Low Energy Internet using Software Defined Networking

Ian Beatty-Orr
B.A.I. Engineering
Final Year Project, April 3rd, 2013
Supervisor: Prof. Donal O’Mahony

School of Computer Science and Statistics
O’Reilly Institute, Trinity College, Dublin 2, Ireland
Low Energy Internet using Software Defined Networking

April 3rd, 2013
DECLARATION

I hereby declare that this project is entirely my own work and that it has not been submitted as an exercise for a degree at this or any other university.

________________________________        ___________________________

Ian Beatty-Orr                             April 3rd, 2013
Permission to Lend

I agree that the Library and other agents of the College may lend or copy this thesis upon request.

_____________________________  ________________________________
Ian Beatty-Orr                  April 3rd, 2013
I would like to thank my supervisor Prof. Donal O’Mahony for his continued support throughout this project. He has been a great source of information, ideas and encouragement whenever needed.

I would once again like to thank Prof. O’Mahony as well as my father for taking a look at early drafts of this report and providing feedback.
Abstract

Computer Networks are wasteful in their energy consumption using the same amount of electricity under full load as they do when idle. This report looks at ways to scale network power usage relative to demand while maintaining a reliable service level. Software Defined Networking will be used to monitor network traffic levels and channel this traffic onto the same routes creating a Minimal Energy Network. Testing revealed savings of 41.5% and 70% for maximum and idle load scenarios in small networks. These tests confirm the energy saving to be gained by scaling network power usage relative to demand.
# Table of Contents

1. **Introduction**  7
   1.1. Motivation  7
   1.2. Project Goal  8
   1.3. Outline  9
   1.4. Terminology  10

2. **State of the Art**  10
   2.1. Introduction  10
   2.2. Energy Usage of Today's Networks  11
      2.2.1. Global Energy Usage of the Internet  11
      2.2.2. Energy Usage of the Physical Network  11
   2.3. How Networks Currently Work  11
      2.3.1. Conventional Routing Protocols  13
      2.3.2. Minimum Spanning Tree  14
   2.4. How Software Defined Networking Works  15
      2.4.1. OpenFlow  16
      2.4.2. Operation of OpenFlow  17
      2.4.1. Advantages of OpenFlow and Software Defined Networking  17
      2.4.2. Controllers  19
      2.4.3. Pox  19
   2.5. Test Network  20
      2.5.1. Mininet  21
      2.5.2. Wireshark  22
      2.5.3. Ostinato  22
   2.6. Energy Efficient Networks  23
      2.6.1. Elastic Tree  23
      2.6.2. Energy Efficient Ethernet  24
      2.6.3. Ad-Hoc Wireless Networks  24
   2.7. Summary  25

3. **Design**  26
   3.1. Introduction  26
   3.2. Routing Algorithm Selection  27
      3.2.1. Route Vector vs Link State Protocols  27
      3.2.2. Floyd-Warshall Algorithm  28
   3.3. Minimal Energy Network  29
3.3.1. Algorithm Modification

3.4. Load Balancing
   3.4.1. Alternative Route Storage
   3.4.2. Monitoring Traffic Level
   3.4.3. Alternative Route Selection
   3.4.4. Traffic Division between Routes

3.5. Power Management States
   3.5.1. Assigning Power Management Levels to Switches

3.6. Switch Efficiency Weighting

3.7. Review

4. Implementation
   4.1. Introduction
   4.2. Operating System, Languages and Libraries
   4.3. POX
      4.3.1. Forwarding
      4.3.1. Traffic
   4.4. Data Structures
      4.4.1. Data Structures within Discovery
      4.4.2. Data Structures within Forwarding
      4.4.3. Data Structures within spanning_tree module
      4.4.4. Data Structures within Traffic module
   4.5. Principal Features
      4.5.1. Controller Start-up and topology Discovery
      4.5.2. Route Weighting Calculation
      4.5.3. Choosing Route and Checking Capacity
      4.5.4. Installing Route
   4.6. Mininet and Test Network
   4.7. Graphical Display
   4.8. Review

5. Evaluation
   5.1. Introduction
   5.2. Test Environment
      5.2.1. Fat Tree Topologies
      5.2.2. GÉANT
      5.2.3. Network Switch
      5.2.4. Traffic Procedure
   5.3. Experiment One
      5.3.1. Expected Results
1. Introduction

1.1. Motivation

Over the past decade energy efficiency has become a hot topic due to diminishing resources, the rising cost of electricity and climate change. In almost every walk of life efforts are being made to reduce energy usage and become as efficient as possible.

The information and communication technology sector is no different. Western Digital released a new series of hard drives that reduce their rotational speed during periods of low use to cut power consumption\(^1\). Facebook and Google are building data centres within the Arctic Circle to eliminate the power needed to cool their servers\(^2\). Computers and peripherals are being designed to use as little power as possible while running and slip into lower power sleep states when inactive.

However very little work has been successfully implemented to reduce the power consumption of networks. New switches and routers are designed to consume less power than their predecessors but these savings are small in comparison to the phenomenal growth of the internet.

“Reduce” is one of the popular three R’s but it won’t work for networks, as people are not going to cut back on surfing the web, streaming video and downloading files. In fact it is probably harder to find a faster growing service than broadband Ethernet connections. Hence we have to be prepared for networks to become an even more significant user of power in the future.

One area that can be examined to help save energy is scaling the network relative to demand. Currently all of the links and nodes in computer networks are on all of the time irrespective of usage. This means that the local internet uses as much power at six in the morning when the majority of users are in bed as it does at six in the evening when large amounts of users are streaming videos to watch for the night.

Minimal Energy Networks maintain only the minimum amount of connections needed to satisfy user demand. By examining algorithms to achieve this and shutting down excess capacity the energy usage of computer networks can be substantially reduced.

To implement a minimal energy network I will look at utilising a new form of network control known as Software Defined Networking (SDN). In SDN control over how each switch forwards packets is handled by a centralised server instead of the switches themselves. The use of SDN is necessary as conventional large networks don’t behave predictably\(^3\) enough to implement minimal energy networks and can be difficult to configure and operate exactly as intended.
1.2. Project Goal

The main aim of this project is twofold, an initial exploration and familiarization with the various components and protocols in Software Defined Networking. Then utilizing this knowledge to evaluate the potential energy savings of several Minimum Energy Algorithms by testing them on a virtual network.

I intend to create a virtual network based upon the GEANT research network currently in place on continental Europe. This network will have several paths between various end nodes and links of different bandwidth and latency.

A custom centralized controller will act upon this network constantly evaluating the topology and traffic levels. It will apply a minimum energy algorithm dynamically scaling the network to ensure a minimum quality of service is maintained. As many links and nodes as possible will be powered down hence the network should consume as little power as possible.

A personal aim of this project was to extend my knowledge of computer networks and the tools used to manipulate them. This was done by becoming familiar with Linux Command Line and the high level scripting language Python, both commonly used in networking applications. Investigating the infrastructure, protocols and algorithms of today’s computer networks and finally looking at various method of multipath routing.

1.3. Outline

Chapter 2 – State of the Art: This chapter gives some background information on the energy usage of today’s networks, particularly the internet. It then gives the reader an introduction and basic understanding of conventional networks followed by Software Defined Networking and several network simulation tools. It finishes off with a look at some previous work in the field of reducing Ethernet energy usage.

Chapter 3 – Minimal Energy Algorithm Design: This chapter discusses the various networking algorithms that could be used and then selects one. It evaluates how the algorithm can be changed to choose energy efficient routes through a network. At the same time it looks at how to ensure a minimum level of service is maintained for the user.

Chapter 4 – Controller Implementation: This chapter discusses problems encountered when integrating the chosen algorithm design into a functional controller. It also has an in depth look at the tools used to create the test environment.
5 – **Evaluation:** This chapter documents the testing of the controller design in virtualised networks, both symmetrical and realistic asymmetrical ones. It notes the initial results and any problems encountered during testing. The expected and actual obtain results are compared and evaluated.

6 – **Future Work:** Lays out several routes this project can be continued along and improved upon based upon the findings in the evaluation.

7 – **Conclusion:** Sums up the conclusions of the project.

8 – **Appendix:** Gives detailed information on packet formats.

9 – **References:** Lists the various sources of information used in this report.

### 1.4. Terminology

**MEN** – Minimal Energy Network

**SDN** – Software Defined Networking

**ARP** – Address Resolution Protocol

**LLDP** – Link Layer Discovery Protocol

**MAC** – Medium Access Control

**IP** – Internet Protocol

**LAN** – Local Area Network

**OSI** – Open System Interconnection

**MST** – Minimum Spanning Tree

**OSPF** – Open Shortest Path First

**BGP** – Border Gateway Protocol

**ONF** – Open Networking Foundation
2. State of the Art

2.1. Introduction

To implement the most effective minimal energy network using SDN this report looks at the way conventional networks use energy and how much they use. Evaluating their operation, previous work on minimizing energy usage and current SDN technology will give us the background required.

2.2. Energy Usage of Today’s Networks

2.2.1. Global Energy Usage of the Internet

The internet as it stands today is a large and very inefficient user of energy and it is undergoing a significant increase in size. In 2007 it was estimated by J. G. Koomey of Stanford University\textsuperscript{4} from measured data of the most popular servers that the internet required around 900 billion kWh of electricity. This is approximately 1.5\% of global electricity consumption\textsuperscript{4,5} and increasing at a rate of around 20-25\% each year. Setting up computer networks and the internet to become one of the top users of electricity. The energy efficiency of a fixed backbone network is relatively low, 8-10 times less than wireless networks\textsuperscript{6}. As the internet is such a large and inefficient user of electricity it is easy to realise the large potential for savings.

2.2.2. Energy Usage of the Physical Network

Around 37\% of total internet energy usage comes from the telecoms infrastructure\textsuperscript{7}, another 23\% from data centres\textsuperscript{8} and the remainder is end users. The main energy users within both the backbone and data centres networks are switches and routers. Tests on 48-port switches\textsuperscript{9} conducted for a project at Stanford have shown that turning on a switch without connections causes the power consumption of the device to rise to 75\% of full power. Connecting links to all of the ports causes’ power consumption to elevate to 95\% full power when the links are idle. Hence a fully connected switch only experiences 5\% deviation in power usage between periods of no traffic and peak loads\textsuperscript{9}. As the majority of networks are always turned on fully regardless of load it results in high unnecessary energy usage.
2.3. How Networks Currently Work

Large networks such as the internet are formed from many smaller networks which in turn are made of even smaller networks. At the lowest level are local area networks (LAN) consisting of end user terminals and several switches connecting them. LANs can take one of several topology forms as shown in Fig. 1 all with their own advantages or disadvantages.

![Network Topologies for LANs](image)

Figure 1 - Various Network Topologies for LANs

The majority of switches today operate with manufacturers firmware utilising one of the many routing protocols described below. They communicate as peers sending each other packets containing information about their connections. Utilising returned information each switch builds its own picture of the network and create tables linking various end destinations with a port to forward packets out. Incoming packets are checked against all of the entries in a table to find the best match and forwarded accordingly.

Switches tend to operate at layer 2 of the Open Systems Interconnection (OSI) reference model learning the Medium Access Control (MAC) addresses of all nodes within the network. MAC addresses are assigned to the network card by the manufacturer and are designed not to change. No two devices should share the same MAC address.

The next step is a wide area network (WAN) that connects several LANs together. The switch that connects a LAN to a WAN and connects other WANs to each other is known as a router. As well as the basic switch functionality a router will decide the next network a packet should visit and gives it a next hop address. The rest of the switches in the LAN will forward the packet to this next hop address instead of the
destination address. If the destination happens to be within that LAN the destination and next hop addresses will be identical.

Internally routers are more complex but operate similarly to switches. They have to store more information as they need to have knowledge of other networks as well as all the information relating to the local network. A router will also possess more intelligence compared to a switch.

### 2.3.1. Conventional Routing Protocols

Routing Protocols specify how switches and routers communicate with each other and select routes between nodes on a network. Each router has prior knowledge of its immediate neighbours in the network and the protocol dictates how this knowledge is circulated so other routers can use it. The protocol also specifies the routing algorithm to be used in determining path selection.

![Wide Area Network (WAN) Typical Schematic](image)

**Figure 2**: Each LAN is an autonomous system. Routers are used to connect all of the LANs together while switches are used within the LANs.

An autonomous system presents itself to the rest of the internet as one network and operates under a single set of management rules. Protocols can be divided into two levels, firstly Interior Gateway Routing for routing within an autonomous system such
as the LANs in fig. 2. Secondly Exterior Gateway Routing between autonomous systems such as from one LAN to another. Interior Gateway Routing can be further subdivided into protocols that make use of distance-vector algorithms or link-state algorithms.

### 2.3.1.1. Distance-Vector Routing Protocols

In distance-vector protocols, routers do not possess information about the full network topology. They merely know the cost to reach destinations and the next router on the route. In its simplest form this cost to reach a destination is the number of hops to the, however more advanced protocols assign different weights to links as seen in fig. 3. Each router uses the lowest cost for a destination they have received from other routers and appends the cost of their link to that router. In effect distance-vector protocols operate similarly to the Bellman-Ford Algorithm\(^{31}\). The main weakness of distance-vector protocols is that there is no way to check for in-correct information from other routers and errors can propagate. An example of distance-vector routing is the Routing Information Protocols (RIP) and its dependents\(^{12}\).

### 2.3.1.2. Link-State Routing Protocols

In contrast with link-state protocols, such as Open Shortest Path First (OSPF), each router possesses its own information on the complete topology. Routers use this information along with algorithms such as Dijkstra or Floyd-Warshall to calculate the next-hop destination for each packet. Link-state protocols have proven to be much more reliable and flexible than distance-vector and hence have become the prevalent form of routing within autonomous networks.

### 2.3.1.3. Exterior Gateway Routing Protocols

Protocols such as Border Gateway Protocol (BGP) dictate how traffic flows between autonomous networks such as in the internet. Due to the complexity of routing between networks it would be impossible for each router to know the exact topology at a destination or the distance cost to reach it. Hence these protocols operate using path-vector routing instead of link-state or distance-vector routing. Path-vector protocols operate similarly to distance-vector however the entire path traversed to reach a destination network is circulated. In addition routers can store several paths to the same destination instead of only the shortest one as happens in distance-vector\(^{13}\).
2.3.2. Minimum Spanning Tree

In graph theory, a spanning tree is a graph in which all nodes are connected to each other only once. In effect there are no loops. A Minimum Spanning Tree (MST) is one where the combined weight of the links is the lowest possible. A MST can be applied to a network with multiple links between switches to work out the minimal topology needed to maintain connectivity. All links not in the MST can be shut down and routers or switches instructed not to forward to them.

Switches that do not have end users connected to them can be excluded from the MST. If we assign an energy weighting to all of the links and switches in a network, MST algorithms could be used to work out the Minimal Energy Network (MEN).

MSTs are useful when creating a computer network as many interior gateway protocols do not support loops in the topology. For instance if there were two routes between A and B they would form a loop. A MST algorithm can be implemented over a topology to administratively break links by prevent forwarding on them such as in IEEE 802.1d.

2.4. How Software Defined Networking Works

Software Defined Networking (SDN) is a new method of defining network control protocols with software applications abstracted from the underlying hardware. This way different control applications and different manufacturers’ switches can be mixed and matched. This is in contrast to conventional networks where operation is defined by firmware unique to the switch variant. SDN applications can be changed, updated
or customised as often as network administrators want. Conventional firmware is only updated periodically by the manufacturers.

SDN was developed due to the increasing deployment of cloud architecture and dynamic resource allocation. Network operators wanted a way to dynamically specify network services. With SDN the software applications specifying management policies and network operation will be separate to the actual hardware that forwards packets. This enables network operators to write their own software applications and customise network control as needed.

### 2.4.1. OpenFlow

OpenFlow is a communication protocol that makes use of SDN by physical separating the control plane and the data plane found in a traditional switch. The data plane will remain on the switch\(^1\), while the control plane will be shared between several switches and operate from a remote server. From here on this server is referred to as the controller. This gives greater flexibility in defining how the network operates and what protocols it uses to forward packets, if any.

OpenFlow is an open source project supported by the Open Networking Foundation (ONF). The ONF is a non-profit consortium that has the backing of several major corporate partners such as Google, Microsoft, Facebook, Yahoo and countless others.

---

\(^1\) OpenFlow, the protocol used for this project is only a forwarding protocol, once a device is OpenFlow compatible it can operate as either a switch or router with the functionality decided by the centralized controller.
It is by far the most popular and best developed example of SDN and the only one commercially deployed\textsuperscript{18}. 

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{image.png}
\caption{Checking for Matching OpenFlow Flow Table Entry}
\end{figure}

\subsection{2.4.1. Operation of OpenFlow}

An Openflow enabled switch only takes care of forwarding, all intelligence and storage of information is handled by the central controller. The switch houses a flow table and its operation can be compared to a cache with only the most recent routing information temporarily residing on it. When a packet arrives at an OpenFlow switch the following steps are executed:
The packet header is compared against flows in the switches flow table, where to speed up operation these tables can be multi-level. The flows are made up as follows:

- Matching criteria: Each flow has a set of criteria to compare against the header of incoming packets. These criteria can be set very loose such as destination MAC or IP address only. Alternatively they can be set strict so a packet must match exactly. A detailed description of the matching criteria is given in the implementation section.
- Actions: These are the instructions that will be carried out if a packet matches the flow. This field will usually instruct the packet to be forwarded out a certain port. Alternatively the flow can send the packet to a switch’s conventional control plane, another table or drop the packet.
- Timer: Flows are only present in the switch’s table for a finite amount of time before being removed. Usually two timers are given, an idle timeout of 30 seconds and an absolute timeout of 60 seconds, although they can be varied. These times are the norm in OpenFlow to ensure streams of traffic are regularly referred to the controller.
- Counters: Each time a flow is executed a counter is incremented to record traffic levels.

If a match is found the packet follows the contained instructions. If not it is sent to the controller as in fig. 6.

The controller will make a routing decision and install a flow in the switch for the packet to follow if it knows the destination.

### 2.4.1. Advantages of OpenFlow and SDN

Network intelligence is logically centralized across a common control plain, eliminating the chance of conflicting routing tables in different switches. This comes at the cost of eliminating the redundancy of distributing intelligence among the switches. The centralized controller now becomes a single point of failure and the link between it and the switch introduces a second potential point of failure. These
limitations can be overcome by having a backup server ready to take over in case of the first failing. In addition most OpenFlow enabled switches will have a conventional control plain built in that can take over in a loss of communication with the controller.

**Network virtualization** is enabled allowing several different virtual networks to operate from the same physical network. Each of these virtual networks can have their own controller with completely differing operational rules. An umbrella controller can be instructed to house all of these guest controllers. It will take care of dividing bandwidth between the virtual networks and directing packets to the appropriate controller.

This enables different services to be treated differently and prioritized over one another. For instance voice over IP streams could be identified and given high priority low latency routes from source to destination. A very large data file could be routed onto a high capacity route but interrupted occasionally to let other services through. Less critical and less data intensive streams such as web pages could be given routes with lower capacity. This way we can ensure that different services get the connections and service level they require.

**Customisation:** One of the biggest advantages of OpenFlow is that it enables network operators to be no longer constrained by manufacturers firmware. They are free to write their own control applications and customise network operation to suit individual requirements. Control applications can also be changed regularly so that an application optimized for current conditions is in use. One control application may operate during peak loads in the morning and evening while another may take over during the dead of night.

**Hardware** can be dumbed down as switches only use their data plain to forward packets; they no longer need their own control plain. As the switch only houses a reduced flow table the look up time for flows that do not visit the controller is vastly reduced.

While switches no longer need their own control plain to enhance backward capability many OpenFlow enabled switches have been designed with one. This enables the switch to operate in either a conventional mode or OpenFlow mode. Several advanced switch designs can even operate in a combinational mode where the OpenFlow controller can install a flow instructing the switch to handle the packet in a conventional manner for certain packets.
2.4.2. Controllers

The controller is the heart of an OpenFlow network. It lays out the policy for how the network will operate and stores additional information about the network. The algorithms coded into the controller by the network administrator will dictate how it responds to load and forwards traffic.

There are several open source controllers already being developed that users can customise to a high degree.

- NOX\(^{19}\): This is a highly optimized high performance controller written in C and developed by a research group in Stanford\(^{20}\). It has very low latency and can handle more packets per second than its brethren POX and NOX-classic however it takes longer to write custom code. Hence it is intended for non-test deployment in networks and is only supported by Linux based operating systems.

- POX\(^{21}\): Is developed by the same group in Stanford\(^{18}\) and written in Python. It is aimed more at research than NOX having a higher latency and lower throughput rate. However because it is written in a high level language it is much easier to code with and implement new ideas. It is also supported by Linux, Max and Windows based operating systems.

- NOX-Classic\(^{22}\): This is the ancestor of both NOX and POX and was intended to be a compromise between high performance and ease of customisation. The core elements were optimized in C to give high performance with extended functionality created in Python. It was found that trying to satisfy both of these requirements resulted in sub-optimal performance. Hence it was split into the two mentioned above, however many groups continue to develop and maintain this controller such as RouteFlow\(^{23}\).

- Beacon: An event based Java controller running under the GPL licence. It was created by David Erickson and is primarily aimed at research uses.

- Floodlight: Another Java based controller this was split from Beacon by Big Switch Networks to aim it at commercial deployment hence it is more geared toward reliable operation and less to research. It also operates under the Apache licence in contrast to the GPL licence.

- Trema: This is a Ruby and C based controller maintained by a Japanese group for research purposes.
Figure 8: Full System Diagram

Pox, POXDesk and Mininet and associated tools are run from the a guest version of Linux Ubuntu running on the host machine which uses Windows 7 and Google Chrome to display the GUI.

Mininet instantiates a virtual network at its core specified by a topology file. An inbuilt CLI module handles communication with external traffic generation devices such as Ostinato, Ping and IPerf.

When a packet arrives into a software switch in Mininet it is compared against the flow table for a matching flow. If one is found it is forwarded, if not it is sent to the controller.

POX is the controller, forwarding will receive packets that switches could not forward themselves. If it knows the destination it will install the required flows to route the packet to the destination. If not it will instruct the switch to flood the packet in a message requesting the location of the destination. When a reply makes its way to the control it will note the location in a table of mac addresses.

The discovery module instructs all switches to send LLDP packets to discover links in the network and then informs other modules of the topology. Spanning Tree disables flooding on certain ports to prevent flooded messages from looping around. Traffic sends stats requests to all switches in the network and works out the traffic levels on the network from the replies.

tinyTopo.py is used to communicate with POXDesk and the graphical display. Within POXDesk topoViewer.js constructs the graphical display as a web page using springy.js to provide real world physics to the display. The webpage can then be viewed in a conventional browser such as Google Chrome.

2.4.3. Pox

OpenFlow is not a controller itself, the OpenFlow Switch protocol serves to link the controller to the switches within a network. POX is one of these controllers and is a python implementation of NOX (Network Operating System).

Rather than being a single application POX is structured as a collection of python modules as seen in fig. 8. When invoking POX different combinations of modules can be included or excluded to change the behaviour of the controller. Additional modules can also be called to provide more information about the network without interfering with it such as debugging and GUI backends. A more detailed description of how the controller works is provided under the implementation section.

It should be noted that there are two current builds of POX available, a master version and the betta version. The former is for long term release sacrificing newer features and regular updates for stability. The latter is an active branch undergoing regular development and constant updates. Many additional features and third party extensions such as GUIs only work in the betta version. This is due to a different internal naming convention and prevents easy conversion of projects between the versions.

2.4.3.1. POXDesk

PoxDesk$^{24}$ is a web based graphical display developed to work with POX. It is generated as a web page by the machine running the controller. It can then be
accessed using a conventional web browser from any machine with a connection to the control server.

![Image of POXDesk](image)

**Figure 9**: POXDesk. Topology viewer (left), switch flow table viewer (top centre), terminal input (top right), log view (bottom right)

It puts several useful graphical displays of the network at the disposal of the administrator:

- A physical network topology map showing all switches and their links.
- Tables of all of the flows currently installed on each switch in real time.
- It can re-display terminal debugging messages from the controller.

POXDesk utilises springy a third party physics engine to dynamically animate topology maps. Springy evenly spaces switches apart within the window and spreads out links using real world physics.

### 2.5. Test Network

When testing OpenFlow controllers it would be ideal to have a real network that traffic could be sent across. This would enable actual implementation of the controller and gives the most realistic results. Unfortunately the expense in acquiring all of the hardware needed for a real network is prohibitive. In addition a lot of time would be needed to individually configure each device. Simulators can give us some idea of how a controller will perform in a large network but are not as realistic as other methods. As there is no traffic flowing across the network there is no path to hardware as we have not implemented a test version of the controller. Networked
Virtual Machines (VM) save a lot of the expense and time involved in configuring real networks as the entire network can be run as a collection of virtual machines on one host. However they cannot be used to re-create large networks as each VM has its own associated memory overhead which quickly exhausts the real machines resources. Mininet, a tool developed at Stanford is a possible solution; it follows the VM example but uses OS level virtualisation to reduce the overhead of each device. Hence the entire network can be run from one kernel enabling large networks to be run from one machine.

2.5.1. Mininet

Mininet is a tool for running real traffic on large virtualized networks within one machine. It recreates network topologies either from command line input or from pre-built topology scripts. Each switch in these networks is a virtual implementation of a real switch designed to recreate real life behaviour as much as possible. Traffic can then be run across this network and measured from various points.

Mininet has been designed from the outset to be flexible, deployable, interactive, scalable, realistic and share-able. It achieves flexibility by enabling a user to re-create any network topology they want through an inbuilt python API. Topologies can be saved as a file and shared with other developers enabling deploying an identical copy in seconds. Realism is also a key component with each virtual component designed to replicate its real life counterpart as much as possible. This enables Mininet test networks to re-create real world conditions as closely as possible. An OpenFlow controller tested on a Mininet network should be able to deploy to a real network without additional changes. Real time management and interaction with the network is supported by an inbuilt command line interface.
In normal virtual machine networks each node is a virtual machine and has an associated computation and memory overhead. This is too heavy weight with computation resources, particularly memory being utilized far too quickly preventing large networks from being simulated on a single real machine. Mininet only recreates the bare essentials to virtualise a host, in particular virtual network spaces and Ethernet pairs enabling large networks to be re-created. This is known as OS level virtualisation which enables successful booting of up to 4,096 node networks on one machine or over 25,000 on a cluster. Mininet therefore combines the best features of emulators, hardware test beds and simulators into one package. It boots fast, is inexpensive, quickly reconfigurable and highly interactive. Topologies can be typed out and configured in python in fewer lines than XML.

Mininet does have several limitations:

- It is only support by Linux OS.
- Mininet only supports wired networks as it does not simulate the constraints found on wireless networks.
- All nodes use the same file system preventing the use of different OS kernels simultaneously on nodes. In other words all the switches in a Mininet network have to be one uniform type.
- The large amount of nodes that can be created in a Mininet network leads to generated traffic exceeding the bandwidth and CPU capacity of the computer. When this happens performance fidelity is lost as CPU resources are handled by a default Linux scheduler. Hence packets ready for sending may not be sent instantaneously.
2.5.2. Wireshark

When a test network is running on Mininet we need a way to capture the packets flowing along the virtual links and analyse them. Wireshark is the foremost open source packet analyser available at the moment. It enables deep inspection of all traffic from a specific port or group of ports. It began as an open source project in 1998 named Ethereal.

It makes use of an easy to read graphic user interface that enables the user to easily identify different packets from a stream through colour coding and sorting of the stream based on packet characteristics. It enables the user to closely examine each portion of the packet and labels all of the levels to make analysis easy.
2.5.3. Ostinato

Figure 12: Various screen captures from Ostinato. The left most is the main console used to select individual ports to start and stop transmission. The second one is use to select protocols. The third one is used to enter in source and destination information for the chosen protocols. The fourth screen is used to control the amount of packets transmitted in each stream.

Ostinato is a tool used to create and edit packets before broadcasting them over a network. It has been designed to complement Wireshark and in effect operate as wireshark in reverse.

While ostinato can interact with several computers and send packets to several port groups it also has the ability to discover virtual ports connected within a machine. Hence it was able to view all of the connections within a simulated Mininet network.

The packaged GUI is easy to use and comes with a simple menu that allows users to specify packet header format by hand. A custom payload can then be added and the tool’s flexibility extends as far as enabling editing of individual octets within the packet. In addition support is provided for sending several different packets out the same interface at various different times and speeds. Ostinato can broadcast on several different ports simultaneously which enables easy traffic generation to test loading of a network.

2.6. Energy Efficient Networks

To get the best energy savings out of OpenFlow it is advisable to look at other work conducted on making networks more energy efficient. Some of this work can possibly be incorporated into our design or we can avoid areas where significant work has already been done.

2.6.1. Elastic Tree

Elastic Tree is a research project carried out by Brandon Heller at Stanford which explores how to reduce the power consumption of fixed data centre networks. Savings are achieved by scaling the number and links and nodes up and down...
Elastic Tree works by taking a network with redundant connections as indicated above. It evaluates the traffic flowing through the network and works out if the network can function without several of the connections. If this is the case it will shut down as many duplicate links as it can without compromising performance. It will also shut down switches as the eradication of links leaves them un-necessary. The network is shut down in stages and the number of active stages can be tailored to match demand.

For instance if the network was operating at full capacity all of the links and switches would be on to spread the load as evenly as possible. As the traffic levels reduce the network further the network would be reduced to the bare minimum spanning tree (MST) required to keep all hosts connected. This offered energy savings of up to 40% when operating with 16 hosts and increased to over 60% with 1024 hosts on a network\(^9\).

Elastic Tree expanded upon this by including the ability to specify extra redundancy within a network. For instance one could specify a redundancy of MST+1 which would dictate that all edge switches on the network are connected by at least two links at all times regardless of the network activity.

This reflects the redundancy required in data centres where it is desired to keep redundant links between major nodes in networks to prevent a single failure from bringing down a large portion of the network. This reduced the power saving potential of elastic tree in small networks by a large margin, at MST+1 the energy
saving in a 16 switch network dropped to 15%. When the size of the network expanded the energy savings began to match those of the original MST.

### 2.6.2. Energy Efficient Ethernet

Another method of increasing the energy efficiency of the internet is the work of the Energy Efficient Ethernet Taskforce 802.3az. This works by throttling down links and switches when they are not required to be operating at full capacity for instance a 1000MB/s link to 100Mb/s to 10Mb/s. It has been estimated that if this was implemented across all machines and networks operating today it would result in a saving of 3.73 – 5.86 TWh/year excluding cooling power usage\(^27\). There is still some work required to ensure that data surges do not overwhelm links operating at lower capacities. The transition time from a low rate to a high rate will need to take place in under 10ms\(^28\). In addition current machines are not capable of fully benefiting from this technology unless designed to do so from scratch, in particular routers and switches.

### 2.6.3. Ad-Hoc Wireless Networks

Work on energy efficient routing has already been conducted on ad-hoc wireless networks\(^4\). In these networks nodes are often powered by a battery and hence the routing algorithm has to ensure that the battery life is maximised hence preserving the network. There are two distinct cases, the first where all the nodes are mobile and treated equally and the second case where several of the nodes have a fixed power supply and don’t have to worry about energy exhaustion. In the second case the routing paths can be centred on the powered node enabling it to do most of the power intensive sending and receiving.

Several techniques can be looked at to minimise the energy usage such as routing to nodes with more battery life. This keeps low power nodes operating for longer at the expense of increased battery life degradation in the high battery life node. In addition a path through multiple nodes may actually use less energy than a direct one. This is because the amount of power required to broadcast to distant nodes is far far greater than the power required to broadcast to near nodes with an adaptive transmitter. Hence even if a node could broadcast to one within range it would be more efficient to involve an intermediate node.

There are also two different ways we can look at keeping nodes alive:

- The first maximises the time the network remains fully operational with all nodes able to transmit and receive. These networks will tolerate nodes with low battery life by routing around them as much as possible. While this
maintains full functionality as long as possible once nodes start to shut down the network tends to disintegrate quickly. This is because a node will only fail when it has exhausted its battery and all nodes around it have also exhausted their battery from routing around it.

- The second looks at keeping a usable network alive for longer. Nodes that are not necessary for the networks full operation are allowed to die early sacrificing their battery life to keep key nodes running for longer. These networks will not maintain full operation as long as the first kind. However they will enable partial operation for much longer.

A balance has to be struck between keeping the network fully alive for as long as possible vs keeping the majority of it usable and sacrificing the un-important parts.

2.7. Summary

As we can see there are several on-going projects to reduce the energy usage of a network. However none of these projects have been commercially implemented. Hence there is considerable scope to look at how software defined networking can be used to incorporate these ideas seamlessly into computer networks.

This chapter began by examining the energy usage of the internet and the underlying infrastructure in section 2.2. Section 2.3 looked at the way conventional networks worked and examined current routing protocols and minimal spanning trees. Section 2.4 compared conventional operation to OpenFlow and Software Defined Networking taking a look at POX as well. Section 2.5 looked at the tools required to create a realistic test network and measure its performance. Section 2.6 looked at previous work into the field of making networks more energy efficient.

In particular it examined scaling the network to match demand in elastic tree. This area warrants more investigation as it was only carried out on fixed symmetrical networks. It could be expanded on by examining:

- Scaling realistic asymmetrical networks.
- Incorporating Power Management States into the design.
3. Design

3.1. Introduction

To maximum savings a customised controller will be designed to minimise the energy usage of a network. The design starting point will be to choose an algorithm that is capable of being centralized and has scope to integrate energy saving calculations.

Using POX as a platform the controller will be customised to reduce power consumption by creating a minimal energy network. To ensure a minimal quality of service is maintained the design will also feature load balancing. Incorporating power management states into the design will reduce power consumption further during periods of low traffic. It will also reduce the time needed to bring a sleeping switch back online. Finally the controller will evaluate the efficiency of hardware to see if routes can be optimized to avoid power hungry switches in favour of greener ones.

3.2. Routing Algorithm Selection

The controller will be based upon an algorithm that will evaluate the energy cost of routes and choose the best one. Current protocols are fast, highly optimized and scale well; these are all elements that should be incorporated into the design. The design will start with a routing protocol and algorithm that can be adjusted to solve the following requirements:

- Most of today’s protocols are designed to work in a network with distributed intelligence. The design needs to be one that will transfer to the centralised architecture found in Openflow.
- The controller will choose paths that consume the least amount of energy. In addition it will do this dynamically taking into account the state of the rest of the network. Therefore the routing algorithm needs to dynamically evaluate the energy cost of the entire route.
• As the path weights are going to re-calculated dynamically in real time we need an algorithm that keeps calculation down to a minimum.

### 3.2.1. Route Vector vs Link State Protocols

The majority of routing protocols today are link-state protocols. As switches in link-state protocols maintain a topology map of the entire network these protocols are well suited to operating from a centralized controller. Here are several link state algorithms candidates for our controller:

- **Floyd-Warshall**: Compares all possible paths between each pair of switches to find the lowest weight link. Manages to complete this task in only $O(|V|^3)$ comparisons in a network that could contain $O(|V|^2)$ edges. However at worst case there are only two comparisons, one addition and one assignment for every iteration of the algorithm. Hence while it does grow at a rate of $O(|V|^3)$ there are only four operations in every cycle. The algorithm can also handle negative link weights which could prove useful if links are given weights to indicate their efficiency.

- **Dijkstra**: Works out the distance from a single source switch to all the other switches in the network. It completes this task in steps finding first all of the directly connected nodes and then all of the one hop nodes and so on. Operates in a distributed manor as it needs to be repeated for every switch in the network. Has a computational complexity of $O(|V||E|)$ however E can approach $V^2$ in densely connected networks. The algorithm cannot handle negative weight links.

- **Bellman-Ford**: An extension of Dijkstra algorithm incorporating negative link weights. Also operates in a distributed manor with a computational complexity of $O(|V||E|)$.  

- **Johnson**: Applied the Bellman-Ford algorithm to eliminate all negative weights followed by the Dijkstra algorithm to find links through the graphs. Results in $O(|V|^2 log|V| + |V||E|)$ calculations. In very sparse networks it will calculate faster than the Floyd-Warshall but the later will calculate faster in dense networks. If a Fibonacci heap is created for the Johnson algorithm it becomes very efficient however Fibonacci heaps can be expensive to construct.

### 3.2.2. Floyd-Warshall Algorithm

The Floyd-Warshall algorithm has been chosen as it compares every possible path. This will enable comparison of all routes to find the most energy efficient possible. One main reason to choose this algorithm over the others is that it will make the best use of the centralized architecture of OpenFlow. After comparison the database of all routes can be stored on the central server. The other algorithms discussed above will
all work in a distributed manor maintaining routes for each node. It may not be as easy to compare these as it was with the centralized case.

The Floyd-Warshall algorithm can also handle negative edge weights. It does not store the entire route only the total number of hops. The storage of the total route is explained in the implementation section.

\[
\text{let } \text{path_map} \text{ be an array with size (number of switches) by (number of switches) consisting of minimum distances initialized to } \infty
\]

\[
\text{for each switch } v
\]

\[
\text{path_map}[v][v] = 0
\]

\[
\text{for each link } (u,v)
\]

\[
\text{path_map}[u][v] = 1
\]

\[
\text{for } k \text{ from 1 to } |V|
\]

\[
\text{for } i \text{ from 1 to } |V|
\]

\[
\text{for } j \text{ from 1 to } |V|
\]

\[
\text{if } ((\text{path_map}[i][k] + \text{path_map}[k][j]) < \text{path_map}[i][j])
\]

\[
\text{then } \text{path_map}[i][j] = \text{path_map}[i][k] + \text{path_map}[k][j]
\]

Equation 1: Original Floyd-Warshall Algorithm in pseudo code

The Floyd-Warshall algorithm works by comparing all possible routes through a network to find the lowest weighted route.

1. Upon initialization it assigns weights to all possible links in the network.
   a. Zero from a switch to itself.
   b. Either 1 or an assigned weight w(uv) for directly connected switches.
   c. Infinity for all other switch pairs.

2. It then selects every possible switch in the network
   a. It tries it as a connection between every possible pair of switches.
   b. If the route using this switch has a lower weight OR there is no route already in place it assigns this route as the new route.

3.3. **Minimal Energy Network**

To reduce the energy consumption of networks we want to turn off unnecessary switches and links. Maintaining only the bare minimum number needed to connect all elements of the network yielding a minimal energy network.
This is relatively easy to achieve in symmetrical networks such as fat tree networks found in data centres where all routes are the same length as indicated in elastic fig. 13 and in fig. 14. In this case the Floyd-Warshall Algorithm will always favour routes through lower MAC numbered switches. This occurs as the algorithm organises all of the switches into a list by MAC number. Hence switch 11\textsuperscript{ii} will be tried as an intermediate switch in routes across the network before switches 12, 13 and 14. As the routes evaluated using the later switches will have the same weighting as the earlier route and not less they will be dropped.

This will not work as well in real networks as they are not symmetrical and will have routes of varying length as seen in fig. 15. A minimal energy network should make use of already active routes with traffic flowing along them, even if it is not the shortest path. This means that the shorter path can be powered down. Un-modified the Floyd-Warshall Algorithm will favour shorter routes irrespective of traffic levels. Hence the algorithm needs to be changed to take into account routes that already have traffic on them.

### 3.3.1. Algorithm Modification

\begin{verbatim}
for each edge (u,v)
  dist[u][v] ← 4
\end{verbatim}

Equation 2: Modification of Floyd-Warshall Algorithm to increase default edge weighting

\textsuperscript{ii} To keep the page tidy switches are referred to by the last two digits of the MAC number instead of the whole number. Hence 11 is 00-00-00-00-00-11
To combat this problem the Floyd-Warshall algorithm has been changed to assign a weight of four to directly connected nodes instead of one. When a flow is installed through a link the weight of the link is reduced via subtraction, until it reaches one. When a flow is removed the path-weight is increased up to a maximum of four. Hence the algorithm will now favour routes with traffic already flowing across them, creating a minimal energy network.

3.4. Load Balancing

Concentrating as much traffic as possible onto the same routes reduces the energy consumption of the network. However it can also result in those routes becoming saturated with traffic levels exceeding capacity. This results in packet delays that may exceed the required minimum level of service and cause certain traffic such as Voice over IP to become un-usable. In addition interrupted packet sequences may time out causing extra traffic such as re-acknowledgements and the re-transition of packets.

To solve this problem we need to ensure a minimum service level is maintained across the network. Traffic levels need to be monitored and alternative routes stored and selected if the load on a route is about to exceed its capacity. In addition we need to work out how to balance load between multiple routes.

3.4.1. Alternative Route Storage

As it stands the design only maintains the route with the lowest path weight between source and destination. By adapting the algorithm to also store N routes with the next lowest rating we provide redundancy. N is dictated by the size of the network and its total traffic capacity.
3.4.2. Monitoring Traffic Level

The controller will need to dynamically monitor traffic levels across the network. OpenFlow already has inbuilt messages designed to query ports and find the total number of packets or bytes transmitted through them. Recording the time these messages were issued from the switch and subtracting the previous time and totals enables us to work out the number of bits per second. Adding this rate with the identical rate from the port at the opposite end gives the total traffic level on the link.

When the controller evaluates a route for traffic levels it checks each link sequentially until one of them is found to have traffic levels above an acceptable threshold. If this is the case an alternative route will need to be selected. If none of the links fall above this threshold the route may be used. In the event all available routes fall above the threshold the route that is the least amount over the threshold will be selected.

The threshold is set just below the ultimate capacity of the links. This means the network will keep putting traffic on the same link almost until it reaches the point where the link physically cannot accept any more traffic. It is un-advisable to set the threshold any higher as when the link begins to reach capacity any momentary spikes will cause it to exceed capacity. Resulting in excessive packet queues in switch buffers.

3.4.3. Alternative Route Selection

The alternative route selected will be the one with the next lowest routing weight. This weight will gradually reduce as more and more traffic is installed on the route. Hence the minimal network philosophy will be maintained with only the minimal
amount of links required to connect all parts of the network and maintain a minimum level of service quality.

### 3.4.4. Traffic Division between Routes

Traffic is divided between routes at the flow level. When the controller needs to install a flow it will place them on the new route until enough flows have expired on the original route to bring traffic levels below the threshold. Once this happens the controller will install more flows for the original path until traffic levels once again exceed the threshold in which case it will revert to installing flows on the second path. Hence one route will remain close to capacity while the second route only has the minimum amount of excess flows. It can be shut down as soon as the traffic falls to levels that can be satisfied with one switch.

This method ensures that packets from the same sequence of communication always follow the same route through the network. For instance a TCP connection will not have packets arriving out of sequence due to different route lengths.

If this is confusing it is useful to remember that in OpenFlow traffic can be split into flows based upon numerous constraints either alone or in combination:

- source and destination MAC address fields
- source and destination IP address fields
- source and destination TCP/UDP ports
- switch input port
- VLAN ID and priorities
- Ethernet frame type
- IP protocol

Hence traffic between two locations in a network can be divided up as much as is needed while keeping all packets from one sequence along the same route.

### 3.5. Power Management States

While this report has referred to turning off un-used switches this tends to be un-feasible in realistic operation for a number of reasons:

- Switches as with any hardware have associated turn on and start up times meaning it can take several seconds, if not minutes, for them to come back online after cold shutdown.
- Large amounts of temporary routing data can be lost during cold shutdown which takes time and computation power to rebuild before the switch can operate effectively again.
- Sudden traffic spikes on a network require the ability to bring switches online fast to handle the sudden excess traffic.
- In the event of a switch failure we need to be able to hand over to another redundant switch quickly to prevent service interruption on the network.

Hence it is more realistic to put a switch into a low power sleep state instead where it can be brought back online in seconds. Currently there are very few commercial network switches available that reduce their power levels by this method. Research in the elastic tree project at Stanford\textsuperscript{9} has suggested that most of the power consumption from a switch is caused by it being turned on regardless of the traffic levels flowing through it.

In addition such a switch could be designed to have several intermediate levels along the lines of the Advanced Configuration and Power Interface (ACPI) standards presently used in computers. The switch would consume less power while still being able to forward limited amounts of traffic. Energy Efficient Ethernet or Low Energy Ethernet could also be integrated to reduce power consumption. Some links and switches may be required to ensure connectivity to a low usage part of the network. They are necessary but will not receive high traffic levels. Hence switch power states will enable them to consume a smaller fraction of the power they would otherwise have consumed.

Research is being conducted at Hanoi University of Science and Technology, Vietnam, into a switch design that accepts commands from OpenFlow telling it to choose a more power efficient state\textsuperscript{34}. The advantage of utilising the controller to dictate switch power states instead of relying upon the switch to decide itself are numerous. The controller is directing traffic so it knows when a large amount of traffic will reach a switch before the switch does. Hence it can power up the switch and have it ready to process the traffic before it arrives. In addition occasional broadcasts on a link may cause a self-assigning power level switch to keep itself active. The controller would know that no traffic of use is being sent on that link and could tell the switch to power down irrespective of any interfering noise. Finally there is no need for the switch to become active again on all ports to listen for traffic; it only needs to listen to its OpenFlow Management port. Therefore it can go into a deeper level of sleep.

### 3.5.1. Assigning Power Management Levels to Switches

For the controller to determine the power level a switch should adopt it needs to examine the amount of traffic being sent to that switch. We need to examine the amount of traffic being sent through links to the switch enabling it to power up before that traffic reaches the switch. To do this we examine the rate of traffic flowing from each the egress ports of all adjacent switches directed at the switch in question. By
summing all of this traffic we obtain the total expected traffic that is likely to arrive at
the switch. Hence we can assign power management levels to the switches. The
controller will need to send control messages down to the switch to achieve this
power down.

3.6. Switch Efficiency Weighting

One of the limitations of the controller is that it is reducing energy usage by trying to
reduce the number of switches in the active network. It does not take into account the
efficiency of the underlying infrastructure. Most OpenFlow switches available today
do not have the ability to assign power management levels. These switches could be
avoided in favour of future switch designs that will assign power levels.

An extension of this could be to assign an energy efficiency value to switches that
take into account not only hardware’s efficiency but the power supply as well. For
instance a plant located in Germany has to pay 131.9% as much for the electricity
supplied to it as one located close by in the Netherlands based on 2011 industrial
figures\textsuperscript{35}.

Currently when faced with the choice between several routes of equal path weighting
the one with the lowest MAC number will be chosen first. By assigning path weights
proportional to the energy efficiency of the switches this situation is avoided.

3.7. Summery

This chapter began in section 3.2 by examining several different algorithms to use the
controller design. Section 3.3 looked at how to change the algorithm so it created a
minimal energy network. This was followed in section 3.4 with an examination on
how to ensure that the traffic on any one link never exceeded its capacity. Section 3.5
looked at increasing efficiency improvements even more by integrating power
management states into switch design. Finishing off in section 3.6 with a look at how
hardware efficiency could be taken into account to route around in-efficient parts of
the network. Integrating energy efficient features such as those described above into a
control should help reduce the power consumption of computer networks.
4. Implementation

4.1. Introduction

This chapter takes a look at how the design was integrated into a SDN controller using the OpenFlow protocol. It is divided into five sections. Section 4.2 takes a look at the choice and languages and libraries as well as the operating environment the controller was run within. Section 4.3 gives a brief overview of how POX operates and interacts with the other components. Section 4.4 takes a look at placing the enhanced energy efficient Floyd-Warshall algorithm discussed in chapter 3 into the POX controller. Section 4.5 deals with generating realistic traffic patterns to drive across the virtual network. Section 4.6 takes a brief look at the graphical display and associated libraries. Finally section 4.7 gives a brief review of the entire chapter.

4.2. Operating System, Languages and Libraries

Linux based operating systems tend to be the norm in networking applications and servers. They allow a high degree of control over how the system operates and carry no licence fee. This project was designed for and run from Linux Ubuntu 11.04 or later. The POX controller is also compatible with Windows and Mac OS’s and with only a few library changes can be easily ported.

Python was chosen as the controller language due to the large number of networking related libraries available. The availability of a controller written in python, namely POX with an active support community. As a high level scripting language it enabled the creation of the core routing functionality in only a few hundred lines. This enables fast experimenting and testing not possible in lower level languages such as C/C++.

The python packet library contains numerous functions and class definitions for various networking protocols. This makes creating, handling and decoding Ethernet packet such as LLDP, IPV4 and ARP easy. In addition the open networking foundation has created additional OpenFlow protocol libraries. Simplifying the creation and handling of data or control messages.
4.3. **POX**

POX is an OpenFlow controller developed at Stanford, it is related to NOX but written in python. Currently POX only supports OpenFlow-Switch specification v1.0.0 hence this project is designed for that specification. Initial plans are in place to implement support up to specification v1.3.1 however they have not progressed enough for use at this stage. A backup branch of the active beta version was taken on Jan 4th to avoid additional update commits from interfering with the project.

Figure 18 (next page): Full System Diagram

Pox, POXDesk and Mininet and associated tools are run from the a guest version of Linux Ubuntu running on the host machine which uses Windows 7 and Google Chrome to display the GUI.

Mininet instantiates a virtual network at its core specified by a topology file. An inbuilt CLI module handles communication with external traffic generation devices such as Ostinato, Ping and IPerf.

When a packet arrives into a software switch in Mininet it is compared against the flow table for a matching flow. If one is found it is forwarded, if not it is sent to the controller.

POX is the controller, forwarding will receive packets that switches could not forward themselves. If it knows the destination it will install the required flows to route the packet to the destination. If not it will instruct the switch to flood the packet in a message requesting the location of the destination. When a reply makes its way to the control it will note the location in a table of mac addresses.

The discovery module instructs all switches to send LLDP packets to discover links in the network and then informs other modules of the topology. Spanning Tree disables flooding on certain ports to prevent flooded messages from looping around. Traffic sends stats requests to all switches in the network and works out the traffic levels on the network from the replies.

tinyTopo.py is used to communicate with POXDesk and the graphical display. Within POXDesk topoViewer.js constructs the graphical display as a web page using springy.js to provide real world physics to the display. The webpage can then be viewed in a conventional browser such as Google Chrome.
POX is designed around modules with the functionality of the controller changing depending upon the included modules. The messenger module offers support for GUIs and Traffic offers the ability to measure traffic levels on the network. This project will concentrate on developing the modules Forwarding and Traffic. These modules handle the forwarding decisions and traffic monitoring respectfully. Following is a list of the un-modified modules shipped with POX that have been used in implementation of the controller. Forwarding and Traffic and discussed in their own sections afterwards.

- **Discovery**: is a module that registers hosts and switches as they connect to the network. It sends customised 802.1ab LLDP messages (see appendix) to all connected switches and listens for the replies to gain information. Based on this it discovers all links within the network and maintains a table of all links.

- **py**: An interactive python interpreter for debugging and experimentation.

- **Messenger**: This component specialises in sending messages from POX to external extensions such as POXDesk. It is used to keep the display informed of the current network state.

- **spanning_tree**: The spanning tree module that ships with POX is not the standardised IEEE 802.1D version commonly implemented in today’s networks. Instead it is a custom version that eliminates flooding on ports that would otherwise create loops. It leaves the original network available for use by the forwarding elements of POX.

It works as follows:
1. Sorts all discovered switches into an ordered list.
2. Create an empty list of seen switches and good links.
3. For the first switch add it to the list of seen switches.
4. Precede to checks each link on this switch that is not in good.
5. If the switch at the other end of this link is in seen this link creates a loop: do nothing for now. Else put the new switch in seen and the link in good.
6. Choose the next switch in list that is not in seen.
7. Send the below message to each port in the network with config as 0 if the link is in good to do nothing or of.OFPPC_NO_FLOOD to disable flooding if the link is not in good.
4.3.1. **Forwarding**

Forwarding is the heart of the controller and decides how packets are routed through the network. It is based upon the l2_multi module that comes as standard with POX. Before modification l2_multi received un-weighted topology information from Discovery and choose routes with the lowest number of hops.

It was renamed Forwarding and modified to assign weights to links based upon the efficiency of the underlying hardware. It also received information from Traffic and dynamically adjusted these link weights to favour routes with traffic already on them. Hence forming a MST around efficient hardware. It was also modified to ensure traffic levels did not exceed a certain threshold and form a second route to balance the load.

4.3.1. **Traffic**

Traffic: This component was created to send control messages asking for traffic statistics from the switches in the network. It then builds up a map of traffic along the network and sends this information to other modules.

4.4. **Data Structures**

Given the large amount of network data requiring storage it is important that efficient data structures are used to enable quick access. Here is a list and description of all the data structures used to store information:

```c
# Blocks flooding on a port
pm = of.ofp_port_mod( port_no=p.port_no,
    hw_addr=p.hw_addr,
    config = OFPP_FLOOD or OFPP_NO_FLOOD,
    mask = OFPP_NO_FLOOD )
con.send(pm)
```

Code 3: An OpenFlow protocol message to disable flooding on ports
4.4.1. Data Structures within Discovery module

- **Link**
  
  Link = namedtuple("Link",("dpidl","port1","dpid2","port2"))
  
  Link stores the DPID of both switches for each link and the port number for those two switches. DPID is an ID within the controller for each connected switch. The upper 16 bits are implementer defined, in this instance they are set to 0. The lower 48 bits are the MAC address hence the DPID is based upon the MAC address for this project.

- **linkTimerList**

  self.linkTimerList[link] = time.time()

  linkTimerList is stored in the discovery module and is a set that stores all links known to the controller and the time since a LLDP packet passed through the link.

4.4.2. Data Structures within Forwarding module

- **adjacency**

  adjacency[sw1][sw2] = sw1.portToSw2

  adjacency = defaultdict(lambda:defaultdict(lambda:None))

  adjacency stores the port a packet should exit switch 1 to get to switch 2, it stores None if this route is not possible. linkPortList is arranged as a default dictionary hence it can be queried about switches it has no information for and still return None. Data is entered into this structure from the Discovery modules adjacency. Sw 1/2 are the MAC address of their respective switches

- **switches & switchInv**

  switches[dpid] = (MAC)

  switches = {}

  switchInv[MAC] = {dpid}

  switchInv = {}

  These enable easy switching between the MAC and DPID addresses of the switches within Forwarding.

- **mac_map**

  mac_map is used to store the MAC addresses of the switches hosts are connected to. When the controller starts up it does not know the location of hosts. It only learns of their location when it sees messages with the hosts as
sources. When one of these messages is seen the information is extracted and stored in mac_map for future use.

- **path_map**
  
  ```python
  path_map = defaultdict(lambda:defaultdict(lambda:defaultdict(lambda:None)))
  ```
  path_map stores N different paths between each pair of switches listed by MAC. It is used for calculating the routes between switches and comparing their weights. For each of these paths it stores the weighting of the route and an intermediate switch's MAC address. The path weight is None if the route is not possible, 0 if the route is from a switch to itself and a positive value for all other routes. The intermediate switch is None for directly connected switches, from a switch to itself or for non-existent routes.

### 4.4.3. Data Structures within spanning_tree module

- **adj**
  
  ```python
  adj = defaultdict(lambda:defaultdict(lambda:[]))
  ```
  Similar to the structure found in Forwarding also constructed from adjacency in Discovery, except it lists the switches by DPID instead of MAC.

### 4.4.4. Data Structures within Traffic module

- **flow_rate**
  
  ```python
  flow_rate = defaultdict(lambda:defaultdict(lambda:(0, 0, 0, 0)))
  ```
  This stores the number of received and transmitted packets and bytes for each port on each switch.

- **rate**
  
  ```python
  rate = defaultdict(lambda:defaultdict(lambda:(0, 0, 0, 0)))
  ```
  This contains the same as flow_rate but in total amount since recording began.

- **cap_rate**
  
  ```python
  cap_rate = defaultdict(lambda:0)
  ```
  This is the received and transmitted byte values from rate combined together. The values have been converted from bytes/5 seconds to bits/second.
4.5. **Principal Features**

The best way to describe the implementation of the controller is to give a walkthrough of its operation. This begins by describing the connection of the switches to the controller and the discovery of the general topology. Following this the controller then works out the path weighting of all routes across the network. It then checks the traffic levels across several routes and chooses the most efficient one with excess capacity. Finally we describe packets arriving at a switch and having flows installed.

4.5.1. **Controller Start-up and topology Discovery**

The initial stages of controller start-up are handled by the Discovery module. This process has remained unchanged from the original l2_multi module. When a switch, either real or a virtual Mininet one, starts up it checks in with its assigned controller. The controller then sends a control message to the OpenFlow enabled switch installing a default flow instructing it to send all received LLDP packets to the controller.

```java
log.debug("Installing flow for %s", dpidToStr(event.dpid))
match = of.ofp_match(dl_type = ethernet.LLDP_TYPE,
              dl_dst = NDP_MULTICAST)
msg = of.ofp_flow_mod()
msg.match = match
msg.actions.append(of.ofp_action_output(port = of.OFPP_CONTROLLER))
event.connection.send(msg)
```

**Code 4: Installing flow to send all LLDP packets to the controller**

Discovery then instructs the new switch and all connected switches to send LLDP packets out of all active ports. All of the switches then send the LLDP packets they receive back to the controller. These packets are deciphered by Discovery to find out which switches are directly connected. This information is stored within adjacency where it is accessible to other modules. Periodically Discovery will send LLDP packets through the network to ensure the topology has not changed. Each link has a timestamp that notifies the controller of when it last successfully transmitted a LLDP packet. If these timestamp becomes outdated the controller removes the link from the topology.

4.5.2. **Route Weighting Calculation**

Whenever the network topology changes due to the addition or removal of switches and links the path map is reset. During controller start up and periods of topology
change many switches and links will connect or disconnect in rapid succession. To prevent recalculation each time path_map is only calculated when it is needed to route a packet. Forwarding was based upon another module called l2_multi so a quick description of the original and modified module is given.

4.5.2.1. l2_multi’s Operation

Forwarding is based upon l2_multi which loaded in default path weight values into path_map. The value for a switch to itself is zero and all directly connected switchers were given a value of 1. The Floyd-Warshall algorithm is then called and starts combining routes and comparing route weights to find the shortest number of hops between each and every switch.

- for every possible switch within the network (k)
- try using it as a possible intermediate between every pair of switches (i -> j)
- our new route is i -> k -> j = i -> k + k -> j
- if the number of hops in this route is less than the previous route replace it
- move onto the next route and/or switch

path_map then stored a N*N table for the routes between all switches within the network however it does not have information on where hosts are connected within the network. When a host sends a message the controller learns which switch it is connected to from the packet. This information is stored in mac_map for future use, any message destined for that host will be sent to the switch it is connected to. If a packet is sent looking for a destination not know to the controller it floods the packet to all links on all switches to see if it can find the host. If the host is within the network it will reply and the controller will learn its location from the reply packet. This way over time the controller builds up a map of how all the hosts are connected to the network.

It is the process of flooding these messages that requires the operation of a spanning tree protocol on the network. Otherwise the messages would keep circulating around the network indefinitely.
4.5.2.1. Forwarding’s Operation

In **Forwarding** the default value for a switch to itself is zero to avoid the possibility of a route through another switch evaluating with a lower path weight. In the plain minimal energy network controller a value of 4 is given to directly connected switches. This value is not reflective of the number of paths and its operation is explained later.

When the controller was modified to take account of hardware efficiency this was changed to retrieve the information from a file. This file contains a value for the energy efficiency of each switch. The values for the switch at each end of the link are averaged together to get the efficiency of the link.

---

### Code 5: The initial link weightings being loaded into path_map

```python
### assigns path_weights to path_map
### for every switch (f) in network assign a path_weight
### of 0 indicating it's itself
### for every switch (i) directly connected to f assign a
### path_weight based on the energy efficiency of the switch
for f in sw2:
    i = switches[f]
    for j, port in adjacency[switches[f]].iteritems():
        if port is None: continue
        path_map[i][j][0] = {(g[f]),None}
        path_map[i][j][1] = {(g[f]),None}
        path_map[i][j][2] = {(g[f]),None}
        path_map[i][j][3] = {(g[f]),None}
    path_map[i][i][0] = {0,None} # distance, intermediate
    path_map[i][i][1] = {0,None} # distance, intermediate
    path_map[i][i][2] = {0,None} # distance, intermediate
    path_map[i][i][3] = {0,None} # distance, intermediate
```
The Floyd-Warshall algorithm is then called and starts combining routes and comparing their weighting to find the lowest possible weight between each and every switch. This is roughly how the code show in fig XX operates:

- for every possible switch within the network (k)
- try using it as a possible intermediate between every pair of switches (i -> j)
- for every route n from (i -> k) if that route exists
- for every route nn from (k -> j) if that route exists
- our new route is i -> k -> j = i -> k + k -> j
- if intermediate is not already installed, the source or the destination:
- for every route s from (i -> j) if that route does not exist load in new route and exit
- for every route s from (i -> j) if that route is larger than the new route replace it with the new route and exit
- for the first route s from (i -> j) if that route is equal to the new route increment s until two routes using the same intermediate node are found and replace the second one
move onto the next route and/or switch

4.5.3. Choosing Route and Checking Capacity

```python
# calls _calc_paths()
# called by _get_path()
# gets the path between two switches
# works by recursively calling itself until entire route is found
# hence we keep cutting the route in half until we find no more nodes
def _get_raw_path( src, mnn, dst):
    nn = mnn
    if path_map is empty has it calculated
    if len(path_map) == 0: _calc_paths()
    if src is dst:
        return []
    if current value of path_map for N(mnn) is empty (None) decrement N(nnnn) and check again
    if path_map[src][dst][nn][0] is None:
        nnnn = nn
        while path_map[src][dst][nnnn][0] is None:
            nnnn -= 1
        nnnn = 0
        nn = nnnn
    path_map[src][sw][N] = [path weight, swN] // intermediate = swK
    intermediate = path_map[src][dst][nn][1]
    if intermediate == src:
        print 'ERROR SRC'
        if intermediate == dst:
            print 'ERROR DST'
    intermediate = None
    if intermediate is None:
        return []
    #log info(' %s -> %s -> %s', src, intermediate, dst)
    path_map[sw][intermediate] = [path weight of sw3] + _get_raw_path(sw1, N, sw5)
    return _get_raw_path(sw1, N, sw3) + [path weight of sw3] + _get_raw_path(sw3, N, sw5) + _get_raw_path(intermediate, nn, dst)
```

Code 7: Recursive calling of _get_raw_path to find all of the nodes on a route

The route retrieval algorithm remains un-changed from l2_multi. Once all of the path weights are calculated the one needed for the individual packet query can be retrieved. Path_map does not store the full route; it only gives the total weighting and an intermediate node. The function _get_raw_path is called to return the full route as follows:

- It extracts the intermediate node from path_map
- recursively calls itself from the source to the intermediate
- recursively calls itself from the intermediate to the destination
- returns the route from source to intermediate + intermediate + intermediate to destination
- it will keep recursively calling itself until it has extracted all intermediate nodes.
4.5.3.1. Modifications to check Capacity

```python
# checks capacity of each link and returns True if traffic is within capacity
def check_cap(p):
    for i in range(len(p) - 1):
        if flow_stats.cap_rate[switch.inv[p[i]]][adjacency[p[i]][p[i+1]]] > LINK_CAPACITY_THRESHOLD:
            return False
        else:
            return True
```

**Code 8: Checking of Threshold**

Before modification this route would have been installed straight away. However in **Forwarding** each separate link is checked from source to destination of the proposed route. If any one of these links fall above a certain threshold the route is rejected. An alternative route is then selected and tested to see if its traffic levels are within the capacity. Once a route is selected the controller can proceed to installing the flow in the necessary switches along the path.

```python
# sends out a states request to retrieve port statistics every 5 seconds
def timer_func():
    for connection in core.openflow_.connections.values():
        connection.send(of.ofp_stats_request(body=of.ofp_port_stats_request()))
```

**Code 9: The ofp_port_stats_request message**

The module **Traffic** makes use of the openflow stats request (ofp_stats_request) message to take care of capacity checking. The reply to this message gives transmitted and received values in both packets and bytes for each port in a switch. The values from the previous request stored in prev_rate are subtracted to get the difference and stored it in flow_rate. As the messages are sent every 5s we can convert the difference into a rate. This rate is then changed into bits per second and the transmitted and received values are averaged together. This gives us the traffic level for that port and hence the link connected to it, this rate is stored in cap_rate.

4.5.4. Installing Route

The **Forwarding** module of the controller then proceeds to install the required flows in all of the switches along the route. This operation is un-changed from l2-multi. It sends the flow modification message in fig. 7 to all switches taking care to send it to the switch that the packet originated from last. This way those flows should be ready by the time the packet reaches them. Subsequent packets on that flow will be switches without any involvement on the part of the controller.
When sending flow modify (ofp_flow_mod) message quite a strict set of matching criteria are assigned. Hence only subsequent identical packets will follow this flow. The flow timeout values are set at around 30 seconds for the idle timeout and 60 seconds for the absolute timeout. This means that the flow entries will expire after 30 seconds of inactivity or 60 seconds in any case. Hence the controller can dynamically change the way the stream is routed every minute.

The original design intention was to reduce the path weighting of each individual link as traffic was installed on it. This would have resulted in these routes being favoured for new flows keeping traffic on already active links. However this would require re-calculation of the path_map each time a flow was installed for it to have an effect on the total route weighting. This would have resulted in too much computation and slowed down the controller to un-acceptable levels.

Instead the total route weight is reduced when a flow is installed along it instead of the individual link weights. No re-calculation is required this way so the controller maintains its speed. This method is not as robust as the original as it has no effect on other routes. A second route that shares most of the links as the first one will not have its weighting reduced. Hence the integrity of the minimal energy network is compromised. However this compromise is necessary and shouldn’t make the controller deviate from ideal behaviour too much.
4.6. Mininet and Test Network

Mininet can create virtual test networks either from command line input or by reading scripts that define complex layouts. In this project scripts were used to create the topologies by entering the following command line argument:

```
sudo mn --custom ~/mininet/custom/topo-GEANT_BACKBONE.py --mac --topo mytopo
```

The `--custom` argument let mininet know to start up using a custom topology specified afterwards. The `--topo` argument specified which topology in this file location was to be used. As there was only one file in the file it was redundant but is still required for Mininet to start successfully. `--mac` told it to assign MAC address based on the numerical ID of the node, for instance node 1 received MAC address 00-00-00-00-00-01 to make debugging easier. The controller functioned perfectly without this argument assigning random MAC addresses.

Mininet uses a software implementation of an OpenFlow v1.0.0 compatible switch. This can be upgraded to a newer software switch that supports up to v1.3.1 however this was not needed for the project. The software switch has been designed to mimic the behaviour of a real switch as much as possible.

4.7. Graphical Display

![Graphical Display](image)

Figure 19 & 20: In this instance the graphical display turns switches from green -> orange -> red to indicate increasing levels of traffic.
The graphical display was based on POXDesk\textsuperscript{36} and runs from the internal webserver of the host machine for the controller. Running it on the webserver enables it to be displayed on another machine or terminal remote from the server controlling the network. JavaScript was the language of choice due to the additional web support it provides.

POXDesk makes use of the QooXDoo framework\textsuperscript{37} to create a GUI app that would work independent of platform. The springy library\textsuperscript{38} was used to create and animate topology maps. Springy is a force directed graph layout algorithm that makes use of real world physics to keep elements equally spread out.

The GUI interfaces with the controller through a small python backend file known as tiny_topo.py. This module takes information from the Discovery, Forwarding and flow_stats modules and passes them to the JavaScript front end. The backend file amalgamates all of the required data into one table. It also causes the GUI to refresh every time there is a topology change. By keeping the backend contained in one non-intrusive file we ensure that the controllers operation is not affected by the GUI.

Originally the following information was passed from the backend to the front end:

- the number of switches
- the links between them

Using this information the frontend constructs a force directed topology map consisting of the switches and links without any active information about traffic levels.

This was modified to also send the following information:

- if the link is active or not
- the traffic levels on the links
- the traffic levels flowing in the switches
- the energy efficiency value of the switch

Using this switches are coloured either according to the level of traffic flowing through them or their energy efficiency depending on the particular experiment being conducted. Green indicates a switch under light traffic load or high efficiency. The colour then changes to yellow, followed by orange and finally red to indicate increase load or decreasing efficiency. The links are coloured light grey to indicate when they are placed in a low power sleep state and then green to indicate they are online. Depending upon the traffic levels of the link it will gradually turn orange and then red.
4.8. Summery

This chapter discussed the languages chosen and the environment used. It gave an in depth overview of POX, its functionality and structure. Following this an overview of the data structures used to hold all of the data was conducted. Using this information the operation of the controller was explored. Mininet was set up and we finishing up with a quick look at the graphical display.

At this stage in the project we now have a working controller and test network. Hence we are ready to move onto running experiments to test how well the system handles load.
5. Evaluation

5.1. Introduction

Now that the controller is fully functional and implemented the focus of this project turns to testing it. To quantify the performance of the design there are several characteristics to examine:

- The reliability of the controller under harsh traffic conditions, does it break down and cease to function?
- The ability of the controller to handle large volumes of traffic, in particular levels exceeding the capacity of the physical network. In what way does it respond and how does it prioritize traffic.
- Does the controller meet minimum quality of service requirements by spawning alternative routes to balance the load?
- How much energy is saved by the minimal energy networking features? Is it the most optimal minimal energy network possible?
- Does taking the energy efficiency of switches into account have a significant effect?
- Can the controller scale and cope with large, networks with complex topologies.
- How fast does the controller converge on the most efficient layout? Do packets experience more or less of a delay from this controller than they would in a conventional network.

This chapter starts by defining the test environment and quantifying its realism. This will involve looking at the various network topologies created and instantiating them with Mininet. Following will be an examination of the traffic patterns created using Iperf and Ostinato.

Following this is a section detailing the two tests lined up for the controller. First how much energy can be saved by integrating power management states and a minimal energy network algorithm into the controller. The second looks at the savings gained from modifying the minimal energy algorithm to take into account the efficiency of the underlying hardware. In both cases the expected results are defined and then the experiment is documented. The real results are compared with those expected and observations made on why they match or differ.
5.2. Test Environment

The types of networks virtualized, their topology and capacity will have a big effect on the controller’s behaviour. In particular the realism level of the virtual networks is important to ensuring the results obtained are applicable to real world conditions. Testing was initially conducted on a symmetrical twenty node fat tree network with identical links. Once this was successful testing moved onto a larger forty-five node fat tree network with varying link capacities. The third network used was a realistic implementation of the GÉANT European Research Institute network.

5.2.1. Fat Tree Topologies

Both tests were initially conducted on two simple Fat Tree topologies in Mininet. These were chosen as they contained many symmetrical routes of equal length allowing parallel paths to be shut down. In the 20 switch network all switches had at least two connections, most had four, and in the 45 switch network connections numbered from three to six. In the 20 node network each edge switch had 2 hosts connected to it. Each edge switch in the 45 nodes network had 3 hosts connected to it. This gave plenty of options for the controller to route around inefficient hardware in the second experiment. These networks provided an easy de-bugging and initial test environment ensuring the controller was fully functional before moving on the GÉANT network. The simple nature of these networks also made them quite useful for providing demonstrations of the controller in action.
5.2.2. GÉANT

To test the controller on a more challenging and realistic asymmetrical network GÉANT was implemented on Mininet. GÉANT is a pan European network that connects several research and higher education institutions together. This network was chosen as its topology is publically available with accurate maps enabling recreation. It is also quite a large network spread across the continent with several routes of different length between locations. For instance one can send messages from the Greek peninsula to Ireland in several ways. Either through warm Mediterranean countries such as Italy and Spain, a northern route through Bulgaria, Romania, Slovakia, Germany and the Czech Republic or a central route through France and Switzerland.

These different routes enable us to evaluate situations where the hardware or power supply in one country is more desirable. This may be due to the abundance of energy, lower cost or a more environmentally friendly supply. For instance during blazingly hot weather with little cloud cover solar generating capacities would be maximised. Routes going via southern European countries could be favoured to make use of this extra green energy. Countries along the North-West have invested heavily in wind, Germany gets 10% of its total electricity supply from this supply\(^{39}\). Hence during periods of intense wind, routes through the North-West would be ideal. Switzerland and France have significant portions or their electricity generated by Hydro and Nuclear respectively. Routes through these two countries may be favoured at other times as they are less likely to experience supply fluctuations.
5.2.3. Minimizing Network Switch Power Consumption

Now that a test network is in place we need to have a test switch to populate the network. There are few commercially switches available today that enable power usage to be scaled as traffic levels through the switch vary. Research in Hanoi University of Science & Technology, Vietnam\textsuperscript{34}, is being conducted into one that will make use of OpenFlow to enable the controller to set the power management level of the switch. The power state will reflect how much traffic the switch can handle as well as how much energy it consumes. As no figures have been released from a prototype yet the potential savings will have to be estimated. This will be done by taking the power usage value from a current power scaling network switch. These values will be applied to an output from the controller giving the power management state of each switch. MatLAB will be used to combine these two figures and generate an estimate of the potential energy savings.

<table>
<thead>
<tr>
<th>Traffic Levels (Mb/s)</th>
<th>Power State</th>
<th>Power Consumption (W)</th>
</tr>
</thead>
<tbody>
<tr>
<td>~0</td>
<td>5</td>
<td>3.8</td>
</tr>
<tr>
<td>&lt; 100</td>
<td>4</td>
<td>5.8</td>
</tr>
<tr>
<td>&lt;1000</td>
<td>3</td>
<td>7.8</td>
</tr>
<tr>
<td>&lt;2000</td>
<td>2</td>
<td>9.8</td>
</tr>
<tr>
<td>&lt;3000</td>
<td>1</td>
<td>11.7</td>
</tr>
</tbody>
</table>

Table 1: Table of the Various Power consumption levels implemented in the switch and based on the D-Link DGS-10016D

The real switch chosen as the basis for the power model is a D-Link DGS-1016D\textsuperscript{40} sixteen port switch as it has the ability to scale between 3.8 W of power consumption when idle up to 11.7 W while under full load.

The switch used in Mininet is the OpenFlow Reference Software Switch for Linux. It recreates the operation of an OpenFlow switch for use with Mininet.

5.2.4. Traffic Procedure

These switches and the test network they reside on now need to be loaded with various levels of traffic. To ensure the full range of controller operation is tested light, medium and heavy traffic loads will be placed on the network. Both realistic and un-realistic maximum load scenarios will be tried. In all tests the same procedures was followed when loading the network with traffic:

1. The network was given 2.5s from the time the last switch connected to settle.
2. For 3s four hosts rapidly pinged between each other to create a medium load scenario.
3. The network was given 1.5s to cool down.
4. During the next 2s one host pinged another one to simulate minimum load conditions with only one active link.
5. The network is given 5s to cool down after this.
6. All hosts ping each other to simulate a heavy load.
7. The network is given 2.5 seconds to settle again.
8. Finally simulated variable level TCP traffic is driven across the network using Ostinato with the test culminating in ultimate load condition using all available hosts at maximum sending rate.

The idea behind this test procedure is that it tries several different load conditions from light to heavy and different formats of traffic. This creates several scenarios testing the ability of the controller to scale up the number of routes and balance load across them.

### 5.3. Experiment One

![Figure 25 & 26 & 27: Fully on network (left) Fully on Minimal Energy Network (middle) All Idle Minimal Energy Network (right)](image)

Experiment one looks at the ability of the controller to create a minimum energy network and scale it in response to changing traffic levels. It will examine the efficiency of the minimal network created to see if it is the most efficient possible. The energy consumption of the network with all switches on *fig. 25* will be compared to the scaling minimal energy network. There will be two cases of scaling network used for comparison. The first in which at the very least the minimum energy network is power on *fig. 26* and the second in which it shuts down as well as the rest of the network *fig. 27*. This second case is referred to as an all idle network with and the switches and links that have traffic on them will be powered up.
5.3.1. Expected Results

1: 20 Node All On Network
   20 switches * max power usage
   20 * 11.7 W = 234 W

2: 20 Node Minimal Energy Network
   7 switches * min + 13 * max
   7 * 3.8 W + 13 * 11.7 W = 178.7 W

3: 20 Node All Idle Network
   20 switches * min power usage
   20 * 3.8 W = 76 W

4: % saving
   \[
   \frac{\text{Minimal Energy Network}}{\text{Always On Network}} \times 100 = 23.6 \%
   \]

5: % saving
   \[
   \frac{\text{All Idle Network}}{\text{Minimal Energy Network}} \times 100 = 57.5 \%
   \]

6: % saving
   \[
   \frac{\text{All Idle Network}}{\text{Always On Network}} \times 100 = 67.5 \%
   \]

Calculation 1: Estimated Power Savings from implementing Minimal Energy Networks with and without the ability to place all links into deep sleep state.

By implementing a Minimal Energy Network in which all of the un-required links are put into a sleep state savings of around 23.6% are estimated. If we include the ability to put the active links of the MEN to sleep while not being used we improve the efficiency by 57.5%, 67.5% compared to the original always on network.

With the all idle network it is believed that as traffic levels begin to rise the controller will install flows for the traffic bringing the network out of its default sleep state. When traffic level rise towards the capacity of the links along one route a second route should be installed. The load should be mainly distributed on the original path with the additional path only carrying the needed excess load. As more and more traffic is pushed across the network this behaviour should be repeated with third and fourth parallel routes being installed as needed. When the traffic levels reduce these extra paths should shut down promptly. Following the total suspension of traffic across the network all switches should return to a sleep state.

There should be significant savings during periods of low traffic with large portions of the network being placed into a low power sleep state. The savings will be reduced during periods of high traffic as switches will be un-able to remain turned off and will be brought back online.
5.3.2. Testing Difficulties

Graph 1: Start up Times of Various 2N Trees, the Route Calculation Time can be seen to increase at a phenomenal rate. Theoretically it should follow the number of switches.

<table>
<thead>
<tr>
<th>2N Tree Depth, Width</th>
<th>No. of Switches</th>
<th>Start Up Time (s)</th>
<th>Route Calculation Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3, 7</td>
<td>57</td>
<td>21</td>
<td>6</td>
</tr>
<tr>
<td>3, 8</td>
<td>73</td>
<td>25</td>
<td>14</td>
</tr>
<tr>
<td>3, 9</td>
<td>91</td>
<td>28</td>
<td>26</td>
</tr>
<tr>
<td>4, 4</td>
<td>85</td>
<td>25</td>
<td>19</td>
</tr>
<tr>
<td>4, 5</td>
<td>156</td>
<td>52</td>
<td>FAILED</td>
</tr>
<tr>
<td>20 Switch Fat Tree</td>
<td>20</td>
<td>12</td>
<td>1</td>
</tr>
<tr>
<td>45 Switch Fat Tree</td>
<td>45</td>
<td>17</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 2: Start Up Time for Various 2N Trees

Problems were encountered with the scaling ability of the controller during testing. As the start-up time was not an initial priority no work had been done to optimise the
time it takes for switches to connect to the controller or for the spanning tree algorithm to operate on the network. It went from roughly 20 seconds in a 50 switch network to roughly 50 seconds in a 150 switch network as indicated in graph. 1. While the time for the controller to start up was slow it is not likely to grow uncontrollable and ideas about how to minimise it are mentioned in section 6.3.

A much bigger problem was the time it takes to calculate the routes. Theoretically this will grow relative to the number of switches cubed irrespective of the level of links within the network. Graph. 1 certainly indicates this rate of growth and when tried with a 150 switch network the algorithm failed to generate a path_map. Potential solutions to this problem are outlined in section 6.4 of future work.

Additional problems were encountered when Mininet began to crash during intense load traffic scenarios on the networks that did start. One of the limitations of running an entire virtual network from one machine that the host machine may not be able to handle the load. According to the Mininet introduction page a 3 GHz CPU can switch around 3 Gbps of traffic. Loosely applying this logic to the Dell Inspiron 15R with a 4*2.1 GHz i7 CPU I’m running my machine can switch approximately ~8.4 Gbps. Hence only small networks could be tested at maximum capacity.
5.3.3. Results

Graph 2: 20 node network. The red line is a control where all of the nodes are always on regardless of traffic level. The blue is a minimal energy network with the minimal energy network maintained on at all times. The green is a minimal energy network that only maintains the minimal amount of nodes needed for traffic. The blue and green graphs both have an identical spike at the start hence why the green obscures the blue.
Graph 3: 45 node network. The blue line is a control where all of the nodes are always on regardless of traffic level. The green is a minimal energy network with the nodes being turned on only as needed.

Figure 28: Two routes between 5f and 12 on a recreation of the GÉANT network.
5.3.4. Evaluation

In both tests a rapid spike was noted at the beginning of the recording. This is due to the initial traffic caused by the Discovery module flooding LLDP packets to build a topology map. LLDP packets are issued from every port of every switch in the network. This is repeated every time a new switch connects or a new link is discovered. When the controller is started up it results in a runaway situation as large amounts of link states messages are sent for a short period of time. This results in all switches being set to a high capacity state ready to accept large volumes of traffic. It should be noted that the controller converges back down relatively quickly hence it does not take long for the network to reach its normal operating state. In addition this only happens when the controller starts up.

A surprise is the 41% power saving achieved during intense periods of traffic, compared to an always on network. This is due to natural bottle necks within the network that prevented all components from running at full capacity. In fig. XX the central switches 11, 12, 13 and 14 would reach capacity preventing surrounding switches from sending at their maximum possible rate.

During periods of medium traffic levels the power savings were also impressive at around 60%. However these savings only increased by a small amount during periods of low traffic to around 70%. This reflects that there will be basic power consumption needed to keep switches in a sleep state. In addition a network with incredibly low levels of traffic may require the same amount of active links as one with medium levels of traffic. These figures are in line with finding from the Elastic Tree research project where similar findings in fixed data centre networks achieved savings of 59%.[42]

Figure 29 & 30: Load can be seen to swap between two alternative routes

69
One notable deviation from the expected behaviour was observed during load balancing. It was expected that the original route would take the majority of the load and the secondary route would only take the extra traffic needed to satisfy demand. Instead the majority of the load would swap between all available paths. Route 1 would carry most of the load for 1 second. Then the load would change to route 2, then to route 3 and so on.

This happened as most of the traffic generated was of the same type. Hence it all followed the same flow. Whichever route this flow was installed in would carry most of the load. This problem is solved by introducing several different types of traffic so that multiple flows would be created. The traffic on a real network is going to consist of many different types so this problem is unlikely to be encountered then.

5.4. Experiment Two

![Diagram](image-url)  
Figure 31: In experiment two the colours of the switches shows their energy efficiency with green being the most efficient and turning orange as the efficiency level drops.
Experiment two expanded the work conducted in experiment one. All of the installed hardware was given a random efficiency value and the controller built a minimal energy network trying to avoid in-efficient hardware. In particular attention was paid to how the controller spawned secondary routes due to capacity constraints and weather it still avoided in-efficient switches. The results were then compared to the scaling minimal energy network from experiment one.

5.4.1. Expected Results

It is expected that a similar minimum energy network to that found during experiment one will be formed. However it is believed that that switches used will be different to avoid in-efficient hardware. It is also likely that more switches and links will be brought online in the network at any given time. This is likely to happen as the most energy efficient route may not be the shortest route. At the end of this experiment there should be a slight energy saving compared to the results from experiment one.

5.4.2. Testing Difficulties

Problems were encountered during testing as the output duration would vary each time. Even though it was meant to last the same amount of time making it harder to compare results. The cause is a limitation of Mininet as during periods of intense loading the traffic volume exceeds the forwarding capability of the test machine. In these cases it is up to the python scheduler to decide when packets are sent. It does not guaranty performance fidelity hence packets that are meant to be sent immediately may not be sent at the same time during runs of the same experiment.
5.4.3. Results

![Graph 4: 20 Node network in Experiment Two. There was a slight deviation in the time it took for both controller to execute the same test.](image)

Compared to experiment one the savings were smaller this time with only an improvement of around 6% compared to the first experiment. However when compared to the original always on network the savings were once again significant.

5.4.4. Evaluation

During this experiment the controller created more routes across the network than in the previous experiment. This happens due to the presence of in-efficient switches in the branches from an already instantiated route to part of the network. An alternative route through efficient switches may have lower energy cost than this short branch to a pre-existant. Hence the alternative route would instantiated and used as can be seen in fig. 13 where switch 13 is used to reach the top part of the network to avoid routing through 0f.

It should be noted as well that there are a few low traffic periods when the controller taking hardware efficiency into account had higher power spikes than the controller from experiment one. This is due to the alternative routes created in situations described above lingering past there period of need. It takes upto 30 seconds for the
flows on these routes to timeout and be removed. It should be noted that the occurrence of these was very low.

Power usage graphs are smoother for the controller that takes energy efficiency into account than the one that does not. This is because the controller in experiment one is un-aware of topology efficiency. It therefore switches between efficient and inefficient hardware regularly causing power usage spikes.

Figure 32: 20 Node network where capacity has been exceeded one original routes causing the installation of a second route.

As can be seen in fig. 32 when a secondary route needs to be installed to alleviate traffic it utilises the next most efficient route.
5.5. **Summery**

Both tests performed well on small networks giving conclusive results. The controller was able to handle all of the test conditions while adhering to its designed specification. Larger networks proved to be more of a problem as the controller would crash when calculating routes for networks with 150 switches. Mininet also ran into problems handling high levels of traffic on networks of only 45 switches preventing conclusive testing on networks of this size.

Table 33: Occasionally the realistic network would spawn in-efficient alternative routes such as this one

While being overloaded there were a few times the realistic network suffered limitations. An alternative load balancing route may share one of the links in the original route. In this case this shared link acted as a bottleneck slowing down the network, this can be seen in fig. 33 between 0b and 12. However these scenarios only happened under excessive load and the majority of time load balancing enabled minimum service qualities to be met.
As can be seen there are significant savings to be made from implementing this energy efficient design. Due to natural bottlenecks several switches and links cannot run at maximum capacity. Hence they can be powered down or off altogether.

Taking hardware efficiency into consideration results in a slight improvement compared to the first test. These results are very susceptible to being influenced by the efficiency, or more correctly the variability of the efficiency of hardware within a network. Hence if all of the elements within a network are the same this algorithm will not generate any additional savings. On the other hand in a network where all of the elements are connected together by several different pieces of underlying hardware with varying levels of in-efficiency this second experiment brings large scale savings to the network by avoiding the in-efficient nodes and favouring the efficient ones.

Overall saving from the design were satisfactory. The energy consumption of the control 20 switch always on network was reduced from a constant 235 W to only 75 W while idle. Theoretically it should be possible power consumption close to 0 W in this case as the switches could be turned off completely. However with turn on times low enough to be used in real networks are not available. Therefore the level of savings achieved was quite reasonable.
6. Future Work

This chapter identifies several areas where work on this project could be continued. It takes a look at the centralized nature of OpenFlow and questions whether some of these benefits could have been obtained from a distributed implementation of the design. This chapter then looks at various ways to improve the design such as incorporating cross sub-net routing. During testing the time the controller took to start up and calculate routes was identified as a weakness. As a counteraction this chapter looks at reducing the time required to compute routes across the network. Finishing up with a look at how to reduce the amount of times the network is re-calculated.

6.1. Distributed Architecture SDN?

Implementing energy efficient algorithms and customising energy efficient network operation was very easy with software defined networking (SDN). SDN is a very powerful way of networking that puts control in the hand of network administrators. However OpenFlow as an implementation of SDN has limitations in the way it operates, mainly stemming from its centralized nature:

- Redundancy is lost and a single point of failure is introduced.
- The headers of the initial packets in streams of packets have to visit the controller before flows are installed. This procedure is repeated every 30-60 seconds to re-install flows. This results in un-necessary lag and traffic as the header makes its way to the controller and then back with the flow.
- This requires the creation of dedicated and reliable channels of communication between the controller and all switches in a network.
- Most of today’s networking protocols have been developed around distributed architecture, transitioning to centralized architecture leads to them being thrown out.
- Many OpenFlow implementations make use of a switches conventional control plain as a backup. Hence there is no cost saving associated with having cheaper hardware.

These points could possibly be resolved by designing a new form of SDN that relies upon distributed architecture. Many of the advantages of SDN, and the reason it was used for this project, stem from the ability to write control application for the network. A better way to do this might be to design switches that can have control applications installed in their control plain. This way we retain a network of switches that can have a custom controller installed on them. However this controller is replicated
multiple times and physically resides on each switch. It can then operate in a
distributed manor like current networks do. This will have the following advantages:

- no lag as packet headers are not sent to a remote server
- redundancy of distributed architecture
- can make use of conventional networking protocols and research
- No need for a dedicated communication channel between all switches and a
centralized database.

### 6.1.1. Will this design work as a Distributed Design?

With or without SDN it would be worthwhile to investigate if similar energy savings
could be achieved with distributed network control architecture. The design described
in this project made use of the centralized controller by combining information from
all switches. This was done to work out traffic levels in the network and balance load
on alternative routes if needed.

The creation of a minimal energy network can most likely be replicated in distributed
architecture. Messages sent between switches giving route information can be
changed to reflect the energy efficiency of the switch. However dynamic
measurement such as the traffic levels on a switch may not propagate through a
network quickly enough to make load balancing as discussed in this project feasible.
However it may be possible to develop a new algorithm that does not need traffic
levels from the entire network to keep traffic within acceptable levels.

### 6.2. Cross Sub-net routing

A serious limitation of the current design is that it can only route at the OSI (Open
Systems Interconnection) level 2 within one subnet. For the controller to be useful in
a real world situation it would need to route between different subnets at OSI level 3.
Currently the control uses only MAC addresses to identify switches and hosts. This
would need to be changed to using IP addresses. In addition a protocol such ARP
would need to be integrated into the design to associate MAC and IP addresses.

### 6.3. Reducing Network Start-up Time

Another limitation with the design in the start-up time need for the controller from
switch on to full network operational capability. The first main contributor to this
delay is the time needed for all of the switches within the network to connect to the
controller and for the spanning tree algorithm to block forwarding on certain ports. The second component is the time needed for the Floyd-Warshall algorithm to compare all paths within the network.

6.3.1. Network Connect Up

This project mainly concentrated on the algorithms involved in routing traffic across the network. Hence the time needed to connect up all of the components was not a priority and was not optimized in the controller design. Hence there should be potential savings if this area was looked at more closely. In particular the execution of the spanning tree module is not very efficient nor is it compliant with IEEE 802.1d the official standard for constructing minimum spanning trees. With a centralized controller it may be possible to query the entire network for the location of a host without flooding packets. If this was the case there would be no need to execute the spanning tree module on the network.

6.3.2. Route Calculation

As the Floyd-Warshall algorithm executes proportional to the number of switches cubed the time needed to compare all routes grows quite quickly. Hence it will be beneficial to look at keeping the number of calculations to be completed in each loop of the algorithm down to a minimum.

The Floyd-Warshall algorithm compares every possible route so its computation time is dependent only upon the number of switches and irrespective of the number of links between them. Hence it takes the same amount of time to computer the connections in a ten switch network where every switch is connected to each other, 100 links, as it does in a ten switch network with only nine links. As many networks, in particular realistic networks can be sparsely connected the Floyd-Warshall may not be the ideal algorithm. It would be beneficial to compare the performance to a controller with an algorithm that executes relative to the number of links such as the Johnson Algorithm 32.

6.4. Reducing Network Re-Computations

Another way to increase the speed of the controller would be to reduce the number of times it has to re-calculate the path_map. Currently the controller has to re-compute the entire path_map every time the physical topology changes. However this may not be necessary as only routes that would be altered by the addition or removal of switches or links need to be re-calculated. Using the Floyd-Warshall algorithm this
could be achieved by only computing the algorithm for the switch(s) that have been changed. In equation 1, $k$ would only have to be the switching that have had a link added or removed. This would help reduce the total amount of calculation needed in an active network as new switches or links are introduced.

6.5. Summery

This chapter has taken a look at the potential for recreating this project with distributed architecture in section 6.1. In section 6.2, we looked at some of the problems that need to be addressed before the controller can be implemented in a real network. Section 6.3 addressed the need to reduce the time it takes for the controller to get going and examined the potential for using a different algorithm. Section 6.4 expanded upon this by examining ways to reduce the amount of times the network would need to use the full algorithm to re-calculate the path weights.
7. Conclusion

The energy used by computer networks is growing at remarkable rates, between 2000 and 2012 usage of the internet increased by 566.4%. Yet only 34.3% of the world’s population have access to the internet and most of these connections are low quality low data rate connections. Hence the energy usage of the internet is likely to continue growing for the foreseeable future.

The purpose of this project was to tackle perceived inefficiencies in the telecommunications infrastructure. Today’s networks are always on regardless of traffic levels, individual hardware has no ability to scale its power usage. Networks consume as much energy at 18:00 in the evening when users are streaming large amounts of data as they do at 06:00 in the morning, statistically the period of lowest use.

The proposed solution to this problem was to create a Minimal Energy Network with only the bare minimum amount of switches needed to connect the entire network when active. All un-needed components were put into a sleep state to investigate the power savings. The savings were significant at around 60% during idle periods and surprisingly only dropped to 41.5% during periods of high traffic. Minimum Quality of Service Levels were ensured by activating parallel routes and sharing load during periods of high traffic. Therefore implementing Minimal Energy Networks is a way to extract significant savings from the physical infrastructure.

An improvement to this solution looked at taking the power efficiency of individual hardware components into account. The Minimal Energy Network could then be constructed with routes utilising the most efficient switches in the network. This achieved additional savings of 6.5% compared to a plane minimal energy network. Applying this improvement enabled networks with a combination of older and newer hardware to operate in the most efficient manner possible.

These solutions were applied using an enhanced version of the Floyd-Warshall algorithm. It managed to calculate all routes and converge on a relatively efficient Minimal Energy Network. Unfortunately it ran into scalability issues when trying to run larger networks. Therefore it would be beneficial for an alternative design using a more efficient algorithm such as Johnson’s to be investigated.

Software Defined Networking was used to implement this design and algorithm in a virtualised test network. Changing a network from conventional to SDN requires the new hardware, a centralized controller and a way to creating a dedicated connection between controllers and switches. Hence it may be worthwhile implementing a minimal energy network design in a conventional distributed environment. This
could lead to similar power savings without the need to invest in software defined networking and the extra hardware that this entails.

Overall the controller was successfully tested and performed satisfactorily under controlled conditions. These tests were conducted on a virtual network re-creating real world conditions as much as possible. Therefore the controller should be able to work in a real test network with no modifications. Improvements needed to ensure reliable operation and integration with a real network have been detailed in the future work chapter.

The aims of this project have been investigated and large energy savings achieved at various levels of load. To build on this project several areas identified in the future work chapter should be investigated.
8. Appendix

8.1. Source Code

Please find all source code for this project on the CD attached to the front page of the report.

8.2. IEEE 802.1AB Link Layer Discovery Protocol (LLDP)

IEEE 802.1AB Link Layer Discovery Protocol (LLDP) is a vendor neutral protocol used in link-state networks for devices to advertise their capabilities to other devices. These packets are flooded from each switch and each receiving switch also floods them so that all switches on the network will learn about the original device. All switches in the network will send LLDP packets periodically or when they detect a change in the topology. To prevent these packets from flooding indefinitely they have a time to live value that decrements each time the packet passes through a switch. In addition a special multicast address that 802.1D compliant bridges do not forward is used as the destination MAC. The following information is contained within LLDP packets:

- System name and description
- Port name and description
- VLAN name
- IP management address
- System capabilities (switching, routing, etc.)
- MAC/PHY information
- MDI power
- Link aggregation
9. References


