Richard Whitehouse

Homerton College, 2011

Part II Computer Science Tripos

Implementation of Data Link Layer Protocols for a Network Simulator

---

Author:
Richard Whitehouse

Supervisor:
Malcolm Scott

May 20, 2011

Proforma

Name:	**Richard Whitehouse**
College:	**Homerton College**
Project Title:	**Implementation of Data Link Layer Protocols for a Network Simulator**
Examination:	**Part II, Computer Science Tripos, July 2011**
Word Count:	**10000**
Project Originator:	Malcolm Scott
Supervisor:	Malcolm Scott

Original Aims of the Project

To write a implementation of MOOSE for a modern network simulator which functions correctly with respect to the protocol, in that it rewrites packets, and performs correctly in a network of other simulated MOOSE and Ethernet switches. To form network topologies which seek to show the differences between MOOSE and Ethernet. To evaluate said topologies by showing the difference under different metrics. To collect metrics showing the efficiency of the routing, the size of the state table and the amount of overhead.

Work Completed

The creation of a implementation of a MOOSE switch for the network simulator ns3 which performs address rewriting in Ethernet frames, rewrites ARP packets and routes according to a statically determined routing algorithm. The extension of a Ethernet switch as a comparison including the facility to forward packets according to a statically implemented spanning tree protocol. The evaluation of the said switches under a number of different topologies and the collection of metrics to show the differences between the protocols.

Special Difficulties

None.

Declaration

I, Richard Whitehouse 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: Richard Whitehouse

Date: May 20, 2011

Contents

List of Figures

Chapter 1

Introduction

Data Link Layer protocols are responsible for data transfer between adjacent nodes, routing of data within a local network segment as well as providing and regulating access to the physical network layer and allowing services to be run on top. Ethernet, the dominant protocol in this area, has been around since the early 1970s [27]. Research has indicated [20] that Ethernet has a number of issues as the size of the network scales up. A number of different proposals have been made to improve Ethernet, notably SEATTLE [16] and MOOSE [25].

Since the number of hosts will need to be very large in order to demonstrate the issues associated with Ethernet, and they will be arranged in a number of different topologies, the most suitable mechanism in order perform this evaluation is under simulation, where the topology of the network can easily be altered.

In this paper, I will describe the creation of a representative implementation of Ethernet and one of the proposed improvements, MOOSE, for a network simulator. I have chosen to focus on MOOSE as other protocols, like SEATTLE, have already been evaluated [15]. I will then use these implementations to run tests to explore their differences and make a quantitative evaluation under a number of different situations.

1.1 Background

1.1.1 Protocol Stack

When implementing a computer network architecture, it is advantageous to separate the different functions into different protocol layers, with each layer providing services the layer above, and requiring services from the layers below. By performing this separation, we can divide up the complexity required to be implemented. This is done by each layer providing an abstraction for the layer above and below. This allows each layer to concentrate on only providing a small amount of functionality. This technique also allows the system to be modular,

and allows us to select different protocols to perform different tasks for different applications on different architectures.

Several different proposals have been made for the different layers of the network stack, the seven layer ISO OSI Model [30] and the three layers in the Internet Protocol Suite [4]. In practice the model used by the Internet Protocol is largely representative, along with a additional physical layer and link layer.

Each layer describes the protocol data unit (PDU) used, as well as a the services it provides, and the terminilogy used.

Physical Layer

The stack starts at the bottom with the physical layer. This is primarily concerned with the physical interconnection of the two computers and how the basic signalling works in terms of cabling or radio. Standards on this layer include 100BASE-TX [10] and 1000BASE-T [11] - implementations of Fast Ethernet amd Gigabit Ethernet respectively. The data on this layer is in the form of a raw bitstream.

Data Link Layer

The data link layer provideees basic addressing and controlling access to the physical layer, as well as subnet routing. A subnet is a small part of the complete network, often this may be a Local Arean Network (LAN), though it may be as wide as an entire city, or just a small component of a LAN. This layer provides easy mobility of hosts among the subnet, without requiring reconfiguration. The dominant protocol in this area is Ethernet as part of IEEE 802.3. Data sent to this layer is encapsulated in a Ethernet frame, which then has additional metadata placed at the header and trailer of the frame.

Network Layer

The network layer provides routing across the wider network, and identification of hosts on a global scale, with current networks having estimated as having a billion hosts. The primary protocol in this area is IP - Internet Protocol, in particular IPv4 [23] although IPv6 [6] is starting to gain traction due to impending address exhaustion [9]. This provides IP Packets as the primary abstraction.

Transport Layer

The transport layer provides multiplexing of data between applications on end hosts. On this level, protocols can also specify the capability for connection setup, reliability, encryption and ordering. Examples of protocols on this layer are UDP [22] and TCP [5]. TCP provides TCP segments which are reliably transmitted,

ordered and provided at the other end, whereas UDP provides UDP datagrams which are simply sent to the opposite host, with no reliability or ordering.

Application Layer

The top layer of the protocol stack in the Internet Protocol Suite model and the top three layers in the OSI Reference Model, Application, Presentation and Session, can be considered as a singular discrete layer, the application layer. These provide user level services situated on end hosts. Examples include HTTP [8] and DNS [18]. These provide for the transmission of material in the form of hypertext documents and other media and a distributed directory of names and addresses.

1.1.2 Simulation

Network simulation is a evolving area of research, with many different simulators and simulation techniques being used. It is rapidly becoming the most popular way of performing large scale network research for both local area networks, and the wider internet due to the low cost and speed of iteration of different tests that it allows. When testing networks of significant scale, the approach commonly taken is to verify it works on a small testbed, before scaling up under simulatuion.

There are a number of different approaches to simulating a computer network architecture. The first is discrete event simulation. Here a simulator is primed with a topology and a number of event sources. These event sources fire events, which are executed in turn, which may then fire other events. The events are stored in a priority queue. Each event is executed according to the time associated with it, with a global variable containing the current time of the simulation being advanced upon the executing of each event. This is continued until either there are no more events left in the queue, or some predefined time has passed. This simulation technique has the advantage that periods of time in which nothing of interest occurs are passed over quickly, while events which are notable are where the majority of the computation is spent.

Other types of network simulation include Markov chain simulations which are useful for modelling queueing systems. These depend on systems with little state that can be modelled by a stochastic process. This is not the case for packet simulation, where there is a large amount of state and no overall defined equation across the entire simulation.

A further type of simulation is continuous simulation where the timestep is incremented, and events ocurring between the two states are modelled. This works well for physical phonomena, such as radio waves or propagation of signals in wires where the situation is continually evolving. However, they spend the same processing power on all periods of time during the simulation, whether

the period is one of high activity or low activity. Also, in order to get a good representation of the underlying phonemena, you have to use very small time increments which gives a high simulation cost.

1.2 Context

The most relevant work in this area is set out by Scott et al [25] which sets out the ideas behind MOOSE which this paper aims to test under simulation. A prototype NetFPGA implementation also exists by Wagner-Hall et al [28] which outlines a practical implementation of MOOSE. No implementation of MOOSE exists for any network simulator.

1.3 Aims

I aimed to implement a simulation capable of simulating MOOSE and Ethernet networks on a commodity PC to give a realistic comparison of MOOSE and Ethernet.

This will involve designing and developing a number of components:

- The size of the state data held by the switch.
- The efficiency of the routes used by the protocol.
- The different behaviours when broadcasting data and unicasting data.

I aim to show how MOOSE and Ethernet compare as the number of hosts under simulation rises.

1.4 Relevant Courses

I have used the knowledge from the following courses: Digital Communications 1 and Principles of Communication for network concepts, Algorithms 1 and 2 for general algorithm design, Programming in C and C++, as most the majority of the code will be written in C or C++, Software Design and Software Engineering for general software engineering and best practice, Computer Systems Modelling for ideas about simulation, Concurrent and Distributed Systems for looking at how the switches co-operate and Object-Orientated Programming, which is the main software design paradigm used in designing simulators.

Chapter 2

Preparation

In preparation for the implementation, there were a number of areas which I researched, and a number of decisions I made which affect the implementation of the project.

2.1 Learning

2.1.1 Data Link Layer

The first part of the background material is to look at the two different Data Link Layer protocols under evaluation, Ethernet and MOOSE.

2.1.2 Ethernet

Ethernet is the most widely deployed and used Local Area Networking - LAN - technology currently in the market. While the physical layer of the standard has undergone a number of iterations, the data link layer protocol has remained largely stagnant due to the lack of motivation and a desire for interoperability with current equipment.

Ethernet was first described in a memo on May 22, 1973 at Xerox PARC by Bob Metcalfe [17] who designed it to interconnect workstations and printers in a modern computer network [27, p. 125]. It was to be a shared medium network based on the previous work by the University of Hawaii in the late 1960's with the Aloha protocol [2]. Ethernet used the CSMA - Carrier Sense Multiple Access - schema designed for ALOHA for and added Collision Detection, to create a cable connected shared medium network. This allowed the original Ethernet networks to be a single shared medium network where all the computers were connected together using a series of cables. Every computer on the network would receive all the packets and, based on the header, the Ethernet interface would inform the system when a packet arrived which was for the host in question.

This relied on each computer having a unique address to which it could be sent packets, which the controller would listen out for. This was also the only required property - the address did not need to be transferable, it did not need to describe any information about the owner and it did not need to contain routing information. In order to provide this guarantee, Ethernet controllers were allocated a 48 bit long address. The first three bytes, 24 bits, consist of one bit which determines whether the packet is multicast / broadcast or unicast, one bit which determines whether the address is locally administered or universally administered, and then the remaining 22 bits, which in the case of unicast, universally administered addresses, are used to designate a manufacturer, and are assigned by the Institute of Electrical and Electronics Engineers.The last three bytes are assigned by the manufacturer, with the manufacturer guaranteeing that the address is unique, with additional block of of addresses allocated when a manufacturer had run out. Each manufactuer gives each interface a unique address by incrementally assigning the addresses within each block.

As network traffic increased, and more computers were added to local area networks, the reality of all computers sharing a single shared media with which any packet would collide with other traffic became impractical. Computers were also being placed further away from each other. Ethernet defines a limit on the distance between hosts, as when it increases, it decreases the utility of Carrier Sense which increases the likelihood of collision, which causes a process where each host backed off before trying again, thus decreasing utilisation of the network.

The solution to this was to split the single collision domain into a set of different domains which would be joined together using network bridges to form a single broadcast domain [12]. The bridge, or switch as it is also known due to the manner in which it operates, filters traffic between the different collision domains, only sending traffic where it is required. As the number of hosts on the network, and the speed of network traffic increased, more and more collision domains were created. Modern Ethernet networks are entirely switched networks with each host having a full duplex point to point link to the local switch, and each switch being connected directly to other switches, to form a network topology.

The next problem was that as networks became more vital to the operation of a business, it became desirable to have redundant links, so that when a link failed, traffic could be diverted. In order to allow this the switches were often arranged in a way that formed topological loops. However, the Ethernet protocol relies on traffic being sent to a tree, and not a graph with cycles in it, in order to prevent a broadcast storm. This is where broadcast traffic cycles the network endlessly using up all the available bandwidth. In order to solve this Ethernet uses a spanning tree protocol in order to convert the graph of switches into a tree. This tree is not guaranteed to be a minimum spanning tree, instead it is guaranteed that all switches have the shortest possible path to a root bridge, which may be specified by the network administrators. The current iteration of this protocol is the Rapid Spanning Tree Protocol, RSTP, which changes the state

of links until that condition is met. In practical terms this means that redundant links which could be used to increase the available network bandwidth are instead disabled. Should the state of the network change, they will be re-enabled to allow the network to continue to operate.

Bridge Implementation

Modern Ethernet bridges work using a filtering technique based on learned data, alongside the Rapid Spanning Tree Protocol implementation. The RSTP implementation disables ports to form a loopless network.

When the bridge receives an Ethernet frame, it looks at the source address and learns the combination of the source and the port in came in on. It then looks up the destination address and if it finds a match, it will send it out on that port. However, if it fails to find a match it will flood the data on all ports, except the one it came in on.

Should the state table become full, it will be forced to drop an entry from the state table in order to store the new one. This will mean that when a new packet arrives for that destination, it will be flooded.

In any Ethernet network, all destinations will almost always appear in all the state tables. This is because each host utilises ARP, the Address Resolution Protocol, to resolve IP addresses to Ethernet addresses. When a query is sent out, it is broadcast, and as data kept by hosts is only cached for a short amount of time, each must send a broadcast packet regularly if it is sending traffic. When a host sends a broadcast packet, it will cause it's address to be entered into all the state tables.

Limitations

As Ethernet addresses contain no routing information, unlike IP addresses in which parts of the network space are delegated to regions of the network, it is impossible in a practical scenario to perform any form of address aggregation. This means in order for network switches to switch traffic between different network segments it has to contain a lookup between each address and the port to send the data. Due to the high speed at which modern network links operate, this has to be done quickly. Depending on the switch, this might be done using Content Addressable Memory, CAM, [21] which provides $O(1)$ lookup time for unsorted data, however is very expensive in large blocks and consumes considerable amounts of power. Alternatively it can be done using a lookup table, which while cheaper, provides worse performance as the number of entries grows. Since the time spent processing a frame is critical, this limits the feasible size of the state table. This means that modern Ethernet switches have a practical state table size maximum of between 8000 and 32000 hosts [1].

Current trends in data centres, with large numbers of machines in a dense

configuration [13], with each machine able to contain multiple hosts via virtualisation software such as Xen [3], giving each host a virtual Ethernet controller, it is increasingly likely that this limit will be exceeded [25]. This will cause any host whose state isn'tstored in the switch to have their frames entering the switch to be flooded as a new frame on every port. As such the overall traffic on the network will increase, decreasing the utilisation and increasing the delay per packet.

Another limitation is that when the spanning tree protocol converts the network to a tree from the initial graph topology, it disables the number of available links. By doing this, it reduces the available network bandwidth. With a better routing protocol, these links could be put to use to decrease the shortest path between two hosts, as well as being used for multipath routing.

2.1.3 MOOSE

MOOSE, Multi-level Origin-Organised Scalable Ethernet [25], is a proposed improvement on the Ethernet protocol that is designed to be backwards compatible with the large number of ethernet controllers currently in the market place. MOOSE can be implemented in a network by replacing the code operating the bridges. This can either be done by replacing the bridges with a MOOSE bridge, or by installing the MOOSE code on the existing bridges, if they are capable.

MOOSE performs in place rewriting of Ethernet datagrams on the switch in order to provide the aggregation of addresses on a per switch basis. By providing for routing information within the address, the MOOSE switch allows for traffic to be routed between MOOSE switches, allowing for better utilisation of network links.

This alleviates the problems associated with the deployment of RSTP, and also allows smaller state tables as each switch only needs to know about the hosts directly attached to it and the other switches, resulting in a significantly lower size of state table.

MOOSE addresses are assigned by using the locally administered portion of the Ethernet address space. By using this section of the address space, none of the addresses will conflict with any manufactured assigned addresses, which are set as universly administered. Each address is also marked as a unicast address.

The MOOSE address is then constructed in two separate parts. The first is the switch identifier, which is the switch to which the host is directly attached. The second part is the host identifier. This gives MOOSE addresses a hierarchical nature, allowing routing based on the identifier, as well as allowing aggregation of host addresses under one switch.

Each host is assigned a single host identifier, and when the switch receives traffic from the host, it rewrites the source address to contain the MOOSE address of the host. For example, when a host is attached to a switch with a MOOSE address of 02:00:01:00:00:00, it will be assigned a MOOSE address such as 02:00:01:00:00:01 and any traffic it sends will be rewritten to this address.

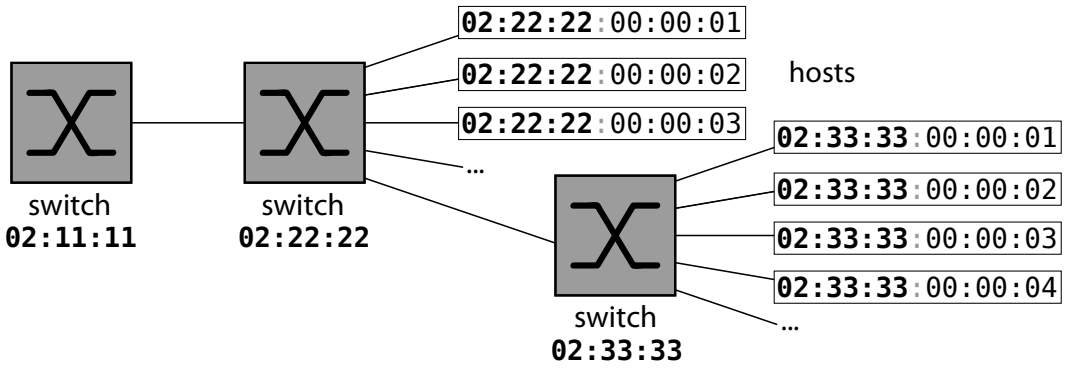

Figure 2.1: MOOSE Switch and Host Hierarchy, from [25]

When traffic is received by the switch at this address, it will be rewritten back to the host's MAC address and sent to the host. If another host is attached, it will be assigned 02:00:01:00:00:02 and so forth. A host attached to a different switch on the same subnet, might be assigned to 02:00:02:00:00:01. This heirarchy is shown in Figure 2.1

In the state table, a MOOSE switch contains a mapping between host identifiers and MAC addresses and ports, and a separate state table mapping between switch identifiers and ports. By aggregating all of a switch's hosts under a single entry in the state table, a vast reduction in size is achieved.

When a packet enters the switch, if it contains a normal Ethernet address, it is rewritten to a MOOSE address and this mapping is stored in the table. It is then directed using the host identifier for local traffic, that is hosts directly attached to the switch, and the switch identifier for remote traffic, i.e. traffic where the host is attached to a different MOOSE switch. This process is shown in Figure 2.2.

2.1.4 Simulation

Due to the large number of hosts, switches and links I will be aiming to simulate to demonstrate the weaknesses of Ethernet, and how it compares to MOOSE, I will be using simulation instead of implementing the topology physically. While simulators pose an enormous advantage in terms of the time it takes to change the topology, implement new protocols and explore larger topologies, there are a number of disadvantages, namely that as it is a simulation, it takes time to program a simulator, and this may approach the difficulty in creating a real network, execution can be computationally expensive, and statistical analysis of the simulation in order to verify the results can be difficult.

As discussed in the introduction, there are a number of different type of simulators - the main distinction being a discrete event simulation or a continuous event simulation. The most suitable for performing network simulation of a

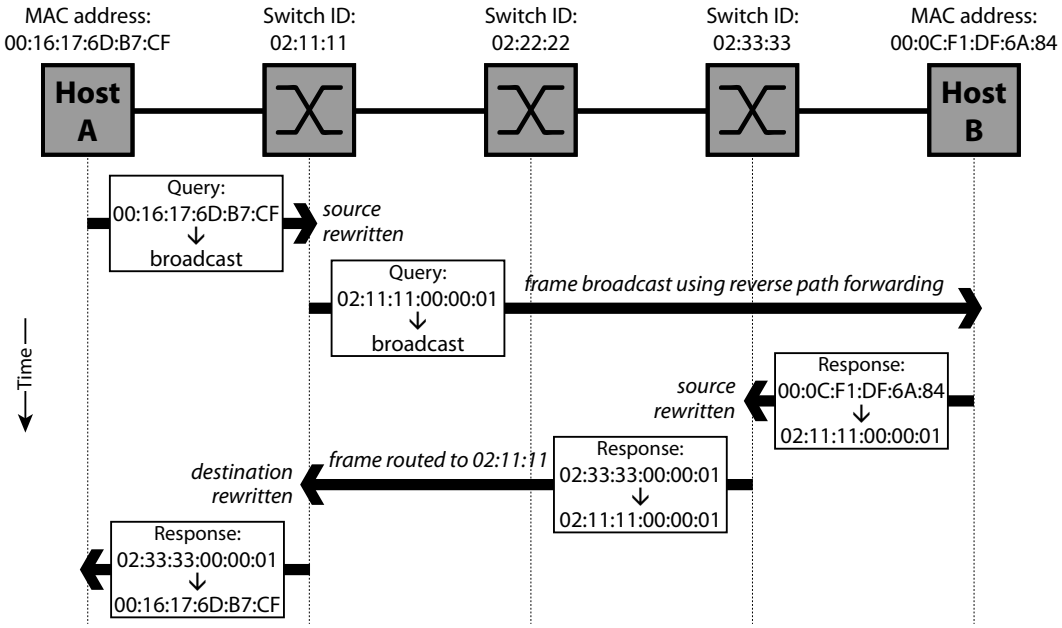

Figure 2.2: MOOSE Packet Rewriting Sequence, from [25]

packet based architecture is a discrete event simulator, as each movement of a packet may be viewed of as an event in the system.

In order to ensure that the goals of the project are achievable, and the results are repeatable and useful to others, I will be building upon a existing network simulator. Some time was spent investigating the different network simulators available. A number of differnet simulators are used, though the most popular series is ns - network simulator - which is available as ns2 and ns3.

ns2

ns2 is a discrete event network simulator written for research purposes, and began as a variant of the REAL network simulator in 1989. It has been supported by a number of different institutions since, and has evolved to become one of the most popular network simulators used in network research. ns2 is written in C++, with network scripts written in Tcl. Most of the work in ns is devoted to network layer and above protocols, with the data link layer protocol being largely simplified as a stochastic model underneath the network layer.

Tcl, Tool Command Language, is a scripting language specifically designed for customising and configuring the running of large projects by embedding a Tcl interpreter in the application. ns2 uses a extended, non standard, version of this OTcl which provides support for object orientated programming.

ns3

ns3 is a complete rewrite of the ns network simulator, again in C++, with Python bindings instead of Tcl. It began in 2006 as a new software development effort intended to focus on improving the software design, architecture, integration and modularity of the existing ns codebase. It was designed as a four year project to produce the next version of ns. It was based on the design of GTNets and is built using the Python based waf build program.

ns3 includes a representative implementation of Ethernet. It includes the necessary support structures to resemble both LLC, Logical Link Control, and DIX, DEC, Intel and Xerox, the three major participants in the design, Ethernet frame types. However, it is only a CSMA, Carrier Sense Multiple Access implementation, as it provides no modelling of the physical layer. As I will be simulating fully switched networks using full duplex links, the lack of Collision Detection will not affect the results I collect.

ns3 includes a sample Ethernet Bridge implementation, however this is not 802.1D compliant as it is lacking support for a Spanning Tree protocol, it simply filters and floods packets.

ns3 allows the use of Python to control the simulation. However in ns3 Python is designed as a optional part of the interface to ns3, and it is perfectly possible to run large simulations written entirely in C++. By solely using C++, writing applications which link with libns3, the application can scale to far more hosts, and doesn't need a heavy binding which is required when using Python, or when using Tcl with ns2.

Unlike ns2, ns3 is in active development and is considered the natural successor to ns2, once the remaining protocols have been implemented for ns3.

2.1.5 Conclusion

While the original proposal indicated that the project would be written for ns2, after research into the different simulators, I decided that it would advantageous to utilise ns3 instead. I originally chose ns2 as it was the apparent defacto standard for network simulation. However, due to its age, and the improved data link layer support in ns3, I have decided to use ns3 instead. This will improve the longevity of the results of my project, as they will be able to be used with the new simulator.

ns3 will also improve the quality of the code I produce, as I will be able to follow the improved software architecture provided, which will make it easier to debug the code I write, and verify that it is performing correctly.

2.2 Strategy

In order to ensure the project was a success, certain development decisions were made.

2.2.1 Software Development Methodology

I decided to use a iterative development method, in order to maintain quality of the code, and to ensure the code was completed on time and with the necessary features. This was done by iteratively designing a new feature in accordance with the overall aim, implementing the feature and then testing it. This cycle was repeated until all the necessary features were implemented. This enforced encapsulation of a feature with a single element of code, providing separation of concerns. It also ensured that each feature was built on a stable base. On each iteration of the code, all the tests were run to ensure that no regression had occured. These tests provide a good set of debugging features which ensured that refactoring the code did not impact code quality.

2.2.2 Source Control

In order to version and backup my code, I used a distributed version control system, Git. Git allowed me to maintain multiple copies of my code, review changes and manage the code effectively. My git repository was held on both my desktop and a laptop, and a copy of the repository was pushed to my server hosted offsite, and a commercial service GitHub [14] which provides hosting for Git repositories. It also provides a good interface to view changes between different branches of the code.

In addition to my code, documentation and test routines were held in separate git repositories, which were again held in multiple places to protect against failure.

Chapter 3

Implementation

3.1 System Design

The system is structured into three different phases, each of which performs a different task. These are generation, simulation and analysis. These are connected together using a series of file formats. This section describes the three phases and how they are structured and the file formats used.

3.1.1 Generation

Due to the large number of hosts under simulation and the hetrogeneous nature of local area network topologies, it was decided that the best way to proceed would be to dynamically generate different network topologies in order to send the data over. The main output of the Generation phase is a topology file which details the number of hosts, number of bridges and the connections between them.

The core component of the generation phase of the Topology Helpers. These are based on the Factory design pattern, and each helper creates a different kind of network topology. The topology created is represented in the Topology class, which contains the number of hosts in the topology, the number of bridges, a map between hosts and bridges (since each host may only connect to one bridge) and a set data structure containing pairs of bridge to bridge links. This is a totally ordered list of bridge to bridge links where order in the pair is unimportant as the links are bidirectional.

The following types of topology are currently implemented:

- Tree
- Mesh
- Cube
- Torus

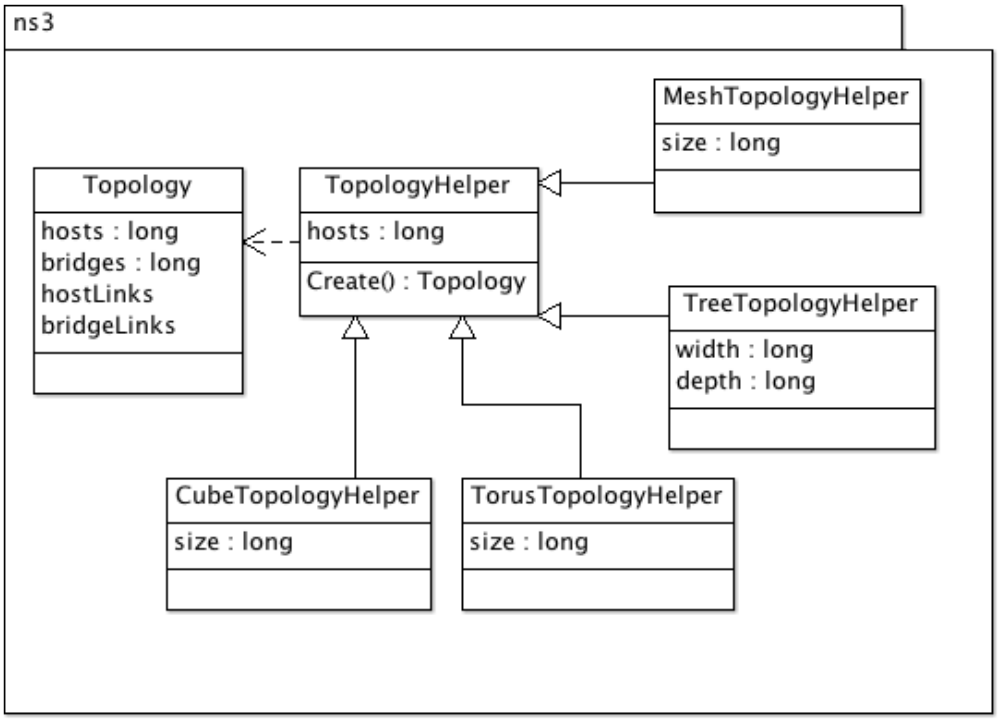

Figure 3.1: Class diagram of components involved in generation of a topology.

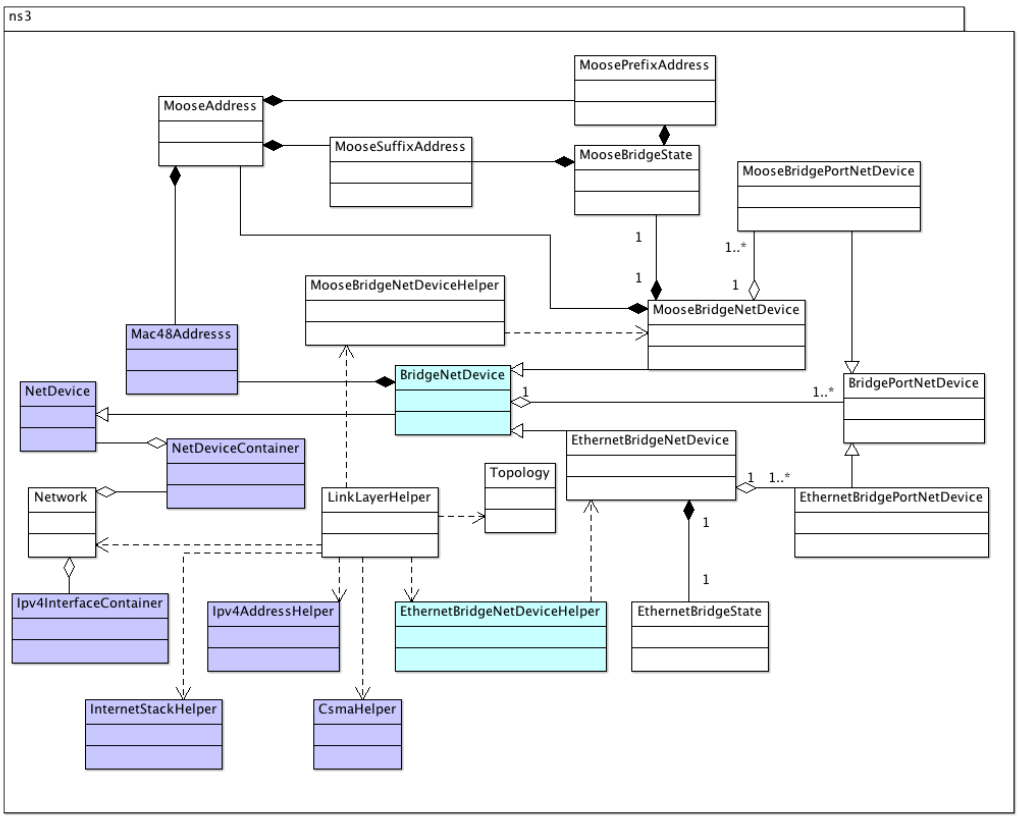

Figure 3.2: Class diagram of components created for simulation. Classes shaded in dark blue are standard as part of ns3. Classes shaded in light blue are shipped with ns3 but have been heavily modified. Classes in white are new.

Each topology type takes a number of parameters and outputs a topology as a result. This topology is then stored in a file. The topology types were chosen as they provide a sample of different topologies being considered in data centers, which is the suggested target for MOOSE.

3.1.2 Simulation

Once the topology has been determined, either via generation from the Generation phase, or from manually creating a topology file, the simulation phase can begin. This is the core phase of the project and responsible for most of the work. The simulation phase uses the ns3 simulator as a core component, with the necessary components to perform Ethernet and MOOSE simulation built on top of it.

Network

The first stage is to setup up the network. This is done by reading in a topology file and converting it into a series of ns3 objects. These are then stored in the Network class. This is done by loading the topology file into a Topology object, and then passing it to the Link Layer Helper. This also takes whether the simulation should be done using MOOSE or Ethernet and whether dynamic or static routing should take place.

First it creates a node to represent each of the hosts and bridges in the topology. It then creates the Ethernet links between the hosts and bridges, and the links between the bridges as specified in the topology file. Each Ethernet link is Gigabit, with a 20 nanosecond delay.

Once it has done this, it configures the routing. Early in the project, I decided to allow two different forms of routing, static and dynamic.

Static routing is done at setup time by running an algorithm across all the nodes to determine the network map. In the case of Ethernet it runs a static spanning tree algorithm, whereas in MOOSE it runs a all pairs shortest path algorithm by running Dijkstra's algorithm on each node. It can do this more efficiently than running Johnson's algorithm as all the path costs are known to be positive. This is implemented using the Boost Graph library implementation of Dijkstra's algorithm [26].

Dynamic routing on the other hand relies on the interchange of BPDU - Bridge Protocol Data Units for Ethernet as outlined in [12] for the Rapid Spanning Tree protocol. MOOSE however can use a routing protocol - OSPFM (Open Shortest Path First for MOOSE), an implementation of OSPF [19], is suggested in [29]. This happens during the running of the simulation.

Once the routing has been configured, the Link Layer Helper initialises a BridgeNetDevice on each bridge node in the case of Ethernet, or a MooseBridgeNetDevice in the case of MOOSE. This represents a switch and contains the implementation of the Link Layer protocol on the switch. The BridgeNetDevice and MooseBridgeNetDevice objects are created through the use of another factory class, BridgeNetDeviceHelper and MooseBridgeNetDeviceHelper respectively. These are responsible for creating the bridge and ports on the bridge.

A pre-existing BridgeNetDevice implementation existed in ns3 for Ethernet; it was incomplete, with no support for RSTP (Rapid Spanning Tree Protocol). It also had no limit on the size of the state table, and was implemented in a single class. I decided to split it up into several different classes to separate the different concerns. It now uses a BridgeNetDevice, with each port on the Bridge represented as a number of BridgePortNetDevice objects which controls the reception and transmission of packets on each port, and a BridgeState class which holds the state table for the Bridge. By performing this separation of concerns, the implementation ensures that each concern is handled efficiently.

A similar approach was taken to implementing the MooseBridgeNetDevice,

which again has a number of MooseBridgePortNetDevice objects to represent each port on the bridge, and a MooseBridgeState object to hold the bridge's state.

Along side these core classes, a number of different types were created to hold the MOOSE data required. These were MooseAddress, MoosePrefixAddress and MooseSuffixAddress, which hold the MOOSE Address components required to implement MOOSE. The former can readily be convered to and from a Mac48Address, ns3's implementation of a MAC Address, which allows it to be used to interogate MAC addresses. MooseAddress gives three types of MAC Address, HOST which is used for a globally administered MAC Address, which is used by normal Ethernet devices, MOOSE which represents an address assigned by a MOOSE switch and MULTICAST which is a MAC address used for multicast/broadcast, and thus not translated to a MOOSE address.

The final stage of network setup is to initialise the hosts. Each hosts is given a standard IPv4 stack, complete with IP, ARP, ICMP, UDP and TCP/IP. This is the base from which we simulate packets across the network

Data

The next stage is to setup the data streams across the network. This is setup based on a file input which details which hosts to send what data to and from which hosts. The data file lists the time to start sending data, the host to send from, the host to send to, and the number of packets to send. This flexible format allows the simulation to simulate a large variety of different traffic scenarios.

Each UDP packet is sent from a UdpClient installed on the sender for each set of packets, to a UdpServer which is installed once per recipient, on port 9, the standard port for discarding packets [24].

Once the file has been read in, the UdpClient and UdpServers are installed using the ns3 Helper classes designed for this purpose.

Tracing

Once this is setup we enable tracing on each of the nodes. Two types of tracing are done. The first is a ASCII trace of all the Ethernet links. This notes enqueue and dequeue operations on each of the Ethernet queues, dropped packets and reception of packets by a device. This is human readable, and can also be parsed by programs to generate statistics on the network.

The second is a PCAP - Packet Capture - file format trace, where one file is generated per link. Each file details the packets transmitted across that link. PCAP files are useful as they are widely used by packet trace programs such as Wireshark. They can be loaded into Wireshark to see what would have been seen by a packet capture software running on that host, verifying that the other traces are correct.

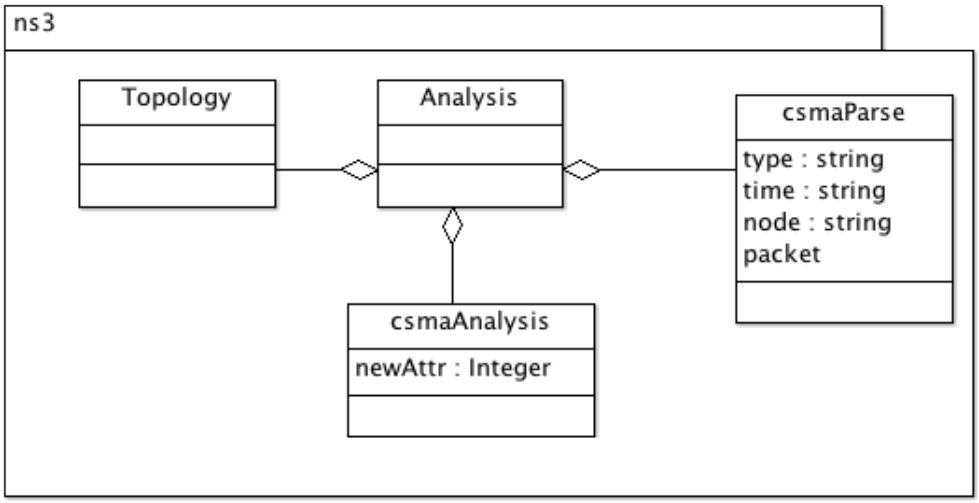

Figure 3.3: Class diagram showing components involved in analysis of a simulation.

Run

Finally, once the setup is complete and the trace files are connected to the correct sources, the simulation is run to gather the date. When the simulation is finished, the program captures the state data in each bridge and outputs it in a common file format for later analysis.

3.1.3 Analysis

In order to get useful data, the files emitted from the simulation stage must be analysed and compared. The primary resource for this stage is the CSMA ASCII trace file which lists each time a packet is enqueued onto a Ethernet transmission queue, when it is dequeued, if it is dropped and when it is received by a node. It lists the time of each action, the node it was operated on and the packet headers for the packet in question.

By analysing this file we can gain an insight into how efficient the routing protocols are, how long each packet took to traverse the network and other simulation metrics.

The analysis program parses the CSMA file and collects statistics on the packets observed before outputting them to a file. These include the number enqueued, dequeued, recieved and dropped. The packets are classified into ARP requests, ARP responses and UDP packets. Counts are also kept for unicast and broadcast packets, as well as a total. It also looks at the state table, and counts the number of entries for each table, for each bridge.

It also creates a graph displaying the nodes in the network topology.

3.1.4 File Formats

In order to pass data between the different sections of the project, a number of different file formats have been devised. These are all designed to be editable using a text editor in order to manually verify the contents. The exact layout of the files is detailed in the appendix.

Each file begins with a magic marker, "ns-moose", for identification, followed by the file type, and file type version. This allows multiple file types to be defined and allows the system to check the correct file has been input. It also allows the file types to be changed in a later version of the code.

Topology

The topology file details the graph of the network. Specifically it lists the number of hosts in the network, the number of bridges and the links between them. When describing the links between switches, the switches are zero indexed, with the hosts following afterwards. When the file is interpreted by the Topology class it ensures that each hosts can only link to one bridge.

Data

The data file details the data sent across the network. After the file header, it lists in order, the sets of packets to be transferred across the network, The first field contains the time to start, interpreted as a double. After this follows descriptors containing the hosts (zero indexed) to send to and from. Finally each section has the number of packets to send across the link.

State

The state file details the data in the state tables on the bridges after the simulation has finished. After a header, including a magic, file type, 3, and the file version, it lists for each bridge the bridge address and the contents of the state tables.

For MOOSE bridges, this includes the MOOSE address, the MAC address, and then the contents of the two state tables, host and switch. The host table contains MAC address, MOOSE suffix, Port identifier and timestamp. For the switch table it contains the MOOSE prefix, the Port identifier and timestamp.

For MAC bridges, this includes the bridge MAC address, and then the contents of the stable table. Each entry includes the MAC address, Port identifier and timestamp.

Trace

The two trace file formats used are a ASCII format included with ns3, as described in the ns3 manual and the included PCAP format as used notably by tcpdump and Wireshark. A ASCII trace and PCAP traces for each link can be created for each run of the simulator.

3.2 Core Algorithms and Data Structures

3.2.1 Topology Representation

The topology is represented internally with a a std::map between hosts and bridges, since each host may only appear on one bridge, and a std::set of pairs which contains bridge to bridge links with a custom comparator. The custom comparator is designed in order to give a total order to links, without caring about order, as links are bidirectional. Thus $< 1, 2 >$ is treated equal to $< 2, 1 >$. The order defined is that $< 1, 2 >< < 1, 3 >< < 4, 1 >< < 3, 4 >$, in other words, the lower number is used for the primary sort, and the larger one as a secondary sort. These provided adequate performance for the purpose. A total order is required so that the set can ensure that each link is only stored once. The simplest way to do this is to provide a total order, and then a binary search can ensure it is unique.

3.2.2 MOOSE State Tables

In order to hold the data required for the MOOSE implementation a series of tables were designed. These are implemented using the std::map data structure. This is used as it provides fast, $O(log(n))$, lookup speed. This is important for the state table in which data is fetch by a predefined index.

The first is the Prefix table. This handles MOOSE traffic which is remote to this bridge. This provides a Prefix to Port and Expiration Time mapping. This is updated when traffic from that bridge is arriving at the bridge, the switch stores a bridge to port mapping. This will also be updated by the routing protocol. When data is to be sent to the specified prefix, the port is looked up, and after making sure it is still valid, the packet is sent on that port.

The second is the Suffix table. This maps MOOSE Suffix Addresses (i.e. the host part) to a Ethernet address and a Port.. This is indexed by both Ethernet address (48 bit MAC Address) and the suffix. The former index is used when converting incoming packets to the correct address, the later when converting incoming packets from the host, to the correct MOOSE address. This is done by providing one std::map which maps between Ethernet address and a structure containing the suffix, the port and the MAC address, and another std::map which maps between suffix and a reference to this structure. When an entry is added

or removed to one map, the same takes place in the other map to ensure they remain in sync.

3.2.3 Reverse Path Forwarding

To prevent flooding causing network starvation in Ethernet, the system converts the graph to a spanning tree. While this approach could be used in MOOSE, a different approach is prefered, which also allows routing to take place. This is a technique derived from implementing multicasting in IP networks without resorting to TTL - Time To Live - expiry.

Reverse path forwarding works by only forwarding packets which arrive on a port which they would be sent on. E.g, if a packet from Switch 6 is recieved by switch 4, it is only transmitted on if switch 4 would send packets to switch 6 on that interface. This creates a natural broadcast prevention by creating a implicit tree from the root switch of the broadcast.

In my implementation this is used for all broadcast packets to prevent flooding causing network degradation. Broadcast packets are important as they are used to implement ARP, the Address Resolution Protocol which converts IP addresses into hardware addresses.

3.2.4 Static Spanning Tree

The static spanning tree algorithm is used to implement RSTP in the setup phase. By doing it here, I was able to check the network was performing correctly, before I had implemented RSTP.

The spanning tree algorithm is implemented using a priority queue and two maps. One map takes pairs of bridges and states whether they are part of the spanning tree. The second looks at bridges and checks whether they have been included in the tree. The queue contains a list of links to check.

The algorithm executes in the following way.

1. The root bridge with the smallest MAC address (in my implementation, this is always numbered 0, as switch MAC address are assigned increasing from 00:00:00:00:00:00) is chosen.

2. The bridge is marked as part of the tree, and each of its links are added to the queue with the cost of traversing the link as the weight.

3. The first link is then removed from the queue. If the bridge it points to is already in the tree, then it is discarded.

4. Otherwise, the node it points to is added to the tree, and all that bridge's links are added to the end of the queue, with the weight being the cost of traversal from the root bridge.

5. The next links is removed from the queue and the algorithm repeats. The algorithm terminates when either there are no more links on the queue, or all the bridges are part of the map.

The algorithm assumes that all the nodes are connected, and that all links are bidirectional, which is the case is the networks I am testing. In the case I am testing the priority queue is also replaceable by a normal queue under the condition that all the links have equal cost traversal, which is also the case.

3.2.5 Static Routing

The MOOSE implementation has a static routing algorithm. This simply consists of insert all the bridges and links into a graph, assigning a weight for each link, in this case, since all the links are the same, they are all given a weight of 1, and then running Dijkstra's algorithm [7] across all pairs of nodes.

While Johnson's algorithm which returns all-pairs shortest paths would be the usual choice for such a operation, as we know that there are no negative weights, there is no need to run the Bellman-Ford algorithm, which is designed to eliminate such an occurence.

As Dijkstra's algorithm has a complexity of $O(e+nlog(n))$, the static routing has a complexity of $O(ne+n^2log(n))$ since there are n switches to run it across. This is the same as Johnson's algorithm, but faster due to a smaller constant factor.

Chapter 4

Evaluation

4.1 Experiments and Results

The following experiments were carried out on the simulator with a Gigabit Ethernet network, with trials being done with both MOOSE and Ethernet as the Data Link Layer protocol. In each case UDP packets, encapsulated in IP datagrams are sent on Ethernet frames from the hosts.

4.1.1 Operational Validation

The first section of trials was done on the network shown in Figure 4.1. These were done to check the protocols were performing correctly.

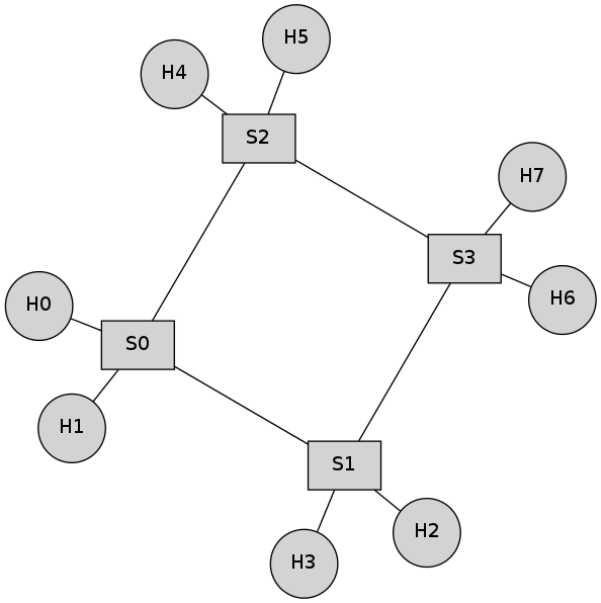

Figure 4.1: Operational Validation Network

In this set of tests, the first host sends a packet, and then a second later, the second host replies.

Local Transmission

The first test shows that the local packets, i.e. packets which only traverse their local switch, are transmitted correctly. In this case, packets are sent between Host 0 and Host 1. In both the Ethernet and MOOSE case the packet traverse the switch and go to the host correctly. In the MOOSE case, the address is rewritten in both directions. In order to facilitate this, the ARP packet is also rewritten, to ensure that the sending host gets the correct hardware address. In addition, observation of the state file produced by the simulation show that both protocols have correctly learned the sending hosts and associated them with the correct addresses.

Remote Transmission

The second test provides further evidence that the switches are operating correctly. In this test, a datagram sent to a host attached to a different switch is performed, in this case, between Host 0 and Host 2. Here we have evidence that the addressing and routing between switches is being performed correctly and that the switch addressing is working correctly in the case of MOOSE. Again, the switch performs ARP and header rewriting on the packets.

Forwarding protocol

Evidence of the effect of the different forwarding systems can be found in the third test, where Host 4 sends a packet to Host 6. In the case of Ethernet, this travels via Switch 2, Switch 0, Switch 1 and then Switch 3, before arriving at Host 6, and the reverse for 6 to 4. This is because the spanning tree protocol turns the graph into a tree by disabling the link between Switch 2 and Switch 5. In the case of MOOSE however, it travels from Host 4, to Switch 2 then Switch 3 then Host 6, and the reverse for 6 to 4, as MOOSE uses a more advanced forwarding protocol. For the unicast packet this results in a considerable increase in the number of Ethernet datagrams required.

4.1.2 Efficiency of Packet Forwarding

In this test, we are looking at the efficiency of the packet forwarding system.

We can measure this by counting the total number of Ethernet frames for a packet to go between two hosts. With a inefficient forwarding system it will require more frames than in an efficient forwarding system.

I will use this to compare Ethernet's system, where the forwarding protocol keeps a state table storing which host is off which port, which is used with the

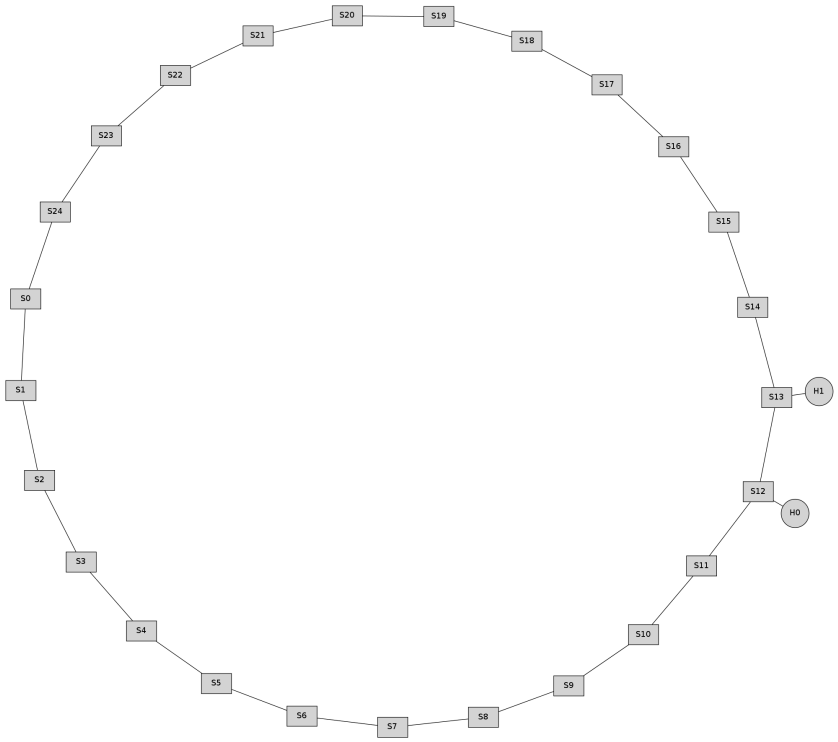

Figure 4.2: Unicast Routing Network

spanning tree to forward packets, to the MOOSE system, which uses a Link State routing protocol to keep track of switches, as well as learning the address of local hosts.

We need to look at two different types of addressing as well. Packets in Ethernet can either be broadcast, or unicast, so both mechanisms will be studied.

Unicast

Here we have a ring network topology, shown in Figure 4.2 with 25 switches in the ring. Switch 12 and 13 each have a host attached, 0 and 1 respectively. Host 0 sends a packet to host 1, and then host 1 sends a packet back to host 0.

The network has been designed to demonstrate the deficiencies of the Ethernet protocol. This has been done by constructing the spanning tree such that the the hosts are at the bottom of different branches of the tree, as shown in Figure 4.3. As such, the direct link between switches 12 and 13 is disabled by the spanning tree protocol, and the packets must traverse the entire ring. While this is unlikely, it is a possible scenario, as if all hosts on a ring are communicating with each other, occasionally the worst case will be hit.

Due to this, the Ethernet protocol performs very badly here, requiring 26 frames in each direction for the UDP packet and the ARP Response packet - giving a total of 52 unicast UDP packets, and 52 unicast ARP packets, combining

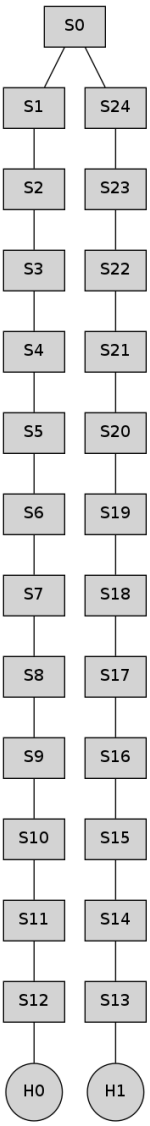

Figure 4.3: Unicast Routing Spanning Tree

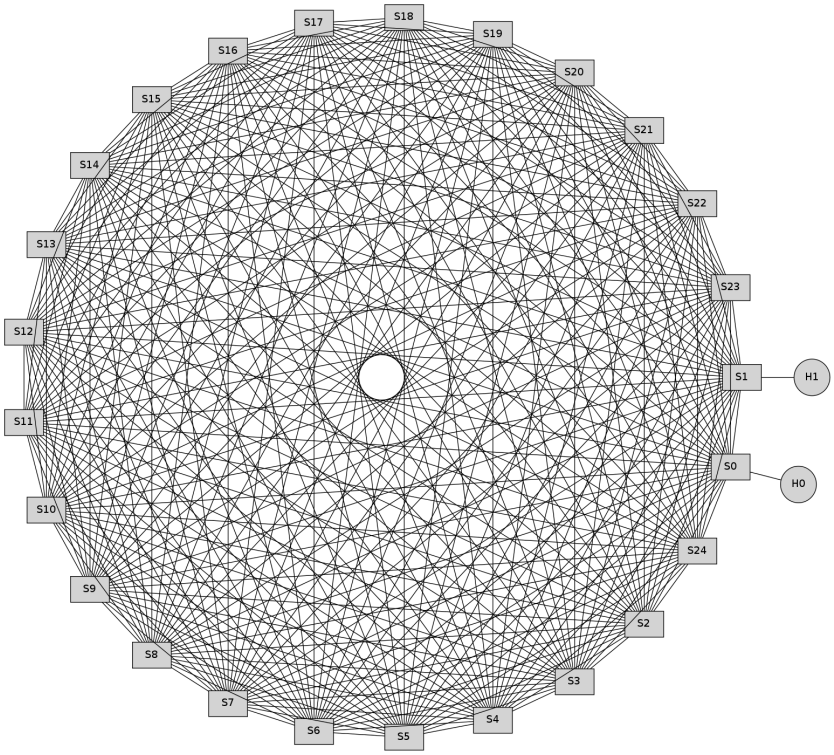

Figure 4.4: Broadcast Network

to make a total of 104 unicast frames.

MOOSE on the other hand, uses a routing protocol, which allows it to perform much better. As the unicast packets only need to traverse between switch 12 and switch 13, the number of frames which have to be unicast is vastly reduced. Only a total of 12 unicast frames are sent, with 3 for each UDP packet and 3 for each ARP Response.

Broadcast

Broadcast is the mechanism by which each host on the network is delivered a copy of the packet. It is primarily used for ARP, the Address Resolution Protocol, which resolves IP addresses to hardware addresses, in this case, Ethernet MAC addresses.

Here we have a mesh network topology with 25 switches in the mesh, shown in Figure 4.4. Switch 12 and 13, like in the unicast example, have a host attached, 0 and 1 respectively. Host 0 sends a packet to host 1, and then host 1 sends a packet back to host 0. Before the transmission of a packet, the network must resolve the address of the host, by using an ARP Request, causing a broadcast packet to be emitted.

MOOSE performs poorly when packets are broadcast. This is due to the

selection of Reverse Path Forwarding to prevent broadcast storms. Broadcast storms are scenarios in which packets that have been set as broadcast loop around a series of links, causing network saturation, reducing the effective utility of the network. In order to prevent this, some mechanism must be used to prevent this happening by culling duplicate frames.

Reverse Path Forwarding works by detecting that a frame did not come from the most efficient route from source, and thus it is a duplicate, which allows it to be safely discarded. In MOOSE this works by looking at the source and port combination with which a frame arrives at the switch and checking in the switch that a frame sent to that switch would be sent on the same port. If it is, then the frame is flooded on all other ports, in the case of broadcast, otherwise it is dropped. This means that each frame is forwarded to one hop more than it would if a spanning tree existed, as it is only detected as a inefficient route on arrival at a switch, not before it is flooded.

In a fully connected mesh network, this causes poor broadcast performance, as after the first switch floods the frame to all other ports, each switch which recieves it will have received it on the most efficient port and will flood the frame again. However, all of the switches to which it floods the frame will already have recieved the frame and thus it will be a duplicate.

As a result of this, the MOOSE protocol performs especially badly here, requiring 1156 broadcast frames to perform the simulation, 578 from each transmission. One frame is from the host to the switch, then 24 frames to each of the connected switches, followed by each of those switches sending another 24 frames, plus an additional frame from one switch to the host.

In comparison, Ethernet, using the Rapid Spanning Tree Protocol (RSTP), uses only 52 broadcast frames, which is far lower. This is 26 frames for each broadcast of a ARP request, with one from the host to the nearest switch, 24 switch to switch and then one to destination host.

4.1.3 State Table Size

One of the major predicted advantage of the MOOSE protocol is that of reduced state table sizes. Here I will show two different cases which show the difference between MOOSE and Ethernet.

Ethernet

Ethernet performs better than MOOSE when the majority of switches have no connected hosts. This is because Ethernet only holds state for hosts, whereas MOOSE holds state for each switch and each local host.

In this situation 15 switches have been connected in a row, with a single host connected to the first and last switch.

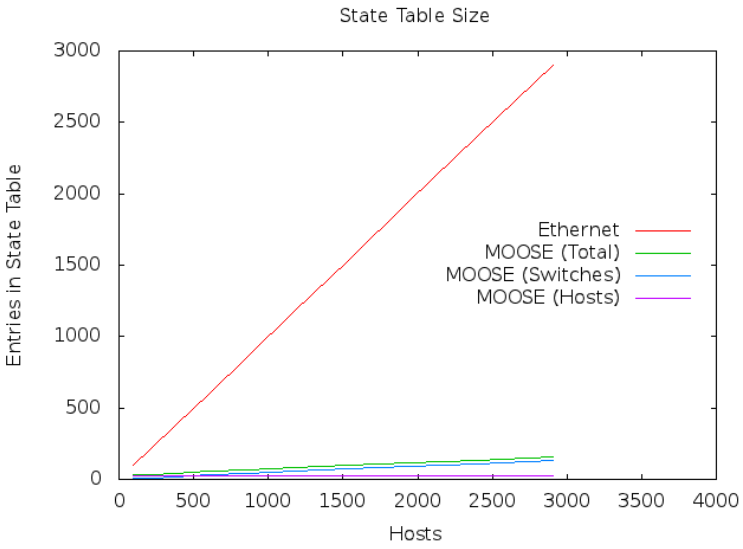

Figure 4.5: Rate of growth of state tables

Here the Ethernet protocol gives a state table size of 2 in the first and last switch, while MOOSE gives a host table with a size of 1 in both the first and last switches and, in addition, a switch table size of 14 in all of the switches.

MOOSE

MOOSE performs much better than Ethernet when the hosts are distributed across a number of switches.

To demonstrate this, in this scenario, I have used a series of tree topologies. Each tree has three layers of switches, with one switch on the node, and then branching out to n switches on the layer below, with n^2 on the final layer. Each of the final layer of switches has 24 hosts directly attached.

The first host is designated as a sink, while each of the other hosts send two packets to the sink. This ensures that each host broadcasts an ARP query, which will ensure that all the state tables contain the relevant data. Sending a broadcast packet, which causes a hosts address and location to be stored in every Ethernet switch is likely due to the usage of ARP. ARP, which resolves IP addresses to Ethernet addresses, is required to communicate using the IP layer on a local area network. ARP entries expire, and when they have, and a packet is attempted to be sent to the host, they must be refetched by performing a broadcast query.

As n increases, the size of the Ethernet state table grows at the same speed as n, while the MOOSE state table grows as the number of switches grow, which a much lower state of growth. In Figure 4.5, the different rate of growth of the largest state tables is clearly demonstrated. Here the amount of memory required to realise the same topology is greatly increased.

4.2 Limitations of the Simulations Performed

The topoloigies used in the above dissertation have been generated for the purpose of demonstrating certain phonemena. They are uniform - they have all links set to produce the same delay and same data rate. As such they should not be taken to represent the conditions found in a data center, or other network, however, they do provide a useful benchmark for comparing the two protocols and exploring how MOOSE might be deployed advantageously.

The traces used for testing the topologies are very simplistic. They are designed to demonstrate certain phonemena without incurring a high simulation overhead caused by further complexity. While this won't have any effect on the observations made here, it is worth noting that interactions between the observations made and application specific traffic should be explored further.

4.3 Possible Improvements

Testing against real world traces, and incorporating dynamic versions of the routing protocols, and implementing the SAMI protocol [29] would increase the use of the data presented here. The SAMI protocol includes the ability to dynamically assign MOOSE addresses, and adding host mobility would cause additional overhead in the use of MOOSE.

4.4 MOOSE

I propose that MOOSE should be modified to incorporate a spanning tree protocol instead off relying on Reverse Path Forwarding, as this simulation has shown that it performs poorly under broadcast, especially in well connected networks. This would require the link state information to be interpreted in order to perform this, however it should provide little further overhead.

Chapter 5

Conclusion

5.1 Future Work

Based on the work outlined in this dissertation, there are a number of areas of possible enhancement to the implementation described here.

5.1.1 Dynamic Routing

At the moment this dissertation relies on static routing, based on a each switch being told of its neighbours. While in some situations this is a possibility, by directly managing the switch, it would be realistic to have the switches dynamically configure themselves.

In terms of Ethernet, this would be done by implementing the Rapid Spanning Tree Protocol described in [12].

For MOOSE, I would suggest that this should be done by implementing the SAMI protocol outlined in [29]. Once specified, this could be incorporated into the simulation to provide additional metrics for comparison.

5.1.2 Spanning Tree for MOOSE

By using the link state data further, it would be possible for the MOOSE protocol to be smarter about how it forwards broadcast packets. Currently these are handled by using Reverse Path Forwarding, having, as shown in the evaluation this causes additional traffic, that does not occur under RSTP, as in RSTP, each broadcast packet is sent along more links than are necessary. However, the MOOSE switches could potentially use a spanning tree of the switches to provide for a minimum spanning tree implementation.

5.2 Summary

In this dissertation I have successfully created working implementations of MOOSE, and Ethernet for a modern network simulator, ns3. This has involved teaching myself a large number of technologies: MOOSE, the workings behind ns3 and the build system it uses, the detailed implementation of Ethernet, to support knowledge I have learned elsewhere in the Tripos, such as in Principles of Communication. I have expanded on my knowledge of using C++ to create a system of significant size which can be used by other people for research in the future.

The MOOSE implementation is fully functional, maintaining state information, rewriting packets, including ARP requests and responses. I have designed experiments to show that my switch works under a variety of conditions and have included information to show the differences between the protocols. These included data on the different properties of the routing protocols, with MOOSE outperforming Ethernet in terms of unicast routing, while Ethernet performs better than MOOSE under broadcast.

I have also shown how the different protocols cause different sizes of forwarding tables in the switches, with MOOSE performing drastically better than Ethernet under some situations, while Ethernet performs marginally better than MOOSE in others.

I have also made suggestions for how the MOOSE protocol may aim to overcome some of the limitations identified, by extending the routing protocol to improve broadcast performance.

The insight I have gained into the MOOSE protocol and ns3 has allowed me to suggest further developments for the protocol, beyond the scope of this dissertation. I also aim to have the code produced in this dissertation pushed upstream into ns3 to allow other developers to expand on the work outlined here.

Bibliography

[1] 3Com Corporation. Switch 5500G 10/100/1000 family data sheet. `http://www.lctinc.net/downloads/3com_5500_10-100-1000_fam_ds.pdf`.

[2] N. Abramson. The ALOHA system - another alternative for computer communications. In *Fall 1970 AFIPS Computer Conference*, pages 281–285, 1970.

[3] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proc. SOSP*, pages 164–177, New York, NY, USA, 2003. ACM.

[4] R. Braden. Requirements for Internet Hosts - Communication Layers. RFC 1122 (Standard), October 1989. `http://www.ietf.org/rfc/rfc1122`.

[5] Vinton Cerf and Robert Kahn. A protocol for packet network interconnection. *IEEE Transactions on Communications*, 22:637–648, March 1974.

[6] S. Deering and R. Hinden. Internet Protocol, Version 6 (IPv6) Specification. RFC 1883 (Proposed Standard), December 1995. `http://www.ietf.org/rfc/rfc1883`.

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

[8] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. `http://www.ietf.org/rfc/rfc2616`.

[9] ICANN. Available Pool of Unallocated IPv4 Internet Addresses Now Completely Emptied. `http://www.icann.org/en/news/releases/release-03feb11-en.pdf`.

[10] IEEE Computer Society. Std 802.3u media access control (MAC) parameters, physical layer, medium attachment units, and repeater for 100 mb/s operation, type 100BASE-T. Technical report, IEEE, New York, 1995.

[11] IEEE Computer Society. Std 802.3ab access method and physical layer specifications - physical layer parameters and specifications for 1000 mb/s operation over 4 pair of category 5 balanced copper cabling, type 1000base-t. Technical report, IEEE, New York, 1999.

[12] IEEE Computer Society. Std 802.1D: Standard for local and metropolitan area networks: Media access control (MAC) bridges, 2004.

[13] Facebook Inc. Open Compute Project Server Specifications. `http://www.opencompute.org/specs/Open_Computer_project_Server_Chassis_Triplet_v1.0.pdf`.

[14] GitHub Inc. GitHub - Social Coding. `http://www.github.com`.

[15] C. Kim, M. Caesar, and J.Rexford. Floodless in SEATTLE: a scalable Ethernet architecture for large enterprises. In *Proc. SIGCOMM*, pages 3–14, 2008.

[16] Changhoon Kim, Matthew Caesar, and Jennifer Rexford. SEATTLE: A Scalable Ethernet Architecture for Large Enterprises. *ACM Transactions on Computer Systems*, 29(1), February 2011.

[17] Robert M. Metcalfe and David R. Boggs. Ethernet: distributed packet switching for local computer networks. *Commun. ACM*, 19(7):395–404, 1976.

[18] P.V. Mockapetris. Domain names - implementation and specification. RFC 1035 (Standard), November 1987. `http://www.ietf.org/rfc/rfc1035`.

[19] J. Moy. OSPF Version 2. RFC 2328 (Standard), April 1998. `http://www.ietf.org/rfc/rfc2328`.

[20] Andy Myers, Eugene Ng, and Hui Zhang. Rethinking the service model: Scaling ethernet to a million nodes. In *ACM SIGCOMM HotNets*, 2004.

[21] K. Pagiamtzis and A. Sheikholeslami. Content-Addressable Memory (CAM) circuits and architectures: a tutorial and survey. *IEEE Journal of Solid-State Circuits*, 41:712–727, 2006.

[22] J. Postel. User Datagram Protocol. RFC 768 (Standard), August 1980. `http://www.ietf.org/rfc/rfc768`.

[23] J. Postel. Internet Protocol. RFC 791 (Standard), September 1981. `http://www.ietf.org/rfc/rfc791`.

[24] J. Postel. Discard Protocol. RFC 863 (Standard), May 1983. `http://www.ietf.org/rfc/rfc863`.

[25] Malcolm A. Scott, Andrew W. 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.

[26] Jeremy Siek, Lieu-Quan Lee, and Andrew Lumsdaine. The Boost Graph Library (BGL). `http://www.boost.org/doc/libs/1_46_1/libs/graph/doc/index.html`.

[27] Charles Spurgeon. *Ethernet: The Definitive Guide*. O'Reilly Media, 1st edition, 2000.

[28] Daniel Wagner-Hall. A Prototype Implementation of MOOSE on a NetFPGA/OpenFlow/NOX Stack. In *First European NetFPGA Developers' Workshop, Cambridge*, September 2010.

[29] Daniel Wagner-Hall. NetFPGA Implementation of MOOSE. In *Computer Science Tripos Part II, Homerton College, University of Cambridge*, May 2010.

[30] Hubert Zimmermann. OSI reference model — the ISO model of architecture for open systems interconnection. *IEEE Transactions on Communications*, 28(4):425–432, April 1980.

Appendix A

Test Data

These files are from the first simulation, and show sample output from the simulation framework.

A.1 Topology

```
ns-moose
1
1
8
4
0  1
0  2
1  3
2  3
4  0
5  0
6  1
7  1
8  2
9  2
10  3
11  3
```

A.2 Data

```
ns-moose
2
1
1  0  1  1
```

2 1 0 1

A.3 Ethernet

A.3.1 State

```
ns-moose
3
1
1
00:00:00:00:00:02
2
00:00:00:00:00:01
0x20f1740
302000002593ns
00:00:00:00:00:03
0x20f1860
302000012703ns
1
00:00:00:00:00:06
2
00:00:00:00:00:01
0x20f2010
301000001062ns
00:00:00:00:00:03
0x20f2010
302000001062ns
1
00:00:00:00:00:0a
2
00:00:00:00:00:01
0x20f26c0
301000001062ns
00:00:00:00:00:03
0x20f26c0
302000001062ns
1
00:00:00:00:00:0e
2
00:00:00:00:00:01
0x20f2d70
301000001593ns
00:00:00:00:00:03
```

```
0x20f2d70
302000001593ns
```

A.4 Analysis

```
ARP
          Request          +: 22 -: 22 r: 22
          Response         +: 4 -: 4 r: 2
          Total   +: 26 -: 26 r: 24

UDP       +: 4 -: 4 r: 2

Unicast           +: 8 -: 8 r: 4
Broadcast         +: 22 -: 22 r: 22

Total     +: 30 -: 30 r: 26

State Table
        00:00:00:00:00:02
                  Entries         2

        00:00:00:00:00:06
                  Entries         2

        00:00:00:00:00:0a
                  Entries         2

        00:00:00:00:00:0e
                  Entries         2
```

A.5 MOOSE

A.5.1 State

```
ns-moose
3
1
2
00:00:00:00:00:02
02:00:00:00:00:00
3
1
0x1720490
```

9223372036854775807ns
2
0x1720640
9223372036854775807ns
3
0x1720490
9223372036854775807ns
2
00:00:00:00:00:01
1
0
302000002593ns
00:00:00:00:00:03
2
0
302000012703ns
2
00:00:00:00:00:06
02:00:01:00:00:00
3
0
0x1720760
9223372036854775807ns
2
0x1720760
9223372036854775807ns
3
0x1720ea0
9223372036854775807ns
0
2
00:00:00:00:00:0a
02:00:02:00:00:00
3
0
0x1720fc0
9223372036854775807ns
1
0x1720fc0
9223372036854775807ns
3
0x1721700
9223372036854775807ns

```
0
2
00:00:00:00:00:0e
02:00:03:00:00:00
3
0
0x1721820
9223372036854775807ns
1
0x1721820
9223372036854775807ns
2
0x1721f60
9223372036854775807ns
0
```

A.6 Analysis

```
ARP
          Request          +: 26 -: 26 r: 26
          Response         +: 8 -: 8 r: 2
          Total   +: 34 -: 34 r: 28

UDP       +: 8 -: 8 r: 2

Unicast           +: 16 -: 16 r: 4
Broadcast         +: 26 -: 26 r: 26

Total     +: 42 -: 42 r: 30

State Table
        02:00:00:00:00:00
                  Hosts  2
                  Switches        3
                  Total  5

        02:00:01:00:00:00
                  Hosts  0
                  Switches        3
                  Total  3

        02:00:02:00:00:00
                  Hosts  0
```

```
                Switches        3
                Total   3

        02:00:03:00:00:00
                Hosts   0
                Switches        3
                Total   3
```

A.7 Packet Captures

A.7.1 Ethernet

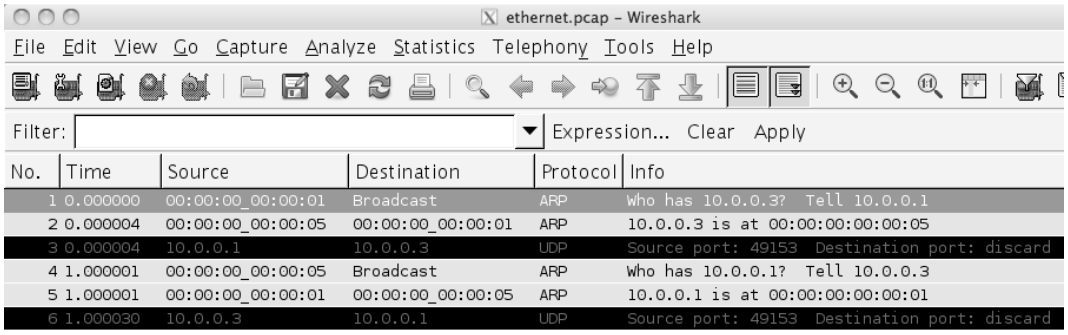

Figure A.1: Wireshark output of simulation under Ethernet

A.7.2 MOOSE

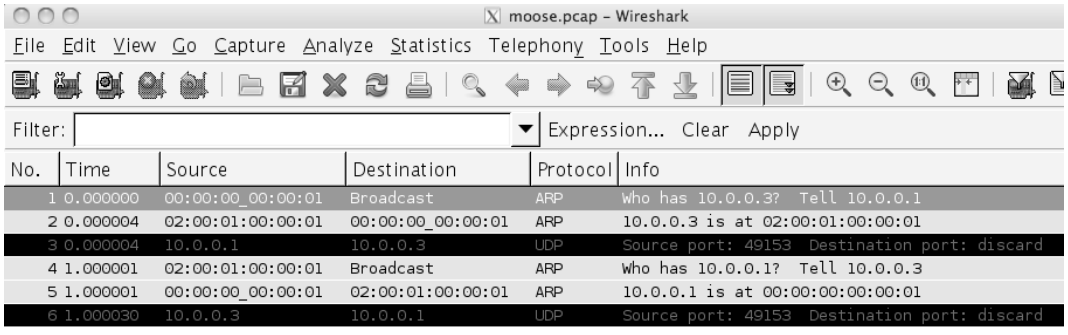

Figure A.2: Wireshark output of simulation under MOOSE

Appendix B

Proposal

Part II Computer Science Project Proposal

Simulation of Data Link Layer Protocols

R. J. Whitehouse, Homerton College

Originator: M. Scott

14 October 2010

Special Resources Required

The use of my own desktop workstation (2.6 GHz Intel Core 2 Q6600, 2 GB RAM and 3 TB Disk).
File space on PWF – 500 Mbytes
File space on my own server – 500 Mbytes
The use of my laptop (2.13 GHz Intel Core 2 Duo, 2 GB RAM and 128 GB Disk).

Project Supervisor: M. Scott

Director of Studies: Dr R. K. Harle and Dr B. Roman

Project Overseers: Dr A. Blackwell & Dr C. Mascolo

Introduction

In local area networks, Ethernet is the dominant data link layer protocol. Many attempts have been made to improve on its perceived weaknesses in terms of large amount of broadcast traffic, large forwarding tables required and poor routing.

My aim is to compare one of these proposals, MOOSE (Multi-level Origin-Organised Scalable Ethernet) with Ethernet in a simulation in order to examine the advantages and disadvantages of such a replacement.

My work will be focussed on providing an implementation of MOOSE in a commonly used network simulator - ns2, preparing a number of simulation scenarios and then generating data in order to allow a comparative analysis. The final stage of my project will be to evaluate the data produce in order to provide a useful comparison between the protocols.

ns2 is a heavily used network simulator used in research. It is available under the GNU General Public License and is written in C++ and Tcl.

The data gathered for comparative analysis will aim to show the differences and similarities between MOOSE and Ethernet. This will consist of, among other metrics, the size of the forwarding tables in the switches in the network, the number of messages sent across all switches and a representation of how many of these are control messages.

The simulation will be done on a mixture of generated data, and anonymized data traces which is publicly available. This is to attempt to ensure that valid metrics are obtained for a variety of different network configurations. Alongside the anonymized data traces, the generated data will be designed to conform to publicly available statistics of network data.

Work that has to be done

The project breaks down into the following main sections:-

1. The exploration of the implementation of Ethernet on ns2 - confirming it passes various tests and analysis of the implementation.

2. The implementation of MOOSE on ns2.

   This will involve modifying the Ethernet protocol provided by ns2 to incorporate MOOSE semantics. This should include switches that are able to accept hosts being attached to them, rewrite packets with Ethernet addresses into ones with MOOSE addresses and vice versa, and participate in a network - including being able to transmit and receive packets from other hosts.

   As an extension, various different routing protocols should be implemented to allow their characteristics with MOOSE to be varied.

3. The design of a number of simulations designed to show the differences between Ethernet and MOOSE.

   MOOSE is designed to perform better on large networks, while Ethernet was originally designed for networks with fewer than a thousand hosts. Several different network topologies should be experimented with in order to show the differences between the protocols. Realistic networks should be used to show the practical applications of both, while extreme network topologies should be designed to test the limits of the design.

4. The comparative analysis of MOOSE and Ethernet using data gathered by simulation.

   A variety of metrics should be gathered - ideally showing all the differences between MOOSE and Ethernet. These should be analysed to see how they would affect real world performance and how relevant the simulation is to potential real world implementations.

   This will also involve using data provided from elsewhere which should be modified into a form acceptable by the simulator. Additional data will also need to be generated and effort should be taken to ensure that this data is realistic.

Evaluating the success of the project

The success of this project will be measured against a number of criteria. These are:

1. The correct implementation of simulated MOOSE switch which performs addressing rewriting and performs correctly in a network involving both other MOOSE switches and other Ethernet switches. This will be tested by simulating the switch inside ns2 and then using a number of unit tests to ensure it is performing correctly.

2. The formation of a number of network topologies which seek to show the difference between Ethernet and MOOSE. This will be evaluated under simulation by showing the difference in a number of metrics.

3. The collection of a number of metrics showing the difference between the protocols, including the following:

   - The size of the state data held by the switch and change in performance characteristics as this reaches a predefined limit.
   - The amount of overhead in terms of control messages relative to the amount of data transfered.
   - The efficiency of the routes used by the protocol.

Difficulties to Overcome

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

- To learn how to implement a protocol in ns2.
- To learn how to program in Tcl.
- To improve the use of git to allow version control of the project.

Starting Point

I have knowledge of how Ethernet works from Digital Communications 1 and I am adept in writing C++ code. I have experience in compiling and building programs in a Linux architecture.

ns2 includes a implementation of Ethernet which I will use as a basis for creating the MOOSE protocol implementation. It also includes several implementations of different routing algorithms.

Resources

In this project, I will be writing a fair amount of code in order to allow ns2 to simulate MOOSE. This code will be version controlled and backed up in order to ensure it is safe from any form of data loss. In order to achieve this I will be using a Ubuntu Linux desktop machine using git. This will allow the code to be version controlled. In order to back it up, I will push my changes in git onto both the PWF servers provided by the University Computing Service, and a separate Ubuntu Linux box that I own away from Cambridge.

The project will also require a dissertation to be written. This will be done on my laptop, which will again be backed up to the PWF and my server, and a external hard drive.

Work Plan

The planned start date is 25th October 2010. Up until this point, I will do preparatory work such as arranging materials, and doing background research. The work will be divided up into fortnightly segments.

- **25th October 2010 - 14th November 2010** This time will be spent doing preparatory work such as learning how to use the ns2 simulator and learning Tcl which will be used in the project.

- **15th Novemer 2010 - 28th November 2010** I will start the implementation of the MOOSE protocol for ns2. In this week I shall explore the current Ethernet implementation and work out what code needs replacing. I shal start implementing the address rewriting.
- **29th November 2010 - 12th December 2010** I will complete the address rewriting for the switches and begin work on the reworking the forwarding table.
- **13th December 2010 - 26th December 2010** By the end of this period, I will have completed the work on the forwarding table and shall have a working MOOSE switch.
- **27th December 2010 - 9th January 2011** During this time, I shall work on implementing extensions identified in MOOSE such as improving the routing protocol and implementing auto-configuration.
- **10th January 2011 - 23rd January 2011** At this point, I will have a working MOOSE switch in ns2. At this point I shall start work on using the switch for simulation. I shall spend this week preparing the publicly available data for testing and designing network topologies.
- **24th Janary 2011 - 6th February 2011** During this fortnight I shall write the progress report and start work on testing the infrastructure to collect metrics. I shall then start inputting data for the simulation.
- **7th February 2011 - 20th February 2011** By the end of this fortnight, I aim to have a complete set of simulation data. This will involve completing the network tests and compiling the metrics gathered.
- **21st February 2011 - 6th March 2011** At this point I will begin the dissertation write up. By the end of this period I aim to have the introduction and preparation chapters complete.
- **7th March 2011 - 20th March 2011** I will continue the dissertation write up. By the end of this period I aim to have the implementation and evaluation chapters complete.
- **21st March 2011 - 3rd April 2011** By the end of this period I will have completed the dissertation write up. This will involve completing the conclusion chapter and ensuring that it is a coherent document.
- **4th April 2011 - 17th April 2011** I shall use this time for making any changes to the dissertation necessary.
- **18th April 2011 - 1st May 2011** I have left this time empty to allow any catchup necessary and to do final finishing touches.

- **2nd May 2011 - 16th May 2011** Final changes made to dissertation. Print and bind report ready to hand in. Remind supervisor to complete report form ready for the deadline.