VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wānanga o te Ūpoko o te Ika a Māui

School of Engineering and Computer Science
Te Kura Mātai Pūkaha, Pūrorohiko

PO Box 600
Wellington
New Zealand

Tel: +64 4 463 5341
Fax: +64 4 463 5045
Internet: office@ecs.vuw.ac.nz

Performance Visualizer and Planner for SDN

Jordan Ansell

Supervisors: Prof. Winston Seah, Dr. Bryan Ng, Dr. Stuart Marshall

Submitted in partial fulfilment of the requirements for
Bachelor of Engineering (Hons).

Abstract

Software-Defined Networking (SDN) is recent a networking technology of heavy interest, thus far only applied to data center and research campus environments. Despite meticulous effort in planning and equipment selection prior to deployment, there remain unknowns that can affect the network's performance after equipment has been deployed and is fully operational.

Network administrators and planners would benefit from a tool that is able to monitor the load on various network entities and visualize this in real-time; allows them to make prompt decisions to prevent seemingly small hotspots from becoming major bottlenecks. The tool will also allow a network planner to examine how their network's performance will be affected as the traffic loads and network utilization changes. Performance prediction is based on analytical models of the network configuration using real-time measurements taken from the network devices.

Acknowledgments

I would like thank my supervisors for being patient with me over the year and giving me direction when I need it.

Contents

Figures

Chapter 1

Introduction

The network administrators, those responsible for deploying networks and making architecture decisions would make use of a tool that can bring the tools of mathematical analysis closer to them. Currently there is a gap between the network administrators and the technical ability or practical feasibility of performing such analysis at scale.

Analysis and modelling are the means toward determining what the expected performance and limitations of real world systems are. In terms of networks, through the use of analytical models the limitations of scale and bottlenecks of a particular network can be determined without simulation or physical deployment. Queueing theory is a well established branch of analysis frequently used and well suited to describing networks. Use of such mathematical tools existing networks could also provide insight into how the performance changes as traffic volumes change.

Software Defined Networking (SDN) is a recently emerging network paradigm. SDN increases the flexibility of a network through a programmable, centralized control plane distinct from the network's packet-forwarding data plane [1, 2]. It also encourages innovation by opening the development of the network. Though network monitoring methods exist in traditional networks, a central controller with access to all the traffic information within the network simplifies this process.

This report details an application which equips users with the mathematical tools to perform analysis on a live network, and simple traffic prediction by leveraging the information provided by the global view of an SDN controller.

1.1 Project Description

The aim of this project is to develop an application which makes the analytical approach available to the network administrators of software defined networks. To achieve this, such such an application must process data from an existing SDN network in real-time and make the results available to a network administrator visually. It must predict the effects of traffic and topology changes to the network's performance.

As such, the requirements for the application are that it must:

- Display the state of the network in real-time.
- Allow the user to artificially alter the traffic characteristics of the network.
- Allow the user to artificially alter the network topology.
- Allow alternative models to be used to predict traffic.

The benefit of this application is both making queueing models easier to use and potentially also assisting in the validation of new models. Software defined networking is new and an application such as the one described in this report could also assist in the exploration of how a software defined network behaves. A simple interface for incorporating new equations and models is thus desirable.

1.1.1 Scope of Project

Analysis of network performance can be made on different levels of granularity. Whole network performance, individual connection (called a 'flow' or 'end-to-end' analysis), or at a device level. Taking the mid-way approach, the intended focus of this analysis application is the total traffic entering and leaving a switch. Rather than measuring on a per-flow basis, the total traffic through each port is measured.

Developing new models or finding a good model is not the aim of this project, though the application it creates will benefit from such investigation. This aim of this project is to create a tool that makes using models, performing analysis and exploring new models easier.

1.2 Report Structure

The structure of this report is as follows. Chapter 2 gives an overview of the queueing theory and analysis using it, a description of the application developed, and surveys some of the existing network monitoring tools highlighting what makes this tool different.

Chapter 3 details the design of the application, discussing the technologies used. Chapter 4 describes how the application was implemented and how it is used. Chapter 5 evaluates the application developed, how well it fulfils the requirements above. Chapter 6 then concludes this report, and discusses both the contributions made and potential future work. Appendix A covers the steps taken to obtain the average service rate of a switch using a netFPGA and Pica8 OpenFlow switch and within Mininet.

Chapter 2

Background Survey

This chapter introduces the topics of queueing analysis and software defined networking, surveys what tools currently exist for a network administrator to view network performance and looks briefly at how prediction has been done for network performance in the past.

2.1 Analytical Models and Queueing Theory

Mathematical models are used to describe a system. They do this by abstracting the components of a system and modelling them and their interactions as equations. This enables a system can be quantitatively analysed. If a proposed model is seen as a good fit to a system it can be used to recognise areas where issues could arise – such as the theoretical maximum throughput or the maximum scale of a network.

Queueing theory is a well established branch of mathematics which can be used to analyse networks and is useful in modelling computing systems, with numerous articles and books on the topic. More generally it is used to model any system where jobs wait and are then serviced, exactly as packets are buffered and processed at switches in a network. As seen in figure 2.1 jobs arrive at a rate of λ and are serviced at a rate of μ. Following this, the load of a node then is defined as $\rho = \frac{\lambda}{\mu}$. Relating this back to a network, how busy a switch is can be defined by ρ, and depending on the queueing and servicing behaviour of the node performance values can be calculated. Section 5.4 works through how this can be done in more detail.

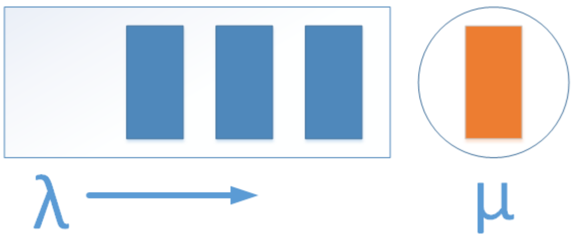


Figure 2.1: A simple queueing node

This report uses Kendall Lee notation in parts to concisely describe the queueing behaviour of a node. For example, a system which has an exponentially distributed arrival rate, a single serving node with exponentially distributed service times, and features no upper limit on the number of jobs in the system is denoted as M/M/1, where the first 'M' is for exponential arrivals, the second for service times, and finally '1' is the number of service nodes. The a larger-form M/M/1/K is also used, where K defines an upper bound on the number of jobs, a K-sized buffer.

Queueing theory allows, for example, the average waiting time of a job (also known as sojourn time), the average length of a queue, and the expected number of jobs to be lost, in the case of a limited capacity system. Where the rate of jobs entering system is greater than the rate at which they can be processed ($\rho > 1$) the service limits of a system are reached and waiting times for new jobs increase dramatically. For a network, higher waiting times introduce delay to traffic and can impact quality of a connection. Packet loss results in interruptions too. Knowing what a network's limits are is useful information for a network administrator, and queueing theory can provide them with these limits.

Other methods of analysis exist, such as numerical or discrete-event based simulations and direct experimentation. But these can be costly in setup, running time and purchasing equipment. Modelling a system with mathematics can be quicker and cheaper than setting up a whole network or running an simulation. It can provide theoretical limits before they are met.

Examples of the use of queueing theory within networking can be seen in the literature. Presented here are two recent examples with a networking focus, one extracting performance information and another using a queueing model to identify and reduce a bottleneck in the system.

Juizhen et al. [3] identify the limitations of using CMP (chip multiprocessors) to parse the signalling messages of voice over IP (VoIP). The system is expressed using a queueing model and the bottlenecks are identified and average delay of signalling messages is calculated.

Zhu et al. [4] use queueing theory model the end-to-end delay of a Multi-Protocol Label Switching (MPLS) network. Through this model they are able to extract performance information and identify a relationship between the management information stored on a router and end-to-end delay, proposing a solution to fix this.

For SDN networks, queueing models of a lone switch and its interactions with a controller have been proposed [5, 6]. The results of experimentation show that the model is valid for the single switch case. However, a single switch is not immediately useful. There have been suggestions for modelling combinations of switches [6, 7] and controllers [8], but these have yet to be validated through real-world experimentation. Other models of SDN networks, using mathematical methods other than queueing theory have also been suggested, but suffer from various limitations [6].

While there is value in the use of analytical models of networks, the scale and dynamic nature of a live network, and the mathematical competency required can pose a boundary for network administrators and their use of analytical models.

2.2 Software Defined Networks

Building on networking concepts explored in the past [9, 10], a software defined network (SDN) brings to life the ideas of control-data plane separation and provides the network domain with a programmable interface.

Traditionally networks are distributed in nature, routing and forwarding decisions are made within every network device. Packets move through the network on a hop-by-hop basis. When they enter a device, their destination is looked up in route tables, a suitable next device is selected and the packets are then passed on.

Unlike the vendor-driven traditional networks, where development is closed and new protocols take a long time to be implemented and released, software defined networks feature an open, programmable interface. Applications can be written which extend the decision making functionality and allow new protocols to be tested and deployed in just one network. By opening up the network, users can define their own network behaviour. SDN is encouraging innovation within an ossifying domain[11].

In an SDN network the controller is a logically central but physically separate device connected to all the switches. It is consulted each time unfamiliar traffic enters the network. A connections are also known as a flow. By being connected to the entire network the controller can easily access information on each switch. However, as a point of control it also becomes a bottleneck as the scale of a network increases.

Applications run on the controller which influence the forwarding decisions made on arriving flows. Acting as an application programming interface (API), the controller can share the network state with such applications. This is a key point of SDN and simplifies the method by which information about the network is obtained.

2.2.1 OpenFlow

OpenFlow is the de-facto SDN standard promoted by the Open Networking Foundation. The OpenFlow standard defines how network switches and controllers communicate and is a basis for how the switch and controller should behave.

In the OpenFlow standard a network switch uses flow tables to store the rules governing how to treat a connection between two hosts. These rules are matched against a subset of the flows within the network and are associated with actions performed on the traffic, such as whether to forward or drop packets [2]. The match-action principle is at the heart of how OpenFlow works.

Flows are identified in a request to the controller by a selection of header fields. These are sent to a controller as a *packet_in* message and based on which a controller will determine the rules and actions to push these back to the network switches.

Each switch uses counters to measure the traffic going through each port and matching table rules. This information is accessible to the controller through the use of statistics requests.

The initial 1.0 release of OpenFlow late in 2009 [12] has a good share of vendor support, and while versions up to 1.5 exist, version 1.3 is the current standard level of support in available switches and controllers. Version 1.3 brings together the major changes and improvement of versions 1.1 and 1.2 which change how some messages are structured and definitions of metrics that the switches must support. Versions 1.4 and 1.5 build upon these changes rather than replace the standard, making 1.3 and excellent point for application development.

Ryu[13] and NOX are popular controller implementations [14] of OpenFlow. OpenVSwitch is a software switch implementation of OpenFlow.

2.3 Existing Network Monitoring Products

Many tools exist for monitoring and visualizing the performance of a network. These existing applications display the network topology with device and performance information. Generally agents are deployed within the network to perform the monitoring. These are applications which collect device information, then communicate with a central server which brings all the data together.

A description of some commonly used applications is given below. Table 2.1 summarises the main points of the applications surveyed.

Oberservium [15] A network monitoring tool with many features. Displays network load graphs over time for each network device and collects statistics. Provides baseline usage of historical data.

Ganglia [16] Designed for monitoring clusters and grids, the load is taken as an average across the whole grid. Application and device performance is monitored using software agents installed within the network. This tool is free and open source.

Solarwinds [17] A network performance monitoring application with live readings from the network. Can show statistical performance baselines based on historical data.

Nagios [18] A popular network and device monitoring application. A user can plan infrastructure upgrades and it offers a live view of the network performance. The core is open source, but for advanced features several licensed applications exist.

HP Network Node Manager i (NNMi) [19] This is a large tool a "comprehensive network management solution". It is used for troubleshooting and monitoring network devices. Offers historical performance management.

Application	**Uses SNMP**	**Own Agent**	**Prediction**	**Price**
Oberservium	Yes	No	No	Free & Paid
Ganglia	No	Yes	No	Free
Solarwinds	Yes	Yes	No	Paid
Nagios	Yes	Yes	No	Free & Paid
HP NNMI	Yes	Yes	Historical	Paid, Proprietary

Table 2.1: Comparison of network management software

All require some installation on the server and some of these require agents to be installed on the network devices to relay information back to the server application.

2.3.1 Simple Network Management Protocol

Simple network management protocol (SNMP), is the standard system for communicating information about the devices within a network, monitoring their status and performance. Agents running on managed devices connected to the network send device status information to a central point managing entity. Information ranges from device specifications and configuration settings to performance and application information.

Alternatives to SNMP, such as Windows Management Instrumentation (WMI) exist. It performs similar functionality to SNMP, but provides Windows-specific host information. Applications which support SNMP commonly offer methods of using WMI to allow the extra features available on Windows systems to be used.

2.3.2 Compared to OpenFlow and Queueing Analysis

None of the above products incorporate the use of analytical models to predict the load and performance of specifically SDN network devices. They have a network management focus, collecting and displaying the information of all devices attached to the network, and monitoring for exceptional behaviour. For example, other behaviour includes analysing the types of traffic to see what sites are used most visited to inform caching policies.

The use of SNMP for monitoring network devices is similar to how OpenFlow uses counters to monitor switch network interface and flow statistics. However SNMP uses

UDP based messages to communicate with the managing entity across the regular network. OpenFlow uses either physical or VLAN ('in-band') links to the controller, separating it from the packet-forwarding data plane, a channel dedicated to communication with the controller.

2.4 Network Performance in the Literature

The performance of a network can be described in two ways: end-to-end performance and device performance. End-to-end performance measures the performance of the connection from one host to another through the network. Device performance looks at all the network's traffic and how the devices behave. For example, the products above all follow a device view of the network, some view the overall network delay and how it impacts end-to-end performance.

Two articles surveyed show how the performance of a network can be predicted.

Proteus[20] Short term prediction of end-to-end network performance for mobile networks carrying time sensitive media. Predictions are enabled up to 0.5s for packet loss and one-way delay. Predictions are based on regression trees.

Network Bandwidth Predictor[21] Improves on a similar, previous predictor 'Network Weather Service'. NBP uses a neural network trained on historical data, rather that statistical analysis, to predict future traffic and available bandwidth in the network in several datacenter environments.

Neither of these papers predict how a change will affect performance of network devices, or can anticipate the effect of adding new devices to the network, focusing on end-to-end performance and network-wide bandwidth. To manually predict such effects in other applications can be a long process – the results becoming available only after-the-fact.

The controller in an SDN network can be seen as a network monitoring tool. It aggregates the network information making it available to applications. The Ryu controller [13] used in this project, comes equipped with tools for viewing topology and collecting switch and link statistics.

The proposed tool would inform a network planner of the performance limits, using an established branch of mathematics, and allow them to see overall load given certain throughput. Combined with the information of other monitoring tools, such as the products detailed above, an administrator could better make policy and routing decisions, and even improve the topology of virtual network switches.

2.5 Ryu Topology Viewer

It must be mentioned that the Ryu controller includes a topology view application – see fig 2.2. Using the internal view of the network topology maintained by the controller, the topology view application can build a live, browser-based, visual representation of the network topology.

The topology viewer application uses a simple webserver, serving a webpage and opening a restful interface for accessing topology-based information about the network. Once a client connects and builds a display of the current network, remote procedure call (RPC) updates are sent from the controller whenever changes occur in the network topology, such as links or switches being added or removed.

The application architecture of this small application fulfils the architectural design requirements of this project, as defined in chapter 3, and thus was built upon. Additionally

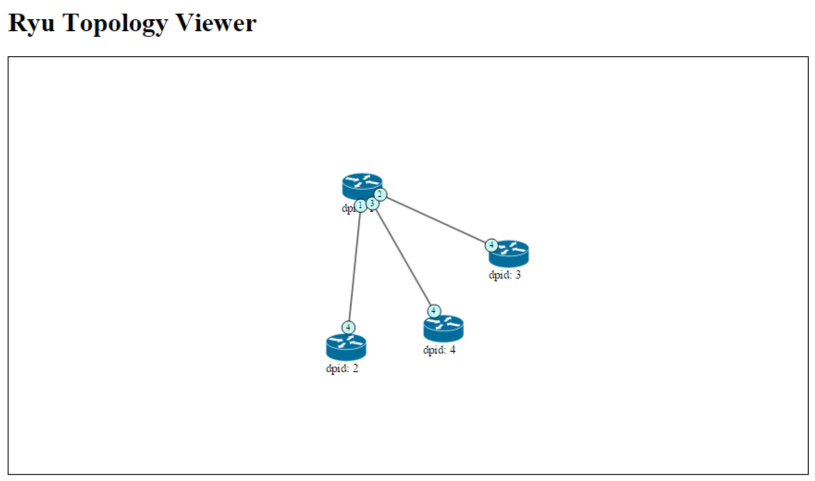


Figure 2.2: Screenshot of the default topology view in Ryu showing a four-node network

within Ryu a restful interface exists for accessing network statistics, but as explained in chapter 3 by creating a Ryu application to perform this operation automatically this allows the data to be restructured and avoids any connected clients polling the server for updates.

The client topology view is used within the performance application to display the network topology and to learn which switches are adjacent when building a spanning tree graph of the network for user adjustments, as per section 3.4.

Throughout this report the words 'switch' and 'node' is used interchangeably and refer to the same thing. In the design chapter additional nodes are described to explain the data structures used to store information on the switches, though these are mentioned sparingly.

Chapter 3

Design

This chapter describes how the application is be built. It covers the requirements, considerations of the architecture, the software structure of the application and how the queueing models will be incorporated.

3.1 Design Requirements

The requirements for building an application which improves the usability of queueing analysis in a network are:

- Giving the user a live view of the network's performance
- Include the facility to artificially alter the traffic in one node, and see the effect on the network
- Ability to artificially add switches to the network
- Allow alternative models to be used to view performance

Exactly how these will be addressed will be explained later in the chapter. An overview of the application's architecture, the foundations for meeting the requirements will first be discussed.

3.2 Architecture Considerations

A network monitoring applications such as this can be described as an application which receives, processes and displays network data. There are three main stages in the pipeline of network-to-user: 1) An acquisition stage, where information is extracted from the network and received by the application. 2) A processing stage, where the data is analysed and performance is determined. 3) Displaying the results of processing.

Specific to this application, being within an OpenFlow environment, the controller manages the first stage: collecting data from the network using the OpenFlow protocol and passing that data to the application. The application handles the final two stages, processing this extracted data and then displaying it. The application must also process user input and perform calculations using queueing models. The relationship between these three stages is shown in figure 3.1.

Between the design requirements and the application framework described, there are two architectural design considerations:

1. The relationship between the network being monitored, the application and a user.

2. Which is the most suitable OpenFlow controller?

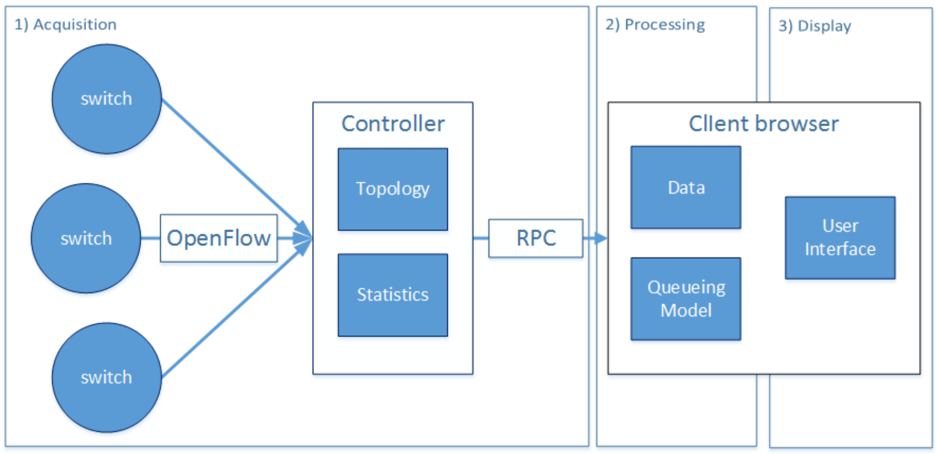


Figure 3.1: Flow of data from network to client

3.2.1 Network-Application-User Relationship

This section justifies the use of a client-server relationship and why a browser based application was used instead of a dedicated client application. This relationship is important, encapsulating the three stages of data acquisition, processing and displaying defined above.

An OpenFlow controller is generally run on commodity hardware servers rather than vendor specific. As such, the controller is run within a desktop operating system – eg. Linux or Windows. These servers can be located separate from network administrators. The resulting application architecture will need to work alongside the controller in this environment and efficiently extract data from the controller, even when accessed remotely.

Where the stage two processing is performed is a concern. The controller is a central point in the network, making the forwarding decisions for the switches comprising the the network. It can become a performance bottleneck from excessive network access, and running applications should not place unnecessary burden upon it.

To embrace remote access and reduce the controller's load, processing the data and modelled features should be done remotely. Should the modelling be done on the server, user interaction would create unnecessary load on the controller. Thus, together with the visual and interaction processing, the data processing and modelling will be done remotely, following a client-server model between the controller and the user.

Given a client-server model, there are two possible options for the client. A dedicated piece of software, handling stages two and three above and reception of data from the server, or an browser-based application for handling stages two and three. While these appear similar, basing the application within a browser takes away the need for managing the network interface between the server and the client, at the cost of potential delays from this loss of control, which could challenge the real-time requirement of a live view of the network. Table 3.1 compares these two options, when considering a Java or Python based dedicated application and a HTTP based browser application.

Accessibility is a primary motivator in the decision between dedicated and browser based, as with performance and ease of development. Building the client application as a dedicated program, using a portable language such as Java can boost the performance of the application, rather than relying on a potentially slower scripting language such as

Desired Qualities	Dedicated	Browser	Preferred
performance	high	low	dedicated
platform independence	medium-high	very-high	browser
technical competency	high	medium	browser
level of network control	high	low	dedicated
library support	high	high	equal

Table 3.1: Desired qualities of the application client

JavaScript. However Java relies on the Java Virtual Machine and all the required libraries being installed.

In terms of the processing required, queueing equations reduce to reasonably small equations for calculating performance measures, meaning the primary load will come from displaying results and will be related to the scale of the network being analysed.

Browsers also remove the barrier of platform, and due to the high use and interest in the web development domain, many simple libraries exist to reduce the development burden.

While coming out relatively even, ultimately the potential loss of control and performance in using a browser based application, is outweighed by the ease of development, reduced effort on the part of the user and an increase portability. To evaluate the limitations on performance, delay in running the application in a browser is investigated in chapter 5.

In terms of the method of moving the data from the server to the client, remote procedure calls (RPC) are used instead of HTTP requests and responses. A connection is first established, followed by regular updates of statistics information read from the network. HTTP responses would require the client to poll the server before the server responds, and place the client in control of when to send the data rather than the controller.

3.2.2 OpenFlow Controller

As with the choice of architecture, ease of development is again a desirable quality. There are a number of OpenFlow supporting SDN controllers available, however most support only the initial 1.0 release of OpenFlow. As discussed in chapter 2, the protocol has evolved over the past 6 years as features were added and protocol specifications changed. As of January 2015, a survey of SDN [2] found only two controllers available which support version 1.3: OpenDaylight and Ryu.

This project requires acquiring information about the network which is suitable for use in a queueing model. The information of interest to this project is the counters kept by the switches. Each OpenFlow switch is required to keep a measure of the packets passing through it. Packets are classified by flow, table and port, amongst a few others. Each counts the packets sent, received and dropped and how long the port or flow has been active.

This application is designed around the 1.3.0 version of OpenFlow. At this version the range of required information for a switch to measure and began to include both number of packets per port and duration. While version 1.2.0 saw the reduction of supported measurement counters, the port counters only required the number of packets to be measured and not duration. Versions beyond 1.3 also support the same features, allowing support for this application to continue into the future [22, 23].

Ryu is used in third year Network Engineering classes. It is an open source controller built in python with ongoing development. OpenDaylight is built in Java and is supported by the Linux Foundation. Both have been kept up to date with the standards as they develop since the survey above.

In terms of the application developed these controllers both support the required counter features and support a easy API to use. Familiarity with Ryu won over, and thus the application is based on this.

3.3 Application Design

This section will explain the software structure within the application, defining objects and their functionality, followed by the interaction between these components. Using the flow of data as a guide, it will begin at the server with the Ryu API and end with the results displayed on the client's user Interface . Figure 3.2 gives a complete overview of the application design.

Server objects:

Ryu API The interface to the network. It facilitates statistics calls to the network and creates an internal model of the network topology.

Performance Application Holds together the separate parts of the server application. It communicates with the Ryu API, receiving responses to statistics requests, and storing and processing the data ready for sending it to the client.

Monitor Placed within the server, this sets the interval for when statistics requests are made to the Ryu API and when to send the acquired data to connected clients.

Server This is the HTTP and WebSocket server created by the performance application for handling communication with the client. It hosts both a static server for distributing HTML and JS files to the client and also handles creation of WebSockets for updating both statistics and topology information.

Client objects:

WebSocket Establishes a link to the server for receiving statistics and topology updates, sending it to the appropriate objects within the client application.

RyuTopology An already developed object provided by Ryu. Models the network topology and renders this in the user interface.

Performance Data Records the statistics data for each switch in the SDN network. Provides other objects in the client application with access to this information.

Model Acts as the interface for the queueing models, proving performance measures given input network data. This defines what information is required from the network for the performance calculations.

Spanning Tree Used to distribute user adjustments on one switch across the network for prediction.

Edit Mode Halts live updates, and enables a user to artificially add extra switches to the network.

User Interface (UI) Displays the application to the user and accepts input for interaction. Displays the graphs of live statistics and adjustments.

Graphing Processing of data from the switches and model to a form which can be easily displayed by the UI.

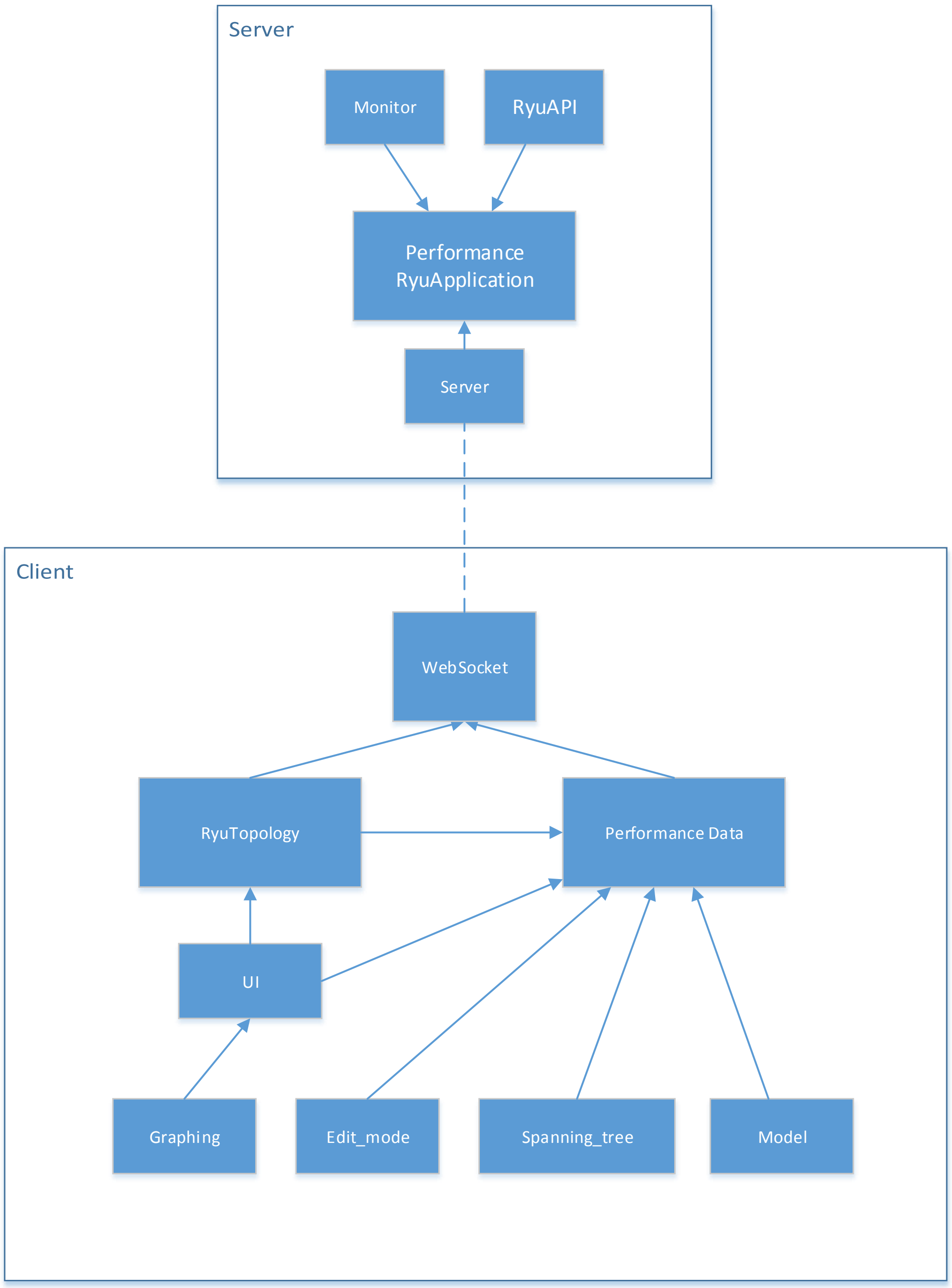


Figure 3.2: Overview of Application

Using a client-server model the complete application is split in two communicating subsystems, the client and the server. Figure 3.1 gives an overview of the flow of data through the three stages of the application. The controller, extracting the topology and statistics information from the SDN network using the OpenFlow protocol via the Ryu API. It packages this data and delivers it to the client via a webserver interface. The client then processes this data and presents it to the user.

3.3.1 Server Objects

The server Ryu performance application built accesses the SDN network's switch statistics using the Ryu API and OpenFlow Protocol. It keeps a record of all the switches connected to the controller. Within the application is a monitor object which sends out a request each second for the OpenFlow statistics information from the switches. When these return a reply, the performance application receives these, creates a object for each switch, and stores these in a dictionary object referenced by switch ID.

The port statistics are requested from each switch in the network. Specifically, this is comprised of:

1. The number of packets received since the port became active
2. The number of packets sent since the port became active
3. The duration of time the port has been active

This data is processed to obtain the number of packet since the last request and the exact change in time since the last request. It is then formed into the rate of data sent and received over the last second and included in the data saved for sending to the client.

Additionally the number of *packet_in* requests received by the controller are recorded. *Packet_in* requests are the requests the SDN switches send to the controller with headers or full packets from the network data plane to process – note *packet_in* requests do not include the statistics requests, as these operate of a different channel. Requests are recorded per switch and also sent to the client. This allows the probability of the controller being consulted by a switch (P_{nf}) to be accessed on the client, indicates how often the controller is being used and is useful for more advanced models of an OpenFlow network.

During its loop, the monitor also sends the existing switch port information out to any connected clients. It is at this point the information leaves the server application destined for the client.

The sequence diagram in figure 3.3 presents the process above visually.

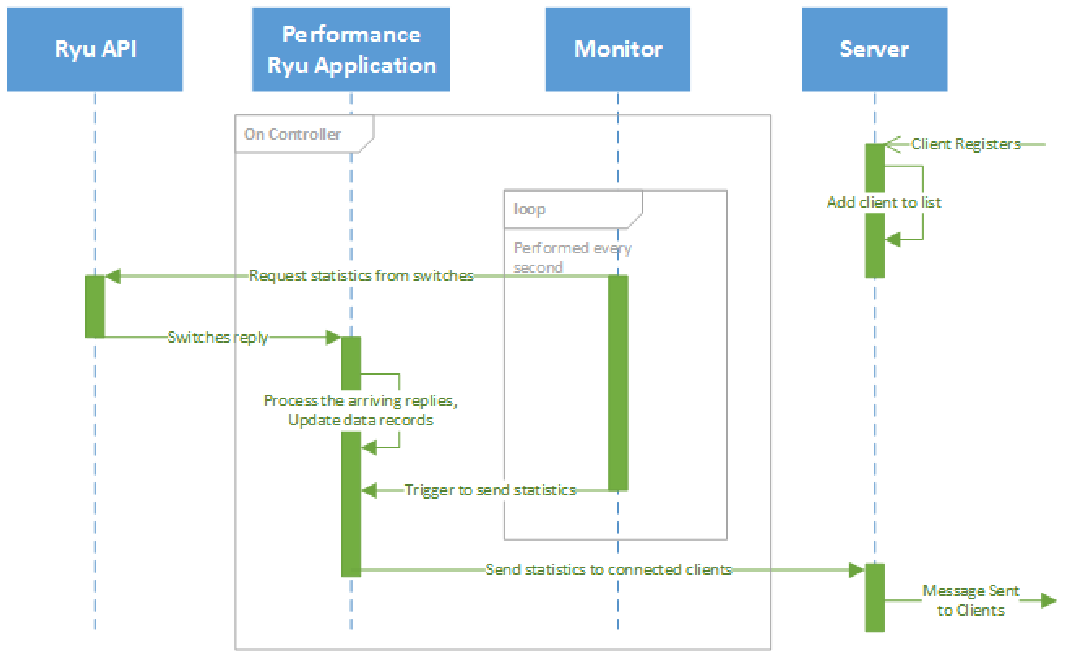
3.3.2 Client Objects

The client objects are enclosed within a web browser which connects to the server, loading the HTML and JavaScript files. The client objects are all defined in the JavaScript downloaded from the server.

Using the client web browser for displaying the visualization and interaction simplifies the setup from the users perspective, given they already have a web browser installed. It also allows the use of JavaScript and web libraries, such as D3, WebSockets and the HTTP protocol for communicating with the server.

Update data is received through the WebSockets interface. When data arrives, an RPC calls the appropriate function for handling the data sent, whether it be performance or topology data.

The performance statistics data RPC calls a function within the Performance Data object with all the new switch statistics. The switches whose data is not already stored have new



Ryu API
Performance Ryu Application
Monitor
Server
On Controller
loop
Performed every second
Client Registers
Add client to list
Request statistics from switches
Switches reply
Process the arriving replies, Update data records
Trigger to send statistics
Send statistics to connected clients
Message Sent to Clients

Figure 3.3: Sequence diagram showing an update occurring within the server system

performance node objects created – one node for the switch's data itself and one for the live data. A switch's data node carries user manipulated information such as the queueing model to apply to that node, the brand of the switch, whether the switch is an actual network switch and any adjustments made on it. The live data node is where updates from the network are stored once processed.

The updates received carry information about the traffic passing through a switch in the network, as detailed in the previous section. When read into the client it is processed to fit the data structure there. These nodes are stored in a dictionary referenced by a switch's ID. Performance results are be based on a combination of live network readings and the user adjusted value. The final structure for this data within the Performance Data object is detailed below, in the form of false JavaScript Object Notation (JSON).

```
node_data = {
  dpid:            /* Id of the switch in the data plane */
  switch_brand:    /* Name, corresponds to the service rate */
  queueing_model:  /* Functions for performance info */
  node_status:     /* either:
                      active  - default
                      removed - removed in edit mode
                      extra   - added in edit mode    */
  service_rate:    /* Determined by the switch selected */
  queue_capacity:  /* Not obtained from network, default to 0 */
  adjustments:     /* Acts on the live readings */
      {
        service_rate:
        arrival_rate:
        queue_capacity:  },
}
live_data = {
  dpid:
  pnf:
  ports: [port_no,port_no,..]
  total_tx:        /* total packets sent by switch */
  aggregate: {
    arrival_rate,
    departure_rate,},
  proportion_in: [{port_no:,proportion:}, ..],
      /* proportion of traffic entering through each port,
          determined from arrival rate */
  proportion_out: [{port_no:,proportion:}, ..],
      /* proportion of traffic leaving through each port,
          determined from departure rate */
  adjacent_nodes: [{port_no: , neighbour:{dpid:, port_no:}}, ..],
      /* neighbouring switches, determined from topology data.
          Can be a host or a switch */
}
```

The controller information, including the information required to calculate the switch P_{nf} values and the topology information also arrives via the WebSocket. The controller information is simply combined with the existing live information for each switch. The topology information is processed and transformed into an object which represents the network topology ready for display.

Following an update of the performance information the user interface object is called upon to update the display. The UI object reads the service rate and aggregate arrival rate values from the Performance Data object, then uses this information to calculate the current performance data using the model interface. Graphs are then constructed using these two sets of data and displayed alongside the topology image, as per figure 3.5.

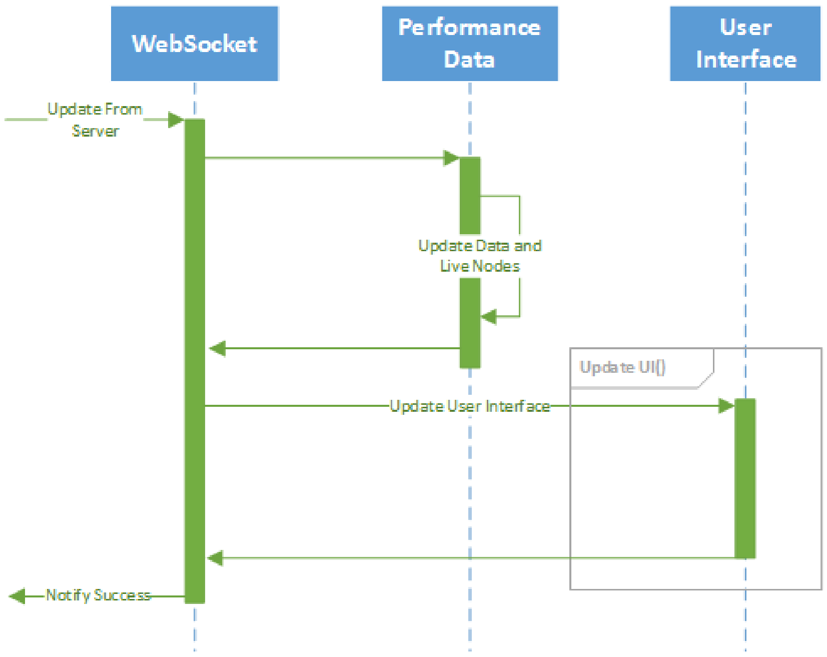


Figure 3.4: Sequence diagram showing a data update occurring within the client objects

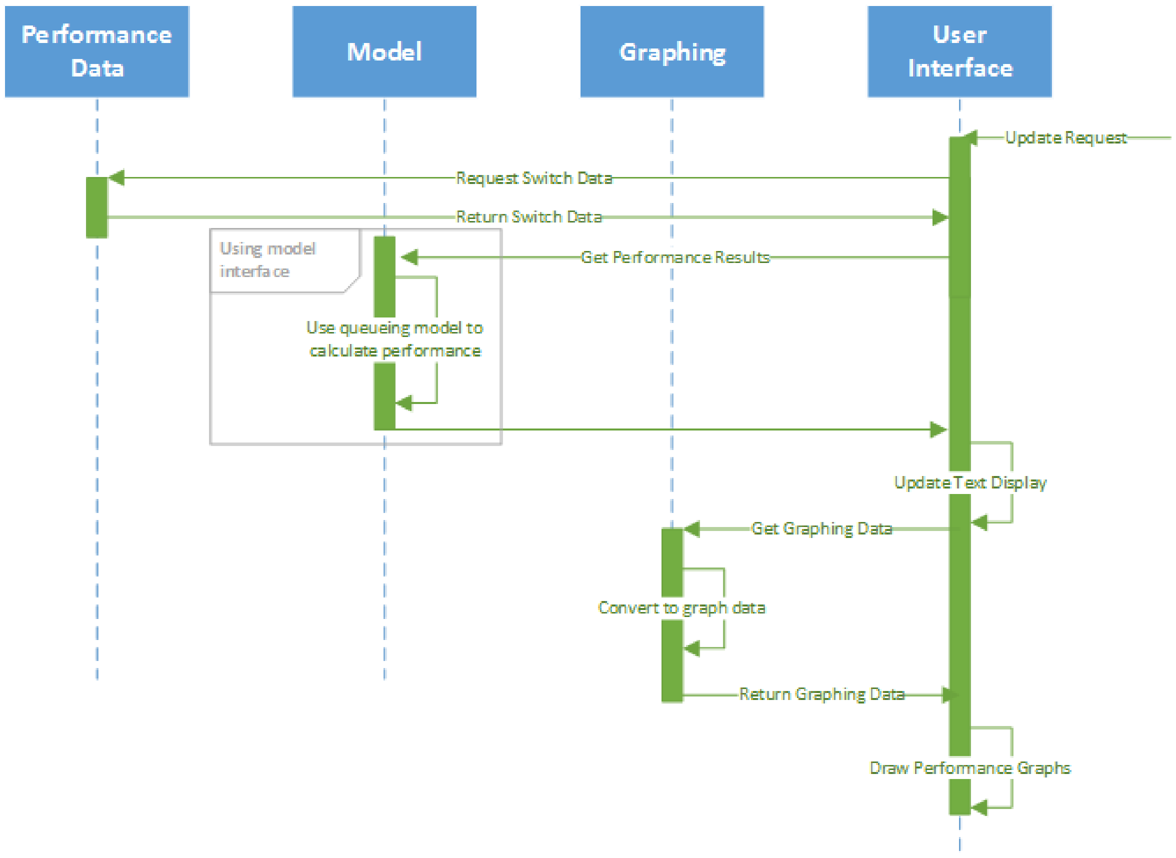


Figure 3.5: The process of updating the user interface – UpdateUI()

3.4 Queueing Models

The previous section explained how the components of this application interact, including the relationship between the performance data and the queueing model interface (See figure 3.5. The section explains how the model interface functions and expands further on the design decisions related to the data. It then explains how prediction is performed using the Spanning Tree object and how the 'Edit Mode' functions to artificially alter topologies. This section addresses the remaining requirements from section 3.1.

3.4.1 Data required

The simplest queueing models, as explained in chapter 2 require an arrival and a service rate to give results. In the case of a network, more information can be utilised, such as where packets go when they leave a node. Below is a list of information which could be useful in a network queueing model.

1. Arrival rates of traffic to each device
2. Service rates of each device
3. Capturing the probability of traffic passing to node j node from node i
4. Probability of no flow, where traffic is sent to controller (P_{nf})
5. Maximum buffer sizes in each device

Using the information extracted from the network in section 3.3.1, (1) (3) and (4) above are available. OpenFlow does not provide access to the service rate of a device, nor does it allow the buffer size of the device to be accessed.

The service rate of a device can be measured in advance while the switch is 'offline' and used in the queueing calculations. Service rate is an average not a value determined for each packet individually. Appendix A of this report describes how the service rate can be measured for both a physical and a virtual switch, the method of which is based on the process of an existing paper[5]. The size of the device buffers could be set by a network administrator within the device, so these could be entered manually. The important information – the dynamically changing values, (1), (3) and (4), are available through the OpenFlow Protocol.

The monitor object within the server determines the time between updates – how 'real' real-time is – and is set to one second. Queuing models assume long-time averages, or 'steady-state' exists within the network. For a network which is always changing however, this is not an assumption that can easily be met. One second between updates is selected as it is short enough to provide a user with live reading and long enough to provide the queueing model data to work with. Each interval will be assumed to be a steady state of the network for the live readings, despite this providing some random fluctuations in performance results.

Port Vs. Flow Statistics

Port statistics give the total flow into and out of a device. OpenFlow 1.3 also places statistics counters on the individual flow rules within, allowing statistics of flows to be requested. While this is out of the scope of this project it's important to note the impact of this within a network monitoring application.

Port statistics can give both an aggregate view of the traffic flow of a device and the probabilities of a port being used.

However, knowing with perfect knowledge where the flows are heading removes the need for any prediction, supplying perfect knowledge of where traffic will go. A user could then know exactly where traffic is coming from and where it is destined and could provision for this, instead of consulting a tool such as this.

The use of flows as a basis for the arrival rate data also poses three challenges: How to interpret the destination of traffic from each flow's rule without access to the switch logic, as not all rules provide this information; the overhead of monitoring a large number of destinations instead of just a finite number of ports; and flows are temporary, and thus the information on each is not always available – some may simply vanish.

The aim of this application is not to measure the end-to-end performance but to predicted the effect on the network devices' performance for an increase in traffic, such as that of adding a new device. However, access to flow counters could offer additional information and a monitoring tool could use flow statistics to estimate an average end-to-end performance.

Using port statistics makes use of reliable input rate of the network devices to be determined, even as the installed flows change.

3.4.2 Distributing User Adjustments Throughout the Network

Artificially altering the traffic in a single node and seeing the performance measures for the new values is straight forward. However the goal of this application is to anticipate the effect of a change on the rest of the network. How would increasing the volume of a single switch increase the other switches in the network?

Through the separation of live data and artificial adjustments, this application propagates the change throughout the network in a breadth-first manner. To traverse the network a spanning tree rooted at the user-adjusted node is created. At each neighbouring node the artificial traffic entering the node through one port is divided up among the other ports and on to any neighbouring nodes.

This process is described as a sequence diagram in figure 3.6. The algorithm for performing a cascade is defined below.

Cascading algorithm

As the artificial traffic effectively cascades from the source node, an algorithm to do this is required. Before an alteration can be made, a spanning tree must be created at the desired node. For the creation of a tree that will be of use for this algorithm, the following must be known about each node and recorded by the node:

- The node's neighbours
- Each neighbour's associated port
- The proportion of traffic that exits the node through a neighbouring node's port
- The proportion of traffic that exits via other means (ie. traffic that is destined for a connected host)

After the creation of a spanning tree alterations can then to be entered by the user, beginning at the root node. The effect must then be divided up among the ports. To accurately represent the traffic flows existing in the network, the proportion of traffic current leaving each node is used to determine how much of the adjusted traffic to pass on to the neighbouring node.

For finer control, the distribution of traffic exiting a node could be altered on a node-by-node basis, or the proportions of traffic travelling out each port. However this is not currently implemented. The algorithm succeeds in creating a spanning tree around a network with cycles (or loops) present, this may not accurately represent how a real networks traffic might behave, but given the assumption traffic will take just one path, and ususally the most efficient path, it seems a valid limitation.

An alternative algorithm, simply flooding the traffic from each port equally could also be used. This may be useful to determine how a flooding or multicast message might flow out of a node. But this does not represent the current behaviour of the network.

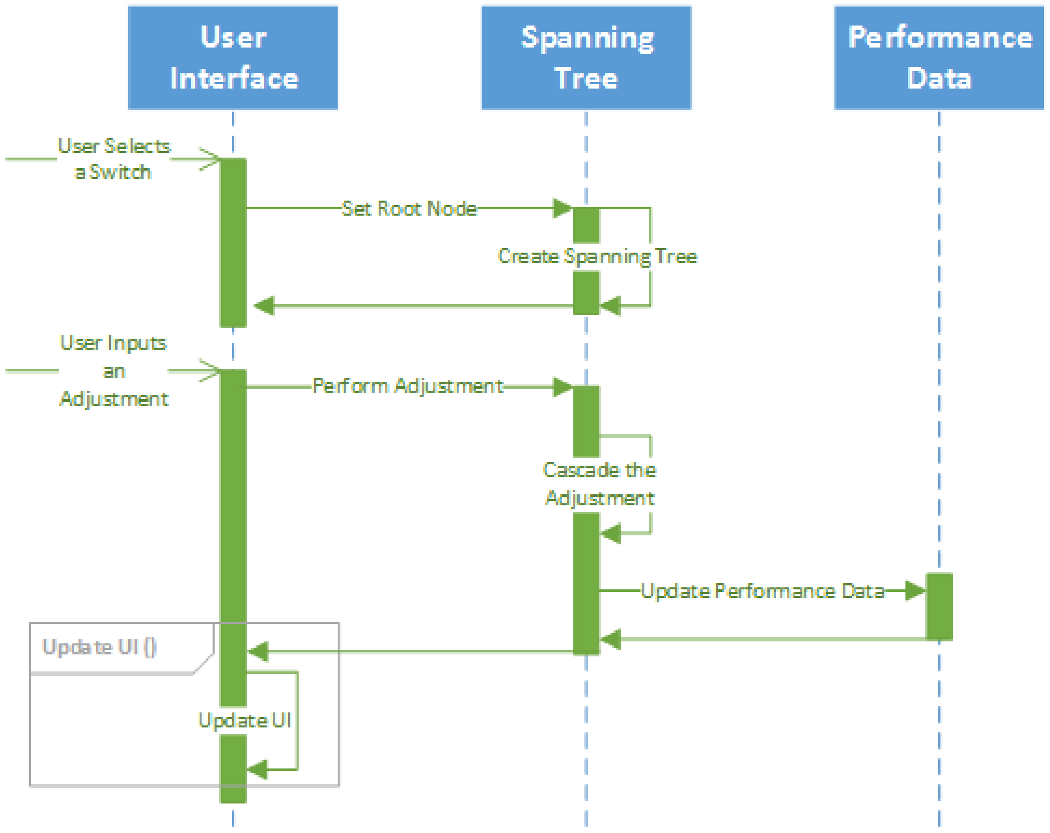


Figure 3.6: Sequence diagram showing the process of performing an adjustment on a switch

3.4.3 Adding Devices to Network Model

Enabling a user to alter the topology of a network can allow them to foresee how adding a new device with its own extra traffic could influence the rest of the network. A user is able to add new switches, define the traffic that it directs into the network. A user is also able to alter the proportions of traffic through the neighbouring switches to send traffic through an artificial modelled switch.

In order to add devices to the topology, updates from the server must be paused from performing updates. The present state must be saved, to be restored upon exiting the edit mode.

When adding a device it needs to be added to both the data model and the topology model – the Performance Data and Topology objects. A device will not initially have connections to other switches, these must be added manually. Each time a link is added the ports within the data model are updated and links in the topology model are created. The opposite is true for removing a link. The user will need to alter the proportion of traffic flowing into the additional switch from the adjacent switches, these adjustments are performed directly on the live data within the data model and not through 'adjustments'.

Traffic flowing through the node is defined purely through the adjustment mechanism, and this behaves as it would in a regular adjustment, any artificial nodes are transparent to the Spanning tree object.

When the user is satisfied they can exit the edit mode which restores the original objects in their previous state, then awaits a fresh performance update from the server to continue the live view of the network performance.

The actions of enabling edit mode and adding a switch are presented in figure 3.7 and figure 3.8. Exiting the edit mode is a much simpler process of restoring the Performance Data and Topology Data states saved and unblocking updates from the server.

3.4.4 Interface for Queueing Models

An interface for queueing models must support the implementation of alternative equations, the use of various parameters and the output of unknown data. This poses an interesting design challenge. An interface which supplies the model with all the data available, and simply handles any data it returns is used. This allows data for graphs to be created and used to view and compare the performance of network devices.

The relationship between the application and queueing models is shown in figure 3.9. An intermediary object is used to stage the model code and supplies a place for the model to declare its functionality to the rest of the application.

The user interface uses this declaration to supply the user with a choice of models on each switch. When the performance is calculated, the model interface uses the selected model on each switch to determine the code to be used.

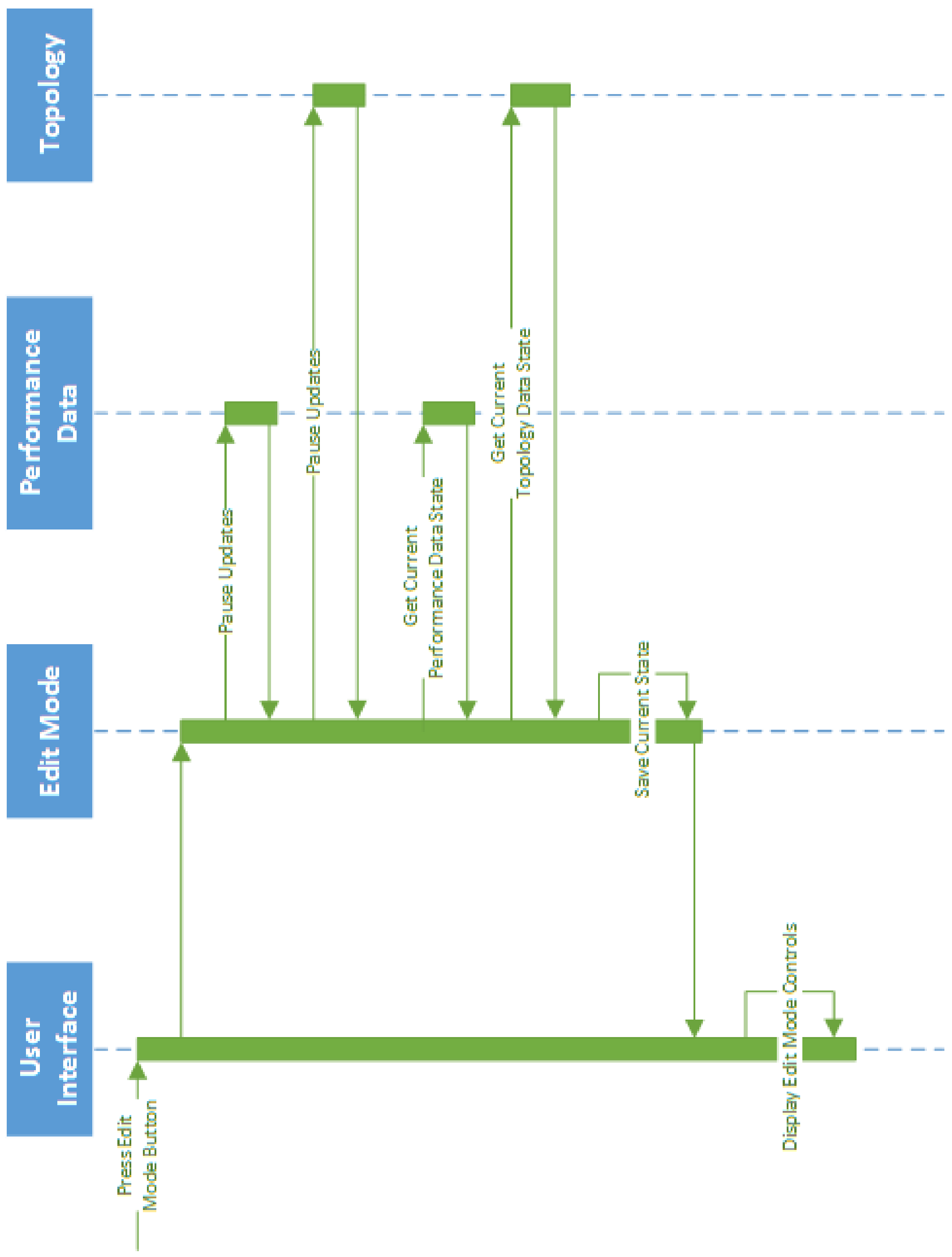


Figure 3.7: Explanation of the processes of engaging the edit mode

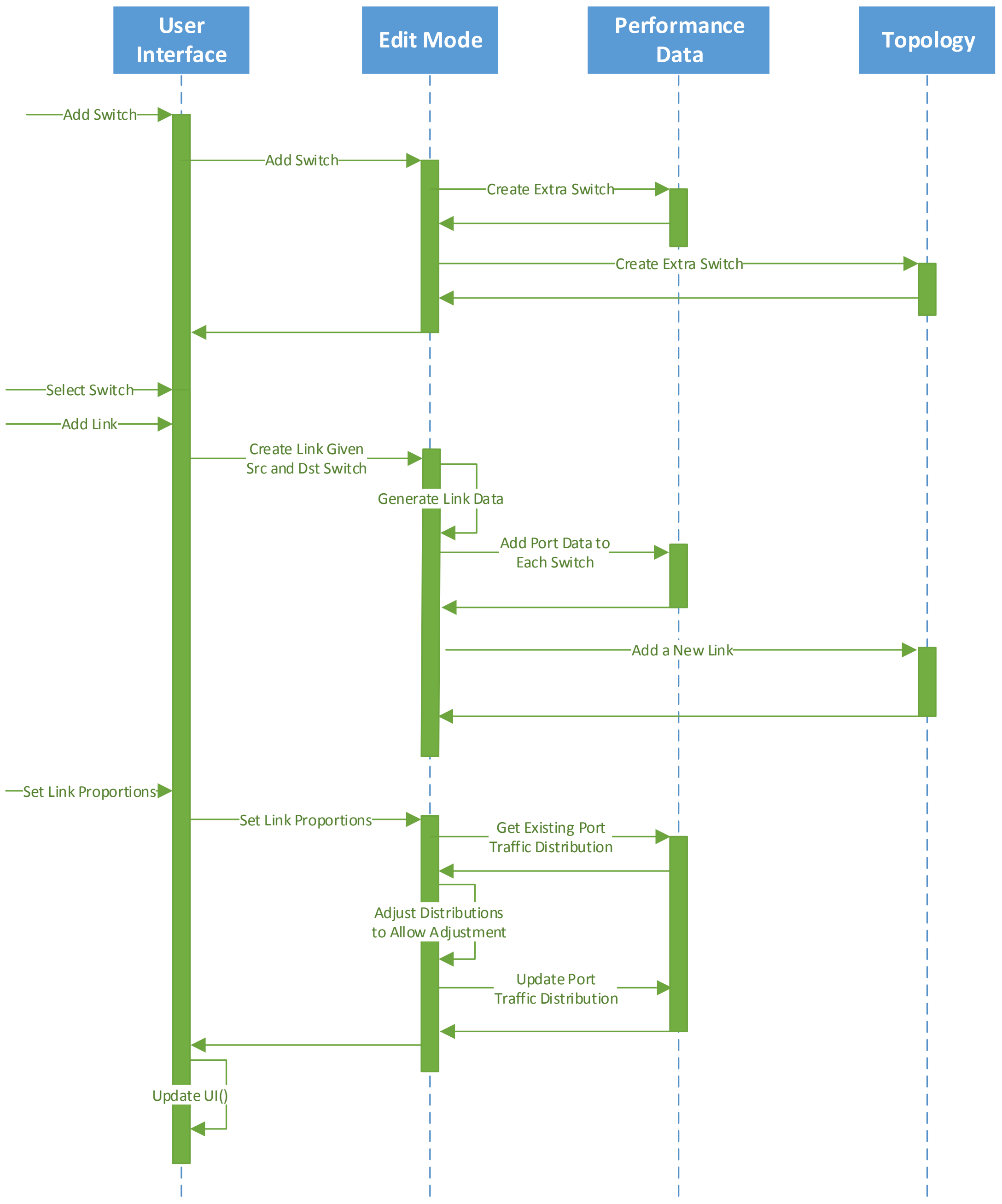


Figure 3.8: Explanation of the processes of artificially editing the network topology

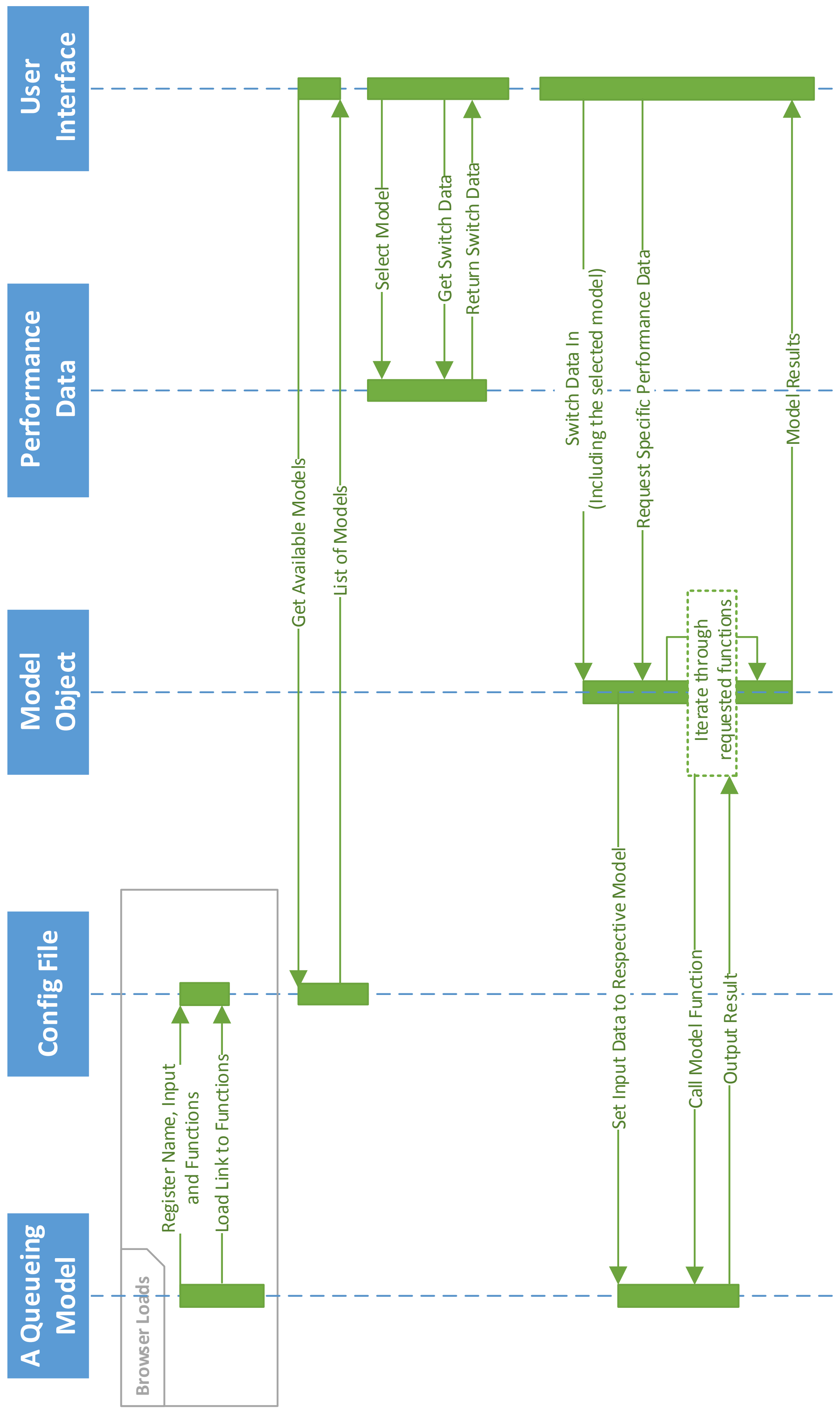


Figure 3.9: Sequence Diagram Showing How to use the Model Interface

3.5 User Interface

Using the requirements of this project as a guide, the user interface must make each of the functions available to the user. Below is a summary of how each requirement affects the user interface:

Live view of performance A view of the current traffic, the performance calculated using the models and the network topology.

Perform traffic adjustments The ability to select a switch and increase/decrease the traffic in it and see how the other switches are affected. Traffic can be altered by changing the arrival rate, editing the service rate, and changing the queue capacity.

Accessing the topology 'edit mode' A button to pause the live updates and enter this mode. Present the user with controls for adding and removing extra switches.

Select a queueing model for each switch Upon selecting a switch the user is presented with a selection of queuing models to choose.

The dynamic nature of displaying live performance, and given that simple bar graphs are not the only information the needs to be displayed – for example, the network topology and allowing a user to interact with this – means a powerful in-browser graphics library must be used. The range of browsers supporting this library must also be high.

The browser visualisations library D3 was selected for rendering the topology and the performance graphs. Other libraries exist, but the availability of online examples was a key factor in sticking to using D3. It also features an interesting way of handling data based around selections and nesting selections [24], which makes dividing data across a number of entities very simple, important when splitting the data across node, ports within nodes, and data for each port.

Looking into the future, extending the functionality or altering how the information will be presented, d3 was chosen because it's not limited specifically to charts, and is able to visualize node-graphs and information flows, and accept data dynamically.

Given the use of D3 and JavaScript a user's browser must support these features. Support for D3 is listed as available on 'modern browsers'. Support can instead be evaluated in terms of browser support for SVG, as this is the mechanism for creating the visual elements manipulated by the JavaScript functions. The website, *caniuse.com*, which lists the features supported by available browsers, covering 97.8% of all browser usage, and aquire their statistics from gs.statcounter.com. Caniuse.com ranks the current 1.1 version of SVG as having a international support of 96.17% among browsers. This information solidified the decision to use of D3 to display the performance data.

Chapter 4

Implementation

4.1 Running the Application

This section gives an overview of running the application and how the functionality designed in chapter 3 is used to perform analysis of a network. This is shown mostly through screenshots of the application running.

4.1.1 Server Application

To start the application on the server requires copying the application files into the Ryu directory and then start the Ryu controller with the Performance Application. The application files contain a script for copying the files and starting the application in a user-friendly way.

The script runs the following simple commands to copy and then start the Ryu controller:

```
$RYU=~/ryu
cp -r /performance_application $RYU/ryu/app
PYTHONPATH=$RYU $RYU/bin/ryu-manager --observe-links $RYU/ryu/app/perfvis/perfvis.py
```

This requires the user to first declare the Ryu directory in the script if it is not in their home directory. The second command runs ryu-mananger with the *–observe-links* option, instructing the switches in the network to forward LLDP packets carrying topology information to the controller.

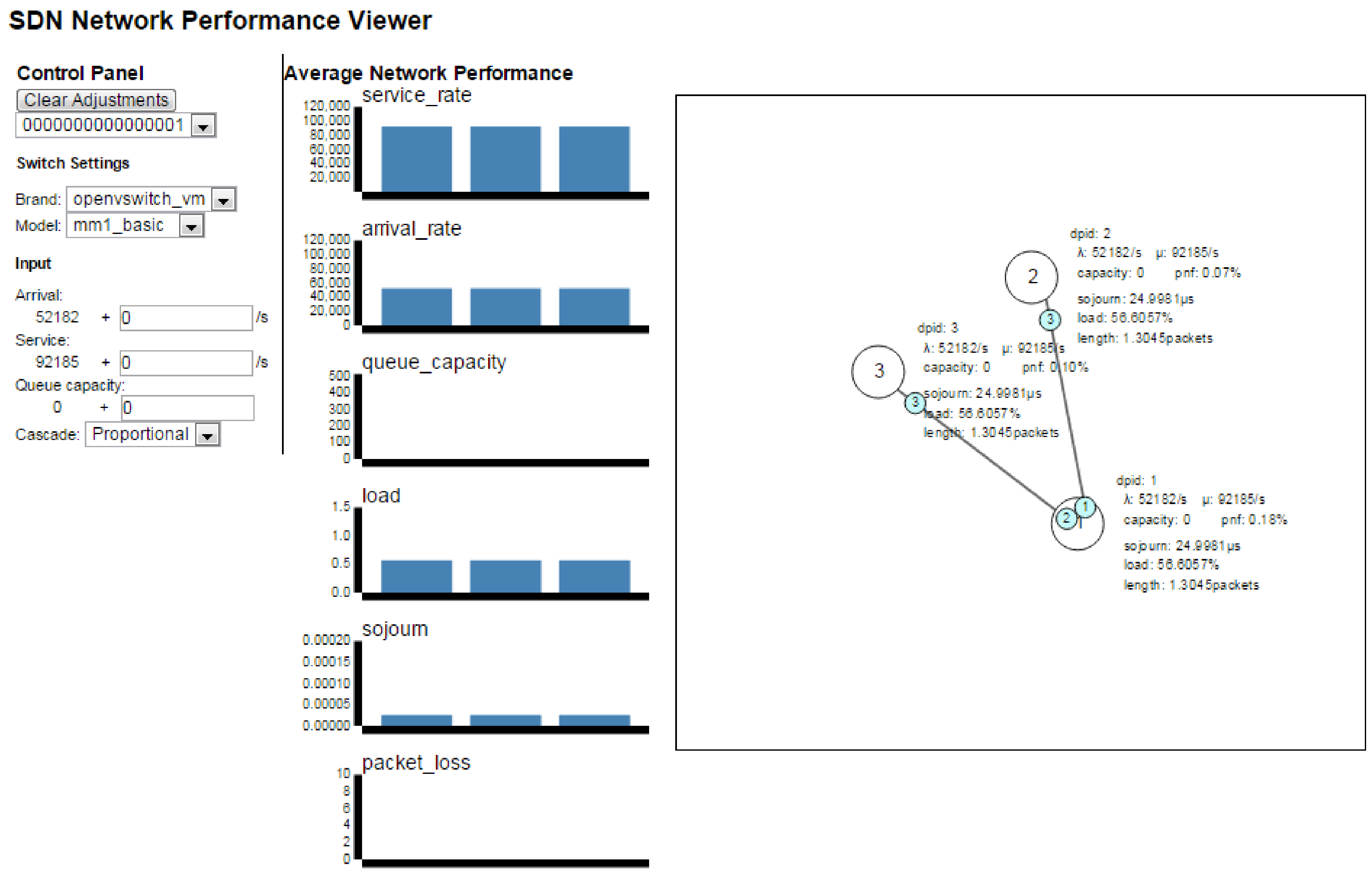
An alternative way to run the application is to include the application files in another Ryu application, using the following code within the new application (providing the Performance Application files have previously been copied):

```
  app_manager.require_app('ryu.app.performance_app')
```

4.1.2 Client Application

The web server is run on port 8080 and available at the controller's IP address, accessed through a web browser once the server has been started. When first started the application presents the user with a loading screen while it connects to the server and awaits the first update of performance data.

Following a successful update from the server the user it presented with the application's user interface, as per figure 4.1



0000000000000001

Brand: openvswitch_vm
Model: mm1_basic

Arrival: 52182 + 0 /s
Service: 92185 + 0 /s
Queue capacity: 0 + 0
Cascade: Proportional

dpid: 2
λ: 52182/s μ: 92185/s
capacity: 0 pnf: 0.07%
sojourn: 24.9981μs
load: 56.6057%
length: 1.3045packets

dpid: 3
λ: 52182/s μ: 92185/s
capacity: 0 pnf: 0.10%
sojourn: 24.9981μs
load: 56.6057%
length: 1.3045packets

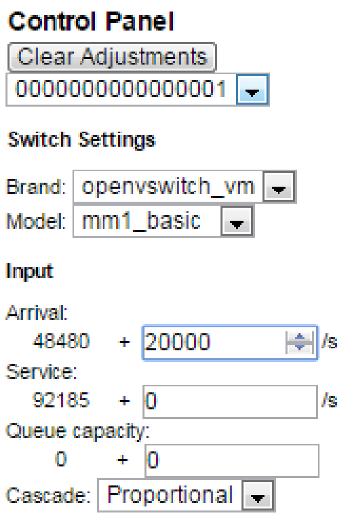
dpid: 1
λ: 52182/s μ: 92185/s
capacity: 0 pnf: 0.18%
sojourn: 24.9981μs
load: 56.6057%
length: 1.3045packets

Figure 4.1: Complete view of the client's user interface

Control Panel

The control panel presents most of the functions available to the user in one place. The user can select a node to change its input parameters to see the effect they have on the performance of the switch and the surrounding network. Each adjustable parameter is shown with its current value and a box for the user to enter any adjustments they wish to make.

The brand and model option menus allow a user to select a different switch model, determining the service rate, and a different queueing model from the available selection. The control panel is shown in closer detail in figure 4.2.


Control Panel

Clear Adjustments

0000000000000001

Switch Settings

Brand: openvswitch_vm

Model: mm1_basic

Input

Arrival:
48480 + 20000 /s

Service:
92185 + 0 /s

Queue capacity:
0 + 0

Cascade: Proportional

Figure 4.2: User control panel for performing adjustments

Performance Graphs

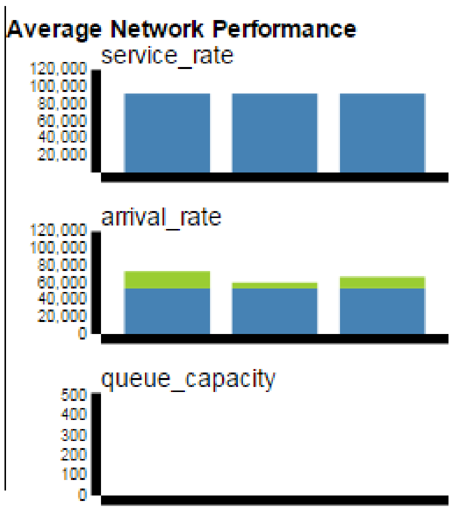
The graphs produced by the application enable a user to compare the performance across the network. Each bar of the graph represents a switch in the network, and they are arranged from left to right in ascending order of switch ID. The live data arriving from the network is indicated by a blue bar on the graph, and adjustments are shown by a change in colour. Solid green indicates an increase and a red border indicates a decrease.

These graphs are shown in figure 4.3. Graph (a) shows the input of a three switch network with an increase of traffic on the first switch, and graph (b) shows the performance information of a ten switch network.

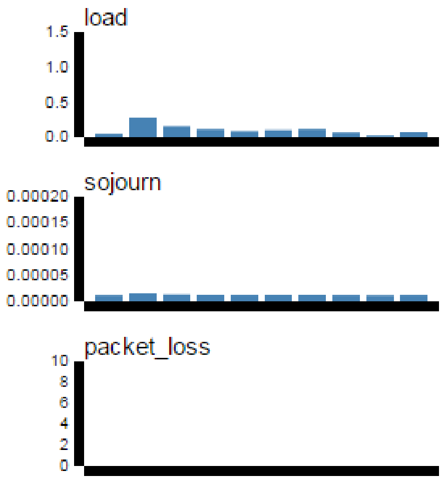
Topology View

Borrowing the foundation from the Ryu topology app, the topology view now includes the performance information available for each node. The nodes offer a view of the network layout and which ports the switches are connected to.

A three node network is shown in figure 4.4. When a user sets the arrival rate too high for a given service rate, the node turns red to indicate it is overloaded, as in figure 4.4 (b).

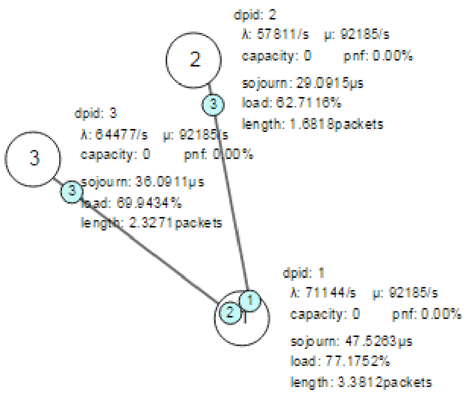


(a) Input Data

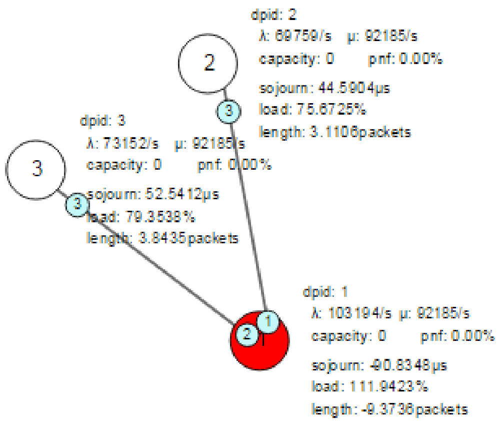


(b) Performance Dataa

Figure 4.3: Input and performance graphs displayed by the user interface



(a) Topology with Performance Data



(b) Switch’s load is too high

Figure 4.4: The application’s topology view

4.2 Queueing Models' Implementation

4.2.1 Models Currently Supported

The application implements two initial models, the M/M/1 and also M/M/1/K models. These allow basic performance measures to be calculated for either an infinite capacity node, with M/M/1 or a finite capacity node, with M/M/1/K. The use of M/M/1/K allows the possibility of packet loss to be measured when a buffer is full. When a switch has either model selected the performance measures instantly change to reflect the new model.

The user interface allows the use of queue capacity in an adjustment setting, however only the M/M/1/K model supports this, so only when a node is set to use this model will results in the packet loss performance measure be seen.

4.2.2 Model Interface Description

As described in section 3.4.4, the model interface allows different models to be incorporated into the application. The method for including additional models is defined here.

Each model must support the following operations:

- a setInput(data) function, where data contains fields lambda and mu
- at least one performance measure for output
- the model must register itself with the config object, using *config.get_model['model_name'] = model_object*
- include the outputs desired from the model in the *graphed_outputs* field array of the config object

These steps allow a model to be used and the output displayed across all the displayed switches for comparison.

4.3 Interaction

Immediately the topology view is available and the nodes can be moved around in the view to arrange them for clarity, this is a feature of the Ryu topology view and uses D3's force type graph to animate the nodes. The application also allows the user to adjust the performance parameters of each switch and is designed to allow the topology to be edited.

Adjusting Switch Performance Parameters

The user interface allows a user to select a switch and adjust a number of its parameters. Beyond the arrival rate and queueing model the user can also configure the size of modelled buffer and adjust the service rate to see how a faster or slower switch will perform.

Additionally the user can select different models or brands for a switch, defining how the performance measures behave and the base service rate of the switch.

Altering Topology Modelled

Unfortunately, while the design for this feature was completed, given the time remaining this feature was not completely implemented and cannot be included here. However, this would be accessed through a button, similar to the 'clear adjustments' button (which clears all adjustments across the model) to initiate the edit mode and pause updates.

Once edit mode is entered, the user would be presented with an additional set of controls allowing them to add a new switch to the network. And following the addition of switches, links can be formed on these switches by selecting them from a list of switches in the new controls displayed. These links allow a user to define the proportions leaving each additional port. It will also allow the neighbouring switch's departure proportions to be edited to direct traffic to the additional node.

The edit mode allows a user to explore the effect of adding new switches to the network model, but the same functionality can be explored in a less featured way using the existing traffic adjustments feature on a live network.

Chapter 5

Evaluation

This section compares the final application with the initial aims of the project, and assess how well the application performs.

Tests were included in the client application to assess various aspects, including latency in the applications behaviours, the correctness of the models implemented and how well the adjustments compare to real traffic behaviour in the network. These tests are in an additional file, test.js which is imported to the webpage. This includes code to perform the various tests.

5.1 Fulfilment of Project Goals

As stated in previous chapters, the requirements of this project are:

- Display the network performance in real-time.
- Allow the user to artificially alter the traffic of the network.
- Allow alternative models to be used to predict traffic.
- Allow the user to artificially alter the network topology.

Following from the previous chapter, the first three of these four goals has been met. The application is able to display live performance information of an SDN network. It is also enables a user to adjust the traffic in a device to see its impact on the rest of the network. An interface which allows new models to be developed is also included.

The following sections evaluate how well each of these goals has been met.

5.2 Live View of the Network

To evaluate how well the application can display network data, the application was run with various scales of network to display. This tests the limits of the application and shows how fast it can update the live display and whether it is fast enough to be considered real-time. To collect the data to measure this, the application was modified to generate and collect events throughout various stages of execution. Four areas were measured.

1. The update loop. The operations performed when the performance updates are received from the server (see figure 3.4)
2. The initial startup time. The time taken to initalise the display and data files

3. The creation of a spanning tree for user adjustments (first half of figure 3.6)

4. Performing adjustments on a switch in the network (second half of figure 3.6)

The testing system is running the Windows 7 operating system and has a Intel i5-3317U, dual core 1.7 GHz processor with 8GB of memory.

With regard to the performance the application is capable of rendering networks on the scales measured, but because of a limitation on graph axis sizes, the current UI only supports graphs up 250 switches in size. The axis was modified to allow it to increase to display all the switches for the testing below.

5.2.1 Testing Method

During development of the client webpage, a 'local' mode was created. This removed reliance on the server and sped up development and debugging. It included dummy topology and statistics data, copied from real data output by the server, with which to run on. To simulate updates from the server a loop which generates updates with randomized network statistics values was created. To run this, the webpage files are locally hosted using a simple static webserver.

This environment allows arbitrary sizes of network to be tested without implementing each network device. Because the application still receives update values it can render larger networks as if they existed on a monitored network. The topologies were generated using a python script written to create topology structures and the node data required within these. The series of network data generated uses N nodes, where N={1, 2, 5, 10, 20, 50, 100, 200, 500, 1000, 1500, 2000}. The topologies generated are of a binary tree structure. Nodes are allocated in a breadth-first manner.

To automate the process of testing the client application, the webserver was also modified to change the pre-generated network data upon each page request, cycling through each topology size and looping back to 1 node once all sizes were tested. This prevents one large file from being served to the client, instead breaking it up into only the data it needs, and avoids increasing loading times. The tests file included an object to capture the times that events were triggered within the application and a facility to save the event data. These two features allowed testing to happen automatically and indefinitely. The performance of seventeen complete loops was captured. For each network size in the loop the average is calculated, and it is these averages which are then combined to form the results below.

The local mode loop was also modified to capture various events for periods of time before reloading the page and starting the test again. The order of these events is as follows.

0-20s	Monitor only the time to perform an update
21-150s	Once a second, select a random node and create a spanning tree rooted on the that node Then perform an adjustment using 100x<the number of seconds>as the adjustment value
155s	Save the event data
160s	Reload, receiving the next topology from the server

The browsers cache was disabled during this process so each page reload requests all the files from the server.

5.2.2 Results

The results of the four periods of time measured above are presented here. A solid red line is shown on some graphs to indicate one second. If an update takes longer than one second on average it exceeds the time between updates and indicates the application has surpasses its limits in terms of network scale. Confidence intervals shown are of 95%.

Update Delay

The delay of an update was split into how long the performance data updates take and how long UI takes to process the new data. The total time was also measured to include both of these and any other processing that occurs.

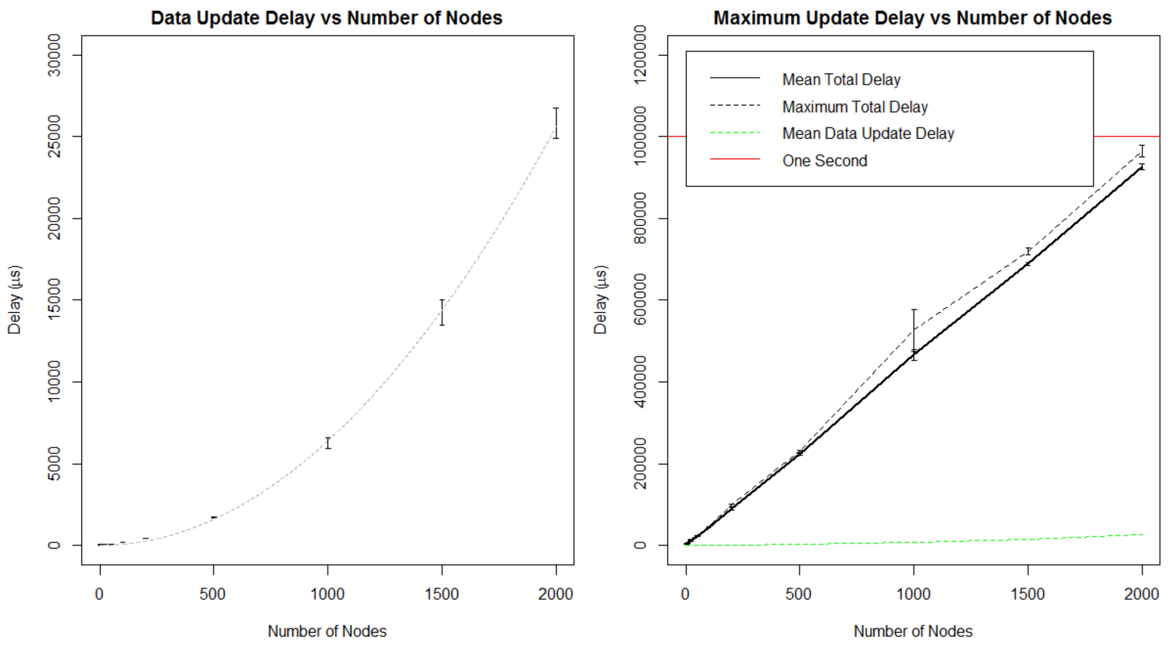


Figure 5.1: Time to perform an update

Shown in figure 5.1 there is a significant difference between the time taken for the performance data ('Data Update' on graph') to be updated and the UI to be updated. The performance data taking below 30 microseconds for all measured network sizes, and the UI reaching ten times this at only 500 switches. As indicated on the graphs, the time taken to update data increases quadratically compared to a linear total increase in total time to perform an update. While measured, the time taken to update the UI is not shown. This is because it almost matched the total update delay times, the difference being close to the data update time.

Past trials showed that the maximum time for 2000 nodes occasionally exceeded the 1 second limit, however this exceeds the confidence interval shown. It's safe to say that up to 1500 nodes the application still performs well computationally. From a user's perspective though the interface topology view becomes cluttered and difficult to use. More sensible network sizes prove more usable.

Following the curve fitting the average performance of the data calculation, this would be expected to reach 1 second only with a network consisting of 12500 switches. While practically a network of that size is unlikely to be monitored by this application and the user interface already pushes the limits at 2000 nodes, this figure of 12500 can be considered the absolute theoretical limit for the application.

The equation used to form this curve is $\frac{x}{12}^{(}12)$. It is obtained via sight and gives a value with more importance placed on the larger network sized. It serves as a guide to show the general increasing trend.

Startup Time

The initial update time measured when the page first loads is significantly longer than the following of the updates. It does not includes the time taken to download the HTML file and all the JavaScript files containing the client application's code. It begins only once the code has been executed, and includes time from the beginning of the 'development' loop used to run the application offline. It includes initialising the topology view, initialising the performance data and the first call to the updateUI() function.

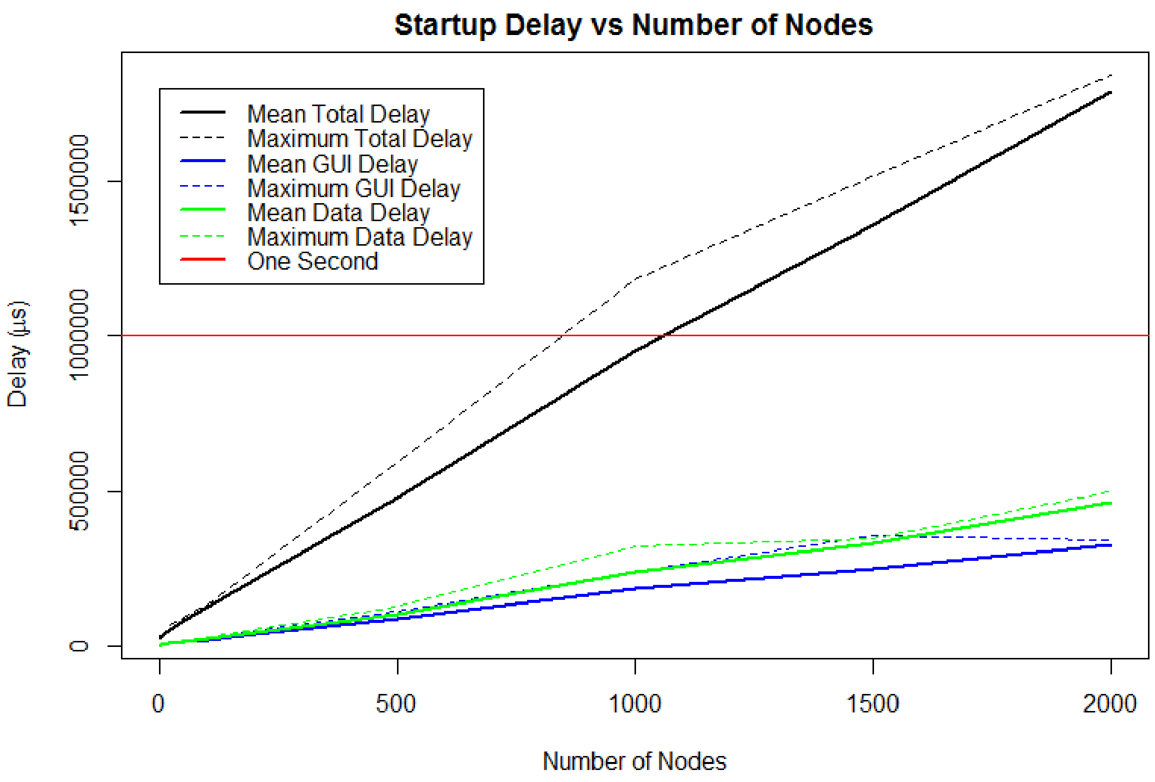


Figure 5.2: Latency of initial loading time for web page

Surprisingly the time taken to load the data is greater than the time taken to process and update the user interface. The majority of time taken is used by initialising the data structures, so perhaps this is a slow process in JavaScript. While the total time taken to update almost crosses the one second limit for an update, this is more acceptable of an initial update. To acknowledge this though, a network of 1000 nodes will be considered the upper limit.

Traffic Adjustments

A traffic adjustment is the time taken to perform a traffic adjustment, and is broken into the two stages of creating a spanning tree and inputting a value to 'cascade' through that spanning tree. As it is a key pair of algorithms in the program, fulfilling one of the project requirements, it is worth measuring and examining. It should not impact the rest of the application, and should be scalable or least as scalable to the limits seen it the previous latency measurements of 1000 nodes.

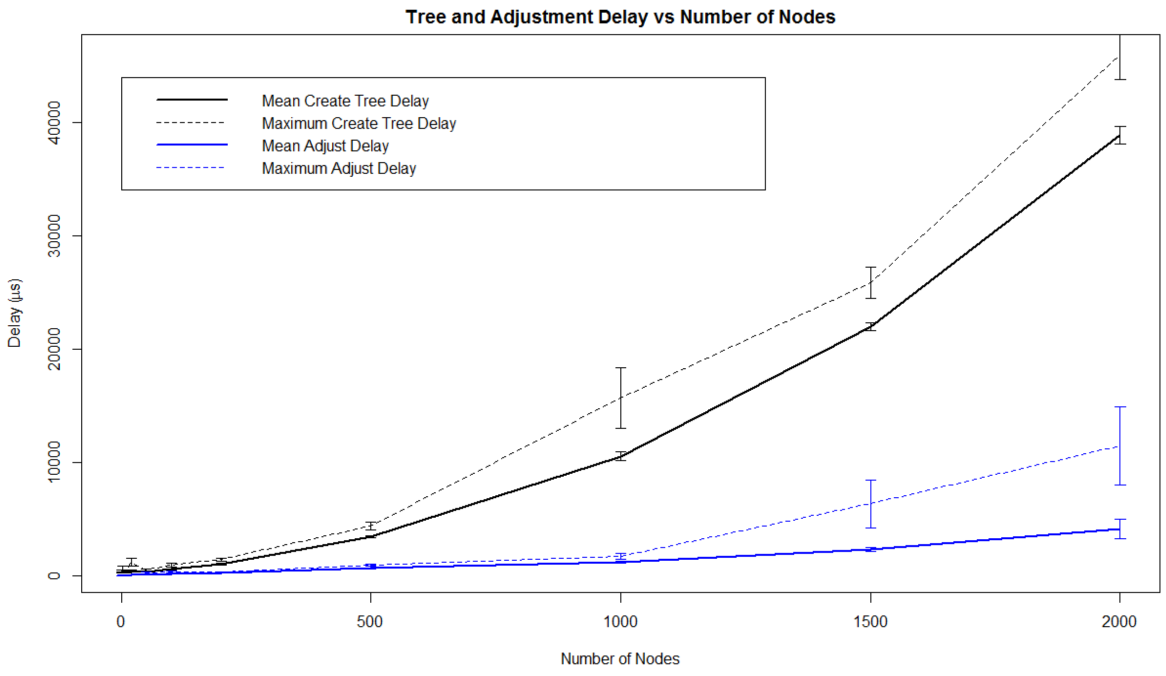


Figure 5.3: Latency of creating a spanning tree and calculating an adjustment

The most significant time is seen in creating a spanning tree, which occurs when a user selects a node in the user interface. Following the selection of a node the user can enter several adjustments, meaning the several values can be entered for one spanning tree operation. The time taken to create the spanning tree is on average eight times greater than the time to perform an adjustment using a created tree. They both appear to follow a polynomial increase as the size of the network grows.

While the delay in the creation of a spanning tree is much higher than the time to perform an adjustment on it, even with a network of 2000 nodes the time to do this is only 40 microseconds, $\frac{1}{25000}^{th}$ of a second. This value has minimal impact on the performance of the program.

5.3 Reliability of Predictions

The section evaluates the effectiveness of adjusting the traffic and the values given. There isn't an existing predictor to compare against. As the methods used in existing tools are based on past predictions this is not a viable option either. It is difficult to quantify exactly what a good prediction, given the device focus rather than an end-to-end or whole network view.

It is important to consider the three situations:

- The increase is along the path of an existing traffic or not
- The increase is not along the path of an existing flow
- The characteristics of traffic being investigated. ICMP and UDP traffic.

Adjustments are on the arrival rate, so assuming the models are correct, this is the only factor that needs to be tested.

This project is not about verifying models, as they will be decided upon by a user, and added to over time. Instead it is about a prediction in traffic volumes given a user specified adjustment. The queueing models act as a way to extract meaning from this data.

5.3.1 Protocol used

They types of traffic which could be used to perform this evaluation are TCP, ICMP and UDP. Figure 5.4 compares how TCP, ICMP and UDP appear in the network on a switch, where a flow of 100 packets per second is created using each protocol and then measured.

TCP and ICMP traffic have replies when traffic is received at the destination. These are seen as an increase in network traffic more than the amount sent by one host. TCP elicits acknowledgements form the receiving host. Depending on the implementation and assuming no packet loss this is any amount up to the total amount sent – not all packets are acknowledged but some acknowledgements may apply to a group of packets. ICMP traffic elicits a reply for each packet received. UDP traffic however, in the test environment, does not elicit replies so only the increase in sent traffic effects the network. Thus UDP is simpler method of evaluating the network and this the protocol used for measuring adjusted traffic.

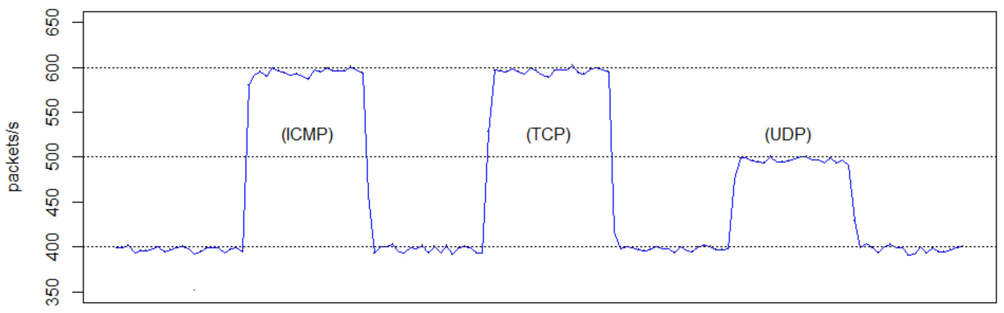


Figure 5.4: Comparison of ICMP, TCP and UDP traffic rate changes

5.3.2 Testing method

This method compares the validity of the results to simple UDP flows being introduced to a populated network.

As mentioned in appendix A, nping, the tool used for creating and sending traffic has difficultly accurately resolving sub-millisecond inter-packet delay. Because of this, all flows generated are above this threshold.

Data was captured using test.js. It automatically recorded the arrival rate value on the switches, and outputs this to file upon request.

Each of the baseline traffic flows in figure 5.5 are created with ICMP traffic flows of 100 packets per second. They are created using nping within mininet. They generate a traffic rate of 200 packets per second at each node they cross, 100 for requests and 100 for replies.

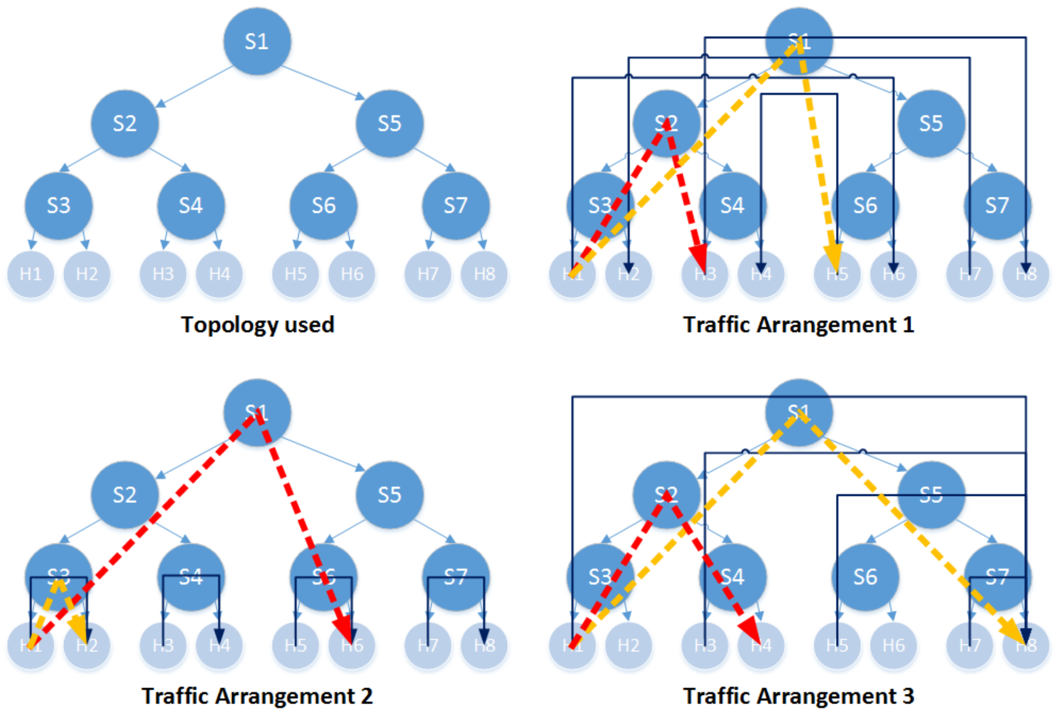


Figure 5.5: Topology used to test adjustments algorithm, in various traffic arrangements

To show how the algorithm works, and pointing out its limitations, for each network environment an artificial adjustment is made to the live network data, which is then removed. Following this two flows of traffic are added to the network, one at a time. The first is a flow which agrees with the current flows of traffic, the second is one which does not.

5.3.3 Results

The graphs below show how an agreeing flow (figure 5.6) compares to a disagreeing flow (figure 5.7). The results are obtained from arrangement 2 of figure 5.5.

The adjustment algorithm functions on the existing traffic flow in the network. Where the adjustment made is following the existing traffic behaviour it works well, and where it does not it does not work, as evident here.

The algorithm is designed to take the average flows of traffic and distribute the new traffic by those means. For gauging how a network is affected it can provide some useful information but the predictions cannot be considered precise.

5.4 A Worked Example

This section works through the calculations which this application makes easier to perform. The model used is the M/M/1 model (assumes exponentially distributed service and arrival rates, a unlimited buffer capacity and one serving unit), and a contrived example of a network with three nodes in a linear tree topology – switch one connected to switch two, and switch two connected to switch three. The equations used are those used in basic queueing theory, and are derived from a queueing system following M/M/1 characteristics.

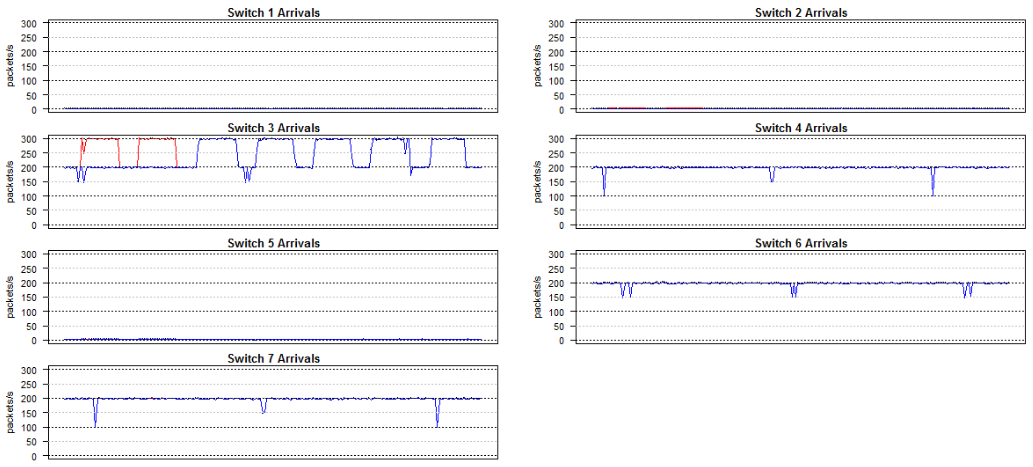


Figure 5.6: Comparison between the adjustment in the network and the creation of a flow that aggrees with the traffic existing in the network, (arrangement 2)

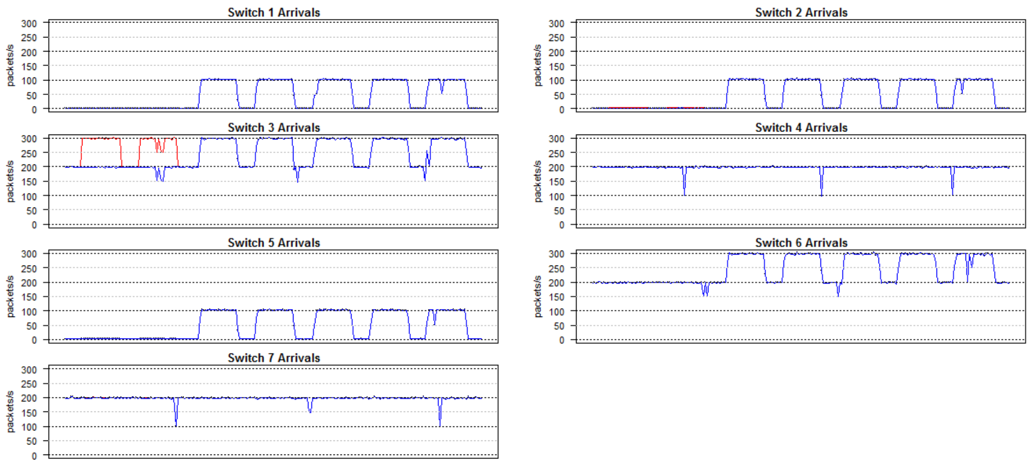


Figure 5.7: Comparison between the adjustment in the network and the creation of a flow that does not aggrees with the traffic existing in the network, (arrangement 2)

5.4.1 A single node

The live view of the performance application calculates the performance on a node-by-node basis. For a pica8 switch the service rate (μ) would be 106558 packets per second. This example is based on an average of 25000 packets arriving each second (λ).

The load on the switch (ρ) would be calculated as $\frac{\lambda}{\mu} = \frac{25000}{106558} = 0.235$. From this the average delay experienced by a packet is calculated as $\frac{1}{\mu-\lambda} = \frac{1}{106558-25000} = 0.00001226$ seconds, or 12.26 micro seconds. The average length of the buffered packet queue would be $\frac{\rho}{1-\rho} = 0.3072$ packets.

These performance measures are displayed by the application on each switch in the topology view. As this is a simple model the values are straight forward. At scale this is made more difficult needing to be performed many times over. The current value for the each network device at a given time also needs to be acquired.

An increase in the arrival rate would affect the performance as per these equations. As shown below.

5.4.2 Effect on the network

Assuming each node of the three switch network example has a stable traffic rates, how would the neighbouring switches be affected by a traffic increase in switch two, the centre switch, from connecting a fourth switch device to that switch?

The first question is how great is the change? The value of 2000 is used. The next question is, where will this traffic go? Assuming an equal distribution of traffic exists sending half to one neighbour and the remaining half to the other is a safe option. So switch one and switch three each get an additional 1000 packets every second following the increase in traffic.

Using the new traffic arrival rates of switches one, two and three the performance values can be calculated as in the single node case. Assuming the previous value for the packet arrival rate of switch two, the new load would be 0.253, an increase of 0.018, and the average delay experienced by a packet passing through switch 2 would increase by 3.1 microseconds on average.

The application developed uses the existing split of traffic between neighbours to select which neighbour to push an increase of traffic to. It greatly simplifies the analysis done above. The application allows more complicated models to be used in the future too, increasing the value of this simplification.

5.5 Model Correctness

The models implemented are the simplest forms of queueing models, using exponential distributions for inter-arrival and service times. They are long established and frequently used for simple queueing systems. The aim of this project is to bring queueing theory closer to the network administrators, it is not to propose new models or implement any complicated models. As such, only simple infinite-buffer and finite-buffer queues have been implemented.

To check their correctness, unit tests were written with pre-calculated values – checked by hand and by software. The tests covered input for valid, invalid and edge cases. The output of the code used in this application matches that of the unit tests, all 56 tests passing. This shows the models used to produce performance measures are correctly implemented.

Chapter 6

Conclusion

A software application which the gathers traffic information of an SDN network and displays it live for a user has been detailed in this report.

It provides the user with tools for examining how additional volumes of traffic could behave and the effect they have on the rest of the network.

The application makes performance measures available to a user through the use of queueing models. By way of reducing the experience needed to operate a queueing model it makes queueing theory more accessible to network administrators who can use this tool to find performance bottlenecks in their networks or see how adding traffic into their network could affect device performance.

6.1 Contributions

This application does something which is not performed by existing network monitoring applications – it applies queueing theory to a live network to extract performance information.

It provided the facility to predict performance of a network using queuing theory. This has not been attempted in the literature in this way.

These contributions combine to allow the performance of an OpenFlow network to be monitored in a useful way for network administrators.

6.2 Evaluation of Applition

The calculations used in the application require very little time to perform, allowing them to perform as real-time updates arrive.

The limits of this real-time calculation placed at monitoring a large network of 1000 switches. The theoretical limitation, based on the calculations alone and disregarding the user interface place the maximum network size to be 12500 switches.

It is now easier for a user to perform queueing analysis on a large networks with live data from the network.

6.3 Future Work

Should this project be done again with a little more time implementing both flow and port based measurements in the prediction would allow a finer granularity on the adjustment predictions. This would allow an accurate end-to-end based strategy for measuring network performance.

The tool could be extended to include the controller's performance measurement, allowing more sophisticated models and the limitations of an SDN network to be used and potentially explored.

As more switches have their service rate catalogued and more models are implemented the value of this application will increase and measurements on more varied network environments will be possible.

Bibliography

[1] Michael Jarschel, Thomas Zinner, Tobias Hofeld, Phuoc Tran-Gia, and Wolfgang Kellerer. Interfaces, attributes, and use cases: A compass for sdn. pages 210–217, 2014.

[2] D. Kreutz, F.M.V. Ramos, P. Esteves Verissimo, C. Esteve Rothenberg, S. Azodolmolky, and S. Uhlig. Software-Defined Networking: A Comprehensive Survey. *Proceedings of the IEEE*, 103(1):14–76, Jan 2015.

[3] Jiuzhen Jin, Jianmin Pang, Zheng Shan, and Rongcai Zhao. Queuing network performance model for evaluation of cmp-based voip sps. In *Computer and Information Science, 2008. ICIS 08. Seventh IEEE/ACIS International Conference on*, pages 589–594, May 2008.

[4] Yanfeng Zhu, Y. Zhang, C. Ying, and W. Lu. Queuing model based end-to-end performance evaluation for mpls virtual private networks. In *Integrated Network Management, 2009. IM '09. IFIP/IEEE International Symposium on*, pages 482–488, June 2009.

[5] Michael Jarschel, Simon Oechsner, Daniel Schlosser, Rastin Pries, Sebastian Goll, and Phuoc Tran-Gia. Modeling and performance evaluation of an openflow architecture. In *Proceedings of the 23rd International Teletraffic Congress*, ITC '11, pages 1–7. International Teletraffic Congress, 2011.

[6] Kashif Mahmood, Ameen Chilwan, Olav N. sterb, and Michael Jarschel. On the modeling of openflow-based sdns: The single node case. 2014.

[7] K. Mahmood, A. Chilwan, O. sterb, and M. Jarschel. Modelling of openflow-based software-defined networks: the multiple node case. *Networks, IET*, 4(5):278–284, 2015.

[8] Jie Hu, Chuang Lin, Xiangyang Li, and Jiwei Huang. Scalability of control planes for software defined networks:modeling and evaluation. In *IEEE/ACM International Symposium on Quality of Service*.

[9] L Yang, R Dantu, T Anderson, and R Gopal. Forwarding and Control Element Separation (ForCES) Framework, Apr 2004. RFC 3746.

[10] K Psounis. Active networks: Applications, security, safety, and architectures. *Communications Surveys, IEEE*, 2(1):2–16, First 1999.

[11] Nick Mckeown. How SDN Will Shape Networking. Open Networking Summit, Stanford, 2011.

[12] OpenFlow Switch Specification v1.0.0. `https://www.opennetworking.org/`. Accessed: 2015-05-10.

[13] Ryu SDN Framework. `http://osrg.github.io/ryu/`. Accessed: 2015-07-07.

[14] Anirudh Sivaraman, Keith Winstein, Suvinay Subramanian, and Hari Balakrishnan. No Silver Bullet: Extending SDN to the Data Plane. *ACM Hotnets*, November 2013.

[15] Observium. `http://www.observium.org/`. Accessed: 2015-07-07.

[16] Ganglia Monitoring System. `http://ganglia.sourceforge.net`. Accessed: 2015-07-07.

[17] Solarwinds network performance monitor. `http://www.solarwinds.com/network-performance-monitor.aspx`. Accessed: 2015-07-07.

[18] Nagios. `http://www.nagios.org/about/overview/`. Accessed: 2015-07-07.

[19] HP Network Node Manager i. `http://www8.hp.com/nz/en/software-solutions/network-node-manager-i-network-management-software/`. Accessed: 2015-07-07.

[20] Qiang Xu, Sanjeev Mehrotra, Z. Morley Mao, and Jin Li. PROTEUS: Network Performance Forecast for Real-Time, Interactive Mobile Applications. 2003.

[21] R. Wolski. Forecasting network performance to support dynamic scheduling using the network weather service. In *High Performance Distributed Computing, 1997. Proceedings. The Sixth IEEE International Symposium on*, pages 316–325, Aug 1997.

[22] OpenFlow Switch Specification v1.2.0. `https://www.opennetworking.org/`. Accessed: 2015-05-10.

[23] OpenFlow Switch Specification v1.3.0. `https://www.opennetworking.org/`. Accessed: 2015-05-10.

[24] How Selections Work. `http://bost.ocks.org/mike/selection/`. Accessed: 2015-07-30.

Appendix A

Appendix

A.1 Measuring Switch Service Rate

The calculations used in queueing theory require the service rate of a service node to be known. The application developed requires the service rate of a switch to be known in advance. This section discusses how this is measured in both a physical switch and a virtual environment.

A.1.1 Pica8

The Pica8 is a switch running the PicaOS SDN network operating system, which implements OpenVSwitch. Packets are generated and captured by a netFPGA. The netFPGA is a programmable network card enabling high precision packet generation at high rates. Specifically the hardware measured is a Pica8 P-3290.

The time of packets being emitted is seen at t_1 and the time they arrive as t_2. The delay introduced by the cabling in this configuration is so small it can be ignored, meaning the difference between the two read times represents the service time of the connected switch.

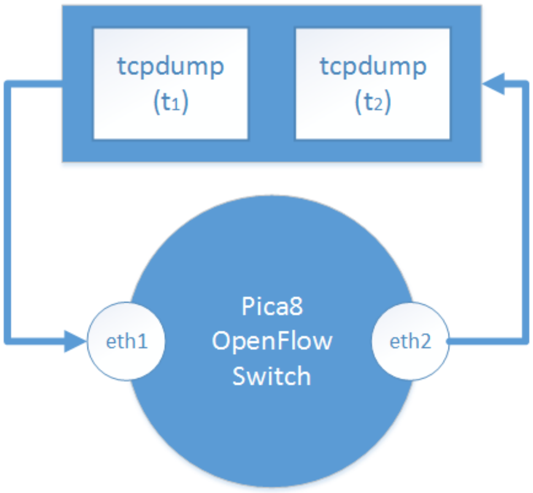


Figure A.1: View of the service rate experiment

Tcpdump is a tool for recording the packets of a network and the time the enter an interface. It is used to record the times of packets arriving and leaving the netFPGA.

The following section explains in greater depth using this tool in an alternative setup.

A.1.2 Mininet

This section describes how to measure the service rate of an OpenVSwitch switch within the Mininet virtual environment. Tcpdump is used to capture the times that packets enter and leave the virtual switch. The difference between them is then used to find the service time of the switch.

The range of payloads used is based on the values used in Jarchel's 2011 experiment[5], where 7 sizes of Ethernet frame ranging from 64 to 1514 bytes are used. Here the TCP payloads are payload={0,50,200,600,1000,1400,1500} and each is performed 10 times. The average of the ten trials is taken. These are then used to determine the mean of the service rate.

Equipment

The whole process is performed within a virtual Ubuntu 14.04 OS machine, using the following tools:

- Mininet v2.2.0
- OpenVSwitch v2.3.1
- Dpctl v1.0.0
- Tcpdump v4.7.4
- Nmap v0.6.40

The above version of Tcpdump are required for nanosecond precision measurements, controllable packet rates and the OFv1.3 protocol

Procedure

Mininet is started with a single switch, 2-host topology, running the OpenFlow 1.3 protocol. The following command is used:

```
$ mn --controller remote  --mac --topo single
     --switch ovsk,protocols=OpenFlow13
```

A diagram of this experiment is given in Fig. A.2 showing the traffic flow from host 1 to host 2 through the virtual switch, and the placement of the interface listeners.

No controller is run during the experiment to avoid additional flow rules being installed during the experiment. The flow table is instead populated manually from within mininet, using the following dpctl commands:

```
mininet> dpctl add-flows in_port=1 action=output:2 -O OpenFlow13
mininet> dpctl add-flows in_port=2 action=output:1 -O OpenFlow13
```

These rules allow traffic to flow in one direction, from host 1 to host 2.

To avoid any ARP requests (requests to attain host 2's MAC address) from being sent during the experiment host 1 has a static ARP entry entered. Again, from within Mininet:

```
mininet> h1 arp -i h1-eth0 -s 10.0.0.2 00:00:00:00:00:02
```

To avoid the switch receiving more than one packet to process at a time, traffic in the reply direction is stopped. Host 2 is set to drop all traffic from host 1 instead of replying:

```
mininet> h2 route add -host 10.0.0.1 reject
```

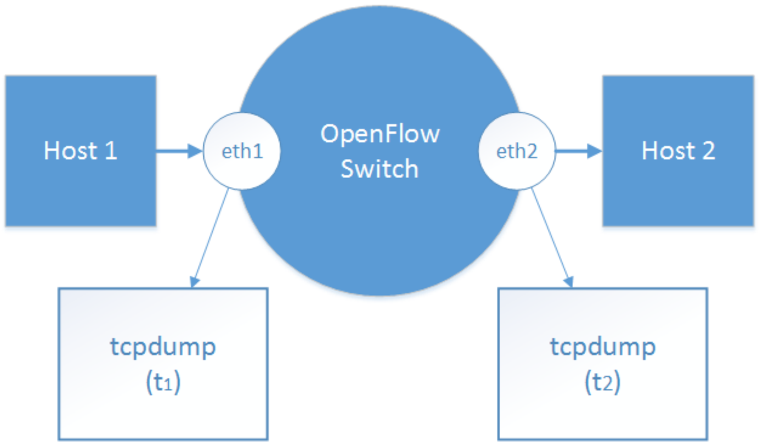


Figure A.2: View of the service rate experiment

As the Mininet switch interfaces are accessible from the host OS, the tcpdump sessions are run from Ubuntu. A separate run is made for each payload size, and for each a new tcpdump session is started on both of the data-plane interfaces of the switch. The example given is for watching s1-eth1, but a duplicate process will be started for s1-eth2:

```
$ tcpdump -i s1-eth1 --time-stamp-precision=nano >
  ~/s1-eth1_svc_100b.dump
```

The output of each session is written to a *.dump file for processing later.

Nping (a program within nmap) is then used to generate the traffic, 10000 packets at a rate of 100/s. Again, the example command given is for a 100B payload.

```
$ h1 nping h2 --tcp --data-length 100 --rate 100 --c10000 -HN
```

A Note on the Traffic Rate

While using nping, it was discovered that rates above 1000 packets per second produced inconsistent traffic volumes, resulting in bursts of traffic where packets were dropped by tcpdump. After some investigation it was found that this is a common limitation among software packet generators, due to submitting packets to the network interface where their sending is out of control for the program. Thus the slow rate of 100/s in used to guarantee safe capture of all the packet times.

A.2 Results

The results are extracted from the tcpdump output. The difference between each ordered timestamp is calculated using a simple python script, as $t_2 - t_1$. The script dumps the differences in a file which is then graphed using R. The following results were obtained. The shown error bars are placed at the 95% confidence marks and payload sizes are measured in bytes.

A.2.1 Pica8

The results of the Pica8 measurements shows a roughly linear increase in service times as the size of the packets increases.

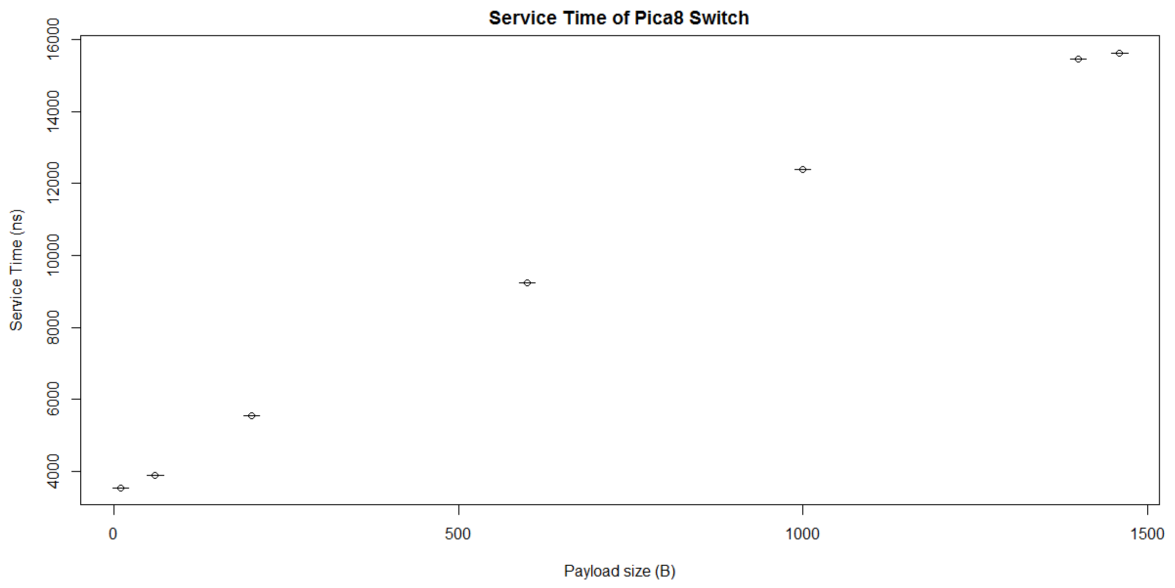


Figure A.3: Service Rate Measurement Results of Pica8

The mean of this data is an average of 9384.576 nanoseconds to process a packet. This results is an average rate of 106,557.82 packets per second through the Pica8 switches.

A.2.2 Mininet

The mean of this data is an average of 10,847.71 nanoseconds to process a packet. This results is an average rate of 92,185.35 packets per second through the Mininet OpenVSwitch switches.

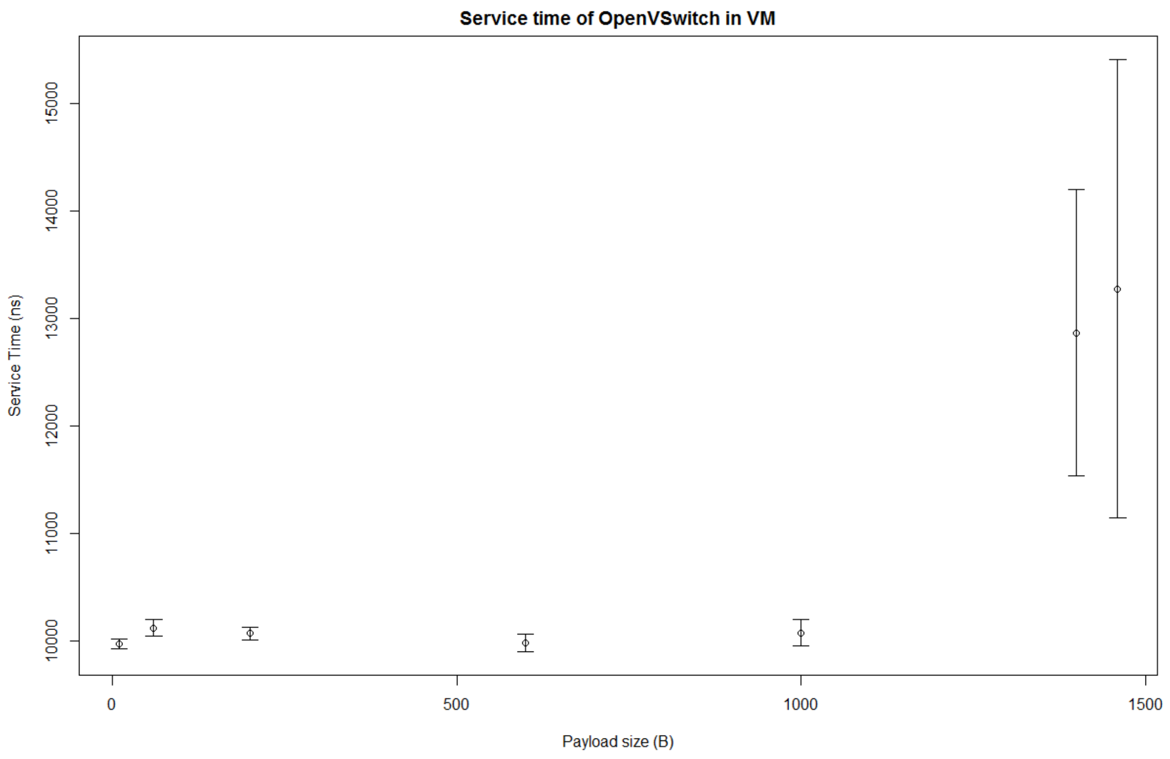


Figure A.4: Service Rate Measurement Results of Mininet OpenVSwitch