791 Internet Protocol, September 1981.

[17] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[18] Guido Appenzeller and Brandon Heller. OpenFlow Sigmetrics 2009 Hands-on Tutorial Instructions. Available from http://www.openflowswitch.org/wk/index.php/OpenFlow_Sigmetrics_2009_Hands-on_Tutorial_Instructions — Retrieved 2010-04-14.

[19] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, second edition, 2001.

[20] IETF. RFC5681 TCP Congestion Control, September 2009.

[21] IETF. RFC793 Transmission Control Protocol, September 1981.

[22] Rick Graziani and Allan Johnson. *Routing Protocols and Concepts, CCNA Exploration Companion Guide*. Cisco Press, 2007.

[23] ISO. ISO/IEC 10589:2002(E) Telecommunications and information exchange between systems — Intermediate System to Intermediate System intra-domain routeing information exchange protocol for use in conjunction with the protocol for providing the connectionless-mode network service (ISO 8473), November 2002.

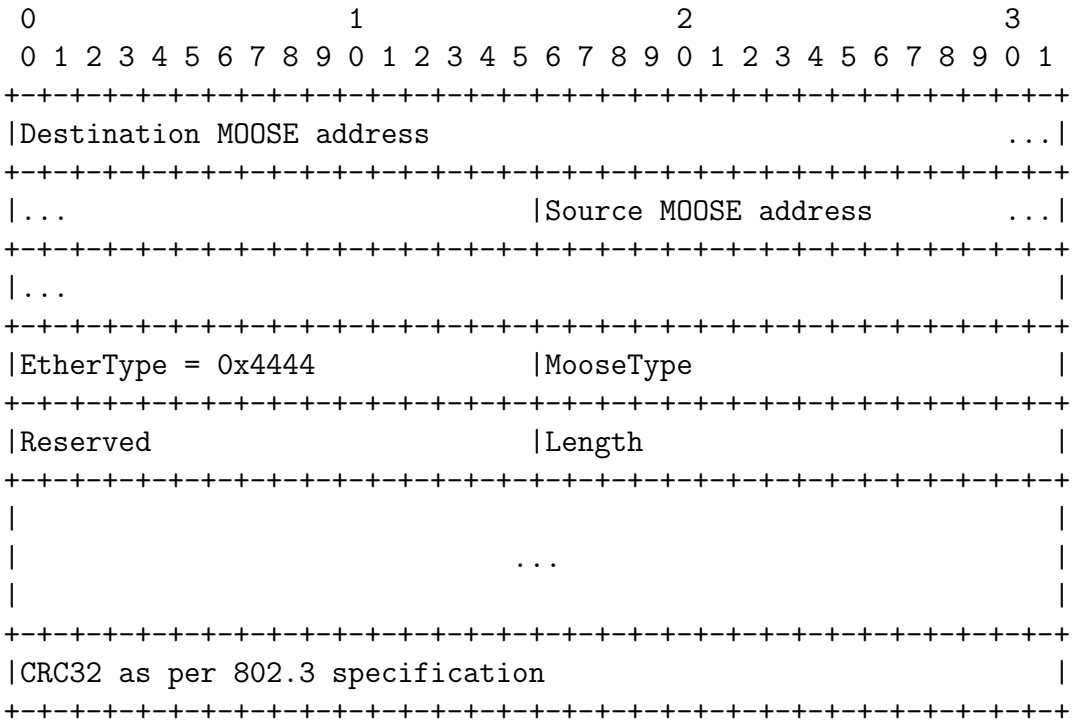
[24] IEEE. 802.1AB Station and Media Access Control connectivity discovery, May 2005.

[25] IETF. RFC2131 Dynamic Host Configuration Protocol, March 1997.

Appendix A

Status and Management Interface Frame Format

All Status and Management Interface (SAMI) frames are valid Ethernet frames. I have tentatively used EtherType 0x4444 to indicate SAMI frames. The following format should be used for SAMI frames:



Where length refers to the length of rest of the frame after the length field, not including the CRC32 in bytes (i.e. the total length of the frame in bytes minus 48). If the total length of the frame is less than 64 bytes, there must be sufficient padding before the

CRC32 to pad it to exactly 64 bytes, but the length field should refer to the length of the actual data without padding.

MooseTypes and expected additional data are specified as follows:

MooseType	**Meaning**	**Additional data**	**Notes**
0x0000	Reserved	None	None
0x0001	Echo request	None	Must be responded to with an echo reply when received by the unicast destination address.
0x0002	Echo reply	None	None
0x0003	Conflict notification	None	Advises unicast destination to change its address on receipt. Does not require that destination changes its address.
0x0004	Host attachment notification	The host's MOOSE address, followed by the real Ethernet address of the host.	Normally sent to the *All MOOSE switch* multicast address `FF:00:00:00:00:FF`
0x0005	Encapsulated broadcast message	A 64-bit sequence number, followed by the entire broadcast packet (including full Ethernet headers)	Sent to `All MOOSE switch` multicast address. Should be decapsulated and forwarded to all hosts if it has not been received before
0x0006	OSPFM Packet		

Support for echo request, echo reply and conflict notification are required in any MOOSE device.

Appendix B

Figures of Results

Loop test — wireshark screenshots

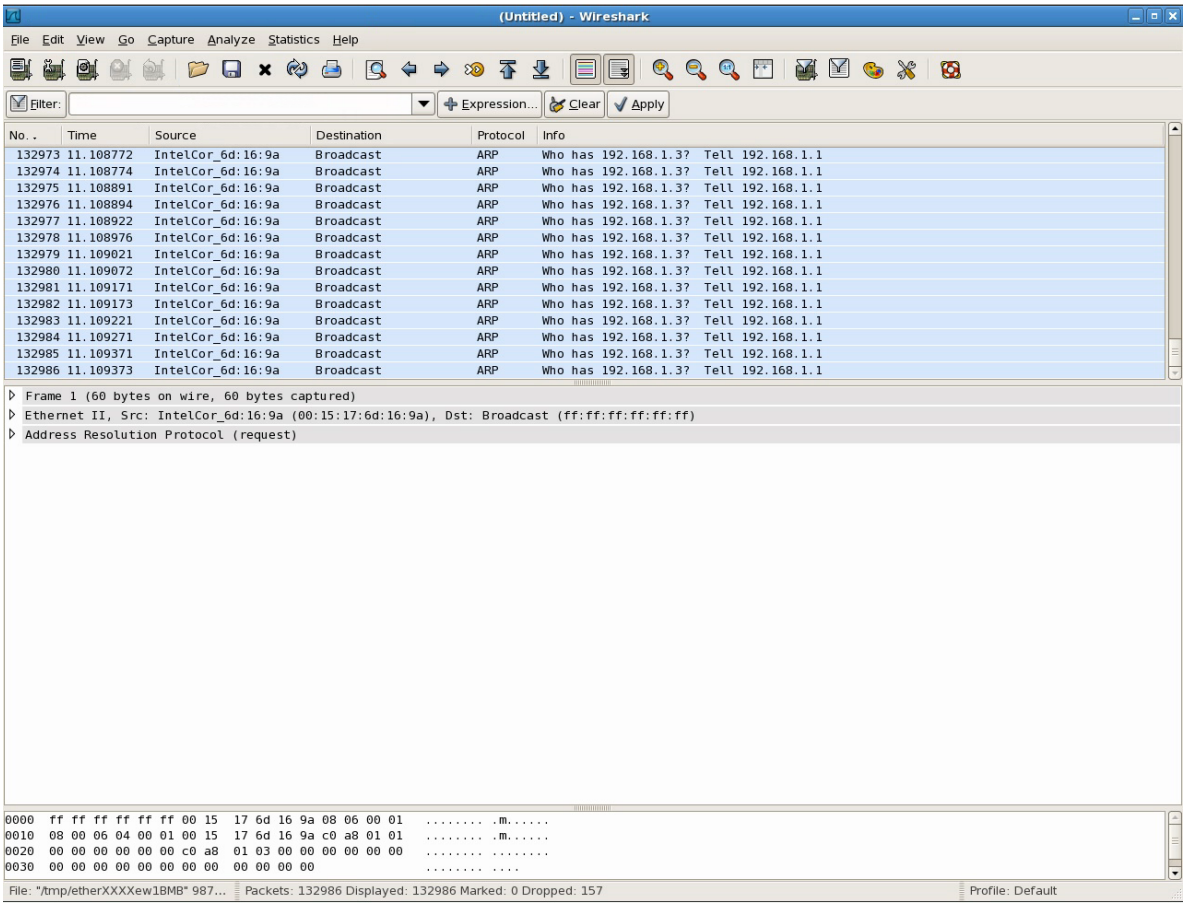


Figure B.1: Screenshot of Wireshark packet capture after broadcast on a loop using Ethernet

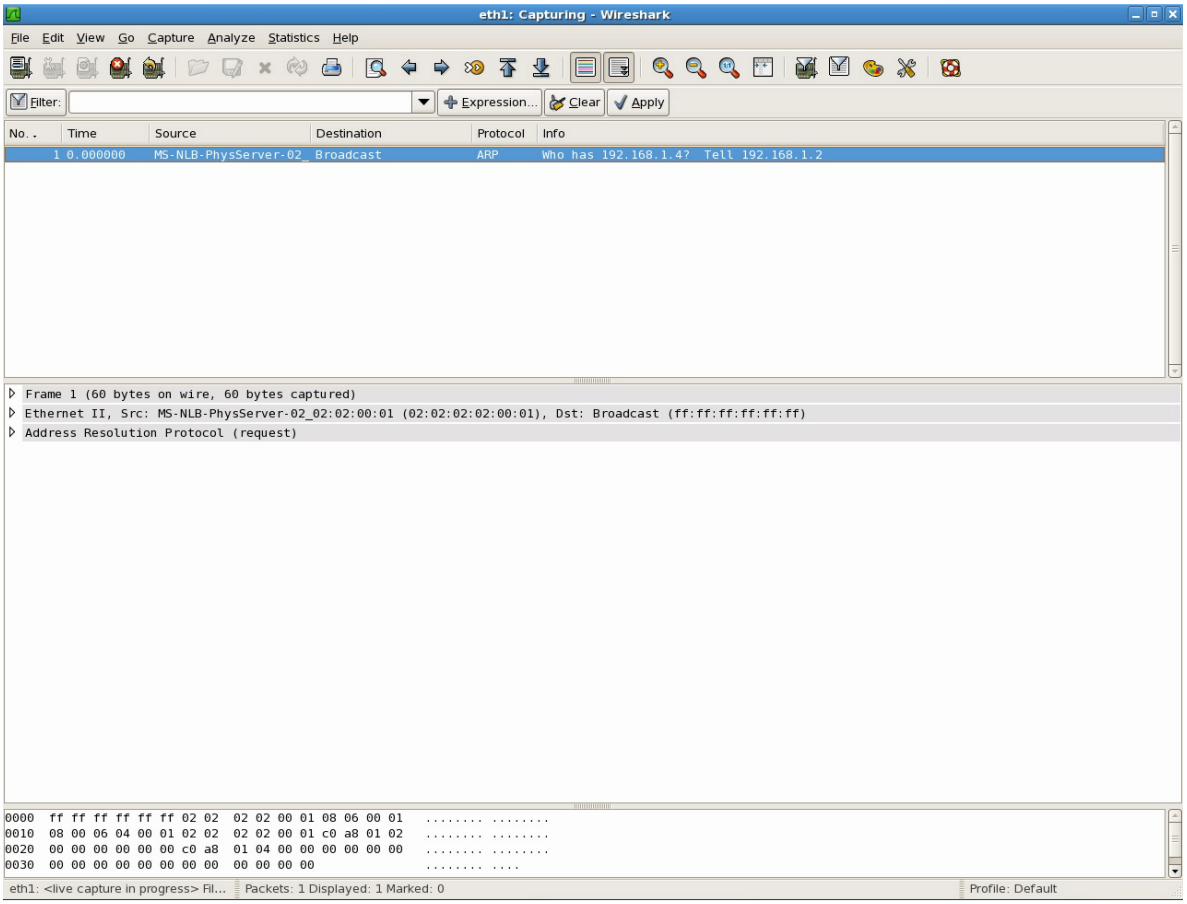


Figure B.2: Screenshot of Wireshark packet capture after broadcast on a loop using MOOSE

Appendix C

Project Proposal

Part II Computer Science Project Proposal

NetFPGA Implementation of MOOSE

D. A. Wagner-Hall, Homerton College (daw63)

Originator: Malcolm Scott (mas90)

October 20, 2009

Special Resources Required

Duration of project:

- Two NetFPGAs
- Three gigabytes of PWF storage space
- Account on CUCL, NetFPGA, NetOS groups
- Account on MPhil machines in SW02
- Access to rooms SW02, SE18

Short period (End of February):

- As many NetFPGAs as can be obtained (hopefully 10)
- Up to six Ethernet switches
- Up to several dozen (12-50) general purpose non-virtual computers

Project Supervisor: Dr A. W. Moore (awm22)

Director of Studies: Dr R. K. Harle (rkh23)

Project Overseers: Dr S. B. Holden (sbh11) and Dr C. Mascolo (cm542)

Introduction

MOOSE [3] is a proposed backwards-compatible replacement for Ethernet as a layer-2 network protocol, which claims to offer benefits over Ethernet in the following areas:

- reduction in size of routing table;
- better utilisation of physical links;
- reduced end-to-end latency of transmission in the common case;
- better migration of hosts between switches;
- shortest-path routing (subject to a routing protocol).

I will prototype a realistic, representative MOOSE switch using the NetFPGA platform, in order to set up a test network of several NetFPGA switches, interspersed with Ethernet switches. I will send IP traffic across this network over several hours. The NetFPGA systems will then be set to operate as standard Ethernet switches, and the traffic replayed. Data will be gathered during these runs to compare the above metrics so that the effectiveness of MOOSE as a protocol can be evaluated.

Work to do

The project breaks down into the following main sections:

1. Create a NetFPGA MOOSE switch with the following capabilities:

   Have attached hosts: The switch must:

   - recognise hosts when they attach to it and send packets;
   - listen for frames with a MAC source address and allocate a MOOSE address to that host;
   - rewrite the source address of all frames from that MAC address to its MOOSE address;
   - rewrite the destination address of any frame destined for that MOOSE address to its MAC address.

   Participate in a network: The switch must:

   - receive frames;
   - send frames;
   - build up and update a routing table;
   - interoperate flawlessly with Ethernet.

   Some possible extensions:

   Join a network without prior configuration: The switch could:

- choose its switch identifier, rather than manually configure this;
- resolve address conflicts without administrative intervention;
- increase its available host address space by changing to a shorter prefix, if it exhausts its current host address space.

Support host mobility: The switch could:

- send notifications of new host attachments to other MOOSE switches;
- receive those notifications;
- forward packets for that host accordingly, where relevant.

2. Finding or devising a routing protocol (likely some OSPF variant) to be used.

3. I will then set up a test network of these MOOSE switches, along with some Ethernet switches and several attached PCs, and send traffic across the network to measure the metrics listed above.

Initial Learning Tasks

The following main learning tasks will have to be undertaken before the project can be started:

- learning about switching fabric;
- learning about NetFPGA as a tool;
- devising topologies and traffic patterns for reasonable evaluation of the metrics.

Starting Point

I have a thorough knowledge of MOOSE, having been working on it with Malcolm Scott for some months.

I may modify an existing open-source routing protocol implementation, rather than write my own.

There exists a reference NetFPGA Ethernet switch which I will likely modify, rather than write the entire switch from scratch.

Resources

A NetFPGA system is required throughout the project on which to actually make the prototype. A second NetFPGA system is required for testing switch-switch interaction (the *participate in a network* work to be done) while prototyping.

The other switches and machines will be used to set up a network in which to test traffic.

I intend to keep my code, documentation and write-up in a Subversion repository on the PWF, which is automatically backed up. I will also keep an out-of-Cambridge working copy, though not the entire history, in case of catastrophic disaster.

I have talked with Andrew Moore, my supervisor, and he is happy to arrange the resources listed above.

Work Plan

	Date	Objective
	2009-10-26	**Start of work**
1	2009-11-15	Finish experimenting with NetFPGA, Verilog, OpenFlow, the NetFPGA reference switch, and NetFPGA PWOSPF implementation
2	2009-11-22	Put together skeleton Ethernet-based switch
3	2009-11-29	Source and destination address rewriting
4	2009-12-06	Automatic host identifier allocation
5	2009-12-27	*Draft of introduction written*
6	2010-01-08	Routing protocol implemented
7	2010-01-24	*Draft of preparation written*
	2010-01-29	**Progress report deadline**
8	2010-01-31	Automatic switch identifier selection and conflict resolution
9	2010-02-07	Switch can update to shorter prefix to grow address space
10	2010-02-21	Host mobility implemented
11	2010-03-05	All test-network data collected
12	2010-03-31	*Draft dissertation written*
	2010-05-14	**Dissertation Deadline**