University of Southern Queensland

Faculty of Health, Engineering & Sciences

Green IT - Dynamic Network Topologies

A dissertation submitted by

Daniel Costantini

in fulfilment of the requirements of

ENG4111/2 Research Project

towards the degree of

Bachelor of Computer Systems Engineering (Honours)

Submitted: October, 2015

Abstract

All engineering disciplines are influenced by the global focus on energy consumption reduction and sustainability. Due to its resident inefficiency, The ICT sector is of particular concern, and there has been extensive work to develop sustainability enhancements to networks and/or network devices. Previous work presented dynamic topology concepts in which the behaviour and topology of the devices and the network react dynamically in response to traffic demands, with the intent of placing devices into standby states to reduce energy consumption. The key aim of this study is to develop a dynamic topology mechanism implementation; it proposes a testbed environment and corresponding dynamic topology mechanism that makes use of two programs: one running on a centralised controller, and one running on the network nodes. The former determines the optimal topology based on energy consumption reductions and network traffic, while the latter uses MPLS to implement the topology. The testbed is used to determine the dynamic topology mechanism's effectiveness and impact on network performance, and does so by subjecting it to controlled variations in network traffic. Quantitative measurements of the dynamic topology mechanism's network performance metrics are presented and analysed relative to baseline measurements. The analysis shows that the dynamic topology mechanism is quite effective, as the effect on network performance is mostly minimal and the reaction to network traffic variations is sufficiently swift. The system takes approximately 30 seconds to react to traffic variations and implement topology changes, and has negligible effect on jitter, packet loss, and the number of out of order packets. However, it produces an average increase in delay of 8 ms, the source of which requires further investigation. This study proves the feasibility of dynamic topology mechanism implementation, and provides a framework for further development and eventual widespread deployment.

University of Southern Queensland

Faculty of Health, Engineering & Sciences

ENG4111/2 *Research Project*

Limitations of Use

The Council of the University of Southern Queensland, its Faculty of Health, Engineering & Sciences, and the staff of the University of Southern Queensland, do not accept any responsibility for the truth, accuracy or completeness of material contained within or associated with this dissertation.

Persons using all or any part of this material do so at their own risk, and not at the risk of the Council of the University of Southern Queensland, its Faculty of Health, Engineering & Sciences or the staff of the University of Southern Queensland.

This dissertation reports an educational exercise and has no purpose or validity beyond this exercise. The sole purpose of the course pair entitled "Research Project" is to contribute to the overall education within the student's chosen degree program. This document, the associated hardware, software, drawings, and other material set out in the associated appendices should not be used for any other purpose: if they are so used, it is entirely at the risk of the user.

Dean

Faculty of Health, Engineering & Sciences

Certification of Dissertation

I certify that the ideas, designs and experimental work, results, analyses and conclusions set out in this dissertation are entirely my own effort, except where otherwise indicated and acknowledged.

I further certify that the work is original and has not been previously submitted for assessment in any other course or institution, except where specifically stated.

Daniel Costantini

0061002765

Acknowledgments

My supervisor, Dr. Alexander Kist, has provided concise and illuminating guidance throughout the project; my friends and family have been vital to the maintenance of my sanity as welcome distractions and eager sounding boards; the Linux community has been a valuable and functionally infinite resource. The following people deserve a special mention, as they agreed to review my dissertation and provide feedback, probably without knowing what they were getting themselves into:

- Courtney Azzopardi
- Matt Boardman
- Chris Fardell
- Jenny Fardell
- Daniel Henry
- Benjamin Sharpe

Daniel Costantini

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The increasing focus on sustainability in modern engineering should be common knowledge for all engineers. Computer systems engineering is no exception, and the broad aim of this project is to reduce the energy consumption of computer networks through the use of dynamic network topologies. Telecommunications networks provide fertile ground for energy consumption improvements, as their design goals are typically limited to resilience to link or node failure and the maintenance of services during peak periods; as a result, networks are mostly under-utilised. The periods of low utilisation can be exploited to reduce energy consumption by rerouting network traffic and placing nodes and links into standby states. Dynamic network topologies can satisfy this need, with an important caveat: the traffic rerouting must facilitate reductions in energy consumption without a disproportionate degradation of network performance. The effectiveness of dynamic topology mechanisms has been theoretically proven through simulation, with one solution reporting energy consumption reductions of 30-50% (Aldraho & Kist 2011*b*).

1.1 Project scope and requirements

Previous work related to dynamic topologies for energy consumption reduction in networks has been focussed on theoretical performance analysis through the use of models and simulation and/or ILP solvers (Aldraho & Kist 2010, 2011*a*,*b*, Aldraho et al. 2012, Amaldi et al. 2011, Chu et al. 2011, Cianfrani et al. 2012, Polverini et al. 2015, Yang et al. 2015, Zhang et al. 2005, 2010). While the results from these simulations and computations provide a proof of concept, the use of models always contains some form of an assumption, and is therefore inherently inaccurate to some degree.

This project aims to develop a testbed that extends the previous work — mainly that of Aldraho & Kist (2011*b*), Aldraho & Kist (2010), and Polverini et al. (2015) — from simulation into a physical implementation. The successful development of a testbed will prove feasibility of implementation of a dynamic topology mechanism in a physical system. Furthermore, it will facilitate the measurement of the mechanism's effects on network performance that are based on a real system, as opposed to simulated systems. This project also aims to reduce the work required for future implementations that will further develop dynamic topology mechanisms.

As stated in the project specification, which has been included as appendix A, the main requirement of the project is to use Linux's MPLS implementation to develop and test a dynamic topology mechanism that reacts to changes in network traffic. Another project specification is the control of the network nodes' transition between the active and standby states with the aim of decreasing total network energy consumption. Note that the node state transitions are performed by modifying the routing information, and the project does not examine the implementation of standby states. It was previously stated that network design goals favour resilience to failure and the maintenance of services during peak periods; this project does not consider the former, but preserves the latter.

The project can be divided into three main requirements: the development of the hardware and software configuration of the testbed, the development of the dynamic topology mechanism programs, and the measurement of the dynamic topology mechanism's performance. The configuration of the testbed is required to provide the capability that will be used by the dynamic topology mechanism, and the performance measurement reflects the suitability of the mechanism's implementation in a live system. As the implementation of standby states is not examined, only the network performance is measured.

1.2 Project methodology

As this project contains software development and other elements of uncertainty, such as the testbed implementation requirements, the development of the dynamic topology mechanism is based on agile methods that can adapt to changes in specifications. The project begins with a review of the current literature to prevent unnecessary rework and to determine the project's "user stories", from which the initial specifications of the dynamic topology mechanism testbed are be determined. After the initial research has been completed, which is detailed in chapter 2, the incremental development, experimentation, and research can commence. The development is focussed on producing functionality for the dynamic topology mechanism, and includes the development of the testbed and the dynamic topology mechanism programs. This often reveals additional functionality and/or component requirements, prompting additional research and experimentation prior to resuming development.

Once incremental development reaches a point where the testbed is successfully routing traffic using the dynamic topology mechanism, the focus shifts to performance validation to determine the suitability of this implementation. The performance validation is aimed at discerning the dynamic topology mechanism's effect on network performance, and the degree to which the project requirements are met. Once the performance validation methods have been developed, and the dynamic topology mechanism's performance has been measured, the system is dismantled to prevent unpredictable use. The procedure that was used to design the system, and the resultant design itself, are detailed in chapters 3 and 4 respectively, while the testing methodology and associated results and discussion are shown in chapters 3 and 5 respectively.

1.3 Dissertation overview

This dissertation is organised as follows:

Chapter 2 analyses previous work that relates to the dynamic topology mechanism

Chapter 3 details the methodology used to develop the system and dynamic topology mechanism, and the performance validation methods used to test the resultant design

Chapter 4 fully describes the final system design, including the hardware and software configuration and the programs used to implement the dynamic topology mechanism

Chapter 5 examines the results of system tests and discusses the performance of the dynamic topology mechanism in terms of the project requirements

Chapter 6 concludes the dissertation and proposes potential future work related to the dynamic topology mechanism

Chapter 2

Related Work

There has been significant research into areas directly or indirectly related to this project, and this chapter contains a review of the current literature. The literature review details the motivation behind the project, previous implementations of dynamic topology mechanisms, energy aware traffic engineering, the applicability of Multi-Protocol Label Switching (MPLS), topology optimisation and associated heuristics, traffic matrix calculation and measurement, and software defined networking.

2.1 Literature review

The energy consumption of the ICT sector is of significant concern; recent attention has been directed at optimising the design and use of networks and network components with consideration given to the reduction of energy consumption. The main impetus for the reduction in energy consumption of network devices is a combination of their large contribution to the ICT carbon footprint, their current inefficiency, and their role as an enabling technology for further energy consumption reductions.

2.1.1 Motivation

A 2012 estimate of the ICT sector's contributions to global carbon emissions was 2% (Koenigsmayr & Neubauer 2015), and it is estimated that carbon emissions from network devices contributes between 30% (Gartner 2007) and 37% (Webb 2008) of ICT emissions. The energy consumption of network devices is exacerbated by the almost criminally inefficient utilisation of telecommunications networks, which typically ranges from <30% (Nedevschi et al. 2008) to <50% (Fraleigh et al. 2003); this is due to the over-provisioning of networks to maintain connectivity and Quality of Service (QoS) during peak periods, which are present for only short periods in a diurnal traffic cycle (Bolla et al. 2011). Furthermore, current network devices themselves are also inefficient, with the energy cost of data transmission at 0.128-0.225 Joules/Byte; compared to the benchmark of 802.11b radios, which use ∼1.6 μJ/B over a 100m link, the wireless link is 2-3 times more efficient (Gupta & Singh 2003). While the cost per byte of traffic has been decreasing due to performance improvements, the rate at which line card speeds increase has resulted in an overall increase in power density (Chabarek et al. 2008).

ICT-based low carbon technological solutions are estimated to reduce 15% of global GHG emissions by 2020, and the energy-aware focus of ICT is an attempt to secure the enabling effect of ICT in other sectors (Koenigsmayr & Neubauer 2015). Further to this, increased energy efficiency of network devices would enable greater deployment, particularly in developing countries, and allow greater network availability in the event of a disaster when power is scarce to retain data and connectivity for longer (Gupta & Singh 2003).

It is difficult to market new technologies to the carriers that maintain the internet's core infrastructure. They have previously demonstrated satisfaction with over-provisioning, as well as techniques such as traffic caching and compression, rather than addressing the root cause (Roberts 2009). If history is any indication of the future of the internet's core infrastructure, there will need to be a more robust solution that moves away from these temporary fixes, which can be seen as analogous to the previous use of VLSM and NAT to temporarily solve the rapidly decreasing IPv4 address space. However, in order for a solution to be widely accepted, it should align with the requirements stated in Coiro et al. (2013):

1. No new communication protocol or new functionalities in current routing and signalling protocols
2. Interoperability with standard networks
3. Automatic adaptation to network condition
4. No packet loss
5. No congestion even for limited periods

2.1.2 Energy consumption reduction in computer networks

The approaches to achieving the reduction in energy consumption of network devices are many and varied, and include the design of networks and network devices, link and node shutdown and standby, and traffic engineering. Most of them, if not all, either implicitly or explicitly adhere to the aforementioned requirements, as does the dynamic topology mechanism described in chapter 4.

Traffic engineering for load balancing and QoS in general

When considering the use of networks, there are several dynamic topology mechanisms that are not specifically focused on facilitating or directly influencing power consumption reduction, but which are the foundation for related works.

Traditional routing protocols, such as OSPF, support dynamic reconfiguration after topology changes. However, these are not suitable for the implementation of dynamic topologies as they can take several minutes to converge (Aldraho et al. 2012). MPLS is quite versatile in terms of traffic engineering, and can provide most of the functionality of the previously used overlay model, which was implemented to address traffic engineering shortcomings of conventional IP networks, at low cost and in an integrated manner (Elwalid et al. 1998, Awduche 1999). As these energy aware traffic engineering methods typically require explicit route definition, the majority of them state that MPLS must be used. However, as highlighted by Gupta & Singh (2003), consideration needs to be given to routing protocols (OSPF, EIGRP, IS-IS, RIPv2, etc.) and protocols higher in the OSI layer when testing the effectiveness of these dynamic topology mechanisms. Suryasaputra et al. (2005) demonstrates the versatility of the MPLS traffic engineering by using the NS-2 network simulator to implement explicit routing using MPLS with two different objectives: maximisation of residual link capacity, and minimisation of network cost in terms of link weights. The dynamic topology mechanism described in chapter 4 makes use of MPLS for its low cost and explicit route definition.

The optimisation problem addressed by Zhang et al. (2005) is aimed at determining the best splitting ratio for multiple paths over a range of circumstances by analysing multiple traffic matrices and formulating a compromise between the worst-case and average-case scenarios; it also uses MPLS to implement the resultant optimal configuration. As the problem is NP hard for even a single traffic matrix, heuristics presented by Fortz & Thorup (2000) are used. Similarly, but conversely, the work of Ben Ameur et al. (2002) provides an optimal routing algorithm that routes the traffic for a pair of nodes along only one path, defined by an MPLS Label Switched Path (LSP). It uses the shortest path if possible, but also attempts to maximise the remaining capacity of the links to protect the network in the event of an increase in traffic. Kvalbein & Lysne (2007) showed an alternative use of multiple topologies that does not utilise MPLS. Their multiple topologies each have one or more links excluded to spread traffic across different paths by modifying split ratios. This method does not use explicit routing, like MPLS does, but simply distributes the traffic among several logical networks that utilise the same physical network components. This can respond quickly to traffic dynamics without requiring a demand matrix, but does not take power reductions into account.

A common aim in the use of dynamic topologies is to ensure the resilience of the network in the event of link failure. Wang et al. (2010) uses an extension of MPLS called MPLS-ff to perform reconfiguration of the network when a link failure is detected. It involves an offline precomputation phase, which takes a few seconds for a 20 node network, and an online reconfiguration phase. ICMP is used to advertise link failures across the network and traffic is then distributed among the remaining precomputed alternative links. Hundessa & Domingo-Pascual (2002) introduces a method for ensuring minimal/no packet loss during topology changes, but is focused on unplanned changes due to failures rather than traffic dependent reconfigurations. It uses pre-defined alternate LSPs for a fast switchover, and uses buffers and link failure detection to minimise packet loss. While this project does not consider network reliability and failure resilience, the concepts are still useful.

Traffic engineering for energy consumption reduction

One group of methods uses the foundation work shown above and is centred around modifying the network topology to allow specific energy saving methods. These energy saving methods include link and/or node shutdown, and several methods have been devised to allow link and/or node standby, or a combination of both. A unique approach to dynamic topologies that indirectly reduces GHG emissions considers the energy production method used to power the network devices, and aims to reduce the energy consumption of those powered by high GHG emitting energy sources (Wang et al. 2012). Other methods are more direct and aim to reduce power consumption in general; Chabarek et al. (2008) proposes power awareness in the design of network devices, the allocation of those devices in the network, and the use of power aware protocols. Their definition of power aware protocols includes the use of dynamic topologies to allow network devices and/or components to move into a state with lower power consumption, and is a common area of research. Table 2.1 is derived from the work of Coiro et al. (2013), and shows an overview of a number of energy aware traffic engineering techniques. As can be seen, the majority of techniques rely on centralised control, many of them can be applied to and/or require MPLS networks, and are typically routed using a shortest path or flow-based method; Coiro et al. (2013) also proposes their own energy aware traffic engineering technique that is distributed and uses MPLS. All these techniques are focused on rerouting traffic to allow the maximum allowable number of links to be shutdown to reduce energy consumption, while preserving the functionality of the network as a whole.

The work of Bolla et al. (2011) also aims at reducing power consumption by rerouting traffic and shutting down unused links, but more explicitly specifies the logical connection rerouting and line card shutdown methods. It uses a central entity to analyse the traffic matrix and iteratively remove the line cards with the lowest traffic load and test whether the network can still support the traffic matrix. It states that the central entity should be dedicated to collecting traffic load information and consequently applying the traffic engineering criterion to perform the virtual links' reconfiguration while meeting QoS constraints, and that the diurnal fluctuation of traffic is the motivation behind the selection of only a few reconfiguration thresholds at 25%, 50%, and 75% of maximum demand. Yang et al. (2015) similarly aims to shut down unused line cards, but focuses specifically on the minimisation of trunk link utilisation, i.e. single logical links that are

EA-TE Technique	Searching Algorithm	Operation architecture	Network Scenario	Routing strategy
(Chiaraviglio et al. 2012)	Heuristic, iterative greedy	Centralised	Pure IP	SP
(Cuomo et al. 2012)	Heuristic, iterative greedy	Centralised	Pure IP	SP
(Amaldi et al. 2011)	Heuristic, iterative greedy, MILP based	Centralised	Pure IP	SP, link weight variation
(Lee et al. 2012)	MILP, Lagrangian relaxation	Centralised	Pure IP	SP, link weight variation
(Eramo et al. 2012)	Heuristic, SPT exportation	Centralised	Pure IP	Modified SPT
(Zhang et al. 2010)	Heuristic, MILP based	Centralised	IP/MPLS	SP + Flow-based
(Takeshita et al. 2012)	Exhaustive search	Centralised	IP/MPLS	SP
(Vasi et al. 2011)	Not defined	Distributed + offline	IP/MPLS	Flow-based
(Vasi & Kosti 2010)	Load adaptation on multiple paths	Distributed	IP/MPLS	Flow-based, multiple paths
(Kim et al. 2012)	Ant colony-based next hops selection	Distributed	Pure IP	Destination based
(Coiro et al. 2013)	Routing-based, load-dependent link weights	Distributed	IP/MPLS	Flow-based, single path

Table 2.1: Energy Aware Traffic Engineering Techniques (Coiro et al. 2013)

comprised of several parallel physical links. By minimising trunk utilisation, less parallel links are required, and line cards can be shutdown. The proposed algorithm is distributed and uses heuristics and a hop-by-hop routing mechanism to determine the network path. The work of Pan et al. (2015) is related to the above dynamic topology mechanisms that aim to minimise the active links, as it describes an enabling technology; it proposes a modification of the line card boot sequence to drastically reduce boot time, with results in a 127.27 ms transition from the sleep to active state in their prototype hardware.

Additional techniques have been developed to expand the energy consumption reduction options to include more than just the minimisation of active links. Chu et al. (2011) presents a relatively simple method of reducing power consumption during off-peak periods by using the predicted traffic matrix and topology to determine the set of routers that can be turned off while satisfying traffic demands. It then determines the explicitly defined LSPs to reroute the traffic that would ordinarily traverse the now shutdown nodes. A variation of this discusses changes to routing decisions, based on network load, to aggregate traffic over fewer devices and links to put devices to sleep, and suggests power savings from idle components by either clocking them at a slower rate or putting them to sleep completely (Gupta & Singh 2003).

Aldraho & Kist (2011*b*) suggest both an energy saving method for nodes and links and a method of traffic engineering to allow its implementation, and is the main inspiration for this project. The traffic matrix is analysed with the aim of maximising the number of nodes in the a standby state. The network node standby power model is initially described by Aldraho & Kist (2010), and subsequently developed by the same authors (Aldraho & Kist 2011*b*, Aldraho et al. 2012, Aldraho & Kist 2011*a*) and others (Polverini et al. 2015, Suryasaputra et al. 2005); it proposes the temporary removal of the routing functionality, which consumes approximately 80% of a router's energy and space (Roberts 2009). Note that when MPLS is used to facilitate the modification of topologies to support this standby mode, the MPLS-related modules must remain active to maintain labels and forwarding tables in the standby devices (Aldraho & Kist 2011*a*).

Aldraho & Kist (2011*a*) provides a solution for implementation of dynamic networks using multiple topologies through a number of MPLS label sets, which requires only minor additions at ingress Label Switch Routers (LSRs). This implementation divides the roles of LSRs into three functions: Network Management Function (NMF), Topology and Flow Tracker (TFT), and Power Management Function (PMF). The NMF is the centralised optimisation and control of the network, which signals the TFT and PMF on each of the network's nodes to control the LSPs and active/standby transitions respectively. This implementation also uses timers to control the overlap period of the old and new topologies to minimise the effects on traffic. As further explored by Aldraho et al. (2012), the length of the timer has minimal impact on the power savings in the network, and is below 1% for all instances up to and including 10 power state changes in a 24 hour period for a timer value of 180 seconds.

2.1.3 Topology optimisation and traffic matrices

The solution of the optimisation problems in the vast majority of dynamic topology mechanisms involve the use of Mixed-Integer Linear Programming (MILP) solvers; the duration of the solution generation can be in the order of minutes or hours (Aldraho & Kist 2011*b*), and the calculation and resultant control implementation typically needs to be centralised (Gupta & Singh 2003). When testing the effectiveness of these optimised topologies, OSPF is a common benchmark, as it is widely used (Suryasaputra et al. 2005). Similarly, this project also uses OSPF routing as the benchmark of the dynamic topology mechanism's performance measurements, as described in chapter 3.

In order to respond to the current traffic demands of the network, the optimisation methods generally require an accurate traffic matrix. Traffic matrices are difficult to measure in IP networks, but Awduche (1999) shows that statistics derived from MPLS LSP tunnels can be used to construct a rudimentary traffic matrix. Schnitter & Horneffer (2004) stated that the use of MPLS LSP statistics may not be usable for many networks as it requires the logical network to be fully meshed and is not scalable. However, if the nodes all perform both ingress and egress functions, as is the case in the test networks in the work of Aldraho & Kist (2011*b*) and in this project, the logical network is already fully meshed, and this is a suitable method to directly measure the traffic matrix. As an alternative to traffic matrix measurement, Ohsita et al. (2010) provides a method to estimate the traffic matrix using monitored link loads and the paths between source and destination nodes. This method also includes a reduction in estimation error by performing iterative calculations during the incremental reconfiguration of the network, the effectiveness of which is proven by simulation.

As stated by Aldraho et al. (2012), Aldraho & Kist (2011*b*) and Aldraho & Kist (2011*a*), the optimisation problems are Non-deterministic Polynomial-time (NP) hard, so heuristics may be required for large networks to find optimal or near optimal solutions for standby node selection. Aldraho & Kist (2010) presents two such heuristics, which iteratively transitions active nodes into the standby state until the remaining active links' utilisation becomes unacceptable, at which point it restores the latest node. The two heuristics differ in their determination of node standby priorities: the Lightest Node First (LNF) algorithm considers the node with the least connections first, while the Least Loaded Node (LLN) algorithm considers the node that handles the least traffic first. The former outperforms the latter, and the performance of both decreases sharply compared to the ILP alternative as traffic load increases. A similar heuristic is presented by Cianfrani et al. (2012), and further developed by Polverini et al. (2015), that determines the minimum number of active nodes, places the remaining ones in standby, and determines the standby nodes' active outgoing link. It is based on the Floyd-Warshall algorithm, and can be solved for a 100 node network in approximately 25 seconds. Their algorithm always outperforms the Floyd-Warshall node standby heuristic, and is only slightly outperformed by the heuristic presented by Aldraho & Kist (2010) for networks with utilisations below 25%. However, note that this algorithm uses a different standby state that uses one outgoing link but accepts all incoming links; this is similar to the "bridged-local" state defined by Aldraho & Kist (2011*b*), while the work of Aldraho & Kist (2010) is based on the "bridged-all" state in the work of Aldraho & Kist (2011*b*). This project's standby state is based on the aforementioned "bridged-all" state, which has been considered in the implementation of optimisation heuristics, and the heuristics themselvs are a combination of several of the aforementioned algorithms.

Wang et al. (2014) examines energy aware traffic engineering specifically in software defined networks, which is very similar to that performed in physical networks. As with previously discussed methods, it also focuses on rerouting traffic to shutdown links. Of note is the scalability, effectiveness, and execution speed of the heuristic algorithm it presents; it executes much more quickly than the ILP optimisation — 63ms compared to 12hrs for a 25 node network — and gives similar results, and claims that it is scalable to any network size. The authors do highlight the fact that the network is more vulnerable to link failures and traffic bursts with the increased traffic load on the remaining active links, but qualifies this by stating that there is currently no research on energy aware traffic engineering that also considers reliability.

2.1.4 Software defined networking and virtualisation

While the physical systems that are utilised in this paper are not dedicated network devices, they maintain the same functionality through Software-Defined Networking (SDN) and virtualisation. As stated by Bozakov & Papadimitriou (2013, p. 196): ‘The implemented virtual routers are functionally and logically indistinguishable from traditional routers’. This project uses a design centered on SDN, as described in chapter 4, so the confirmation of the virtual routers’ effectiveness provides a reassuring precedence.

The use of SDN aligns well with the use of the network devices; SDN using OpenFlow allows centralised visibility and dynamic reprogrammability, and the use of multiple flow tables can make flow management more flexible and efficient (Akyildiz et al. 2014). These characteristics have either been included in the requirements of the traffic engineering design stated by Aldraho & Kist (2011*b*), or facilitate the design’s implementation. Akyildiz et al. (2014) explores a wide range of traffic engineering options that use OpenFlow to implement SDN. It states that fully meshed LSPs are generally not scalable, but the simplicity of SDN can alleviate the complexities of the MPLS control plane with scalability and efficiency (Sharafat et al. 2011). Han et al. (2015) explores the effects of packet loss and routing inconsistencies in OpenFlow switches as a result of flow table modifications. This is exacerbated by the fact that a single policy modification can translate into multiple potentially parallel and asynchronous updates across different flow tables in a single device. The solutions proposed are the blocking and draining of the flow, and the use of a shadow table to which the changes are made and which is swapped with the active table once configuration is complete. As described in chapter 4, these factors have been considered in the solution’s design.

Chapter 3

Methodology

The development of the dynamic topology mechanism, and the testbed on which it runs, followed a defined procedure based on Agile project management concepts. The development methodology and the testing procedure that determines the effect of the dynamic topology mechanism on network performance are described in this chapter. The resultant system design and test results are detailed in chapters 4 and 5 respectively.

3.1 Development methodology

The procedure that has been used to develop the dynamic topology testbed has been separated into the following phases: Research, investigation, and experimentation; Incremental development; Performance testing; Analysis; and Decommissioning. The testing phase is described in the following section, the analysis phase is described in chapter 5, and the remaining phases are described below.

The two initial phases consumed the majority of the project's time, as they are related to the creation of the dynamic topology mechanism and the resolution of issues associated with the specific technology being used. As described below, the current status of research was first determined, followed by experimentation and investigation to prepare the system and bestow the functionality required of the dynamic topology mechanism. Once this was completed, the dynamic topology mechanism's development could begin, which was performed using an incremental approach based on the concepts of Agile project management and test driven development. Note that the two sub-phases described below were executed in parallel, as the development of the dynamic topology mechanism often presented problems that had to be solved by additional research, investigation, and experimentation. The resultant design is detailed in chapter 4.

3.1.1 Research, investigation, and experimentation

The literature review presented in chapter 2 represents the initial research to determine the current status of work related to the dynamic topology mechanism that is the focus of this project. From this, the project's direction was determined, and an initial appraisal of the work required to achieve the project's aims could be completed.

From the initial research, the components and capabilities required of the system could be determined. Once the physical components were selected based on the required functionality, experimentation and further research was performed to develop a software configuration for the system, such that it could support the dynamic topology mechanism.

Once an initial solution for the software configuration was determined, the incremental development described below could begin. At several points during the incremental development, further functionality was deemed necessary; to provide a solution for the software configuration, further investigation, experimentation, and testing was performed. As such, the system's underlying software configuration was changing almost as frequently as the dynamic topology mechanism during development.

3.1.2 Incremental development

A dynamic topology mechanism based on that described by Aldraho & Kist (2011*b*) and Aldraho et al. (2012) was developed using an incremental development methodology based on the concepts of Agile project management (Schwalbe 2014). The "user stories" were initially defined based on requirements discerned during the literature review, but were amended when additional requirements were identified as a function of the project's progression. An amendment to the list of user stories typically had cascade effects on the underlying software configuration of the system, and prompted additional research, investigation, and experimentation, as alluded to above.

The development of the dynamic topology mechanism was accompanied by an unscheduled, coincidental configuration of the test network and overall system. As the development methodology also contained elements of test-driven development, the test network to be used was a main focal point. As such, to test elements of the software, various portions of the network were setup and configured during development. Note, however, that full setup and configuration of the network was not performed until the dynamic topology mechanism's development had been completed, as the finalisation of the mechanism prompted the progression to baseline determination, described below.

3.1.3 Decommissioning

Once all the required data had been measured and it was confirmed that no additional development or testing was required, the network components were dismantled to prevent unpredictable future use. The likelihood of malicious use is remote, but the precaution has be taken nonetheless. Setup and testing data has been preserved to allow replication and further analysis if necessary.

3.2 Performance validation

The solution to the requirements detailed above must be tested to determine whether it is sufficient. This testing is focussed on network performance, and can be divided into the four sub-phases described below: measurement method determination, test scenario development, baseline determination, and benchmarking. A discussion of the results of the tests, along with the results themselves, are shown in chapter 5.

3.2.1 Measurement method determination

The network performance metrics to be used are determined from common practice (Simsek & Pospiech 2013), and include the latency, jitter, packet loss, and packets received out of order. These measurements need to be taken for each source/destination node pair to provide easy comparison between the test scenarios described below. The network's traffic matrix is controlled through the generation of traffic using the program "iPerf". For UDP streams, iPerf automatically produces a report of the connection's jitter, packet loss, and packets received out of order, among other statistics. The standard ping tool can be used to provide the latency measurement. The programs used to collect these network performance data on the hosts and process it on the controller are detailed in chapter 4.

3.2.2 Test scenario development

Prior to the performance of any testing and the associated results measurement, the test scenarios first need to be determined. These scenarios are constructed to allow the collection of performance data that is easily comparable between configurations. Note that in this context, "configurations" refers to the routing mechanism being used: OSPF, MPLS, or the dynamic topology mechanism. Each of these are explained in further detail in the following sections.

As the main focus of the testing is to determine the dynamic topology mechanism's performance when compared to conventional routing, the test scenarios are developed relative to the dynamic topology mechanism and applied to all configurations. The two key aspects that are controlled are the traffic matrices and the interval between traffic matrix changes.

The traffic matrices are varied during the measurement interval to determine the effects of the dynamic topology mechanism during topology changover. To this end, the traffic matrix progression shown in tables 3.1, 3.2, and 3.3 below is applied to all configurations, and have been selected to elicit a specific topology selection when the dynamic topology mechanism is used. Note that the traffic matrices are constructed by cumulatively generating traffic over the test's duration. Figure 3.1 depicts a representation of this traffic generation method that applies to all demands other than those originating from node six. The cumulative generation of traffic allows the network performance measurements to be taken during topology changeover, as the program used to generate the traffic also provides the majority of the performance measurements. To allow adequate time to measure the performance of both the topology changeover and the stable topology, each traffic matrix is generated for a period of 10 minutes.

The traffic matrices shown in tables 3.1, 3.2, and 3.3 have been selected based on the topology selection they will elicit from the dynamic topology mechanism. The topologies

		Destination node								
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**Total**
Source node	**1**	-	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.14
	2	0.02	-	0.02	0.02	0.02	0.02	0.02	0.02	0.14
	3	0.02	0.02	-	0.02	0.02	0.02	0.02	0.02	0.14
	4	0.02	0.02	0.02	-	0.02	0.02	0.02	0.02	0.14
	5	0.02	0.02	0.02	0.02	-	0.02	0.02	0.02	0.14
	6	0.02	0.02	0.02	0.02	0.02	-	0.02	0.02	0.14
	7	0.02	0.02	0.02	0.02	0.02	0.02	-	0.02	0.14
	8	0.02	0.02	0.02	0.02	0.02	0.02	0.02	-	0.14
Total		0.14	0.14	0.14	0.14	0.14	0.14	0.14	0.14	**1.12**

Table 3.1: Traffic matrix 1 (Mbps)

		Destination node								
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**Total**
Source node	**1**	-	0.05	0.05	0.05	0.05	0.05	0.05	0.05	0.35
	2	0.05	-	0.05	0.05	0.05	0.05	0.05	0.05	0.35
	3	0.05	0.05	-	0.05	0.05	0.05	0.05	0.05	0.35
	4	0.05	0.05	0.05	-	0.05	0.05	0.05	0.05	0.35
	5	0.05	0.05	0.05	0.05	-	0.05	0.05	0.05	0.35
	6	0.10	0.10	0.10	0.10	0.10	-	0.10	0.10	0.70
	7	0.05	0.05	0.05	0.05	0.05	0.05	-	0.05	0.35
	8	0.05	0.05	0.05	0.05	0.05	0.05	0.05	-	0.35
Total		0.40	0.40	0.40	0.40	0.40	0.35	0.40	0.40	**3.15**

Table 3.2: Traffic matrix 2 (Mbps)

		Destination node								
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**Total**
Source node	**1**	-	0.10	0.10	0.10	0.10	0.10	0.10	0.10	0.70
	2	0.10	-	0.10	0.10	0.10	0.10	0.10	0.10	0.70
	3	0.10	0.10	-	0.10	0.10	0.10	0.10	0.10	0.70
	4	0.10	0.10	0.10	-	0.10	0.10	0.10	0.10	0.70
	5	0.10	0.10	0.10	0.10	-	0.10	0.10	0.10	0.70
	6	0.10	0.10	0.10	0.10	0.10	-	0.10	0.10	0.70
	7	0.10	0.10	0.10	0.10	0.10	0.10	-	0.10	0.70
	8	0.10	0.10	0.10	0.10	0.10	0.10	0.10	-	0.70
Total		0.70	0.70	0.70	0.70	0.70	0.70	0.70	0.70	**5.60**

Table 3.3: Traffic matrix 3 (Mbps)

that correspond to the given matrices are shown in figures 3.2, 3.3, and 3.4 below. Note that grey nodes are those that have been selected to be placed in a standby state, and

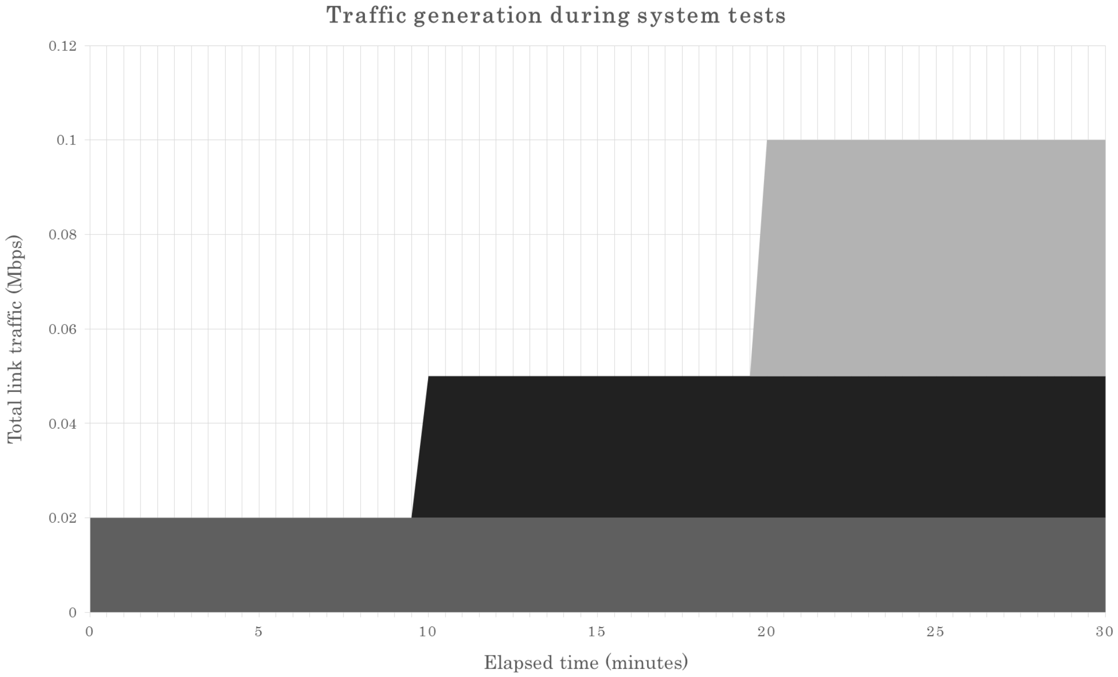

Figure 3.1: Traffic generation during system tests

which have only two active interfaces: the connection to their access network, and a connection to one active node in the network. A full description of the system's network and the dynamic topology mechanism is provided in chapter 4, and an analysis of the traffic measurement accuracy is provided in chapter 5.

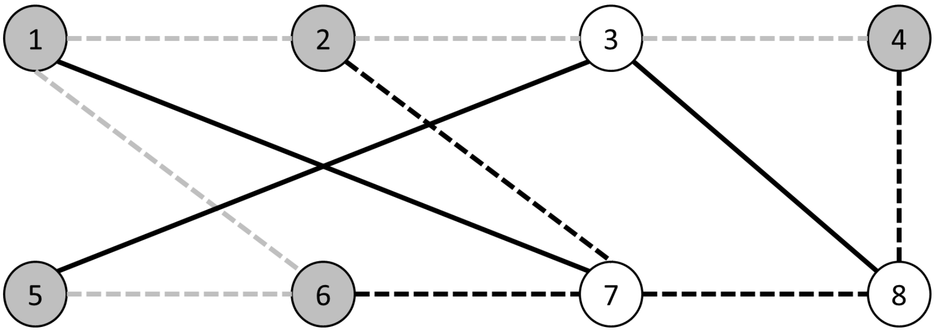

Figure 3.2: Topology selection from traffic matrix 1

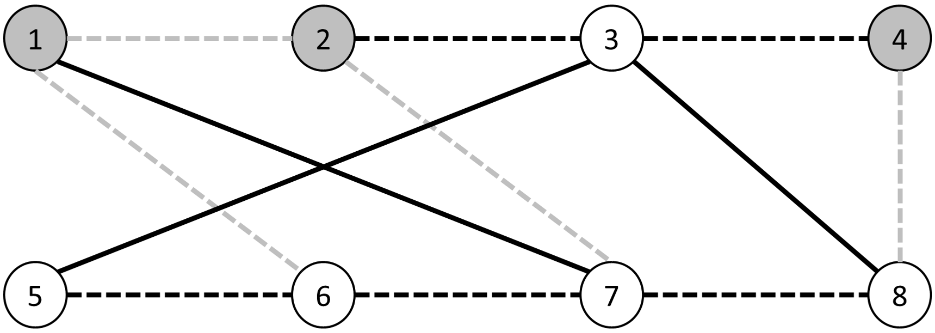

Figure 3.3: Topology selection from traffic matrix 2

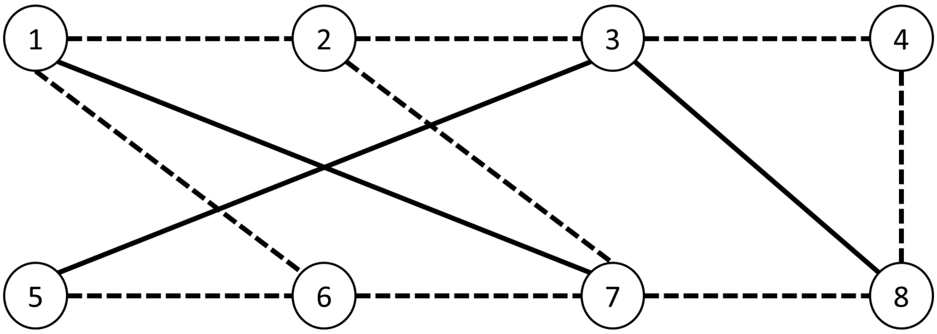

Figure 3.4: Topology selection from traffic matrix 3

3.2.3 Baseline determination

This phase was concerned with the collection of data that can be used as a reference point for the dynamic topology mechanism's performance analysis, as shown in chapter 5. To take the measurements, the system was fully setup using the test network and software configuration that was previously determined, which is detailed in chapter 4. To allow a suitable comparison of the data, minimal changes were made to the underlying software between tests. The test cases that were used were based on conventional network operation: all nodes in an active state, traffic routed using OSPF; all nodes in an active state, traffic routed using MPLS with a static topology. OSPF routing was selected as it is widely used in current networks, and is frequently used as a baseline for network performance (Suryasaputra et al. 2005). MPLS was selected to isolate the effects of the dynamic topology mechanism, which also uses MPLS.

As the intent is to provide a comparison for the performance of the dynamic topology mechanism, the same traffic matrices are used for both the baseline determination and benchmarking, the latter of which is described in general below. For each of the test configurations described above, using a range of traffic matrices, network performance metrics were measured as described above.

This phase was withheld until the conclusion of the development phase. This prevents a premature collection of data that may be made unusable by a change in the underlying software configuration. Note that the conventional network operation using OSPF could have been examined very early in the project, as the required software is quite easy to configure. However, the use of a standard software configuration, i.e. one that can be used for both conventional network operation using OSPF and for network operation using the dynamic topology mechanism, allows the performance variations to be attributed to the differences in network operation, rather than differences in the underlying software configuration.

3.2.4 Benchmarking

The data to be used in the analysis of the performance of the dynamic topology mechanism was collected in this phase. There is little explanation required at this point, as the same procedure is used for the baseline determination, with the key difference being that the dynamic topology mechanism is used to place nodes in a standby state. The measurements that were made in this phase are detailed and analysed in chapter 5.

Chapter 4

Dynamic Topology Mechanism and System Design

This chapter describes the final system design that is the result of the incremental development and testing outlined in chapter 3. The sections below describe a holistic view of the system, the specific hardware and software used for the controller and network nodes, and the dynamic topology mechanism's operation. Note that the procedure used to configure the components of the system is provided at appendix B, and a full listing of the dynamic topology mechanism's source code is provided at appendix C.

4.1 Overall system functionality and configuration

The system's logical layout is shown in figure 4.1 below, and is based on Telstra network AS1221, which has been used as a test network in several studies, namely the work of Aldraho & Kist (2010) and Aldraho & Kist (2011*b*). The inter-node links have speeds of 1 or 10 Mbps, with dashed lines in figure 4.1 signifying the former, and solid lines signifying the latter; the link speeds have been scaled down by a factor of 1000 from those in AS1221 to allow the network to be replicated using readily available hardware. Each node is capable of routing traffic using both OSPF and MPLS, with the latter being the foundation of the dynamic topology mechanism described below; each node must also monitor the controller's current topology selection, and implement topology changes as appropriate. The hosts connected to the access network of each of the nodes are used to

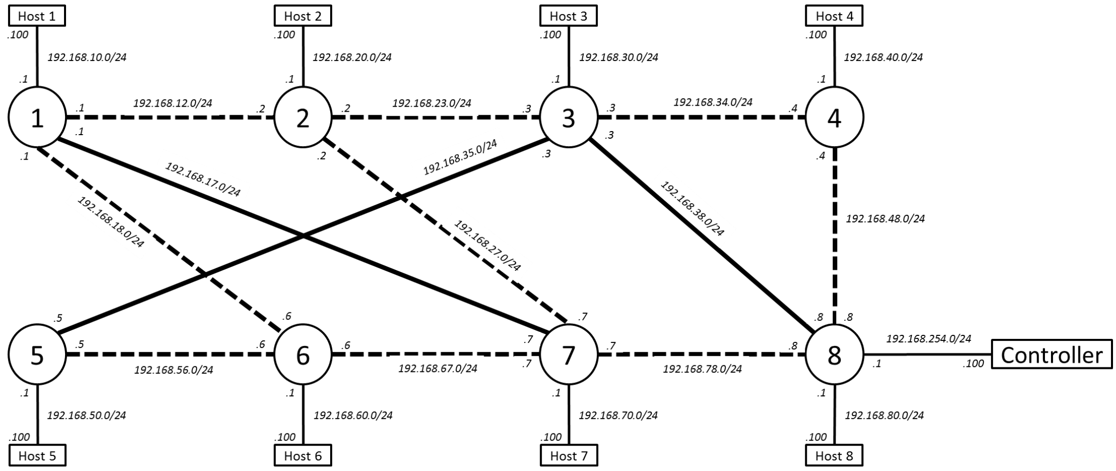

Figure 4.1: Logical system layout

simulate a local network of hosts through traffic generation and reception to and from each of the seven other access networks. The controller connected to node eight constructs a traffic matrix by monitoring the traffic between each of the node's access networks, and uses this data to optimise the network topology and communicate the topology change to the nodes. Node eight was selected as the controller's connection to the network, as it is in an active state for 97% of the randomised traffic matrices applied by Aldraho & Kist (2011*b*). According to the same work, node 3 was never in a standby state, but was not selected as the controller's connection to avoid processing bottlenecks, as it's connected links total bandwidth is 22Mbps, while node 8's connected links total bandwidth is only 12Mbps.

The hardware and software required to deliver the previously stated functionality, and to provide a foundation for the operation of the dynamic topology mechanism, is described below.

4.1.1 Hardware configuration

The hardware configuration is quite straightforward and relies on readily accessible components. Each of the eight node and host pairs runs on a single Raspberry Pi model 1B+ with an 8GB SD card for its main memory, and USB to Ethernet adapters to augment the Raspberry Pi's standard capability of a single Ethernet connection. The controller is a Dell Vostro V13 with no additional hardware.

Providing power to all eight Raspberry Pis simultaneously required a moderately unique solution, and relies on a four-way split of the DC provided by two AC/DC converters. The total energy consumption of all devices at 100% CPU utilisation and all links active, using the formulae provided by Kaup et al. (2014), is approximately 34.6W, i.e. 6.92A at 5V; this is evenly divided between the two groups of nodes, nodes 1-4 and nodes 5-8. While this results in each group of four nodes drawing 3.46A at 5V – 460mA more than the rated current of the DC adapters – it is modelled on the worst case scenario and is not likely to occur; the fact that the Raspberry Pi model 1B+ requires 0.5-1W less than the Raspberry Pi model 1B on which the model was based also reduces the likelihood of this scenario occurring.

The physical devices utilised by the system are shown in figure 4.2 below, and include the following components:

- **240V AC General Purpose Outlet:** Household G.P.O., provides 240V AC up to 10A
- **AC/DC converter - 5V 3A:** Converts the 240V AC from the G.P.O. into 5V DC with a maximum current draw of 3A, i.e. maximum current draw of 62.5mA at 240V
- **4-way DC power splitter:** Splits the single 5.5mm x 2.5mm DC connector given by the DC adapter to provide four physical 5.5mm x 2.5mm DC connections
- **DC power to micro USB adapter:** Converts the 5.5mm x 2.5mm DC connector into micro USB connector, which can be used to power the Raspberry Pi model 1B+
- **Raspberry Pi model 1B+:** Physical system that provides functionality of a single host and node pair
- **USB2.0 10/100Mbps Ethernet adapter:** Provide network connectivity; the Raspberry Pi model 1B+ only has a single on-board 10/100 Ethernet port but four USB ports
- **8GB micro SD card:** Main storage for the Raspberry Pi model B+.
- **Dell Vostro V13:** Physical system to provide central controller functionality. Note that this includes a proprietary AC/DC converter, not shown in figure 4.2

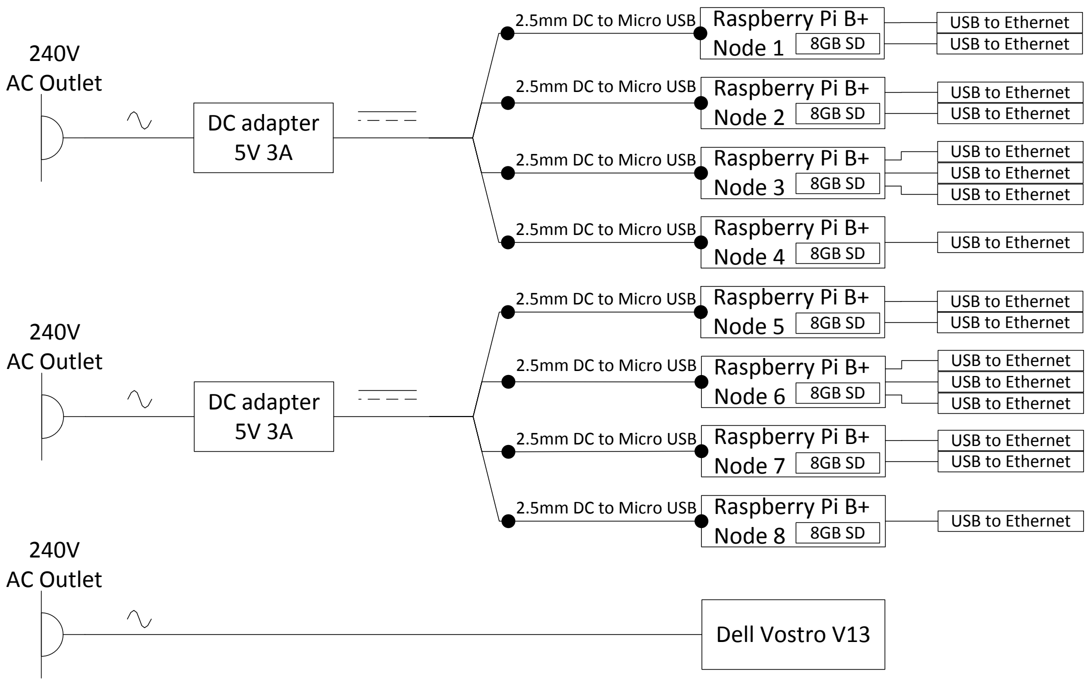

Figure 4.2: Physical system layout

- **Ethernet cables:** Provides network connectivity between devices. Not shown in figure 4.2 for simplicity

4.1.2 Software configuration

The software configuration is slightly more involved than the hardware configuration, particularly for the nodes and hosts running on the Raspberry Pis, but still uses readily available components. The requirements are quite specific, as the software configuration provides a foundation for the capability utilised by the dynamic topology mechanism. This section details the software configuration of the controller, nodes, and hosts. A full setup instruction to recreate the controller's, network nodes', and hosts' configuration for the dynamic topology mechanism is provided at appendix B.

Controller

The controller's software configuration is quite lightweight, as the majority of the functionality is either provided by the operating system by default, or is provided by the dynamic topology mechanism. The operating system and the software packages used on the central controller are listed in table 4.1 below. Note that this is not a comprehensive list of installed software packages; the dependencies for the packages listed above, in addition to the operating system's default packages, were also installed.

Software name	Version	Description
Ubuntu	14.04.2	Linux operating system.
sFlowTool	3.35	sFlow analyser. Collects sFlow messages.
NTP	1:4.2.6.p5+dfsg-3ubuntu2.14.04.3	Network Time Protocol server.

Table 4.1: Controller software

sFlowtool is used to collect the sFlow messages generated by each of the nodes, which provide information regarding the source and destination of traffic, and can be used to calculate demands between source/destination node pairs. The NTP server provides the software clock synchronisation for the nodes, as an accurate time source is a requirement to minimise network disruption during the dynamic topology mechanism's topology changes. Note that the Quagga routing suite, which provides OSPF and other routing functionality, is not required, as the controller's connection to node eight is used as a default gateway, and node eight advertises the controller's network via its OSPF configuration.

Network nodes and hosts

The software that provides the required functionality for the network nodes and hosts is described below, and elaborated further in the explanation of the dynamic topology mechanism. Note that a single Raspberry Pi is configured as a network node, but also runs the network host in an LXC container, which is similar to a virtual machine running the same operating system as its host machine. The network nodes' software is listed in table 4.2 below.

The OSPF routing functionality is provided by the quagga suite, and is used as a baseline for the dynamic topology mechanism's performance, as described in chapter 3. The cURL package is generally useful for command line transfer of files using FTP, but the tool is not used directly; the cURL libraries are the main requirement, as the dynamic topology mechanism requires them for the transfer of topology configuration files from the controller. Somewhat similarly, the Autoconf and Debootstrap packages are not used directly, but are required to install LXC and create the LXC container that will provide

Software name	Version	Description
Raspbian	May 2015	Linux operating system.
Quagga	0.99.22.4-1+wheezy1	BGP/OSPF/RIP routing daemon.
cURL	7.26.0-1+wheezy13	Command line tool for transferring data with URL syntax.
Autoconf	2.69-1	Automatic configure script builder.
LXC	1.1.0	Userspace container object for full resource isolation and resource control.
Debootstrap	1.0.48+deb7u2	Debian bootstrapper.
Open vSwitch	2.4.90	Multilayer software switch.
Wondershaper	1.1a-6	Traffic shaping script.

Table 4.2: Network node software

the host's functionality. The Wondershaper package is simple but vital for the emulation of Telstra network AS1221, as it applies bandwidth limitations to the inter-node links. As alluded to in the controller's software description, NTP synchronisation is required for the network nodes, but no additional packages are required as NTP is provided by default in the listed Raspbian distribution.

Open vSwitch is perhaps the most vital component in terms of supporting functionality for the dynamic topology mechanism, as it provides MPLS traffic processing functionality that can be explicitly programmed. This multilayer switch is programmed using flow tables, which specify criteria for matching a packet, and the actions to apply to the packet once it is matched. For normal operation using OSPF, the flow tables are configured to drop all packets to prevent the generation of duplicate packets in the network. For routing using MPLS, with or without the dynamic topology mechanism, the flow tables are configured as shown in figure 4.3. Note that this example is for node one. This flow table configuration has been designed to minimise the number of flow table entries that need to be modified during topology changes. For an n-node network, the n×n entries in table 0 are static, and only n-1 changes need to be made; for the flow tables shown in figure 4.3, only the entries in table 2-8 need to changed. The flow tables are described in more detail in section 4.4.

The hosts' software is quite simple, as the only role the host perform is traffic generation and network performance testing. The hosts' software configuration is shown in table 4.3 below.

To test the performance of the dynamic topology mechanism, the network traffic must be controlled. This control is attained through traffic generation using iPerf on the host

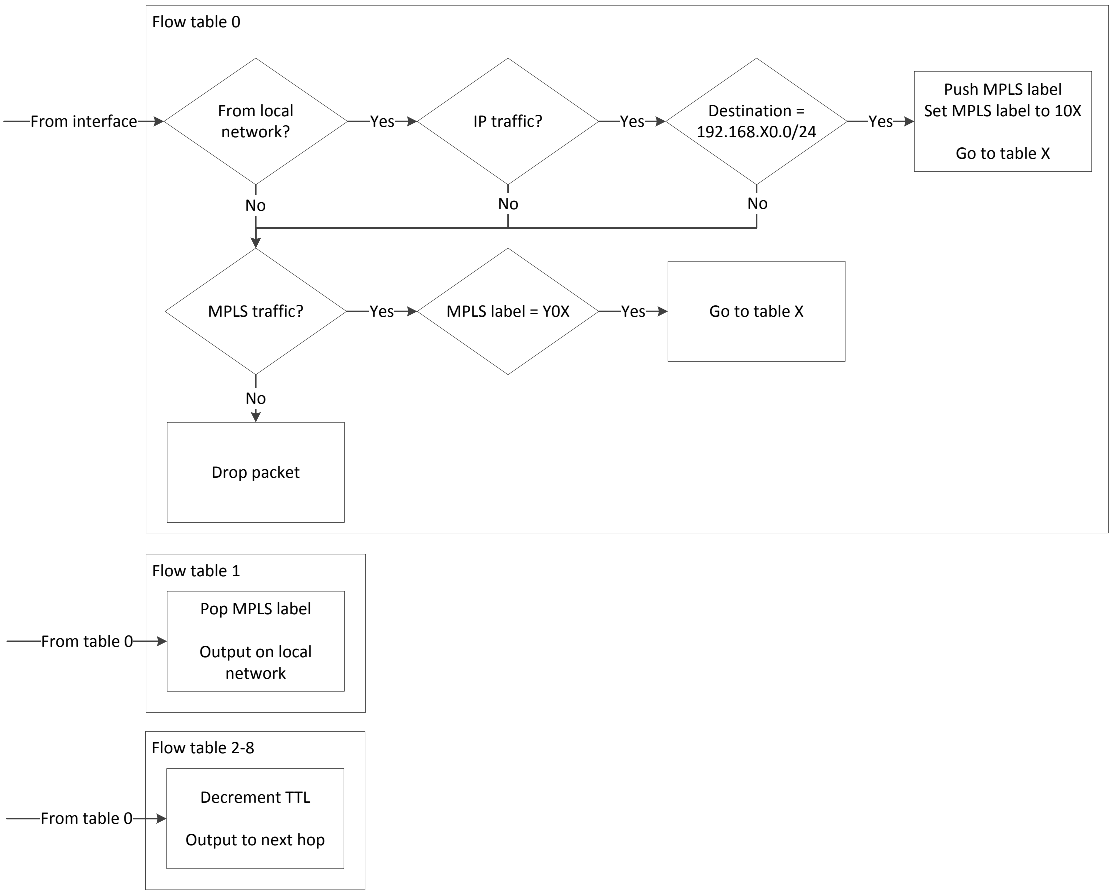

Figure 4.3: Flow table configuration example - Node one

connecteed to the node's access network. All of the metrics used to determine network performance, as detailed in chapter 3, with the exclusion of latency, are provided by iPerf by default. To determine the latency during the testing, the ping utility must be installed, which is surprisingly not installed by default. The hosts also require NTP synchronisation to control the traffic generation timing during testing, but this is controlled through the network node and automatically passed to the LXC container, so no additional software is required.

To provide insight into the purpose of the network node's and host's software, a diagrammatic representation of the relationship between some of the software is shown below in figure 4.4. The grey lines represent the processing of network traffic for the host connected to the node's access network. Communication between the host and the node is via virtual Ethernet ports, and the traffic is either processed using the routes generated by Quagga, or by the flows in Open vSwitch. Note that the packets are duplicated, so Open vSwitch must drop the packet to prevent the duplicates from reaching the network.

Software name	Version	Description
Raspbian	May 2015	Linux operating system.
iPerf	2.0.5+dfsg1-2	Traffic generation and measurement.
iputils-ping	3:20121221-5	Tools to test the reachability of network hosts.

Table 4.3: Network host software

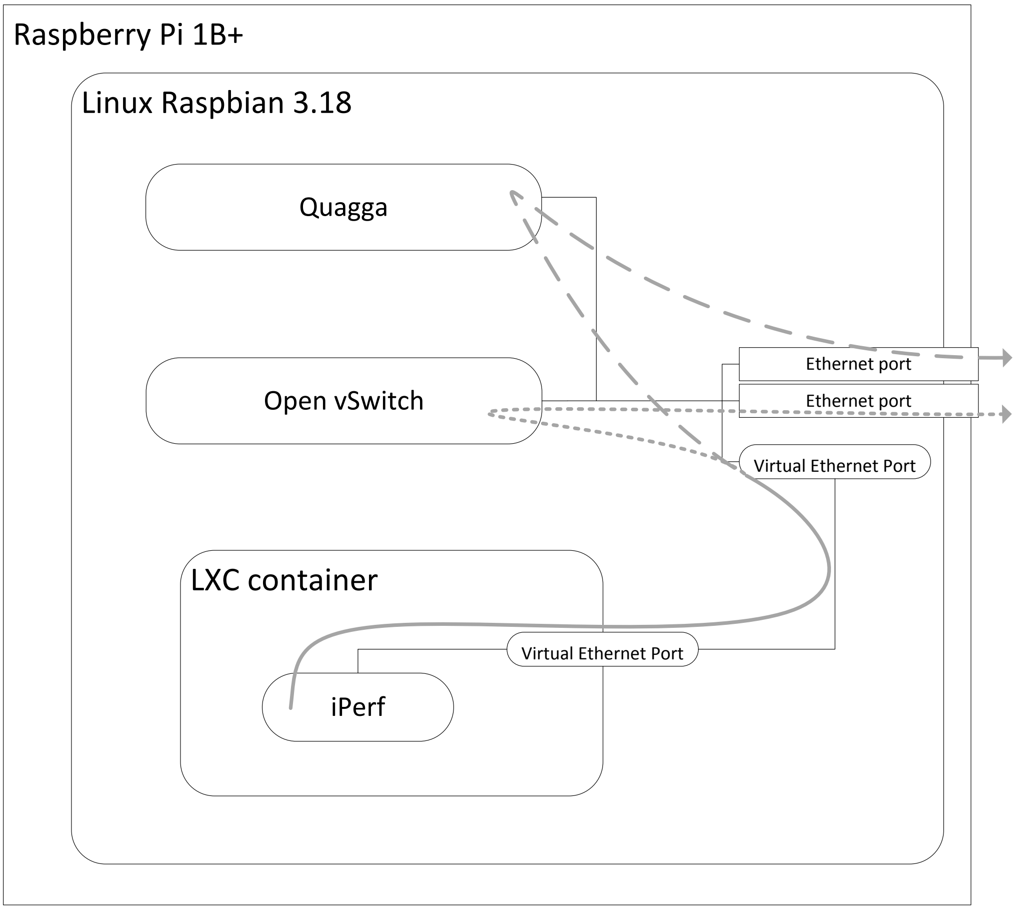

Figure 4.4: Network node software

4.2 Dynamic topology mechanism

The dynamic topology mechanism performs functions very similar to that proposed by Aldraho & Kist (2011*b*), but incorporates other work (Aldraho & Kist 2010, Aldraho et al. 2012, Polverini et al. 2015) in addition to original work. The operation of the dynamic topology mechanism relies on two programs: one on the controller, and one on each of the nodes. These programs, in addition to the programs written to test the system, are explained in detail in the following sections, and outlined below:

1. Controller: Monitor network traffic
2. Controller: Optimise network topology
    - Focus on energy consumption reduction through the use of router standby states
    - Ensure link utilisation does not rise above a threshold value for any of the links
    - Apply optimisation heuristics
3. Controller/Nodes: Communicate the topology change
4. Nodes: Implement the topology change
5. Repeat

4.3 Controller program operation

The controller's dynamic topology mechanism program sequentially and iteratively performs four main functions: network traffic monitoring, traffic matrix generation, topology optimisation, and topology communication. These are each explained in detail below, along with pseudocode that corresponds to the source code shown at appendix C.

4.3.1 Network traffic monitoring and traffic matrix generation

As the nodes in the network are logically fully meshed, with every node possessing a path to the seven other nodes, the traffic matrix can be directly constructed from the traffic data statistics for each of the 56 source/destination node pairs. The traffic data statistics are calculated by the controller program using a stream of sFlow messages. The stream is piped from sFlowTool, which collects the sFlow messages from each nodes' instance of Open vSwitch. The command used to do so is shown below:

```
daniel@controller:~$ sflowtool -l | sudo ./controller
```

The use of the `-l` switch formats the sFlow messages in the stream as comma-separated values, an example of which is shown below:

```
FLOW,192.168.80.1,9,4,9a3a524b5b22,fe960a04a4d4,0x0800,0,0,
192.168.80.100,192.168.10.1,17,0x00,64,37156,5001,0x00,1516,1498,32
```

The fields that are underlined in the above example are those used to construct the traffic matrix. These fields contain the agent IP address (i.e. reporting node's IP address), the source and destination IP addresses of the traffic, the traffic's packet size, and the sampling rate, respectively.

The controller program captures and analyses each sFlow message, and uses its data to determine the traffic demands of the network, which are used to directly construct the traffic matrix. The sFlow messages are captured for an interval of at least 15 seconds, which is measured to μs resolution for use in the traffic calculation. At the conclusion of the measurement interval, the traffic demands are calculated for each source/destination pair using the following formula:

$$\text{Traffic demand (Mbps)} = \frac{n \times \lambda \times \alpha \times 8}{1,000,000 \times \tau} \tag{4.1}$$

In equation 4.1 above:

- n = number of samples captured during interval τ
- λ = sampling rate (packets/sample)
- α = packet size (bytes)
- τ = measurement interval (seconds)

To improve the calculation's resilience to momentary fluctuations in network traffic, the traffic demand calculation uses data gathered over two measurement intervals, i.e. 30 seconds. Another measure, which aims to minimise the processing burden and to prevent duplicate processing of traffic, is to only capture sFlow messages if the agent and destination are on the same network, i.e. traffic is destined for the reporting node or its access network. The accuracy of the network traffic monitoring implementation is examined in chapter 5.

4.3.2 Topology optimisation

Once the traffic matrix has been constructed, the controller program can then attempt to optimise the topology with the aim of minimising the energy consumption of the network by maximising the number of nodes in the standby state. The topology optimisation algorithm is relatively simple and has been influenced by the work of Aldraho & Kist (2010) and Polverini et al. (2015). The general concept is to iteratively select a node to be placed into the standby state and calculate the predicted effect on the network links' utilisation. If the result of placing the node in standby forces any link utilisation above a threshold value, or if any node is isolated from the network, the node is restored and optimisation is reattempted until no options remain. To allow sufficient capacity to withstand increases in network traffic and inaccuracies in traffic matrix calculation, the threshold value of 70% link utilisation has been selected. Note that the optimisation topology only performs calculations, and does not impact the network until the topology selection has been finalised.

As previously stated, while the optimisation is focused on reducing energy consumption, the implementation of standby states, similar to those described by Polverini et al. (2015) and Aldraho et al. (2012), and the measurement of their effect on the network has been left as future work. In this implementation, the standby state is only implemented through the modification of traffic processing, which is detailed in the explanation of the network node program's operation.

The optimisation algorithm uses the traffic matrix and an implementation of Dijkstra's algorithm to determine the link utilisation for a topology with all nodes active; link weights are determined using equation 4.2 below:

$$\text{Link weight} = \frac{10}{\text{Link bandwidth (Mbps)}} \tag{4.2}$$

To provide data to the optimisation heuristics, the shortest paths are examined to determine the frequency of each node's use as a transit node. The least used node is selected as a candidate to be placed in standby; as the standby node must maintain only a single connection to the network, the link to the most used adjacent node is preserved and all others are logically removed from the standby node. The node's traffic load is used to resolve any conflicts, i.e if two nodes have identical and maximum or minimum usage in the network, and is calculated as the node's total connected link traffic load. The nodes' adjacencies are temporarily updated and the link utilisations are recalculated using the traffic matrix and the modified topology; the topology is also examined to ensure none of the nodes have been isolated. As previously alluded to, the topology's failure criteria are the utilisation of any of the links rising above the threshold value of 70%, or the isolation of any of the nodes from the network. If either of the failure criteria are met, the latest topology modifications are reversed and the node that was in standby is omitted from the set of standby candidates. The optimisation algorithm executes until it has attempted to place each node into standby, and the resultant topology is written to a local file to prepare it for communication and implementation.

Note that the dynamic topology mechanism can be disabled by providing any number of command line arguments. Disabling the dynamic topoogy mechanism simply produces a topology configuration file that has all nodes active and uses shortest path routing. This functionality was included to enable measurement of the second baseline described in chapter 3: all nodes in an active state, traffic routed using MPLS with a static topology.

4.3.3 Topology communication

Once the optimised topology has been written to a local file, it is compared to the current topology; if there are differences between the two topologies, the oldest is overwritten, and the topology change is communicated to the nodes. The topology change is not directly communicated to the nodes, and relies on the nodes monitoring the modification time of the controller's topology configuration file, as described in the following section. The topology configuration file contains the next hop node for each source/destination node pair, and is comprised of eight rows of eight comma-separated values; the rows signify source nodes and the fields in the rows signify destination nodes. An example of the topology configuration file's contents is shown in listing 4.1 below.

Listing 4.1: Topology configuration file example

```
1,7,7,7,7,7,7,7,
7,2,3,3,3,7,7,3,
8,2,3,8,5,8,8,8,
8,8,8,4,8,8,8,8,
3,3,3,3,5,3,3,3,
7,7,7,7,7,6,7,7,
1,2,8,8,8,6,7,8,
7,3,3,4,3,7,7,8,
```

In the above, you can see that nodes one, four, five, and six have been placed in standby for this topology, as the next hop for all destinations is the same for a single source. Note that if the source and destination are equal, i.e. along the diagonal from top-left to bottom-right, the next hop is the current node, and the value is ignored in the nodes' program.

After the optimised topology has been written to a local file, it is compared line-by-line with the topology file currently being used. If any line is different, the old file is overwritten, which has the appearance of updating the file contents and modification time from the nodes' perspective, which prompts each of them to pull the new topology configuration file. The network node program operation description covers the use of the file modification times and data in further detail, but it is evident that the nodes and the controller's software clocks must be synchronised, as previously described in the software configuration.

4.3.4 Pseudocode

The source code for the controller's dynamic topology mechanism program is shown in section C.1, and is shown below as pseudocode for the purpose of clarity. There are two components to the controller's program; the main program in listing C.1, and the optimisation algorithm in listing C.3.

Main program

```
1. Define measurement interval and topology configuration filename, and declare variables
2. Initialise adjacency matrix with full topology
3. Capture measurement start time
4. Enter infinite loop
    (a) Clear sFlow data from last captured message
    (b) Get sFlow message from stream input
    (c) Process sFlow message
        i. Dump unwanted sFlow message fields
        ii. Get the node number of the message agent
            A. Continue to next sFlow message capture if the node number is not one of the expected values.
        iii. Dump unwanted sFlow message fields
        iv. Get the node number of the traffic's source
            A. Continue to next sFlow message capture if the node number is not one of the expected values.
        v. Dump unwanted sFlow message fields
```

vi. Get the node number of the traffic's destination

A. Continue to next sFlow message capture if the node number is not one of the expected values

vii. Dump unwanted sFlow message fields

viii. Get the traffic's packet size

ix. Dump unwanted sFlow message fields

x. Get the sFlow sampling rate

(d) If the agent and destination nodes are equal, multiply the packet size by the sample rate and add it to the current traffic count for the corresponding source/destination pair.

(e) Measure the time difference between now and the measurement start time

(f) If the time difference is greater than the measurement interval, construct the traffic matrix and optimise the topology

i. Capture a new measurement start time

ii. Use the current and previous traffic counts, the current and previous time differences, and equation 4.1 to calculate the traffic demand for every source/destination node pair

iii. Store the current traffic count and time difference as the previous traffic count and time difference

iv. Display the traffic matrix at standard output

v. Create a new, empty topology configuration file

vi. Run the topology optimisation algorithm and write the output to the new file

vii. Compare the new topology with the current topology

viii. If the files are identical, delete the new file, otherwise, overwrite the old file with the new file

(g) Return to the start of the loop

Optimisation algorithm

1. Store the traffic matrix, adjacency list, output filename, and dynamic topology mechanism deactivation switch state (all passed by value from calling function)

2. Determine number of nodes in the network, initialise link utilisation threshold, and declare variables

3. Set all nodes to be active

4. Calculate shortest path for each source/destination node pair using Dijkstra's algorithm

5. Use the traffic matrix and the shortest paths to determine the traffic load on each link

6. Calculate the traffic loading on each node as the sum of the traffic load on all connected links

7. Use the link traffic load and link bandwidths to calculate the link utilisation percentage

8. Find maximum link utilisation percentage

9. If the dynamic topology mechanism is not disabled, loop while the maximum link utilisation is less than the threshold value

   (a) Break the loop if attempts have been made to place each node in standby

   (b) Clear the previously calculated node and link traffic loads

   (c) Set the variable that tracks the reachability of the nodes to false (guarantees at least one execution of the loop below)

   (d) Loop while the link utilisation is above the threshold or one or more nodes are unreachable

      i. Use the shortest paths to determine the frequency of each node's use as a transit node

      ii. From the set of nodes that are candidates for standby, find the node that is used as a transit node the least. Use the nodes' traffic load to resolve any conflicts

      iii. Break the loop if a least used node cannot be determined

      iv. Remove the node from the set of nodes that are candidates for standby

      v. Store the node's adjacency information. This is only used if placing the node in standby has to be reversed

        vi. From the set of nodes adjacent to the node to be placed in standby, find the node that is used as a transit node the most. Use the nodes' traffic load to resolve any conflicts
        vii. Update the adjacency matrix of the node to be placed in standby and all its adjacent nodes
        viii. Recalculate shortest paths for new topology
        ix. Recalculate the traffic load on each link
        x. Recalculate the link utilisation percentage
        xi. Find maximum link utilisation percentage
        xii. Test if all nodes are reachable from every node
        xiii. If the link utilisation is above the threshold or one or more nodes are unreachable, reverse the actions taken to modify the adjacency matrix and return to the start of the loop
    (e) Recalculate the traffic load on each link
    (f) Recalculate the link utilisation percentage
    (g) Find maximum link utilisation percentage
    (h) Return to the start of the loop if the maximum link utilisation is less than the threshold value

10. Write the next hop in the shortest path for each source/destination node pair to the specified output file

4.4 Network node program operation

The network nodes' dynamic topology mechanism program initialises the Open vSwitch configuration that provides MPLS functionality, ensures the host connected to the access network is running, and iteratively checks for topology updates and implements the topology change via Open vSwitch flow table entry modifications if required. The components of the network nodes' program are explained in detail below, along with pseudocode that corresponds to the source code shown at appendix C.

4.4.1 System initialisation

The system initialisation is extremely important for the operation of the dynamic topology mechanism, as it sets the foundation for the swift, synchronised reconfiguration of the network. The same program runs on all eight nodes, but with actions varying slightly for each node, so the program must first determine which node it is running on. It does this by examining the node's hostname, which is in a standard format of "`nodeX`" where "`X`" is a digit between one and eight. Another key element is the synchronisation among the nodes and with the controller; the synchronisation is performed using NTP, with the controller's NTP server being used as the reference for all nodes.

The final and arguably the most important action during node initialisation is the configuration of Open vSwtich. Once it has been confirmed that Open vSwtich and the LXC container are running, or started if necessary, the MAC addresses of the LXC container's interface and the node's interface to the LXC container are retrieved. These MAC addresses are important as they allow the flow table entries to be configured to process traffic destined for the node's access network, i.e. destined for the access network's host running in the LXC container. Using these MAC addresses and the node number, the Open vSwitch flows can be setup as previously described in section 4.1.2 and shown in figure 4.3. The flow table entries can be divided into five different categories: transit, terminating, and originating traffic processing flows, output handling flow tables, and firewall flows.

Originating traffic processing flows

As the nodes perform MPLS ingress and egress actions, they act as the barrier between the IP network of their access network and the MPLS network between the nodes. The traffic coming from the node's access network will be IP traffic with a destination of one of the other nodes' access networks. The node must push an MPLS label onto the packet, set the label as per the network's convention, and send the packet to the relevant output handling flow table. The label numbering convention used in the network is quite simple, and is based on the source and destination node numbers. For example, a packet with the label "306" has originated from node three and is destined for node six. An example of a set of originating traffic processing flows for node four are shown in listing 4.2 below.

Listing 4.2: Originating traffic processing flows example (node four)

```
table=0,priority=1000,in_port=1,dl_type=0x0800,nw_dst
    =192.168.10.0/24,actions=push_mpls:0x8847,set_mpls_label:401,
    goto_table:1
table=0,priority=1000,in_port=1,dl_type=0x0800,nw_dst
    =192.168.20.0/24,actions=push_mpls:0x8847,set_mpls_label:402,
    goto_table:2
table=0,priority=1000,in_port=1,dl_type=0x0800,nw_dst
    =192.168.30.0/24,actions=push_mpls:0x8847,set_mpls_label:403,
    goto_table:3
table=0,priority=1000,in_port=1,dl_type=0x0800,nw_dst
    =192.168.50.0/24,actions=push_mpls:0x8847,set_mpls_label:405,
    goto_table:5
table=0,priority=1000,in_port=1,dl_type=0x0800,nw_dst
    =192.168.60.0/24,actions=push_mpls:0x8847,set_mpls_label:406,
    goto_table:6
table=0,priority=1000,in_port=1,dl_type=0x0800,nw_dst
    =192.168.70.0/24,actions=push_mpls:0x8847,set_mpls_label:407,
    goto_table:7
table=0,priority=1000,in_port=1,dl_type=0x0800,nw_dst
    =192.168.80.0/24,actions=push_mpls:0x8847,set_mpls_label:408,
    goto_table:8
```

In the above:

- `table=0` is the first table that is examined for an entry that matches the packet
- `in_port=1` is the Open vSwitch port that corresponds to the node's virtual Ethernet port connected to the LXC container
- `dl_type=0x0800` matches to IP packets
- `nw_dst=192.168.10.0/24` matches to a specific destination network
- `push_mpls:0x8847` changes the protocol form IP to MPLS
- `set_mpls_label:401` sets the packet's MPLS label value
- `goto_table:1` resubmits the packet for processing by another flow table, which in this implementation is one of the output handling flow tables

Transit and terminating traffic processing flows

The transit and terminating traffic processing flows are identical for every node, as they simply provide a match for every possible MPLS label in the system, and define the output handling flow table to be used. A snippet of the 56 flow table entries that fall into this category are shown in listing 4.3 below.

Listing 4.3: Transit and terminating traffic processing flows snippet

```
...

table=0,priority=500,dl_type=0x8847,mpls_label=401,actions=
    goto_table:1
table=0,priority=500,dl_type=0x8847,mpls_label=402,actions=
    goto_table:2
table=0,priority=500,dl_type=0x8847,mpls_label=403,actions=
    goto_table:3
table=0,priority=500,dl_type=0x8847,mpls_label=405,actions=
    goto_table:5
table=0,priority=500,dl_type=0x8847,mpls_label=406,actions=
    goto_table:6
table=0,priority=500,dl_type=0x8847,mpls_label=407,actions=
    goto_table:7
table=0,priority=500,dl_type=0x8847,mpls_label=408,actions=
    goto_table:8

table=0,priority=500,dl_type=0x8847,mpls_label=501,actions=
    goto_table:1
table=0,priority=500,dl_type=0x8847,mpls_label=502,actions=
    goto_table:2
table=0,priority=500,dl_type=0x8847,mpls_label=503,actions=
    goto_table:3
table=0,priority=500,dl_type=0x8847,mpls_label=504,actions=
    goto_table:4
table=0,priority=500,dl_type=0x8847,mpls_label=506,actions=
    goto_table:6
table=0,priority=500,dl_type=0x8847,mpls_label=507,actions=
    goto_table:7
table=0,priority=500,dl_type=0x8847,mpls_label=508,actions=
    goto_table:8

...
```

In the above:

- `table=0` is the first table that is examined for an entry that matches the packet
- `dl_type=0x8847` matches to MPLS packets
- `mpls_label:401` matches to a specific MPLS label value
- `goto_table:1` resubmits the packet for processing by another flow table, which in this implementation is one of the output handling flow tables

Output handling flow tables

The output handling flow tables are the only ones that are modified when the topology change is implemented. The network traffic does not directly match with the output handling flow tables, and must first be processed by one of the flow table zero entries.

Each table only contains one entry, and apart from the flow table that handles traffic destined for the current node, they only perform two actions. These are described in further detail in section 4.4.3, and the initial entry only performs normal processing, which results in the packets being dropped as the system cannot process MPLS packets. The flow table that handles traffic destined for the current node is more detailed; it pops the MPLS label, which converts the packet back to IP, modifies the source and destination MAC addresses to be the node's interface to the LXC container and the LXC container's interface address respectively, and outputs the packet on the access network interface. An example of the initial set of output handling flow tables for node four are shown in listing 4.4 below.

Listing 4.4: Initialised output handling flow tables example (node four)

```
table=1, actions=normal
table=2, actions=normal
table=3, actions=normal
table=4, actions=pop_mpls:0x0800,mod_dl_src:11:22:33:44:55:66
   mod_dl_dst:66:55:44:33:22:11, 1
table=5, actions=normal
table=6, actions=normal
table=7, actions=normal
table=8, actions=normal
```

In the above:

- `table=1` specifies the table number, which corresponds to the destination node
- As there are no match criteria, all packets sent to the table are matched
- `actions=normal` sends the packet to the OS for normal processing. As the packet must be an MPLS packet to reach this point, and the OS cannot process MPLS packets, this has the same effect as dropping the packet
- `pop_mpls=0x0800` removes the MPLS packet and changes the protocol to IP
- `mod_dl_src:11:22:33:44:55:66` changes the source MAC address to that specified. In this implementation, the address of the node's interface to the access network, i.e. the virtual interface to the LXC container, is used
- `mod_dl_dst:66:55:44:33:22:11` changes the destination MAC address to that specified. In this implementation, the address of the host connected to the node's access network, i.e. the LXC container's interface to the node, is used
- `1` specifies the output Open vSwitch port. In this implementation, port 1 is the connection to the access network

Firewall flows

As previously described in section 4.1.2, any remaining packets, i.e. packets that can be processed by Quagga, must be dropped. Any actions other than dropping the packet will almost certainly result in duplicate packets in the network, as the packet will be processed by both Open vSwtich and Quagga. The single flow table entry that acts as a firewall on all nodes to prevent the duplicate packets from entering the network is shown in listing 4.5 below.

Listing 4.5: Firewall flow table entry

```
table=0, priority=5, actions=drop
```

In the above:

- `table=0` is the first table that is examined for an entry that matches the packet
- `priority=5` places the entry at a higher priority to the default "catch-all" entry that forwards the packet for normal processing
- As there are no match criteria, all packets that have not matched another entry are matched
- `actions=drop` discards the packet and prevents any further processing

4.4.2 Topology change monitoring and communication

As previously described in section 4.3, the controller does not send topology updates to the nodes, the nodes must monitor the controller for topology changes. The controller overwrites its topology configuration file and updates the file modification time whenever there is a topology change; for the nodes to be aware of topology changes, they must repeatedly pull the modification time of controller's topology configuration file and compare it to the modification time of their local copy of the file. This implementation uses cURL to retrieve the file header, which contains the modification time. If the controller's copy is newer than the local copy, the node retrieves the updated file and implements the topology change, as described below. Otherwise, the node sleeps for five seconds before checking the modification times again.

If a topology change has occurred, the node must pull the updated configuration file from the controller. This implementation uses cURL to get the file; if cURL is unsuccessful, the node sleeps for 2.5 seconds before trying again. Once the file has been successfully retrieved from the controller, the modification time is updated to ensure it matches that of the controller's copy, and the node proceeds to implement the topology change.

4.4.3 Topology change implementation

Once the node's topology configuration file has been updated, the topology change can be implemented. This must be synchronised across the network to minimise disruptions to the network traffic. As previously discussed, the flow tables have been constructed to minimise the number of changes that need to be made to implement topology changes. The output processing flow tables are modified to align with the next hops specified in the topology configuration file. The impact of topology changes on network performance is discussed in chapter 5.

The use of timers for topology changeover, as discussed by Aldraho et al. (2012), has been replaced by a changeover at the flow level on each node, synchronised using NTP. The topology changeover is synchronised to the closest five second interval to swiftly react to topology changes while still allowing sufficient time for all nodes to synchronise. The accuracy of NTP synchronisation is heavily dependant on the number of hops between the server and the client (Machizawa & Iwama 2013). The absolute worst case scenario in this project's network is a distance of eight hops, which results in a synchronisation accuracy of approximately $\pm 250\mu$s.

Once time has advanced to the topology changeover time, the topology change is implemented. This is achieved by modification of the output processing flow tables, and requires information from the topology configuration file and the node's interfaces. The topology configuration file states the next hop for each destination for all source and destination node pairs. The node must retrieve the relevant data from the file, and then convert the next hops into output ports. Due to the addressing scheme used in this implementation, the interface's IP addresses can be examined to determine the neighbouring node connected to each interface. The interfaces must finally be converted to Open vSwitch port numbers before they can be used to update the flow tables; due to the port numbering scheme used in this implementation, shown below in table 4.4, this is a simple task. The output processing flow tables can then be updated to correspond with the new topology;

Interface	Port number
veth0	1
eth0	2
eth1	3
eth2	4
eth3	5

Table 4.4: Relationship between node interfaces and Open vSwitch port numbers

and example is shown below in listing 4.6

Listing 4.6: Output handling flow tables after topology change example (node four)

```
table=1, actions=dec_mpls_ttl,3
table=2, actions=dec_mpls_ttl,3
table=3, actions=dec_mpls_ttl,3
table=4, actions=pop_mpls:0x0800,mod_dl_src:11:22:33:44:55:66
   mod_dl_dst:66:55:44:33:22:11, 1
table=5, actions=dec_mpls_ttl,3
table=6, actions=dec_mpls_ttl,3
table=7, actions=dec_mpls_ttl,3
table=8, actions=dec_mpls_ttl,3
```

In the above:

- `table=1` specifies the table number, which corresponds to the destination node
- As there are no match criteria, all packets sent to the table are matched
- `dec_mpls_ttl` decrements the MPLS packets time to live (TTL) counter
- `1` specifies the output Open vSwitch port
- `pop_mpls=0x0800` removes the MPLS packet and changes the protocol to IP
- `mod_dl_src:11:22:33:44:55:66` changes the source MAC address to that specified. In this implementation, the address of the node's interface to the access network, i.e. the virtual interface to the LXC container, is used
- `mod_dl_dst:66:55:44:33:22:11` changes the destination MAC address to that specified. In this implementation, the address of the host connected to the node's access network, i.e. the LXC container's interface to the node, is used

4.4.4 Pseudocode

The source code for the node's dynamic topology mechanism program is shown in section C.2, and is shown below as pseudocode for the purpose of clarity. There are three components to the node's program; the main program in listing C.4, the initialisation and topology implementation functions in listings C.5 and C.6, and the cURL functions in listings C.7 and C.8.

Main program

1. Define interrupt handler that removes Open vSwitch configuration and deletes the local topology configuration file
2. Define the time to wait between modification time checks, the controller's URL to pass to cURL, and the configuration filename and declare variables
3. Determine the current node by examining the last character of the hostname
4. Force the software clock to synchronise with the controller's NTP server
5. Start and initialise the LXC container and Opn vSwitch
6. Enter infinite loop
    - (a) Use cURL to get the modification time of the controller's topology configuration file
    - (b) Test if the remote file's modification time is later than the local file's modification time
    - (c) If the remote file is newer:
        - i. Attempt to get the topology configuration file from the controller
        - ii. If unsuccessful, wait for half the defined period of time and return to the start of the loop
        - iii. If successful, update the modification time of the local topology configuration file and implement the topology change
    - (d) If the remote file is not newer:
        - i. Wait for the defined period of time
    - (e) Return to the start of the loop

Initialisation

1. Store the current node number (passed by value from calling function)
2. Check for presence of PID files to determine if Open vSwitch is running
3. If Open vSwitch is not running, start it
4. Check if the LXC container is running
5. If the LXC container is not running, start it
6. Get the MAC address of the LXC container's interface using the ARP cache
7. Get the MAC address of the node's interface that connects to the LXC container using the interface's address file
8. Add the default output handling flow tables to Open vSwitch
9. Add the transit and terminating traffic processing flows to flow table zero of Open vSwitch
10. Modify the output handling flow table for terminating traffic using the node current number and LXC container and node MAC addresses
11. Add the originating traffic flows to flow table zero of Open vSwitch using the current node number
12. Add the firewall flow to flow table zero of Open vSwitch

Topology change implementation

1. Store the topology configuration filename and current node number (passed by value from calling function)
2. Declare variables
3. Get current time
4. Determine topology change time
5. Retrieve the next hop node number for each destination node from the configuration file

6. Use the interfaces' IP addresses to determine which node each interface connects to
7. Convert the next hop node number from the configuration file into an interface number for each destination node
8. Convert the interface number into an Open vSwitch port number for each destination node
9. Wait until the local time equals the topology change time
10. Modify the output processing flow tables using the port numbers for each destination node

4.5 System testing programs

In addition to the dynamic topology mechanism's normal operation, there are two small programs that have been created to meet the testing requirements specified in chapter 3. One program runs inside the LXC container and is used to generate the traffic for the tests and collect the testing data, while the other program simply analyses the testing data. The source code for these two programs are shown in listings C.9 and C.10 respectively.

4.5.1 Host traffic generation

The host does not directly control the operation of the dynamic topology mechanism, but instead controls the generation of traffic to simulate the access network, which in turn results in topology changes when the dynamic topology mechanism is used. Traffic generation has been automated to allow synchronisation with other nodes, facilitate the collection of data for all eight nodes, and to allow replication of tests with varying configurations, as described in chapter 3. The traffic generation program runs on the host connected to the node, i.e. it runs in the LXC container, and behaves differently depending on which node it is connected to.

The traffic generation is controlled by three files, with the demands stated in each of the files cumulatively applied to the network with changes made incrementally and spaced ten minutes apart: the first matrix is applied for 30 minutes, the second matrix is applied for 20 minutes after a delay of 10 minutes, and the third matrix is applied for 10 minutes after a delay of 20 minutes. This results in the application of each of the three matrices specified in chapter 3 for ten minutes.

The synchronisation of the hosts uses the same method as the nodes: the software clock is synchronised with the controller's clock, the current time is captured, and the test is started at the closest 30 second interval. However, the LXC container does not need to synchronise its software clock, as it inherits its timing from the node, which has been synchronised with the controller.

As there are three traffic generation matrices, which are simultaneously applied to the network for the third and final testing interval, three instances of iPerf servers are required to receive traffic generated by the seven other hosts. For traffic generation, each host runs seven instances of iPerf clients, which are connected to servers running on the seven other hosts. To facilitate processing of the testing data, the iPerf clients output data as comma-separated values and redirect their output to files with the following naming convention: `(source node)-(destination node)_(traffic generation matrix).iperf`. For example, the iPerf statistics from the traffic between nodes one and six for the second traffic generation matrix would be named `1-6_2.iperf`. While iPerf statistics provide the majority of the required network performance metrics, ping is used to determine the delay. For each interval on each node, the seven other hosts are pinged, and the output is redirected to files with a naming convention almost identical to that of the iPerf files: `(source node)-(destination node)_(testing interval).ping`. For example, the ping statistics between nodes one and six for the second testing interval would be named `1-6_2.ping`.

At the conclusion of the test, each host uses cURL to transfer their 42 testing data files to the controller. Once every node has transferred their files, the controller has 336 files to process; the files are processed using the program described in the following section.

4.5.2 Network performance measurement

As a result of the system test, each host generates 42 files that contain the network performance data for the full suite of tests, resulting in a total of 336 files that must be processed to determine the system's performance. The files are collated on the controller, and the processing of the files is relatively easy to program due to the standardised nature of the filenames and the data they contain.

As stated above, the naming convention for the files containing the output of the iPerf and ping commands is standardised, and is based on the source node, destination node, testing interval, and program used to produce the data. This makes it very easy to pair each file with a specific scenario and extract its data to be combined into the system report.

When the traffic is generated using iPerf, the switch `-y c` is used to output the reports as comma-separated values; this makes the extraction of the relevant data fields form the file very easy. Similarly, the ping tests have a standard format from which the delay statistics are easily extracted. Examples of these files generated by iPerf and ping are shown in listings 4.7 and 4.8 respectively.

Listing 4.7: iPerf output file example (4-2_3.iperf)

```
20151018094601,192.168.40.100,50021,192.168.20.100,50021,3,0.0-600.5
     ,3752910,49999
20151018094601,192.168.20.100,50021,192.168.40.100,50021,3,0.0-600.5
     ,3752910,50000,1.116,0,2553,0.000,0
```

In listing 4.7 above, the two lines are the statistics reported by the client and the server. The latter is the entry of interest, as it contains the data we wish to extract; the extra fields it reports are the connection's jitter, lost packets, total packets, lost packet percentage, and out of order packets.

Listing 4.8: Ping output file example (4-2_3.ping)

```
PING 192.168.20.100 (192.168.20.100) 56(84) bytes of data.

--- 192.168.20.100 ping statistics ---
600 packets transmitted, 600 received, 0% packet loss, time 599760
   ms
rtt min/avg/max/mdev = 1.613/2.061/3.962/0.238 ms
```

Once the data has been extracted from the files, it is collated into a system report, which shows the system's delay, jitter, packet loss, and packets received out of order for every source/destination node pair across the three test traffic matrices.

4.5.3 Pseudocode

The source code for the host traffic generation and network performance measurement programs are shown in section C.3, and are shown below as pseudocode for the purpose of clarity.

Host traffic generation

Note that this program runs on the host, i.e. the LXC container,

1. Define the traffic generation matrix filenames to be used and declare variables
2. Use the last character of the hostname to determine the node the host is connected to
3. Extract the traffic generation data relevant for this host from the traffic generation matrix files for all three testing intervals
4. Determine the local time and test start time to synchronise with other nodes
5. Wait until the start time
6. Prepare for traffic reception from other hosts
7. For each of the three testing intervals
   (a) Start traffic generation to all other host using the traffic generation matrix that corresponds to this testing interval. The traffic generation is set to run for an integer multiple of the testing interval; the first matrix is run for three intervals, the second matrix is run for two intervals, and the third matrix is run from one interval.
   (b) Start the ping to all other hosts. The ping is set to run for one testing interval
   (c) Wait for one testing interval, then progress to the next testing interval. If all tests have been performed, end

Network performance measurement

1. Define the filed numbers of the iPerf data of interest and declare variables

2. For every iPerf statistic file, dump unwanted data and extract the jitter, loss and out of order packet statistics

3. For every ping statistic file, dump unwanted data and extract the average round trip time

4. Print the collated data to standard output

Chapter 5

Results

This chapter aims to determine the effectiveness of the dynamic topology mechanism specified in chapter 4 by presenting an analysis of network performance measurements and a discussion of their implications. The measurements have been provided as appendix D, and were taken when the system specified in chapter 4 was tested using the procedure described in chapter 3 through the use of the testing programs detailed in chapter 4.

5.1 Analysis

The network performance measurements were captured for three configurations: traffic routed using OSPF with all nodes active in a static topology, traffic routed using MPLS with all nodes active in a static topology, and traffic routed using MPLS with a topology controlled by the dynamic topology mechanism. For the remainder of this chapter, these three configurations will be referred to simply as “OSPF”, “MPLS”, and “DTM” respectively.

In the following sections, the effect of the dynamic topology mechanism on network performance will be analysed for each of the metrics that have been measured: delay, jitter, packet loss, and number out of order packets. The OSPF and MPLS configurations have been selected as the point of reference, and the measurements taken from these configurations form a baseline against which the DTM measurements can be compared.

Each configuration’s set of measurements is comprised of three distinct scenarios. For the measurements taken using iPerf — jitter, packet loss, and out of order packets — the scenarios correspond to the three instances of iPerf used to generate the traffic, as explained in chapter 3. To allow the effects of the topology changeover to be captured, the three scenarios cover varying durations and network traffic levels, which makes it difficult to analyse a single configuration's data in isolation. It is therefore preferable to limit the analysis of the measurements to variations between configurations. This is not the case for the delay, as the scenarios correspond to each of the three intervals where traffic generation is constant, which allows the delay to be compared between scenarios in addition to the comparison between configurations.

One of the dynamic topology mechanism's requirements is the measurement of demands between every source/destination node pair, as described in chapter 4. To provide an indication of the effectiveness with which the system can respond to changes in network demands, the accuracy of the demand measurements analysed below.

5.1.1 Delay

Figure 5.1 shows the delay for each of the three configurations' three scenarios. The measurements for the OSPF and MPLS baseline configurations are almost identical between scenarios, and range from 1.17 ms to 2.9 ms with an average of 1.7 ms. This is to be expected for the small, eight node network, and provides a point of reference for the DTM measurements. The full suite of minimum, average, and maximum values is shown below in table 5.1.

Overall, the DTM's measurements are approximately 8 ms greater than the baseline. Scenario one's measurements are only slightly higher than the baseline, which could be attributed to greater path lengths due to five nodes being in the standby state. It is a reasonable assumption that the DTM's delay measurements would decrease with increasing network traffic; the path between nodes becomes less circuitous due to the dynamic topology mechanism decreasing the number of nodes in the standby state. However, as can be seen in table 5.1 and figure 5.1, the inverse is true. The reason behind the increased delay merits further investigation, but could potentially be a decrease in the performance of the ping used to measure the delay rather than of the network itself. The ping used to measure the delay sends ICMP packets, which are commonly classed as low priority and are typically the first to be affected by increases in network load. The increasing range of

		Delay (ms)		
		Minimum	**Average**	**Maximum**
OSPF	**Scenario 1**	1.169	1.677	2.727
	Scenario 2	1.200	1.727	2.800
	Scenario 3	1.200	1.795	2.900
MPLS	**Scenario 1**	1.167	1.668	2.718
	Scenario 2	1.200	1.718	2.800
	Scenario 3	1.200	1.779	2.900
DTM	**Scenario 1**	2.869	6.952	12.901
	Scenario 2	6.100	9.266	13.00
	Scenario 3	7.200	13.132	22.000

Table 5.1: Delay measurement summary

the values with increasing traffic, most notably between DTM scenarios two and three, may support this reasoning, as the greater variability could indicate the uncertain nature of the traffic's effect on the delivery of ICMP packets.

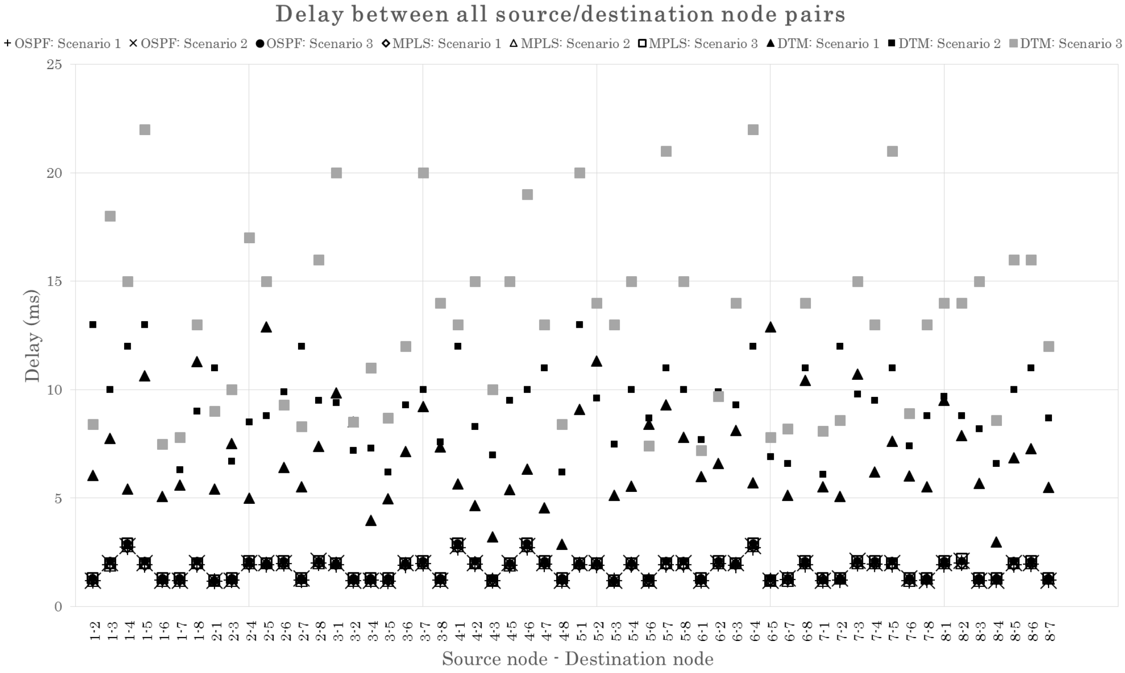

Figure 5.1: Network performance measurements — Delay

5.1.2 Jitter

The system's jitter measurements are shown in figure 5.2 for all configurations and scenarios. The baseline measurements are mostly random and centred around ~0.5 ms with a few outliers extending up to 2.1 ms. The same is true for the DTM measurements, with less frequent but slightly larger outliers, reaching as far as 3.1 ms. The full suite of minimum, average, and maximum values are provided in table 5.2 below. The baseline and DTM are quite comparable, implying that the dynamic topology mechanism has little to no impact on the system's jitter, with any variation in measurements safely attributable to the random nature of the system.

		Jitter (ms)		
		Minimum	**Average**	**Maximum**
OSPF	**Scenario 1**	0.079	0.409	1.300
	Scenario 2	0.091	0.502	1.100
	Scenario 3	0.230	0.713	1.900
MPLS	**Scenario 1**	0.100	0.394	1.200
	Scenario 2	0.120	0.422	1.400
	Scenario 3	0.210	0.700	2.100
DTM	**Scenario 1**	0.170	0.457	3.100
	Scenario 2	0.140	0.453	1.200
	Scenario 3	0.130	0.584	2.200

Table 5.2: Jitter measurement summary

5.1.3 Packet loss

The percentage packet loss for all configurations and scenarios is shown below in figure 5.3. Similar to the jitter shown above, the packet loss is comparable across all configurations; the majority of the measurements are zero, with a few randomly distributed outliers. Note that due to the measurement method, a direct comparison cannot be made between scenarios. Scenario three is the shortest duration and lowest traffic, scenario two is twice as long and has roughly double the traffic, and scenario one is three times as long and has five times the traffic; scenario three needs to lose fewer packets to experience the same percentage loss as the other scenarios. The number of non-zero packet loss instances and

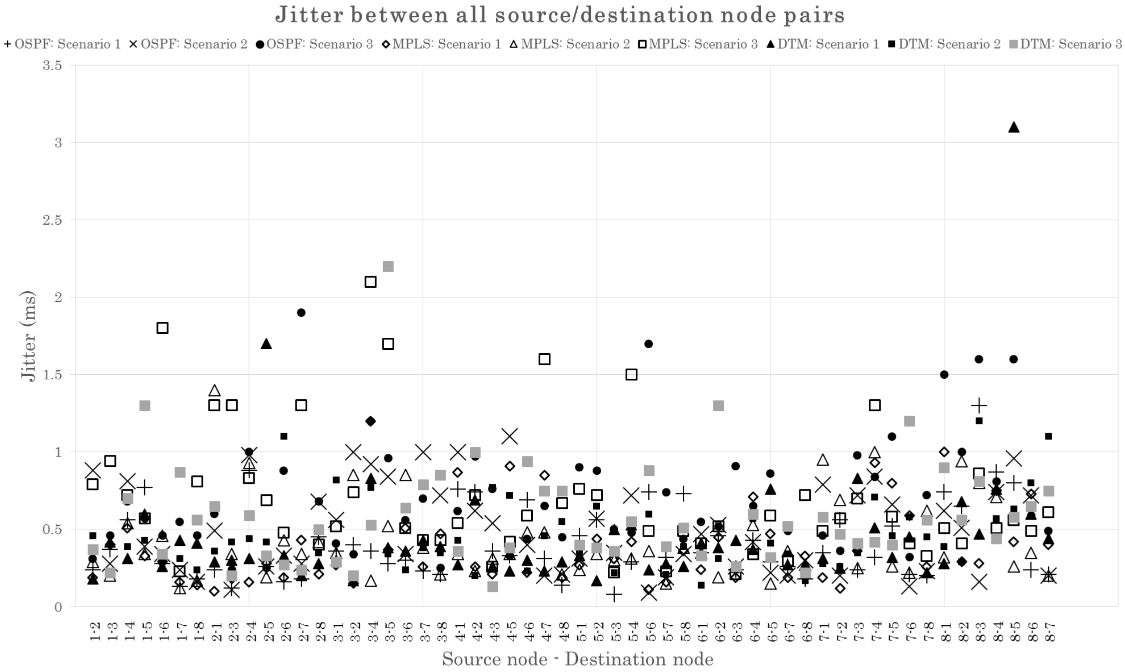

Figure 5.2: Network performance measurements — Jitter

the total packet loss percentages for each scenario are shown in table 5.3 below. One interpretation of the results is that the dynamic topology mechanism actually reduces the system's jitter, as the DTM measurements for each scenario are comparable to the minimum values for the entire baseline. However, this is likely to be a coincidence caused by the random nature of the system. Whatever the case may be, the results show that the dynamic topology mechanism has no negative effect on the system's jitter.

5.1.4 Out of order packets

The number of packets received out of order was zero for all configurations, and has therefore not been graphed. While it may initially seem pointless to report these results, the rationale remains the same as for the other metrics: the baseline provides a point of reference to determine the effect of the dynamic topology mechanism. The fact that there is no variation between the baseline and the DTM measurements shows that the dynamic topology mechanism has no negative effect on the ordering of packets.

		Packet loss	
		Non-zero count	Sum (%)
OSPF	Scenario 1	6	0.262
	Scenario 2	5	0.147
	Scenario 3	1	0.330
MPLS	Scenario 1	3	0.099
	Scenario 2	2	0.130
	Scenario 3	6	0.603
DTM	Scenario 1	3	0.099
	Scenario 2	1	0.033
	Scenario 3	2	0.209

Table 5.3: Packet loss measurement summary

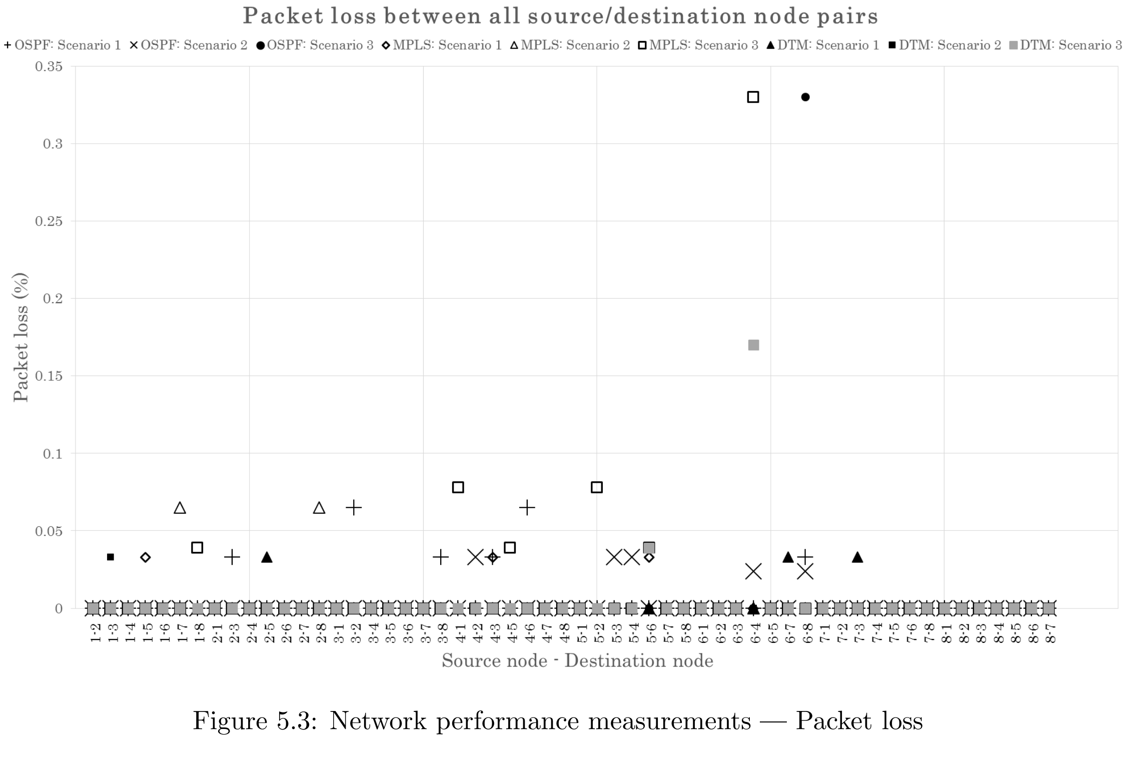

Figure 5.3: Network performance measurements — Packet loss

5.1.5 Traffic demands

In order to optimise the distribution of nodes in the standby state, the dynamic topology mechanism relies on the measurement of the demands between every source/destination node pair. Each of the nodes use sFlow messages to report the traffic demands, which is part of Open vSwitch. As the OSPF configuration uses Quagga rather than Open vSwitch to route traffic, measurements can only be examined for the MPLS and DTM configurations. The use of measurements form MPLS and DTM allows the impact of the topology changes to be examined in addition to the overall accuracy of the measurements.

The total demand measurements taken from the MPLS and DTM configurations, compared with the expected values, are shown in figure 5.4. The majority of the measured values are approximately 10% greater than the expected value, and the system takes 30 seconds for the measurements to adequately reflect the traffic variations. The cause of the positive error is unknown, but the 30 second lag is due to the measurement method, which uses the values collected over the past two 15 second measurement intervals. The percentage error of each of the individual demand measurements were averaged for each point in time, and are shown in figure 5.5. The peak values coincide with the traffic changes due to the measurement method's lag, and the values decrease with increasing demand.

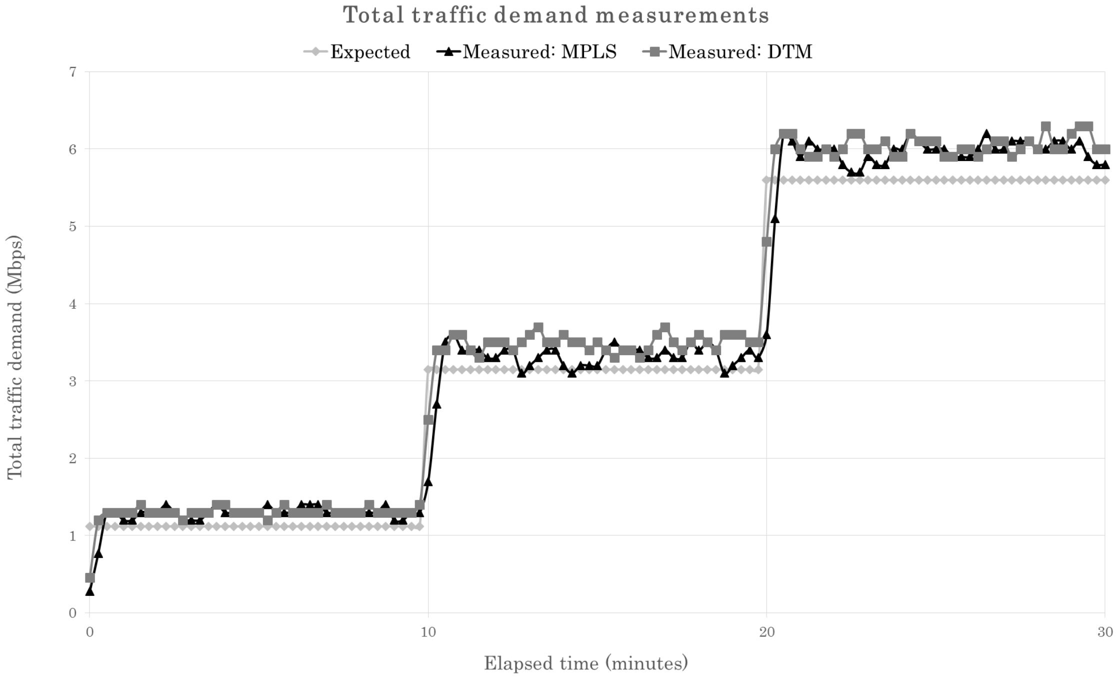

Figure 5.4: Network performance measurements — Total demand

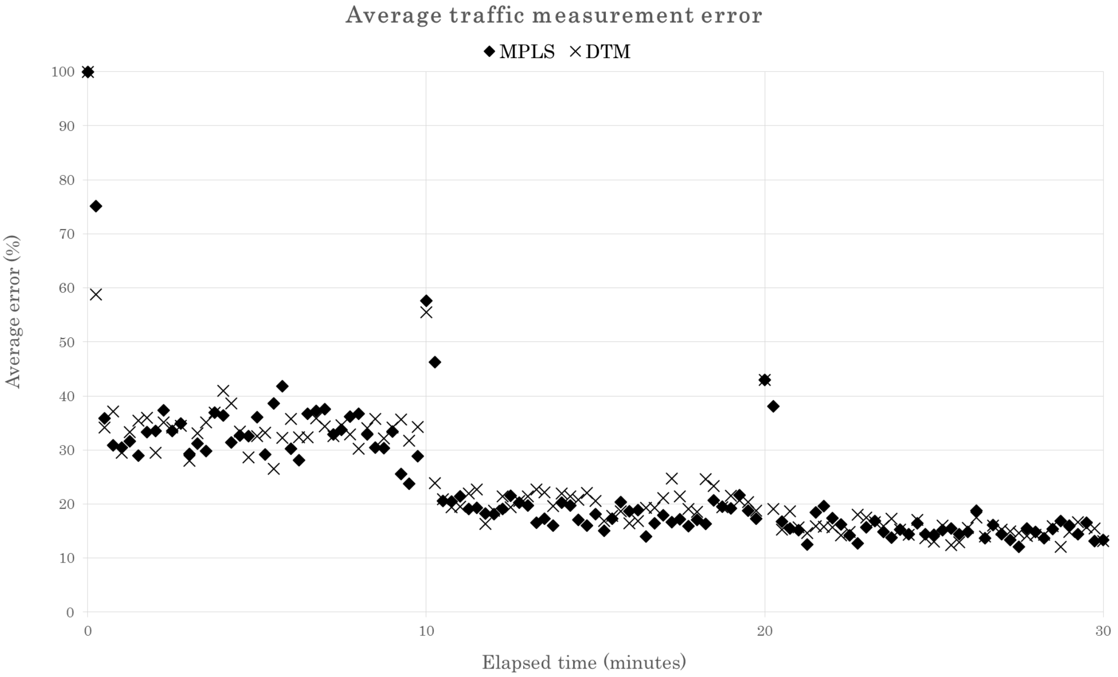

Figure 5.5: Network performance measurements — Average percent demand error

5.2 Discussion

Holistically speaking, the dynamic topology mechanism is quite acceptable, as it does not disproportionately affect the performance of the system when compared the OSPF and MPLS baseline. While not a requirement for its operation, the mechanism could be improved by reducing its effect on delay and increasing the inter-node demand measurement accuracy.

The accuracy of the inter-node demands is vital to the operation of the dynamic topology mechanism, as it provides the data used to optimise the distribution of nodes in the standby state while ensuring the link utilisations remain below the threshold value of 70%. The total demand typically experiences an error of +10%, with the individual measurements' error ranging from ±40% to ±10% and decreasing as demand increases. The operation of the dynamic topology mechanism is not affected by this inaccuracy if the change in demand over time is sufficiently large, but the fluctuation of the measured values could result in erroneous topology changes. Additional research is required to determine a solution that provides a sufficient increase in accuracy, but a preliminary suggestion is the increase in sFlow sampling rate. The inter-node demand measurements also experience a 30 second lag, which has both positive and negative effects. Most obviously, the lag has a negative effect on the mechanism's reaction time. However, this is reasonably minor, and may be mitigated by a reduction in unnecessary topology changes caused by traffic oscillating around a constant value. Additional research is required to determine the lag's effects.

The analysis above shows that the dynamic topology mechanism has negligible impact on the the majority of network performance metrics. There was no effect on the number of out of order packets, and the jitter and packet loss is either comparable to the baseline or attributable to the random nature of the system. However, the increase in delay is significant when the size of the network is considered. Furthermore, while the delay in the scenario with the least network traffic is in line with expectations, the delay's increase with increasing network traffic is not. The correlation between increasing traffic and delay may in fact prove to be a causative relationship, but requires additional research. While the increased delay is an unwanted side effect, the values can still be considered reasonable, and the overall impact on the system is likely to be minimal in practice. This is supported by the effects on the other network performance metrics, or lack thereof.

Chapter 6

Conclusion

This project has satisfied its aims and provided a foundation for future work towards the development of dynamic network topologies. This chapter details the project's achievements as they relate to the aims stated in chapter 1, and the potential future work to further build upon the concepts this project develops.

6.1 Project aims and achievements

As stated in chapter 1, this project aimed to contribute to the reduction of energy consumption in computer networks by exploiting their common under-utilisation. The energy consumption reductions are achieved through the use of node standby states, which require the use of dynamic network topolgies. As a contribution to the previous development of dynamic topology mechanisms, the key aspect of this project was the development of a dynamict topology mechanism implementation and its associated tested. This builds on the previous dynamic topology mechanism work, proves feasibility of implementation in a physical system and provides measurements of the mechanism's effects on network performance that are based on a real system, as opposed to simulated systems. This project also reduces the work required for future implementations that will further develop dynamic topology mechanisms.

The dynamic topology mechanism implementation described in chapter 4 modifies the network's topology based on the current network demands with the aim of placing the maximum number of nodes into a standby state while preventing the link utilisation from rising above a threshold value. The testbed is comprised of a network of eight Raspberry Pi model 1B+ devices acting as network nodes, which are connected to replicate a version of Telstra network AS1221 with link speeds scaled down by a factor of 1,000. Each node runs one of the two dynamic topology mechanism programs developed during this project. The second program runs on the centralised controller, a Dell laptop connected to one of the network nodes. A summary of the interaction between each program is shown below:

1. Controller: Monitor network traffic
2. Controller: Optimise network topology using heuristics
3. Controller/Nodes: Communicate the topology change
4. Nodes: Implement the topology change
5. Repeat

The network nodes route traffic using MPLS, and use Open vSwitch to provide the MPLS processing functionality. Topology changes are implemented by modifying Open vSwitch's flow tables, which have been configured to minimise the number modifications required. Open vSwitch has also be configured to send traffic statistics to the controller using sFlow messages. The controller analyses sFlow messages to determine the demands between each of the source/destination node pairs, and uses the resultant traffic matrix for topology optimisation. The focus of optimisation is the maximisation of the number of nodes in the standby state, while ensuring none of the network's link utilisations exceed the threshold value of 70%. Heuristics are used to simplify the optimisation, which is performed based on traffic measurements every 15 seconds, and the optimised topology is communicated to the nodes if a change has occurred.

Two additional programs were developed to facilitate testing of the network and the collation of the data produced by the tests, both of which are described in chapter 4. The testing program runs on the virtual hosts connected to the nodes. It generates the required traffic, and outputs network performance statistics to a group of files. The second program analyses the 336 files and collates the data into a single report on the network's performance

The analysis of network performance metrics shown in chapter 5 concludes that this project's dynamic topology mechanism's performance is within acceptable limits, but can be improved in a few areas. There is no negative effect on the jitter, packet loss, and out of order packets, but the delay is increased by a substantial amount. While the delay would have a negligible effect on the system's usability and could be considered reasonable in other circumstances, it is uncharacteristically large for the network's size. The delay also increases with increasing network load, which contradicts the initial expectation; the paths between the nodes become less circuitous with increasing traffic, as the dynamic topology mechanism reduces the number of nodes in the standby state. The accuracy of the traffic demand measurement, while sufficient for the dynamic topology mechanism's operation, is also somewhat lacking, with errors ranging from ±40% to ±10%. The mechanism still functions with this error, but this may be due to the large variations in network demand that have been selected for the test conditions. As the mechanism is so dependent on the demand measurements, an increase in accuracy should improve its effectiveness and reduce the likelihood of erroneous topology changes.

6.2 Project limitations and future work

There are some areas of the project that require additional development, and others that can be used as the foundation for future work. The list below provides an overview of these areas:

Delay reduction: As alluded to above, the increase in delay caused by the dynamic topology mechanism is uncharacteristically large. Additional research is required to determine and rectify the cause of the increased delay.

Traffic measurement accuracy improvement: Also alluded to above, an increase in accuracy is likely to improve the effectiveness of the dynamic topology mechanism.

Failure recovery and reliability: Like most of the work on energy consumption reduction in networks, this project does not consider failure recovery and reliability. Future work could include this as a consideration in the topology selection and overall system design.

Optimisation heuristic development: The optimisation heuristic has been developed based on several others, and its performance has not been measured. This project's dynamic topology mechanism could be relatively easily modified to utilise a more effective heuristic algorithm.

Implement standby states: The project has omitted the implementation of standby states, and has simply modified the node's behaviour to reflect the effect of the standby state on the network. As with the above point, this project's dynamic topology mechanism could be modified relatively easily to incorporate the use of standby states once their implementation has been developed.

Energy consumption measurement: As this project did not include the implementation of standby states, the dynamic topology mechanism's effect on energy consumption was not analysed. The implementation of standby states would allow the determination of the mechanism's effect on energy consumption.

Longevity analysis: A side effect of energy consumption reduction may be increased network equipment life. Once standby states have been implemented, future work could include an analysis of the resultant mean time between failure, failure modes, and other metrics.

Specialised hardware development: The prototype nature of the implementation prompted the selection of the readily available Raspberry Pi for the network nodes, and the scaling down of link bandwidths by a factor of 1,000. The use of specialised routing hardware, rather than hobby computers repurposed as routers, would allow for a more accurate analysis of the dynamic topology mechanism's effects on network performance and energy consumption.

References

Akyildiz, I. F., Lee, A., Wang, P., Luo, M. & Chou, W. (2014), 'A roadmap for traffic engineering in SDN-OpenFlow networks', *Computer Networks* **71**, 1–30.

Aldraho, A. & Kist, A. (2010), Heuristics for dynamic topologies to reduce power consumption of networks, *in* 'Telecommunication Networks and Applications Conference (ATNAC), 2010 Australasian', pp. 31–36.

Aldraho, A. & Kist, A. (2011*a*), Enabling Dynamic Topologies in communication networks, *in* 'Australasian Telecommunication Networks and Applications Conference (ATNAC), 2011', pp. 1–6.

Aldraho, A. & Kist, A. A. (2011*b*), 'Dynamic topologies for sustainable and energy efficient traffic routing', *Computer Networks* **55**(9), 2271–2288.

Aldraho, A., Kist, A. & Maxwell, A. (2012), Performance investigation of dynamic topologies in MPLS networks, *in* '2012 International Symposium on Communications and Information Technologies (ISCIT)', pp. 967–972.

Amaldi, E., Capone, A., Gianoli, L. & Mascetti, L. (2011), Energy management in IP traffic engineering with Shortest Path routing, *in* 'World of Wireless, Mobile and Multimedia Networks (WoWMoM), 2011 IEEE International Symposium on a', pp. 1–6.

Awduche, D. (1999), 'MPLS and traffic engineering in IP networks', *IEEE Communications Magazine* **37**(12), 42–47.

Ben Ameur, W., Bourquia, N., Gourdin, E. & Tolla, P. (2002), Optimal routing for efficient Internet networks, *in* '2nd European Conference on Universal Multiservice Networks, 2002. ECUMN 2002', pp. 10–17.

Bolla, R., Bruschi, R., Cianfrani, A. & Listanti, M. (2011), 'Enabling backbone networks to sleep', *IEEE Network* **25**(2), 26–31.

Bozakov, Z. & Papadimitriou, P. (2013), OpenVRoute: An open architecture for high-performance programmable virtual routers, *in* '2013 IEEE 14th International Conference on High Performance Switching and Routing (HPSR)', pp. 191–196.

Chabarek, J., Sommers, J., Barford, P., Estan, C., Tsiang, D. & Wright, S. (2008), Power Awareness in Network Design and Routing, *in* 'IEEE INFOCOM 2008. The 27th Conference on Computer Communications', pp. –.

Chiaraviglio, L., Mellia, M. & Neri, F. (2012), 'Minimizing ISP Network Energy Cost: Formulation and Solutions', *IEEE/ACM Transactions on Networking* **20**(2), 463–476.

Chu, H.-W., Cheung, C.-C., Ho, K.-H. & Wang, N. (2011), Green MPLS Traffic Engineering, *in* 'Australasian Telecommunication Networks and Applications Conference (ATNAC), 2011', pp. 1–4.

Cianfrani, A., Eramo, V., Listanti, M. & Polverini, M. (2012), Introducing routing standby in network nodes to improve energy savings techniques, *in* '2012 Third International Conference on Future Energy Systems: Where Energy, Computing and Communication Meet (e-Energy)', pp. 1–7.

Coiro, A., Listanti, M., Valenti, A. & Matera, F. (2013), 'Energy-aware traffic engineering: A routing-based distributed solution for connection-oriented IP networks', *Computer Networks* **57**(9), 2004–2020.

Cuomo, F., Cianfrani, A., Polverini, M. & Mangione, D. (2012), 'Network pruning for energy saving in the Internet', *Computer Networks* **56**(10), 2355–2367.

Elwalid, A., Widjaja, I., Awduche, D., Chiu, A. & Xiao, X. (1998), 'Requirements for Traffic Engineering Over MPLS'.

Eramo, V., Cianfrani, A., Listanti, M., Polverini, M. & Vasilakos, A. (2012), 'An OSPF-Integrated Routing Strategy for QoS-Aware Energy Saving in IP Backbone Networks', *IEEE Transactions on Network and Service Management* **9**(3), 254–267.

Fortz, B. & Thorup, M. (2000), Internet traffic engineering by optimizing OSPF weights, *in* 'IEEE INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings', Vol. 2, pp. 519–528 vol.2.

Fraleigh, C., Moon, S., Lyles, B., Cotton, C., Khan, M., Moll, D., Rockell, R., Seely, T. & Diot, S. (2003), 'Packet-level traffic measurements from the Sprint IP backbone', *IEEE Network* **17**(6), 6–16.

Gartner (2007), 'Gartner Says Data Centres Account for 23 per cent of Global ICT CO2 Emissions', *Press Release, [Online] at <http://www.gartner.com/it/page.jsp?id=530912>* .

Gupta, M. & Singh, S. (2003), Greening of the Internet, *in* 'Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications', SIGCOMM '03, ACM, New York, NY, USA, pp. 19–26.

Han, J. H., Mundkur, P., Rotsos, C., Antichi, G., Dave, N. H., Moore, A. W. & Neumann, P. G. (2015), Blueswitch: Enabling Provably Consistent Configuration of Network Switches, *in* 'Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems', ANCS '15, IEEE Computer Society, Washington, DC, USA, pp. 17–27.

Hundessa, L. & Domingo-Pascual, J. (2002), Reliable and fast rerouting mechanism for a protected label switched path, *in* 'IEEE Global Telecommunications Conference, 2002. GLOBECOM '02', Vol. 2, pp. 1608–1612 vol.2.

Kaup, F., Gottschling, P. & Hausheer, D. (2014), PowerPi: Measuring and modeling the power consumption of the Raspberry Pi, *in* '2014 IEEE 39th Conference on Local Computer Networks (LCN)', pp. 236–243.

Kim, Y.-M., Lee, E.-J., Park, H.-S., Choi, J.-K. & Park, H.-S. (2012), 'Ant colony based self-adaptive energy saving routing for energy efficient Internet', *Computer Networks* **56**(10), 2343–2354.

Koenigsmayr, M. & Neubauer, T. (2015), 'The Role of ICT in a Low Carbon Society', *IEEE Technology and Society Magazine* **34**(1), 39–44.

Kvalbein, A. & Lysne, O. (2007), How Can Multi-topology Routing Be Used for Intradomain Traffic Engineering?, *in* 'Proceedings of the 2007 SIGCOMM Workshop on Internet Network Management', INM '07, ACM, New York, NY, USA, pp. 280–284.

Lee, S. S. W., Tseng, P.-K. & Chen, A. (2012), 'Link weight assignment and loop-free routing table update for link state routing protocols in energy-aware internet', *Future Generation Computer Systems* **28**(2), 437–445.

Machizawa, A. & Iwama, T. (2013), Exponential degrading of NTP synchronization with number of network hops, *in* 'European Frequency and Time Forum International Frequency Control Symposium (EFTF/IFC), 2013 Joint', pp. 97–100.

Nedevschi, S., Popa, L., Iannaccone, G., Ratnasamy, S. & Wetheral, D. (2008), 'Reducing network energy consumption via sleeping and rate-adaptation', *in: NSDI08: Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation, USENIX Association, Berkeley, CA, USA* pp. 323–336.

Ohsita, Y., Miyamura, T., Arakawa, S., Ata, S., Oki, E., Shiomoto, K. & Murata, M. (2010), 'Gradually Reconfiguring Virtual Network Topologies Based on Estimated Traffic Matrices', *IEEE/ACM Trans. Netw.* **18**(1), 177–189.

Pan, T., Zhang, T., Shi, J., Li, Y., Jin, L., Li, F., Yang, J., Zhang, B., Yang, X., Zhang, M., Dai, H. & Liu, B. (2015), 'Towards Zero-Time Wakeup of Line Cards in Power-Aware Routers', *IEEE/ACM Transactions on Networking* **PP**(99), 1–14.

Polverini, M., Cianfrani, A., Coiro, A., Listanti, M. & Bruschi, R. (2015), 'Freezing forwarding functionality to make the network greener', *Computer Networks* **78**, 26–41.

Roberts, L. (2009), 'A radical new router', *IEEE Spectrum* **46**(7), 34–39.

Schnitter, S. & Horneffer, M. (2004), Traffic matrices for MPLS networks with LDP traffic statistics, *in* 'Telecommunications Network Strategy and Planning Symposium. NETWORKS 2004, 11th International', pp. 231–236.

Schwalbe, K. (2014), *Information Technology Project Management*, 7 edn, Cengage Learning, Boston, USA.

Sharafat, A. R., Das, S., Parulkar, G. & McKeown, N. (2011), MPLS-TE and MPLS VPNS with Openflow, *in* 'Proceedings of the ACM SIGCOMM 2011 Conference', SIGCOMM '11, ACM, New York, NY, USA, pp. 452–453.

Simsek, O. & Pospiech, M. (2013), A network performance measurement tool, *in* '2013 5th IEEE International Conference on Broadband Network Multimedia Technology (IC-BNMT)', pp. 45–48.

Suryasaputra, R., Kist, A. & Harris, R. (2005), Verification of MPLS traffic engineering techniques, *in* ', 2005 13th IEEE International Conference on Networks, 2005. Jointly held with the 2005 IEEE 7th Malaysia International Conference on Communication', Vol. 1, pp. 6 pp.–.

Takeshita, H., Yamanaka, N., Okamoto, S., Shimizu, S. & Gao, S. (2012), 'Energy efficient network design tool for green IP/Ethernet networks', *Optical Switching and Networking* **9**(3), 264–270.

Vasi, N., Bhurat, P., Novakovi, D., Canini, M., Shekhar, S. & Kosti, D. (2011), Identifying and Using Energy-critical Paths, *in* 'Proceedings of the Seventh COnference on Emerging Networking EXperiments and Technologies', CoNEXT '11, ACM, New York, NY, USA, pp. 18:1–18:12.

Vasi, N. & Kosti, D. (2010), Energy-aware Traffic Engineering, *in* 'Proceedings of the 1st International Conference on Energy-Efficient Computing and Networking', e-Energy '10, ACM, New York, NY, USA, pp. 169–178.

Wang, J., Ruepp, S., Manolova, A., Dittmann, L., Ricciardi, S. & Careglio, D. (2012), Green-aware routing in GMPLS networks, *in* '2012 International Conference on Computing, Networking and Communications (ICNC)', pp. 227–231.

Wang, R., Jiang, Z., Gao, S., Yang, W., Xia, Y. & Zhu, M. (2014), Energy-aware routing algorithms in Software-Defined Networks, *in* '2014 IEEE 15th International Symposium on A World of Wireless, Mobile and Multimedia Networks (WoWMoM)', pp. 1–6.

Wang, Y., Wang, H., Mahimkar, A., Alimi, R., Zhang, Y., Qiu, L. & Yang, Y. R. (2010), R3: Resilient Routing Reconfiguration, *in* 'Proceedings of the ACM SIGCOMM 2010 Conference', SIGCOMM '10, ACM, New York, NY, USA, pp. 291–302.

Webb, M. (2008), 'Smart 2020: Enabling the Low Carbon Economy in the Information Age'.

Yang, Y., Xu, M., Wang, D. & Li, S. (2015), 'A Hop-by-hop Routing Mechanism for Green Internet', *IEEE Transactions on Parallel and Distributed Systems* **PP**(99), 1–1.

Zhang, C., Ge, Z., Kurose, J., Liu, Y. & Towsley, D. (2005), Optimal routing with multiple traffic matrices tradeoff between average and worst case performance, *in* '13th IEEE International Conference on Network Protocols, 2005. ICNP 2005', pp. 10 pp.–.

Zhang, M., Yi, C., Liu, B. & Zhang, B. (2010), GreenTE: Power-aware traffic engineering, *in* '2010 18th IEEE International Conference on Network Protocols (ICNP)', pp. 21–30.

Appendix A

Project Specification

ENG 4111/2 Research Project

Project Specification

For:	**Daniel Costantini**
Topic:	Green IT - Dynamic Network Topologies
Supervisor:	Dr. Alexander Kist
Sponsorship:	Faculty of Health, Engineering & Sciences
Project Aim:	Extend the Linux Multi-Protocol Label Switching (MPLS) implementation to develop a simple dynamic topologies mechanism that reacts to changes in network traffic loads by controlling the transition of network nodes between the standby and active states, with the aim of decreasing total network energy consumption.

Program:

1. Perform literature review to determine current status of research regarding implementation of dynamic topologies using MPLS
2. Determine the project scope and specifications of the extended Linux MPLS implementation.
3. Construct a plan for development and testing of the implementation
4. Perform initial development of the implementation
5. Perform testing of the implementation
6. Compile the results of the research, development, and testing into an academic dissertation

As time and resources permit:

1. Refine the implementation's performance through additional development and testing iterations
2. Compare the extended implementation's power consumption to that of the baseline implementation

Agreed:

Student Name: Daniel Costantini
Date: 18 MAR 15

Supervisor Name: Alexander Kist
Date: 18 MAR 15

Appendix B

Software configuration instruction

As described in section 4.1.2, the configuration instructions below can be used to recreate the controller's, network nodes', and hosts' configuration for the dynamic topology mechanism.

B.1 Controller configuration

The following procedure was performed on a Dell Vostro V13 laptop to install the software required for the operation of the dynamic topology mechanism controller.

1. Install Ubuntu 14.04.2

2. Make sure all packages are up-to-date and install extra packages

```
sudo apt-get update && sudo apt-get -y --force-yes upgrade
sudo apt-get install ftp autoconf libtool git
    libcurl4-openssl-dev ntp
```

3. Install sFlowtool

```
git clone https://github.com/sflow/sflowtool.git
cd sflowtool-3.36
automake && autoconf
./configure && make && make install
```

4. Modify `/etc/network/interfaces`

```
sudo vi /etc/network/interfaces
   auto eth0
   allow-hotplug eth0
   iface eth0 inet static
     address 192.168.254.100
```

5. Configure the current timezone

```
sudo dpkg-reconfigure tzdata
```

6. Configure NTP to synchronise to local clock

```
sudo vi /etc/ntp.conf
   server 127.127.1.0
   fudge 127.127.1.0 stratum 10
```

7. Configure hostname

```
sudo vi /etc/hosts
   127.0.1.1     controller
sudo vi /etc/hostname
   controller
sudo /etc/init.d/hostname.sh
```

8. Configure connection to network

```
sudo ifconfig eth0 192.168.254.100
```

B.2 Network nodes and hosts configuration

The following procedure was performed on each of the eight Raspberry Pi model 1B+ computers to install the software required for the operation of the system's nodes and hosts, and allows them to support the dynamic topology mechanism.

1. Flash SD card with Raspbian image

2. Boot once, expand filesystem when prompted, reboot

3. Make sure all packages are up-to-date and install extra packages

```
sudo apt-get update && sudo apt-get dist-upgrade
sudo apt-get install ftp autoconf libtool quagga ncurses-dev bc
```

```
    automake libcap-dev libcurl4-openssl-dev graphviz debhelper
    python-zopeinterface dh-autoreconf libssl-dev python-all
    python-qt4 python-twisted-conch debootstrap wondershaper
sudo apt-get -f -y --force-yes install
```

4. Configure hostname (example given for node 7)

```
sudo vi /etc/hosts
    127.0.1.1     node7
sudo vi /etc/hostname
    node7
sudo /etc/init.d/hostname.sh
```

5. Update Raspberry Pi firmware

```
cd /opt
sudo git clone git://github.com/raspberrypi/firmware.git
cd firmware/boot && sudo cp -r * /boot
cd ../modules && sudo cp -r * /lib/modules
sudo rm -r /opt/firmware
```

6. Clone Raspbian kernel source (for LXC container build)

```
sudo mkdir /opt/raspberrypi
cd /opt/raspberrypi
sudo git clone git://github.com/raspberrypi/linux.git
```

7. Compile kernel with options set for LXC container communication

```
cd /opt/raspberrypi/linux
sudo make bcmrpi_defconfig
sudo make menuconfig
   Device Drivers → Network Device Support →
   Virtual Ethernet pair device → Enabled
sudo make && sudo make modules_install
cd /opt/raspberrypi && sudo git clone
   git://github.com/raspberrypi/tools.git
cd tools/mkimage && sudo python ./imagetool-uncompressed.py
   /opt/raspberrypi/linux/arch/arm/boot/Image
sudo cp kernel.img /boot/
sudo shutdown -r now
```

8. Install the latest LXC (Don't use `apt-get`)

```
sudo mkdir /opt/lxc && cd /opt/lxc
sudo git clone https://github.com/lxc/lxc.git && cd lxc
sudo ./autogen.sh && sudo ./configure
sudo make && sudo make install
sudo rm -r /opt/lxc
```

9. Configure interfaces (example given for node 7)

```
sudo vi /etc/network/interfaces
    auto eth0
    allow-hotplug eth0
    iface eth0 inet static
      address 192.168.17.7
    auto eth1
    allow-hotplug eth1
    iface eth1 inet static
      address 192.168.67.7
    auto eth2
    allow-hotplug eth2
    iface eth2 inet static
      address 192.168.27.7
    auto eth3
    allow-hotplug eth3
    iface eth3 inet static
      address 192.168.78.7
```

10. Configure Quagga (example given for node 7)

```
sudo vi /etc/quagga/zebra.conf
    hostname RouterPi
    password router
    enable password router
    interface lo
    interface veth0
      ip address 192.168.70.1/24
    interface eth0
      ip address 192.168.17.7/24
    interface eth1
      ip address 192.168.67.7/24
    interface eth2
      ip address 192.168.27.7/24
    interface eth3
      ip address 192.168.78.7/24
sudo vi /etc/quagga/ospfd.conf
    hostname RouterPi
    password router
    enable password router
    router ospf
      network 192.168.70.0/24 area 0
      network 192.168.17.0/24 area 0
      network 192.168.67.0/24 area 0
      network 192.168.27.0/24 area 0
      network 192.168.78.0/24 area 0
sudo chown quagga /etc/quagga/ospfd.conf
sudo chgrp quagga /etc/quagga/ospfd.conf
sudo chown quagga /etc/quagga/zebra.conf
sudo chgrp quagga /etc/quagga/zebra.conf
sudo vi /etc/quagga/daemons
    zebra=yes
    ospfd=yes
```

11. Enable IP forwarding

```
sudo vi test
    1
sudo mv test /proc/sys/net/ipv4/ip_forward
sudo vi /etc/sysctl.conf
    net.ipv4.ip_forward=1
```

12. Install Open vSwitch

```
git clone https://github.com/openvswitch/ovs && cd ovs
dpkg-checkbuilddeps
./boot.sh && ./configure && make && sudo make install
sudo mkdir -p /usr/local/etc/openvswitch
```

```
    sudo ovsdb-tool create /usr/local/etc/openvswitch/conf.db
        vswitchd/vswitch.ovsschema
```

13. Create LXC container (example given for node 7)

```
    sudo vi /etc/fstab
        lxc /sys/fs/cgroup cgroup defaults 0 0
    sudo mount -a
    sudo LD_LIBRARY_PATH=/usr/local/lib SUITE=wheezy
        MIRROR=http://archive.raspbian.org/raspbian
        lxc-create -n host7 -t debian
        -- --packages=iperf,iputils-ping
    sudo vi /usr/local/var/lib/lxc/host7/config
        lxc.start.auto = 1
        lxc.start.delay = 5
        lxc.network.type = veth
        lxc.network.veth.pair = veth0
        lxc.network.flags = up
        lxc.network.script.up = /etc/lxc/ovsup
        lxc.rootfs = /usr/local/var/lib/lxc/host7/rootfs
        lxc.network.ipv4 = 192.168.70.100/24
        lxc.network.ipv4.gateway = 192.168.70.1
        lxc.include = /usr/local/share/lxc/config/debian.common.conf
        lxc.utsname = host7
        lxc.arch = armhf
    sudo mkdir /etc/lxc
    sudo vi /etc/lxc/ovsup
        #!/bin/bash
        BRIDGE="br0"
        ovs-vsctl --may-exist add-br $BRIDGE
        ovs-vsctl --if-exists del-port $BRIDGE $5
        ovs-vsctl add-port $BRIDGE $5
        ovs-vsctl set interface $5 ofport_request=1
        ifconfig $5 192.168.70.1
    sudo chmod 777 /etc/lxc/ovsup
```

14. Set timezone and configure NTP to get time from controller

```
sudo dpkg-reconfigure tzdata
sudo vi /etc/ntp.conf
    server 192.168.254.100 prefer
sudo service ntp stop && sudo ntpd -gq && sudo service ntp start
```

15. Apply traffic shaping to interfaces (example given for node 7)

```
sudo wondershaper eth0 10000 10000
sudo wondershaper eth1 1000 1000
sudo wondershaper eth2 1000 1000
sudo wondershaper eth3 1000 1000
```

16. Initial Open vSwitch configuration (example given for node 7)

```
sudo ovsdb-server
    --remote=punix:/usr/local/var/run/openvswitch/db.sock
    --remote=db:Open_vSwitch,Open_vSwitch,manager_options
    --pidfile --detach
sudo ovs-vsctl --no-wait init
sudo ovs-vswitchd --pidfile --detach
sudo ovs-vsctl del-br br0
sudo ovs-vsctl add-br br0 -- set bridge br0 datapath_type=netdev
sudo ovs-vsctl set bridge br0
    other-config:datapath-id=0000000000000007
sudo ovs-vsctl set bridge br0 protocols=OpenFlow10,OpenFlow11,
    OpenFlow12,OpenFlow13,OpenFlow14,OpenFlow15
sudo ovs-vsctl add-port br0 eth0
sudo ovs-vsctl set interface eth0 ofport_request=2
sudo ovs-vsctl add-port br0 eth1
sudo ovs-vsctl set interface eth1 ofport_request=3
sudo ovs-vsctl add-port br0 eth2
sudo ovs-vsctl set interface eth2 ofport_request=4
sudo ovs-vsctl add-port br0 eth3
sudo ovs-vsctl set interface eth3 ofport_request=5
sudo ovs-ofctl add-flow br0 "priority=5, actions=drop"
sudo ovs-vsctl -- --id=@sflow create sflow agent=veth0
    target=\"192.168.254.100:6343\" header=128 sampling=8
    polling=0 -- set bridge br0 sflow=@sflow
```

17. Set IP addresses (example given for node 7)

```
sudo ifconfig eth0 192.168.17.7

sudo ifconfig eth1 192.168.67.7

sudo ifconfig eth2 192.168.27.7

sudo ifconfig eth3 192.168.78.7
```

Appendix C

Dynamic Topology Mechanism Source Code

The operation of the dynamic topology mechanism is detailed in chapter 4. The source code shown below corresponds to four programs: the controller's dynamic topology mechanism program, the nodes' dynamic topology mechanism program, the hosts' traffic generation program, and the controller's traffic statistic collation program.

C.1 Controller program's source code

This program is used on the controller to monitor the network traffic, select the appropriate topology, and communicate it to the network nodes. It uses the output of sFlow, which is piped to the program on the command line as follows:

```
daniel@controller:~$ sflowtool -l | sudo ./controller
```

To execute the program with the dynamic topology mechanism disabled, any number of command line arguments can be provided, an example of which is shown below:

```
daniel@controller:~$ sflowtool -l | sudo ./controller nodtm
```

The code listings below show the main program, responsible for traffic monitoring and calling the optimisation function, and the optimisation algorithm itself. Compilation of the program was performed using g++ as follows:

```
daniel@controller:~$ g++ traffic.cpp topology.cpp -o controller
```

C.1.1 Main program

The controller's main program shown in listing C.1 performs the following functions:

- Processing of the sFlow message stream input
- Construction of the traffic matrix using statistics from sFlow messages
- Calling of the optimisation function
- Checking if a topology change has occurred, and communicating the change if required

Listing C.1: Controller main program C++ source code

```
#include <iostream>
#include <iomanip>
#include <string>
#include <sstream>
#include <vector>
#include <stdlib.h>
#include <sys/time.h>
#include <fstream>
#include <bitset>
#include <cstring>
#include "topology.h"
using namespace std;


typedef unsigned long long timestamp_t;
static timestamp_t get_timestamp (){
  struct timeval now;
  gettimeofday (&now, NULL);
  return  now.tv_usec + (timestamp_t)now.tv_sec * 1000000;
}


int main(int argc, const char * argv[]) {
    int MEASURE_INTERVAL = 15;//interval in seconds between traffic
        calculations

    int NUM_NODES = 8;//number of nodes in network
    //node numbers represented as their matrix indices
    int N1 = 0; int N2 = 1; int N3 = 2; int N4 = 3;
    int N5 = 4; int N6 = 5; int N7 = 6; int N8 = 7;

    fstream configNew, configOld;
    string configFilename = "./topology.conf";

    int i,j, agent, source, dest, packetSize, sampleRate;
    vector<int> trafficCountCurrent(NUM_NODES*NUM_NODES);//traffic
       count for each S-D pair (this interval)
    vector<int> trafficCountLast(NUM_NODES*NUM_NODES);//traffic
       count for each S-D pair (previous interval)
    vector<double> traffic(NUM_NODES*NUM_NODES);//traffic for each
```

```
   S-D pair
double tdiff, totalTraffic;
double tdifflast = 0;
string input, dumpstr, lineOld, lineNew, line;
char tempchar;
stringstream tempstr;
adjacency_list_t adjacency_list(NUM_NODES);
timestamp_t t0;//used to calculate measurement interval
bool topologyChange;

//initialise adjacency matrix (adjacent node, link weight)
adjacency_list[N1].push_back(neighbor(N2, 10));
adjacency_list[N1].push_back(neighbor(N6, 10));
adjacency_list[N1].push_back(neighbor(N7, 1));

adjacency_list[N2].push_back(neighbor(N1, 10));
adjacency_list[N2].push_back(neighbor(N3, 10));
adjacency_list[N2].push_back(neighbor(N7, 10));

adjacency_list[N3].push_back(neighbor(N2, 10));
adjacency_list[N3].push_back(neighbor(N4, 10));
adjacency_list[N3].push_back(neighbor(N5, 1));
adjacency_list[N3].push_back(neighbor(N8, 1));

adjacency_list[N4].push_back(neighbor(N3, 10));
adjacency_list[N4].push_back(neighbor(N8, 10));

adjacency_list[N5].push_back(neighbor(N3, 1));
adjacency_list[N5].push_back(neighbor(N6, 10));

adjacency_list[N6].push_back(neighbor(N1, 10));
adjacency_list[N6].push_back(neighbor(N5, 10));
adjacency_list[N6].push_back(neighbor(N7, 10));

adjacency_list[N7].push_back(neighbor(N1, 1));
adjacency_list[N7].push_back(neighbor(N2, 10));
adjacency_list[N7].push_back(neighbor(N6, 10));
adjacency_list[N7].push_back(neighbor(N8, 10));

adjacency_list[N8].push_back(neighbor(N3, 1));
adjacency_list[N8].push_back(neighbor(N4, 10));
adjacency_list[N8].push_back(neighbor(N7, 10));

//initialise traffic counts
for (i=0;i<trafficCountCurrent.size();i++){
    trafficCountCurrent[i] = 0;
    trafficCountLast[i] = 0;
}
//capture start time
t0 = get_timestamp();
while (1) {
    //clear data from last capture
    totalTraffic = 0;
    tempstr.str(string());
    tempstr.clear();

    //capture traffic data
    getline(cin,line);
```

```
tempstr << line;
getline(tempstr,line,’,’);
//skip processing if sFlow message is not a flow sample
if(strcmp(line.c_str(),("FLOW"))==0){
    //get agent node number
    getline(tempstr,dumpstr,’.’);//dump first octet of
       agent address
    getline(tempstr,dumpstr,’.’);//dump second octet of
       agent address
    getline(tempstr,line,’.’);//capture third octet of
       agent address
    agent = atof(line.c_str());
    if (agent%10 != 0){//discard message if the agent is an
        unexpected value
        continue;
    }
    agent /= 10;

    //Unwanted data
    getline(tempstr,dumpstr,’,’);//dump remainder of agent
       address
    getline(tempstr,dumpstr,’,’);//dump input port
    getline(tempstr,dumpstr,’,’);//dump output port
    getline(tempstr,dumpstr,’,’);//dump source mac
    getline(tempstr,dumpstr,’,’);//dump dest mac
    getline(tempstr,dumpstr,’,’);//dump ethertype
    getline(tempstr,dumpstr,’,’);//dump in vlan
    getline(tempstr,dumpstr,’,’);//dump out vlan

    //get source node number
    tempstr.get(tempchar);
    if (tempchar==’-’){//discard message if the source IP
       is not specified
        continue;
    }
    getline(tempstr,dumpstr,’.’);//dump first octet of
       source address
    getline(tempstr,dumpstr,’.’);//dump second octet of
       sourse address
    getline(tempstr,line,’.’);//capture third octet of
       source address
    source = atof(line.c_str());
    if (source%10 != 0){//discard message if the source is
       an unexpected value
        continue;
    }
    source /= 10;
    getline(tempstr,dumpstr,’,’);//dump remainder of source
        address

    //get destination node number
    getline(tempstr,dumpstr,’.’);//dump first octet of dest
        address
    getline(tempstr,dumpstr,’.’);//dump second octet of
       destaddress
    getline(tempstr,line,’.’);//capture third octet of dest
        address
    dest = atof(line.c_str());
```

```
if (dest%10 != 0){
//    cout << "Halting message processing\n\n";
    continue;
}
dest /= 10;
getline(tempstr,dumpstr,',');//dump remainder of dest
   address

//unwanted data
getline(tempstr,dumpstr,',');//dump IP protocol
getline(tempstr,dumpstr,',');//dump IP TOS
getline(tempstr,dumpstr,',');//dump IP TTL
getline(tempstr,dumpstr,',');//dump TCP/UDP source port
getline(tempstr,dumpstr,',');//dump TCP/UDP dest port
getline(tempstr,dumpstr,',');//dump TCP flags

//unwanted data
getline(tempstr,line,',');//dump packet size

//get IP packet size
getline(tempstr,dumpstr,',');//capture IP packet size
packetSize = atof(line.c_str());

//get sampling rate
getline(tempstr,line);//capture sampling rate
sampleRate = atof(line.c_str());

//increment relevant S/D traffic counter
if (agent==dest) trafficCountCurrent[(source-1)*
   NUM_NODES+(dest-1)] += (packetSize*sampleRate);

//find time since last calculation
tdiff = (double)((get_timestamp() - t0)/1000000.0L);

//Calculate traffic rates if the measurement interval
   has passed
if (tdiff >= MEASURE_INTERVAL){
    //capture new start time
    t0 = get_timestamp();

    //calculate traffic demands using samples from last
        two intervals
    for (i=0;i<trafficCountCurrent.size();i++){
        traffic[i] = (double)((trafficCountCurrent[i] +
            trafficCountLast[i]) * 8 / (tdiff+tdifflast
           ) / 1000000.0L);
    }

    //store time difference
    tdifflast = tdiff;

    //store samples from last interval
    trafficCountLast = trafficCountCurrent;

    //reinitialise traffic counters and show current
       traffic
    cout << "= = = = = = = = = = = = = = = = = = = = =
       = = = = = = = = = = = = = = = = = = = = = = = =
    = = = = = = = = = = = = = = = = = = = = =\n";
cout << "Traffic (Mbps): \n" << setprecision(2) <<
    left;
for (i=0;i<trafficCountCurrent.size();i++){
     trafficCountCurrent[i] = 0;
     cout << "    " << (int)(i/NUM_NODES)+1 << "-"
         << (i%NUM_NODES)+1 << ": " << setw(6) <<
         traffic[i] << " Mbps";
     if (i%NUM_NODES==7)
          cout << "\n";
     totalTraffic = totalTraffic + traffic[i];
}
cout << "Total Traffic (Mbps): " << totalTraffic;
cout << "\n- - - - - - - - - - - - - - - - - - - - -
    - - - - - - - - - - - - - - - - - - - - - - - -
    - - - - - - - - - - - - - - - - - - - - -\n";

//create new topology file
system(("sudo touch " + configFilename + ".new").
    c_str());
system(("sudo chown daniel " + configFilename + ".
    new").c_str());
system(("sudo chgrp daniel " + configFilename + ".
    new").c_str());


//write config to new topology file
//If argc > 1, the dynamic topology mechanism will
    be disabled
topology_select(traffic, adjacency_list,
    configFilename + ".new", argc);

//Open config files
configNew.open((configFilename + ".new").c_str());
if (configNew.fail()){
     cout << "Error opening " << configFilename << "
         .new\n";
     return 1;
}
configOld.open(configFilename.c_str());
if (configOld.fail()){
     //In case the file doesn't exist, try to create
          it and try again
     system(("sudo touch " + configFilename).c_str()
         );
     configOld.open(configFilename.c_str());
     if (configOld.fail()){
          cout << "Error opening " << configFilename
              << "\n";
          return 1;
     }
}

//check if the current topology matches the new
    topology (prevents unneccessary processing by
    the nodes)
topologyChange = false;
while(!configOld.eof() && !configNew.eof()){
```

```
                    getline(configOld,lineOld);
                    getline(configNew,lineNew);
                    if (strcmp(lineOld.c_str(),string().c_str())==0
                        || strcmp(lineOld.c_str(),lineNew.c_str())
                       !=0){
                        topologyChange = true;
                        break;
                    }
                }
                configOld.close();
                configNew.close();
                if (topologyChange){
                    //overwrite old config file
                    cout << "Updating topology\n";
                    system(("sudo mv " + configFilename + ".new " +
                        configFilename).c_str());
                }
                else{
                    cout << "No topology change\n";
                    system(("sudo rm " + configFilename + ".new").
                       c_str());
                }

                cout << "= = = = = = = = = = = = = = = = = = = = =
                   = = = = = = = = = = = = = = = = = = = = = = = =
                   = = = = = = = = = = = = = = = = = = = = =\n";
            }
        }
    }
    return 0;
}
```

C.1.2 Optimisation algorithm header

Listing C.2 simply shows the optimisation algorithms' header file.

Listing C.2: Optimisation algorithm C++ header

```
//Adapted from <http://rosettacode.org/wiki/Dijkstra's_algorithm>
//Source code retrieved 05OCT15
#ifndef TOPOLOGY_H
#define TOPOLOGY_H
#include <iostream>
#include <vector>
#include <string>
#include <list>
#include <limits>
#include <set>
#include <utility>
#include <algorithm>
#include <iterator>
#include <cmath>
```

```
#include <iomanip>
#include <queue>
#include <sstream>
#include <fstream>
using namespace std;

typedef int vertex_t;
typedef double weight_t;

const weight_t max_weight = numeric_limits<double>::infinity();

struct neighbor {
    vertex_t target;
    weight_t weight;
    neighbor(vertex_t arg_target, weight_t arg_weight)
        : target(arg_target), weight(arg_weight) { }
};

typedef vector<vector<neighbor> > adjacency_list_t;


void DijkstraComputePaths(vertex_t source, const adjacency_list_t &
   adjacency_list, vector<weight_t> &min_distance, vector<vertex_t>
    &previous);

list<vertex_t> DijkstraGetShortestPathTo(vertex_t vertex, const
   vector<vertex_t> &previous);

int topology_select(vector<double> traffic, adjacency_list_t,
   string configFilename, int argc);

vector< vector< list <vertex_t> > > path_calc(adjacency_list_t);

vector<double> calc_link_traffic(vector<double> traffic, vector<
   vector< list <vertex_t> > > path);

vector<double> calc_link_util(vector<double> linkTraffic,
   adjacency_list_t adjacency_list);

void display_traffic(vector<double> linkTraffic, vector<double>
   linkUtil, vector<double> nodeLoad);

#endif
```

C.1.3 Optimisation algorithm

The optimisation algorithm used by the controller's main program is shown in listing C.1 and performs the following functions:

- Use the traffic matrix input to determine the link utilisation
- Apply optimisation heuristics to iteratively place nodes into the standby state until

the link utilisation rises above 70% or a node is cutoff from the network

- Output the optimised topology to the controller's main program

Listing C.3: Optimisation algorithm C++ source code

```
//Adapted from <http://rosettacode.org/wiki/Dijkstra's_algorithm>
//Source code retrieved 05OCT15

#include "topology.h"
using namespace std;

int topology_select(vector<double> traffic, adjacency_list_t
   adjacency_list, string configFilename, int args) {
    unsigned int NUM_NODES = adjacency_list.size();
    unsigned int i,j,k;
    int leastUsed, mostUsed, targetNode, options;
    vector<int> transitUsage(NUM_NODES);
    list<vertex_t>::iterator pos;
    double minUsage = numeric_limits<double>::infinity();
    double minLoad = numeric_limits<double>::infinity();
    int maxUsage = 0;
    int maxLoad = 0;
    bool connected = 0;
    vector< vector<neighbor> > nodeStorage;

    double THRESHOLD = 70;//link utilisation threshold
    double maxUtil;
    vector<double> linkTraffic(NUM_NODES*NUM_NODES);//traffic on
       each link in Mbps
    vector<double> linkUtil(NUM_NODES*NUM_NODES);//traffic on each
       link in % utilisation



    vector< vector< list <vertex_t> > > path(NUM_NODES);//paths
       between each S/D node pair (bidirectional)
    vector<double> nodeLoad(NUM_NODES);//sum of each node's
       connected link's traffic
    vector<bool> active(NUM_NODES);

    stringstream outputString;
    fstream config;


    //Initial path computation with all nodes present/active
    for (i=0;i<NUM_NODES;i++) active[i]=1;

    path = path_calc(adjacency_list);

    //Calculate link usage in Mbps from traffic matrix and paths
    linkTraffic = calc_link_traffic(traffic, path);

    //Calculate node load
    for (i=0;i<linkTraffic.size();i++) nodeLoad[i/NUM_NODES] +=
       linkTraffic[i];

    //Calculate link utilisation percentage
```

```
linkUtil = calc_link_util(linkTraffic, adjacency_list);

//Find max link utilisation percentage
maxUtil = 0;
for (i=0;i<linkUtil.size();i++){
    if (linkUtil[i] > maxUtil)
        maxUtil = linkUtil[i];
}

//Skip optimisation if dynamic mechanism is disabled (defined
   on command line)
if (args==1){
    while (maxUtil < THRESHOLD){
        connected = 0;
        options = 0;
        for (i=0;i<NUM_NODES;i++){
            options += active[i];
        }
        if (options == 0) break;

        //Clear residual node traffic data and calculate node
           load
        for (i=0;i<nodeLoad.size();i++) nodeLoad[i] = 0;
        for (i=0;i<linkTraffic.size();i++) nodeLoad[i/NUM_NODES
           ] += linkTraffic[i];

        //Until all nodes are connected
        while(!connected || maxUtil > THRESHOLD){
            options = 0;
            for (i=0;i<NUM_NODES;i++){
                options += active[i];
            }
            if (options == 0) break;
            //Clear residual data
            for (i=0;i<NUM_NODES;i++) transitUsage[i] = 0;

            //Determine how many times each node is used in a
               shortest path
            for (i=0;i<NUM_NODES;i++){
                for (j=0;j<NUM_NODES;j++){
                    for (pos=path[i][j].begin();pos!=path[i][j
                       ].end();pos++){
                        transitUsage[*pos]++;
                    }
                }
            }

            //find least used node
            minUsage = numeric_limits<double>::infinity();
            minLoad = numeric_limits<double>::infinity();
            leastUsed = 9999;
            for (i=0;i<NUM_NODES;i++){
                if (active[i] && transitUsage[i] <= minUsage){
                    minUsage = transitUsage[i];
                    leastUsed = i;
                }
                else if (active[i] && transitUsage[i] ==
                   minUsage && nodeLoad[i] < minLoad){
```

```
        minLoad = nodeLoad[i];
        leastUsed = i;
    }
}
if (leastUsed == 9999){//no candidates for removal
    break;
}
//Remove the node as a candidate for future removal
active[leastUsed] = 0;
options--;

//Store adjacency information in case node removal
   has to be reversed
nodeStorage.push_back(adjacency_list[leastUsed]);
for (i=0;i<adjacency_list[leastUsed].size();i++){
    nodeStorage.push_back(adjacency_list[
       adjacency_list[leastUsed][i].target]);
}

//find most used adjacency to least used node
maxUsage = 0;
maxLoad = 0;
mostUsed = 9999;
for (i=0;i<adjacency_list[leastUsed].size();i++){
    if (transitUsage[adjacency_list[leastUsed][i].
       target] > maxUsage){
        maxUsage = transitUsage[adjacency_list[
           leastUsed][i].target];
        mostUsed = adjacency_list[leastUsed][i].
           target;
    }
    else if (transitUsage[adjacency_list[leastUsed
       ][i].target] == maxUsage && nodeLoad[
       adjacency_list[leastUsed][i].target] >
       maxLoad){
        maxLoad = nodeLoad[adjacency_list[leastUsed
           ][i].target];
        mostUsed = adjacency_list[leastUsed][i].
           target;
    }

}
cout << "\tRemoving node " << leastUsed+1 << "\t(
   Keep node " << mostUsed+1 << " adjacency)\n";

//Remove all but one adjacency from node to be
   removed
for (i=0;i<adjacency_list[leastUsed].size();i++){
    targetNode = adjacency_list[leastUsed][i].
       target;
    if (targetNode != mostUsed){
        //Remove adjacencies from other nodes
        for (j=0;j<adjacency_list[targetNode].size
           ();j++){
            if (adjacency_list[targetNode][j].
               target == leastUsed){
                for (k=j ; k<adjacency_list[
                   targetNode].size()-1 ;k++){
                    adjacency_list[targetNode][k] =
                        adjacency_list[targetNode][
                       k+1];
                }
                adjacency_list[targetNode].pop_back
                   ();
                j--;
            }
        }
        //Remove adjacencies from node to be
           removed
        for (j=i ; j<adjacency_list[leastUsed].size
           ()-1 ;j++){
            adjacency_list[leastUsed][j] =
               adjacency_list[leastUsed][j+1];
        }
        adjacency_list[leastUsed].pop_back();
        i--;
    }
}

//calculate new paths between nodes
path = path_calc(adjacency_list);

//Calculate link usage in Mbps from traffic matrix
   and paths
linkTraffic = calc_link_traffic(traffic, path);

//Calculate link utilisation percentage
linkUtil = calc_link_util(linkTraffic,
   adjacency_list);

//Find max link utilisation percentage
maxUtil = 0;
for (i=0;i<linkUtil.size();i++){
    if (linkUtil[i] > maxUtil)
        maxUtil = linkUtil[i];
}

//test if each node can reach all other nodes
for(i=0;i<path.size();i++){
    for (j=0;j<path[i].size();j++){
        //if a node is unreachable, the path is
           simply the destination node
        //Therefore, if a node is reachable, the
           start of the path will be the source
           node
        connected = (i == *(path[i][j].begin()));
        if (!connected) break;
    }
    if (!connected) break;
}

//Add node back into topology if one of more nodes
   are unreachable, or if the utilisation is over
   the threshold
if (!connected || maxUtil > THRESHOLD) {
    cout << "\t\tAdding removed node back in";
```

```
                    if(maxUtil > THRESHOLD){
                        if (!connected) cout << " (Utilisation over
                            threshold AND one or more nodes
                           unreachable)\n";
                        else cout << " (Utilisation over threshold)
                           \n";
                    }
                    else cout << " (One or more nodes unreachable)\
                       n";
                    //Add the removed node back into the network
                    adjacency_list[leastUsed] = nodeStorage[0];
                    for (i=0;i<adjacency_list[leastUsed].size();i
                       ++){
                        adjacency_list[adjacency_list[leastUsed][i
                           ].target] = nodeStorage[i+1];
                    }

                    path = path_calc(adjacency_list);
                }
                //Clear the node storage for future use
                while(nodeStorage.size() > 0) nodeStorage.pop_back
                   ();
            }

            //Calculate link usage in Mbps from traffic matrix and
               paths
            linkTraffic = calc_link_traffic(traffic, path);

            //Calculate link utilisation percentage
            linkUtil = calc_link_util(linkTraffic, adjacency_list);

            //Find max link utilisation percentage
            maxUtil = 0;
            for (i=0;i<linkUtil.size();i++){
                if (linkUtil[i] > maxUtil)
                    maxUtil = linkUtil[i];
            }

            //If no candidates remain, exit the loop
            if (options == 0) break;
        }
    }

    //Write topology to config file
    config.open(configFilename.c_str());
    if (config.fail()){
        cout << "Error opening " << configFilename << "\n";
        return 1;
    }
    for (i=0;i<path.size();i++){
        for (j=0;j<path[i].size();j++){
            if (i!=j){
                pos = path[i][j].begin();
                pos++;
                config << *pos+1;
            }
            else config << i+1;
            config << ",";
        }
        if (i<path.size()-1) config << "\n";
    }
    config.close();
    return 0;
}

vector< vector< list <vertex_t> > > path_calc(adjacency_list_t
   adjacency_list){
    unsigned int i,j;
    unsigned int NUM_NODES = adjacency_list.size();
    vector<weight_t> min_distance;//minimum distance between S/D
       node pair (not used)
    vector< vector <vertex_t> > previous(NUM_NODES);//used in path
       calculation
    vector< vector< list <vertex_t> > > path(NUM_NODES);//paths
       between each S/D node pair (bidirectional)

    for (i=0;i<NUM_NODES;i++) DijkstraComputePaths(i,
       adjacency_list, min_distance, previous[i]);

    for (i=0;i<NUM_NODES;i++){
        for (j=0;j<NUM_NODES;j++){
            path[i].push_back(DijkstraGetShortestPathTo(j, previous
               [i]));
        }
    }
    return path;
}

vector<double> calc_link_traffic(vector<double> traffic, vector<
   vector< list <vertex_t> > > path){
    unsigned int i, j;
    unsigned int NUM_NODES = (int)sqrt((double)traffic.size());
    vector<double> linkTraffic(NUM_NODES*NUM_NODES);//traffic on
       each link in Mbps
    list<vertex_t>::iterator pos1, pos2;

    for (i=0;i<NUM_NODES;i++){
        for (j=0;j<NUM_NODES;j++){
            pos1 = path[i][j].begin();
            pos1++;
            pos2 = pos1;
            pos1--;
            while (pos2!=path[i][j].end()){
                linkTraffic[NUM_NODES*(*pos1) + *pos2] += traffic[
                   NUM_NODES*i + j];
                pos1++;
                pos2++;
            }
        }
    }
    return linkTraffic;
}

vector<double> calc_link_util(vector<double> linkTraffic,
   adjacency_list_t adjacency_list){
    unsigned int NUM_NODES = (int)sqrt((double)linkTraffic.size());
    unsigned int i,j;
    vector<double> linkUtil(NUM_NODES*NUM_NODES);

    for (i=0;i<NUM_NODES;i++){
        for (j=0;j<adjacency_list[i].size();j++){
            linkUtil[NUM_NODES*i + adjacency_list[i][j].target] = (
               linkTraffic[NUM_NODES*i + adjacency_list[i][j].
               target] * adjacency_list[i][j].weight) * 10;
        }
    }
    return linkUtil;
}

void DijkstraComputePaths(vertex_t source, const adjacency_list_t &
   adjacency_list, vector<weight_t> &min_distance, vector<vertex_t>
    &previous){
    int n = adjacency_list.size();
    min_distance.clear();
    min_distance.resize(n, max_weight);
    min_distance[source] = 0;
    previous.clear();
    previous.resize(n, -1);
    set<pair<weight_t, vertex_t> > vertex_queue;
    vertex_queue.insert(make_pair(min_distance[source], source));

    while (!vertex_queue.empty()){
        weight_t dist = vertex_queue.begin()->first;
        vertex_t u = vertex_queue.begin()->second;
        vertex_queue.erase(vertex_queue.begin());

        // Visit each edge exiting u
        const vector<neighbor> &neighbors = adjacency_list[u];
        for (vector<neighbor>::const_iterator neighbor_iter =
           neighbors.begin();
             neighbor_iter != neighbors.end();
             neighbor_iter++)
        {
            vertex_t v = neighbor_iter->target;
            weight_t weight = neighbor_iter->weight;
            weight_t distance_through_u = dist + weight;
            if (distance_through_u < min_distance[v]) {
                vertex_queue.erase(make_pair(min_distance[v], v));

                min_distance[v] = distance_through_u;
                previous[v] = u;
                vertex_queue.insert(make_pair(min_distance[v], v));

            }

        }
    }
}

list<vertex_t> DijkstraGetShortestPathTo(
    vertex_t vertex, const vector<vertex_t> &previous)
{
    list<vertex_t> path;
    for ( ; vertex != -1; vertex = previous[vertex])
```

```
        path.push_front(vertex);
    return path;
}
```

C.2 Network nodes program's source code

This program is used on the each of the network nodes to monitor the topology configuration and implement local configuration changes as required. It is a standalone program and does not take any arguments, as shown below for node four:

```
daniel@node4:~$ ./monitor
```

The code listings below show the main program, the initialisation and configuration functions, and the cURL functions. The main program calls the initialisation function, monitors the topology configuration, and calls the topology change implementation function if required. The cURL functions are used to obtain topology information from the controller. Compilation of the program was performed using g++ as follows:

```
daniel@node4:~$ g++ PiMain.cpp raspi_config.cpp libcurl.c -o monitor
    -lcurl
```

C.2.1 Main program

The nodes' main program shown in listing C.4 performs the following functions:

- Calls the initialisation function to ensure the starting configuration is the same for all nodes
- Perpetually checks whether the config file on the controller has been updated using the cURL functions
- If the controller's topology file is newer than the node's, the new config file is pulled from the controller using the cURL functions
- Once the new config file is received, calls the topology implementation function

Listing C.4: Raspberry Pi main program C++ source code

```
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
```

```
#include <unistd.h>
#include <time.h>
#include <string>
#include <sstream>
#include <fstream>
#include <signal.h>
#include <vector>
#include "libcurl.h"
#include "raspi_config.h"
#include <sys/time.h>

using namespace std;


void int_handler(int x){
    cout << "\nRemoving config from OpenVswitch bridge...";
    cout.flush();
    system("sudo ovs-ofctl del-flows br0");
    system("sudo ovs-ofctl add-flow br0 \"table=0, priority=0,
       actions=normal\"");
    system("sudo ovs-ofctl add-flow br0 \"table=0, priority=5,
       actions=drop\"");
    cout << "done\n";
    cout.flush();
    cout << "Removing ./topology.conf ...";
    cout.flush();
    system("sudo rm ./topology.conf");
    cout << "done\n\n";
    cout.flush();
    exit(1);
}


int main(){
    signal(SIGINT,int_handler);
    unsigned int checkWait = 1000 * 1000 * 2.5;//time to wait
       between checks (in us)
    string controllerURL = "ftp://192.168.254.100/";
    string configFilename = "./topology.conf";

    time_t remoteModTime(0);
    time_t localModTime(0);
    struct tm* remoteModtm;
    int status, node;
    ifstream inputFile;
    char tempchar;
    stringstream tempstr;

    //determine node number
    inputFile.open("/etc/hostname");
    if (inputFile.fail()){
        cout << "Error opening /etc/hostname.\n";
        return 1;
    }
    inputFile.seekg(-2,ios::end);
    inputFile.get(tempchar);
    tempstr << tempchar;
    tempstr >> node;
```

```
inputFile.close();

cout << "Current node: " << node << "\n";

//Force NTP synchronisation
system(("sudo service ntp stop && sudo ntpd -gq && sudo service
    ntp start"));

//Start and initialise OpenVswitch
status = OVS_init(node);

if (status){
    cout << "Error while initialising OpenVswitch. Exiting now
       .\n";
    exit(status);
}
cout << "Config update monitor now running.\n";

for(;;){
    cout << "\tGetting remote mod time\n";
    //get controller's file modification time
    remoteModTime = curl_get_info(configFilename.c_str(),
       controllerURL.c_str());
    //if remote newer than local, apply config
    if (remoteModTime > localModTime){
        cout << "\tUpdating local topology config file (" <<
           configFilename << ")\n";

        //Wait until trying again if the get failed
        if(curl_get(configFilename.c_str(),controllerURL.c_str
           ())) {
            cout << "\tCould not get remote topology config
               file.\n";
            usleep(checkWait/2);
            continue;
        }

        //Update local modification time
        remoteModtm = localtime(&remoteModTime);
        tempstr.str(string());
        tempstr.clear();
        tempstr << "sudo touch " << configFilename << " -d \""
                << (remoteModtm->tm_year+1900) << "-" << (
                   remoteModtm->tm_mon+1) << "-" << remoteModtm
                   ->tm_mday << " "
                << remoteModtm->tm_hour << ":" << remoteModtm->
                   tm_min << ":" << remoteModtm->tm_sec << "\""
                   ;
        system(tempstr.str().c_str());
        localModTime = remoteModTime;

        //Implement topology change
        cout << "\tImplementing topology change\n";
        raspi_config(configFilename, node);
    }
    else {
        //wait for next config check
        cout << "No config change. Sleeping for " << checkWait
```

```
                /1000/1000 << " seconds\n";
            usleep(checkWait);
        }
    }
    return 0;
}
```

C.2.2 Initialisation and topology implementation header

Listing C.5 simply shows the initialisation and topology implementation algorithms' header file.

Listing C.5: Raspberry Pi initialisation and configuration C++ header

```
#ifndef RASPI_CONFIG_H
#define RASPI_CONFIG_H
#include <string>
#include <iostream>
#include <fstream>
#include <sstream>
#include <stdio.h>
#include <stdlib.h>
#include <vector>
#include <sys/types.h>
#include <sys/stat.h>
#include <ifaddrs.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <ctime>
#include <sys/time.h>

using namespace std;

int raspi_config(string configFilename, int node);
string get_lxc_mac(int node);
int OVS_start();
int OVS_init(int node);
int LXC_start(int node);
#endif
```

C.2.3 Initialisation and topology implementation

The initialisation and topology implementation algorithms used by the nodes' main program is shown in listing C.6 and performs the following functions:

- Check if Open vSwitch is running and start if necessary
- Prepare Open vSwitch by configuring it with baseline flows that are required for all topologies
- Configure the topology-specific flows based on the current node and the selected topology

Listing C.6: Raspberry Pi initialisation and configuration C++ source code

```
#include "raspi_config.h"
using namespace std;

int raspi_config(string configFilename, int node){
    int NUM_NODES = 8;
    short activeNodes;
    int i, j, port;
    int status = 0;
    stringstream tempstr;
    ifstream inputFile;
    string line;
    vector<int> nextHops(NUM_NODES);
    vector<int> interfaces(4);
    vector<int> outputPorts(NUM_NODES);

    struct ifaddrs* ifAddrStruct = NULL;
    struct ifaddrs* ifa = NULL;
    void * tmpAddrPtr = NULL;

    bool flag;
    timeval t0, t1;
    struct tm* now;
    struct tm* change;

    //synchronise topology change using system clock
    //get local time
    gettimeofday(&t0, NULL);
    now = localtime(&t0.tv_sec);
    change = localtime(&t0.tv_sec);
    //Output current time
    cout << "\t\tCurrent time: ";
    if (now->tm_hour < 10) cout << "0";
    cout << now->tm_hour << ":";
    if (now->tm_min < 10) cout << "0";
    cout << now->tm_min << ":";
    if (now->tm_sec < 10) cout << "0";
    cout << now->tm_sec << ":";
    if (t0.tv_usec/1000 < 1000) cout << "0";
    if (t0.tv_usec/1000 < 100) cout << "0";
    if (t0.tv_usec/1000 < 10) cout << "0";
    cout << t0.tv_usec/1000 << "\n";

    //determine test start time - closest 5s interval
    if (change->tm_sec < 5)
        change->tm_sec = 10;
    else if (change->tm_sec >= 5 && change->tm_sec < 10)
        change->tm_sec = 15;
```

```
else if (change->tm_sec >= 10 && change->tm_sec < 15)
    change->tm_sec = 20;
else if (change->tm_sec >= 15 && change->tm_sec < 20)
    change->tm_sec = 25;
else if (change->tm_sec >= 20 && change->tm_sec < 25)
    change->tm_sec = 30;
else if (change->tm_sec >= 25 && change->tm_sec < 30)
    change->tm_sec = 35;
else if (change->tm_sec >= 30 && change->tm_sec < 35)
    change->tm_sec = 40;
else if (change->tm_sec >= 35 && change->tm_sec < 40)
    change->tm_sec = 45;
else if (change->tm_sec >= 40 && change->tm_sec < 45)
    change->tm_sec = 50;
else if (change->tm_sec >= 45 && change->tm_sec < 50)
    change->tm_sec = 55;
else if (change->tm_sec >= 50 && change->tm_sec < 55){
    change->tm_sec = 0;
    change->tm_min++;}
else if (change->tm_sec >= 55){
    change->tm_sec = 5;
    change->tm_min++;}

if (change->tm_min >= 60){
    change->tm_hour++;
    change->tm_min -= 60;
}
if (change->tm_hour >= 24)
    change->tm_hour -= 24;
t1.tv_sec = mktime(change);
t1.tv_usec = 0;

//Output change time
cout << "\t\tChange time: ";
if (change->tm_hour < 10) cout << "0";
cout << change->tm_hour << ":";
if (change->tm_min < 10) cout << "0";
cout << change->tm_min << ":";
if (change->tm_sec < 10) cout << "0";
cout << change->tm_sec << ":";
if (t1.tv_usec/1000 < 1000) cout << "0";
if (t1.tv_usec/1000 < 100) cout << "0";
if (t1.tv_usec/1000 < 10) cout << "0";
cout << t1.tv_usec/1000 << "\n";

//Open config file
inputFile.open(configFilename.c_str());
if (inputFile.fail()){
    cout << "Error opening " << configFilename << "\n";
    return 1;;
}
//discard lines of the config file until the current node's
   entry is reached
for (i=0;i<node-1;i++) getline(inputFile,line);
//get next hop for each destination
for (i=0;i<NUM_NODES;i++){
    getline(inputFile,line,',');
    nextHops[i] = atof(line.c_str());
```

```
}
inputFile.close();

//Get IP addresses of each interface
getifaddrs(&ifAddrStruct);
for (ifa = ifAddrStruct; ifa != NULL; ifa = ifa->ifa_next) {
    // check it is IP4 and an ethernet port
    if (ifa->ifa_addr->sa_family == AF_INET && ifa->ifa_name[0]
        == 'e') {
        tmpAddrPtr = &((struct sockaddr_in *) ifa->ifa_addr)->
           sin_addr;
        char addressBuffer[INET_ADDRSTRLEN];
        inet_ntop(AF_INET, tmpAddrPtr, addressBuffer,
           INET_ADDRSTRLEN);
        if (addressBuffer[8] == addressBuffer[11])
            interfaces[atoi(&ifa->ifa_name[3])] = atoi(&
               addressBuffer[8])%10;
        if (addressBuffer[9] == addressBuffer[11])
            interfaces[atoi(&ifa->ifa_name[3])] = atoi(&
               addressBuffer[8])/10;
    }
}
if (ifAddrStruct != NULL) freeifaddrs(ifAddrStruct);

//convert next hop node number to interface number using IP of
   interfaces
for (i=0;i<nextHops.size();i++){
    if (nextHops[i]==node){
        outputPorts[i] = 1;
        continue;
    }
    for (j=0;j<interfaces.size();j++){
        if (interfaces[j]==nextHops[i]) outputPorts[i] = j+2;
    }
}

//wait until local time == change time
cout << "\t\tWaiting for topology change time\n\t\t" << ((t1.
   tv_sec - t0.tv_sec) + (t1.tv_usec - t0.tv_usec)/1000000) <<
   " seconds until start";
while((t1.tv_sec - t0.tv_sec)*1000 + (t1.tv_usec - t0.tv_usec)
   /1000 > 0){
    gettimeofday(&t0, NULL);
    if (int((t1.tv_sec - t0.tv_sec) + (t1.tv_usec - t0.tv_usec)
       /1000000) % 30 == 0){
        if (flag){
            cout << "\n\t\t" << (t1.tv_sec - t0.tv_sec) + (t1.
               tv_usec - t0.tv_usec)/1000000 << " seconds until
                start";
            flush(cout);
            flag = 0;
        }
    }
    else if (int((t1.tv_sec - t0.tv_sec) + (t1.tv_usec - t0.
       tv_usec)/1000000) % 3 == 0){
        if (flag){
            cout << ".";
            flush(cout);
```

```
                flag = 0;
            }
        }
        else flag =1;
    }
    cout << "\n";

    //use next hop OVS port numbers to update OVS flows
    for (i=0;i<NUM_NODES;i++){
        if(i!=node-1){
            tempstr.str(string());//clear string stream for use
            tempstr.clear();
            tempstr << "sudo ovs-ofctl mod-flows br0 \"table=" << i
               +1 << ", actions=dec_mpls_ttl," << outputPorts[i] <<
                "\"";
            status = system(tempstr.str().c_str());
        }
    }
    if (status){
        cout << "Error modifying transit traffic's flows\n";
        exit(status);
    }

    return 0;
}




int OVS_init(int node){
    int status = OVS_start();//Start OpenVswitch
    if (status){
        cout << "Error while starting OpenVswitch. Aborting
           initialisation.\n";
        exit(status);
    }
    int i,j;
    char tempchar;
    stringstream tempstr;
    string lxcmac, intmac;
    ifstream inputFile;
    string intFilename = "/sys/class/net/veth0/address";

    //Start the host (runs in an LXC container)
    LXC_start(node);

    //get LXC conatiner's MAC address
    lxcmac = get_lxc_mac(node);

    //get interface to LXC container's MAC address
    inputFile.open(intFilename.c_str());
    if (inputFile.fail()){
        cout << "Error opening " << intFilename << ".\n";
        return 1;
    }
    getline(inputFile,intmac);
```

```
cout << "Initialising OpenVswitch configuration:\n\tOutput
   handling flows...";
cout.flush();
//Output handling tables - Initialised as normal operation,
   modified later using config file
for (i=1;i<=8;i++){
    tempstr.str(string());//clear string stream for use
    tempstr.clear();
    tempstr << "sudo ovs-ofctl add-flow br0 \"table=" << i << "
       actions=normal\"";
    status = system(tempstr.str().c_str());
    if (status){
        cout << "Error adding output tables\n";
        return status;
    }
}
cout << "done.\n\tTransit and terminating traffic flows...";
cout.flush();
//Transit and terminating traffic
for (i=1;i<=8;i++){
    for (j=1;j<=8;j++){
        if (i != j){
            tempstr.str(string());//clear string stream for use
            tempstr.clear();
            tempstr << "sudo ovs-ofctl add-flow br0 \"table=0,
               priority=500, dl_type=0x8847, mpls_label=" << j
               << "0" << i << ", actions=goto_table:" << i << "
               \"";
            status = system(tempstr.str().c_str());
            if (status){
                cout << "Error adding transit/terminating flows
                   \n";
                return status;
            }
        }
    }
}
//Terminating traffic output flow (won't change when topology
   changes)
//In a real network, this wouldn't work, as it would direct all
    traffic to a single host (same MAC).
tempstr.str(string());//clear string stream for use
tempstr.clear();
tempstr << "sudo ovs-ofctl mod-flows br0 \"table=" << node << "
   , actions=pop_mpls:0x0800,mod_dl_src:" << intmac << ",
   mod_dl_dst:" << lxcmac << ",1\"";
status = system(tempstr.str().c_str());
if (status){
    cout << "Error modifying terminating traffic processing
       flow\n";
    exit(status);
}


cout << "done.\n\tOriginating traffic flows...";
cout.flush();
//Originating traffic
```

```
    for (i=1;i<=8;i++){
        if (i != node){
            tempstr.str(string());//clear string stream for use
            tempstr.clear();
            tempstr << "sudo ovs-ofctl add-flow br0 \"table=0,
               priority=1000, in_port=1, dl_type=0x0800, nw_dst
               =192.168." << i << "0.0/24, actions=push_mpls:0x8847
               , set_mpls_label:" << node << "0" << i << ",
               goto_table:" << i <<"\"";
            status = system(tempstr.str().c_str());
            if (status){
                cout << "Error adding originating flows\n";
                return status;
            }
        }
    }
    //Firewall to prevent double-handling by both OVS and Quagga
    status = system("sudo ovs-ofctl add-flow br0 \"table=0,
       priority=5, actions=drop\"");
    if (status){
        cout << "Error adding firewall flow\n";
        return status;
    }
    cout << "done.\n";
    cout.flush();

    return 0;
}


int OVS_start(){
    //Check if OVS is running, start it if required
    cout << "Checking if OpenVswitch is running...";
    cout.flush();
    struct stat buffer;
    int status = 0;
    string ovsdbFilepath = "/usr/local/var/run/openvswitch/ovsdb-
       server.pid";
    string vswitchdFilepath = "/usr/local/var/run/openvswitch/ovs-
       vswtichd.pid";
    //check if ovsdb-server.pid AND ovs-vswtichd.pid exist
    if(stat(ovsdbFilepath.c_str(), &buffer) && stat(
       vswitchdFilepath.c_str(), &buffer)){
        cout << "no.\nStarting OpenVswitch...";
        cout.flush();
        status = system("sudo ovsdb-server --remote=punix:/usr/
           local/var/run/openvswitch/db.sock --remote=db:
           Open_vSwitch,Open_vSwitch,manager_options --pidfile --
           detach");
        status = status + system("sudo ovs-vsctl --no-wait init");
        status = status + system("sudo ovs-vswitchd --pidfile --
           detach");
        if (status){
            cout << "\nError starting OpenVswitch\n";
            return status;
        }
        cout << "done.\n";
    }
    else
        cout << "yes.\n";
    return 0;
}

int LXC_start(int node){
    stringstream tempstr;
    int status = 0;

    cout << "Checking if LXC container \"host" << node << "\" is
       running...";

    tempstr.str(string());//clear string stream for use
    tempstr.clear();
    tempstr << "sudo LD_LIBRARY_PATH=/usr/local/lib lxc-info -n
       host" << node << " | grep STOPPED";

    if (!system(tempstr.str().c_str())){//system returns '0' if
       host is stopped
        cout << "no.\nStarting LXC container \"host" << node << "
           \"...";

        tempstr.str(string());//clear string stream for use
        tempstr.clear();
        tempstr << "sudo LD_LIBRARY_PATH=/usr/local/lib lxc-start -
           n host" << node;
        status = system(tempstr.str().c_str());

        if (status){
            cout << "failed :(\n";
            return status;
        }

        cout << "done.\n";
    }
    else
        cout << "yes.\n";
    return 0;
}

string get_lxc_mac(int node){
    ifstream inputFile;
    string arpFilename = "/proc/net/arp";
    stringstream tempstr;
    string line, dump, lxcmac;
    vector<string> ipaddr(4);
    int i;

    inputFile.open(arpFilename.c_str());
    if (inputFile.fail()){
        cout << "Error opening " << arpFilename << ".\n";
        return "0";
    }
    tempstr.str(string());
    tempstr << "ping 192.168." << node << "0.100 -w 1 >/dev/null";
       //ping lxc container to ensure there is an entry in the ARP
       table
    system(tempstr.str().c_str());
    while (!inputFile.eof()){
        //clear string stream
        tempstr.str(string());
        tempstr.clear();
        //Get line of file for processing
        getline(inputFile,line);
        tempstr << line;
        //store IP field for processing
        tempstr >> ipaddr[0];
        //store remainder of line to free string stream
        getline(tempstr,line);
        tempstr.str(string());
        tempstr.clear();
        //divide IP address into octets
        tempstr << ipaddr[0];
        for(i=0;i<ipaddr.size();i++){
            getline(tempstr,ipaddr[i],'.');
        }
        //check if this table entry is for the LXC container
        if (192==atof(ipaddr[0].c_str()) && 168==atof(ipaddr[1].
           c_str()) && (node*10)==atof(ipaddr[2].c_str()) && 100==
           atof(ipaddr[3].c_str())){
            //retrieve the remainder of the line
            tempstr.str(string());
            tempstr.clear();
            tempstr << line;

            tempstr >> dump;//discard 'HW type' field
            tempstr >> dump;//discard 'flags' field
            tempstr >> lxcmac;//store 'HW address' field for LXC
               container
            break;
        }
    }
    if (lxcmac==""){
        cout << "MAC address of LXC container not found :(\n";
        return "0";
    }
    inputFile.close();

    return lxcmac;
}
```

C.2.4 cURL implementation header

Listing C.7 simply shows the cURL implementation algorithms' header file.

Listing C.7: cURL C header

```
#ifndef LIBCURL_H
#define LIBCURL_H
#include <stdio.h>
```

```
#include <curl/curl.h>
#include <string>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>
#include <unistd.h>
#include <cstring>
#include <stdlib.h>
#include <stdarg.h>
#include <time.h>

char* concat(int, ...);
int curl_get(const char*,const char*);
time_t curl_get_info(const char*,const char*);
#endif
```

C.2.5 cURL implementation

The cURL functions shown in listing C.8 allow the nodes to retrieve topology information from the controller, and performs the following functions:

- Provides an interface to get files from an FTP server
- Provides an interface to get the modification time of a file on an FTP server

Listing C.8: cURL C source code

```
#include "libcurl.h"

char* concat(int count, ...){
    va_list ap;
    int i;

    // Find required length to store merged string
    int len = 1; // room for NULL
    va_start(ap, count);
    for(i=0 ; i<count ; i++)
        len += strlen(va_arg(ap, char*));
    va_end(ap);

    //Allocate memory to concat strings
    char *merged = (char*)calloc(sizeof(char),len);
    int null_pos = 0;

    //concatenate strings
    va_start(ap, count);
    for(i=0 ; i<count ; i++)
    {
        char *s = va_arg(ap, char*);
        strcpy(merged+null_pos, s);
        null_pos += strlen(s);
    }
    va_end(ap);

    return merged;
}



struct FtpFile {
    const char *filename;
    FILE *stream;
};

static size_t my_fwrite(void *buffer, size_t size, size_t nmemb,
   void *stream){
    struct FtpFile *out=(struct FtpFile *)stream;
    if(out && !out->stream) {
    // open file for writing
    out->stream=fopen(out->filename, "wb");
    if(!out->stream)
        return -1; // failure, can't open file to write
    }
    return fwrite(buffer, size, nmemb, out->stream);
}

int curl_get(const char* infilename,const char* inURL){
    CURL *curl;
    CURLcode res;
    struct FtpFile ftpfile={
        infilename, // name to store the file as if successful
        NULL
    };
    char* fullURL = concat(2,inURL,infilename);

    curl_global_init(CURL_GLOBAL_DEFAULT);
    curl = curl_easy_init();
    if(curl) {
        curl_easy_setopt(curl, CURLOPT_URL,fullURL);
        curl_easy_setopt(curl, CURLOPT_USERPWD, "daniel:password");
        curl_easy_setopt(curl, CURLOPT_WRITEFUNCTION, my_fwrite);//
           Define our callback to get called when there's data to
           be written
        curl_easy_setopt(curl, CURLOPT_WRITEDATA, &ftpfile);// Set
           a pointer to our struct to pass to the callback

        //execute defined cURL
        res = curl_easy_perform(curl);

        //Error handling
        if(CURLE_OK != res) fprintf(stderr, "cURL get: Error %s\n",
            curl_easy_strerror(res));
    }

    //Cleanup
    if(ftpfile.stream) fclose(ftpfile.stream);
    curl_easy_cleanup(curl);
    curl_global_cleanup();
    free(fullURL);
    return (int)res;
}



static size_t throw_away(void *ptr, size_t size, size_t nmemb, void
    *data){
    (void)ptr;
    (void)data;
    return (size_t)(size * nmemb);
}

time_t curl_get_info(const char* infilename,const char* inURL){
    CURL *curl;
    CURLcode res;
    long filetime = -1;
    time_t file_time = (time_t)0;
    char* fullURL = concat(2,inURL,infilename);
    const char *filename = strrchr(fullURL, '/') + 1;

    curl_global_init(CURL_GLOBAL_DEFAULT);
    curl = curl_easy_init();

    if(curl) {
        curl_easy_setopt(curl, CURLOPT_URL, fullURL);
        curl_easy_setopt(curl, CURLOPT_USERPWD, "daniel:password");
        curl_easy_setopt(curl, CURLOPT_NOBODY, 1L);//Don't download
            the file data
        curl_easy_setopt(curl, CURLOPT_FILETIME, 1L); //Ask for
           filetime
        curl_easy_setopt(curl, CURLOPT_HEADERFUNCTION, throw_away);
            // No header output

        //Suppress standard output (otherwise dumps file info)
        int normal, bit_bucket;
        fflush(stdout);
        normal = dup(1);
        bit_bucket = open("/dev/null", O_WRONLY);
        dup2(bit_bucket, 1);
        close(bit_bucket);
        //execute defined cURL
        res = curl_easy_perform(curl);
        //restore standard output
        fflush(stdout);
        dup2(normal, 1);
        close(normal);


        if(CURLE_OK == res) {
            //file modificaiton time processing
            res = curl_easy_getinfo(curl, CURLINFO_FILETIME, &
               filetime);
            if((CURLE_OK == res) && (filetime >= 0)) {
                file_time = (time_t)filetime;
            }
        }
        //Error handling
        else fprintf(stderr, "cURL get info: Error %s\n",
            curl_easy_strerror(res));
    }

    //cleanup
    curl_easy_cleanup(curl);
    free(fullURL);
    curl_global_cleanup();

    //Return file modification time
    return file_time;
}
```

C.3 System testing program's source code

While not directly related to the operation of the dynamic topology mechanism, the two programs below provide a function that is vital to the project: testing of the system and collection of testing data. One program runs on each of the eight virtual hosts connected to the nodes, and synchronises traffic generation for the test scenarios described in chapter 3. The second program runs on the controller, and collates the 336 individual files into a single test report.

C.3.1 Host traffic generation

The program shown in listing C.9 is used to control the traffic on the network during testing, and performs the following functions:

- Determine test start time and synchronise with other hosts
- Generate network traffic as per the testing requirements
- Capture network performance statistics

The program was compiled using g++ as follows:

```
daniel@node4:~$ g++ host.cpp -o host
```

The resultant standalone program takes no command line input or arguments, as shown below for host four:

```
daniel@host4:~# ./host
```

Listing C.9: Traffic generation C++ source code

```
#include <iostream>
#include <ctime>
#include <string>
#include <sstream>
#include <fstream>
#include <vector>
#include <stdlib.h>
#include <unistd.h>
#include <sys/time.h>

using namespace std;

int main(){
    timeval t0, t1;
    struct tm* now;
    struct tm* start;
    bool flag = 1;
    ifstream inputFile;
    stringstream nodestr, tempstr;
    vector<string> trafficFile(3);
    vector< vector<double> > traffic(3);
    string line, trafficstr;
    string controllerURL = "ftp://192.168.254.100/";
    int node,i,j;
    int interval = 10*60;//testing interval in seconds
    char tempchar;

    //Traffic generation matrix filenames
    trafficFile[0] = "./1.matrix";
    trafficFile[1] = "./2.matrix";
    trafficFile[2] = "./3.matrix";

    //determine node number
    inputFile.open("/etc/hostname");
    if (inputFile.fail()){
        cout << "Error opening /etc/hostname.\n";
        return 1;
    }
    inputFile.seekg(-1,ios::end);
    inputFile.get(tempchar);
    nodestr << tempchar;
    nodestr >> node;
    inputFile.close();

    //get this node's traffic from matrix files
    for (i=0;i<3;i++){
        inputFile.open((trafficFile[i]).c_str());
        for (j=0;j<8;j++){
            getline(inputFile,line);
            if(j==node-1) break;
        }
        tempstr.str(string());//clear string stream for use
        tempstr.clear();
        tempstr << line;
        for (j=0;j<8;j++){
            getline(tempstr,trafficstr,',');
            traffic[i].push_back(atof(trafficstr.c_str()));
        }
        inputFile.close();
    }


    //get local time
    gettimeofday(&t0, NULL);
    now = localtime(&t0.tv_sec);
    start = localtime(&t0.tv_sec);

    //Output current time
    cout << "Current time: ";
    if (now->tm_hour < 10) cout << "0";
    cout << now->tm_hour << ":";
    if (now->tm_min < 10) cout << "0";
    cout << now->tm_min << ":";
    if (now->tm_sec < 10) cout << "0";
    cout << now->tm_sec << ":";
    if (t0.tv_usec/1000 < 1000) cout << "0";
    if (t0.tv_usec/1000 < 100) cout << "0";
    if (t0.tv_usec/1000 < 10) cout << "0";
    cout << t0.tv_usec/1000 << "\n";

    //determine test start time - closest 30s interval
    if (start->tm_sec < 15)
        start->tm_sec = 30;
    else if (start->tm_sec >= 15 && start->tm_sec < 45){
        start->tm_sec = 0;
        start->tm_min++;}
    else if (start->tm_sec >= 45){
        start->tm_sec = 0;
        start->tm_min++;}


    if (start->tm_min >= 60){
        start->tm_hour++;
        start->tm_min -= 60;
    }
    if (start->tm_hour >= 24)
        start->tm_hour -= 24;
    t1.tv_sec = mktime(start);
    t1.tv_usec = 0;

    //Output start time
    cout << "Start time: ";
    if (start->tm_hour < 10) cout << "0";
    cout << start->tm_hour << ":";
    if (start->tm_min < 10) cout << "0";
    cout << start->tm_min << ":";
    if (start->tm_sec < 10) cout << "0";
    cout << start->tm_sec << ".";
    if (t1.tv_usec/1000 < 100) cout << "0";
    if (t1.tv_usec/1000 < 10) cout << "0";
    cout << t1.tv_usec/1000 << "\n";

    //wait until local time == start time
    cout << "Waiting for test start time\n" << ((t1.tv_sec - t0.
       tv_sec) + (t1.tv_usec - t0.tv_usec)/1000000) << " seconds
```

```
   until start";
while((t1.tv_sec - t0.tv_sec)*1000 + (t1.tv_usec - t0.tv_usec)
   /1000 > 0){
    gettimeofday(&t0, NULL);
    if (int((t1.tv_sec - t0.tv_sec) + (t1.tv_usec - t0.tv_usec)
       /1000000) % 30 == 0){
        if (flag){
            cout << "\n" << (t1.tv_sec - t0.tv_sec) + (t1.
               tv_usec - t0.tv_usec)/1000000 << " seconds until
                start";
            flush(cout);
            flag = 0;
        }
    }
    else if (int((t1.tv_sec - t0.tv_sec) + (t1.tv_usec - t0.
       tv_usec)/1000000) % 3 == 0){
        if (flag){
            cout << ".";
            flush(cout);
            flag = 0;
        }
    }
    else flag =1;
}
cout << "\n";

//start iperf server for traffic reception
tempstr.str(string());//clear string stream for use
tempstr << "iperf -s -p 5000" << (node-1) << " -B 192.168." <<
   node << "0.100 -u &";
system(tempstr.str().c_str());

tempstr.str(string());//clear string stream for use
tempstr << "iperf -s -p 5001" << (node-1) << " -B 192.168." <<
   node << "0.100 -u &";
system(tempstr.str().c_str());

tempstr.str(string());//clear string stream for use
tempstr << "iperf -s -p 5002" << (node-1) << " -B 192.168." <<
   node << "0.100 -u &";
system(tempstr.str().c_str());

for (i=0;i<traffic.size();i++){
    for (j=0;j<traffic[i].size();j++){
        if (j!=(node-1)){
            //start iperf clients for traffic generation
            tempstr.str(string());//clear string stream for use
            tempstr.clear();
            tempstr << "iperf -c 192.168." << (j+1)*10 << ".100
                -p 500" << i << j << " -B 192.168." << node <<
               "0.100 -t " << interval*(3-i) << " -b " <<
               traffic[i][j] << "M -u -y c >./" << node << "-"
               << j+1 << "_" << i+1 << ".iperf &";
            system(tempstr.str().c_str());

            //start ping
            tempstr.str(string());//clear string stream for use
            tempstr.clear();
```

```
                tempstr << "ping 192.168." << (j+1)*10 << ".100 -w
                   " << interval << " -q >./" << node << "-" << j+1
                    << "_" << i+1 << ".ping &";
                system(tempstr.str().c_str());
            }
        }
        //wait until test finishes
        usleep(1000*1000*interval);
    }
    cout << "\n==============================================\n";
    cout << "TEST FINISHED";
    cout << "===============================================\n";
    return 0;
}
```

C.3.2 Controller traffic statistic collation

The program shown in listing C.10 is used to collate the data from the host traffic generation program. The program was compiled using g++ as follows:

```
daniel@controller:~$ g++ measure.cpp -o measure
```

The resultant standalone program takes no command line input or arguments, but relies on the host data files being present in the local directory. An example execution is shown below:

```
daniel@controller:~$ ./measure
```

Listing C.10: Test data processing C++ source code

```
#include <iostream>
#include <string>
#include <sstream>
#include <fstream>
#include <vector>
#include <stdlib.h>
#include <iomanip>
using namespace std;

int main(){
    int NUM_NODES = 8;
    int FIELDS = 14;
    int JITTER = 9;
    int LOSS = 12;
    int ORDER = 13;
    int i, j, k, m;
    ifstream inputFile;
    stringstream tempstr;
    string line, temp, dump;
```

```
vector<double> delay(NUM_NODES*NUM_NODES);
vector<double> jitter(NUM_NODES*NUM_NODES);
vector<double> loss(NUM_NODES*NUM_NODES);
vector<double> order(NUM_NODES*NUM_NODES);

vector< vector<double> > delayFull, jitterFull, lossFull,
   orderFull;

for (i=0;i<3;i++){
    for (j=0;j<NUM_NODES;j++){
        for (k=0;k<NUM_NODES;k++){
            if (j!=k){
                //Get statistics from iperf files
                tempstr.str(string());
                tempstr.clear();
                tempstr << "./" << (j+1) << "-" << (k+1) << "_"
                    << (i+1) << ".iperf";
                inputFile.open(tempstr.str().c_str());
                if (inputFile.fail()){
                    cout << "Failed to open file " << tempstr.
                       str() << "\n\n";
                    return 1;
                }

                tempstr.str(string());
                tempstr.clear();
                while(getline(inputFile,line)){temp = line;}
                tempstr << temp;
                for (m=0;m<FIELDS;m++){
                    getline(tempstr,temp,',');
                    if (m==JITTER) jitter[NUM_NODES*j + k] =
                       atof(temp.c_str());
                    if (m==LOSS) loss[NUM_NODES*j + k] = atof(
                       temp.c_str());
                    if (m==ORDER) order[NUM_NODES*j + k] = atof
                       (temp.c_str());
                }
                inputFile.close();

                //Get statistics from ping files
                tempstr.str(string());
                tempstr.clear();
                tempstr << "./" << (j+1) << "-" << (k+1) << "_"
                    << (i+1) << ".ping";
                inputFile.open(tempstr.str().c_str());
                if (inputFile.fail()){
                    cout << "Failed to open file " << tempstr.
                       str() << "\n\n";
                    return 1;
                }
                tempstr.str(string());
                tempstr.clear();
                while(getline(inputFile,line)){temp = line;}
                tempstr << temp;

                tempstr >> temp;//discard unwanted data
                tempstr >> temp;//discard unwanted data
                tempstr >> temp;//discard unwanted data
                getline(tempstr,temp,'/');//discard unwanted
                   data

                getline(tempstr,temp,'/');//get the average
                   round trip time
                delay[NUM_NODES*j + k] = atof(temp.c_str());

                inputFile.close();
            }
        }
    }
    delayFull.push_back(delay);
    jitterFull.push_back(jitter);
    lossFull.push_back(loss);
    orderFull.push_back(order);
    for(j=0;j<delay.size();j++){
        delay[j] = 0;
        jitter[j] = 0;
        loss[j] = 0;
        order[j] = 0;
    }
}

for (i=0;i<delayFull.size();i++){
    cout << "= = = = = = = = = = = = = = = = = = = = = = = = =
       = = = = = = = = = = = = = = = = = = = = = = = = = = = =
       = = = = = = = = = = = = =\n";
    cout << " SCENARIO " << i+1 << "\n";
    cout << "= = = = = = = = = = = = = = = = = = = = = = = = =
       = = = = = = = = = = = = = = = = = = = = = = = = = = = =
       = = = = = = = = = = = = =\n";
    cout << "Average round trip time (ms):\n";
    for (j=0;j<delayFull[i].size();j++){
        cout << "\t" << (int)(j/NUM_NODES)+1 << "-" << (j%
           NUM_NODES)+1 << ": " << setw(5) << delayFull[i][j];
        if (j%NUM_NODES==7)
            cout << "\n";
    }
    cout << "\n- - - - - - - - - - - - - - - - - - - - - - - -
       - - - - - - - - - - - - - - - - - - - - - - - - - - - -
       - - - - - - - - - - - - - -\n";
    cout << "Jitter (ms):\n" << setprecision(2);
    for (j=0;j<jitterFull[i].size();j++){
        cout << "\t" << (int)(j/NUM_NODES)+1 << "-" << (j%
           NUM_NODES)+1 << ": " << setw(5) << jitterFull[i][j];
        if (j%NUM_NODES==7)
            cout << "\n";
    }
    cout << "\n- - - - - - - - - - - - - - - - - - - - - - - -
       - - - - - - - - - - - - - - - - - - - - - - - - - - - -
       - - - - - - - - - - - - - -\n";
    cout << "Packet loss (%):\n";
    for (j=0;j<lossFull[i].size();j++){
        cout << "\t" << (int)(j/NUM_NODES)+1 << "-" << (j%
           NUM_NODES)+1 << ": " << setw(5) << lossFull[i][j];
        if (j%NUM_NODES==7)
            cout << "\n";
    }
```

```
        cout << "\n- - - - - - - - - - - - - - - - - - - - - - - -
            - - - - - - - - - - - - - - - - - - - - - - - - - - - -
            - - - - - - - - - - - - - -\n";
        cout << "Packets received out of order (count):\n";
        for (j=0;j<orderFull[i].size();j++){
            cout << "\t" << (int)(j/NUM_NODES)+1 << "-" << (j%
               NUM_NODES)+1 << ": " << setw(5) << orderFull[i][j];
            if (j%NUM_NODES==7)
                cout << "\n";
        }
    }
    return 0;
}
```

Appendix D

Network performance measurements

As detailed in chapter 3, the network performance measurements were captured for three configurations: traffic routed using OSPF with all nodes active in a static topology, traffic routed using MPLS with all nodes active in a static topology, and traffic routed using MPLS with a topology controlled by the dynamic topology mechanism. The demand measurement accuracy for the MPLS and dynamic topology mechanism configurations are also given. Chapter 5 provides an analysis of the results shown below.

The network performance measurements are each comprised of three scenarios, as described in chapter 5. The three values in each of the table's cells correspond to the measurements for these three scenarios, with the top being scenario one and the bottom being scenario three.

D.1 OSPF baseline

The tables below show the delay, jitter, packet loss, and out of order packet measurements taken for the configuration where traffic is routed using OSPF with all nodes active in a static topology.

		Destination node							
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Source node	**1**	- - -	1.202 1.2 1.3	1.922 2 2.1	2.727 2.8 2.9	1.931 2 2.1	1.21 1.2 1.3	1.194 1.2 1.3	1.937 2 2.1
	2	1.169 1.2 1.2	- - -	1.203 1.2 1.3	1.932 2 2.1	1.934 2 2	1.97 2 2.1	1.224 1.3 1.3	2 2.1 2.1
	3	1.915 2 2	1.202 1.2 1.3	- - -	1.205 1.2 1.3	1.17 1.2 1.3	1.904 2 2	1.968 2 2.1	1.219 1.2 1.3
	4	2.718 2.8 2.9	1.953 2 2.1	1.182 1.2 1.3	- - -	1.906 2 2	2.715 2.8 2.9	1.956 2 2.1	1.2 1.2 1.3
	5	1.916 2 2	1.905 2 2	1.179 1.2 1.2	1.913 2 2	- - -	1.184 1.2 1.3	1.925 2 2.1	1.922 2 2.1
	6	1.202 1.2 1.3	1.977 2 2.1	1.887 2 2	2.721 2.8 2.9	1.182 1.2 1.3	- - -	1.235 1.3 1.3	1.935 2 2.1
	7	1.178 1.2 1.2	1.209 1.3 1.3	1.984 2.1 2.1	1.942 2 2.1	1.918 2 2.1	1.201 1.3 1.3	- - -	1.211 1.2 1.3
	8	1.92 2 2.1	2.012 2.1 2.1	1.19 1.2 1.3	1.209 1.2 1.3	1.926 2 2.1	1.95 2 2.1	1.203 1.2 1.3	- - -

Table D.1: OSPF baseline — Delay (ms)

		Destination node							
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Source node	**1**	-	0.24	0.37	0.56	0.77	0.28	0.13	0.18
		-	0.88	0.28	0.81	0.39	0.34	0.24	0.16
		-	0.31	0.46	0.68	0.57	0.46	0.55	0.46
	2	0.24	-	0.12	0.88	0.26	0.16	0.18	0.45
		0.49	-	0.11	0.98	0.26	0.33	0.28	0.68
		0.6	-	0.24	1	0.25	0.88	1.9	0.68
	3	0.36	0.4	-	0.36	0.28	0.3	0.23	0.21
		0.56	1	-	0.92	0.84	0.34	1	0.72
		0.41	0.34	-	1.2	0.96	0.56	0.7	0.25
	4	0.76	0.74	0.36	-	0.33	0.69	0.31	0.14
		1	0.62	0.54	-	1.1	0.4	0.2	0.21
		0.62	0.97	0.76	-	0.33	0.44	0.65	0.45
	5	0.46	0.56	0.079	0.29	-	0.74	0.32	0.73
		0.31	0.57	0.35	0.72	-	0.091	0.19	0.34
		0.9	0.88	0.5	0.48	-	1.7	0.74	0.44
	6	0.35	0.46	0.24	0.43	0.29	-	0.26	0.18
		0.47	0.53	0.26	0.42	0.22	-	0.2	0.3
		0.55	0.52	0.91	0.65	0.86	-	0.49	0.23
	7	0.35	0.56	0.24	0.32	0.52	0.21	-	0.2
		0.79	0.2	0.72	0.84	0.66	0.13	-	0.23
		0.46	0.36	0.98	0.84	1.1	0.32	-	0.72
	8	0.74	0.65	1.3	0.87	0.8	0.24	0.21	-
		0.62	0.51	0.16	0.74	0.96	0.72	0.2	-
		1.5	1	1.6	0.81	1.6	0.59	0.49	-

Table D.2: OSPF baseline — Jitter (ms)

		Destination node							
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Source node		-	0	0	0	0	0	0	0
	1	-	0	0	0	0	0	0	0
		-	0	0	0	0	0	0	0
		0	-	0.033	0	0	0	0	0
	2	0	-	0	0	0	0	0	0
		0	-	0	0	0	0	0	0
		0	0.065	-	0	0	0	0	0.033
	3	0	0	-	0	0	0	0	0
		0	0	-	0	0	0	0	0
		0	0	0.033	-	0	0.065	0	0
	4	0	0.033	0	-	0	0	0	0
		0	0	0	-	0	0	0	0
		0	0	0	0	-	0	0	0
	5	0	0	0.033	0.033	-	0	0	0
		0	0	0	0	-	0	0	0
		0	0	0	0	0	-	0	0.033
	6	0	0	0	0.024	0	-	0	0.024
		0	0	0	0	0	-	0	0.33
		0	0	0	0	0	0	-	0
	7	0	0	0	0	0	0	-	0
		0	0	0	0	0	0	-	0
		0	0	0	0	0	0	0	-
	8	0	0	0	0	0	0	0	-
		0	0	0	0	0	0	0	-

Table D.3: OSPF baseline — Packet loss (%)

		Destination node							
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Source node	**1**	-	0	0	0	0	0	0	0
		-	0	0	0	0	0	0	0
		-	0	0	0	0	0	0	0
	2	0	-	0	0	0	0	0	0
		0	-	0	0	0	0	0	0
		0	-	0	0	0	0	0	0
	3	0	0	-	0	0	0	0	0
		0	0	-	0	0	0	0	0
		0	0	-	0	0	0	0	0
	4	0	0	0	-	0	0	0	0
		0	0	0	-	0	0	0	0
		0	0	0	-	0	0	0	0
	5	0	0	0	0	-	0	0	0
		0	0	0	0	-	0	0	0
		0	0	0	0	-	0	0	0
	6	0	0	0	0	0	-	0	0
		0	0	0	0	0	-	0	0
		0	0	0	0	0	-	0	0
	7	0	0	0	0	0	0	-	0
		0	0	0	0	0	0	-	0
		0	0	0	0	0	0	-	0
	8	0	0	0	0	0	0	0	-
		0	0	0	0	0	0	0	-
		0	0	0	0	0	0	0	-

Table D.4: OSPF baseline — Out of order packets (count)

D.2 MPLS baseline

The tables below show the delay, jitter, packet loss, and out of order packet measurements taken for the configuration where traffic is routed using MPLS with all nodes active in a static topology.

		Destination node							
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Source node	**1**	- - -	1.187 1.2 1.3	1.904 1.9 2	2.707 2.8 2.9	1.877 2 2	1.207 1.2 1.3	1.183 1.2 1.3	1.892 2 2
	2	1.18 1.2 1.2	- - -	1.209 1.2 1.3	1.918 2 2.1	1.912 2 2	1.942 2 2.1	1.239 1.2 1.3	1.957 2 2.1
	3	1.901 2 2	1.206 1.2 1.3	- - -	1.207 1.2 1.3	1.188 1.2 1.3	1.892 2 2	1.949 2 2.1	1.216 1.2 1.3
	4	2.704 2.8 2.9	1.914 2 2	1.206 1.2 1.2	- - -	1.869 1.9 2	2.708 2.8 2.9	1.97 2 2.1	1.207 1.2 1.3
	5	1.898 2 2	1.889 2 2	1.189 1.2 1.2	1.867 2 2	- - -	1.195 1.2 1.2	1.893 2 2	1.888 2 2
	6	1.208 1.2 1.3	1.949 2 2.1	1.899 2 2	2.718 2.8 2.9	1.167 1.2 1.2	- - -	1.214 1.2 1.3	1.927 2 2.1
	7	1.296 1.2 1.3	1.216 1.3 1.3	1.959 2 2.1	1.939 2 2.1	1.902 2 2	1.208 1.2 1.3	- - -	1.212 1.3 1.3
	8	1.935 2 2.1	1.943 2 2.2	1.193 1.2 1.3	1.221 1.3 1.3	1.895 2 2	1.929 2 2.1	1.228 1.3 1.3	- - -

Table D.5: MPLS baseline — Delay (ms)

		Destination node							
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Source node	**1**	-	0.19	0.4	0.51	0.33	0.29	0.16	0.14
		-	0.29	0.2	0.54	0.34	0.46	0.12	0.17
		-	0.79	0.94	0.72	0.57	1.8	0.23	0.81
	2	0.1	-	0.18	0.16	0.28	0.19	0.43	0.21
		1.4	-	0.34	0.93	0.19	0.43	0.34	0.42
		1.3	-	1.3	0.83	0.69	0.48	1.3	0.4
	3	0.27	0.15	-	1.2	0.37	0.51	0.26	0.47
		0.35	0.85	-	0.17	0.52	0.85	0.38	0.21
		0.52	0.74	-	2.1	1.7	0.51	0.43	0.43
	4	0.87	0.26	0.21	-	0.91	0.22	0.85	0.19
		0.34	0.23	0.31	-	0.34	0.48	0.48	0.21
		0.54	0.72	0.26	-	0.42	0.59	1.6	0.67
	5	0.27	0.44	0.31	0.42	-	0.11	0.16	0.36
		0.24	0.34	0.3	0.31	-	0.36	0.15	0.38
		0.76	0.72	0.22	1.5	-	0.49	0.23	0.49
	6	0.24	0.45	0.19	0.71	0.47	-	0.19	0.33
		0.35	0.19	0.26	0.53	0.15	-	0.36	0.27
		0.41	0.52	0.21	0.34	0.59	-	0.3	0.72
	7	0.19	0.12	0.36	0.93	0.8	0.59	-	0.26
		0.95	0.69	0.25	1	0.26	0.22	-	0.62
		0.49	0.57	0.7	1.3	0.58	0.41	-	0.33
	8	1	0.29	0.28	0.75	0.42	0.73	0.4	-
		0.32	0.94	0.8	0.71	0.26	0.35	0.2	-
		0.51	0.41	0.86	0.51	0.56	0.49	0.61	-

Table D.6: MPLS baseline — Jitter (ms)

		Destination node							
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Source node	**1**	-	0	0	0	0.033	0	0	0
		-	0	0	0	0	0	0.065	0
		-	0	0	0	0	0	0	0.039
	2	0	-	0	0	0	0	0	0
		0	-	0	0	0	0	0	0.065
		0	-	0	0	0	0	0	0
	3	0	0	-	0	0	0	0	0
		0	0	-	0	0	0	0	0
		0	0	-	0	0	0	0	0
	4	0	0	0.033	-	0	0	0	0
		0	0	0	-	0	0	0	0
		0.078	0	0	-	0.039	0	0	0
	5	0	0	0	0	-	0.033	0	0
		0	0	0	0	-	0	0	0
		0	0.078	0	0	-	0.039	0	0
	6	0	0	0	0	0	-	0	0
		0	0	0	0	0	-	0	0
		0	0	0	0.33	0	-	0	0
	7	0	0	0	0	0	0	-	0
		0	0	0	0	0	0	-	0
		0	0	0	0	0	0	-	0
	8	0	0	0	0	0	0	0	-
		0	0	0	0	0	0	0	-
		0	0	0	0	0	0	0	-

Table D.7: MPLS baseline — Packet loss (%)

		Destination node							
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Source node	**1**	-	0	0	0	0	0	0	0
		-	0	0	0	0	0	0	0
		-	0	0	0	0	0	0	0
	2	0	-	0	0	0	0	0	0
		0	-	0	0	0	0	0	0
		0	-	0	0	0	0	0	0
	3	0	0	-	0	0	0	0	0
		0	0	-	0	0	0	0	0
		0	0	-	0	0	0	0	0
	4	0	0	0	-	0	0	0	0
		0	0	0	-	0	0	0	0
		0	0	0	-	0	0	0	0
	5	0	0	0	0	-	0	0	0
		0	0	0	0	-	0	0	0
		0	0	0	0	-	0	0	0
	6	0	0	0	0	0	-	0	0
		0	0	0	0	0	-	0	0
		0	0	0	0	0	-	0	0
	7	0	0	0	0	0	0	-	0
		0	0	0	0	0	0	-	0
		0	0	0	0	0	0	-	0
	8	0	0	0	0	0	0	0	-
		0	0	0	0	0	0	0	-
		0	0	0	0	0	0	0	-

Table D.8: MPLS baseline — Out of order packets (count)

D.3 Dynamic topology mechanism

The tables below show the delay, jitter, packet loss, and out of order packet measurements taken for the configuration where traffic is routed using MPLS with a topology controlled by the dynamic topology mechanism.

		Destination node							
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Source node	**1**	-	6.032	7.747	5.423	10.645	5.068	5.606	11.295
		-	13	10	12	13	7.4	6.3	9
		-	8.4	18	15	22	7.5	7.8	13
	2	5.413	-	7.511	4.987	12.883	6.416	5.507	7.386
		11	-	6.7	8.5	8.8	9.9	12	9.5
		9	-	10	17	15	9.3	8.3	16
	3	9.842	8.506	-	3.973	4.969	7.154	9.213	7.358
		9.4	7.2	-	7.3	6.2	9.3	10	7.6
		20	8.5	-	11	8.7	12	20	14
	4	5.644	4.651	3.206	-	5.395	6.335	4.542	2.869
		12	8.3	7	-	9.5	10	11	6.2
		13	15	10	-	15	19	13	8.4
	5	9.081	11.318	5.114	5.538	-	8.407	9.285	7.807
		13	9.6	7.5	10	-	8.7	11	10
		20	14	13	15	-	7.4	21	15
	6	5.989	6.586	8.111	5.703	12.901	-	5.121	10.424
		7.7	9.9	9.3	12	6.9	-	6.6	11
		7.2	9.7	14	22	7.8	-	8.2	14
	7	5.528	5.078	10.723	6.194	7.622	6.016	-	5.513
		6.1	12	9.8	9.5	11	7.4	-	8.8
		8.1	8.6	15	13	21	8.9	-	13
	8	9.519	7.891	5.673	2.97	6.848	7.275	5.499	-
		9.7	8.8	8.2	6.6	10	11	8.7	-
		14	14	15	8.6	16	16	12	-

Table D.9: Dynamic topology mechanism — Delay (ms)

		Destination node							
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Source node	**1**	-	0.18	0.42	0.31	0.6	0.26	0.43	0.41
		-	0.46	0.22	0.39	0.43	0.29	0.31	0.24
		-	0.37	0.22	0.7	1.3	0.34	0.87	0.56
	2	0.29	-	0.3	0.31	1.7	0.33	0.2	0.28
		0.36	-	0.42	0.44	0.42	1.1	0.24	0.35
		0.65	-	0.2	0.59	0.33	0.27	0.24	0.5
	3	0.3	0.17	-	0.83	0.38	0.36	0.43	0.39
		0.82	0.21	-	0.77	0.34	0.24	0.38	0.35
		0.29	0.2	-	0.53	2.2	0.64	0.79	0.85
	4	0.27	0.69	0.26	-	0.23	0.3	0.23	0.29
		0.43	0.2	0.77	-	0.72	0.23	0.46	0.55
		0.36	1	0.13	-	0.38	0.94	0.75	0.75
	5	0.33	0.17	0.5	0.53	-	0.24	0.28	0.26
		0.36	0.65	0.22	0.55	-	0.6	0.21	0.39
		0.4	0.38	0.36	0.55	-	0.88	0.39	0.51
	6	0.41	0.38	0.43	0.37	0.76	-	0.27	0.27
		0.14	0.31	0.24	0.57	0.41	-	0.34	0.17
		0.33	1.3	0.26	0.6	0.32	-	0.52	0.22
	7	0.31	0.25	0.83	0.51	0.32	0.45	-	0.22
		0.28	0.26	0.35	0.71	0.46	0.58	-	0.45
		0.58	0.47	0.41	0.42	0.4	1.2	-	0.56
	8	0.28	0.68	0.47	0.76	3.1	0.6	0.44	-
		0.39	0.29	1.2	0.57	0.63	0.8	1.1	-
		0.9	0.56	0.81	0.44	0.58	0.65	0.75	-

Table D.10: Dynamic topology mechanism — Jitter (ms)

		Destination node							
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Source node	**1**	-	0	0	0	0	0	0	0
		-	0	0.033	0	0	0	0	0
		-	0	0	0	0	0	0	0
	2	0	-	0	0	0.033	0	0	0
		0	-	0	0	0	0	0	0
		0	-	0	0	0	0	0	0
	3	0	0	-	0	0	0	0	0
		0	0	-	0	0	0	0	0
		0	0	-	0	0	0	0	0
	4	0	0	0	-	0	0	0	0
		0	0	0	-	0	0	0	0
		0	0	0	-	0	0	0	0
	5	0	0	0	0	-	0	0	0
		0	0	0	0	-	0	0	0
		0	0	0	0	-	0.039	0	0
	6	0	0	0	0	0	-	0.033	0
		0	0	0	0	0	-	0	0
		0	0	0	0.17	0	-	0	0
	7	0	0	0.033	0	0	0	-	0
		0	0	0	0	0	0	-	0
		0	0	0	0	0	0	-	0
	8	0	0	0	0	0	0	0	-
		0	0	0	0	0	0	0	-
		0	0	0	0	0	0	0	-

Table D.11: Dynamic topology mechanism — Packet loss (%)

		Destination node							
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Source node	**1**	-	0	0	0	0	0	0	0
		-	0	0	0	0	0	0	0
		-	0	0	0	0	0	0	0
	2	0	-	0	0	0	0	0	0
		0	-	0	0	0	0	0	0
		0	-	0	0	0	0	0	0
	3	0	0	-	0	0	0	0	0
		0	0	-	0	0	0	0	0
		0	0	-	0	0	0	0	0
	4	0	0	0	-	0	0	0	0
		0	0	0	-	0	0	0	0
		0	0	0	-	0	0	0	0
	5	0	0	0	0	-	0	0	0
		0	0	0	0	-	0	0	0
		0	0	0	0	-	0	0	0
	6	0	0	0	0	0	-	0	0
		0	0	0	0	0	-	0	0
		0	0	0	0	0	-	0	0
	7	0	0	0	0	0	0	-	0
		0	0	0	0	0	0	-	0
		0	0	0	0	0	0	-	0
	8	0	0	0	0	0	0	0	-
		0	0	0	0	0	0	0	-
		0	0	0	0	0	0	0	-

Table D.12: Dynamic topology mechanism — Out of order packets (count)

D.4 Traffic demand measurement accuracy

Table D.13 below shows measurements related to the traffic demand measurement accuracy for the MPLS and dynamic topology mechanism configurations. The expected and measured total traffic values are shown, in addition to the average error of the individual link measurements for each time sample.

	Total network demand (Mbps)			Average error (%)	
Time (min)	**Expected**	**MPLS**	**DTM**	**MPLS**	**DTM**
0:00	1.1	0.3	0.5	100	100
0:15	1.1	0.8	1.2	75.1	58.8
0:30	1.1	1.3	1.3	35.9	34.2
0:45	1.1	1.3	1.3	30.9	37.1
1:00	1.1	1.2	1.3	30.5	29.5
1:15	1.1	1.2	1.3	31.6	33.3
1:30	1.1	1.3	1.4	28.9	35.4
1:45	1.1	1.3	1.3	33.3	35.9
2:00	1.1	1.3	1.3	33.5	29.5
2:15	1.1	1.4	1.3	37.4	35.1
2:30	1.1	1.3	1.3	33.5	34.1
2:45	1.1	1.2	1.2	34.9	34.5
3:00	1.1	1.2	1.3	29.3	28
3:15	1.1	1.2	1.3	31.2	33.1
3:30	1.1	1.3	1.3	29.9	35.1
3:45	1.1	1.4	1.4	36.9	36.9
4:00	1.1	1.3	1.4	36.4	40.9
4:15	1.1	1.3	1.3	31.4	38.6
4:30	1.1	1.3	1.3	32.7	33.4
4:45	1.1	1.3	1.3	32.5	28.7
5:00	1.1	1.3	1.3	36.1	32.6
5:15	1.1	1.4	1.2	29.2	33.2
5:30	1.1	1.3	1.3	38.7	26.5
5:45	1.1	1.3	1.4	41.8	32.2

Table D.13: Total measured traffic and measurement error (continued on next page)

	Total network demand (Mbps)			Average error (%)	
Time (min)	Expected	MPLS	DTM	MPLS	DTM
6:00	1.1	1.3	1.3	30.2	35.7
6:15	1.1	1.4	1.3	28.1	32.4
6:30	1.1	1.4	1.3	36.7	32.4
6:45	1.1	1.4	1.3	37.3	35.9
7:00	1.1	1.3	1.4	37.5	34.4
7:15	1.1	1.3	1.3	32.9	32.6
7:30	1.1	1.3	1.3	33.8	34.6
7:45	1.1	1.3	1.3	36.2	32.9
8:00	1.1	1.3	1.3	36.7	30.3
8:15	1.1	1.3	1.4	32.9	34.1
8:30	1.1	1.3	1.3	30.4	35.8
8:45	1.1	1.4	1.3	30.4	32.1
9:00	1.1	1.2	1.3	33.4	34.2
9:15	1.1	1.2	1.3	25.6	35.6
9:30	1.1	1.3	1.3	23.7	31.7
9:45	1.1	1.3	1.4	28.9	34.2
10:00	3.2	1.7	2.5	57.6	55.5
10:15	3.2	2.7	3.4	46.3	23.9
10:30	3.2	3.5	3.4	20.6	20.9
10:45	3.2	3.6	3.6	20.5	19.4
11:00	3.2	3.4	3.6	21.4	19.5
11:15	3.2	3.4	3.4	19.1	22
11:30	3.2	3.4	3.3	19.3	22.7
11:45	3.2	3.3	3.5	18.3	16.4
12:00	3.2	3.3	3.5	18.1	18.8
12:15	3.2	3.4	3.5	19.1	21.5
12:30	3.2	3.4	3.4	21.5	19.4
12:45	3.2	3.1	3.5	20.3	20.6
13:00	3.2	3.2	3.6	19.8	21.5
13:15	3.2	3.3	3.7	16.6	22.7
13:30	3.2	3.4	3.5	17.3	22.2

Table D.13: Total measured traffic and measurement error (continued on next page)

	Total network demand (Mbps)			Average error (%)	
Time (min)	Expected	MPLS	DTM	MPLS	DTM
13:45	3.2	3.4	3.5	16	19.6
14:00	3.2	3.2	3.6	20.2	22
14:15	3.2	3.1	3.5	19.7	21.4
14:30	3.2	3.2	3.5	17.1	20.8
14:45	3.2	3.2	3.4	16.1	22
15:00	3.2	3.2	3.5	18.1	20.6
15:15	3.2	3.4	3.4	15	16.9
15:30	3.2	3.5	3.3	17.3	17.8
15:45	3.2	3.4	3.4	20.4	18.7
16:00	3.2	3.4	3.4	18.7	16.4
16:15	3.2	3.4	3.3	18.9	16.9
16:30	3.2	3.3	3.4	14	19.3
16:45	3.2	3.3	3.6	16.4	19.3
17:00	3.2	3.4	3.7	17.9	21.1
17:15	3.2	3.3	3.5	16.6	24.7
17:30	3.2	3.3	3.4	17.2	21.4
17:45	3.2	3.5	3.5	15.9	19.1
18:00	3.2	3.4	3.6	17.1	18.5
18:15	3.2	3.5	3.5	16.3	24.7
18:30	3.2	3.4	3.4	20.7	23.3
18:45	3.2	3.1	3.6	19.6	19.4
19:00	3.2	3.2	3.6	19.2	21.5
19:15	3.2	3.3	3.6	21.7	20.7
19:30	3.2	3.4	3.5	18.8	20.3
19:45	3.2	3.3	3.5	17.3	18.8
20:00	5.6	3.6	4.8	43	43
20:15	5.6	5.1	6	38.1	19.1
20:30	5.6	6.2	6.2	16.7	15.3
20:45	5.6	6.1	6.2	15.5	18.7
21:00	5.6	5.9	6	15.2	15.7
21:15	5.6	6.1	5.9	12.5	14.7

Table D.13: Total measured traffic and measurement error (continued on next page)

	Total network demand (Mbps)			Average error (%)	
Time (min)	Expected	MPLS	DTM	MPLS	DTM
21:30	5.6	6	5.9	18.5	15.9
21:45	5.6	6	6	19.6	15.8
22:00	5.6	6	5.9	17.4	15.7
22:15	5.6	5.8	6	16.3	14.2
22:30	5.6	5.7	6.2	14.3	14.9
22:45	5.6	5.7	6.2	12.7	18
23:00	5.6	5.9	6	15.7	17.5
23:15	5.6	5.8	6	16.9	16.6
23:30	5.6	5.8	6.1	14.8	15.9
23:45	5.6	6	5.9	13.8	17.3
24:00	5.6	6	5.9	15.3	15.3
24:15	5.6	6.2	6.2	14.4	14.3
24:30	5.6	6.1	6.1	16.4	17.1
24:45	5.6	6	6.1	14.4	13.7
25:00	5.6	6	6.1	14.3	13
25:15	5.6	6	5.9	15.2	16.1
25:30	5.6	5.9	5.9	15.5	12.4
25:45	5.6	5.9	6	14.5	12.9
26:00	5.6	5.9	6	14.8	15.6
26:15	5.6	6	5.9	18.8	17.5
26:30	5.6	6.2	6	13.6	14.1
26:45	5.6	6	6.1	16.1	16.1
27:00	5.6	6	6.1	14.4	15.3
27:15	5.6	6.1	5.9	13.3	15
27:30	5.6	6.1	6	12.1	14.6
27:45	5.6	6.1	6.1	15.5	14.1
28:00	5.6	6	6	14.8	14.4
28:15	5.6	6	6.3	13.7	14.3
28:30	5.6	6.1	6	15.4	15.9
28:45	5.6	6.1	6	16.9	12.1
29:00	5.6	6	6.2	16.1	14.9

Table D.13: Total measured traffic and measurement error (continued on next page)

	Total network demand (Mbps)			Average error (%)	
Time (min)	**Expected**	**MPLS**	**DTM**	**MPLS**	**DTM**
29:15	5.6	6.1	6.3	14.4	16.7
29:30	5.6	5.9	6.3	16.5	15.7
29:45	5.6	5.8	6	13.2	15.5
30:00	5.6	5.8	6	13.4	13.1

Table D.13: Total measured traffic and measurement error